Enterprise Application Developer's Guide

VERSION 5

Borland®
JBuilder™

볼랜드 코리아 주식회사
서울특별시 강남구 삼성동 159-1 ASEM 타워 30 층
연락처 :(02)6001-3162
www.borlandkorea.co.kr

제품 구입 문의: Tel:02-6001-3194, 02-6001-3191
Fax:02-6001-3199
볼랜드 코리아
서울시 강남구 삼성동 159-1 ASEM 타워 30 층

Chapter

1

서문

*Enterprise Application용 개발자 안내서*에서는 JBuilder를 사용하여 기업용 애플리케이션을 개발하는 방법에 대해 설명합니다.

*Enterprise Application용 개발자 안내서*는 다음 세 가지 부분으로 나뉘어 있습니다.

- Part I "Team Development Using JBuilder"

  "Team Development Using JBuilder"에서는 버전 관리 및 팀 개발을 훨씬 더 쉽게 만드는 개정판 처리 기능을 설명합니다. JBuilder는 광범위하게 사용되는 버전 제어 프리웨어인 CVS(Concurrent Versions System)를 지원하고 CVS 명령 관리 및 연결 제어를 JBuilder IDE와 통합합니다.

- Part II "Enterprise JavaBeans Developer's Guide"

  "Enterprise JavaBeans Developer's Guide"에서는 JBuilder로 EJB(Enterprise JavaBeans)를 만드는 방법과 분산 시스템을 구축할 때 사용하는 방법에 대해 설명합니다. JBuilder에는 엔터프라이즈 빈의 구성, 디버깅, 테스트 및 배포를 아주 간단하게 하는 일련의 마법사 및 툴이 있습니다. Inprise Application Server 4.1이나 WebLogic Application Server 5.1에 배포할 목적으로 엔터프라이즈 빈을 만들 수 있습니다.

  EJB를 처음 개발하는 경우라면 *Enterprise JavaBeans Developer's Guide*에서 설명하고 있는 세션 빈, 엔티티 빈과 홈 및 원격 인터페이스가 무엇인지와 작동하는 방법을 참고하십시오. 엔터프라이즈 빈 클라이언트 작성, Deployment Descriptor 사용 방법 및 엔터프라이즈 빈을 사용하여 트랜잭션을 관리하는 방법 등을 배우게 됩니다.

  능숙한 EJB 개발자의 경우에도 JBuilder의 EJB 마법사와 툴을 사용하면 개발 시간을 현저하게 줄일 수 있음을 알 수 있습니다. JBuilder에는 세션 빈 및 엔티티 빈을 만드는 Enterprise JavaBean 마법사가 들어 있습니다. JBuilder의 EJB Entity Bean Modeler를 사용하면 기존의 데이

터베이스 테이블을 기반으로 엔티티 빈을 만들 수 있습니다. EJB 마법사를 사용하여 빈을 개발하면 JBuilder로 필요한 Deployment Descriptor를 작성한 다음 Deployment Descriptor 에디터를 사용하여 편집할 수 있습니다. 일단 빈을 만들었으면 EJB Test Client 마법사가 빈의 기능을 테스트하는 데 사용할 수 있는 클라이언트 애플리케이션을 빠르게 만들어 줍니다. JBuilder의 디버거는 엔터프라이즈 빈 디버깅을 완벽하게 지원합니다. 일단 빈을 배포할 준비가 되었으면 EJB Deployment 마법사를 사용하여 Inprise Application Server에 빠르고 쉽게 배포할 수 있습니다. 또한 빈을 구축하고 BEA WebLogic Server 5.1에 배포하는 옵션도 사용할 수 있습니다.

- Part III "Distributed Application Developer's Guide"

  "Distributed Application Developer's Guide"에서는 CORBA 및 RMI를 사용하여 기업용 애플리케이션을 개발하는 방법을 설명합니다. JBuilder는 분산 애플리케이션 개발을 완벽하게 지원합니다. JBuilder 개발 환경은 다계층 분산 애플리케이션을 만드는 데 필요한 파일을 많이 생성하여 분산 애플리케이션을 매우 간단하게 만들 수 있습니다. 애플리케이션을 생성하고 나면 생성된 코드에 필요한 비즈니스 로직을 추가할 수 있습니다. JBuilder의 뛰어난 개발 환경을 사용하는 동안 분산 컴퓨팅은 RAD(Rapid Application Development) 세계로 들어갑니다. *Distributed Application Developer's Guide*에서는 다음과 같은 내용을 다룹니다.

  - CORBA-Common Object Request Broker Architecture.CORBA는 분산 컴퓨팅 환경을 위한 개방형 표준 기반 솔루션입니다. CORBA의 주요 장점은 클라이언트와 서버가 모든 언어를 사용할 수 있다는 점입니다. JBuilder 개발 환경은 IDL 모듈을 사용하고 필요한 모든 인터페이스 파일, 클라이언트 스텁(stub) 및 서버 뼈대를 생성하므로 CORBA 애플리케이션을 매우 쉽게 만들 수 있습니다. 마법사를 사용하여 CORBA 서버 및 클라이언트를 생성할 수 있습니다.
  - RMI-Remote Method Invocation.RMI를 사용하면 분산된 Java 간 애플리케이션을 만들 수 있으므로 다른 호스트 상에서도 다른 Java 가상 머신이 원격 Java 객체의 메소드를 호출할 수 있습니다.

Borland 개발자 지원 문의

Borland는 다양한 지원 옵션을 제공합니다. 여기에는 인터넷의 무료 서비스가 포함되는데, 여기서 광범위한 정보 베이스를 찾고 Borland 제품을 쓰고 있는 다른 사용자를 접할 수 있습니다. 그 외에도 Borland 제품의 설치 지원에서부터 유료 컨설팅 및 광범위한 지원에 이르기까지 여러 지원 방식 중에서 선택할 수 있습니다.

Borland의 개발자 지원 서비스에 대한 자세한 내용은 웹 사이트, http://www.borland.com/devsupport/를 참조하거나 Borland Assist((800) 523-7070) 또는 영업 부서((831) 431-1064)로 문의하십시오.

지원에 대하여 문의할 때는 사용자 환경에 대한 모든 정보, 사용 중인 제품 버전, 문제에 대한 상세한 설명 등을 준비해야 합니다.

협력 업체 툴 또는 설명서에 대해서는 툴 공급 업체에 문의하십시오.

온라인 리소스

다음과 같은 온라인 소스로부터 정보를 얻을 수 있습니다.

World Wide Web	http://www.borland.com/
FTP	ftp.borland.com 특정 ftp를 통해 사용 가능한 기술 문서
Listserv	전자 뉴스레터에 등록하려면 다음 주소의 온라인 양식을 사용하십시오. http://www.borland.com/contact/listserv.html 또는 Borland의 국제적인 listserver인 경우에 다음 주소의 온라인 양식을 사용하십시오. http://www.borland.com/contact/intlist.html

World Wide Web

www.borland.com를 정기적으로 확인합니다. JBuilder Product Team은 신규 및 기존 제품에 대한 백서, 경쟁사 분석, FAQ에 대한 답변, 샘플 애플리케이션, 업데이트된 소프트웨어, 업데이트된 설명서, 정보를 게시합니다.

구체적으로 다음 URL에서 확인할 수 있습니다.

- http://www.borland.com/jbuilder/(업데이트된 소프트웨어 및 기타 파일)
- http://www.borland.com/techpubs/jbuilder/(업데이트된 설명서 및 기타 파일)
- http://community.borland.com/(개발자용 웹 기반 뉴스 잡지 포함)

Borland 뉴스그룹

JBuilder를 등록하면 JBuilder를 많이 사용하고 있는 대다수의 논의 그룹에 참여할 수 있습니다.

http://www.borland.com/newsgroups/에서 JBuilder 및 기타 Borland 제품용 사용자 지원 뉴스그룹을 찾을 수 있습니다.

유즈넷 뉴스그룹

다음 유즈넷 그룹은 Java 및 관련 프로그래밍 문제를 주로 다루는 그룹입니다.

- news:comp.lang.java.advocacy
- news:comp.lang.java.announce
- news:comp.lang.java.beans
- news:comp.lang.java.databases
- news:comp.lang.java.gui
- news:comp.lang.java.help
- news:comp.lang.java.machine
- news:comp.lang.java.programmer
- news:comp.lang.java.security
- news:comp.lang.java.softwaretools

참고 이러한 뉴스그룹은 사용자들에 의해 관리되며 공식적인 Borland 사이트는 아닙니다.

버그 보고

소프트웨어의 버그로 생각되는 것을 발견하면 http://www.borland.com/devsupport/jbuilder/의 JBuilder Developer Support 페이지에 보고하십시오. 또한 이 사이트에서 기능 요청을 제출하거나 이미 보고된 버그 목록을 볼 수 있습니다.

버그를 보고할 때, 버그가 다시 발생할 수 있는 모든 단계, 이를테면 사용자가 사용했던 모든 특별한 환경 설정과 JBuilder와 함께 사용한 다른 프로그램과 같은 내용을 포함해야 합니다. 예상한 동작과 실제 발생한 것의 차이를 구체적으로 파악해야 합니다.

JBuilder 설명서에 대한 격려, 제안 사항 또는 문제점 등에 대한 의견이 있으면 jpgpubs@borland.com으로 전자 메일을 보내십시오. 단, 설명서와 관련된 문제만 해당됩니다. 지원에 관한 문제는 개발자 지원에 알려야 합니다.

JBuilder는 개발자가 개발자를 위해 만든 것입니다. 당사는 제품을 개선하는 데 도움을 주는 사용자의 제안을 가치 있게 생각합니다.

설명서 규칙

JBuilder용 Borland 설명서에서는 특별한 텍스트를 표시하기 위해 다음 표에서 설명하는 글꼴 및 기호를 사용합니다.

Table 1.1 글꼴 및 기호 규칙

글꼴	의미
`Monospace type`	다음은 고정 폭 형식을 나타냅니다. • 화면에 보이는 것과 같은 텍스트 • "애플리케이션 마법사의 Title 필드에 `Hello World`를 입력하는 것"과 같이 입력해야 하는 모든 것 • 파일 이름 • 경로 이름 • 디렉토리 및 폴더 이름 • `SET PATH`, `CLASSPATH`와 같은 명령 • Java 코드 • `boolean`, `int`, `long`과 같은 Java 데이터 타입 • 변수, 클래스, 인터페이스, 컴포넌트, 속성, 메소드, 이벤트 등과 같은 Java 식별자 • 패키지 이름 • 인수 이름 • 필드 이름 • `void`와 `static`과 같은 Java 키워드
Bold	Bold는 자바 툴, bmj(Borland Make for Java), bcj(Borland Compiler for Java) 및 컴파일러 옵션에 사용됩니다. 예를 들면, 다음과 같습니다. **javac**, **bmj**, **-classpath**.
Italics	이탤릭체 단어는 정의되는 새로운 용어, 책 제목, 그리고 간혹 강조를 위해 사용됩니다.
Keycaps	이 글꼴은 키보드에 있는 특정 키를 나타냅니다. 예를 들면, 다음과 같습니다. "메뉴를 종료하려면 ***Esc***를 누릅니다. "
[ ]	텍스트나 구문 목록에서 대괄호는 옵션 항목을 묶습니다. 괄호를 입력하지 마십시오.
< >	텍스트 또는 구문 목록에서 각괄호는 변수 문자열을 나타내고 사용자의 코드에 적합한 문자열을 입력합니다. 각괄호를 입력하지 마십시오. 각괄호는 HTML 태그에도 사용됩니다.
…	코드 예제에서 생략 부호는 예제에서 생략된 코드를 나타냅니다. 버튼에서 생략 부호는 버튼이 선택 대화 상자에 연결되었음을 나타냅니다.

JBuilder는 다중 플랫폼에서 사용할 수 있습니다. 설명서에서 사용된 플랫폼 및 디렉토리 규칙에 대한 설명은 아래 표를 참조하십시오.

Table 1.2 플랫폼 규칙 및 디렉토리

Item	의미
Paths	설명서에서 모든 경로는 슬래시 (/) 로 나타냅니다 . Windows 플랫폼에서는 백슬래시 (₩) 를 사용합니다 .
홈 디렉토리	홈 디렉토리의 위치는 플랫폼에 따라 다양합니다 . • UNIX와 Linux의 경우 홈 디렉토리가 다를 수도 있습니다. 예를 들면, 홈 디렉토리가 `/user/[username]` 또는 `/home/[username]`일 수 있습니다. • Windows 95/98에서 홈 디렉토리는 `C:\Windows`입니다. • Windows NT에서 홈 디렉토리는 `C:\Winnt\Profiles\[username]`입니다. • Windows 2000에서 홈 디렉토리는 `C:\Documents and Settings\[username]`입니다.
`.jbuilder5` 디렉토리	JBuilder 설정이 저장되어 있는 `.jbuilder5` 디렉토리는 홈 디렉토리에 위치합니다 .
`jbproject` 디렉토리	프로젝트 , 클래스 , 소스 파일을 포함하는 `jbproject` 디렉토리는 홈 디렉토리에 위치합니다 . JBuilder 는 파일을 이 기본 경로에 저장합니다 .
스크린 샷	스크린 샷은 다양한 플랫폼에서 JBuilder 의 Metal Look & Feel 을 보여 줍니다 .

Macintosh 규칙

JBuilder는 Macintosh OS X를 매우 완벽하게 지원하도록 디자인되어 있으므로 원시 애플리케이션의 룩앤필(look and feel)을 그대로 유지합니다. Macintosh 플랫폼에는 JBuilder 자체에서 변경된 모양 및 스타일에 관한 규칙이 있으며, 거기서 JBuilder는 Mac 룩앤필을 지원합니다. 따라서 Mac에 나타나는 JBuilder의 모양과 설명서에 표시된 모양 사이에는 여러 가지 차이가 있습니다. 예를 들어, 이 설명서에서는 "디렉토리"라는 단어를 사용하지만 Mac에서는 "폴더"라는 단어를 사용합니다. Macintosh OS X 경로, 용어 및 UI 규칙에 대한 자세한 내용은 OS X 설치 시 함께 제공되는 설명서를 참조하십시오.

Part

I

Team Development Using JBuilder

Chapter

2

JBuilder를 사용한 팀 개발 (Team Development) 소개

다음은 JBuilder Enterprise 에디션의 기능입니다.

팀 개발은 효율적인 버전 제어를 사용 시의 안전한 개발을 의미합니다. 버전 제어는 효율적인 버전 추적 및 개정판을 사용하는 동안 실수로 정보가 손실되지 않도록 합니다. 버전 제어를 사용하는 가장 중요한 두 가지 이유는 다음과 같습니다.

- 버전 제어를 사용하면 여러 명의 개발자가 서로의 변경 내용을 덮어쓰지 않고 동일한 파일 집합으로 작업할 수 있습니다.
- 버전 제어는 적합하게 액세스한 사용자가 변경된 시기를 알 수 있도록 로그 및 버전 추적 정보를 제공합니다. 어떤 이유에서든지 이전 버전의 파일을 거꾸로 추적하거나 참조해야 하는 경우 버전 제어를 사용하는 것이 좋습니다.

많은 버전 제어 시스템들 또한 같은 제품에 대한 다중 개발 추적을 유지 관리할 수 있는 브랜칭(branching) 기능과 모든 개발 단계에서 전체 파일 집합의 스냅샷을 기록할 수 있는 버전 레이블링(labeling) 기능을 제공합니다.

JBuilder는 개정판 처리 기능을 히스토리 뷰에서 제공하므로 통합된 버전 제어 시스템의 유무에 관계 없이 파일 개정판을 유지 관리할 수 있습니다. JBuilder 기업용 에디션은 JBuilder에서 많은 버전 제어 작업을 수행할 수 있게 하는 CVS(Concurrent Versions System), Visual SourceSafe 및 Rational ClearCase를 인터페이스에 제공합니다. 뿐만 아니라 JBuilder의 Version Control OpenTool 및 확장 가능한 개방형 아키텍처를 사용하여 다른 버전 제어 시스템을 통합할 수도 있습니다.

이러한 버전 제어 시스템과 JBuilder의 통합은 상황에 따라 완전히 달라집니다. 현재 환경에서 액션을 수행할 수 없으면 메뉴에서 사용할 수 없습니다. IDE는 메시지 대화 상자, 마법사 및 유용한 통지에 많으므로 JBuilder가 작동하고 있을 때 알리거나 IDE가 필요할 때 많은 도움을 줍니다.

지원되는 모든 버전 제어 시스템에서 프로젝트를 수행하려면 비어 있는 프로젝트를 열고 File|New를 선택한 다음 Team 탭을 선택합니다. 그런 다음 프로젝트를 수행할 버전 제어 시스템을 선택하고 OK를 클릭하거나 Enter를 누릅니다. 이렇게 하면 연결을 구성하고 버전 제어 상태에 있는 프로젝트에서 작업을 시작하는 데 필요한 모든 사항을 설정하는 마법사가 나타납니다.

버전 제어 시스템 사용

VCS(Version Control System)는 개발자가 개정을 계속할 수 있게 하면서 파일 개정판의 전체 기록을 저장하는 방법을 제공합니다. 총칭하여 이러한 저장소 영역을 *리포지토리(repository)*, 작업하는 영역을 *작업 영역(workspace)*이라고 할 수 있습니다. 서로 다른 VCS는 다른 구조를 가지고 있으며 VCS마다 다른 용어를 사용하지만 일반적인 개념 정보를 위해 "리포지토리"와 "작업 영역"이라는 용어는 여기서 정의된 대로 사용합니다.

일반적으로 리포지토리는 각 파일의 현재 마스터 버전을 저장하고 각 파일에서 변경된 모든 내용에 대한 기록을 유지 관리합니다. 작업 영역에는 각 사용자가 가장 최근에 업데이트하고 수정한 파일 버전들이 있습니다. 어떤 VCS에서는 한 번에 한 사용자만 파일을 사용할 수 있고(pessimistic 또는 locking 모델), 다른 VCS에서는 여러 사용자가 동시에 같은 파일을 사용할 수 있으며(optimistic 또는 concurrent development 모델), 또 다른 VCS에는 이 두 가지 방법이 혼합되어 있습니다.

작업 영역에서 리포지토리로 개정판을 포스트하면 버전 제어 시스템은 마스터 복사본의 변경된 부분을 저장합니다. 파일의 나머지 부분은 변경되지 않습니다. 개정된 부분은 그 부분의 원래 위치 및 변경된 시기에 대한 정보와 함께 리포지토리에 저장됩니다.

버전 제어의 이점을 얻으려면 사용자는 다음을 수행해야 합니다.

- 리포지토리에서 파일이나 파일 그룹을 가져와서 개별 작업 영역에 놓습니다.
- 파일의 로컬 버전을 리포지토리 버전과 일치시킵니다.
- 파일 작업을 완료하면 개정판을 다시 리포지토리에 갖다 놓습니다.

개발자가 이 프로세스를 사용하면 해당 리포지토리에 액세스한 모든 개발자는 모든 변경 내용을 이용할 수 있습니다.

Chapter

3

JBuilder의 CVS

JBuilder 기업용 에디션은 널리 사용되는 Open Source 버전 제어 시스템인 CVS와 매끄럽게 통합됩니다. JBuilder에는 AppBrowser 내에서 일반 CVS 명령에 문맥에 따른 액세스를 제공하는 응답하는 인터페이스가 포함되어 있습니다.

지원되는 Windows 및 Solaris 플랫폼에서는 JBuilder를 설치할 때 CVS가 `jbuilder/bin` 디렉토리에 자동적으로 설치됩니다.

Linux에서 JBuilder는 CVS의 호환 버전이 설치되어 있는지 검색합니다. 설치되어 있으면 JBuilder는 해당 버전으로의 심볼릭 링크를 생성합니다. 설치되어 있지 않으면 JBuilder는 `jbuilder/bin` 디렉토리 안에 CVS를 설치합니다.

JBuilder는 작업할 프로젝트를 필요로 합니다. 프로젝트를 CVS에 체크인하려는 경우 CVS에 액세스하는 해당 프로젝트를 엽니다. CVS에서 프로젝트를 체크아웃하려는 경우 비어 있는 프로젝트를 엽니다. 채워진 프로젝트를 CVS에서 체크아웃했으면 CVS에 액세스하는 데 사용했던 비어 있는 프로젝트를 삭제할 수 있습니다.

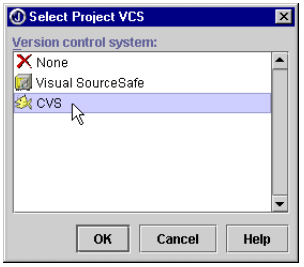
JBuilder IDE에서는 모듈 생성, 프로젝트 체크아웃, 파일 추가 또는 삭제, 파일 또는 프로젝트의 CVS 상태 확인, JBuilder 인터페이스로부터의 변경 사항 커밋, 워치(watch) 및 CVS 편집 사용, 버전 레이블 생성 및 분기 액세스 등의 이러한 모든 작업을 수행할 수 있습니다. 이러한 작업이나 기타 CVS 용어의 의미를 잘 모르겠으면 3- 23페이지의 "CVS 용어집"을 참조하십시오.

CVS는 텍스트 기반 파일을 처리하도록 디자인되었습니다. 이 툴은 다른 모든 파일을 바이너리로 간주합니다. CVS의 바이너리 파일 처리 방법에 대한 자세한 내용은 3- 21페이지의 "CVS에서의 바이너리 파일 처리"를 참조하십시오.

CVS를 버전 제어 시스템으로 선택

다음과 같은 방법으로 열려 있는 프로젝트에서 CVS를 버전 제어 시스템으로 선택합니다.

1 Team|Select Project CVS를 선택합니다. Select Project CVS 대화 상자가 표시됩니다.



2 대화 상자에서 CVS를 선택합니다.

3 OK를 클릭하거나 *Enter*를 눌러 대화 상자를 닫습니다.

그러면 IDE 화면으로 돌아갑니다.

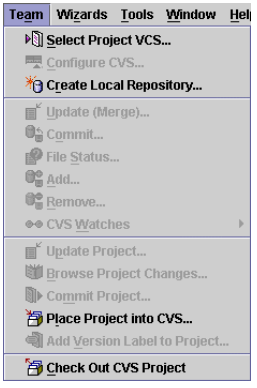
CVS 연결 구성

연결 구성은 Team|Place Project Into CVS 또는 Team|Check Out CVS Project를 사용할 때 자동으로 요청됩니다. JBuilder에서 CVS 연결을 구성하면 JBuilder에서 레포지토리에 액세스할 수 있습니다.

Team|Configure CVS를 선택하여 현재 프로젝트의 CVS 구성을 표시합니다. 연결이 사용되었으면 이 화면은 읽기 전용이 됩니다.

CVS 모듈 액세스

CVS를 현재 프로젝트의 버전 제어 시스템으로 선택했으면 다음 번에 Team 메뉴를 열 때 다음과 같이 표시됩니다.

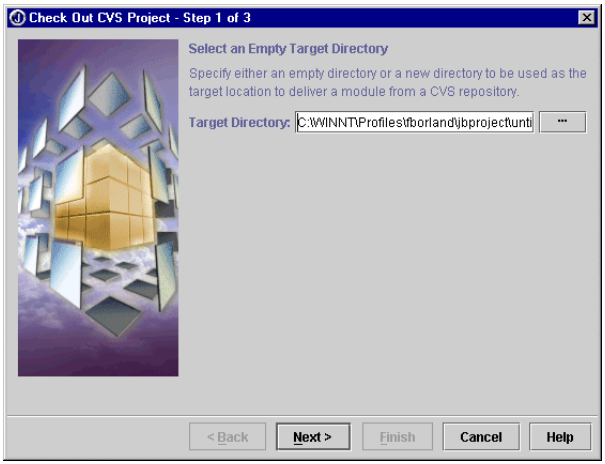


이미 Select Project CVS를 선택했습니다. Create Local Repository를 사용하면 사용자 자신의 시스템에 CVS repository를 생성할 수 있습니다. 나머지 두 개의 명령, 즉 Check Out CVS Project와 Place Project Into CVS 명령은 레포지토리에서 기존 프로젝트를 열고 각각 열려 있는 프로젝트를 포함할 새 CVS 모듈을 생성합니다.

Check Out CVS Project

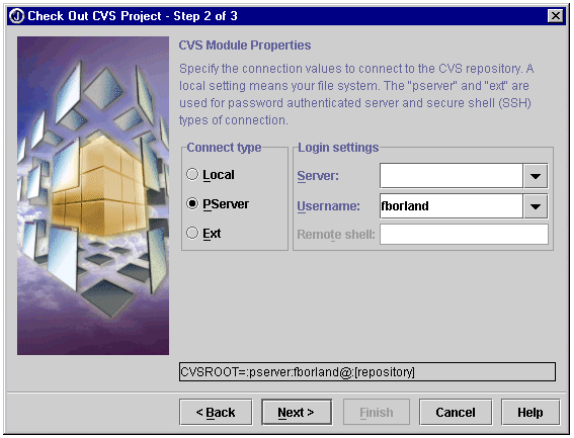
JBuilder에서는 프로젝트 내에서 버전 제어 시스템에 액세스해야 합니다. 따라서 다음과 같은 방법으로 CVS에서 기존 모듈을 체크아웃합니다.

1. File|New Project를 선택한 후 New Project 마법사의 처음 단계에서 Finish를 클릭하여 비어 있는 프로젝트를 생성합니다.
2. Team|Check Out CVS Project를 선택합니다. 그러면 Check Out CVS Project 마법사가 나타납니다.



3. JBuilder는 기본적으로 비어 있는 디렉토리를 생성합니다. 디렉토리가 존재하지 않거나 완전히 비어 있는 경우 기본 디렉토리 이름을 체크아웃하려는 모듈의 이름과 일치하도록 바꿀 수 있습니다.

4 Next를 클릭하여 2 단계로 넘어갑니다.

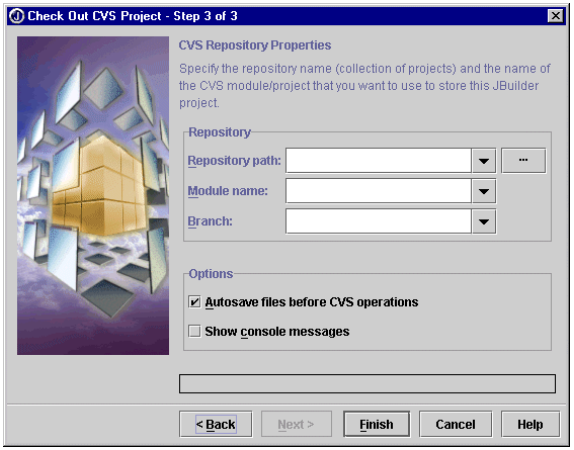


Connection Type란에 있는 Local은 로컬 레포지토리에 연결되고, PServer는 암호가 보호되는 레포지토리에 연결되며, Ext는 보안 서버 상에 있는 레포지토리에 연결됩니다.

CVSROOT 경로는 마법사의 하단부에 표시됩니다. 이 경로는 이 단계에서 입력된 정보를 반영합니다. 시스템의 사용자 환경 변수를 확인하거나 정확한 CVSROOT 변수가 무엇인지 알아보려면 CVS 관리자에게 문의하십시오.

참고 JBuilder는 선택할 수 있는 이전 연결 목록을 유지합니다. 마법사를 처음 사용할 경우에는 이 목록들이 비어 있지만 차후에는 입력하지 않고 드롭다운 목록에서 선택할 수 있습니다.

5 항목을 작성한 후 Next를 클릭하여 3 단계로 넘어 갑니다.

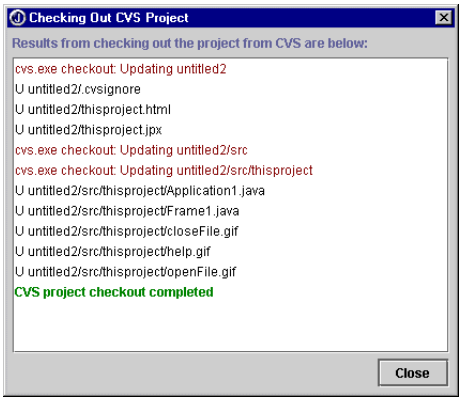


마법사의 하단에 CVS Helper 표시가 나타나는 경우 이 표시는 CVS 관리자가 서버 상에 구현할 수 있는 관리 툴을 나타냅니다. 이 툴은 CVS 통합 사용에 전혀 영향을 미치지 않으며 IDE에서 표시되지 않습니다.

6 레포지토리 경로를 입력하거나 선택합니다. 로컬 레포지토리를 사용하려는 경우 생략 버튼을 클릭하여 해당 레포지토리를 찾습니다.

7 모듈 이름을 입력하거나 선택합니다. 메인 분기가 아닌 분기에서 작업하려면 해당 분기를 입력하거나 드롭다운 목록에 열거된 사용 가능한 분기 중에서 선택합니다.

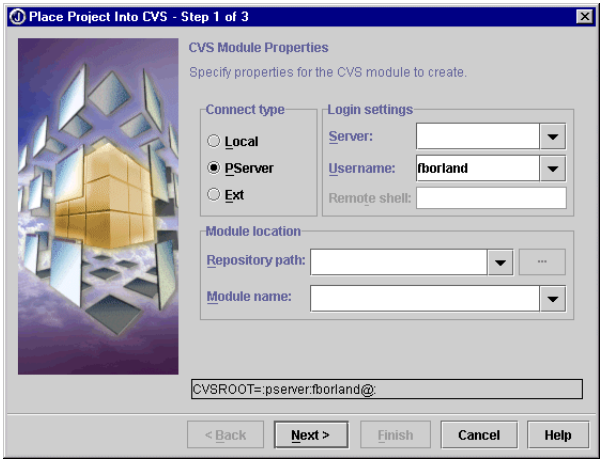
8 마법사를 완료하려면 Finish를 클릭합니다. Checking Out CVS Project 대화 상자에서 체크아웃 프로세스에 대한 보고를 표시합니다.



Project를 CVS에 두기

열려 있는 프로젝트에서 다음과 같이 합니다.

1 Team|Place Project Into CVS를 선택하여 Place Project Into CVS 마법사를 표시합니다. 이 마법사의 처음 단계에서는 레포지토리에 연결을 구성합니다.



Connection Type란에 있는 Local은 로컬 레포지토리에 연결되고, PServer는 암호가 보호되는 레포지토리에 연결되며, Ext는 보안 서버 상에 있는 레포지토리에 연결됩니다.

2 Login Settings에서 서버 이름을 입력합니다. 기본적으로 시스템에서 사용하는 사용자 이름이 JBuilder에 의해 자동으로 입력됩니다. 사용자 이름이 서버 상의 이름과 다를 경우 일치하도록 변경해야 합니다.

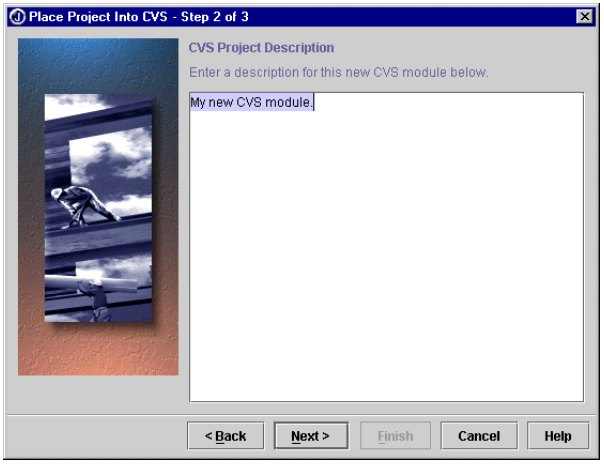
3 Module Location에서 새 모듈을 저장할 레포지토리의 경로를 입력합니다. 로컬 레포지토리를 사용 중이면 생략 버튼을 클릭하여 해당 레포지토리를 찾을 수도 있습니다. 새 모듈에 사용할 이름을 입력합니다.

참고 JBuilder는 선택할 수 있는 이전 연결 목록을 유지합니다. 마법사를 처음 사용할 경우에는 이 목록들이 비어 있지만 차후에는 입력하지 않고 드롭다운 목록에서 선택할 수 있습니다.

팁 CVSROOT 경로는 마법사의 하단부에 표시됩니다. 이 경로는 이 단계에서 입력된 정보를 반영합니다. 시스템의 사용자 환경 변수를 확인하거나 CVS 관리자에게 정확한 CVSROOT 변수를 문의한 다음 그 변수를 입력합니다.

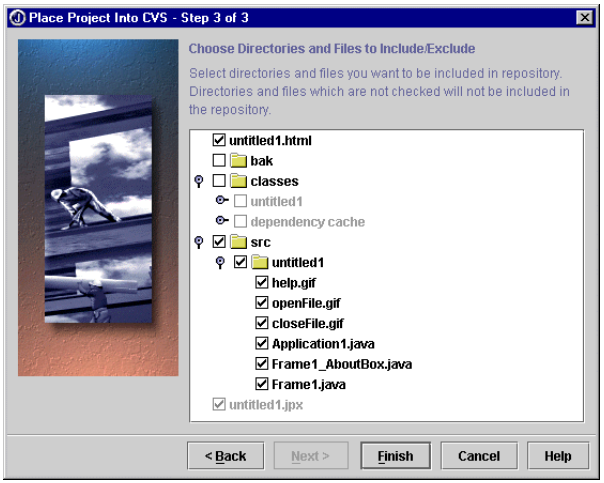
4 계속 진행하려면 Next를 클릭합니다.

5 2 단계에서는 새 모듈을 설명하는 주석에 대한 공간을 제공합니다.



6 계속 진행하려면 Next를 클릭합니다.

7 3 단계에서는 CVS에서 프로젝트의 디렉토리를 제외시킬 수 있습니다.



백업용 디렉토리 및 파생된 파일들은 기본적으로 제외됩니다. 소스 디렉토리는 기본적으로 포함됩니다. 목록에 있는 모든 디렉토리를 확장하여 각 디렉토리에 포함하려는 파일을 세부적으로 조정합니다.

주의 CVS에서 프로젝트의 하위 디렉토리를 제외시킨 다음 나중에 제외된 하위 디렉토리의 파일을 추가하려는 경우 JBuilder에서는 해당 파일을 작업 영역에 추가할 수 없습니다. 따라서 사용자가 파일을 작업 영역으로 직접 복사해야만 Team|Add 명령을 사용하여 CVS 제어 시스템에 추가할 수 있습니다.

8 Finish를 클릭하여 CVS 모듈을 생성하고 프로젝트를 CVS에 체크인합니다.

JBuilder에서 CVS 모듈을 생성하는 경우 몇 가지 작업이 수행됩니다.

- JBuilder에서 프로젝트를 추가할 모듈을 생성합니다. 프로젝트에서 .jpx 프로젝트 파일을 사용하는 경우 프로젝트 파일이 모듈에 추가됩니다. 프로젝트에서 .jpr 프로젝트 파일을 사용하는 경우 .jpx로 변환하라는 메시지가 표시됩니다. CVS 통합에는 .jpx 타입의 프로젝트 파일이 필요합니다. .jpx 프로젝트 파일은 공유할 수 있는 XML 파일 타입입니다. 프로젝트 파일 타입을 변환해도 프로젝트 파일에 포함된 모든 정보는 변경되지 않고 그대로 유지됩니다.

참고 프로젝트를 CVS로 import해도 디렉토리 트리에 있는 파일은 변경되지 않으며 트리가 작업 영역으로 변환되지 않습니다. 이로 인해 원래 파일의 무결성이 유지됩니다.

- 레포지토리를 즉시 사용할 수 있도록 프로젝트 파일을 레포지토리에 둔 후 바로 모듈이 작업 영역으로 체크아웃됩니다. JBuilder에서는 원래 디렉토리의 이름을 사용하여 모듈에서 프로젝트를 다시 체크아웃하지만 이름 뒤에 .precvs를 추가하여 원래 디렉토리의 이름 자체를 변경합니다. 그 외에는 원래 디렉토리에 대한 변경 사항이 없습니다.

이 추가 단계에서는 모듈에 포함하려는 모든 파일이 존재하며 올바른지 확인할 수 있습니다.

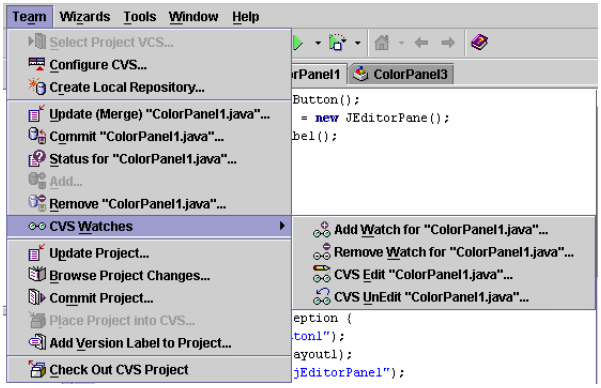
CVS 메뉴 액세스

Team 메뉴에서는 전역 유틸리티와 파일 수준 및 프로젝트 수준의 명령들을 사용할 수 있습니다. 또한 프로젝트 창의 컨텍스트 메뉴에서도 다중 선택을 지원하는 파일 수준의 명령을 사용할 수 있습니다. 이러한 메뉴는 CVS를 버전 제어 시스템으로 선택한 다음 CVS 모듈을 액세스하고 나서야 사용할 수 있습니다.

CVS 메뉴는 문맥에 따른 메뉴이므로 현재 상황에서 수행될 수 있는 경우에만 명령을 사용할 수 있습니다.

Team 메뉴 사용

Team 메뉴를 선택하여 JBuilder에서 사용할 수 있는 모든 명령을 봅니다.



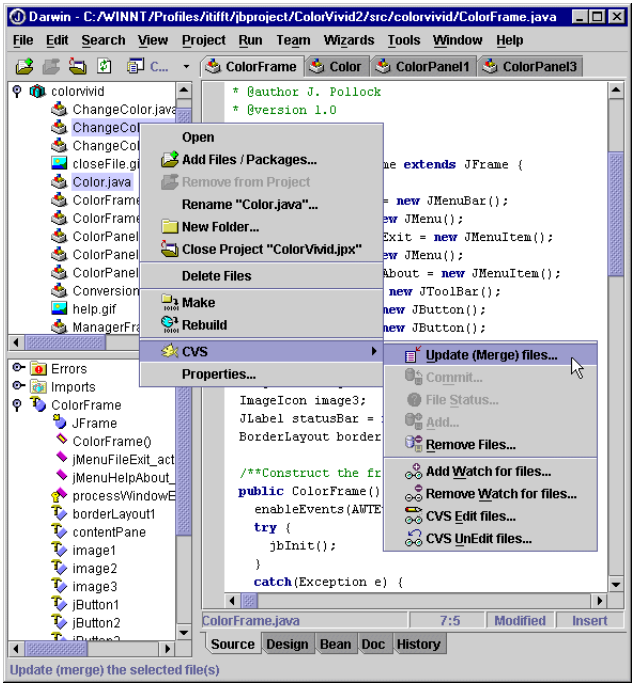
CVS 컨텍스트 메뉴 사용

다음과 같은 방법으로 컨텍스트 메뉴 명령에 액세스합니다.

1 프로젝트 창에서 대상 파일을 선택합니다.

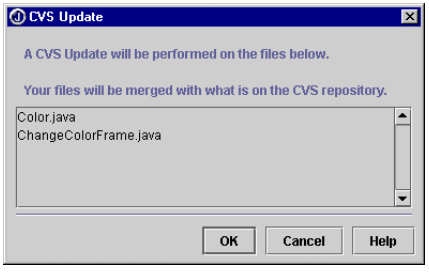
2 선택된 파일에서 마우스 오른쪽 버튼을 클릭합니다.

3 컨텍스트 메뉴에서 CVS를 선택합니다. CVS 하위 메뉴에서 명령을 선택합니다.



컨텍스트 메뉴는 문맥에 따른 메뉴입니다.

4 구분자 위에 있는 명령의 경우 JBuilder에서 선택했던 파일과 호출했던 명령을 확인합니다.



5 계속 진행하려면 OK를 클릭하고 작업을 취소하고 파일을 원래 상태로 두려면 Cancel을 클릭합니다.

파일 수준의 CVS 명령 사용

파일 수준의 CVS 명령은 선택된 파일에만 영향을 줄 뿐 전체 프로젝트에는 영향을 주지 않습니다.

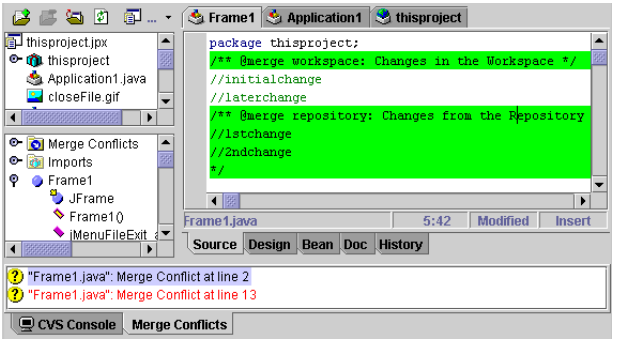
이 명령들은 메인 menu bar에 있는 Team 메뉴와 프로젝트 창의 컨텍스트 메뉴에서 사용할 수 있습니다. Team 메뉴의 이러한 명령들은 컨텐트 창에 활성화되어 있는 파일에만 적용됩니다. 프로젝트 창에서는 모든 파일에 대해 이러한 명령들을 사용할 수 있습니다. 프로젝트 창에서 파일을 선택한 다음 마우스 오른쪽 버튼을 클릭하여 CVS를 선택한 후 명령을 선택합니다.

명령에는 Update (Merge), Commit, Status, Add, Remove 및 CVS Watches의 하위 메뉴 명령들이 있습니다.

Update (Merge)

레포지토리의 변경 내용으로 작업 영역을 업데이트하는 경우 CVS는 사용자의 변경 내용과 레포지토리 변경 내용을 자동으로 병합합니다. 이로 인해 레포지토리로 다시 변경 내용을 커밋할 때 병합 충돌이 줄어듭니다.

동일한 영역의 텍스트에 다른 변경 내용이 있는 경우 CVS에서 충돌을 등록하므로 나중의 사용자는 해당 파일에 변경 내용을 커밋할 수 없습니다. JBuilder에서는 병합 충돌을 발견하면 메시지 창에 충돌 메시지를 표시하고 파일의 충돌 영역에 `@merge` 태그가 추가됩니다.



참고 CVS는 논리적 충돌이 아닌 텍스트 충돌만 인식합니다. 텍스트 충돌은 중복된 텍스트 영역을 말합니다. 즉, 동일한 영역에 서로 다른 문자들이 기록되어 있는 경우입니다. 논리적 충돌은 프로그램 상에서 발생하는 것으로서 동일한 프로그램에서 맞지 않거나 문제가 있는 프로그래밍 요소를 사용할 때 발생합니다. CVS는 프로그램의 논리적 평가를 위한 것이 아니라 파일의 물리적 평가만을 처리하기 위해 디자인되었습니다.

CVS는 텍스트 기반 파일을 처리하도록 디자인되었습니다. 이 툴은 다른 모든 파일을 바이너리로 간주합니다. CVS는 바이너리 파일을 업데이트하거나 커밋할 수 있지만 병합할 경우에는 문제가 발생할 수도 있습니다. 3- 21페이지의 "CVS에서의 바이너리 파일 처리"를 참조하십시오.

커밋하기 전에 업데이트합니다. 충돌이 있는 파일을 업데이트하면 JBuilder에서 충돌을 플래그(flag)할 수 있으며 병합 충돌 문제를 훨씬 쉽게 해결하는 기능을 설정할 수 있습니다.

AppBrowser에서 수동 또는 자동으로 병합 충돌을 조정할 수 있습니다.

- 다음과 같은 방법으로 충돌을 수동으로 조정합니다.
  1. 메시지 창에서 병합 충돌 메시지를 더블 클릭합니다. 그러면 소스 에디터 내의 첫 번째 병합 충돌 지점에 커서가 놓여집니다.
  2. 충돌이 해결될 때까지 보통의 경우처럼 텍스트를 입력 및 조작합니다.
  3. `@merge` 태그 및 관계없는 텍스트를 선택하여 삭제합니다.
  4. 파일을 저장한 후 변경 내용을 커밋합니다.
- 다음과 같은 방법으로 충돌을 자동으로 조정합니다.
  1. 메시지 창에서 첫 번째 병합 충돌 메시지를 더블 클릭합니다. 그러면 소스 에디터 내의 첫 번째 병합 충돌 지점에 커서가 놓여집니다.
  2. 충돌 옆의 여백에 있는 버튼을 클릭하거나 병합 충돌 소스를 마우스 오른쪽 버튼으로 클릭하여 에디터 메뉴를 표시합니다.
  3. 작업 영역을 유지할 것인지 레포지토리 버전을 유지할 것인지 선택합니다.
  4. 충돌이 두 개 이상이면 그 다음 병합 충돌 메시지를 더블 클릭하여 다음 충돌로 이동합니다.

참고 충돌에 `@merge` 태그가 있으면 CVS는 이 충돌이 해결되었다고 간주합니다.

팁 컴파일러는 컴파일 시 병합 충돌 태그를 발견하면 사용자에게 알려줍니다. 그리고 충돌 해결을 나중으로 미룰 경우 컴파일하고 난 다음 컴파일러의 메시지를 더블 클릭하여 에디터에서 충돌을 찾아낼 수 있습니다.

모든 병합 충돌이 해결되면 변경 내용을 커밋하고 정상적으로 업데이트합니다.

Commit

이 명령은 변경 내용을 레포지토리에 포스트합니다. 커밋 작업을 통해 레포지토리 버전을 최신 상태로 유지할 수 있습니다.

커밋하기 전에 업데이트합니다. 커밋 전에 업데이트하면 병합 충돌의 위험을 크게 줄일 수 있으며 병합 충돌이 발생하는 경우 JBuilder의 병합 지원 메커니즘을 사용합니다.

Add

파일을 레포지토리와 프로젝트 디렉토리에 추가합니다.

참고 JBuilder 프로젝트에서 참조되는 파일은 다른 디렉토리 트리에 상주할 수 있지만 CVS 모듈의 파일은 단일 트리 안에 상주해야 합니다. 그러나 이전에 CVS에게 해당 파일의 부모 디렉토리를 무시하라고 지시했다면 JBuilder 외부에 있는 프로젝트 디렉토리에 파일을 추가해야 합니다. 디렉토리 무시에 대한 자세한 내용은 3- 5페이지의 "Project를 CVS에 두기"를 참조하십시오.

Remove

레포지토리와 작업 영역에서 모두 활성 파일을 삭제합니다.

File Status

CVS에서 선택한 파일의 현재 상태, 즉 로컬로 변경했는지 원격으로 변경했는지 여부 또는 충돌이 발견되었는 여부 등을 표시합니다.

CVS Watches

이 항목 아래 있는 하위 메뉴는 다른 개발자의 파일 사용에 대한 정보를 제공하고 파일 사용을 제어할 고급 기능을 제공합니다. 이에 대한 자세한 내용은 3- 12페이지의 "CVS Watches"를 참조하십시오.

프로젝트 수준의 CVS 명령 사용

모든 명령이 버전 제어 시스템으로 CVS를 이미 선택했으며 CVS 버전 제어 하에 있는 프로젝트를 사용하고 있다고 가정합니다.

이 파일은 기존 프로젝트를 표시하거나 변경하는 Update Project, Browse Project Changes, Commit Project, 및 Add Version Label To Project와 같은 명령을 사용하는 방법을 알려 줍니다.

Update Project

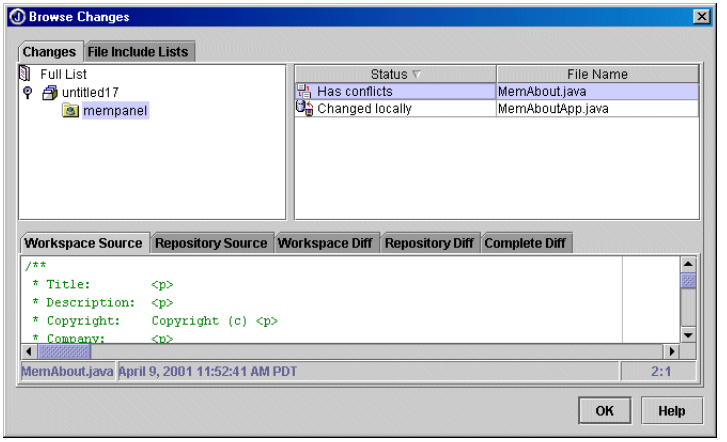
작업 영역의 모든 파일에 레포지토리의 변경 내용을 업데이트합니다. 해당 파일의 레포지토리 버전과 작업 영역 버전 간의 차이가 자동으로 병합됩니다. 충돌이 플래그(flag)됩니다.

병합 및 병합 충돌 처리에 대한 자세한 내용은 3- 10페이지의 "Update (Merge)"를 참조하십시오.

Browse Project Changes

Team|Browse Project Changes를 선택합니다. Browse Changes 대화 상자가 나타납니다. 프로젝트의 모든 파일 및 각 파일의 상태를 표시합니

다. 여기에는 두 개의 페이지, 즉 Changes 페이지와 File Include Lists 페이지가 있습니다.



Changes 페이지의 왼쪽 창에는 디렉토리 트리 노드가 있고 오른쪽 창에는 파일 목록이 있습니다. Full List 노드를 선택하면 프로젝트에 있는 모든 파일의 목록을 볼 수 있습니다. 디렉토리 노드를 선택하면 해당 디렉토리에 있는 파일 목록을 볼 수 있습니다.

각 파일의 버전 제어 상태는 목록의 Status 열에 표시됩니다.

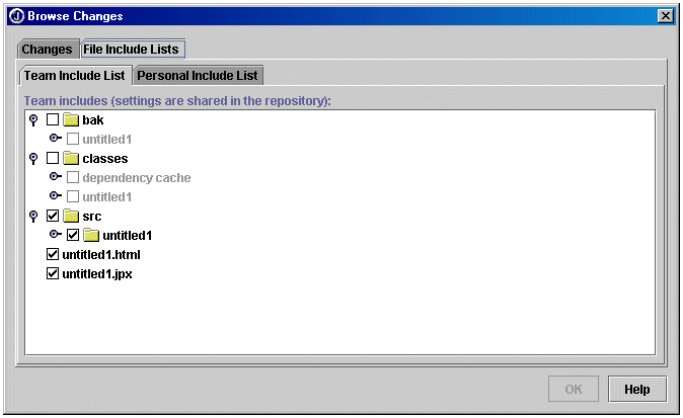
파일 목록에서 하나의 파일을 선택합니다. 파일 목록 아래에 있는 탭 모양의 창을 통해 적절한 방법으로 선택한 파일의 소스 코드를 볼 수 있습니다. 로컬로 변경된 파일을 선택하면 Workspace Source, Repository Source 및 Workspace Diff 탭을 사용할 수 있습니다. 레포지토리에서 변경된 파일을 선택한 경우 Workspace Source, Repository Source, Repository Diff Complete Diff 탭을 사용할 수 있습니다.

Changes 페이지의 탭들은 다음과 같은 정보를 보여 줍니다.

Workspace Source	이 파일의 현재 작업 영역 버전 소스 코드.
Repository Source	이 파일의 현재 레포지토리 버전 소스 코드.
Workspace Diff	작업 영역에 있는 이 파일의 최신 변경 사항.
Repository Diff	레포지토리에 있는 이 파일의 최신 변경 사항.
Complete Diff	레포지토리에 있는 이 파일의 현재 버전과 작업 영역에 있는 현재 버전의 차이점.

File Include Lists 페이지를 사용하면 Changes 페이지에서 볼 수 있는 파일과 디렉토리를 변경할 수 있습니다. 여기에는 Team Include List와 Personal Include List 두 개의 페이지가 있습니다. 이 목록은 기본적으로 CVS 제어 하에 있는 파일을 반영합니다. Browse Project Changes 대화 상자에서 이 파

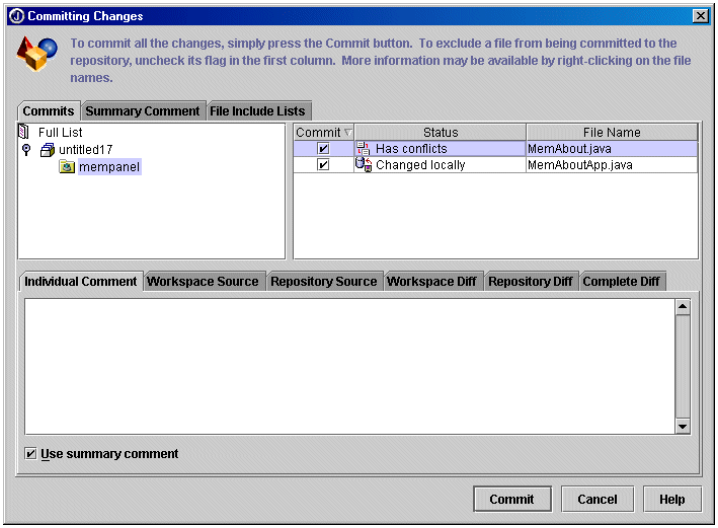
일들을 변경할 때 CVS 제어 하에 있는 파일 목록은 전혀 영향을 받지 않습니다.



이 목록에는 Changes 페이지의 뷰에서 선택된 파일과 디렉토리가 들어 있습니다. 파일이나 디렉토리를 선택 해제하면 Changes 페이지에 표시되지 않습니다. 이렇게 하면 Changes 페이지가 간단해지므로 실제로 관심 있는 파일만 볼 수 있습니다.

Commit Project

Committing Changes 브라우저는 Browse Changes 대화 상자의 모든 기능을 제공하며 개별적으로 또는 그룹으로 파일을 커밋하기 위한 기능을 추가합니다. 또한 이 브라우저에서는 CVS 제어 하에 포함시킬 파일들과 엄격하게 로컬로 유지되어야 할 파일들을 결정할 수 있습니다. 여기에는 Commits, Summary Comment, File Include Lists 세 개의 페이지가 있습니다. 기본적으로 Commits 페이지를 표시합니다.

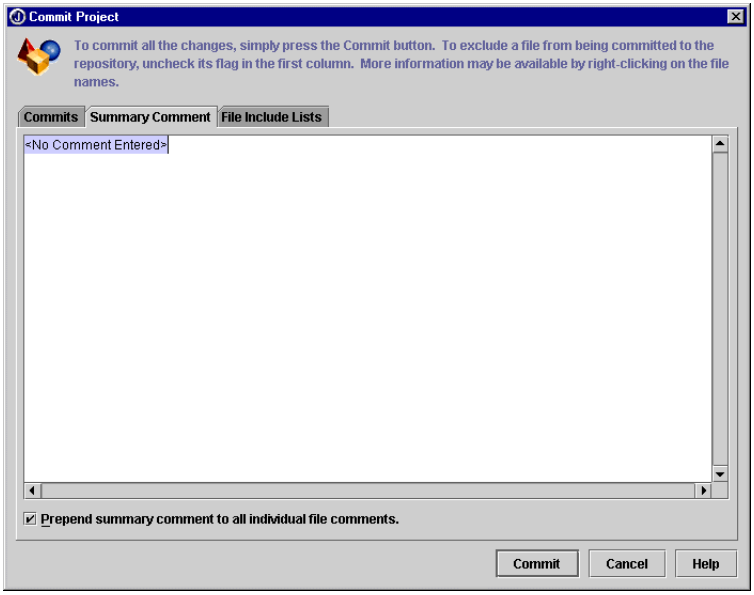


왼쪽 창에는 확장 가능한 디렉토리 트리 뷰가 있으며 오른쪽 창에는 파일 목록이 있습니다. Full List 노드를 선택하면 프로젝트에 있는 모든 파일의 목록을 볼 수 있습니다. 디렉토리 노드를 선택하면 해당 디렉토리에 있는 파일들의 목록을 볼 수 있습니다. 테이블에서 파일을 선택하면 아래의 탭 모양 창에서 개별 주석을 볼 수 있습니다.

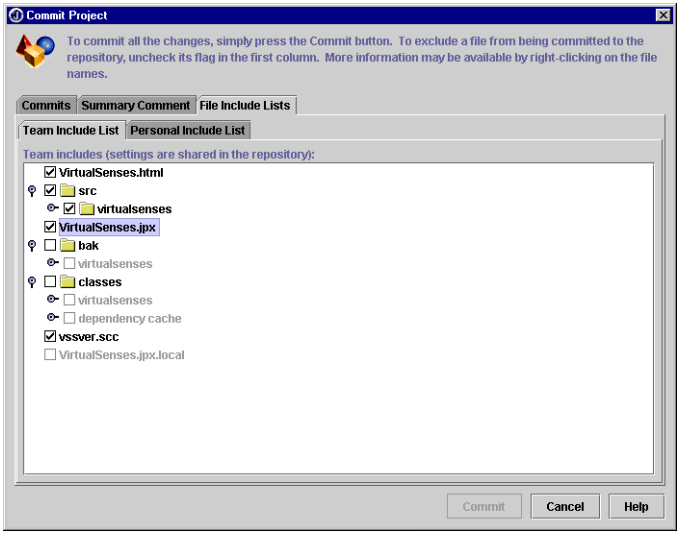
탭 모양의 창은 선택한 파일의 소스를 다음과 같은 다섯 가지 방식으로 표시합니다.

Workspace Source	이 파일의 현재 작업 영역 버전 소스 코드.
Repository Source	이 파일의 최신 레포지토리 버전 소스 코드.
Workspace Diff	작업 영역에 있는 이 파일의 최신 변경 사항.
Repository Diff	레포지토리에 있는 이 파일의 최신 변경 사항.
Complete Diff	이 파일의 현재 로컬 버전과 최신 레포지토리 버전 간의 차이점

Individual Comment 창 하단에 있는 Use Summary Comment 체크 박스를 사용하여 여러 개의 파일에 동일한 요약 주석을 추가할 수 있습니다. 이 요약 주석은 각각의 파일에 대해 개별적으로 작성된 주석과 함께 유지됩니다. Summary Comment 탭을 선택하면 요약 주석을 작성할 수 있습니다.



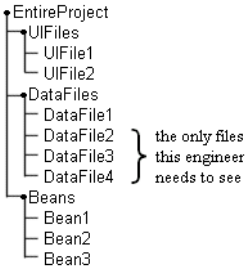
Committing Changes 브라우저의 File Include Lists는 체크인에 포함될 파일을 결정합니다. 두 개의 페이지가 있습니다. Team Include List 및 Personal Include List.



Team Include List 파일들은 <projectname>.jpx 파일에서 추적합니다. 물론, 여기에 입력한 정보는 팀 수준의 프로젝트 설정이므로 해당 파일에 저장됩니다. 여기에서 선택 표시된 파일은 모든 사람이 사용할 수 있어야 합니다.

일반적으로 bak 파일과 <projectname>.local.jpx 파일은 제외됩니다(선택 표시되지 않습니다). JBuilder에서 CVS를 유지하기 위해서는 공유된 .jpx 파일이 포함되어야 합니다. 팀 프로젝트 체크인에서 포함시킬 파일과 제외할 파일에 관한 회사 정책을 확인하십시오.

Personal Include List 파일들은 이 테이블의 정보가 저장되어 있는 <projectname>.local.jpx 파일에 의해 추적됩니다. 이 목록은 개인적인 용도로 사용하기에 편리합니다. 프로젝트의 모든 파일을 사용할 필요는 없으므로 체크인을 수행할 때마다 모든 파일을 살펴 볼 필요는 없습니다. 이 간단한 차트는 개인 개발자가 필요로 하는 파일과 비교하여 사용할 수 있는 파일의 개념을 보여 줍니다.



Personal Include List 페이지를 사용하면 사용자가 필요로 하는 파일만 표시할 수 있습니다. 나머지 파일은 숨김 상태로 유지되며 보고 싶을 때 다시 볼 수 있습니다.

Create Version Label

버전 레이블 또는 태그를 사용하면 언제라도 전체 프로젝트의 스냅샷을 얻을 수 있습니다. 서로 다른 파일들이 다른 주기로 변경되므로 100개 파일로 구성된 프로젝트는 100개의 서로 다른 현재 수정 번호를 포함합니다. 버전 레이블은 개별 파일의 변경 내용은 참조하지 않고 전체 프로젝트의 진전을 표시합니다.

Team|Add Version Label To Project를 선택하여 현재 프로젝트의 버전 레이블을 생성합니다. 그러면 Add Version Label To Project 대화 상자가 나타납니다.

관례에 따라 버전 레이블의 이름을 지정합니다. 레이블의 용도에 대한 설명을 작성합니다. 완료되면 OK를 클릭합니다. JBuilder는 레포지토리에 상주하는 프로젝트의 모든 파일에 이 레이블을 적용합니다.

고급 CVS 기능

JBuiler의 고급 CVS 기능에는 파일 액세스 제어 툴인 CVS Watches 및 CVS Edit와 보조 유틸리티인 Create Local Repository 및 CVS Helper가 있습니다.

파일 액세스 공지

일반적으로 파일을 업데이트하거나 파일의 상태를 확인할 때 다른 개발자가 파일을 변경했음을 알게 됩니다. CVS는 낙관적인 버전 제어 시스템이므로 두 명 이상의 개발자가 동시에 동일한 파일 집합을 변경할 수 있습니다. CVS Watch 기능을 사용하여 CVS에서 공용 파일을 작업하는 경우 서로 공지할 수 있습니다.

이러한 명령은 CVS Watches의 하위 메뉴에 있는 Team 메뉴와 CVS 하위 메뉴 상에 있는 프로젝트 창의 컨텍스트 메뉴에서 사용할 수 있습니다. 프로젝트 창에서 여러 파일을 선택하고 컨텍스트 메뉴를 사용하여 한 번에 두 개 이상의 파일에 Watch를 적용하거나 CVS Edit를 적용합니다.

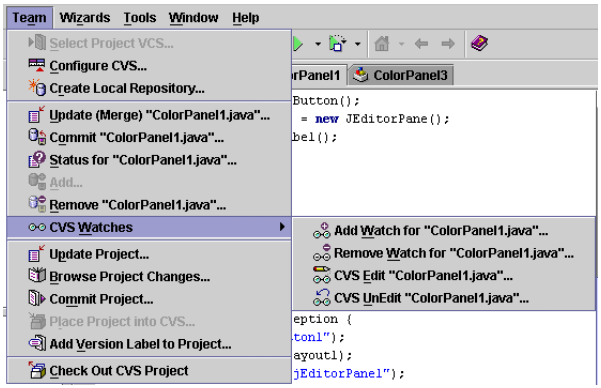
CVS Watches

CVS Watches는 체크아웃했던 파일을 다른 사람이 작업하고 있을 때 그 사실을 알려주는 방법을 제공합니다. 이런 기능은 동일한 파일 집합에 대해 공동으로 작업하는 대규모 사용자 그룹 또는 지리적으로 분산된 사용자 그룹에 특히 유용합니다.

JBuilder에서는 watch를 사용하는 환경을 수용하기 위해 워치(watch) 기능을 지원합니다. 대부분의 버전 제어 명령보다도 많은 Watch 기능을 효과적으로 사용하려면 모든 사용자에 대해 적당한 자격이 필요합니다.

JBuilder에서는 `watch add`, `watch remove`, `cvs edit` 및 `cvs unedit` CVS 명령을 지원합니다. CVS 관리자 또는 그에 상응하는 권한을 가진 사람이 watch를 사용할 수 있게 해야 합니다.

Team|CVS Watches를 선택하여 CVS Watches 하위 메뉴에 액세스합니다.



CVS Edit가 사용될 때는 모든 사용자에게 이 사실이 공지되며 서버 구성에 따라 공지의 형태는 전자 메일, 호출기를 통한 호출 또는 로그 파일 항목이 될 수 있습니다.

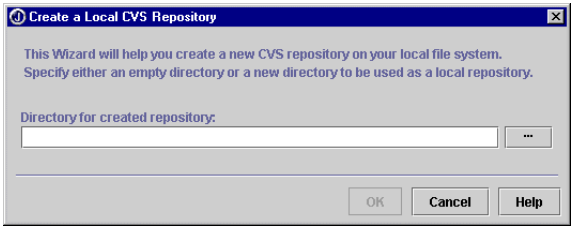
CVS Edit는 변경한 내용을 유지할지 여부에 따라 다음 두 가지 방법 중 하나로 종료될 수 있습니다.

1. 변경한 내용을 유지하려면 파일을 커밋합니다. 그러면 CVS Edit는 제거되지만 파일에 대한 Watch는 유지됩니다.
2. 파일을 변경했지만 변경을 취소하려는 경우 CVS UnEdit를 사용합니다. 다음과 같은 명령이 수행됩니다.
   - Cancel all the changes that you made since applying the CVS Edit
   - Remove the CVS Edit
   - Leave the Watch on
   - Generate a log entry

이러한 명령은 메뉴 방식으로만 작동됩니다. 이 명령들과 관련된 대화 상자는 없습니다. 컨텐트 창에 활성화되어 있는 파일에서 Team|File Status를 사용하거나 Team|Browse Project Changes를 선택한 후 Changes 페이지의 파일 테이블에서 해당 파일을 찾아서 파일의 Watch 및 Edit 상태를 볼 수 있습니다.

Create Local Repository

이 기능은 시스템에 레포지토리를 생성하여 레포지토리를 완전하게 액세스 및 관리 제어할 수 있습니다 시스템에서 허용하는 한도까지 원하는 만큼의 많은 모듈을 레포지토리에 추가할 수 있습니다.



비어 있는 디렉토리를 사용하거나 새 디렉토리를 입력하여 JBuilder가 디렉토리를 생성합니다. CVS의 관리 하위 디렉토리가 자동으로 레포지토리 디렉토리에 생성됩니다.

이 디렉토리는 레포지토리로만 사용해야 합니다. 레포지토리에 파일을 직접 추가하거나 레포지토리에서 파일을 직접 가져와서는 안됩니다. CVS에서 변경되지 않은 내용은 추적할 수 없으므로 해당 레포지토리 안에서 CVS를 사용할 수 없습니다.

CVS Helper

CVS Helper는 원격 레포지토리에 모듈 및 분기의 이름을 쿼리하는 원시 애플리케이션입니다. 이 애플리케이션을 설치한 경우 JBuilder는 CVS Helper를 자동으로 탐지하여 File|New|Team tab|Create Project From CVS를 사용할 때 기존 모듈 및 분기의 드롭다운 목록에 표시합니다.

CVS Helper를 설치하려면 단지 JBuilder CD의 `cvshelper.exe`를 pserver CVS 레포지토리를 호스트하는 시스템에 복사하기만 하면 됩니다. 프로그램을 시작하려면 명령줄에 `nohup cvshelper &`를 입력합니다. 이 프로그램은 런타임 시 필요한 매개변수를 검색하므로 별도의 구성이 필요 없습니다.

CVS Helper는 다음과 같은 네 가지 옵션을 제공합니다.

`-h` 도움말

`-p` 기본 포트 변경(이미 설정된 기본값은 2442)

`-d` 데몬 상태 토글(기본값은 on)

`-q` CVS Helper 종료

기술 정보

CVS Helper는 포트 2442에서 TCP/IP를 통해 JBuilder와 통신하는 소켓 기반 애플리케이션입니다. (기본적으로 포트 번호는 다른 값으로 변경될 수 있습니다.) 이 프로그램은 모든 사람이 사용할 수 있지만 서버가 종료되는 경우 관리자 권한이 있는 사람만 이 프로그램을 다시 시작할 수 있습니다.

통신 채널이 설정되고 나면 프로그램은 명령을 대기합니다. 각 명령은 다음과 같은 형식으로 되어 있습니다.

```
<command> <optional arguments> \n
```

현재 CVS Helper는 다음의 명령들만 인식합니다.

```
about
getVersion
getModules <repository>
getBranches <repository> <module name>
getLabels <repository> <module name>
quit
```

여기서 <repository>와 <module name>은 각각 서버 에서의 CVS 레포지토리의 절대 경로와 CVS 모듈의 이름입니다.

출력 형식은 다음과 같습니다.

```
<text>
line1
.
.
lineN
</text>
```

출력의 내용은 다음과 같습니다.

about	프로그램에 대한 저작권 정보를 반환합니다.
getVersion	버전 이름 및 작성 날짜를 나타내는 문자열을 반환합니다.
getBranches getLabels	<feedback> 데이터 블록을 전송합니다. CVS Helper는 CVS 아카이브 파일에 저장된 분기 및 레이블 정보를 읽어와 이 정보를 포함하는 파일을 생성합니다. 이 데이터 블록은 해당 파일의 이름을 반환합니다.

CVS Helper 보안 정보

CVS Helper는 표준 CVS PServer 구성을 사용하여 단계적으로 실행하도록 디자인되었습니다. 이 프로그램은 서버 시스템에서 CVS Helper 이외의 프로그램 또는 CVS Helper 명령 이외의 다른 명령을 실행하지 않습니다.

CVS Helper에서 제공하는 서비스는 모두 쿼리입니다. 이 프로그램에 의해 노출되는 데이터는 CVS 모듈 및 관련 분기 이름의 목록뿐입니다. 이 데이터를 CVS Helper를 실행하는 서버의 다른 사용자들과 공유할 수 있는 경우 CVS Helper를 실행할 수 있습니다.

방화벽을 통해 CVS를 사용하며 CVS Helper를 사용하려는 경우 CVS 액세스를 허용하는 것과 마찬가지로 CVS Helper 액세스도 허용하도록 방화벽을 구성해야 합니다.

CVS 참조

이 단원에서는 CVS에서의 바이너리 파일에 대한 정보와 시스템 상의 환경 변수 설정 방법을 제공합니다. 또한 일반적으로 사용되는 CVS 용어에 대한 간략한 용어집도 제공합니다.

CVS에서의 바이너리 파일 처리

JBuilder가 CVS에서 텍스트 기반이 아닌 파일 타입을 아직 인식하지 못하는 경우에는 JBuilder의 Generic Binary 파일 타입 목록에 구체적으로 파일의 확장자를 포함시켜야 합니다. 다음과 같은 방법으로 목록에 파일 확장자를 추가합니다.

1. Tools|IDE Options를 선택합니다.
2. File Types 탭을 선택합니다.
3. Generic Binary 파일 타입을 선택합니다.
4. Add를 클릭합니다.
5. 새 확장자를 입력합니다.

필요한 경우 `.precvs` 디렉토리에 저장된 파일을 사용하여 원래 바이너리 파일을 복원할 수 있습니다.

사용자 환경 변수 확인 및 설정

환경 변수에 액세스하는 방법은 Linux, Solaris, Windows NT 또는 Windows 98/2000 등의 플랫폼에 따라 다릅니다.

CVS의 변수는 다음 이름에 대한 값을 포함해야 합니다.

- `cvs`: 이것은 CVS가 설치된 디렉토리입니다. 대개는 최상위 수준의 디렉토리가 되므로 아마도 드라이브의 루트 디렉토리일 것입니다.
- `cvsroot`: 이것은 사용할 서버의 경로입니다. 이 경로의 구문은 아래에 있는 표를 참조하십시오.
- JBuilder에서 CVS를 사용하고 나면 JBuilder는 `.cvspass`라는 변수를 생성합니다. 이 변수는 변경하면 안됩니다.

필요한 경우에는 CVS 마법사에서 구성 페이지를 제공합니다. 이 페이지의 하단부에 CVSROOT 변수가 표시되며 마법사에서 구성 필드를 완료할 때 변경됩니다. JBuilder가 지원하는 각 서버 연결 타입에서 CVSROOT 변수는 다음 구문들로 구성됩니다.

연결 타입	CVSROOT 구문
Local	:local:<repository path>
PServer	:pserver:<user name>@<server name>:<repository path>
Ext	:ext:<user name>@<server name>:<repository path>

Linux와 Solaris

셸의 `init` 파일에서 `PATH` 환경 변수를 편집합니다. 다음을 포함시킵니다.

- CVS가 설치된 디렉토리의 경로
- 서버에 있는 레포지토리의 경로

각 `PATH` 변수는 세미콜론으로 구분해야 합니다.

Windows NT

Windows에서 사용자 환경 변수는 Settings 디렉토리에 저장됩니다. 다음과 같은 두 가지 방법으로 액세스할 수 있습니다. Start|Settings|Control Panel|System을 선택하거나, 바탕 화면에서 내 컴퓨터 아이콘을 마우스 오른쪽 버튼으로 클릭한 다음 등록 정보를 선택하면 됩니다.

Windows NT에서는 다음과 같이 합니다.

1 Environment 탭을 선택합니다.

2 User Variables에서 CVS에 속하는 환경 변수를 찾습니다.

3 변경할 변수를 선택합니다. 새 변수 이름을 입력한 다음 새 변수에 사용할 값을 입력합니다.

4 완료되면 OK를 클릭하거나 ***Enter***를 누릅니다.

Windows 98 및 Windows 2000

Windows에서 사용자 환경 변수는 Settings 디렉토리에 저장됩니다. 다음과 같은 두 가지 방법으로 액세스할 수 있습니다. Start|Settings|Control Panel|System을 선택하거나, 바탕 화면에서 내 컴퓨터 아이콘을 마우스 오른쪽 버튼으로 클릭한 다음 등록 정보를 선택하면 됩니다.

Windows 98/2000에서는 다음과 같이 합니다.

1 고급 탭을 선택합니다.

2 대화 상자 중간에 있는 Environment Variables를 클릭합니다.

3 Edit를 클릭하여 기존 환경 변수를 변경하거나 New를 클릭하여 새 변수를 만듭니다.

CVS 용어집

이 단어들은 CVS에서 특정한 의미를 갖습니다.

작업 영역 (Workspace) 사용자가 직접 영향을 미치는 영역이므로 사용자 스스로 관리해야 합니다. 파일을 변경하는 경우 변경하고 나서 먼저 작업 영역에 저장해야 합니다.

업데이트 (Update) 레포지토리의 변경 내용을 가져와 작업 영역에 적용합니다.

레포지토리 (Repository) 버전 제어 시스템에 추가했던 모듈 및 수정 레코드가 유지되는 곳입니다. 레포지토리는 로컬 시스템에 있을 수도 있고 원격 서버에 있을 수도 있습니다.

제거 (Remove) 레포지토리에서 파일을 제거합니다. 파일을 작업 영역에서 먼저 삭제해야 합니다.

프로젝트 (Project) 다음과 같은 세 가지 의미로 사용될 수 있습니다. 하나의 작업 몸체를 구성하는 파일 및 설정, JBuiler에서 그러한 파일 및 설정의 목록을 관리하는 구성 파일, 또는 "모듈"과 같은 의미로 쓰입니다.

모듈 (Module) 보다 나은 파일 관리와 사용자 편의를 위해 레포지토리에 함께 저장되는 관련 파일 그룹의 컨테이너입니다.

병합(Merge) CVS 및 일부 다른 시스템에서는 파일이 사용 중일 때 파일을 잠그지 않아도 됩니다. 모든 변경 내용을 유지하기 위해 이러한 시스템에서는 merge 명령을 사용합니다. 이 명령은 레포지토리의 변경 내용과 작업 영역의 변경 내용을 결합합니다. 저장된 작업 영역의 변경 내용을 덮어쓰지 않으므로 사용자가 조정하도록 텍스트 충돌이 유지 및 플래그(flag)됩니다. CVS에서는 업데이트에 병합이 포함됩니다.

커밋 (Commit) 변경 내용을 작업 영역에서 레포지토리로 적용합니다. 다른 사용자가 사용할 수 있도록 파일 변경, 파일 추가 및 파일 삭제 내용을 버전 제어 시스템에 커밋해야 합니다.

체크아웃 (Checkout) 모듈을 레포지토리에서 작업 영역으로 포스트합니다. CVS를 사용하는 경우 이 작업은 처음에만 하면 됩니다. 그리고 나면 업데이트에 의해 파일이 동기화될 수 있습니다.

추가(Add) 모듈에 파일을 추가합니다.

Chapter

4

JBuilder에서의 Visual SourceSafe

이것은 JBuilder 기업용 버전의 기능입니다.

JBuilder에 통합된 Visual SourceSafe(VSS)를 통해 개발 환경 안에서 가장 일반적인 버전 제어 작업을 수행할 수 있습니다. JBuilder는 필요한 디렉토리를 찾는 데 대한 힌트 등 사용자 편의성을 제공합니다.

JBuilder에의 VSS 통합은 Visual SourceSafe 6.0 버전용으로 설계, 테스트되었습니다. 이 통합은 VSS의 명령줄 인터페이스에 기반을 두고 있으므로여러 VSS 버전 간에 최상의 호환성을 제공하면서 대부분의 기능을 사용할 수 있습니다. JBuilder에 통합된 VSS는 Windows NT, 98, 2000에서 사용할 수 있습니다.

사용자의 시스템에 Visual SourceSafe 클라이언트가 설치되어 있어야 JBuilder에서 Visual SourceSafe 메뉴 명령에 액세스할 수 있습니다. 기존 VSS 데이터베이스에 연결하기 위해서는 VSS 데이터베이스가 있는 디렉토리에 LAN으로 액세스할 수 있어야 합니다. Windows Explorer에서 VSS 디렉토리를 볼 수 있다면 액세스할 수 있는 것입니다.

Team 메뉴에서 다음과 같은 Visual SourceSafe 명령을 사용할 수 있습니다.

- Select Project VCS
- Configure Visual SourceSafe
- Visual SourceSafe Explorer
- Check Out File
- Check In File
- Add File
- Remove File
- Undo Checkout
- Version Label 생성
- Pull Latest Project File
- Post Current Project File
- Browse Project Changes
- Check In Project
- Place Project Into VSS
- Pull Project From VSS

그 밖의 다른 명령들은 Team|Visual SourceSafe Explorer를 선택하여 VSS를 호출한 후 VSS 내에서 실행해야 합니다.

참고 명령은 문맥에 따라 다릅니다.

Visual SourceSafe 사용

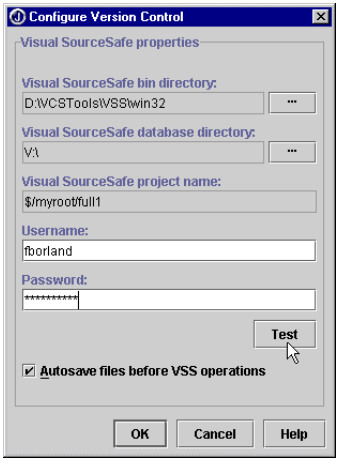
여기에서는 사용자의 시스템에 VSS 클라이언트가 설치되어 있으며 VSS 데이터베이스에 LAN으로 액세스할 수 있다고 가정합니다.

연결 구성

Place Project Into VSS(Team 메뉴)와 Pull Project From VSS(Team 메뉴나 객체 갤러리)는 작업하려는 프로젝트에 대한 연결을 구성하는 마법사를 사용합니다. 각 프로젝트의 구성은 개별적으로 설정됩니다.

마법사는 데이터베이스로의 연결 설정, 작업 폴더 설정, 체크 아웃한 상태로 있을 파일 선택 등의 작업을 도와 줍니다. 해당 프로젝트에 대한 연결이 일단 구성되면 다시 설정할 필요가 없습니다.

Team|Configure Version Control을 선택하여 구성을 보고 테스트할 수 있습니다. Configure Version Control 대화 상자가 나타납니다.



참고 사용자 이름과 암호 필드는 쓰기 가능합니다. Visual SourceSafe에 대한 사용자 이름이나 암호가 변경된 경우, Team|Configure Version Control을 선택하여 이러한 필드들을 현재의 식별 데이터와 일치하도록 변경해 줍니다. Test 버튼을 클릭하여 실제로 사용하기 전에 구성이 제대로 되어 있는지 확인합니다.

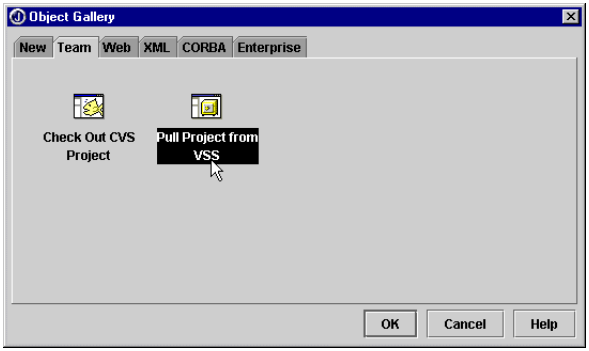
VSS에서 프로젝트 가져오기

JBuilder에서 데이터베이스로부터 프로젝트를 가져오는 방법에는객체 갤러리 또는 Team 메뉴를 이용하는두 가지 방법이 있습니다(이것은 객체 갤러리에서 사용할 수 있는 유일한 VSS 명령입니다).

객체 갤러리에서 다음과 같이 합니다.

1 File|New를 선택합니다. 객체 갤러리가 나타납니다.

2 Team 탭을 선택합니다.



3 Choose Pull Project From VSS를 선택합니다.

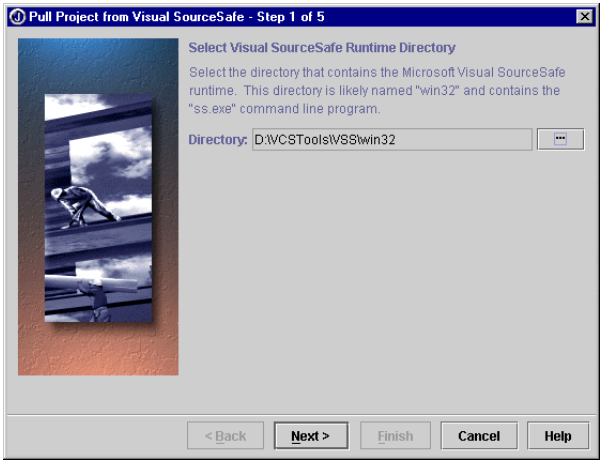
4 OK를 클릭하거나 Enter를 누릅니다.

Team 메뉴에서 다음과 같이 합니다.

1 Team|Pull Project From VSS를 선택합니다.

두 방법 모두 Pull Project From Visual SourceSafe 마법사를 불러 옵니다. 이 마법사에서는 연결을 구성하고 Visual SourceSafe 제어 하의 프로젝트에 대한 작업에 필요한 모든 것을 설정합니다. 유용한 프롬프트들이 사용자를 제대로 갈 수 있게 돕습니다.

JBuilder가 Visual SourceSafe 런타임 디렉토리 위치를 모를 경우 마법사는 다음 다섯 단계를 갖습니다. 첫 단계는 다음과 같이 JBuilder에게 Visual SourceSafe 런타임 디렉토리 위치를 알려주는 단계입니다.



팁 생략 버튼을 클릭하면 탐색 상자가 나타납니다. 마법사에 있는 설명에 일치하는 디렉토리를 찾습니다. 잘못된 디렉토리를 선택하면 잘못 선택하였다는 것을 알리는 메시지가 나타납니다. 그러한 경우에는 생략 버튼을 클릭하여 다시 탐색합니다.

이 단계가 성공적으로 수행되었다면 Visual SourceSafe 하의 모든 프로젝트에 대해 마법사는 다음과 같은 네 단계를 갖습니다. .

1 Visual SourceSafe 데이터베이스 디렉토리를 선택합니다.

2 사용자 이름과 암호를 입력합니다.

3 Visual SourceSafe 프로젝트를 선택합니다.

4 비어 있는 대상 디렉토리를 선택합니다.

참고 JBuilder는 Visual SourceSafe에서 사용자의 액세스 권한을 변경하지 않습니다. JBuilder에서 VSS 명령을 실행하려면 적절한 액세스 권한이 있어야 합니다.

Visual SourceSafe에서 프로젝트를 가져오는 데 대한 자세한 내용은 마법사의 각 페이지에서 Help 버튼을 클릭하십시오.

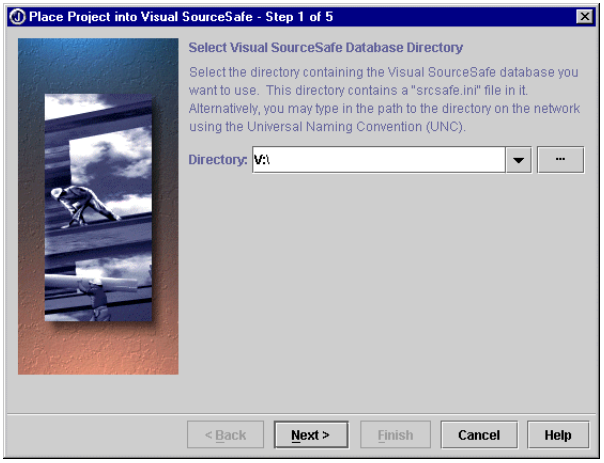
VSS에 프로젝트 놓기

이것은 Place Project Into VSS 마법사를 호출합니다. 이 마법사는 JBuilder 프로젝트를 열고 기존 VSS 데이터베이스에 대응하는 VSS 프로젝트를 생성합니다. 마법사에서 사용자는 VSS 프로젝트에 포함될 파일과 하위 디렉토리 및 VSS에 놓인 뒤에 체크 아웃된 채로 남아있을 파일을 지정할 수 있습니다.

JBuilder가 Visual SourceSafe 런타임 디렉토리 위치를 모를 경우 마법사는 여섯 단계를 갖습니다. 첫 단계는 JBuilder에게 Visual SourceSafe 런타임 디렉토리 위치를 알려주는 단계입니다. VSS 런타임 디렉토리 경로를 모르거나 제공된 힌트에 맞게 보이지 않는다면 VSS 관리자에게 문의하십시오.

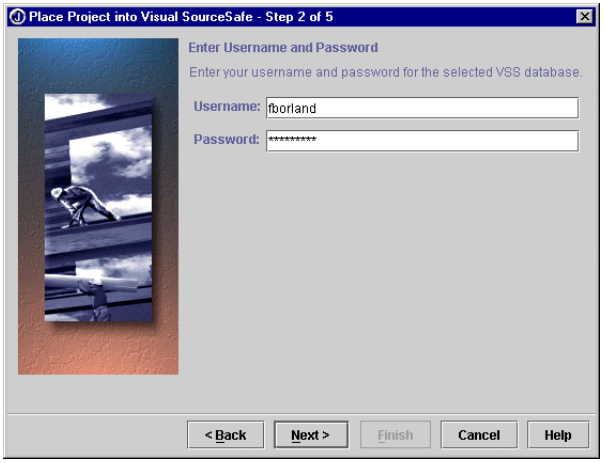
런타임 디렉토리 위치를 알고 있을 경우 마법사는 다음과 같은 네 단계로 이루어집니다. .

1 Team|Place Project Into VSS를 선택합니다. 다음과 같이 Select Visual SourceSafe Database Directory 페이지가 나타납니다.



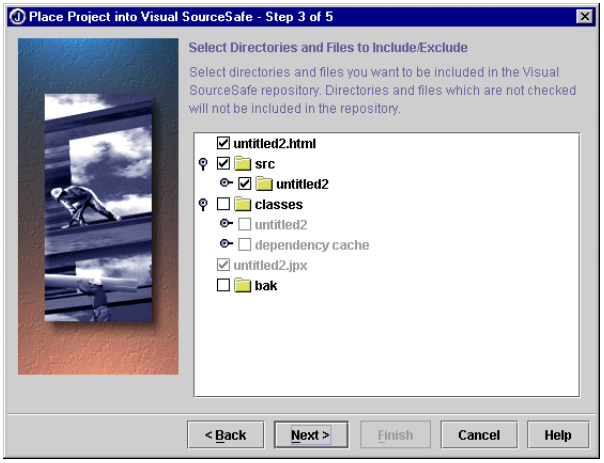
VSS 데이터베이스 디렉토리 경로를 모르거나 제공된 힌트에 맞게 보이지 않는다면 VSS 관리자에게 문의하십시오.

2 Next를 클릭하여 Enter Username and Password 페이지로 넘어 갑니다.



Visual SourceSafe에서 사용자 이름이나 암호가 변경되었다면 Team|Configure Visual SourceSafe를 선택하여 일치시킬 수 있습니다.

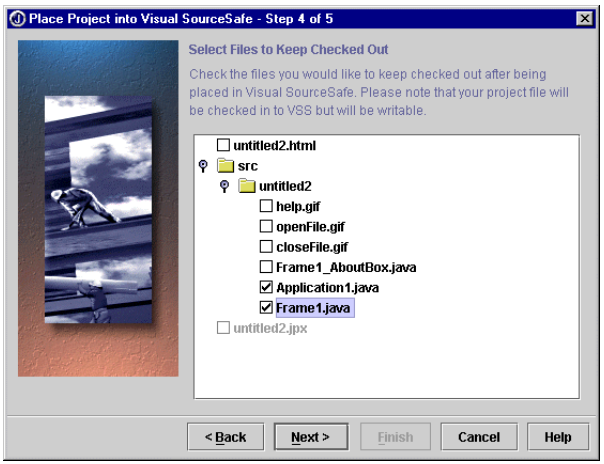
3 Next를 클릭하여 Select Directories And Files To Include 페이지로 넘어 갑니다.



전체적으로 포함시키려는 디렉토리를 선택 표시합니다. 개별적으로 포함시킬 파일을 선택하려면 트리 뷰에서 디렉토리를 확장한 후 원하는 파일에 선택 표시합니다.

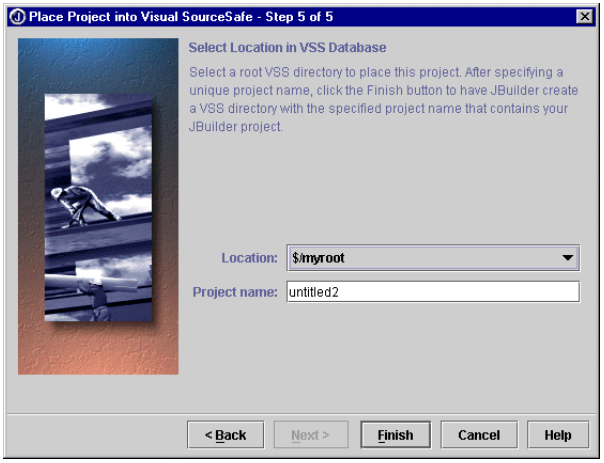
기본적으로 `bak`과 `classes` 디렉토리는 포함되어 있지 않고 `src` 파일은 포함되어 있습니다. `.jpx` 프로젝트 파일은 그 밖의 다른 프로젝트 설정 뿐만 아니라 팀 전체 버전 제어 설정 및 기본 설정을 관리하는 파일이므로 포함되어야 합니다.

4 Next를 클릭하여 Select Files To Keep Checked Out 페이지로 넘어 갑니다.



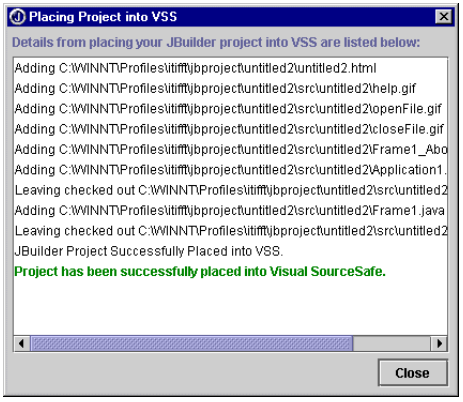
디렉토리를 확장하여 안에 있는 파일을 나타나게 한 후프로젝트가 VSS에 놓인 순간부터 즉시 작업할 수 있도록 자신의 작업 영역으로 체크 아웃하고자 하는 파일을 선택 표시합니다.

5 Next를 클릭하여 Select Location In VSS Database 페이지로 넘어 갑니다.



새 프로젝트가 들어갈 기존의 루트 VSS 디렉토리를 선택합니다. 이 프로젝트에 대한 고유한 프로젝트 이름을 입력합니다. 데이터베이스의 루트 디렉토리 내에 해당 프로젝트 디렉토리가 만들어집니다.

6 Finish를 클릭합니다. Placing Project Into VSS feedback 대화 상자가 나타납니다. 명령의 진행 상황을 보고하고 명령 수행이 완료되면 사용자에게 알려 줍니다.



7 Close를 클릭하여 대화 상자를 닫고 IDE로 돌아갑니다.

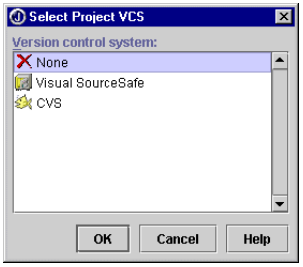
체크 아웃한 파일은 사용자가 다시 체크 인할 때까지 사용자에게 쓰기 액세스가 할당된 상태로 컨텐트 창에 나타납니다.

VSS 명령에 액세스

VSS 명령은 메인 menu bar에 있는 Team 메뉴와 프로젝트 창의 컨텍스트 메뉴에서 사용할 수 있습니다.

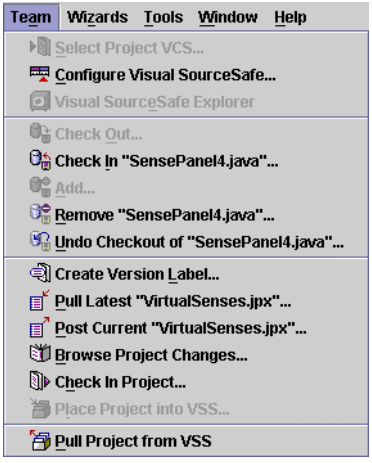
다음과 같은 방법으로 JBuilder IDE 내에서 VSS 명령에 액세스합니다.

- 메인 메뉴에서 Team을 선택합니다.
- Select Project VCS를 선택합니다. Select Project VCS 대화 상자가 나타납니다.



- Select Project VCS 대화 상자에서 Visual SourceSafe를 선택합니다.
- OK를 클릭하여 대화 상자를 닫습니다.

Team 메뉴에 Visual SourceSafe 명령이 반영되어 나타납니다. 각 명령은 VSS에서 액세스 가능할 때 메뉴에서 사용할 수 있도록 활성화됩니다. VSS 프로젝트에 액세스하고 나면 더 많은 명령이 활성화되고 Team 메뉴는 다음과 같습니다.



파일 수준 명령에는 Check Out, Check In, Undo Checkout, Add, Remove 등이 있습니다.

프로젝트 수준 명령에는 Create Version Label, Pull Latest Project File, Post Current Project File, Browse Project Changes, Check In Project 등이 있습니다.

파일 수준 명령과 프로젝트 수준 명령 모두 Team 메뉴에서 액세스할 수 있습니다. Team 메뉴는 컨텐트 창에 활성화되어 있는 파일에 대해 파일 수준 명령을 실행합니다. 이 메뉴는 현재 프로젝트에 대해 프로젝트 수준 명령을 실행합니다.

파일 수준 명령은 프로젝트 창의 컨텍스트 메뉴(마우스 오른쪽 버튼을 클릭하면 나타나는 메뉴)에서도 사용할 수 있습니다. JBuilder는 프로젝트 창의 다중 선택을 지원합니다. 따라서 이 명령들은 동시에 여러 파일에 적용할 수 있습니다.

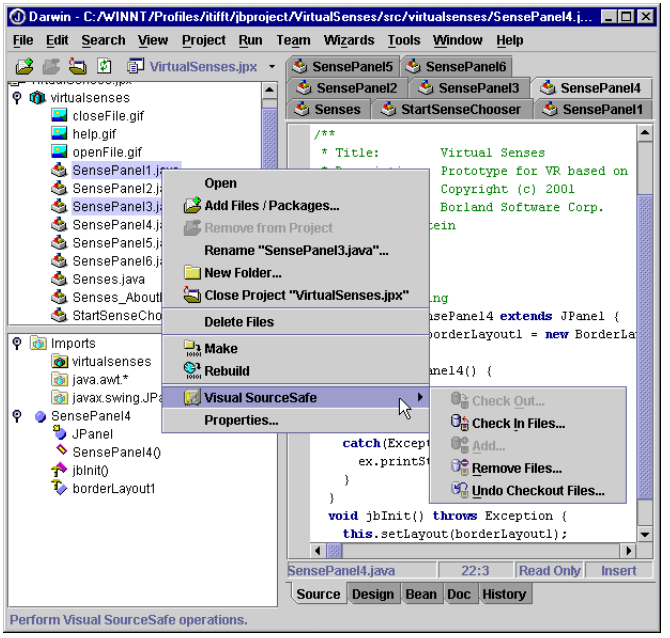
다음과 같은 방법으로 이러한 명령에 액세스합니다.

1 명령을 사용할 하나 이상의 파일을 선택합니다.

2 선택한 노드 중 하나에서 오른쪽 마우스 버튼을 클릭합니다.

3 컨텍스트 메뉴에서 Visual SourceSafe를 선택합니다.

4 Visual SourceSafe 하위 메뉴에서 사용하려는 명령을 선택합니다.



컨텐트 창에는 하나의 파일이 활성화되어 있지만 프로젝트 창에는 Visual SourceSafe 명령을 위한 두 개의 다른 파일이 선택되어 있습니다.

파일 수준 Visual SourceSafe 명령

파일 수준 명령은 Team 메뉴 또는 프로젝트 창의 컨텍스트 메뉴(마우스 오른쪽 버튼을 클릭하면 나타나는 메뉴)에서 사용할 수 있습니다. Team 메뉴는 컨텐트 창에 활성화되어 있는 파일에 명령을 적용하는 반면,컨텍스트 메뉴는 프로젝트 창에서 사용자가 선택한 모든 파일에 명령을 적용합니다. 즉, 동시에 여러 파일에 명령을 적용할 수 있습니다.

- Check Out
- Undo Checkout
- Check In
- Add File
- Remove File

Checkout

이 명령은 선택한 하나 이상의 파일을 현재 VSS 프로젝트에서 사용자의 작업 영역에 있는 폴더로 복사합니다. 파일을 체크 아웃하면 그 파일은 사용자가 다시 체크 인하기 전까지 다른 사용자에게는 읽기 전용이 됩니다.

이 명령을 사용하려면 Check Out 액세스 권한이 있어야 합니다.

Undo Checkout

선택한 파일들에 대한 체크 아웃을 취소하고 모든 변경 사항을 무시합니다. 이 명령을 사용하려면 Check Out 액세스 권한이 있어야 합니다. 이 명령은 해당 사용자에게 체크 아웃되지 않은 파일들은 무시합니다.

`ss.ini` 파일의 `Delete_Local` 초기화 변수가 `Yes`로 설정되어 있지 않은 경우, Undo Checkout은 완료했을 때 `Get` 연산을 수행합니다.

Check In

체크 아웃된 하나 이상의 파일에 만들어진 변경 내용으로 데이터베이스를 업데이트합니다. 체크 인 대화 상자에는 해당 파일들에 대해 계속 작업할 수 있도록 파일을 체크 아웃된 상태로 유지할 것인지 묻는 옵션이 있습니다. 사용자가 체크 아웃된 상태를 유지하지 않겠다고 결정하면 Check In 명령은 해당 파일들에 대한 VSS 마스터 복사본을 잠금 해제하여 다른 사용자들이 체크 아웃할 수 있게 합니다.

이 명령을 사용하려면 Check Out 액세스 권한이 있어야 합니다.

다른 사용자가 변경되지 않은 파일을 체크 인하여 부주의하게 새 버전을 생성한 경우, VSS는 자동적으로 Undo Check Out 명령을 수행하며해당 파일을 잠금 해제하여 다시 액세스 가능하게 하고, 새 버전은 기록하지 않습니다.

Add

VSS 데이터베이스에 파일을 추가합니다. 이 명령을 사용하려면 Add 액세스 권한이 있어야 합니다.

VSS는 AutoDetect 기능을 사용하여 추가된 파일이 텍스트 파일인지 이진 파일인지 여부를 지정합니다.

Remove

VSS Explorer에서 파일들을 제거하고 삭제 표시를 합니다. 하지만 항목은 여전히 존재하며 Recover 명령(Visual SourceSafe에서 액세스 가능)을 사용하여 복구할 수 있습니다.

프로젝트 수준 Visual SourceSafe 명령

프로젝트 수준 명령은 전체 프로젝트나 프로젝트 파일(전체 프로젝트에 대한 정보를 관리하는 파일)에 영향을 미칩니다. 이러한 명령들은 모두 Team 메뉴에서 사용할 수 있습니다.

- Pull Latest .jpx Project File
- Post Current .jpx Project File
- Browse Project Changes
- Check In Project
- Version Label 생성

프로젝트 파일 가져오기 및 포스트

프로젝트 파일은 구축 옵션 및 프로젝트 뷰 구조 등의 프로젝트 설정을 유지합니다. 이 정보는 프로젝트에 참여하는 다른 사용자들과 공유되는 정보입니다. JBuilder를 통해 가능한 많은 사용자들이 사용할 수 있도록 프로젝트 파일을 개별적으로 가져오거나 포스트할 수 있습니다.

실행 및 디버깅 기본 설정 같은 사용자 개별적인 설정은 `<projectname>.local.jpx`라는 로컬 프로젝트 파일에 저장됩니다. 이 파일은 VSS의 영향을 받지 않습니다. 즉 사용자의 로컬 기본 설정은 다른 사용자의 설정에 의해 겹쳐 쓰여지지 않습니다.

프로젝트에 파일에 대한 자세한 내용은 *JBuilder를 이용한 애플리케이션 구축*에서 "프로젝트 생성 및 관리"를 참조하십시오.

Pull Latest .jpx Project File

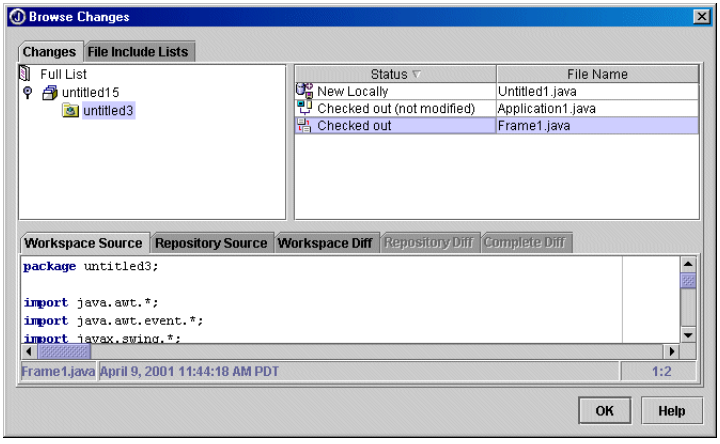
VSS 데이터베이스에서 활성화된 프로젝트에 대해 가장 최근의 공유된 `.jpx` 프로젝트 파일을 가져옵니다.

Post Current .jpx Project File

.jpx 파일의 사용자 작업 영역 버전을 VSS 데이터베이스에 포스트합니다.

Browse Project Changes

Team|Browse Project Changes를 선택합니다. Browse Changes 대화 상자가 나타납니다. 프로젝트에서 변경된 모든 파일 및 각 파일의 상태를 표시합니다. 여기에는 Changes와 File Include Lists라는 두 개의 페이지가 있습니다.



Changes 페이지의 왼쪽 창에는 디렉토리 트리 뷰가 있고 오른쪽 창에는 파일의 목록이 있습니다. Full List 노드를 선택하면 프로젝트에 있는 모든 파일의 목록을 볼 수 있습니다. 디렉토리 노드를 선택하면 해당 디렉토리에 있는 파일들의 목록을 볼 수 있습니다.

각 파일의 버전 제어 상태는 목록의 Status 열에 표시됩니다.

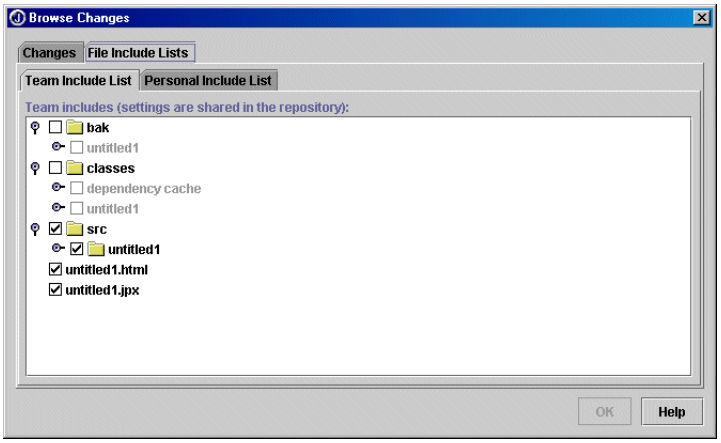
파일 목록에서 하나의 파일을 선택합니다. 파일 목록 아래에 있는 탭 모양의 창을 통해 선택한 파일의 소스 코드를 적절한 방법으로 볼 수 있습니다. 로컬로 변경된 파일을 선택하면 Workspace Source, Repository Source 및 Workspace Diff 탭을 사용할 수 있습니다. 레포지토리에서 변경된 파일을 선택한 경우, Workspace Source, Repository Source, Repository Diff Complete Diff 탭을 사용할 수 있게 됩니다.

Changes 페이지의 탭들은 다음을 보여 줍니다.

Workspace Source	이 파일의 현재 작업 영역 버전 소스 코드.
Repository Source	이 파일의 현재 레포지토리 버전 소스 코드.
Workspace Diff	작업 영역에 있는 이 파일의 최신 변경 사항.

Repository Diff	레포지토리에 있는 이 파일의 최신 변경 사항.
Complete Diff	레포지토리에 있는 이 파일의 현재 버전과 작업 영역에 있는 현재 버전의 차이점.이는 동시에 양쪽 영역을 모두 변경할 수 있는 낙관적인 버전 제어 시스템에서 활성화됩니다. VSS에서는 사용할 수 없습니다.

File Include Lists 페이지를 사용하면 Changes 페이지에서 볼 수 있는 파일과 디렉토리를 변경할 수 있습니다. 여기에는 Team Include List와 Personal Include List 두 개의 페이지가 있습니다. 이 목록은 기본적으로 VSS 제어 하에 있는 파일을 반영합니다. Browse Project Changes 대화 상자에서 이 파일들을 변경할 때 VSS 제어 하에 있는 파일 목록은 전혀 영향을 받지 않습니다.

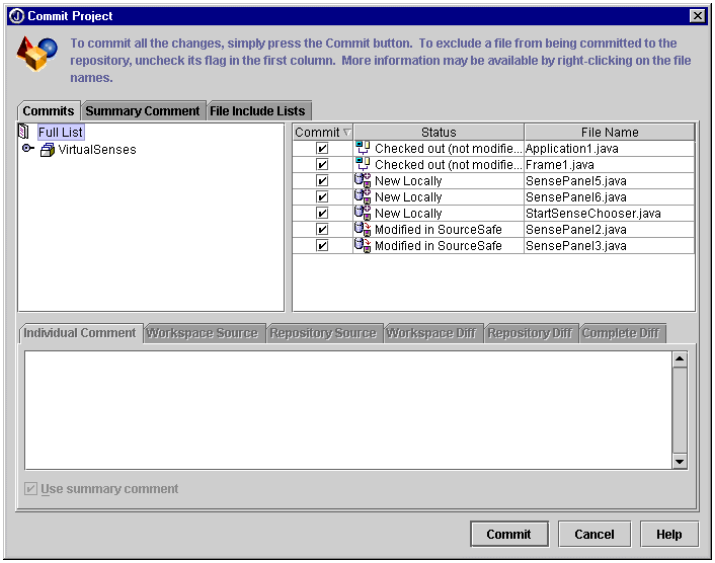


이 목록에는 Changes 페이지의 뷰에서 선택된 파일과 디렉토리가 들어 있습니다. 파일 또는 디렉토리를 선택 해제하면 Changes 페이지에 표시되지 않습니다. 이로 인해 Changes 페이지가 간단해지므로 실제로 관심 있는 파일만 볼 수 있습니다.

Commit Project

Commit Project 브라우저는 Browse Changes 대화 상자의 모든 기능을 제공하며 파일을 개별적으로 또는 그룹으로 선택 표시하기 위한 기능을 추가합니다. 또한 이 브라우저를 통해 VSS 제어 하에 포함시킬 파일과 엄격하게 로컬로 유지되어야 할 파일을 결정할 수 있습니다. Commit Project

브라우저는 Commits, Summary Comment, File Include Lists 세 개의 페이지를 갖습니다. 기본적으로 Commits 페이지를 표시합니다.



왼쪽 창에는 확장 가능한 디렉토리 트리 노드가 있으며 오른쪽 창에는 파일의 목록이 있습니다. Full List 노드를 선택하면 프로젝트에 있는 모든 파일의 목록을 볼 수 있습니다. 디렉토리 노드를 선택하면 해당 디렉토리에 있는 파일들의 목록을 볼 수 있습니다. 테이블에서 파일을 선택하면 아래의 탭 모양 창에서 개별 주석을 볼 수 있습니다.

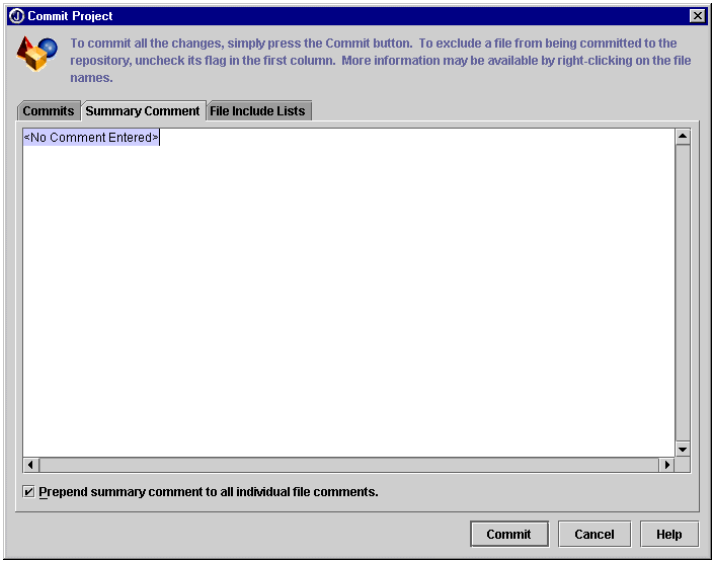
탭 모양의 창은 선택한 파일의 소스를 다음과 같은 네 가지 방식으로 표시합니다.

Workspace Source	이 파일의 현재 작업 영역 버전 소스 코드
Repository Source	이 파일의 최신 데이터베이스 버전 소스 코드.
Workspace Diff	작업 영역에 있는 이 파일의 최근 변경 사항.
Repository Diff	데이터베이스에 있는 이 파일의 최근 변경 사항.

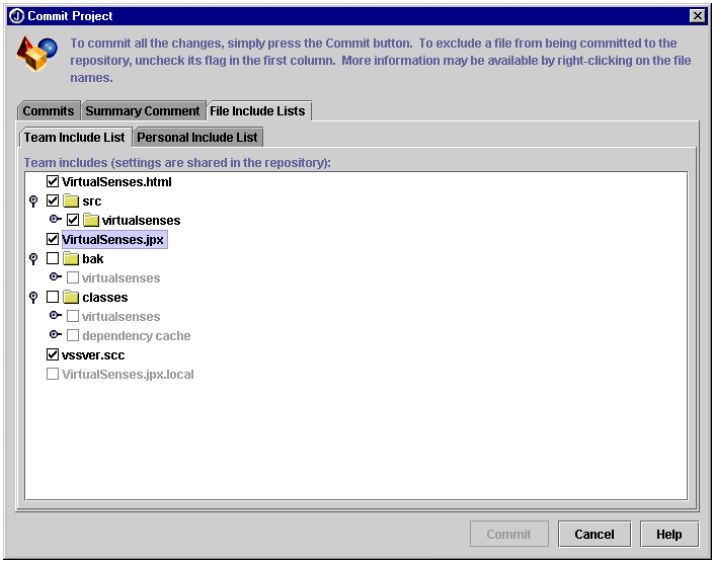
Complete Diff 탭은 이 통합에서 지원되지 않습니다. Visual SourceSafe는 둘 이상의 사람이 동시에 하나의 파일을 사용하지 못하도록 하기 때문입니다.

Individual Comment 창 하단에 있는 Use Summary Comment 체크 박스를 사용하여 여러 개의 파일에 동일한 요약 주석을 추가할 수 있습니다. 이

요약 주석은 각각의 파일에 대해 개별적으로 작성된 주석과 함께 유지됩니다. Summary Comment 탭을 선택하면 요약 주석을 작성할 수 있습니다.



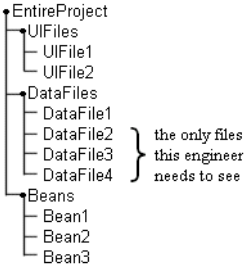
Committing Changes 브라우저의 File Include Lists는 체크 인에 포함될 파일을 결정합니다. 두 개의 페이지가 있습니다. Team Include List 및 Personal Include List.



Team Include List 파일들은 <projectname>.jpx 파일에서 추적합니다. 물론, 여기에 입력한 정보는 팀 수준의 프로젝트 설정이므로 해당 파일에 저장됩니다. 여기에서 선택 표시된 파일은 모든 사람이 사용할 수 있어야 합니다.

일반적으로 bak 파일과 <projectname>.local.jpx 파일은 제외됩니다(선택 표시되지 않습니다).파생된 파일과 같은 다른 파일들도 제외될 수 있습니다. 팀 프로젝트 체크인에서 포함시킬 파일과 제외할 파일에 관한 회사 정책을 확인하십시오.

Personal Include List 파일들은 이 테이블의 정보가 저장되어 있는 <projectname>.local.jpx 파일에 의해 추적됩니다. 이 목록은 개인적인 용도로 사용하기에 편리합니다. 프로젝트의 모든 파일을 사용할 필요는 없으므로 체크 인을 수행할 때마다 모든 파일을 살펴 볼 필요는 없습니다. 이 간단한 차트는 개인 개발자가 필요로 하는 파일과 비교하여 사용할 수 있는 파일의 개념을 보여 줍니다.

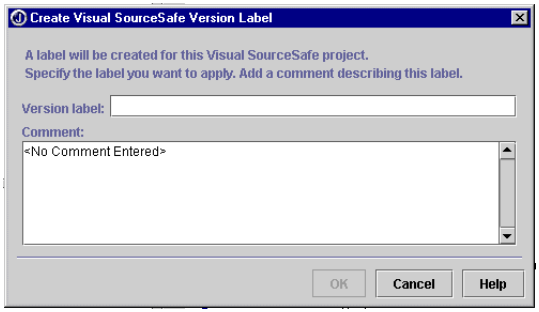


Personal Include List 페이지를 사용하면 사용자가 필요로 하는 파일만 표시할 수 있습니다. 나머지 파일은 숨김 상태로 유지되며 보고 싶을 때 다시 볼 수 있습니다.

Version Label 생성

버전 레이블 또는 태그를 사용하면 언제라도 전체 프로젝트의 스냅샷을 얻을 수 있습니다. 서로 다른 파일들이 다른 주기로 변경되므로 100개 파일로 구성된 프로젝트는 100개의 서로 다른 현재 수정 번호를 포함합니다. 버전 레이블은 개별 파일의 변경 내용은 참조하지 않고 전체 프로젝트의 진전을 표시합니다.

Team|Create Version Label을 선택하여 현재 프로젝트에 대한 버전 레이블을 생성합니다. Create Visual SourceSafe Version Label 대화 상자가 나타납니다.



관례에 따라 버전 레이블의 이름을 지정합니다.레이블의 용도에 대한 설명을 작성합니다.완료되면 OK를 클릭합니다.JBuilder는 데이터베이스에 상주하는 프로젝트의 모든 파일에 이 레이블을 적용합니다.

Chapter

5

JBuilder에서의 Rational ClearCase

이것은 JBuilder 기업용 버전의 기능입니다.

JBuilder에 통합된 Rational ClearCase를 통해 개발 환경 내에서 가장 일반적인 버전 제어 작업을 수행할 수 있습니다. JBuilder에서는 중첩 디렉토리를 추가하는 등의 작업을 쉽게 할 수 있습니다. JBuilder는 사용자가 선택한 하위 디렉토리와 파일을 재귀적으로 추가하므로 각각을 개별적으로 추가할 필요가 없습니다.

JBuilder는 Base ClearCase와 동적 뷰를 지원하므로다른 ClearCase 간에 최상의 호환성을 제공하면서 대부분의 기능을 사용할 수 있습니다.

JBuilder에의 ClearCase 통합은 ClearCase 4.1 버전용으로 설계, 테스트되었습니다. 여기서의 모든 설명은 사용자의 시스템에 호환 가능한 ClearCase 클라이언트가 설치, 구성되어 있으며 사용자가 호환 가능한 ClearCase 서버에 적절한 액세스 권한을 가지고 있다고 가정합니다. ClearCase 설치에 대한 문의는 ClearCase 관리자에게 하시기 바랍니다.

이 설명서에서는 JBuilder에 통합된 ClearCase를 설명합니다. ClearCase 자체에 대한 설명서는 ClearCase의 온라인 도움말을 참조하거나 관리자에게 문의하십시오.

JBuilder에 통합된 ClearCase는 Windows, Solaris, Linux에서 사용할 수 있습니다.

Team 메뉴에서 사용 가능한 ClearCase 명령은 다음과 같습니다.

- Select Project VCS
- Configure ClearCase (현재 구성의 읽기 전용 뷰를 표시합니다)
- ClearCase Tool
- Check In
- Check Out
- Undo Checkout
- Add File
- Place Project Into ClearCase
- Edit/Add Views
- Add Version Label
- Pull Project From ClearCase

다른 명령들은 Team|ClearCase Tool을 선택하여 ClearCase를 호출한 후 ClearCase 내에서 실행해야 합니다.

참고 명령은 사용할 수 있을 때만 활성화됩니다.

Linux 사용자의 참고 사항

ClearCase 통합은 RedHat Linux 6.2 상에서 테스트되었습니다. 다음과 같은 방법으로 Linux에서 ClearCase를 사용합니다.

- Rational이 제공한 커널 패치를 설치하고 커널을 다시 컴파일하여 Linux에 ClearCase를 설치합니다.
- `PATH`에 `cleartool` 명령을 넣어 JBuilder에서의 ClearCase 지원을 활성화시킵니다.

Linux 클라이언트 지원은 ClearCase v. 4.1에 새롭게 추가된 기능이며 이전 버전은 서버만 지원합니다.

Rational ClearCase 사용

여기에서는 사용자의 시스템에 ClearCase 클라이언트가 설치되어 있으며 ClearCase 서버에 액세스할 수 있다고 가정합니다. ClearCase 설치 방법에 대해서는 ClearCase 관리자에게 문의하십시오.

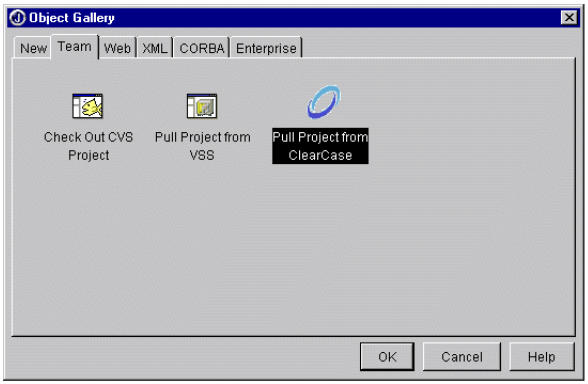
VOB 마운팅

JBuilder에서 VOB를 마운트하는 방법에는 객체 갤러리 또는 Team 메뉴를 이용하는두 가지 방법이 있습니다(이것은 객체 갤러리에서 사용할 수 있는 유일한 ClearCase 명령입니다).

객체 갤러리에서 다음을 수행합니다.

1. File|New를 선택합니다. 객체 갤러리가 나타납니다.
2. Team 탭을 선택합니다.

3 Pull Project From ClearCase를 선택합니다.

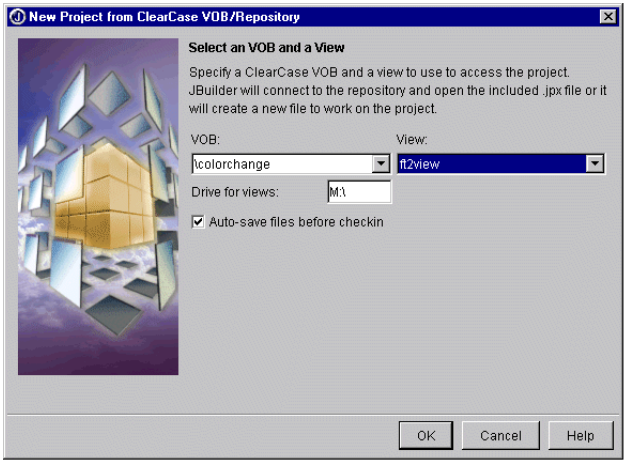


4 OK를 클릭하거나 ***Enter***를 누릅니다.

Team 메뉴에서 다음을 수행합니다.

1 Team|Pull Project From ClearCase를 선택합니다.

두 방법 모두 New Project From ClearCase VOB/Repository 마법사에 액세스합니다. 이 마법사는 사용 가능한 VOB와 뷰의 드롭 다운 목록을 제공합니다.



마운트하려는 VOB를 선택하고, 시작하려는 뷰를 입력하거나 선택하고, 자동 저장 옵션을 사용할 것인지 여부를 결정하고난 다음 OK를 클릭하거나 ***Enter***를 누릅니다.

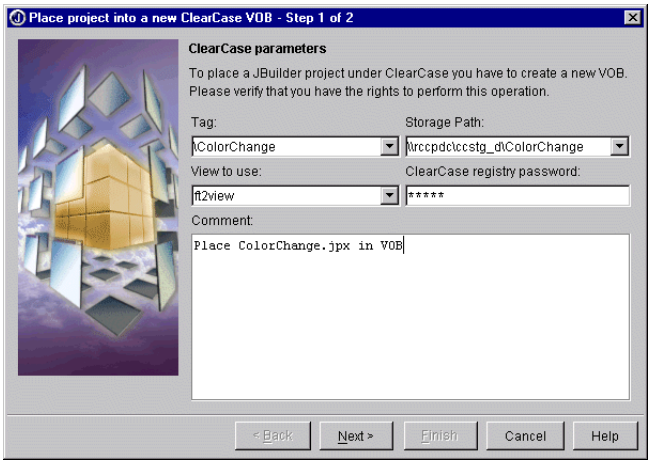
JBuilder는 .jpx 프로젝트 파일에 연결되어 있어서 이 파일이 참조하는 파일들을 사용자의 뷰에 보여 줍니다. 연결된 .jpx 파일이 없으면 JBuilder는 새로 하나 만든 후 선택한 VOB에 있는 파일들을 새로운 JBuilder 프로젝트에 넣습니다.

JBuilder는 사용한 각 뷰를 기억하여 나중에 뷰의 드롭다운 목록에 추가합니다.

Place Project Into ClearCase:새 VOB 생성

여기에서는 사용자가 VOB를 생성하는 데 필요한 ClearCase 액세스 권한을 가지고 있다고 가정합니다.

새 프로젝트를 생성하고 버전 제어 시스템으로 ClearCase를 선택한 다음에는 Team|Place Project Into ClearCase를 선택하여 프로젝트의 새 VOB를 생성합니다. Place Project 마법사가 나타납니다.



필드는 다음과 같습니다.

- Tag:새 VOB의 이름을 입력합니다. 이 필드는 ClearCase가 사용자의 클라이언트에 있는 프로젝트를 참조하는 방법을 나타냅니다.
- Storage Path:서버에 있는 새 VOB의 물리적 위치를 나타내는 경로를 입력하거나 선택합니다.
- View:새 VOB에 액세스하는 데 사용할 기존의 뷰 이름을 입력하거나 선택합니다.
- ClearCase Registry Password:암호를 입력합니다. 액세스 권한이 인증되면 JBuilder는 계속 진행합니다. 그렇지 않은 경우 ClearCase 관리자에게 문의하십시오.
- Comment:자신이 하고 있는 작업에 대한 설명을 입력합니다(귀사의 가이드 라인을 따르십시오).

마법사의 두 번째 단계는 VOB 생성 프로세스에 대한 피드백을 제공합니다.

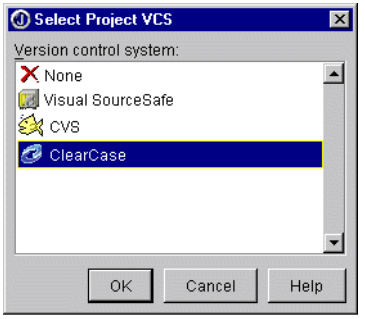
JBuilder는 원래 작업 영역을 보존하므로새로 생성한 VOB에 파일을 이동하지 않고 복사합니다. 프로젝트의 모든 파일(클래스, 백업 파일 및 `.jpx.local`로 끝나는 파일은 제외)이 VOB로 복사됩니다.

ClearCase 명령 액세스

ClearCase 명령은 메인 메뉴의 Team 메뉴와 프로젝트 창의 컨텍스트 메뉴에서 사용할 수 있습니다.

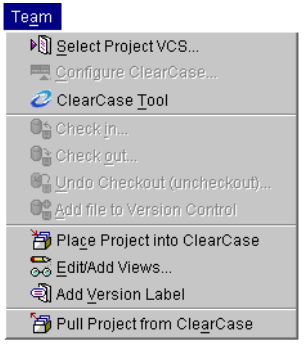
다음과 같은 방법으로 JBuilder IDE에서 ClearCase 명령에 액세스합니다.

- 메인 메뉴에서 Team을 선택합니다.
- Select Project VCS를 선택합니다.
- Select Project VCS 대화 상자에서 ClearCase를 선택합니다.



- 대화 상자를 닫으려면 OK를 클릭합니다.

Team 메뉴에 ClearCase 명령이 반영되어 나타납니다. 각 명령은 실행 가능할 때 활성화됩니다. JBuilder 내에서 그 밖의 다른 ClearCase 명령을 실행할 수 있기 전에 VOB에 액세스해야 합니다.



Team 메뉴는 컨텐트 창에 활성화되어 있는 파일에 대해 파일 레벨 명령을 실행합니다. 이 메뉴는 현재 프로젝트에 대해 프로젝트 레벨 명령을 실행합니다.

프로젝트 창의 컨텍스트 메뉴(마우스 오른쪽 버튼을 클릭하면 나타나는 메뉴)에서 Check In, Check Out, Undo Checkout, Add 명령을 사용할 수 있습니다. JBuilder는 프로젝트 창의 다중 선택을 지원합니다. 따라서 이러한 명령을 여러 파일에 한꺼번에 적용할 수 있습니다.

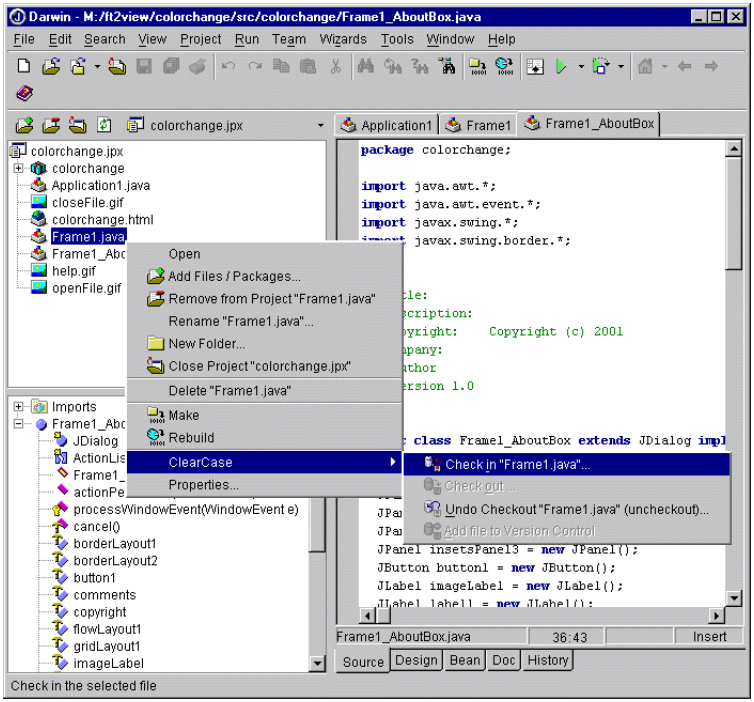
다음과 같은 방법으로 이러한 명령에 액세스합니다.

1 명령을 사용할 파일을 하나 이상 선택합니다.

2 선택한 노드 중 하나에서 오른쪽 마우스 버튼을 클릭합니다.

3 컨텍스트 메뉴에서 ClearCase를 선택합니다.

4 ClearCase 하위 명령에서 사용하려는 명령을 선택합니다.



프로젝트 창에 있는 임의의 파일에 대해 파일 레벨 ClearCase 명령을 사용할 수 있습니다.

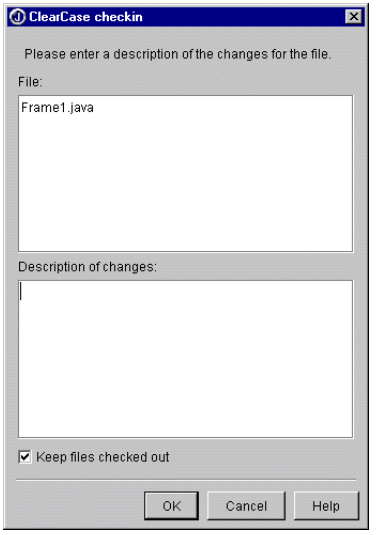
파일 레벨 ClearCase 명령

이 명령들은 모두 파일 레벨 이벤트입니다. 대부분의 명령은 명령에 액세스하는 방법에 따라 하나 또는 여러 개의 파일에 영향을 미칠 수 있습니다.

Check In

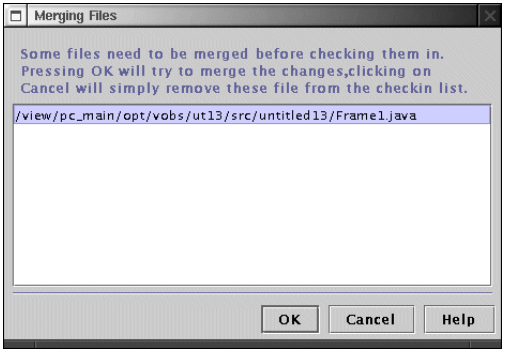
Check In 명령은 Team 메뉴와 프로젝트 창에서의 컨텍스트 메뉴를 통해서 액세스할 수 있습니다. 두 방법 모두 하나의 파일을 체크 인합니다. 여러 개의 파일을 체크 인하려면 프로젝트 창에서 파일들을 선택한 다음 컨텍스트 메뉴를 사용하여 선택한 모든 파일에 한 번에 명령을 적용합니다.

Check In 명령을 적용하면 ClearCase Check 대화 상자가 나타납니다.



Checkin 마법사는 선택된 파일을 나열하고 변경 사항에 대한 요약 설명 입력을 요청합니다. 해당 파일들을 계속 사용할 수 있도록 파일을 체크 인 후에 체크 아웃된 상태로 유지할 것인지 여부를 묻습니다.

Check In 명령이 사용되는 경우 JBuilder는 해당 파일들을 체크 인하기 전에 업데이트된 것이 있는지 찾습니다. 다른 사람에 의해 변경된 파일이 있으면 Merging Files 대화 상자에 나타납니다.



병합하려면 OK를 클릭하고, 해당 파일을 체크 인하지 않고 파일의 로컬 버전을 유지하려면 Cancel을 클릭합니다.

OK를 클릭하면 JBuilder는 파일들을 병합합니다. 이 작업은 대개 간단하며 각 파일은 단순히 하나의 통합된 버전으로 병합되어서 체크 인됩니다.

파일의 동일한 물리적 부분에 대해 각 버전의 내용에 차이가 있을 경우 병합 충돌로서 등록합니다. JBuilder는 다른 대화 상자에 충돌이 일어난 파일들을 표시하고 체크 인하지 않습니다. 병합 충돌은 ClearCase 내에서 해결되어야 합니다.

Check Out

파일을 체크 아웃하면 체크 아웃한 사람이 해당 파일에 쓸 수 있게 됩니다. 한번에 둘 이상의 사용자가 동일한 파일을 체크 아웃할 수 있습니다.

JBuilder에서는 Team 메뉴와 프로젝트 창의 컨텍스트 메뉴에서 Check Out 명령을 사용할 수 있습니다. .JBuilder는 체크 아웃되는 파일의 부모 디렉토리를 자동으로 체크 아웃합니다.

체크 아웃에 대한 대화 상자는 없습니다. 체크 아웃은 순수한 메뉴 방식 명령입니다.

Undo Checkout

Undo Checkout은 쓰기 액세스를 취소하고 체크 아웃된 후의 파일에 변경된 사항을 무시합니다. JBuilder에서는 Team 메뉴나 프로젝트 창에서의 컨텍스트 메뉴에서 Undo Checkout 명령을 사용할 수 있습니다. .

Add file

작업을 하다 보면 새 파일을 만들 일이 종종 생깁니다. .어떤 파일은 새 디렉토리 트리의 일부가 될 수도 있습니다. 이러한 파일 시스템 컴포넌트를 ClearCase 뷰에서 생성하면 뷰 전용 요소가 되므로 공유할 수 있도록 VOB에 추가하여야 합니다. JBuilder에서는 Team 메뉴나 프로젝트 창에서의 컨텍스트 메뉴에서 Add 명령을 사용할 수 있습니다. .

파일 요소는 디렉토리 트리가 ClearCase 아래 있을 경우에만 추가될 수 있습니다. 따라서 JBuilder는 사용자가 ClearCase 밑에 있지 않은 디렉토리에 파일을 추가하면 트리를 탐색하여 ClearCase 밑에 있는 디렉토리를 찾아낸 후, 그 디렉토리를 부모 디렉토리로 하여 새 디렉토리 구조를 추가한 다음 사용자가 선택한 파일을 트리의 적절한 위치에 추가합니다.

프로젝트 레벨 ClearCase 명령

이 명령들은 전체 프로젝트에 작용합니다.

태깅

태그 또는 버전 레이블을 통해 언제라도 전체 프로젝트의 스냅샷을 얻을 수 있습니다. 서로 다른 파일들이 다른 주기로 변경되므로 100개 파일로 구성된 프로젝트는 100개의 서로 다른 현재 수정 번호를 포함할 수 있습니다. 각 파일의 변경 사항을 참조하지 않고도 태그를 통해 전체 프로젝트의 진행 상황을 알 수 있습니다.

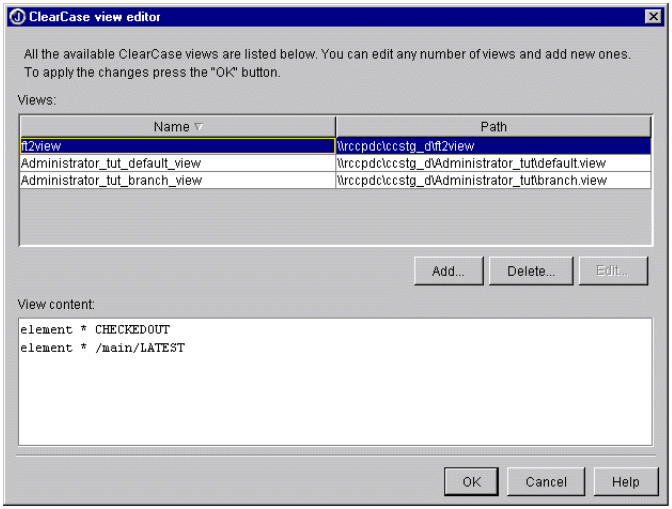
JBuilder는 태그를 신청하고 생성하는 두 단계를 합쳐서 하나의 가시적인 단계로 만듭니다. JBuilder는 적절한 버전 레이블과 해당 설명을 사용자에게 입력 받습니다. JBuilder는 그런 다음 레이블을 신청하고 생성한 후 JBuilder 프로젝트를 위한 ClearCase VOB에 있는 모든 파일에 적용합니다.

ClearCase 구성 보기

Team|Configure ClearCase 옵션은 `cleartool -version` 명령에 의해 반환된 것처럼 ClearCase의 구성을 보여 줍니다.

Edit/Add Views

Team|Edit/Edit/Add Views를 선택하여 새 뷰를 추가하거나 기존의 뷰를 편집하는 ClearCase View Editor를 호출합니다.



기존 뷰 편집

JBuilder는 IDE 내에서 모든 뷰에 대한 구성 스펙을 편집할 수 있게 합니다. 이를 통해 기본 액세스 수준(체크 아웃된 또는 최근의), 트리에서 사용할 기본 가지(branch) 등을 변경할 수 있습니다.

다음과 같은 방법으로 기존 뷰의 구성 스펙을 편집합니다.

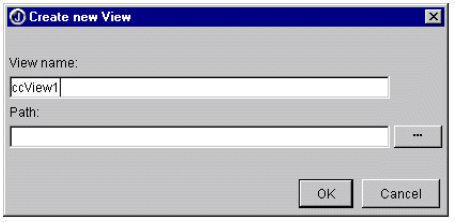
1. View Editor의 상단부에 있는 테이블에서 뷰를 선택합니다. 구성 스펙이 밑에 있는 편집 가능한 창에 나타납니다.
2. 편집할 수 있는 창에 직접 입력하여 뷰를 수정합니다.
3. 완료되면 OK를 클릭합니다.

뷰의 편집을 취소하려면 Cancel을 클릭합니다. JBuilder는 모든 변경 사항을 무시하며 뷰는 변경되지 않습니다.

새로운 뷰 추가

다음과 같은 방법으로 새로운 뷰를 목록에 추가합니다.

1. 대화 상자 중간에 있는 Add 버튼을 클릭합니다. Create New View 대화 상자가 나타납니다.



2. 새로운 뷰의 이름을 입력합니다.
3. 새로운 뷰가 위치할 경로를 선택합니다. 경로를 입력하거나 생략 버튼을 클릭하여 찾습니다.
4. OK를 클릭하거나 ***Enter***를 눌러 뷰를 생성하고 대화 상자를 닫습니다.

새로운 뷰 생성 작업을 취소하려면 Cancel을 클릭합니다. 그러면 JBuilder는 새로운 뷰 생성 작업을 완전히 무시합니다.

View Editor에 있는 Edit 버튼을 사용하여 방금 추가한 뷰의 경로를 변경할 수 있습니다.

참고 일단 View Editor에서 OK를 클릭하고 대화 상자를 닫으면, 새로운 뷰의 경로는 더 이상 편집할 수 없습니다.

Chapter

6

버전 제어 참조

일반적으로 이러한 리소스들은 버전 관리에 대해 좀더 자세한 정보를 제공하며 JBuilder가 지원하는 다양한 버전 제어 툴을 사용하여 버전 관리를 도와 줍니다.

유용한 JBuilder와 Borland 링크 및 뉴스 그룹에 대한 자세한 정보는 1-2 페이지의 "Borland 개발자 지원 문의"를 참조하십시오.

일반적인 개정 관리 리소스

툴 비교:

- CM 제품 안내: http://www.cmtoday.com/yp/configuration_management.html

뉴스 그룹: comp.software.config-mgmt

본 뉴스 그룹의 FAQ는 http://www.iac.honeywell.com/Pub/Tech/CM/index.html로 하이퍼링크 됩니다.

지원 툴에 관한 리소스

CVS

뉴스그룹 목록: http://www.cvshome.org/communication.html

CVS 홈: http://www.cvshome.org/

CVS 설명서: http://www.cvshome.org/docs/manual/index.html

사용자 정보: http://loria.fr/~molli/fom-serve/cache/l.html

특별히 고려할 사항:

- 다양한 플랫폼을 위한 CVS는 http://www.cvshome.org/dev/codes.html에 있습니다.
- 윈도우 사용자 정보는 http://www.computas.com/pub/wincvs-howto/에 있습니다.
- NT용 CVS는 http://www.cvsnt.org/ (홈 페이지)와 http://www.devguy.com/fp/cfgmgmt/cvs/cvs_admin_nt.html (사용자 정보)에 있습니다.
- Mac용 CVS는 (JBuilder의 기능 확장에 대한 내용은 OpenTools 설명서를 참조하십시오.): http://www.maccvs.org/ (홈 페이지)에 있습니다.

Rational ClearCase

뉴스그룹 목록: http://www.bomis.com/ring_home.fcgi?ring=Mconfiguration-management-computers

홈 페이지: http://www.rational.com/products/clearcase/index.jsp

비주얼(Visual) SourceSafe

홈 페이지: http://msdn.microsoft.com/ssafe/

특별히 고려할 사항:

- UNIX용은(JBuilder의 기능 확장에 대한 내용은 OpenTools 설명서를 참조하십시오.) http://www.mainsoft.com/products/visualsourcesafe.html에 있습니다.
- Mac용은(JBuilder의 기능 확장에 대한 내용은 OpenTools 설명서를 참조하십시오.): http://www.metrowerks.com/desktop/mwvss/?faq에 있습니다.

Chapter

7

자습서: JBuilder의 CVS

다음은 JBuilder Enterprise 에디션의 기능들입니다.

이 자습서에서는 대부분의 JBuilder CVS 명령 사용에 대해 설명합니다. JBuilder를 설치했으면 이 자습서를 수행하기 전에 어떤 것도 다시 설정할 필요가 없습니다.

이 단원을 위해 작업할 로컬 CVS 리포지토리를 만듭니다. 리포지토리는 로컬로 만들어지므로 액세스하여 이 자습서의 모든 단계를 완료해야 합니다.

프로젝트를 만들어 모듈에 넣고 두 개의 다른 디렉토리로 모듈을 확인합니다. 두 개의 디렉토리에서 같은 프로젝트 모듈을 처리합니다. 두 개의 다른 디렉토리는 각각 `dir1`과 `dir2`라고 합니다. `dir1`의 모듈을 사용하면 User One이라는 사용자 한 명의 역할을 하게 됩니다. `dir2`의 모듈을 사용하면 User Two라는 다른 사용자 역할을 하게 됩니다. 이러한 방식으로 사용자가 여러 명인 개발 환경을 복제할 수 있습니다. CVS 통합 내에서 가능한 많은 기능을 표시합니다.

JBuilder는 내용 창의 기록 보기에 확장된 차이 처리 기능을 제공합니다. 기록 보기 기능에 대해 자세히 알아보려면 *JBuilder로 애플리케이션 구축*의 "파일 및 버전 비교"를 참조하십시오.

1 단계: 설정

이 단계에서는 JBuilder 프로젝트를 만들고 새 프로젝트에 CVS를 사용한다는 사실을 JBuilder에 알려줍니다. 이렇게 하면 JBuilder의 CVS를 사용할 때 필요한 모든 사항을 제공하게 됩니다.

작업할 JBuilder 프로젝트가 필요합니다. 새 프로젝트를 만듭니다.

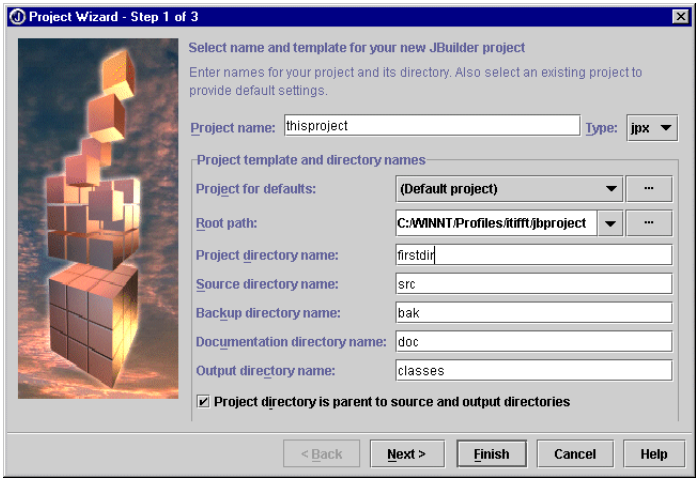
1. File|New Project를 선택합니다. Project 마법사가 나타납니다.
2. Project 마법사 1 단계에서 프로젝트 이름을 `thisproject`로 설정합니다. 프로젝트 파일 형식은 `.jpx`를 사용합니다.

   기본 설정된 프로젝트가 있으면 그 프로젝트를 선택할 수도 있습니다.

3 프로젝트 디렉토리 이름은 `dir1`로 지정합니다.

당분간 다른 기본값을 사용합니다.

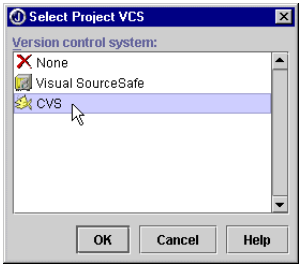
1 단계는 다음과 같이 나타납니다.



4 1 단계에서 Finish를 클릭하여 프로젝트를 만듭니다.

이 프로젝트에 사용할 버전 제어 시스템을 JBuilder에 알려주어야 합니다.

1 Team|Select Project VCS를 선택합니다. Select Project VCS 대화 상자가 나타납니다.



2 목록에서 CVS를 선택합니다.

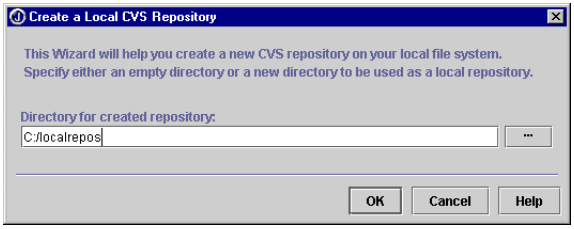
3 OK를 클릭하거나 ***Enter***를 눌러 대화 상자를 닫고 IDE로 돌아갑니다.

Team 메뉴를 선택하고 메뉴가 확장되었는지 봅니다. 메뉴의 일부 항목은 사용할 수 있습니다.

2 단계: 리포지토리 및 모듈 만들기

이제 이 자습서의 나머지 부분에 사용할 로컬 리포지토리를 만듭니다.

1 Team|Create Local Repository를 선택합니다. Create A Local CVS Repository 마법사가 나타납니다.



2 기본 설정된 연습에 따라 루트나 루트에 가까운 위치에 `localrepos`를 입력합니다.

3 OK를 클릭하거나 ***Enter***를 누릅니다.

JBuilder는 `localrepos`라는 이름으로 리포지토리를 만들고 확인 대화 상자로 돌아갑니다.

4 OK를 클릭하거나 ***Enter***를 눌러 확인 대화 상자를 닫고 IDE로 돌아갑니다.

이제 CVS를 사용할 수 있습니다. 먼저 작업할 모듈을 만들어야 합니다. JBuilder로 모듈을 만드는 작업은 CVS에 대한 연결을 구성하고 CVS 모듈을 만들어 리포지토리로 넣는 작업과 관련됩니다. 모듈이 리포지토리에 있으면 모든 내용을 CVS가 제어하고 이 프로젝트에 더 많은 명령을 사용할 수 있습니다.

1 Team|Place Project Into CVS를 선택합니다. Place Project Into CVS 마법사가 나타납니다. 이 마법사의 첫 단계인 CVS Module Properties 단계에서는 CVS에 대한 이 프로젝트의 연결을 구성합니다.

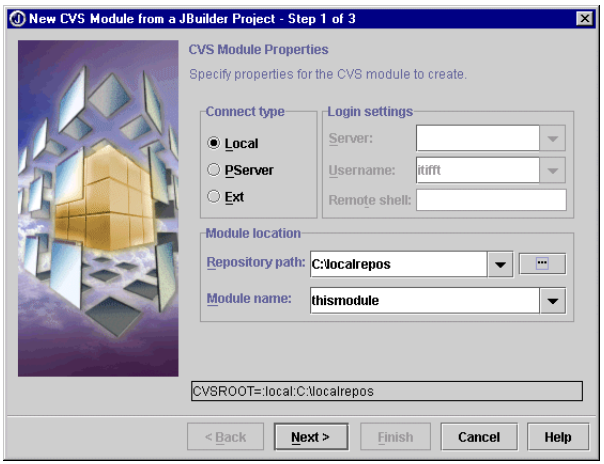
2 Connect Type에서 Local을 선택합니다. 계속하기 위해 기계에 로그인할 필요가 없으므로 Login Settings는 회색으로 표시됩니다.

3 Repository Path에서 직접 경로를 입력하거나 사용자가 만든 `localrepos` 리포지토리를 탐색합니다.

연결 형식이 Local이기 때문에 경로를 입력하지 않고 리포지토리를 탐색할 수 있는 생략 버튼을 사용할 수도 있습니다.

4 모듈 이름은 `thismodule`로 지정합니다.

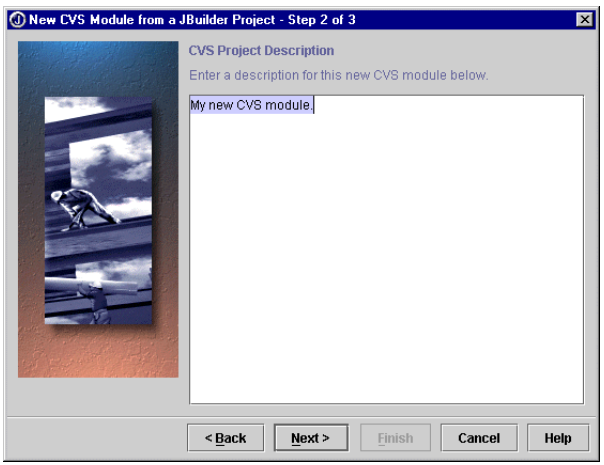
작업을 완료하면 설정이 다음과 같이 나타납니다.



상자의 맨 아래에는 CVSROOT라는 텍스트가 표시됩니다. 계속하기 전에 마법사의 이 페이지에서 요소를 변경하면 CVSROOT가 어떻게 변경되어 일치되는지를 확인할 수 있습니다.

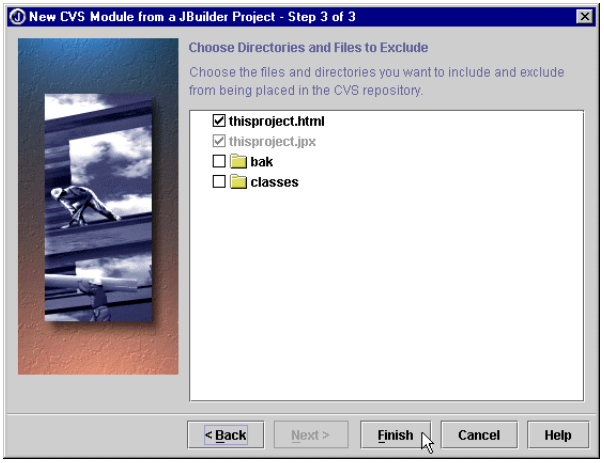
계속하려면 입력한 항목이 이러한 지시 사항과 일치하는지 확인해야 합니다.

5 Next를 클릭하여 2 단계, CVS Project Description 단계로 이동합니다.



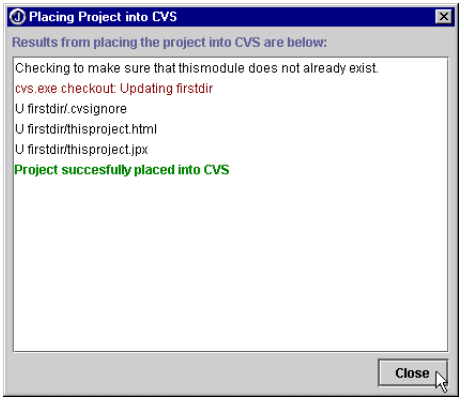
기본 설명을 사용합니다.

6 Next를 클릭하여 3 단계, Choose Directories And Files To Include 단계로 이동합니다.



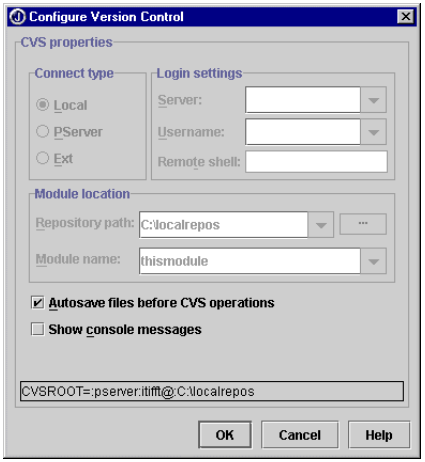
백업 파일(`bak`)과 파생된 파일(`class`)이 있는 디렉토리가 리포지토리에서 자동으로 제외된다는 점에 유의하십시오.

7 기본값을 사용하고 Finish를 클릭합니다. Place Project Into CVS 피드백 대화 상자가 나타납니다.



이 피드백 대화 상자는 돌아올 때 CVS로부터 `stdout` 출력을 표시하고 작업이 성공적으로 완료되는 시기를 알려줍니다.

Team|Configure CVS를 선택하여 프로젝트의 연결 구성을 볼 수 있습니다. 이 대화 상자의 필드는 대부분 읽기 전용이지만 자동 저장 및 콘솔 디스플레이 옵션을 선택할 수 있습니다.



3 단계: 파일 변경 사항 커밋

새 프로젝트용 CVS 모듈을 만들어졌습니다. 그 모듈은 리포지토리에 있습니다. 이제 파일을 추가하고 변경한 다음 커밋합니다. 따라서 작업은 리포지토리에 저장되고 리포지토리에서 버전 제어에 종속되며 다른 사용자가 사용할 수 있습니다.

이 자습서에서 사용된 변경 사항은 매우 간단하므로 코드가 아닌 프로세스에 초점을 맞출 수 있습니다.

먼저 작업할 파일 그룹을 만듭니다.

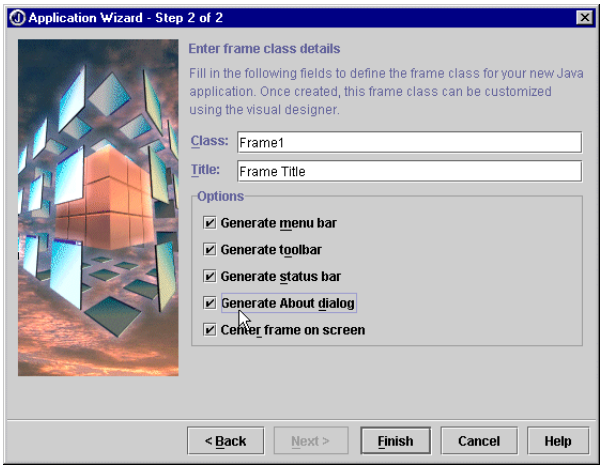
1 File|New를 선택합니다.

2 객체 갤러리의 New 페이지에서 Application을 선택합니다.

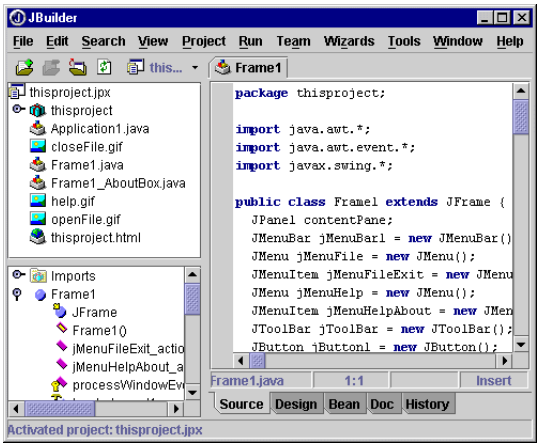
3 OK를 클릭하거나 ***Enter***를 눌러 Application 마법사를 시작합니다.

4 1 단계에서 모든 기본값을 사용하고 Next를 클릭합니다.

5 2 단계의 Options 영역에 있는 모든 체크 박스를 선택합니다.



6 Finish를 클릭하여 마법사를 닫습니다. 여섯 개의 새 파일과 `thisproject` 패키지를 만들고 지금 채운 IDE로 돌아갑니다.



이제 이 자습서의 나머지 부분에서 사용할 많은 원시 정보를 얻었습니다.

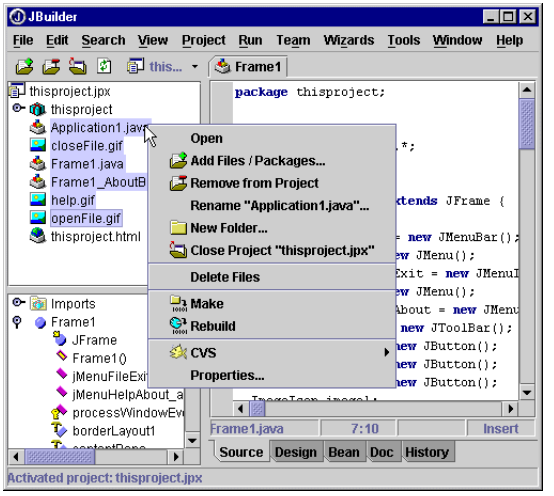
CVS에 새 파일을 추가합니다.

1 프로젝트 창에서 `Application1.java`를 선택합니다.

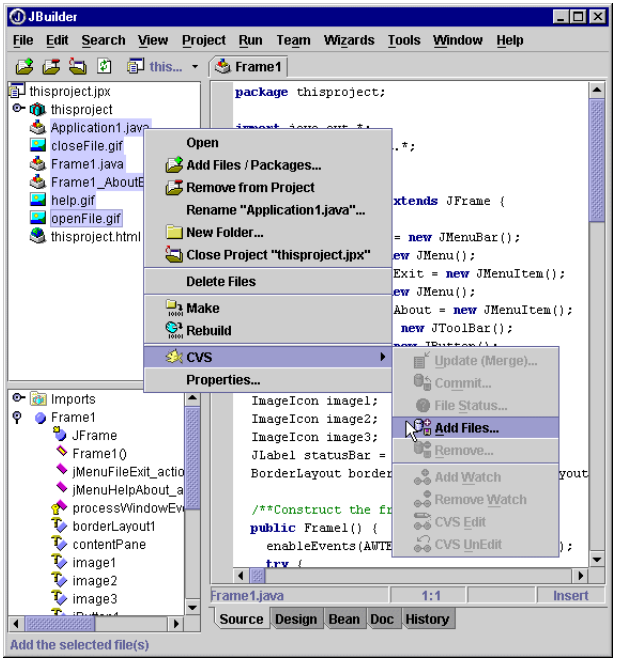
2 ***Shift*** 키를 누른 채 `openfile.gif`를 클릭합니다. 두 파일 사이에 있는 모든 파일이 선택됩니다.

실수로 HTML 파일을 선택했으면 다시 ***Shift*** 키를 누른 채 프로젝트 창에서 파일을 한 번 클릭합니다. 이렇게 하면 HTML 파일 선택은 해제되지만 다른 파일은 선택된 상태로 유지됩니다.

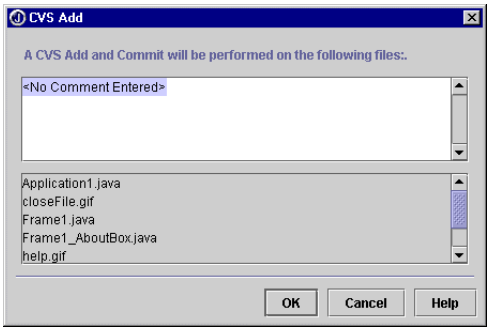
3 선택한 모든 파일을 마우스 오른쪽 버튼으로 클릭합니다. 상황에 맞는 메뉴가 나타나고 해당 명령은 선택한 모든 파일에 적용됩니다.



4 메뉴에서 CVS를 선택합니다. 하위 메뉴에서 사용할 수 있는 유일한 명령은 Add Files입니다.



5 Add Files를 선택합니다. CVS Add 대화 상자가 나타납니다.



6 `Adding default files from the Application wizard`라는 주석을 입력합니다.

7 OK를 클릭합니다.

JBuilder에 CVS에 대한 파일 추가를 확인하는 대화 상자가 나타납니다. 추가된 파일은 버전 제어를 받기 전에 커밋되어야 하므로 JBuilder는 CVS에 추가하는 즉시 자동으로 파일을 커밋합니다.

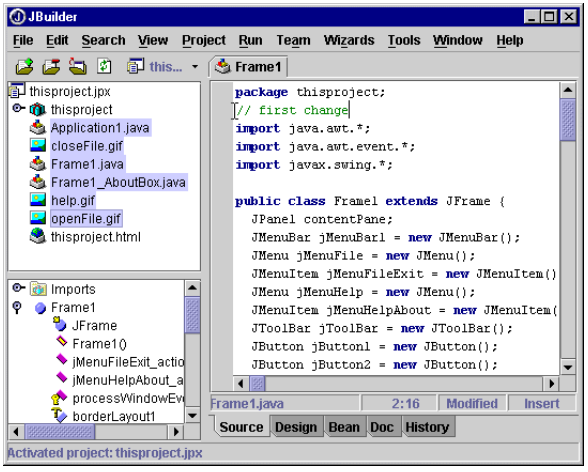
참고 CVS는 작업 영역 내에 있는 파일만 추가할 수 있습니다. 프로젝트 디렉토리 외부에 있는 파일을 추가하려면 프로젝트 디렉토리로 복사하고 디렉토리 내부에 있으면 JBuilder 프로젝트로 추가한 다음 CVS에 추가합니다.

8 OK를 클릭하거나 ***Enter***를 눌러 대화 상자를 닫고 IDE로 돌아갑니다.

파일을 변경하고 변경 사항을 커밋합니다. 변경 사항 커밋은 파일의 수정된 버전을 리포지토리에 포스트한다는 의미이므로 변경 사항은 다른 사용자가 사용할 수도 있습니다.

프로젝트 창에 이미 활성화되어 있는 `Frame1.java`를 변경합니다.

1 아래에 표시된 대로 코드의 첫 줄 아래에 `// first change`를 삽입합니다.



2 Team|Commit "Frame1.java"를 선택합니다.

3 CVS Commit 대화 상자에서 `Made first change` 를 입력합니다.

4 OK를 클릭하여 변경 사항을 커밋합니다.

JBuilder는 커밋 작업이 완료되는 시기를 알려줍니다.

5 OK를 눌러 대화 상자를 닫고 IDE로 돌아갑니다.

6 프로젝트 툴바의 Close Project 아이콘을 클릭하여 `thisproject`를 닫습니다.

JBuilder에 프로젝트 파일을 저장하라는 메시지가 표시됩니다. 그 파일은 CVS 연결을 구성할 때 수정되었습니다. `Frame1.java` 및 추가된 애플리케이션 파일은 목록에 없습니다. CVS 연결에서 자동 저장 기본값을 사용했기 때문입니다. 따라서 해당 파일은 CVS 명령을 수행할 때 이미 저장되었습니다.

7 확인된 상태의 기본값을 사용하고 OK를 클릭하거나 ***Enter***를 누릅니다.

4 단계: 기존 모듈 확인

이미 프로젝트용 모듈을 만들고 CVS에 파일을 추가하여 CVS의 파일을 처리했습니다.

모듈은 한 번 만들지만 프로젝트에서 작업하는 모든 사용자가 확인해야 합니다. 따라서 다음은 다른 사람이 만든 모듈을 사용하는 것처럼 기존 모듈을 확인하고 프로젝트를 변경하는 단계입니다.

객체 갤러리를 사용하면 CVS 리포지토리의 모듈을 빠르게 확인할 수 있습니다.

1 File|New를 선택합니다. 객체 갤러리가 나타납니다.

2 Team 탭을 선택합니다.

3 Check Out CVS Module을 선택합니다.

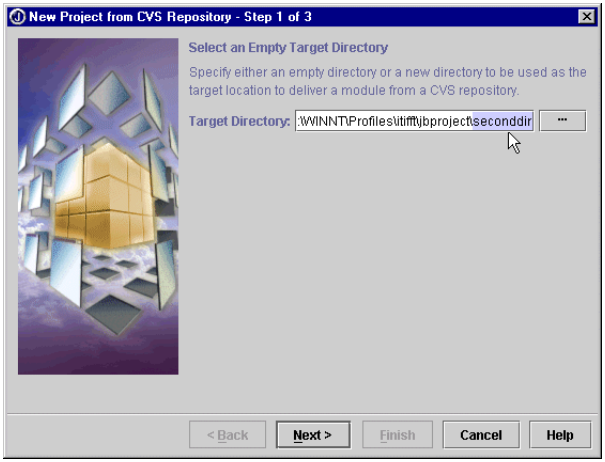
4 OK를 클릭하거나 ***Enter***를 누릅니다.

Check Out CVS Project 마법사가 시작됩니다.

이전에 만든 모듈을 확인합니다. 동일한 프로젝트에서 다른 개발자의 역할을 하므로 작업 영역으로 다른 디렉토리를 사용해야 합니다.

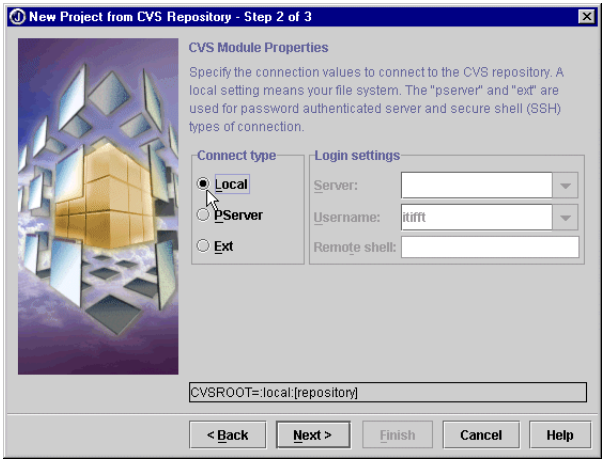
1 Team|Check Out CVS Project를 선택합니다. Check Out CVS Project 마법사가 나타납니다.

2 기본 경로는 `jbproject` 그대로 유지하지만 대상 디렉토리 이름은 `dir2`로 변경합니다.

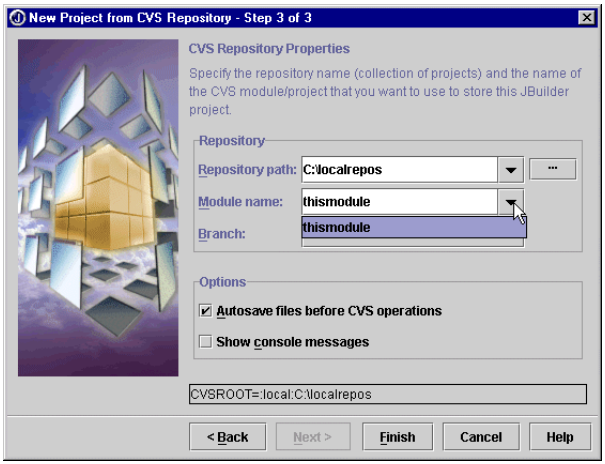


3 Next를 클릭하여 CVS Module Properties 단계로 이동합니다.

4 연결 형식을 Local로 설정합니다.



5 Next를 클릭하여 CVS Repository Properties 페이지로 이동합니다.



JBuilder는 사용하는 리포지토리, 모듈 및 브랜치(branch)를 추적하므로 새 모듈을 만들거나 확인할 때마다 입력하는 대신 해당 영역의 드롭다운 목록에서 선택할 수 있습니다.

기본 리포지토리 설정 및 Autosave Files Before CVS Operations를 사용합니다.

6 Finish를 클릭합니다. Checking Out CVS Project 피드백 대화 상자에서 `thisproject` 모듈을 `dir2`에 넣는 CVS 프로세스를 표시합니다.

7 작업을 완료하면 Close를 클릭하여 IDE로 돌아갑니다.

User Two는 User One으로 만든 모듈을 확인했습니다. 이제 모듈을 변경합니다.

팁 현재 있는 디렉토리와 디렉토리를 표시하는 창 제목 표시줄이 없는지 확인하려면 Project|Project Properties를 선택하고 Paths 페이지를 봅니다. Paths 페이지는 현재 프로젝트의 전체 경로를 표시합니다.

1 경로를 더블 클릭하여 프로젝트 창의 `thisproject` 패키지를 확장합니다.

2 프로젝트 창에서 `Frame1.java`를 더블 클릭합니다. 그러면 에디터에서 열립니다.

3 `// first change`를 `// second change`로 변경합니다.

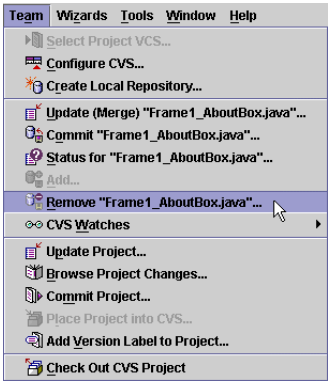
즉시 커밋하지 않아도 됩니다. 바쁘면 대개 커밋하지 않습니다.

프로젝트에서 파일을 제거합니다.

1 프로젝트 창에서 Frame1_AboutBox.java 파일을 더블 클릭하여 내용 창에서 엽니다.

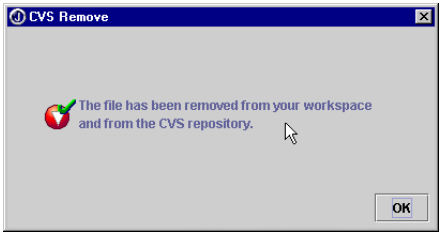
   파일이 내용 창에 있으면 Team 메뉴의 파일 수준 명령이 적용됩니다.

2 Team|CVS Remove "Frame1_AboutBox.java"를 선택합니다.



3 CVS Remove 대화 상자가 표시되고 주석을 입력하라는 메시지가 나타납니다. Removing Frame1_AboutBox.java를 입력합니다.

4 OK를 클릭합니다. 확인 메시지가 나타나면 읽어 봅니다.



   CVS에서 파일을 제거하는 프로세스는 세 단계로 구성됩니다. 작업 영역에서 파일을 제거하고 remove 명령을 적용한 다음 제거를 커밋해야 합니다.

   JBuilder는 이 메뉴 명령의 한 가지 명확한 단계에서 모든 작업을 수행합니다.

5 변경 사항을 커밋하면 OK를 클릭하거나 *Enter*를 누릅니다.

이전에 변경한 사항을 커밋합니다.

1 프로젝트 창에서 Frame1.java를 선택합니다.

2 마우스 오른쪽 버튼을 클릭하여 CVS|Commit "Frame1.java"를 선택합니다.

3 주석 영역에 Second change made 를 입력합니다.

4 OK를 클릭하여 파일을 커밋합니다.

5 파일을 커밋하면 OK를 클릭하거나 *Enter*를 누릅니다.

6 프로젝트를 닫습니다.

모듈을 넣기 위해 비어 있는 프로젝트로 만든 untitled 프로젝트를 닫습니다. 그 프로젝트는 목적을 달성했으며 더 이상 필요하지 않습니다.

5 단계: 프로젝트 업데이트

새 모듈을 만들고 기존 모듈을 확인했습니다. 그런 다음 리포지토리의 변경 사항을 적용하려면 User One으로 프로젝트를 업데이트합니다.

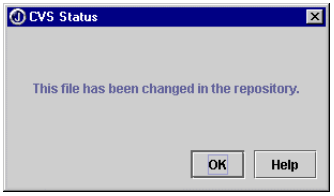
업데이트하면 다른 개발자가 사용 중인 파일에 변경한 사항으로 현재의 작업 영역이 유지됩니다. 이 자습서에서 다른 사람의 변경 사항을 표시할 수 있는 첫 번째 기회이므로 지금 수행해야 합니다.

먼저 CVS와 관련된 파일의 상태를 봅니다. 언제든지 볼 수 있습니다.

1 File|Reopen을 선택하고 /dir1/thisproject.jpx를 선택합니다.

2 내용 창에서 Frame1.java를 엽니다.

3 Team|Status For "Frame1.java"를 선택합니다.



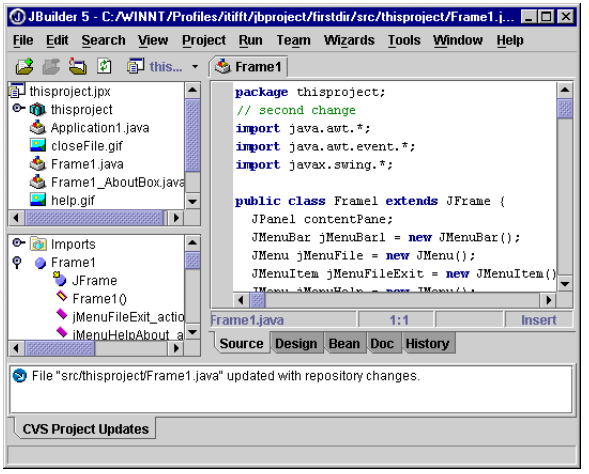
리포지토리에 있는 내용과 작업 영역을 동기화하려면 프로젝트를 업데이트합니다.

1 Team|Update Project를 선택합니다.

2 CVS Update 대화 상자에서 OK를 클릭하거나 *Enter*를 누릅니다.

3 OK를 클릭하거나 *Enter*를 눌러 AppBrowser로 돌아갑니다.

JBuilder는 메시지 창에서 프로젝트 업데이트를 제공하여, 이 업데이트로 작업 영역에서 변경된 사항을 정확하게 알려줍니다.



팁 각 커밋 전에 업데이트하는 것은 매우 좋은 버전 제어 연습입니다. CVS의 경우에는 충돌이 발생하기 전에 리포지토리 변경 사항을 작업 영역으로 병합할 수 있습니다. 그래도 충돌이 생길 경우 JBuilder에 있는 병합 충돌 관리 기능을 사용하면 됩니다.

모두 정리하고 이 자습서의 다음 단계로 넘어 갑니다.

1. 프로젝트를 닫습니다.
2. CVS Project Updates 콘솔 탭을 마우스 오른쪽 버튼으로 클릭하고 Remove "CVS Project Updates" Tab을 선택하여 탭과 메시지를 제거합니다.

6 단계: 프로젝트 커밋

실제 세션 중에 발생할 수 있는 상황을 모델링합니다.

사용자는 한 번에 여러 파일에서 작업하고, 많은 내용을 변경하며, 작업이 제대로 수행될 때는 그 흐름을 끊지 않고 CVS를 유지 관리하려고 합니다. 다른 사람이 같은 파일에서 동시에 작업하고 본인이 알지 못하는 내용을 변경하는 경우도 있습니다.

이 단계에서는 Browse Project 옵션을 사용할 수 있는 기회를 제공하여 새로운 방법으로 파일 버전을 보고 관리할 수 있도록 합니다.

먼저 User One이 같은 파일에서 작업하고 있는 동안 User Two가 수행할 수 있는 작업을 설정합니다.

1 File|Open Project를 선택하고 /dir2/thisproject.jpx를 탐색합니다.

2 Team|Update Project를 선택합니다.

프로젝트의 CVS 상태를 정기적으로 유지 관리해야 합니다. 작업 세션 전에 업데이트하는 것이 가장 좋습니다.

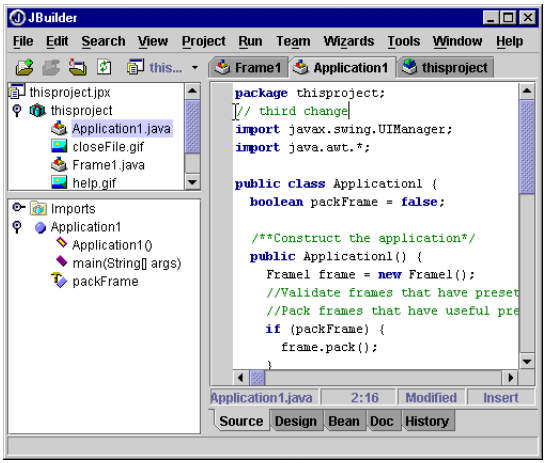
3 OK를 클릭합니다.

4 OK를 클릭합니다.

5 프로젝트 창에서 thisproject 패키지를 확장하여 파일을 표시합니다.

6 프로젝트 창의 Application1.java를 더블 클릭하여 내용 창에서 엽니다.

7 코드의 첫 줄 아래에 // third change를 입력합니다.



8 Team|Commit "Application1.java"를 선택합니다. 주석을 입력하라는 메시지가 나타나면 Third change made in this project를 입력합니다.

9 OK를 클릭합니다.

10 OK를 클릭하거나 *Enter*를 눌러 IDE로 돌아갑니다.

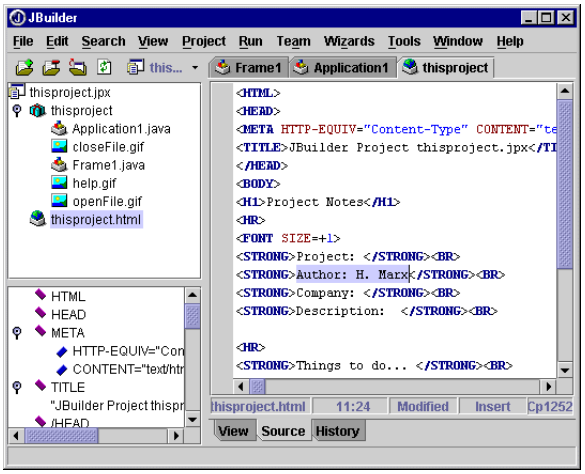
이번에는 HTML 파일에서 다른 내용을 변경합니다.

1 내용 창에서 thisproject.html을 열고 Source 파일 보기 탭을 선택합니다.

2 Author 항목을 찾습니다.

내용 창의 상태 표시줄에 행:열 표시법이 있습니다. Author 항목은 11 행에 있습니다.

3 작성자의 이름으로 H. Marx를 입력합니다.



4 Team|Update "thisproject.html"을 선택합니다.

5 Team|Commit "thisproject.html"을 선택합니다. 주석을 입력하라는 메시지가 나타나면 Fourth change: H. Marx as author를 입력합니다.

6 OK를 클릭한 다음 OK를 클릭하거나 ***Enter***를 누릅니다.

7 프로젝트를 닫습니다.

하나 이상을 변경합니다. 이렇게 변경하는 목적은 자습서의 뒷 부분에서 Workspace Diff 기능을 사용할 수 있도록 하기 위해서입니다.

1 File|Reopen을 선택하고 Frame1.java를 선택합니다.

2 열려 있는 프로젝트 파일의 6행에 // fifth change를 입력합니다. 상태 표시줄에서 코드의 정확한 위치를 찾아 봅니다.

3 파일을 저장하고 *커밋하지는 않습니다.* .

4 이 프로젝트를 닫습니다.

dir2로 다시 돌아가서 충돌의 나머지 반을 만듭니다. 이 디렉토리는 계속 열려 있어야 합니다.

1 User Two 버전의 Frame1.java를 엽니다.

2 Frame1.java에서 6행으로 이동하여 같은 6행에 // conflicts with the fifth change를 입력합니다.

3 Frame1.java 파일을 커밋합니다. Conflict created라는 주석을 입력합니다.

4 OK를 클릭하여 변경 사항을 커밋합니다.

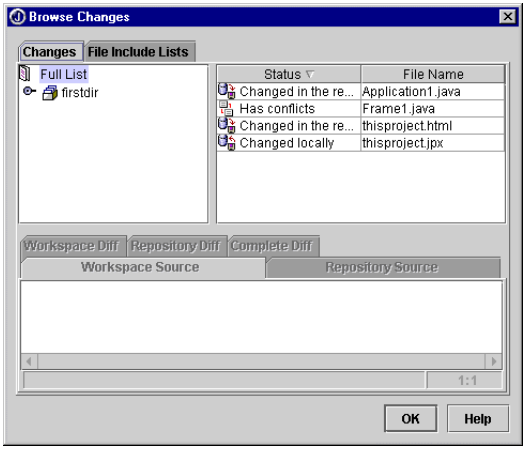
5 피드백 대화 상자에서 OK를 클릭하여 IDE로 돌아갑니다.

6 프로젝트를 닫습니다.

다시 User One이 되어 /dir1/thisproject.jpx를 보면 다음에 발생하는 상황을 볼 수 있습니다.

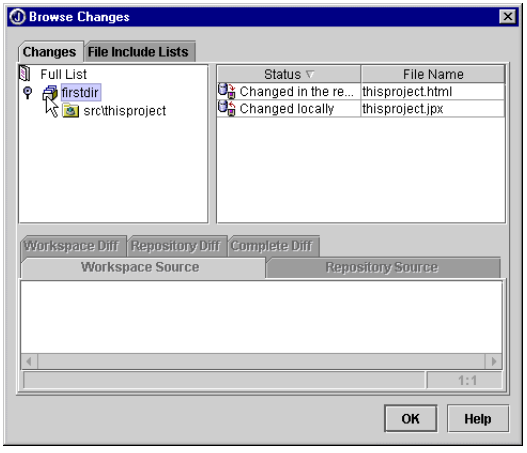
1 Team|Browse Project Changes를 선택합니다. Browse Changes 대화 상자가 나타납니다.

기본적으로 트리의 맨 위에 있는 Full List 노드가 선택되어 있습니다.



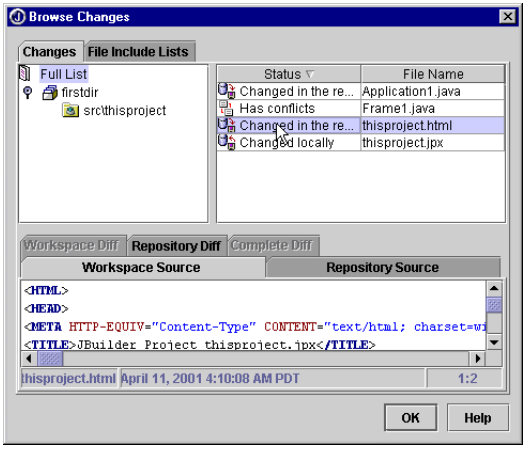
이 대화 상자는 변경된 파일만 표시합니다.

2 dir1 아이콘을 선택하고 확장합니다. 오른쪽에 있는 파일 목록이 어떻게 변경되는지에 유의하십시오.



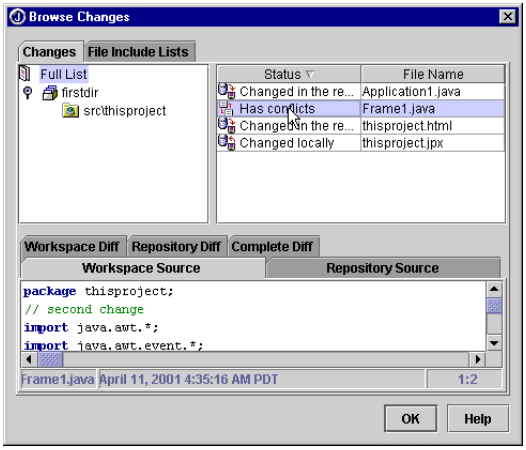
src\thisproject 아이콘을 선택하면 해당 파일은 사라지고 남은 파일이 목록에 나타납니다.

3 Full List를 다시 선택합니다. `thisproject.html`을 선택합니다. 아래에서 사용할 수 있는 파일 탭을 확인합니다.



*Workspace Source*는 버퍼에 있는 내용을 표시합니다. 버퍼는 디스크에 있는 가장 최신의 로컬 버전이며, 저장되지 않은 변경 사항을 포함합니다. *Repository Source*는 모듈의 코드를 표시합니다. *Repository Diff*는 작업 중인 리포지토리 버전과 리포지토리에 있는 가장 최신 버전 사이의 차이를 표시하여 파일에서 작업하는 동안 다른 사람이 변경한 내용을 볼 수 있도록 합니다.

4 충돌이 있는 `Frame1.java`를 선택합니다. 모든 파일 탭을 사용할 수 있습니다.



*Workspace Diff*는 시작한 작업 영역 버전과 작성한 가장 최신 버전 사이의 차이를 표시하여 이 작업 세션에서 변경한 내용을 모두 볼 수 있도록 합니다. *Complete Diff*는 선택한 파일의 최신 작업 영역 버전과 최신 리포지토리 버전 사이의 차이를 표시합니다.

5 각 탭에서 차이(코드의 같은 영역에 있는 서로 다른 텍스트 블록)를 봅니다.

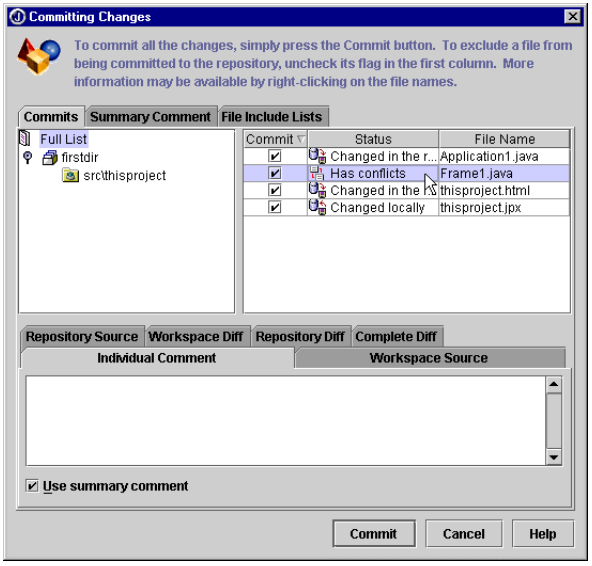
6 OK를 클릭하여 대화 상자를 닫습니다.

Browse Changes 브라우저에서 모든 CVS 명령을 실행하지는 않습니다. 성능이 뛰어난 보기 도구를 사용하면 파일의 상태에 대한 개요를 제공하고 변경된 각 파일의 소스 코드를 빨리 볼 수 있습니다. 개인 파일 및 공유 파일 목록만 변경됩니다.

다음에 Commit Project 브라우저를 봅니다. Commit Project 브라우저는 Browse Changes 브라우저와 같은 기능을 제공하고 주석 기능 및 커밋 기능도 제공합니다.

1 Team|Commit Project를 선택합니다. Committing Changes 대화 상자가 나타납니다.

2 파일 목록에서 `Frame1.java`를 선택합니다.

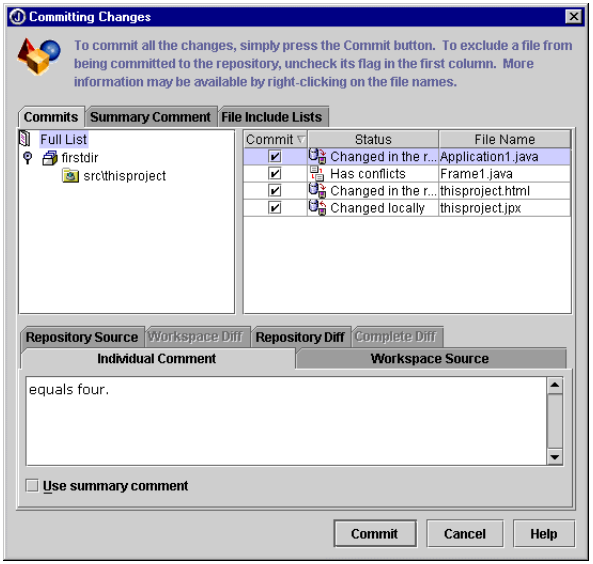


아래쪽에 개별 주석에 대한 추가 탭이 있습니다. 그 창 아래에는 Use Summary Comment 체크 박스도 있습니다. Summary Comment는 커밋되는 모든 파일에 적용할 수 있습니다. 이 페이지에서 Use Summary Comment 체크 박스를 선택하면 Summary Comment 탭에 입력한 주석만 파일 목록에서 선택된 파일에 사용됩니다.

이 파일 중 하나에 주석 기능을 사용합니다. 이러한 변경 사항은 너무 간단하여 길게 설명할 필요가 없으므로 주석은 버전 제어와 관련이 없게 되지만 Jbuilder가 주석 기능을 구축하고 순서를 정하는 방법을 보여준다는 점에서는 서로 관련이 있습니다.

1 목록에서 Application1.java를 선택합니다.

2 Individual Comment 영역에서 equals four를 입력합니다.



입력을 시작하면 Summary Comment 체크 박스 선택이 자동으로 해제됩니다. 여기서 이 체크 박스는 either/or 옵션이며 개별 주석을 사용하거나 요약 주석을 사용합니다.

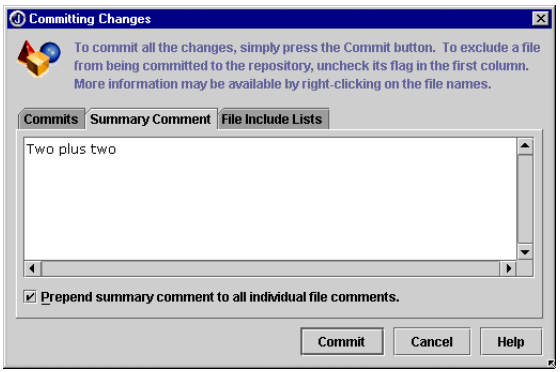
Summary Comment 페이지에서는 개별 주석 및 요약 주석을 모두 사용할 수 있습니다.

3 Summary Comment 페이지를 선택합니다.

4 문장의 처음 반을 다음과 같이 입력합니다.

```
Two plus two
```

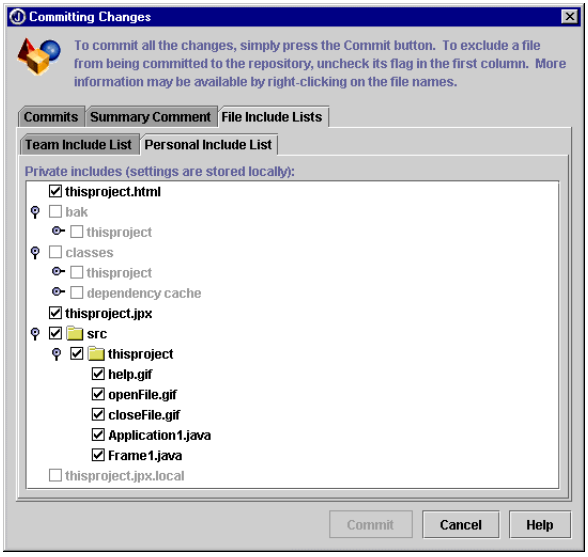
그런 다음에도 Prepend Summary Comment 체크 박스가 계속 선택되어 있어야 합니다.



이 브라우저에서는 주석을 함께 볼 수 없습니다. IDE의 History Info 페이지를 사용하면 전체 주석을 볼 수 있지만 먼저 이 자습서를 완료해야 하므로 History Info 페이지에 표시됩니다. 기록 보기에 대한 자세한 내용은 *JBuilder로 애플리케이션 구축*의 변경 및 버전 비교를 참조하십시오.

다음으로 File Include Lists 페이지를 봅니다.

1. File Include Lists 페이지를 선택합니다.
2. `thisproject` 디렉토리를 확장합니다.
3. Personal Include List 페이지를 선택합니다.



이 체크 박스는 커밋 브라우저에서 보고 작동될 수 있는 파일 및 디렉토리를 표시합니다.

어떤 파일은 관련이 없을 수도 있습니다. 여기서 이러한 파일의 선택을 해제하여 프로젝트의 개인 보기에서 제거합니다.

버전 제어에 적합하지 않은 파일도 있을 수 있습니다. 여기서 해당 파일의 선택을 해제하여 CVS에서 제거합니다.

4. Team과 Personal Include 목록에서 모두 기본값을 유지합니다. Commits 페이지로 돌아갑니다.
5. `Frame1.java` 선택을 해제합니다. 이 연습에서는 커밋하지 않습니다.

마무리할 시간입니다.

1. Commit을 클릭하여 프로젝트를 커밋합니다. 피드백 대화 상자는 커밋 프로세스를 표시합니다.
2. Close를 클릭하여 IDE로 돌아갑니다.
3. 프로젝트를 닫습니다.

축하합니다! CVS 통합에 제공된 JBuilder의 많은 기능들을 성공적으로 사용했습니다.

JBuilder의 CVS에 대한 자세한 내용은 Chapter 3 "JBuilder의 CVS"를 참조하십시오.

Part

II

Enterprise JavaBeans Developer's Guide

Chapter

8

EJB 개발에 대한 소개

http://java.sun.com/products/ejb/docs.html에 있는 "Enterprise JavaBeans(EJB) 사양"은 Java 서버측 컴포넌트 모델 및 애플리케이션 서버를 위한 프로그래밍 인터페이스를 형식적으로 정의합니다. 개발자는 엔터프라이즈의 business 로직을 포함하는 엔터프라이즈 빈이라는 컴포넌트를 구축합니다. 엔터프라이즈 빈은 트랜잭션 관리 및 보안과 같은 서비스를 빈에 제공하는 EJB 서버에서 실행됩니다. 개발자는 이러한 복잡한 저수준 서비스 프로그래밍에 대해 신경쓰지 않으면서도 필요할 때 이러한 서비스를 빈에 사용할 수 있음을 이해하고 조직이나 시스템의 비즈니스 룰을 캡슐화하는 데 집중할 수 있습니다.

Enterprise JavaBeans 사양은 EJB 프레임워크에 대한 최고의 권한을 갖는 동시에 빈이 실행되는 EJB 서버 및 컨테이너를 만드는 Borland와 같은 업체에 유용합니다. 이 설명서는 엔터프라이즈 빈을 개발하고 엔터프라이즈 빈을 사용하는 애플리케이션을 만드는 방법에 관하여 JBuilder 개발자가 알고자 하는 것을 배울 수 있도록 도와 줍니다.

EJB 개발에 이미 익숙해 있고 JBuilder로 엔터프라이즈 빈을 만들고자 할 경우에는 Chapter 8 "EJB 개발에 대한 소개"부터 시작하십시오.

Enterprise JavaBeans이 필요한 이유

많은 사람들이 애플리케이션 개발의 클라이언트/서버 모델을 사용했습니다. 클라이언트 애플리케이션은 로컬 시스템에 상주하고 RDBMS(관계형 데이터베이스 관리 시스템) 같은 데이터 저장소의 데이터에 액세스합니다. 이 모델은 시스템 사용자가 적은 경우에 적합합니다. 이러한 애플리케이션은 증감을 조절하지 못하기 때문에 더 많은 사용자가 데이터에 액세스하려고 할 때에는 사용자의 요구에 응할 수가 없습니다. 클라이언트가 로직을 포함하고 있으므로 각각의 시스템에 모두 설치되어 있어야 합니다. 따라서 관리가 점점 더 어려워집니다.

애플리케이션을 두 개 이상의 클라이언트 서버 모델 계층으로 나누는 것의 이점이 점점 더 분명해지고 있습니다. 다계층 애플리케이션에서는 사용자 인터페이스만이 로컬 시스템에 남아 있게 되고 애플리케이션의 로직은 서버의 중간 계층에서 사용됩니다. 마지막 계층에는 저장된 데이터가 그대로 있습니다. 애플리케이션의 로직이 업데이트되어야 할 경우에는 변경 사항이 서버에 있는 중간 계층의 소프트웨어에서 만들어져서 업데이트 관리를 매우 간소화 합니다.

그렇지만 신뢰성, 안전성 및 관리의 용이성을 갖는 분산 애플리케이션을 만드는 것은 널리 알려져 있듯이 매우 어렵습니다. 예를 들어, 분산 시스템에서 트랜잭션을 관리하는 것은 중요한 작업입니다. 다행스럽게도 EJB 사양을 따르는 컴포넌트를 사용하여 분산 시스템을 구축하면 다음과 같은 작업을 통해 많은 부담을 덜 수 있습니다.

- 분산 시스템의 개발을 전문가에게 할당되는 세부 작업으로 나눕니다.

  예를 들어, 애플리케이션이 회계 시스템일 경우 엔터프라이즈 빈 개발자는 회계를 이해해야 합니다. 시스템 관리자는 배포되어 실행되고 있는 애플리케이션의 모니터링 방법을 알아야 합니다. 각 전문가들은 특정 역할을 맡습니다.

- 엔터프라이즈 빈과 애플리케이션 개발자가 사용할 수 있는 EJB 서버와 컨테이너 서비스를 만듭니다.

  EJB 서버 프로바이더와 EJB 컨테이너 프로바이더(동일한 공급 업체일 경우가 종종 있습니다)가 많은 어려운 작업을 처리해 주므로 개발자가 처리하지 않아도 됩니다. 예를 들어, 엔터프라이즈 빈이 실행되고 있는 컨테이너는 자동으로 빈에 트랜잭션 및 보안 서비스를 제공할 수 있습니다.

- 엔터프라이즈 빈을 이식 가능하게 만듭니다.

  일단 빈이 작성되면 이 빈은 Enterprise JavaBeans 표준을 따르는 모든 EJB 서버에 배포될 수 있습니다. 하지만 각각의 빈에는 공급업체에만 적용되는 요소를 포함시킬 수도 있습니다.

EJB 애플리케이션 개발에서의 역할

EJB 분산 애플리케이션 개발 작업은 6 가지의 명확한 역할로 구분됩니다. 해당 분야의 전문가들이 각각의 역할을 맡습니다. 다음과 같이 작업을 분류하므로 분산 시스템의 생성 및 관리 작업이 훨씬 더 쉽습니다.

애플리케이션 역할

애플리케이션 역할를 맡은 사람들이 엔터프라이즈 빈에 대한 코드 및 엔터프라이즈 빈을 사용하는 애플리케이션을 작성합니다. 두 가지 역할 모두 수준은 다르지만 비즈니스 실행 방법에 대한 이해를 필요로 합니다. 애플리케이션 역할에는 두 가지가 있습니다.

- 빈 프로바이더

  빈 개발자라고도 불리는 빈 프로바이더는 엔터프라이즈 빈을 만들고 엔터프라이즈 빈 안에 비즈니스 메소드의 로직을 작성합니다. 빈 프로바이더는 또한 빈에 대한 원격 및 홈 인터페이스를 정의하고 빈의 배포 디스크립터를 만듭니다. 빈이 어셈블되고 배포되는 방법을 빈 프로바이더가 반드시 알 필요는 없습니다.

- 애플리케이션 어셈블러

  애플리케이션 어셈블러는 엔터프라이즈 빈을 사용하는 애플리케이션을 작성합니다. 이러한 애플리케이션은 일반적으로 GUI 클라이언트, 애플릿, JavaServer Pages 페이지(JSP)와 서블릿과 같은 다른 컴포넌트를 포함합니다. 이러한 컴포넌트는 분산 애플리케이션으로 어셈블됩니다. 어셈블러는 어셈블리 명령을 빈(bean) 배포 디스크립터에 추가합니다. 애플리케이션 어셈블러는 메소드를 호출하기 위해 엔터프라이즈 빈에 포함된 메소드를 잘 알아야 하지만 이러한 메소드가 구현되는 방법에 대해서는 알 필요가 없습니다.

Enterprise JavaBeans에 관심을 가진 JBuilder 사용자들은 대개 빈 프로바이더와 애플리케이션 어셈블러입니다. 따라서 이 설명서는 주로 빈 프로바이더와 애플리케이션 어셈블러를 위해 쓰여졌습니다. JBuilder에는 엔터프라이즈 빈의 개발과 엔터프라이즈 빈을 사용하는 애플리케이션을 단순화하는 마법사, 디자이너 및 기타 툴이 있습니다.

인프라 역할

인프라를 지원하지 않고는 인프라를 사용하는 엔터프라이즈 빈과 애플리케이션을 실행할 수 없습니다. 인프라의 역할은 두 가지로 명확히 구분되지만 거의 대부분의 경우 동일한 공급업체가 두 가지 역할을 수행합니다. 이 두 가지 인프라 모두 시스템 수준의 서버스를 엔터프라이즈 빈에 제공하고 서비스를 실행할 환경을 제공합니다. 인프라의 두 가지 역할은 다음과 같습니다.

- EJB 서버 프로바이더

  EJB 서버 프로바이더는 분산 트랜잭션 관리, 분산 객체 및 기타 저수준 서비스 분야의 전문가들입니다. EJB 서버 프로바이더는 EJB 컨테이너를 실행하는 애플리케이션 프레임워크를 제공합니다. EJB 서비스 프로바이더는 최소한 이름 지정 서비스와 트랜잭션 서비스를 빈에 제공해야 합니다.

- EJB 컨테이너 프로바이더

  EJB 컨테이너 프로바이더는 엔터프라이즈 빈을 배포하는 데 필요한 배포 툴과 빈에 대한 런타임 지원을 제공합니다. 컨테이너는 하나 이상의 빈에 관리 서비스를 제공합니다. 컨테이너는 빈을 위해 EJB 서버와 통신하여 빈이 필요로 하는 서비스에 액세스합니다.

대부분의 경우 EJB 서버 프로바이더와 EJB 컨테이너 프로바이더는 동일한 공급업체입니다. Borland AppServer는 서버와 컨테이너를 모두 제공합니다.

배포 및 작업 역할

EJB 분산 애플리케이션 개발의 마지막 단계는 애플리케이션을 배포하고 엔터프라이즈 컴퓨팅 환경과 현재 실행하는 네트워크 인프라를 모니터하는 것입니다. 배포 및 작업 역할은 다음과 같습니다.

- 배포자(Deployer)

  배포자는 분산 애플리케이션의 작업 환경을 이해합니다. 컨테이너 프로바이더가 제공하는 툴을 사용하여 엔터프라이즈 빈의 속성을 수정함으로써 대상 작업 환경에 EJB 애플리케이션을 적용합니다. 예를 들면 배포 디스크립터에 적절한 속성을 설정함으로써 트랜잭션 및 보안 정책을 설정합니다. 배포자는 또한 기존의 엔터프라이즈 관리 소프트웨어와 애플리케이션을 통합합니다.

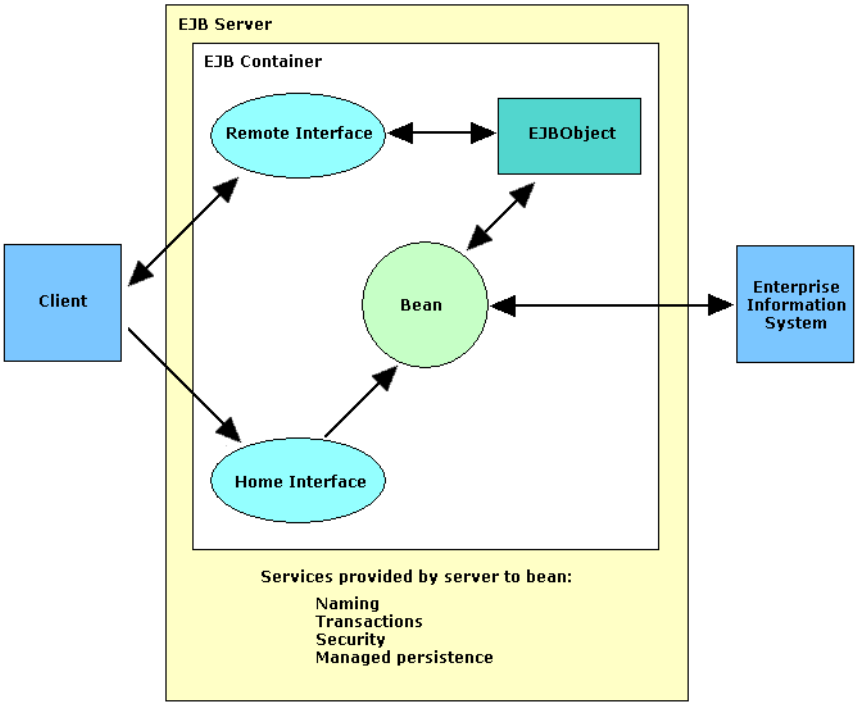
- 시스템 관리자

  일단 애플리케이션이 배포되면 시스템 관리자는 애플리케이션의 실행을 모니터하고 애플리케이션이 비정상적으로 작동할 경우 적절한 액션을 취합니다. 시스템 관리자는 EJB 서버 및 EJB 컨테이너를 포함하는 네트워크 구조와 엔터프라이즈 컴퓨팅 환경을 구성하고 관리를 담당합니다.

EJB 아키텍처

다계층 분산 애플리케이션은 대체로 로컬 시스템에서 실행하는 클라이언트, business 로직을 포함하는 서버에서 실행하는 중간 계층과 엔터프라이즈 정보 시스템(EIS)으로 구성되는 백엔드 계층으로 이루어져 있습니다. EIS는 관계형 데이터베이스 시스템, ERP 시스템, 레거시(legacy) 애플리케이션, 또는 액세스에 필요한 데이터를 저장하는 데이터 저장소가 될 수 있습니다. 다음 그림은 3계층으로 이루어진 전형적인 EJB 다계층 분산 시스템을 보여 줍니다. 클라이언트가 첫 번째, 서버, 컨테이너 그리고 서버와 컨테이너에 배포된 빈이 두 번째과 엔터프라이즈 정보 시스템이 세 번째 계층으로 각각 구성됩니다.

Figure 8.1 EJB 아키텍처 다이어그램



우리의 주된 관심은 엔터프라이즈 빈 개발 방법이므로 중간 계층에 중점을 둡니다.

EJB 서버

EJB 서버는 시스템 서비스를 엔터프라이즈 빈에 제공하고 빈이 실행되는 컨테이너를 관리합니다. EJB 서버는 JNDI(Java Naming and Directory Interface)에 액세스할 수 있는 이름 지정 서비스와 트랜잭션 서비스를 사용할 수 있게 해야 합니다. EJB 서버는 종종 경쟁사 제품과 차별되는 추가적인 기능을 제공합니다. Borland AppServer는 EJB 서버의 한 예입니다.

EJB 컨테이너

컨테이너는 하나 이상의 엔터프라이즈 빈을 위한 런타임 시스템입니다. 컨테이너는 빈과 EJB 서버 간의 통신을 제공합니다. 컨테이너에는 트랜잭션, 보안 및 네트워크 배포 관리 기능이 있습니다. 컨테이너는 코드이면서 동시에 특정 엔터프라이즈 빈에 대한 특정 코드를 생성하는 툴이기도 합니다. 컨테이너는 또한 엔터프라이즈 빈을 배포하기 위한 툴과 애플리케이션을 모니터하고 관리할 수 있는 수단을 제공하기도 합니다.

EJB 서버와 EJB 컨테이너는 모두 빈이 실행되는 환경을 제공합니다. 컨테이너는 하나 이상의 빈에 관리 서비스를 제공합니다. 서버는 빈에 서비스를 제공하지만 컨테이너는 이러한 서비스를 얻기 위해 빈을 대신하여 상호작용합니다.

컨테이너가 Enterprise JavaBeans 아키텍처의 중요한 부분이기는 하지만 엔터프라이즈 빈 개발자와 애플리케이션 어셈블러는 컨테이너를 고려하지 않아도 됩니다. 컨테이너는 EJB 분산 시스템의 배후에 있습니다. 따라서 이 설명서에서는 컨테이너의 정의와 작동 방식을 더 깊이 설명하지 않습니다. 컨테이너에 대한 자세한 내용은 직접 "Enterprise JavaBeans 1.1 사양"(http://java.sun.com/products/ejb/docs.html)을 참조하십시오. Borland EJB 컨테이너에 대한 상세한 정보는 *Borland AppServer's Enterprise JavaBeans Programmer's Guide*를 참조하십시오.

엔터프라이즈 빈 작동 방식

빈 개발자는 다음과 같은 인터페이스와 클래스를 만들어야 합니다.

- 빈에 대한 홈 인터페이스

  홈 인터페이스는 클라이언트가 엔터프라이즈 빈의 인스턴스를 만들고, 찾고 소멸시키는 데 사용하는 메소드를 정의합니다.

- 빈에 대한 원격 인터페이스

  원격 인터페이스는 빈에 구현되는 비즈니스 메소드를 정의합니다. 클라이언트는 원격 인터페이스를 통해 이러한 메소드에 액세스합니다.

- 엔터프라이즈 빈 클래스

  엔터프라이즈 빈 클래스는 빈에 대한 business 로직을 구현합니다. 클라이언트는 빈의 원격 인터페이스를 통해 이러한 메소드에 액세스합니다.

일단 빈이 EJB 컨테이너에 배포되면 클라이언트는 홈 인터페이스에 정의된 `create()` 메소드를 호출하여 빈을 인스턴스화합니다. 홈 인터페이스는 빈 자체에서 구현되지 않고 컨테이너에 의해 구현됩니다. 홈 인터페이스에 선언된 다른 메소드는 클라이언트가 빈의 인스턴스를 찾도록 그리고 더 이상 필요하지 않은 경우에는 빈 인스턴스를 제거하도록 합니다.

엔터프라이즈 빈이 인스턴스화되면 클라이언트는 빈 내에 있는 비즈니스 메소드를 호출할 수 있습니다. 하지만 클라이언트가 빈 인스턴스에 있는 메소드를 직접 호출하는 일은 전혀 없습니다. 클라이언트에 사용할 수 있는 메소드는 빈의 원격 인터페이스에서 정의되며 원격 인터페이스는 컨테이너에 의해 구현됩니다. 클라이언트가 메소드를 호출하면 컨테이너는 그 요청을 받아들여 빈 인스턴스에 위임(delegate)합니다.

엔터프라이즈 빈의 타입

엔터프라이즈 빈은 세션 빈이거나 엔티티 빈일 수 있습니다.

세션 빈

엔터프라이즈 세션 빈은 단일 클라이언트를 위하여 실행됩니다. 어떤 의미로 보면 세션 빈은 EJB 서버의 클라이언트를 나타냅니다.

세션 빈은 클라이언트의 상태를 유지할 수 있는데 이는 세션 빈이 클라이언트에 대한 정보를 보유할 수 있다는 것을 의미합니다. 세션 빈이 사용될 수도 있는 전형적인 예가 웹의 온라인 상점에서 개인의 쇼핑 카트입니다. 쇼핑객이 품목을 선택하여 "카트"에 넣는 것과 같이 세션 빈은 선택된 항목의 목록을 보유합니다.

세션 빈의 수명은 짧을 수 있습니다. 일반적으로 클라이언트가 세션을 종료할 때 클라이언트에서 빈을 제거합니다.

세션 빈은 상태 있음(stateful)이거나 상태 없음(stateless)일 수 있습니다. 상태 없는 빈은 특정 클라이언트의 상태를 유지하지 않습니다. 상태 없는 빈은 대화 상태를 유지하지 않으므로 다중 클라이언트를 지원하는 데 사용될 수 있습니다.

엔티티 빈

엔티티 빈은 데이터베이스에 있는 데이터의 객체 뷰를 제공합니다. 일반적으로 빈은 일련의 관계형 데이터베이스 테이블에 있는 행을 나타냅니다. 엔티티 빈은 보통 하나 이상의 클라이언트를 지원합니다.

세션 빈과 달리 엔티티 빈의 수명은 깁니다. 엔티티 빈은 지속 상태를 유지하므로 특정 클라이언트가 필요로 하는 기간 동안이 아니라 데이터베이스에 데이터가 있는 한 남아 있게 됩니다.

컨테이너는 빈의 지속성을 관리할 수 있으며 빈 자체도 자신의 지속성을 관리할 수 있습니다. Bean 관리 방식(Bean-managed persistence, BMP)인 경우 빈 개발자는 데이터베이스에 대한 호출을 포함하는 코드를 작성해야 합니다.

엔터프라이즈 빈 개발

이후의 몇몇 장에서는 엔터프라이즈 빈을 더 쉽고 빠르게 만들도록 도와주는 JBuilder 마법사, 디자이너 및 툴의 사용 방법에 대해 설명합니다. Enterprise JavaBeans가 무엇이고 어떻게 작동하며 Enterprise JavaBeans의 요구 조건이 무엇인지를 사용자가 이해한다고 가정합니다.

EJB에 대한 지식이 부족하거나 JBuilder의 EJB 마법사와 툴을 사용하기 전에 EJB 개발에 대해 자세히 알고 싶은 경우에는 이 장을 시작하기 전에 Chapter 16, "세션 빈 개발" 그 다음 장을 읽으십시오.

JBuilder를 사용하여 Enterprise JavaBeans를 개발하기 위해서는 다음과 같은 몇 가지 단계를 거칩니다.

1 대상 애플리케이션 서버 설정(Chapter 9 참조)

2 EJB 그룹 생성(10- 2 페이지 참조)

3 엔터프라이즈 빈 및 엔터프라이즈 빈의 홈/원격 인터페이스 생성(10- 4 페이지 참조)

4 빈 컴파일(10- 16 페이지 참조)

5 배포 디스크립터 편집(10- 18 페이지 참조)

6 테스트용 클라이언트 애플리케이션 생성(12- 1 페이지 참조)

7 엔터프라이즈 빈 테스트(12- 5 페이지참조)

8 애플리케이션 서버에 배포(13- 5 페이지 참조)

또한 JDBC를 통해 액세스 가능한 데이터베이스에 있는 기본 테이블을 기반으로 한 엔티티 엔터프라이즈 빈을 만드는 데에도 JBuilder를 사용할 수 있습니다. 11- 1페이지의 "EJB Entity Bean Modeler를 사용하여 엔티티 빈 생성"을 참조하십시오.

엔터프라이즈 빈에 대한 원격 인터페이스를 먼저 만들고자 할 경우에는 EJB Bean Generator를 사용하여 뼈대 빈(skeleton bean) 클래스와 해당 원격 인터페이스를 기반으로 한 홈 인터페이스를 만들 수 있습니다. 자세한 내용은 10- 11페이지의 "원격 인터페이스에서 빈 클래스 생성"을 참조하십시오.

Chapter

9

대상 애플리케이션 서버 설치

Enterprise JavaBeans를 생성하기 전에 먼저 엔터프라이즈 빈을 배포할 애플리케이션 서버를 설치해야 합니다.

참고 어떤 경우라도 Borland AppServer(BAS) 또는 Inprise Application Server 4.1(IAS)이 컴퓨터에 먼저 설치되어 있어야 합니다.

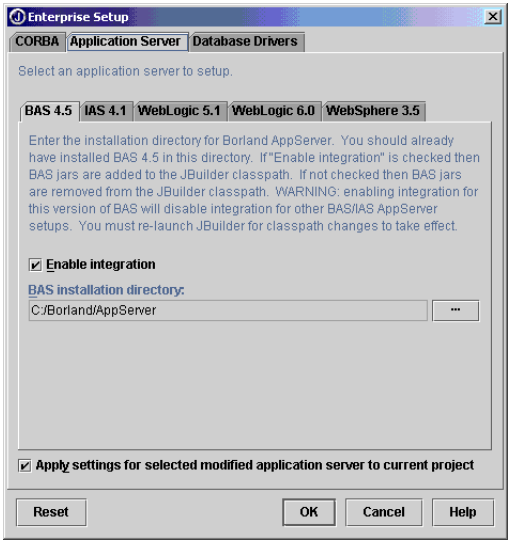
다음과 같은 방법으로 JBuilder를 하나 이상의 대상 애플리케이션 서버에 설치합니다.

1. Tools|Enterprise Setup을 선택합니다.
2. Application Server 페이지를 선택합니다.

참고 엔터프라이즈 빈을 개발하려는 대상 애플리케이션 서버에 상관 없이 다음 단계를 수행해야 합니다. 이 단계를 수행하지 않으면 객체 갤러리에서 EJB 마법사를 사용할 수 없습니다.

3. BAS 4.5 페이지 또는 IAS 4.1 페이지를 선택한 다음 Borland AppServer 4.5 또는 Inprise Application Server 4.1이 설치된 디렉토리를 지정합니다. 이 디렉토리는 일반적으로 BAS 4.5의 경우 `/Borland/`

AppServer입니다. Borland 애플리케이션 서버 페이지 중 하나를 설치하면 다른 서버를 설치할 수 없습니다.



4 WebLogic 또는 WebSphere 서버를 대상으로 할 경우 대상으로 할 버전의 WebLogic 또는 WebSphere 페이지를 선택한 다음 WebLogic 또는 WebSphere 애플리케이션 서버가 설치되는 디렉토리를 지정합니다. WebSphere의 경우 WebSphere와 함께 제공된 IBM SDK의 설치 디렉토리도 지정해야 합니다. WebLogic 6.0의 경우 BEA 홈 디렉토리가 필요합니다.

5 현재 프로젝트에 대한 대상 애플리케이션 서버로 설정할 애플리케이션 서버를 사용하려면 Apply Settings For Selected Modified Application Server To Current Project 체크 박스에 선택 표시를 합니다.

6 OK를 선택합니다.

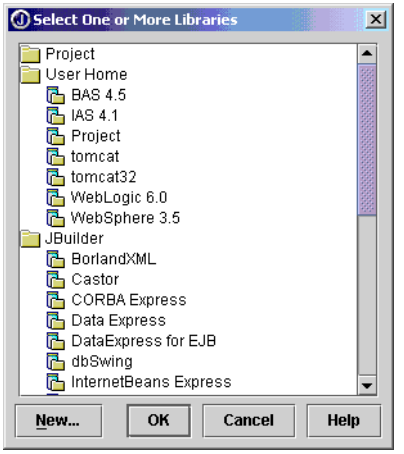
대화 상자를 닫으면 EJB 개발에 필요한 모든 BAS 파일을 포함하는 BAS 4.5 라이브러리가 자동으로 생성됩니다. (Inprise Application Server를 대상으로 할 경우에는 라이이브러리가 IAS 4.1입니다.)

EJB 마법사를 사용하려면 JBuilder를 종료한 후 다시 시작해야 합니다. VisiBroker ORB를 JBuilder에서 사용될 수 있도록 하려면 JBuilder를 종료한 후 다시 시작하기 전에 단계가 완료될 때까지 기다립니다.

프로젝트에 애플리케이션 서버 파일 추가

다음은 애플리케이션 서버 파일을 포함하는 라이브러리를 프로젝트에 추가해야 합니다. 새 프로젝트를 시작할 때마다 이 단계를 수행해야 합니다. Enterprise Setup 대화 상자의 Apply Settings For Selected Modified Application Server To Current Project 체크 박스에 선택 표시를 한 경우 JBuilder에서 이미 현재 프로젝트에 필요한 라이브러리를 추가했으므로 이 단계를 건너뛸 수 있습니다. 하지만 다음 번에 새 프로젝트를 시작할 때는 이 단계를 수행하여 현재 프로젝트에 대한 라이브러리를 추가해야 합니다.

1. Project|Project Properties를 선택한 다음 Paths 탭이 선택되어 있는지 확인합니다.
2. Required Libraries 탭을 클릭합니다.
3. Add 버튼을 클릭하여 Select One or More Libraries 대화 상자를 표시합니다. 사용 가능한 라이브러리는 설치되어 있었던 애플리케이션 서버 및 Tools|Enterprise Setup을 사용하여 설치한 애플리케이션 서버에 따라 다릅니다.



4. 해당 라이브러리를 선택합니다. 예를 들어, 대상이 Borland AppServer일 경우 BAS 4.5 라이브러리를 선택한 다음 OK를 두 번 클릭하여 대화 상자를 닫습니다.

ORB를 JBuilder에서 사용 가능하게 만들기

Tools|Enterprise Setup을 사용하여 Borland AppServer 또는 Inprise Application Server를 설치할 때 CORBA도 동시에 자동으로 설정됩니다. Enterprise Setup 대화 상자의 CORBA 페이지에서 현재 설정을 볼 수 있습니다. 이 페이지를 사용하여 VisiBroker SmartAgent 포트를 고유 번호로 설정합니다. 또한 Add The VisiBroker SmartAgent Item To The Tools

Menu 옵션을 선택 표시하면 Tools 메뉴에 SmartAgent를 시작하는 명령을 추가할 수 있습니다.

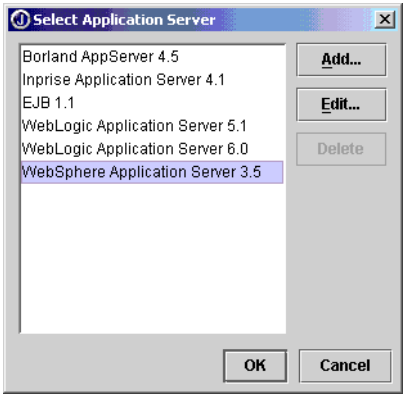
마찬가지로 VisiBroker SmartAgent를 시작하는 하나의 단계를 수행해야 합니다. 이 단계에서는 클라이언트에서 이름 지정 서비스를 찾는 방법 등의 초기 부트스트랩(bootstrap) 문제를 처리합니다.

SmartAgent를 시작하려면 Tools|VisiBroker SmartAgent를 선택합니다.

애플리케이션 서버 선택

JBuilder는 여러 애플리케이션 서버 중 하나를 대상으로 할 수 있습니다. 대상 애플리케이션 서버를 선택하고 난 다음 프로젝트의 속성을 수정하여 JBuilder 설정 작업을 완료합니다.

1. Project|Project Properties를 선택합니다.
2. Servers 탭을 클릭합니다.
3. ... 버튼을 클릭하면 Select Application Server 대화 상자가 나타납니다. 이 대화 상자에서 사용할 수 있는 애플리케이션 서버는 JBuilder에서 지원하거나 JBuilder의 OpenTools API 또는 이 대화 상자의 Add 버튼을 사용하여 추가된 애플리케이션 서버에 따라 다릅니다.



4. 빈을 실행하기 위해 구축할 애플리케이션 서버를 선택합니다. 기본 서버는 Borland AppServer 4.5입니다.

   EJB 1.1은 일반적 옵션입니다. 사용하는 애플리케이션 서버가 현재 JBuilder에서 지원되지 않는다면 EJB 1.1을 선택합니다. 원하는 정확한 설정을 얻으려면 해당 애플리케이션 서버와 함께 제공되는 툴을 사용하여 결과로 생성되는 Deployment Descriptor를 편집해야 할 것입니다. 또한 특정 애플리케이션 서버를 대상으로 하지 않을 경우에 이 옵션을 선택할 수도 있습니다.

5. OK를 선택하여 대화 상자를 닫습니다.

Chapter

10

JBuilder를 사용한 엔터프라이즈 빈 생성

생성할 각 엔터프라이즈 빈은 JBuilder EJB 그룹에 속해야 합니다. EJB 그룹은 단일 JAR 파일 안에 배포될 하나 이상의 빈을 논리적으로 그룹화한 것입니다. 이 그룹은 해당 JAR 파일의 Deployment Descriptor를 생성하는 데 사용되는 정보를 포함합니다. Deployment Descriptor 에디터를 사용하여 EJB 그룹의 내용을 편집할 수 있습니다.

EJB 그룹이 있고 Deployment Descriptor 에디터를 사용하여 편집하고 나면 JAR를 생성할 EJB 그룹을 만들거나 구축할 수 있습니다. JBuilder에서는 Deployment Descriptor를 사용하여 패키지로 만들어질 클래스 파일을 쉽게 식별합니다.

파일 확장자에 따라 EJB 그룹은 다음 두 가지 타입 중 하나가 될 수 있습니다.

.ejbgrp	바이너리 파일.
.ejbgrpx	XML 파일. 정보는 `.ejbgrp` 파일의 정보와 동일하지만 텍스트 파일이기 때문에 버전 제어 시스템을 사용하면 더 쉽게 작업할 수 있습니다.

하나의 프로젝트에서 두 개 이상의 EJB 그룹을 가질 수 있습니다. 단일 프로젝트의 모든 EJB 그룹은 동일한 프로젝트 classpath 및 JDK를 사용하므로 동일한 대상 애플리케이션 서버에 대해 구성됩니다.

아직 그렇게 하지 않은 경우 Chapter 9 "대상 애플리케이션 서버 설치"의 지침을 따르십시오. 애플리케이션 서버 파일을 포함하는 라이브러리를 각 EJB 프로젝트에 추가하기 위한 절차를 따라야 합니다.

EJB 그룹 생성

EJB 그룹을 생성하는 방법에는 두 가지가 있습니다.

- 아직 엔터프라이즈 빈을 생성하지 않은 경우 Empty EJB Group 마법사를 사용하여 비어 있는 EJB 그룹을 생성합니다.
- 기존 엔터프라이즈 빈의 Deployment Descriptor에서 EJB를 생성하려면 EJB Group From Descriptors 마법사를 사용합니다.

EJB 그룹 마법사를 시작하기 전에 열려 있는 프로젝트가 없으면 JBuilder에서 먼저 Project 마법사를 표시합니다. 새 프로젝트를 생성하고 난 다음 선택한 EJB 마법사가 나타납니다.

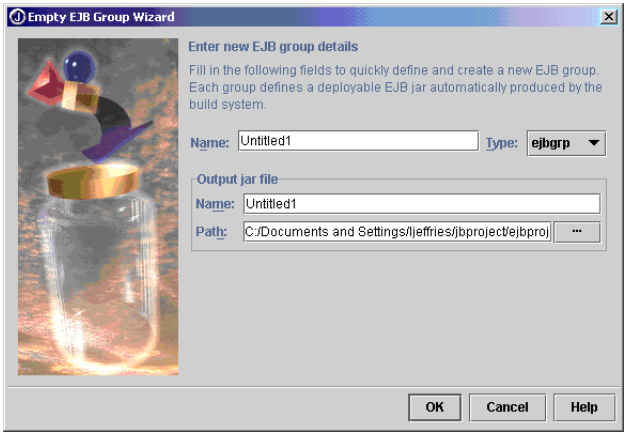
비어 있는 EJB 그룹 생성

아직 엔터프라이즈 빈을 생성하지 않았으면 먼저 비어 있는 EJB 그룹을 만들면서 시작합니다. 다음과 같은 방법으로 비어 있는 EJB 그룹을 만듭니다.

1 File|New를 선택한 다음 Enterprise 탭을 클릭합니다.

참고 애플리케이션 서버를 아직 설정하지 않았으면 Enterprise 페이지에서 EJB 마법사를 사용할 수 없습니다. 이 작업을 수행하는 방법에 대한 내용은 Chapter 9 "대상 애플리케이션 서버 설치"을 참조하십시오.

2 Empty EJB Group wizard 아이콘을 더블 클릭하면 마법사가 나타납니다.



3 EJB 그룹의 이름을 지정합니다.

4 새 그룹의 타입을 지정합니다.

Deployment Descriptor를 Jbuilder 5 이전 버전에 사용되었던 타입인 .zip 형식으로 내부적으로 저장하는 ejbgrp를 선택하거나 XML 형식으로 Deployment Descriptor를 저장하는 ejbgrpx를 선택할 수 있습니다. XML 형식을 사용하면 변경 사항이 소스 제어 시스템에 확인되는 경우

사용자가 변경 사항을 병합할 수 있습니다. ejbgrpx는 JBuilder의 이전 버전과 파일을 공유하지 않는 경우에 사용하는 것이 좋습니다.

5 엔터프라이즈 빈이 포함될 JAR 파일의 이름을 지정합니다.

JBuilder는 사용자의 EJB 그룹 이름과 동일한 이름을 기본 이름으로 제공합니다. 기본 이름을 그대로 사용하거나 다른 이름을 지정할 수 있습니다. JBuilder에서는 해당 프로젝트 경로에 기반한 경로도 입력해 두었습니다. 이 기본 경로를 원하는 다른 경로로 변경할 수도 있고 그대로 사용할 수도 있습니다.

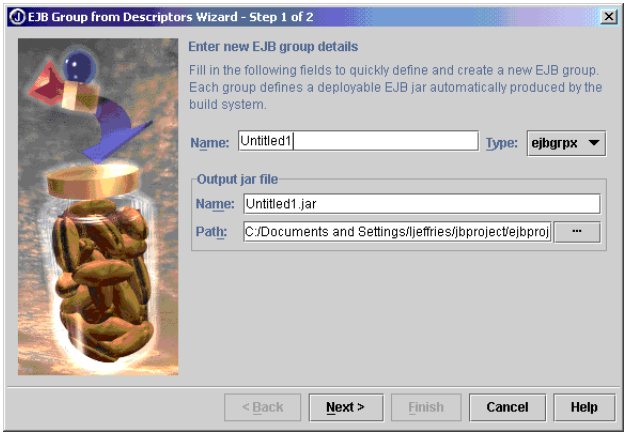
6 OK를 클릭하면 EJB 그룹이 만들어집니다.

기존 엔터프라이즈 빈에서 EJB 그룹 생성

기존 BAS 엔터프라이즈 빈이 있을 경우 다음 절차에 따라 EJB 그룹에 기존 빈을 추가합니다.

1 File|New를 선택한 다음 Enterprise 탭을 클릭합니다.

2 EJB Group From Descriptors 마법사 아이콘을 더블 클릭하면 마법사가 나타납니다.



3 새 EJB 그룹의 이름을 지정합니다.

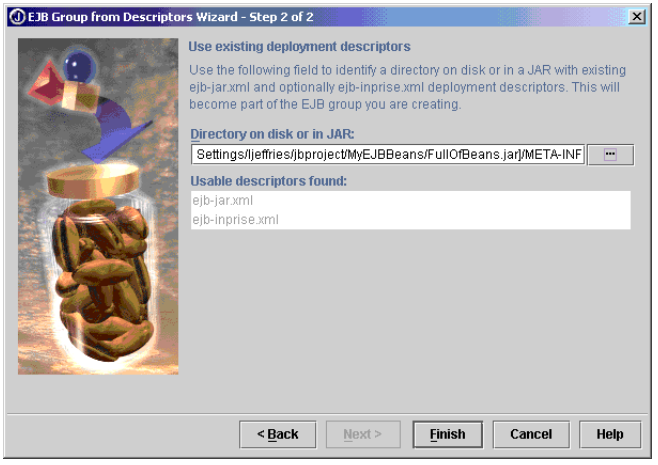
4 새 그룹의 타입을 지정합니다.

Deployment Descriptor를 Jbuilder 5 이전 버전에 사용되었던 타입인 .zip 형식으로 내부적으로 저장하는 ejbgrp를 선택하거나 XML 형식으로 Deployment Descriptor를 저장하는 ejbgrpx를 선택할 수 있습니다. XML 형식을 사용하면 변경 사항이 소스 제어 시스템에 확인되는 경우 사용자가 변경 사항을 병합할 수 있습니다. ejbgrpx는 JBuilder의 이전 버전과 파일을 공유하지 않는 경우에 사용하는 것이 좋습니다.

5 엔터프라이즈 빈이 포함될 JAR 파일의 이름 및 경로를 지정합니다.

JBuilder는 사용자의 EJB 그룹 이름과 동일한 이름을 기본 이름으로 제공합니다. 기본 이름을 그대로 사용하거나 다른 이름을 지정할 수 있습니다.

6 Next를 클릭한 다음 그룹을 구성하려는 기존 Deployment Descriptor가 들어 있는 디렉토리를 지정합니다. 그러면 마법사가 지정된 디렉토리에 있는 Deployment Descriptor를 Usable Descriptors Found 필드에 나열합니다.



7 Finish를 클릭하여 기존 빈의 Deployment Descriptor를 포함하는 EJB 그룹을 생성합니다.

엔터프라이즈 빈 생성

JBuilder 객체 갤러리에는 엔터프라이즈 빈을 생성하기 위한 두 가지 마법사, 즉 Enterprise JavaBean 마법사와 EJB Entity Bean Modeler가 들어 있습니다. Wizard 메뉴에는 EJB Bean Generator가 들어 있습니다. 이 단원에서는 Enterprise JavaBean 마법사를 사용한 엔터프라이즈 빈의 생성을 설명합니다.

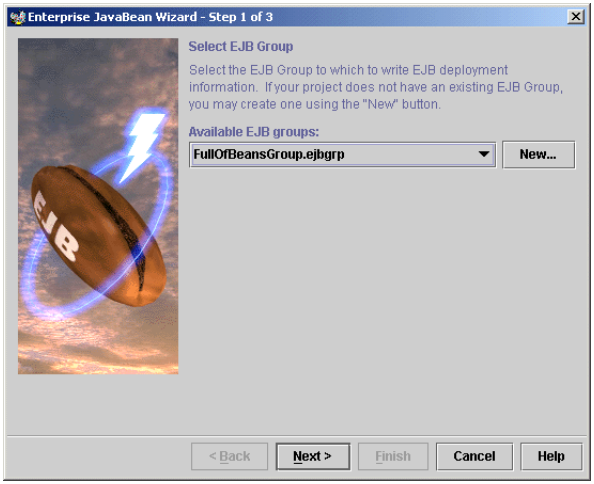
Enterprise JavaBean 마법사와 EJB Entity Bean Modeler는 엔터프라이즈 빈 클래스와 홈 및 원격 인터페이스가 동시에 생성되는 모델을 사용합니다. 먼저 원격 인터페이스를 생성하여 엔터프라이즈 빈 개발을 시작하려는 경우 EJB Bean Generator를 사용하여 사용자가 만든 원격 인터페이스에서 빈 클래스를 생성하는 작업에 대한 정보를 10- 11페이지의 "원격 인터페이스에서 빈 클래스 생성"에서 참조하십시오.

다음과 같은 방법으로 Enterprise JavaBean 마법사에서 엔터프라이즈 빈 생성을 시작합니다.

1 File|New를 선택한 다음 Enterprise 탭을 클릭합니다.

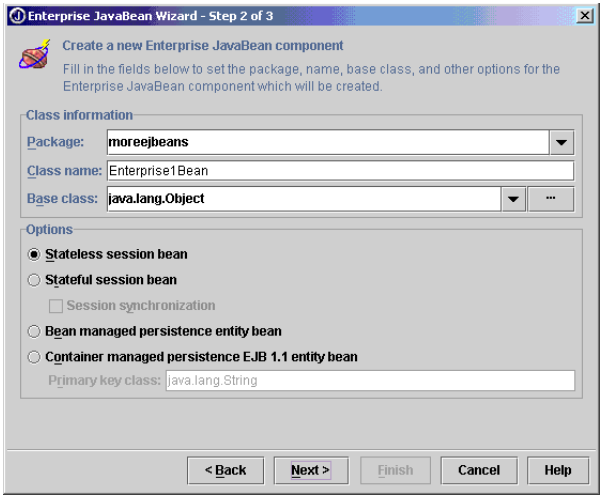
2 Enterprise JavaBean 마법사 아이콘을 더블 클릭합니다.

마법사가 나타납니다.



3 드롭다운 목록에서 엔터프라이즈 빈이 속할 EJB 그룹을 선택합니다. Next를 선택하여 마법사의 2 페이지를 표시합니다.

Enterprise JavaBeans 마법사를 시작하기 전에 EJB 그룹을 정의하지 않았거나 다른 EJB 그룹을 생성하려면 New 버튼을 클릭하여 Empty EJB Group 마법사를 시작합니다. 엔터프라이즈 빈을 생성하기 전에 프로젝트에 최소한 하나의 EJB 그룹이 정의되어 있어야 합니다. 일단 Empty EJB Group 마법사에서 EJB 그룹을 생성했으면 새 그룹을 선택한 다음 Next를 선택하여 Enterprise JavaBean 마법사를 계속 진행합니다.



4 빈(bean) 클래스의 클래스 이름, 클래스가 포함될 패키지 및 빈의 기본 클래스를 지정합니다.

그런 다음 세션 빈과 엔티티 빈 중 어느 것을 생성할 것인지 결정해야 합니다.

세션 빈 생성

세션 빈을 생성하려는 경우

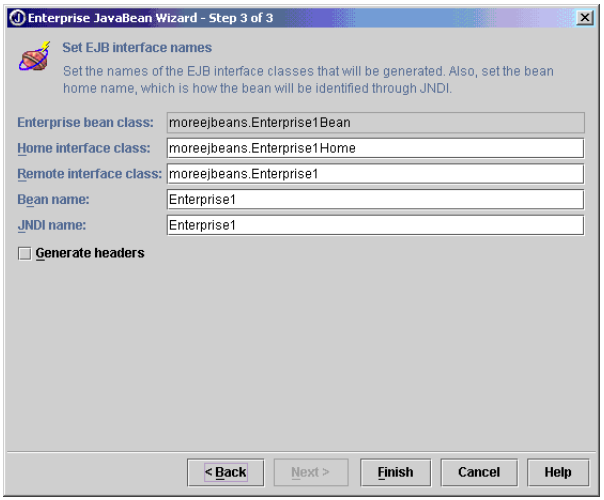
1 Stateless Session Bean 또는 Stateful Session Bean을 클릭합니다.

세션 빈 타입에 대한 자세한 내용은 16-1페이지의 "세션 빈 타입"을 참조하십시오.

2 Stateful Session Bean을 선택하는 경우 Session Synchronization 체크 박스에 선택 표시하여 `SessionSynchronization` 인터페이스 구현을 선택할 수도 있습니다.

`SessionSynchronization` 인터페이스에 대한 내용은 16-7페이지의 "SessionSynchronization 인터페이스"를 참조하십시오.

3 Next를 클릭하여 3 단계로 이동합니다.



4 Home Interface Class, Remote Interface Class 및 Bean Home Name의 이름을 지정합니다. JBuilder는 사용자의 빈(bean) 클래스 이름에 기반한 기본 이름을 제시합니다.

5 Finish를 클릭합니다.

엔티티 빈 생성

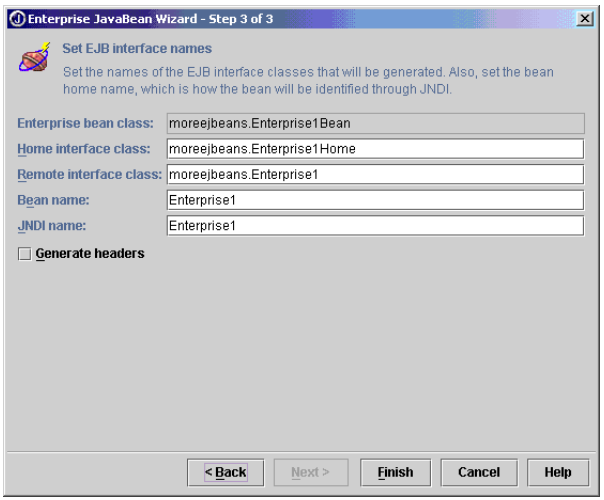
엔티티 빈을 생성하려는 경우

1 Bean Managed Persistence Entity Bean 옵션 또는 Container Managed Persistence 1.1 Entity Bean 옵션을 선택합니다. (WebSphere 3.5가 대상 애플리케이션 서버인 경우 두 번째 옵션은 Container Managed Persistence 1.0 Entity Bean입니다.)

Bean 관리 방식(Bean-managed persistence, BMP)과 Container 관리 방식(Container-managed persistence, CMP)에 대한 내용은 17-1페이지의 "지속성과 엔티티 빈"을 참조하십시오.

2 Primary Key Class를 지정합니다.

3 Next를 클릭하여 3 단계로 이동합니다.



4 Home Interface Class, Remote Interface Class 및 Bean Home Name의 이름을 지정합니다. JBuilder는 사용자의 빈(bean) 클래스 이름에 기반한 기본 이름을 제시합니다.

5 Finish를 클릭합니다.

Finish 버튼을 클릭하고 난 다음 JBuilder에서 빈 클래스와 이 클래스의 홈 및 원격 인터페이스를 생성합니다. 이는 프로젝트 창에 표시될 것입니다. 빈 클래스의 소스 코드를 살펴보면 세션 빈일 경우 클래스에서 `SessionBean` 인터페이스를 구현하고 엔티티 빈일 경우 `EntityBean` 인터페이스를 구현하는 것을 알 수 있습니다. JBuilder는 모든 엔터프라이즈 빈에서 구현해야 할 메소드를 추가했으며 메소드의 몸체는 비어 있습니다. 이러한 메소드가 호출될 때 빈에 필요한 로직을 제공하도록 이 메소드 몸체에 코드를 추가할 수 있습니다.

홈 인터페이스는 `EJBHome` 인터페이스를 확장하며 그 안에는 빈을 생성하는 데 필요한 `create()` 메소드를 포함합니다. 원격 인터페이스는 `EJBObject`를 확장하지만 빈에 대한 비즈니스 로직 메소드를 아직 선언하지 않았으므로 비어 있습니다.

Enterprise JavaBeans 마법사를 사용하여 엔티티 빈을 생성할 수도 있지만 EJB Entity Bean Modeler를 사용하여 생성하는 방법이 더 선호됩니다. Enterprise JavaBean 마법사를 사용하여 생성한 엔티티 빈은 사용자가 좀 더 완벽하게 완성시킬 때까지 Deployment Descriptor 에디터에서 검증을 통과하지 못할 수 있습니다.

빈에 비즈니스 로직 추가

빈(bean) 클래스의 소스 코드 안에 엔터프라이브 빈에서 필요로 하는 로직을 구현할 메소드를 정의 및 작성합니다.

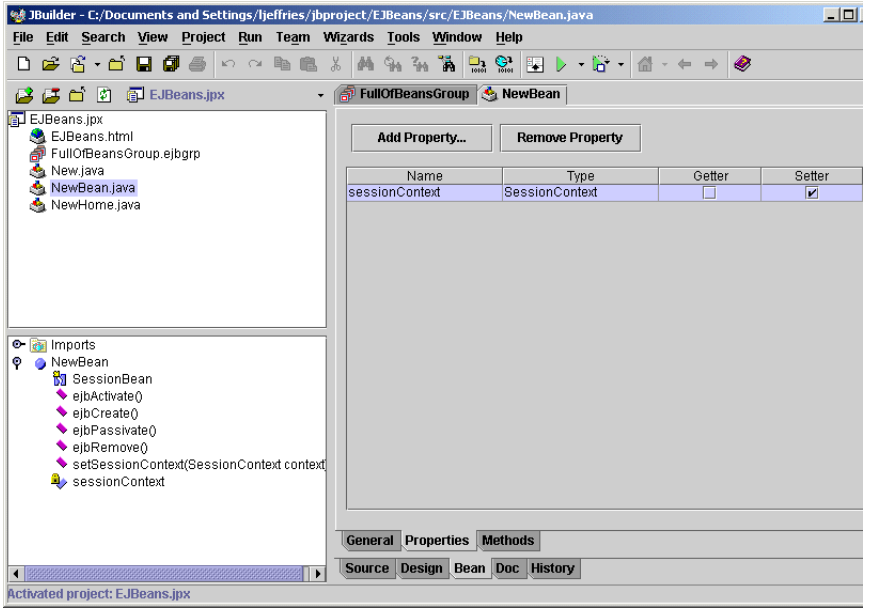
빈에 속성을 추가해야 하는 경우 소스 코드에 직접 속성을 추가하거나 Bean 디자이너의 Properties 페이지를 사용할 수 있습니다.

다음과 같은 방법으로 Bean 디자이너를 사용하여 속성 작업을 수행합니다.

1 프로젝트 창에서 빈 클래스를 더블 클릭합니다.

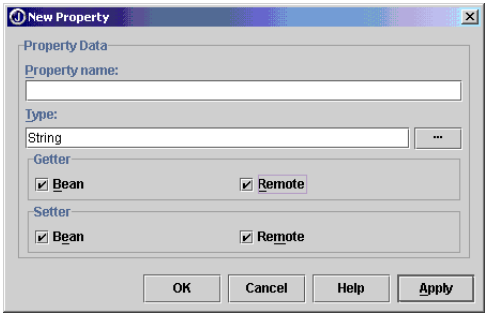
2 Bean 탭을 클릭하여 빈 디자이너를 표시합니다.

3 Properties 탭을 클릭하여 Properties 페이지를 표시합니다.



다음과 같은 방법으로 새 속성을 추가합니다.

1 Add Property 버튼을 클릭하여 New Property 대화 상자를 표시합니다.



2 Property Name 및 해당 Type을 지정합니다.

3 Getter 및 Setter 옵션을 설정하여 액세스 메소드를 지정합니다.

   속성에 getter 액세스 메소드가 필요하다고 판단하는 경우 이 메소드를 빈 클래스 또는 원격 인터페이스에 표시할 것인지 여부도 결정할 수 있습니다. 속성에 setter 액세스 메소드가 필요하다고 판단하는 경우 이 메소드를 빈 클래스 또는 원격 인터페이스에 표시할 것인지 여부도 결정할 수 있습니다.

4 Apply를 선택하면 빈의 소스 코드에 새 속성 정의를 바로 추가할 수 있습니다. 지정한 액세스 메소드는 선택한 옵션에 따라 빈 클래스 또는 원격 인터페이스에 추가됩니다.

5 이 대화 상자에서 새 속성을 계속 추가할 수 있습니다. 완료했으면 OK를 선택합니다.

Enterprise JavaBean 마법사를 사용하여 CMP로 엔티티 빈을 시작하는 경우에는 빈에 속성을 추가해야 합니다. 속성 중 하나를 기본 키로 해야 하며 Deployment Descriptor 에디터의 Persistence 패널에서 기본 키를 구성하는 하나 이상의 필드를 지정해야 함을 기억하십시오 이 작업이 실패하면 Deployment Descriptor 에디터에서 Deployment Descriptor의 유효함을 확인할 수 없을 것입니다.

또한 Properties 페이지를 사용하여 속성을 수정할 수도 있습니다. 예를 들어, 빈의 속성을 선언할 때 setter를 지정하지 않았는데 빈에서 이 속성이 필요해진 경우에는 간단히 Properties 페이지에서 해당 속성의 Setter 체크 상자에 선택 표시만 하면 JBuilder는 소스 코드에 setter 메소드를 추가합니다. 또는 해당 체크 박스를 선택 해제하여 getter 또는 setter 속성을 제거할 수도 있습니다.

다음과 같은 방법으로 Properties 페이지를 사용하여 빈에서 속성을 제거합니다.

1 속성 테이블에서 속성을 선택합니다.

2 Remove 버튼을 클릭합니다.

   JBuilder에서 해당 속성 및 연견된 코드를 제거할 것인지 묻습니다.

3 Yes를 선택합니다.

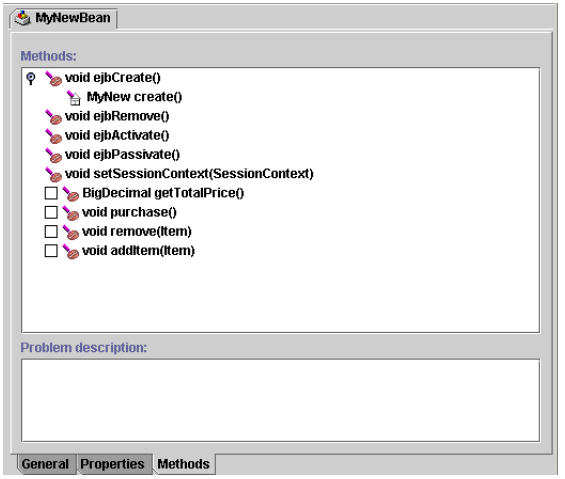
또한 Properties 페이지를 사용하여 속성의 이름 및 해당 타입을 변경할 수도 있습니다. Bean 디자이너는 two-way 툴이므로 Properties 페이지에서 변경한 내용이 코드에 반영되고 코드에서 변경한 내용이 Properties 페이지에 반영됩니다.

원격 인터페이스를 통한 비즈니스 메소드 노출

빈의 소스 코드에 비즈니스 로직 메소드를 선언했으면 원격 인터페이스에 추가할 메소드를 지정해야 합니다. 클라이언트에서는 빈의 원격 인터페이스를 통해 노출되는 이러한 메소드들만 호출할 수 있습니다.

다음과 같은 방법으로 원격 인터페이스에 메소드를 추가합니다.

1. 프로젝트 창에서 엔터프라이즈 빈을 더블 클릭합니다.
2. Bean 탭을 클릭하여 빈 디자이너를 표시합니다.
3. Methods 탭을 클릭합니다.
4. Methods 상자에서 원격 인터페이스에서 노출하려는 메소드 옆에 있는 체크 박스에 선택 표시를 합니다.



Methods 상자에서 메소드에 선택 표시를 하면 원격 인터페이스에 선택된 메소드가 추가됩니다.

원격 인터페이스에서 메소드를 제거하려면 Methods 상자에서 메소드 옆에 있는 체크 박스를 선택 해제합니다.

메소드 중 하나를 편집하려면 편집할 메소드를 마우스 오른쪽 버튼으로 클릭하여 컨텍스트 메뉴를 표시한 다음 Edit Selected를 선택합니다. 코드 에디터에서 파일이 열리고 커서가 해당 메소드 위에 놓여지면 편집을 시작할 수 있습니다.

이 컨텍스트 메뉴에는 유용하게 사용할 수 있는 다른 명령들도 있습니다. Remove Selected를 선택하면 빈 클래스에서 메소드를 제거할 수 있습니다. Check All을 선택하면 모든 메소드가 선택되어 원격 인터페이스에 추가됩니다. Uncheck All을 선택하면 모든 메소드의 선택이 해제되어 원격 인터페이스에 메소드가 하나도 추가되지 않습니다.

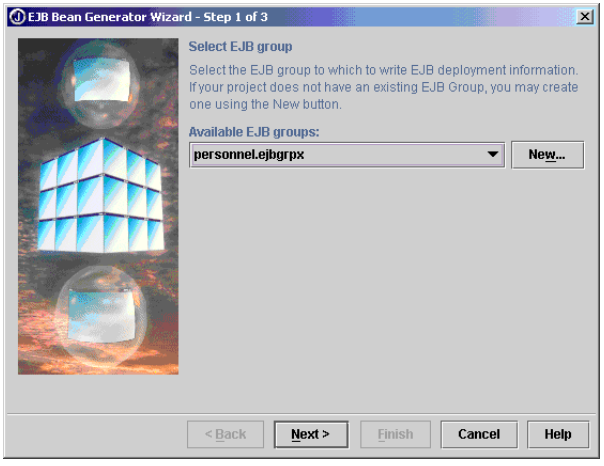
Methods 페이지를 사용하면 빈 클래스에 선언된 메소드가 홈 및 원격 인터페이스에서와 동일한 메소드 시그너처를 가지고 있는지 확인할 수 있습니다. 예를 들어, 빈 클래스의 `ejbCreate()` 메소드에 매개변수 하나를 추가하지만 홈 인터페이스의`create()` 메소드에는 이 매개변수를 추가하지 않는다고 가정합니다. Methods 상자에는 `ejbCreate()` 메소드와 `create()` 메소드가 모두 빨간 텍스트로 표시됩니다. 빨간색 텍스트로 표시되는 메소드 하나를 클릭하면 Problem Description 상자에 무엇이 문제인지 설명하는 메시지가 표시됩니다. 그런 다음 메소드 시그너처를 일치시키고 문제를 고칠 다른 매개변수를 `create()` 메소드에 추가할 수 있습니다. 그리고 빈 클래스에서 메소드를 제거해야 하는데 원격 인터페이스에서 메소드를 제거하는 것을 잊어버렸을 경우에는 원격 인터페이스에서 메소드를 제거해야 함을 알려주기 위해 Methods 상자가 이 메소드를 빨간 텍스트로 표시합니다.

원격 인터페이스에서 빈 클래스 생성

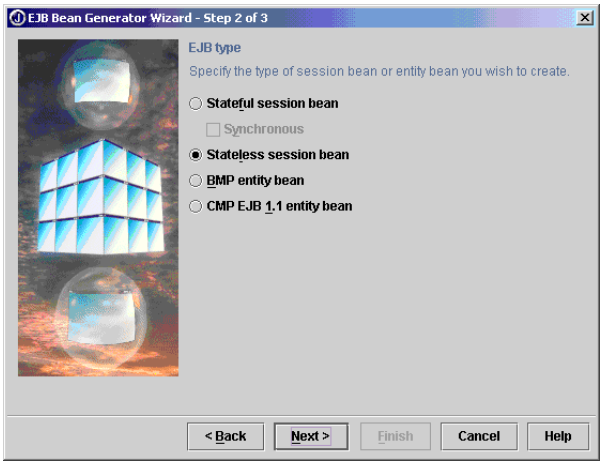
개발자들 중에는 엔터프라이즈 빈을 개발할 때 원격 인터페이스부터 디자인하는 사람도 있습니다. 이 방법을 선호한다면 EJB Been Generator를 사용하여 기존 원격 인터페이스에서 뼈대 빈 클래스를 생성할 수 있습니다.

다음과 같은 방법으로 원격 인터페이스에서 빈 클래스를 생성합니다.

1. 에디터에 원격 인터페이스를 표시합니다.
2. Wizards|EJB|EJB Bean Generator를 선택하여 EJB Bean Generator 마법사를 표시합니다.

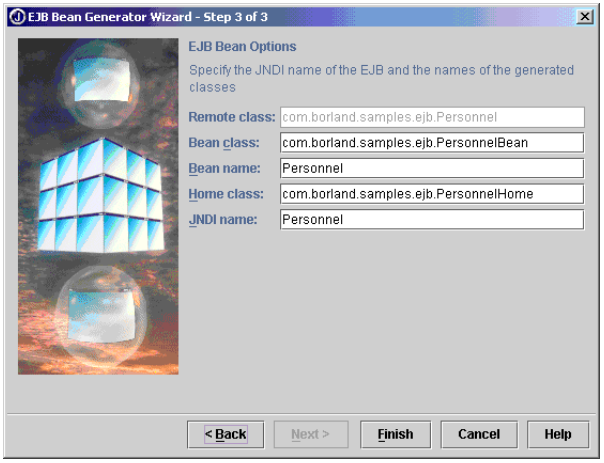


3 빈이 포함될 EJB 그룹을 선택하고 나서 Next를 클릭합니다.



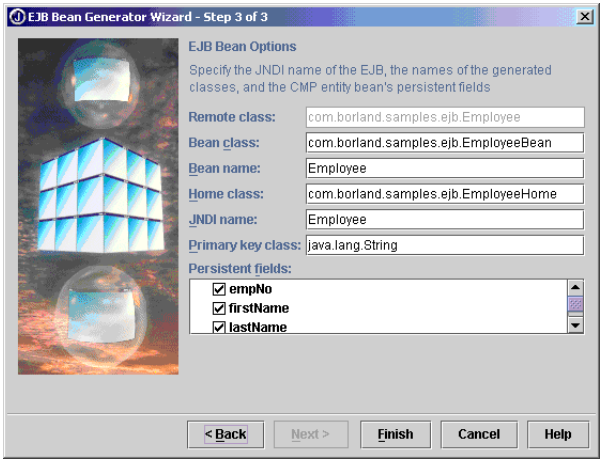
4 생성하려는 EJB 타입을 선택하고 나서 Next를 클릭합니다.

세션 빈 옵션 중 하나를 선택했다면 다음과 같은 페이지가 나타납니다.



- EJB Bean 옵션인 Bean Class, Bean Name, Home Interface 및 JNDI Name을 지정합니다.

CMP 엔티티 빈 옵션 중 하나를 선택했다면 다음과 같은 화면이 나타납니다.



- EJB Bean 옵션인 Bean Class, Bean Name, Home Interface, JNDI Name, Primary Key Class 및 지속적으로 유지하려는 필드를 지정합니다.

5 Finish를 선택합니다.

EJB Bean Generator는 사용자가 지정했던 뼈대 빈 클래스를 생성하며 이 클래스는 원격 인터페이스에 있는 메소드를 포함합니다 생성된 빈 클래스에서 이 메소드들은 사용자에게 구현을 완성하라고 알려주는 주석을 포함합니다. 원하는 대로 메소드를 구현하려면 메소드에 코드를 추가해야 합니다.

또한 EJB Bean Generator는 아직 존재하지 않는 홈 인터페이스를 생성합니다. 홈 인터페이스가 존재하는 경우 EJB Bean Generator는 그 홈 인터페이스를 겹쳐쓸 것인지 물은 후 사용자 대답에 따라 반응합니다.

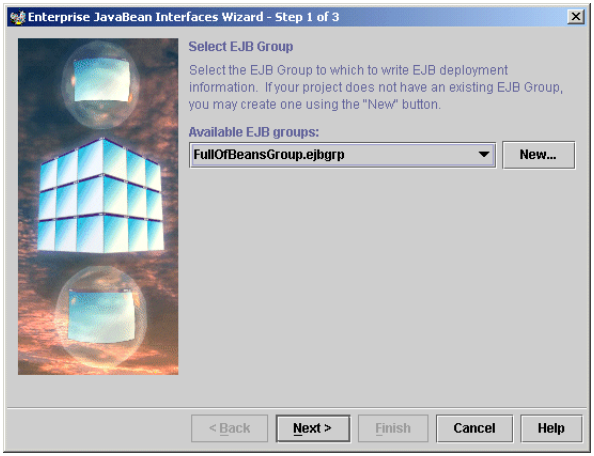
기존 빈에 대해 홈 및 원격 인터페이스 생성

빈 클래스는 있지만 필요로 하는 홈 및 원격 인터페이스가 없다면 EJB Interface Generator 마법사를 사용하여 인터페이스를 생성할 수 있습니다. 또한 빈의 소스 코드를 많이 변경해서 인터페이스에 변경 사항을 반영하려는 경우에도 이 마법사를 사용할 수 있습니다. EJB Interface Generator를 사용하면 수정된 빈 클래스 소스 코드를 기반으로 하여 새 인터페이스를 다시 생성할 수 있습니다.

다음과 같은 방법으로 EJB Interface Generator 마법사를 사용합니다.

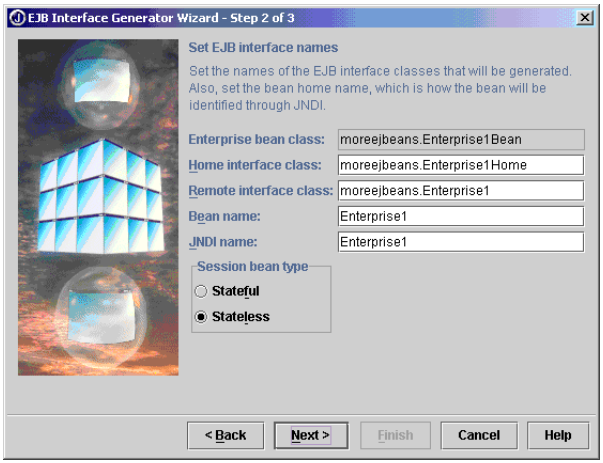
1 코드 에디터에서 빈 클래스의 소스 코드를 엽니다.

2 Wizards|EJB|EJB Interface Generator를 선택합니다.

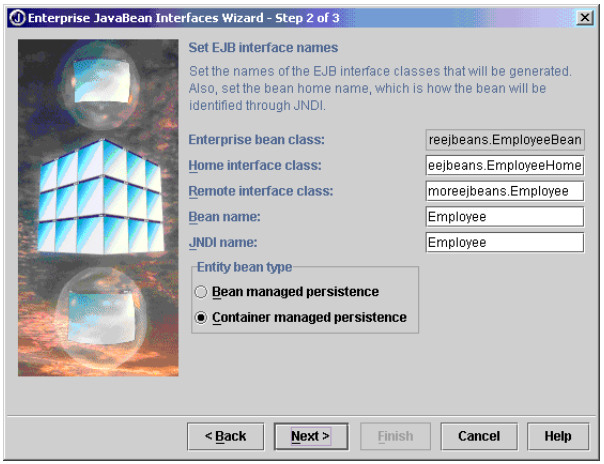


3 빈이 속해 있는 EJB 그룹을 선택하고 나서 Next를 클릭합니다.

빈이 세션 빈이면 다음과 같은 페이지가 나타납니다.



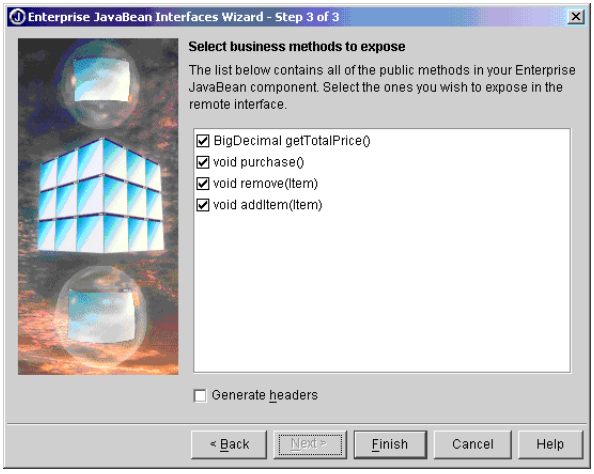
빈이 엔티티 빈이면 다음과 같은 페이지가 나타납니다.



4 기본 이름을 그대로 사용하거나 새 이름을 입력합니다.

5 엔터프라이즈 빈이 세션 빈이면 Stateless 또는 Stateful 옵션을 선택합니다. 엔터프라이즈 빈이 엔티티 빈이면 Bean Managed Persistence 또는 Container Managed Persistence를 선택합니다.

6 Next를 클릭하면 Step 3 화면에서 빈 메소드를 표시합니다.



7 원격 인터페이스에 표시하려는 메소드는 선택된 상태로 두고 표시하지 않을 메소드는 선택을 해제하십시오.

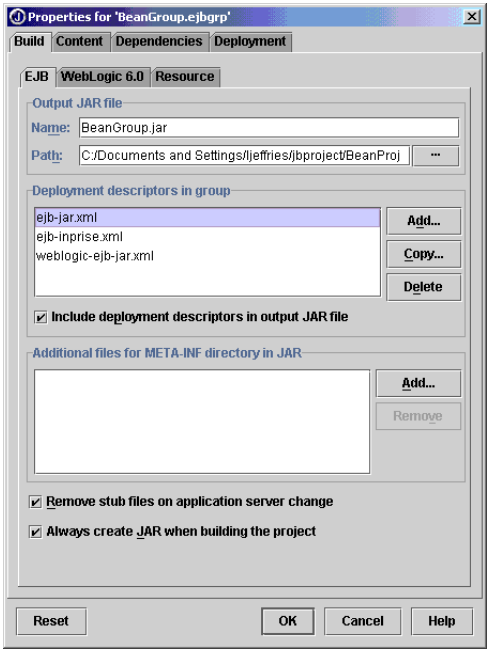
8 Finish를 선택합니다.

빈 컴파일

엔터프라이즈 빈, 인터페이스 및 지원 클래스를 작성하여 저장했으면 컴파일 준비가 거의 다 된 것입니다. 컴파일을 수행하기 전에 Borland AppServer 또는 Inprise Application Server 4.1*만*을 대상으로 하여 지역적으로 빈을 테스트하려면 클래스 클라이언트 스텁을 생성하여 클래스 경로에 추가해야 합니다. 이 작업을 수행하려면 홈 인터페이스의 build 속성을 변경하고 나서 컴파일하면 됩니다.

다음과 같은 방법으로 EJB 그룹의 각 빈에 대해 클라이언트 스텁을 생성하고 클래스 경로에 추가하도록 EJB 그룹의 build 속성을 변경합니다(이 단계는 옵션입니다).

1. 프로젝트 창에서 EJB 그룹을 마우스 오른쪽 버튼으로 클릭한 다음 Properties를 선택합니다.
2. Build 탭을 선택합니다.
3. EJB 탭을 선택합니다.



4. 원하는 대로 build 속성을 편집합니다.

   출력 JAR 파일의 이름과 파일이 생성되는 위치를 변경할 수 있습니다.

   JBuilder의 Deployment Descriptor 에디터가 아닌 다른 툴을 사용하여 Deployment Descriptor를 편집하려는 경우 선택한 툴을 사용하여 Deployment Descriptor를 편집합니다.

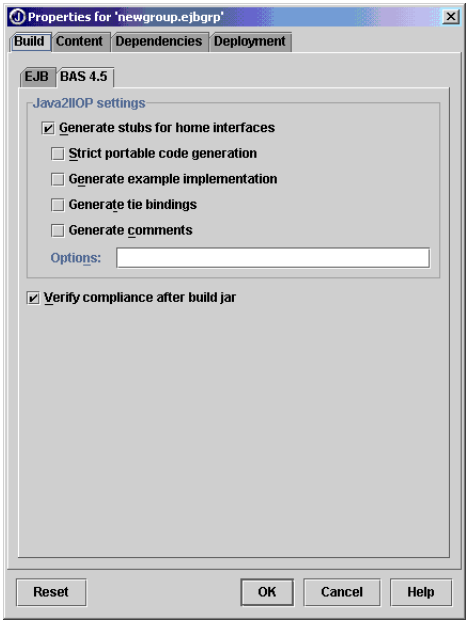
또한 EJB 그룹에 Deployment Descriptor를 삽입하고 다른 곳에 Deployment Descriptor를 복사할 수도 있습니다. 또한 Deployment Descriptor를 삭제할 수도 있습니다.

추가 파일이 JAR 파일에 추가되도록 지정하려면 Add 버튼을 클릭한 다음 파일의 위치를 지정합니다. 예를 들어, 프로젝트에 새 클래스를 추가한 다음 JAR 파일에 포함시키려는 경우 이 작업을 수행해야 합니다. 또는 JBuilder 외부에서 Deployment Descriptor를 편집한 경우 여기에 이러한 Deployment Descriptor를 추가하고 Include Deployment Descriptors In Output JAR File 옵션의 선택을 해제할 수 있습니다. Deployment Descriptors In Group 목록에 표시되는 Deployment Descriptor는 JAR에 추가되지 않으며 Additional Files For META-INF In JAR 목록에 지정된 Deployment Descriptor는 JAR에 추가됩니다.

다른 애플리케이션 서버를 대상으로 하는 경우 새 애플리케이션 서버를 선택할 때 Remove Stubs Files On Application Server Change 옵션을 사용하여 이전 애플리케이션 서버에서 사용했던 클라이언트 스텁을 제거할 수 있습니다.

기본적으로 Always Create JAR When Building 옵션이 사용됩니다. 이 옵션의 선택을 해제하면 테스트 시작이 준비될 때까지 JAR 파일의 빌드 작업을 연기할 수 있습니다.

5 대상으로 하는 애플리케이션 서버의 탭에서 OK를 클릭합니다. 예를 들어, 다음 그림은 BAS 4.5 탭이 선택된 화면입니다.



6 원하는 빌드 옵션을 지정합니다. 사용 가능한 build 옵션에 대한 자세한 내용을 보려면 Help 버튼을 클릭하십시오.

또한 전체 EJB 그룹 대신 각 빈에 대한 build 속성을 수정할 수도 있습니다.

1. 빈의 홈 인터페이스를 마우스 오른쪽 버튼으로 클릭한 후 Properties를 선택합니다.
2. Build 탭을 클릭합니다.
3. VisiBroker 탭을 클릭합니다.
4. Generate IIOP 체크 박스에 선택 표시를 하고 나서 원하는 다른 모든 Java2IIOP 옵션을 선택합니다.
5. OK를 클릭합니다.

프로젝트에 있는 모든 클래스를 컴파일하려면 프로젝트 파일 (<project>.jpx)을 마우스 오른쪽 버튼으로 클릭하고 나서 Make를 선택하거나 간단히 Project|Make Project를 선택합니다.

컴파일 작업 중에 Deployment Descriptor를 사용할 수 없게 만드는 오류가 있을 경우 JBuilder는 Deployment Descriptor에서 그 오류를 찾아낼 것입니다. 이러한 일이 발생하면 Deployment Descriptor 에디터에서 빈을 확인하라는 메시지가 메시지 창에 표시될 것입니다. Deployment Descriptor 확인에 대한 자세한 내용은 14- 27페이지의 "Descriptor 내용 확인"을 참조하십시오.

WebLogic 사용자용. WebLogic Server를 대상으로 할 경우 C:/Documents와 Settings/jbprojects처럼 임시 디렉토리 또는 클래스 경로에 공백이 포함된 경우 빌드 작업 시 오류가 발생할 것입니다.

클라이언트 스텁을 생성하기로 한 경우 프로젝트 창의 홈 인터페이스 노드의 아이콘을 클릭하여 확장하면 이 노드 아래에 몇 개의 파일이 표시됩니다. 생성된 파일들은 필요한 클라이언트 스텁으로서 EJB 작업을 생성하는 helper 클래스입니다.

Deployment Descriptor 편집

EJB 1.1 사양을 따르는 각 엔터프라이즈 빈에서는 Deployment Descriptor 항목이 XML 형식이어야 합니다. JBuilder 마법사를 사용하여 하나 이상의 엔터프라이즈 빈을 생성했으면 하나 이상의 Deployment Descriptor도 만들어졌습니다.

프로젝트를 컴파일할 때 JBuilder는 구성된 이름에 기반으로 하는 JAR 파일을 생성하여 프로젝트 창에 그룹 아래 노드로 표시합니다.

또한 전체 프로젝트를 컴파일하지 않고 JAR 파일만을 컴파일할 수도 있습니다. 프로젝트 창에서 EJB 그룹 노드를 오른쪽 버튼으로 클릭하고 나서 Make를 선택하여 EJB 그룹 노드를 컴파일합니다. Make를 선택하기 전에 build 속성을 수정하려면 Make를 선택하여 JAR 파일을 생성하기 전에 동일한 팝업 메뉴에 있는 Properties 메뉴 항목을 선택한 후 Build Properties 대화 상자의 내용을 수정합니다.

JAR 파일에는 모든 Deployment Descriptor가 들어 있습니다. WebSphere를 제외한 나머지 Deployment Descriptor는 모두 XML 파일

입니다. WebSphere는 각 빈에 대해 .ser 파일을 사용합니다. 각 JAR 파일은 하나 이상의 Deployment Descriptor를 가질 수 있습니다.

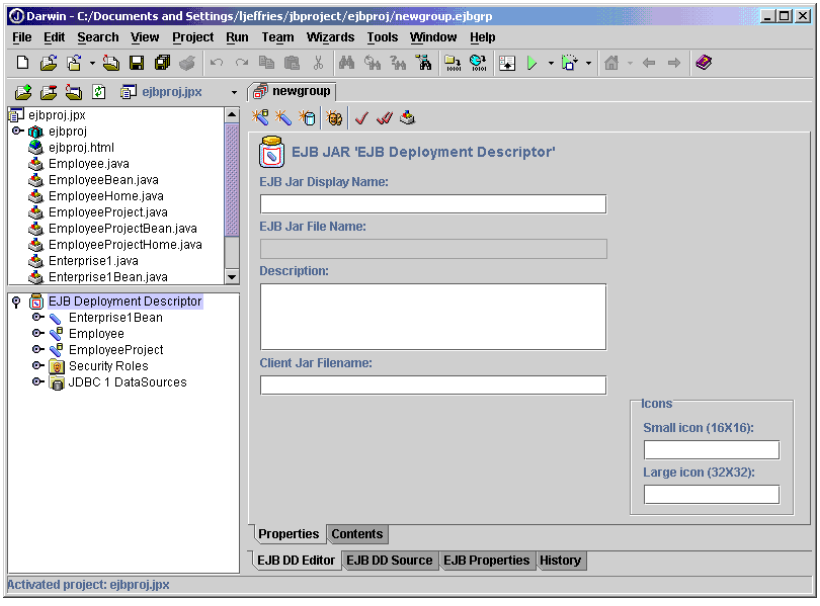
JBuilder는 다수의 애플리케이션 서버 중 하나를 대상으로 합니다. 대상 애플리케이션 서버는 생성된 JAR 파일에 있는 Deployment Descriptor의 수를 확인합니다. 모든 JAR 파일에는 ejb-jar.xml이 있는데(WebSphere 3.5를 대상으로 하는 경우 제외) 이 파일은 모든 애플리케이션 서버 간 공통 그룹에 있는 빈에 대해 배포 속성을 설명합니다. ejb-jar.xml은 EJB 1.1 호환 Deployment Descriptor입니다. 대상 애플리케이션 서버로 EJB 1.1을 선택한 경우 JAR 파일에는 이 Deployment Descriptor만 들어 있을 것입니다.

공급 업체가 Borland가 아닌 다른 업체인 경우라도 공급 업체 특정 정보는 ejb-inprise.xml 파일에 유지됩니다. 컴파일 시 공급업체 특정 XML 파일은 이 정보에서 생성됩니다. 또한 Deployment Descriptor 에디터의 Source 탭을 클릭해도 이 정보가 생성됩니다.

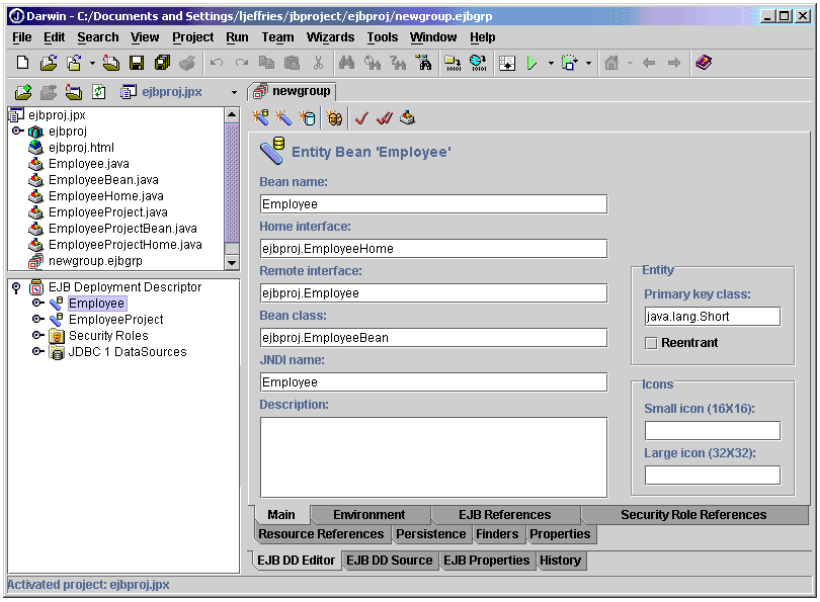
대상이 Borland AppServer인 경우 추가 애플리케이션 서버 특정 XML 파일은 ejb-inprise.xml 단 하나 뿐입니다. 대상이 WebLogic Server인 경우 생성된 JAR 파일에 weblogic-ejb-jar.xml 파일이 들어 있고 또 CMP를 가진 각 엔티티 빈마다 추가 XML 파일 한 개가 더 들어 있습니다. 대상이 WebSphere인 경우 생성된 JAR 파일에는 각 빈마다 .ser 파일이 들어 있을 것입니다.

JBuilder의 Deployment Descriptor 에디터는 기존 Deployment Descriptor를 수정하는 방법을 제공합니다.

Deployment Descriptor 에디터를 표시하려면 프로젝트 창에서 EJB 그룹을 더블 클릭합니다. Deployment Descriptor 에디터가 나타납니다. EJB 그룹 트리가 구조 창에 나타납니다.



Deployment Descriptor 에디터에서 엔터프라이즈 빈에 관한 정보를 보려면 구조 창에서 해당 빈을 클릭합니다. 또는 구조 창에서 EJB Deployment Descriptor 노드를 선택하는 경우 Contents 탭을 클릭하여 EJB 그룹의 내용을 표시하고 확인할 빈의 아이콘을 더블 클릭합니다. Deployment Descriptor 에디터에서 빈이 선택되면 에디터 안에 몇 개의 탭이 나타납니다. 이러한 탭을 사용하면 Deployment Descriptor 정보를 편집하는 패널로 이동할 수 있습니다.



대상 애플리케이션 서버가 Borland AppServer 또는 Inprise Application Server인 경우 EJB Properties 탭이 나타나지 않습니다. EJB Properties 페이지에서는 WebLogic 또는 WebSphere 애플리케이션 서버의 값을 변경할 수 있습니다.

Deployment Descriptor 에디터 사용에 대한 자세한 내용은 Chapter 14 "Deployment Descriptor 에디터 사용"을 참조하십시오.

디스크립터 편집을 완료했으면 파일을 확인하여 디스크립터 정보가 올바른지, 필요한 빈 클래스 파일이 있는지 등을 알아볼 수 있습니다.

디스크립터 정보를 확인하려면 Deployment Descriptor 에디터의 툴바에서 Verify 버튼을 클릭합니다.

다음 사항을 확인합니다.

- 디스크립터가 EJB 1.1 사양에 부합되는지 확인합니다.
- Deployment Descriptor에서 참조하는 클래스가 EJB 1.1 사양에 부합되는지 확인합니다.

확인 작업이 실패하면 Log 창에 실패를 설명하는 메시지가 하나 이상 표시됩니다.

Chapter

11

기존의 데이터베이스 테이블에서 엔티티 빈 생성

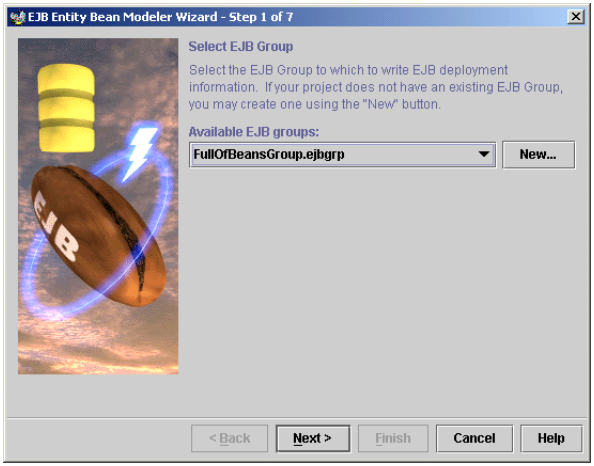
엔티티 빈으로 만들려는 데이터가 이미 데이터베이스에 있는 경우가 종종 있습니다. JBuilder의 Entity Modeler를 사용하면 이러한 엔티티 빈을 생성할 수 있습니다.

EJB Entity Bean Modeler를 사용하여 엔티티 빈 생성

EJB Entity Modeler 마법사는 JDBC를 통해 액세스 가능한 모든 데이터베이스에 있는 기존의 테이블에 기반하여 엔티티 빈을 생성합니다. 마법사를 사용하여 여러 개의 엔티티 빈을 한 번에 생성하고 이러한 빈 사이의 관계를 지정할 수 있습니다.

EJB Entity Bean Modeler를 사용하여 엔티티 빈, 기본 키, 홈 및 원격 인터페이스, Deployment Descriptor의 해당 항목을 구성하는 코드를 생성했으면, Bean 디자이너, Deployment Descriptor 에디터, JBuilder 코드 에디터와 같은 다른 JBuilder 툴을 사용하여 결과를 수정할 수 있습니다.

EJB Entity Modeler를 표시하려면 File|New를 선택한 다음 Enterprise 탭을 클릭하여 EJB Entity Bean Modeler를 선택합니다. 적어도 하나의 EJB 그룹이 프로젝트에 정의되어 있으면 Entity Bean Modeler가 나타납니다.

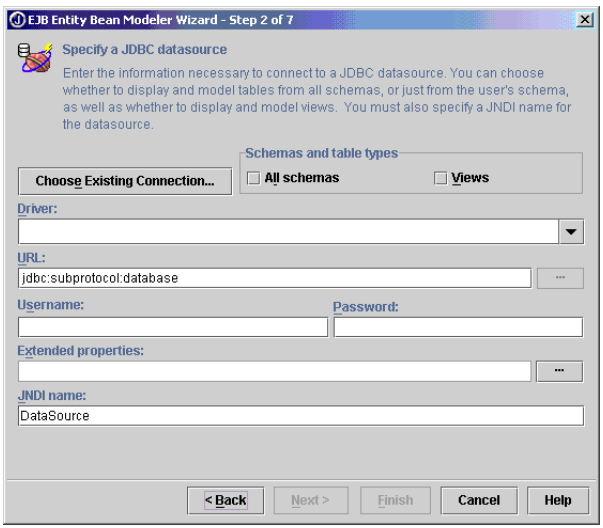


JBuilder로 개발한 모든 엔터프라이즈 빈은 EJB 그룹에 속해야 합니다. 현재 프로젝트에 EJB 그룹이 하나도 없는 경우 New 버튼을 클릭하여 Empty EJB Group 마법사를 시작합니다. Empty EJB Group 마법사를 사용하여 EJB 그룹을 생성한 경우 Entity Bean Modeler가 나타납니다.

기존의 데이터베이스 테이블에서 하나 이상의 빈을 생성하려면 다음과 같은 단계를 따릅니다.

1 빈을 저장할 EJB 그룹을 선택하고 Next를 선택하여 2 단계로 갑니다.

선택한 EJB 그룹은 배포 정보가 작성되는 위치를 결정하는 데 사용됩니다.



2 JDBC 데이터 소스를 지정합니다.

JDBC 데이터 소스를 연결하는 데 필요한 정보를 입력합니다.

기존의 연결을 사용하려면 Choose Existing Connection 버튼을 클릭하고 연결을 선택합니다. 그러면 암호를 제외한 이 페이지에 필요한 기타 정보가 자동으로 채워집니다. 연결 시 암호를 요구하는 경우에는 직접 암호를 입력해야 합니다.

기존 연결이 없거나 다른 연결을 생성하려는 경우 Driver 드롭다운 목록에서 드라이버를 선택하고 URL을 지정합니다.

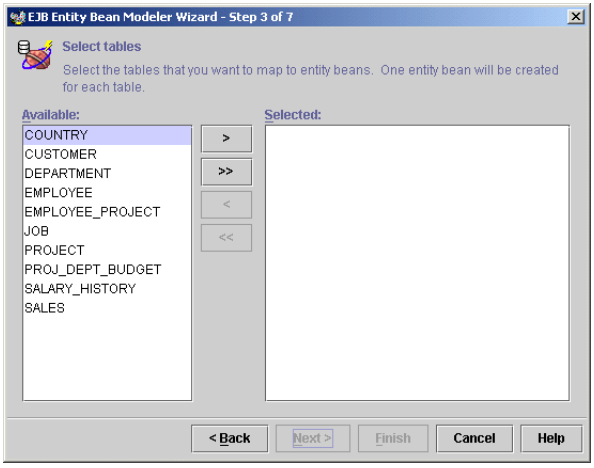
데이터 소스의 Username을 지정하고 암호가 필요한 경우 암호를 입력합니다. 필요한 다른 속성을 선택합니다. 마지막으로 데이터 소스의 JNDI name을 지정합니다.

3 사용하려는 Schemas And Table Types 옵션을 지정합니다.

All Schemas 옵션을 선택 표시한 경우 EJB Entity Bean Modeler는 사용자가 연결 권한을 가지고 있는 모든 스키마를 로드합니다. All Schemas를 선택 해제한 경우 username과 동일한 이름의 스키마는 연결 설정 및 데이터 로딩에 걸리는 시간을 줄일 수 있습니다.

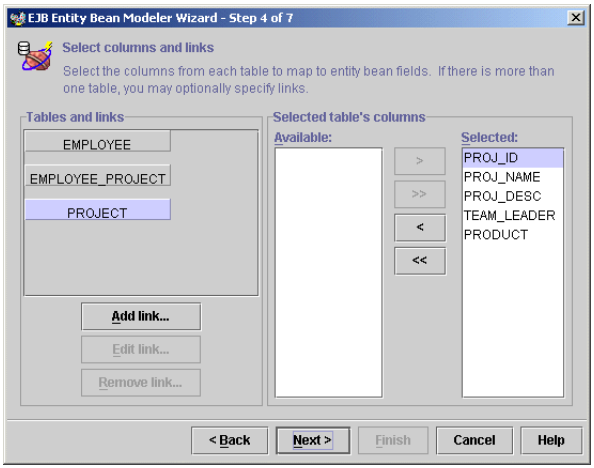
뷰를 EJB Entity Bean Modeler에 로드하려는 경우 Views 옵션을 선택 표시합니다. 뷰를 로드하지 않으려는 경우 Views 옵션을 선택 해제합니다.

EJB Entity Bean Modeler에서 지정된 데이터 소스로의 연결을 시도합니다. 연결이 성공한 경우에만 다음과 같은 페이지가 나타납니다.



4 엔티티 빈에 매핑하려는 테이블을 선택합니다.

선택한 각 테이블에 대해 하나의 엔티티 빈이 생성됩니다. Available 목록에서 사용하려는 테이블을 선택한 다음 > 및 >> 버튼을 사용하여 이를 Select 목록으로 옮깁니다. 모든 테이블을 선택한 경우 Next를 선택합니다.

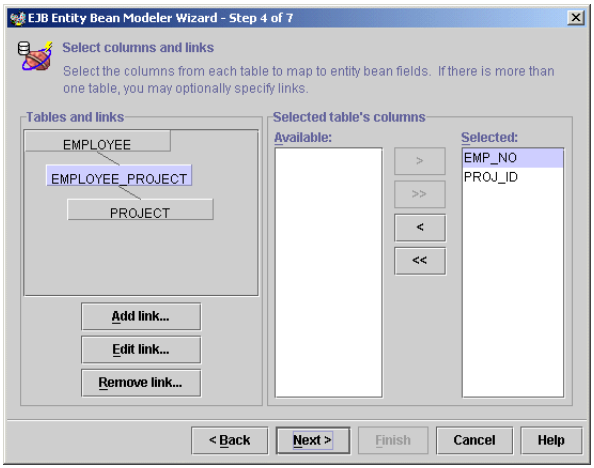


5 엔티티 빈 필드에 매핑할 각 테이블의 열을 선택하여 테이블 간에 설정할 관계를 지정합니다.

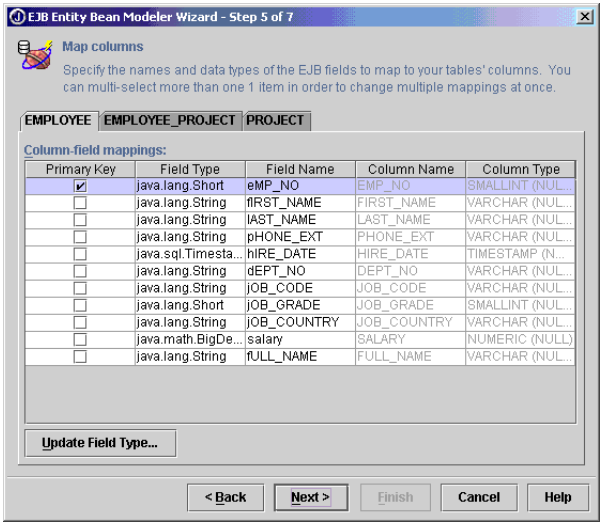
Tables and Links 섹션에 이전 단계에서 선택한 모든 테이블이 표시됩니다. 차례로 각 테이블을 클릭하여 선택한 다음 Selected Table's Columns 섹션을 사용하여 Available 목록과 Selected 목록 사이에서 테이블의 열을 이동합니다. 기본적으로 모든 테이블의 모든 열이 선택됩니다.

또한 왼쪽의 Tables and Links 박스에 있는 테이블 사이에서 마우스 포인터를 드래그하여 테이블 사이의 관계를 지정할 수 있습니다. 또는 Add Link 버튼을 사용하여 동일한 작업을 수행할 수 있습니다. 둘 중 하나의 방법을 사용하면 외래 키, 기본 키, 고유한 인덱스 및 두 테이블의 필드 이름과 타입에 기반하여 관계를 설정하는 대화 상자가 나타납니다. 제안된 관계를 적용하거나 수정하여 원하는 관계를 생성할 수 있습니다. 테이블 사이의 링크를 제거하려면 Remove Link를 선택합니다.

다음은 세 개의 테이블이 같이 연결된 예제입니다.



생성 중인 엔티티 빈의 필드에 매핑하려는 각 테이블의 모든 열을 선택했으면 Next를 선택합니다.



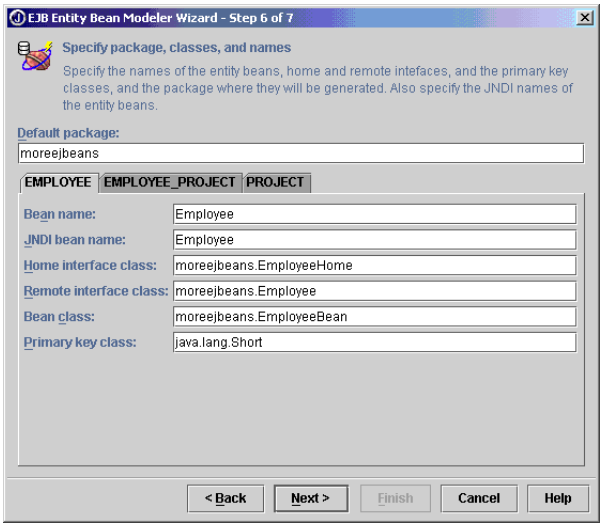
6 테이블의 열에 매핑하려는 엔티티 빈 필드의 이름 및 데이터 타입을 지정합니다.

해당 탭을 클릭하여 매핑 프로세스를 시작하려는 테이블을 선택합니다. 테이블의 각 열에 제안된 Field Name 및 Field Type이 나타납니다. 제안된 이름을 그대로 적용하거나 제안된 이름과 타입을 빈에서 사용하고자 하는 대로 편집합니다.

하나의 타입으로 되어 있는 여러 필드의 데이터 타입을 변경하려면 변경하려는 필드를 선택하고 Update Field Type을 선택합니다. 새 필드

타입을 입력할 수 있는 대화상자가 나타납니다. Apply 또는 OK를 선택하면 선택한 각 필드의 필드 타입이 변경됩니다.

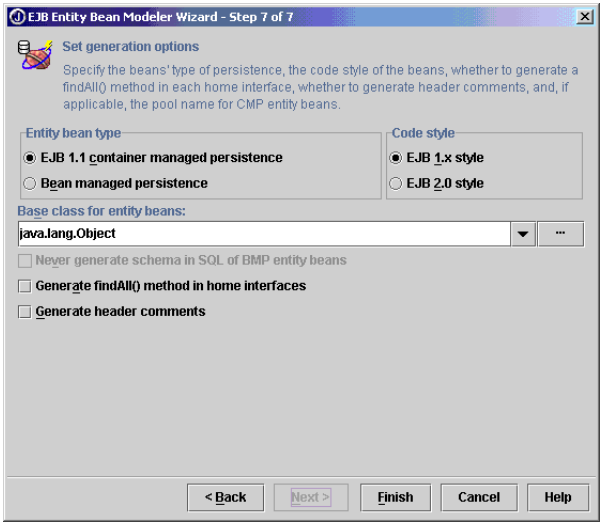
테이블에 이미 기본 키가 있는 경우, Map Columns 페이지가 처음 나타날 때 해당 필드 또는 필드의 집합이 선택됩니다. 기본 키가 없는 경우, Primary Key 열에 있는 해당 필드의 체크 박스를 선택 표시함으로써 기본 키를 구성하는 하나 이상의 필드를 선택해야 합니다. 선택한 모든 열을 각 테이블의 엔티티 빈에서 사용하려는 필드 이름 및 타입으로 매핑을 완료하면 Next를 선택합니다.



7 생성할 각 빈의 패키지, 클래스 및 인터페이스, JNDI 이름을 지정합니다.

각 테이블에 대해 JBuilder는 엔티티 빈의 이름, JNDI가 사용하는 이름, 홈 및 원격 인터페이스의 이름, 빈 클래스의 이름, 기본 키 클래스의 타입을 제안합니다. 이에 대한 다른 패키지를 지정할 수도 있으며 기본적으로는 프로젝트 패키지가 제안됩니다. 제안된 값을 그대로 적용하거나

원하는 대로 수정할 수 있습니다. 각 테이블의 정보 지정을 완료하면 Next를 선택합니다.



8 Container 관리 방식(Container-managed persistence, CMP)과 Bean 관리 방식(Bean-managed persistence, BMP) 중에서 엔티티 빈이 어떤 방식을 가질지 선택합니다.

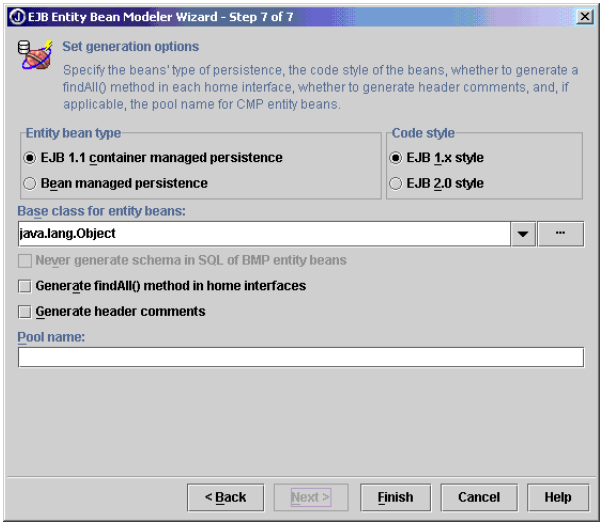
EJB 2.0을 대비하고 생성된 코드가 EJB 2.0 스타일을 따르도록 하려면 EJB 2.0 코드 스타일 옵션을 선택합니다. 이러한 옵션에 대한 자세한 내용을 보려면 EJB Entity Bean Modeler의 Help 버튼을 선택하십시오.

기본적으로 빈의 기본 클래스는 `java.lang.Object`입니다. 엔티티 빈의 기반으로서 다른 클래스를 사용하려면 Base Class For Entity Bean 필드를 사용하여 다른 클래스를 지정합니다.

엔티티 빈에서 데이터 집합의 모든 행을 반환할 수 있도록 하려는 경우 FindAll() Method In Home Interface 옵션을 선택 표시합니다. EJB Entity Bean Modeler는 빈의 홈 인터페이스에 `findAll()` 메소드를 둡니다. 또한 헤더 주석이 결과 파일에 표시되는지 여부를 선택할 수 있습니다.

이 옵션은 대상 애플리케이션 서버에 따라 이 화면에 제공됩니다. 예를 들어, WebSphere 3.5가 CMP를 지원하지 않는 경우 WebSphere 3.5가 대상 애플리케이션 서버라면 container-managed persistence 옵션이 나타나지 않습니다.

대상 애플리케이션 서버가 WebLogic Server이고 CMP를 갖는 엔티티 빈을 생성할 경우 이 페이지는 또한 Pool Name 필드를 갖게 되어 여기에 CMP WebLogic 빈에 대한 풀의 이름을 입력해야 합니다.



9 Finish를 선택합니다.

JBuilder는 각 테이블 및 모든 지원 클래스 인터페이스에 대한 엔티티 빈을 생성합니다. 이제 사용자는 원하는 비즈니스 로직을 빈에 추가하고 클라이언트가 원격 인터페이스에서 호출할 수 있도록 메소드를 정의할 수 있으며 빈을 컴파일하고 빈의 Deployment Descriptor를 편집할 수 있습니다.

Chapter

12

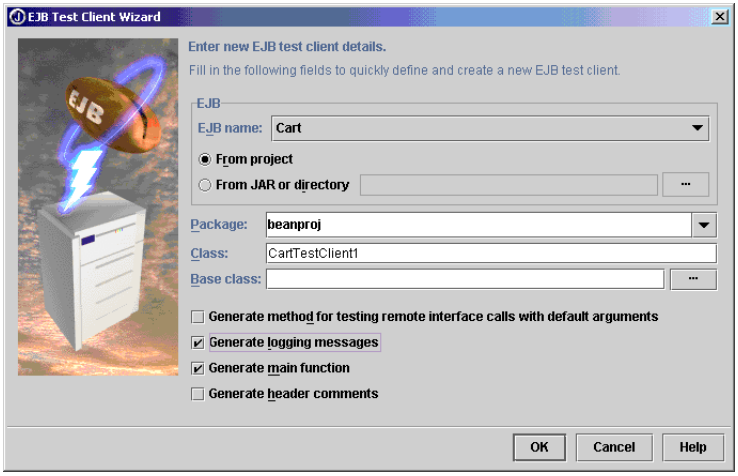
엔터프라이즈 빈 테스트

엔터프라이즈 빈 생성을 완료하면 JBuilder를 사용하여 빈의 기능을 테스트하는 클라이언트 애플리케이션 생성에 도움을 받을 수 있습니다.

테스트 클라이언트 생성

다음과 같은 방법으로 테스트 클라이언트 애플리케이션을 생성합니다.

1. 엔터프라이즈 빈의 EJB 그룹을 포함하는 프로젝트를 엽니다.
2. File|New를 선택하고 Enterprise 탭을 클릭한 다음 EJB Test Client를 더블 클릭합니다.



3. Select EJB 옵션 중 하나를 사용하고 빈을 지정하여 클라이언트를 생성하려는 빈을 선택합니다.
   - 빈이 현재 프로젝트에 있는 경우 From Project를 선택하고 드롭다운 목록에서 빈을 선택하여 지정합니다.
   - 빈이 현재 프로젝트에 없고 JAR 파일 또는 디렉토리에 있는 경우 From JAR Or Directory를 선택합니다. ... 버튼을 사용하여 JAR가 있는 위치로 이동하여 JAR를 선택한 다음 드롭다운 목록에서 사용하려는 빈을 선택합니다.
4. 패키지 목록에서 패키지 이름을 선택합니다. 현재 패키지는 기본값입니다.
5. 테스트 클라이언트 클래스의 이름을 입력하거나 기본 이름을 적용합니다.
6. 사용하려는 옵션을 선택합니다.
   - Generate Method For Testing Remote Interface Calls With Default Arguments

     기본 인수 값으로 원격 인터페이스 호출을 테스트하는 `testRemoteCallsWithDefaultArguments()` 메소드를 추가합니다. 예를 들면 문자열의 기본 인수는 "", 정수의 기본 인수는 0, 등입니다.
   - Generate Logging Messages

     클라이언트가 실행될 때 빈의 상태를 보고하는 메시지를 표시하는 코드를 추가합니다. 예를 들면, 빈 초기화가 시작되어 완료되고 다른 빈 초기화가 시작될 때 메시지가 표시됩니다. 이 옵션은 또한 홈 및 원격 인터페이스에 선언된 모든 메소드와 초기화 함수를 위해 랩퍼를 생성합니다. 마지막으로 메시지는 각 메소드 호출이 완료하는 데 걸리는 시간을 보고합니다.
   - Generate Main Function

     클라이언트에 main 함수를 추가합니다.
   - Generate Header Comments

     제목, 작성자 등과 같은 정보를 입력하는 데 사용할 수 있는 JavaDoc 헤더 주석을 클라이언트에 추가합니다.
7. OK를 선택합니다.

EJB Test Client 마법사는 엔터프라이즈 빈에 대한 참조를 생성하는 테스트 클라이언트를 생성합니다.

Generate Logging Messages 옵션을 선택한 경우 빈의 원격 인터페이스에 선언된 각 메소드에 대해 마법사는 또한 원격 메소드를 호출하는 메소드를 선언하고 구현합니다. 이러한 각 메소드는 원격 메소드 호출의 성공 및 원격 메소드 실행에 걸린 시간을 보고합니다.

생성된 테스트 클라이언트 애플리케이션을 사용하는 방법에는 여러 가지가 있습니다. `main()` 함수를 테스트 클라이언트 애플리케이션에 추가했다면 `main()` 함수에서 엔터프라이즈 빈의 메소드를 호출하는 코드를 작성할 수 있습니다. 이 작업은 먼저 create 또는 find 메소드를 호출하여 수행하는데 원격 참조가 반환되는 경우 빈의 비즈니스 메소드를 호출하는 원격 참조를 사용하여 수행합니다. 또는 마법사가 `main()` 함수에 클라이언트 객체를 선언했기 때문에 이 클라이언트 객체를 사용하여 빈의 원격 메소드를 호출하는 테스트 클라이언트 애플리케이션에 선언된 메소드를 호출할 수 있습니다.

Generate Method For Testing Remote Interface Calls With Default Arguments 옵션을 선택한 경우 클라이언트 클래스는 현재 `testRemoteCallsWithDefaultArguments()` 메소드를 포함합니다. 로깅 옵션을 선택한 경우 이 메소드는 로깅 옵션에서 생성된 원격 메소드 랩퍼를 호출합니다. 각각의 원격 메소드를 테스트하기 위해 클라이언트 클래스의 `create()` 메소드 또는 `findByXXX()` 메소드 중 하나에서 원격 인터페이스 참조를 생성한 후 `testRemoteCallsWithDefaultArguments()` 메소드를 호출할 수 있습니다.

로깅 옵션을 선택하지 않은 경우 `testRemoteCallsWithDeafultArguments()` 메소드는 매개변수로 전달되는 원격 인터페이스를 필요로 합니다. 그런 다음 홈 참조의 `create()` 메소드 또는 `findByXXX()` 메소드 중 하나에서 원격 인터페이스 참조를 만들어야 합니다. 그런 다음 원격 참조를 인수로 전달하는 `testRemoteCallsWithDefaultArguments()` 메소드를 호출하는 코드를 클라이언트 클래스에 추가합니다.

다른 클래스로부터 각각의 비즈니스 메소드를 호출하는 로직의 작성을 선호하는 경우 테스트 클라이언트 애플리케이션 인스턴스를 생성하여 사용하도록 선택할 수 있습니다. 12- 3페이지의 "테스트 클라이언트 애플리케이션 사용"을 참조하십시오.

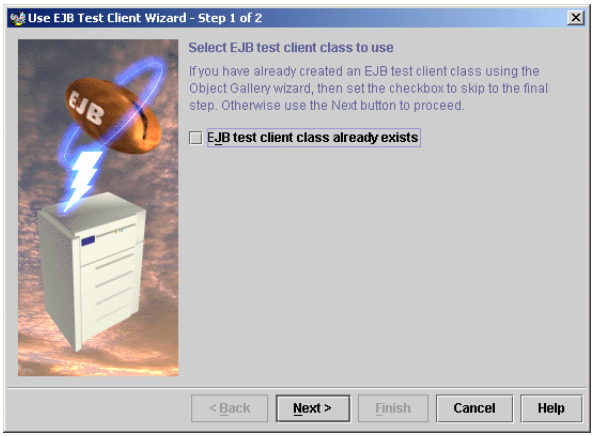
테스트 클라이언트 애플리케이션을 컴파일합니다.

테스트 클라이언트 애플리케이션 사용

클래스에 테스트 클라이언트 클래스의 선언을 빠르게 추가할 수 있습니다.

1 에디터에 나타내려는 선언이 들어 있는 클래스를 표시합니다.

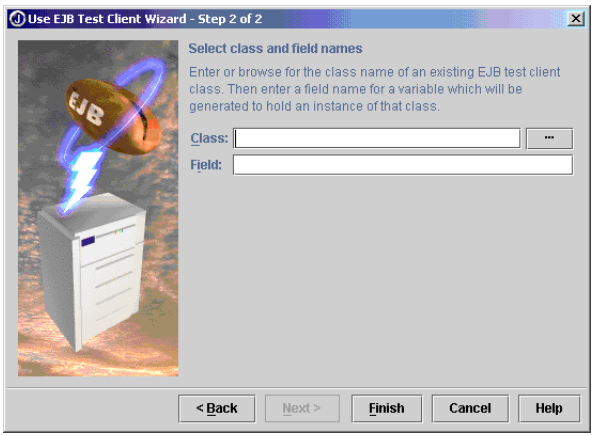
2 Wizards|EJB|Use EJB Test Client를 선택합니다.



3 테스트 클라이언트가 이미 있는 경우 EJB Test Client Class Already Exists 옵션을 선택 표시합니다.

이 옵션이 선택되지 않은 경우 Next를 클릭하면 EJB Test Client 마법사가 시작됩니다. 이 옵션을 사용해 온 경우 Use EJB Test Client 마법사가 재시작됩니다.

4 2 단계로 가려면 Next를 클릭합니다.



5 Class 필드의 경우 사용하려는 테스트 클라이언트 클래스가 있는 위치로 이동합니다.

6 Field 필드에서 테스트 클라이언트 클래스의 인스턴스를 저장하는 변수 이름을 지정하거나 마법사가 제공하는 기본값을 적용합니다.

7 Finish를 선택합니다.

이 마법사에서는 지정한 테스트 클라이언트 애플리케이션의 선언을 예를 들어 다음과 같은 클래스에 추가합니다.

```
EmployeeTestClient1 employeeTestClient1 = new EmployeeTestClient1();
```

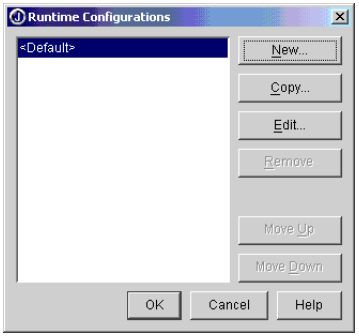
이제 테스트 클라이언트 애플리케이션에 선언된 메소드를 호출할 준비가 되었습니다.

엔터프라이즈 빈 테스트

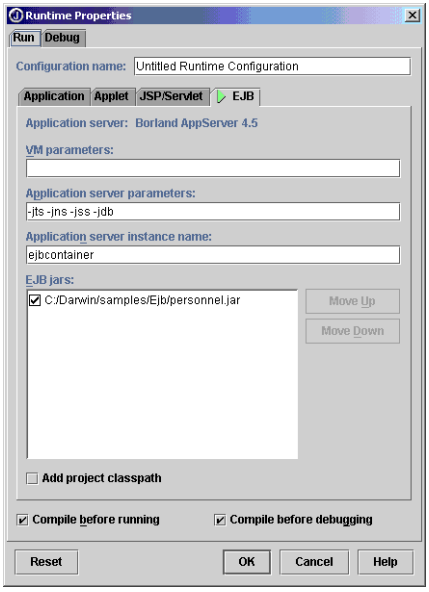
클라이언트 테스트 애플리케이션을 생성했으면 컨테이너를 시작하여 클라이언트 애플리케이션을 실행할 준비가 되었습니다. Server 및 Client의 두 가지 런타임 구성을 생성합니다.

다음과 같은 방법으로 Server 구성을 생성합니다.

1 Run|Configurations를 선택합니다.



2 New 버튼을 클릭한 다음 EJB 탭을 클릭합니다.



3 Configuration Name 필드에 `Server`를 입력합니다.

4 서버를 실행하는 데 필요한 Application Server Parameters 및 Application Server Instance Name을 입력합니다. 9- 4페이지의 "애플리케이션 서버 선택"에 설명되어 있는 대로 대상 애플리케이션 서버를 선택한 경우 기본 Application Server Parameters 및 Application Instance Name이 이미 입력되어 있습니다. 대상 애플리케이션 서버를 선택하지 않으면 기본적으로 애플리케이션 서버는 BAS 4.5가 선택됩니다.

5 테스트하려는 빈을 포함하는 JAR 파일을 EJB JAR 목록에서 선택합니다. 목록에 파일이 하나만 있으면 이미 그 파일이 선택됩니다. 나열된 JAR 파일은 프로젝트의 EJB 그룹에서 가져옵니다.

이 애플리케이션 서버는 실행 서버로의 배포를 지원하지 않으므로 EJB JAR의 목록은 WebSphere 3.5에서 사용할 수 없게 됩니다.

WebLogic Server 6.0의 경우 JAR는 <WLServer6.0 홈>₩config₩<도메인 이름>₩애플리케이션 디렉토리로 복사됩니다.

6 OK를 클릭합니다.

다음과 같은 방법으로 Client 구성을 생성합니다.

1 Run|Configurations를 선택합니다.

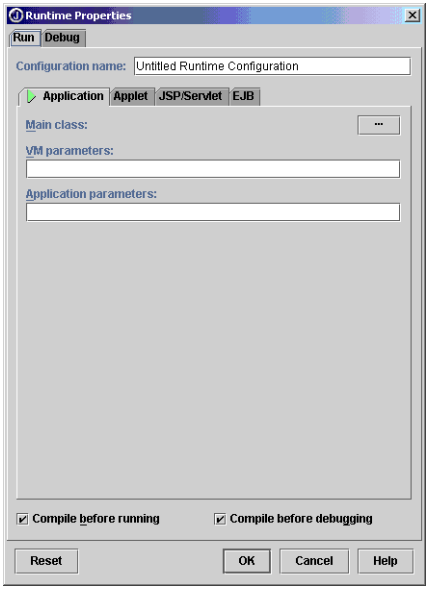
2 New 버튼을 클릭한 다음 Application 탭을 클릭합니다.

3 Configuration Name 필드에 `Client`를 입력합니다.

4 Main Class 필드 옆의 ... 버튼을 클릭하여 생성한 테스트 클라이언트 애플리케이션이 있는 위치로 이동하거나 테스트 클라이언트의 메소드를 호출하는 `main()` 함수가 들어 있는 애플리케이션이 있는 위치로 이동합니다.

5 OK를 두 번 클릭합니다.

이제 컨테이너를 시작할 준비가 되었습니다. JBuilder 툴바의 Run 버튼 옆에 있는 드롭다운 목록에서 Server 실행 구성을 선택합니다.



컨테이너가 시작됩니다. 시작 프로세스가 실행되는 동안 잠시 기다리십시오. 메시지 창에서 시작 프로세스의 진행을 볼 수 있습니다. 발생하는 오류도 모두 여기에 나타납니다.

다음으로 Client 실행 구성을 선택하여 클라이언트 애플리케이션을 실행합니다. 메시지 창에 나타나는 메시지는 클라이언트 애플리케이션 실행의 성공 또는 실패를 보고합니다.

빈을 테스트하는 다른 방법은 프로젝트 창의 EJB 그룹을 마우스 오른쪽 버튼으로 클릭하고 Run을 선택하는 것입니다.

JBuilder를 사용하여 다른 Java 코드와 마찬가지로 엔터프라이즈 빈 또는 클라이언트를 디버그할 수 있습니다. 디버깅에 관한 내용은 *JBuilder를 이용한 애플리케이션 구축*의 "Java 프로그램 디버깅"을 참조하십시오.

Chapter

13

엔터프라이즈 빈 배포

애플리케이션 서버에 대한 엔터프라이즈 빈 배포는 일반적으로 다음과 같은 단계를 거칩니다.

1. Sun의 EJB 1.1 사양에 따라 Deployment Descriptor XML 기반 파일 생성. (WebSphere 3.5는 여기에서 제외됩니다. WebSphere 3.5는 EJB 1.0 사양에 따르므로 XML 기반 Deployment Descriptor를 사용하지 않습니다.}

   JBuilder의 EJB 마법사를 사용하여 빈을 만들 경우 Deployment Descriptor가 동시에 만들어 집니다.

2. 필요할 경우 Deployment Descriptor를 편집합니다.

   JBuilder의 Deployment Descriptor 에디터를 사용하여 JBuilder가 생성한 Deployment Descriptor를 편집할 수 있습니다.

3. Deployment Descriptor 및 EJB(빈 클래스, 원격 인터페이스, 홈 인터페이스, 스텁 및 뼈대, EJB가 엔티티 빈일 경우의 기본 키 클래스 및 그 밖의 연관 있는 클래스)를 작동시키는 데 필요한 모든 클래스를 포함하는 EJB JAR 파일 생성.

   JBuilder 개발 환경을 사용하여 EJB 그룹을 컴파일 할 경우 적절한 JAR 파일이 생성됩니다.

4. EJB 컨테이너에 EJB 배포.

   JBuilder에는 Borland 엔터프라이즈 빈의 배포 과정을 단순화하는 EJB Deployment 마법사가 있습니다. WebLogic 또는 WebSphere가 대상 애플리케이션 서버일 경우, Tools|EJB Deployment를 선택하면 서버에 맞는 Deploy Settings 대화 상자가 나타납니다. 요구 사항에 맞게 설정하고 OK를 선택하여 대화 상자를 닫으면 지정한 대로 EJB가 배포됩니다.

Deployment Descriptor 파일 생성

JBuilder의 EJB 툴을 사용하여 엔터프라이즈 빈을 생성하는 경우, JBuilder는 동시에 Deployment Descriptor를 생성합니다. 그런 다음 Deployment Descriptor 에디터를 사용하여 Deployment Descriptor에 추가적인 정보를 추가하고 속성을 수정할 수 있습니다.

EJB 1.1 사양을 따르는 각 Deployment Descriptor는 다음과 같은 조건을 갖습니다. (WebSphere 3.5에서 사용하는 디스크립터는 제외됩니다.)

- XML을 기반으로 해야 하며 XML의 규칙에 따라야 합니다.
- EJB 1.1 사양의 DTD에 대해 유효해야 합니다.
- DTD에 지정된 의미 규칙을 따릅니다.
- 다음 문장을 사용하여 DTD를 참조합니다.

```
<DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dts/ejb-jar_1_1.dtd">
```

JBuilder의 EJB 툴을 사용하여 배포 디스크립트를 생성 및 편집할 경우, XML을 배운다든지 Sun의 DTD에 지정되어 있는 의미 규칙을 따르는 것에 대해 염려할 필요가 없습니다. Deployment Descriptor 에디터는 사용자가 입력하고 편집하는 데이터에 이러한 규칙들을 적용합니다. Deployment Descriptor 에디터를 사용하여 정보를 입력하는 경우 어떠한 데이터가 필요한지 사용자에게 알려줍니다. JBuilder 툴은 `ejb-inprise.xml` 파일에 Borland 특정 확장자를 자동으로 설정합니다. Deployment Descriptor 에디터에 대한 자세한 내용은 Chapter 14 "Deployment Descriptor 에디터 사용"을 참조하십시오.

Deployment Descriptor의 역할

Deployment Descriptor의 역할은 특정 JAR파일에 번들되고 배포될 각 EJB에 대한 정보를 제공하는 것입니다. 이것은 EJB JAR 파일을 사용하는 고객을 위해 만들어진 것입니다. 배포 디스트립터를 생성하는 것은 빈 개발자의 책임입니다. 일단 엔터프라이즈 빈이 배포되면 사용자도 Deployment Descriptor를 수정할 수 있습니다.

Deployment Descriptor의 정보는 엔터프라이즈 빈 속성을 설정하는 데 사용됩니다. 이러한 속성은 특정한 환경에서 엔터프라이즈 빈이 작동하는 방법을 정의합니다. 예를 들어, 빈의 트랜잭션 속성을 설정하면 빈이 트랜잭션에 따라 어떻게 작동하는지가 정의됩니다. Deployment Descriptor는 다음과 같은 정보를 가집니다.

- 홈 인터페이스 및 원격 인터페이스에 대한 클래스와 빈 클래스의 타입 또는 이름을 정의하는 타입 정보
- 엔터프라이즈 빈의 홈 인터페이스가 등록되어 있는 이름을 설정하는 JNDI 이름

- Container 관리 방식(Container-managed persistence, CMP)을 가능하게 하는 필드
- 빈의 트랜잭션 동작을 통제하는 트랜잭션 규칙
- 엔터프라이즈 빈에 대한 액세스를 통제하는 보안 속성
- 데이터베이스에 연결할 때 사용되는 데이터 소스 정보 같은 Borland 특정 정보

Deployment Descriptor의 정보 유형

Deployment Descriptor의 정보는 두 가지 기본 유형으로 나뉩니다.

- 엔터프라이즈 빈의 구조에 관한 정보

  구조에 관한 정보는 엔터프라이즈 빈의 구조를 설명하고 엔터프라이즈 빈의 외부 종속성을 나타냅니다. 이 정보는 필수 항목입니다. 구조에 관한 정보를 수정하면 빈의 기능이 훼손될 수도 있으므로 구조에 관한 정보는 대체로 변경되지 않습니다.

- 애플리케이션 어셈블리 정보

  애플리케이션 어셈블리 정보는 `ejb-jar.xml` 파일에 포함된 엔터프라이즈 빈이 더 큰 애플리케이션 배포 유닛으로 구성되는 방법에 대해 설명합니다. 이 정보는 옵션입니다. 어셈블리 레벨 정보를 수정하여 어셈블된 애플리케이션 동작이 변할 수는 있으나 빈의 기능을 훼손하지 않고 어셈블리 레벨 정보를 변경할 수 있습니다.

구조에 관한 정보

빈 개발자는 EJB jar에 있는 각각의 빈에 대해 다음과 같은 구조에 관한 정보를 제공해야 합니다.

모든 엔터프라이즈 빈

- Deployment Descriptor에서 빈을 참조하는 데 사용되는 니모닉(mnemonic)인 엔터프라이즈 빈의 이름
- 엔터프라이즈 빈의 클래스
- 엔터프라이즈 빈의 홈 인터페이스
- 엔터프라이즈 빈의 원격 인터페이스
- 세션 빈 또는 엔티티 빈과 같은 엔터프라이즈 빈의 타입
- 빈이 구성 매개변수를 가진 경우의 환경 항목
- 리소스 팩토리 참조
- 엔터프라이즈 빈이 다른 엔터프라이즈 빈을 참조할 경우의 EJB 참조
- 엔터프라이즈 빈이 특정 역할에 액세스해야할 경우의 보안 역할 참조

세션 빈

- 상태 없음(stateless) 또는 상태 있음(stateful)과 같은 세션 빈 상태 관리의 타입
- 동기화 콜백을 가지는 상태 있음(stateful) 빈에 대한 세션 빈 트랜잭션 구분(demarcation) 타입

엔티티 빈

- 엔티티 빈의 지속성 관리 방식
- 엔티티 빈의 기본 키 클래스
- 컨테이너에 의해 관리되는 빈을 위한 컨테이너 관리 필드

애플리케이션 어셈블리 정보

다음과 같은 애플리케이션 어셈블리 정보를 지정할 수 있습니다. 애플리케이션을 어셈블하는 동안 이 정보는 옵션입니다. 이러한 동일한 정보가 배포자의 역할에 대해서는 옵션이 아닙니다.

- 엔터프라이즈 빈 참조의 바인딩
- 보안 역할
- 메소드 권한
- 보안 역할 참조의 연결
- 트랜잭션 속성

애플리케이션 어셈블리 또는 배포 프로세스 중에 다음과 같은 구조에 관한 정보를 수정할 수 있습니다.

- 환경 항목의 값 애플리케이션 어셈블러는 기존 속성을 바꾸거나 환경 속성의 값을 정의할 수 있습니다.
- Description 필드 애플리케이션 어셈블러는 기존 설명을 바꾸거나 새로운 설명 요소를 생성할 수 있습니다.

다른 타입의 구조에 관한 정보는 수정할 수 없습니다. 하지만 배포 시, 다른 애플리케이션 어셈블리 정보를 수정할 수는 있습니다.

보안

애플리케이션 어셈블러는 일반적으로 Deployment Descriptor에 다음과 같은 정보를 지정합니다.

- 보안 역할
- 메소드 권한
- 보안 역할 참조와 보안 역할 사이의 연결

보안 역할

개발자는 Deployment Descriptor의 보안 역할 요소를 사용하여 하나 이상의 보안 역할을 정의할 수 있습니다. 엔터프라이즈 빈의 클라이언트에 필요한 보안을 정의합니다.

메소드 권한

개발자는 Deployment Descriptor의 메소드 권한 요소를 사용하여 메소드 권한을 정의할 수 있습니다. 메소드 권한은 보안 역할과 엔터프라이즈 빈의 원격 및 홈 인터페이스의 메소드 사이에 쌍을 이루는 관계입니다.

보안 역할 참조의 연결

보안 역할이 정의되어 있는 경우, 개발자는 Deployment Descriptor의 역할 연결 요소를 사용하여 보안 역할을 보안 역할 참조에 연결해야 합니다.

대상 애플리케이션 서버가 Borland AppServer라면 엔터프라이즈 빈,`ejb-inprise.xml`에 대한 추가적인 Deployment Descriptor를 갖게 됩니다. Deployment Descriptor의 Borland 특정 정보에 대해서는 *Borland AppServer's Enterprise JavaBeans Programmer's Guide*의 "Deploying Enterprise JavaBeans"를 참조하십시오.

애플리케이션 서버에 배포

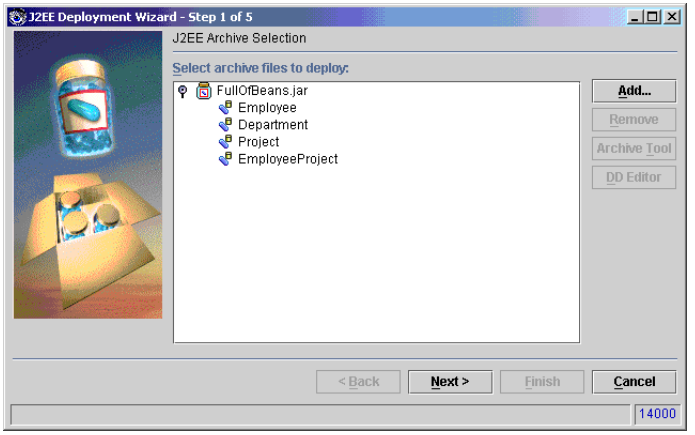
빈의 작동에 만족하고 현재 프로젝트에 대한 애플리케이션 서버로 Borland AppServer를 선택한 경우에는 JBuilder의 EJB Deployment 마법사를 사용하여 Borland AppServer에 빈을 배포할 수 있습니다. 대상 애플리케이션 서버가 WebLogic 서버나 WebSphere 애플리케이션 서버일 경우 Tools|EJB Deployment wizard를 선택하면 WebLogic 서버나 WebSphere 애플리케이션 서버를 배포하는 데 사용할 수 있는 Deploy Settings 대화 상자가 보입니다.

여기서 설명하는 단계에서는 사용자가 Borland 컨테이너에 배포한다고 가정합니다. 애플리케이션 서버를 시작한 다음 EJB Deployment 마법사를 사용하십시오.

하나 이상의 JAR 파일 배포

다음과 같은 방법으로 하나 이상의 JAR 파일을 Borland 애플리케이션 서버에 배포합니다.

1 Tools|EJB Deployment를 선택하여 EJB Deployment 마법사를 표시합니다.

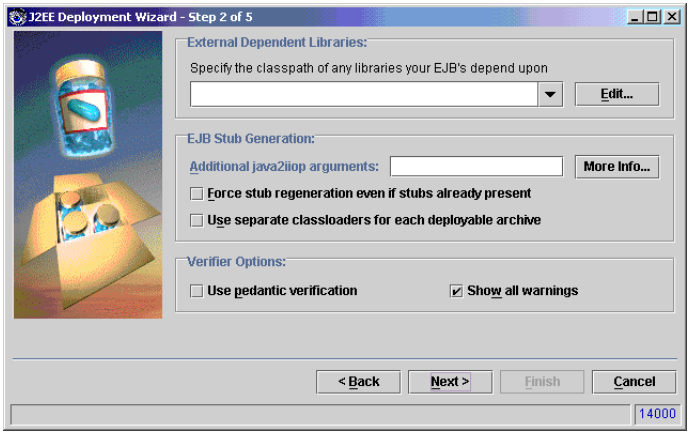


현재 프로젝트에 들어 있는 JAR 파일이 나타납니다.

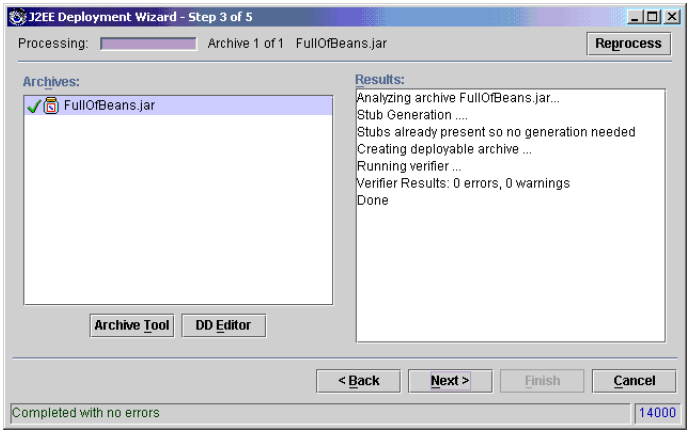
2 배포하려는 빈을 포함한 JAR 파일을 지정합니다.

JAR를 추가로 더 배포하려면 Add 버튼을 클릭하고 추가하려는 JAR를 지정합니다. 목록에서 JAR를 지우려면 선택한 후 Remove를 클릭합니다. 이 단계에서 BAS 4.5 아카이브 툴과 Deployment Descriptor 에디터에 액세스할 수도 있습니다.

3 Next를 클릭하여 다음 단계로 갑니다.

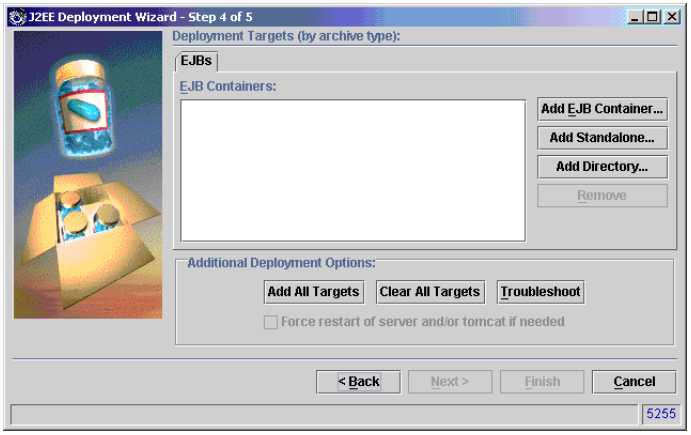


4 엔터프라이즈 빈이 의존하는 라이브러리의 classpath와 원하는 스텁 생성 옵션을 지정하고 입증자 옵션(verifier option)을 설정합니다. Next를 클릭합니다.



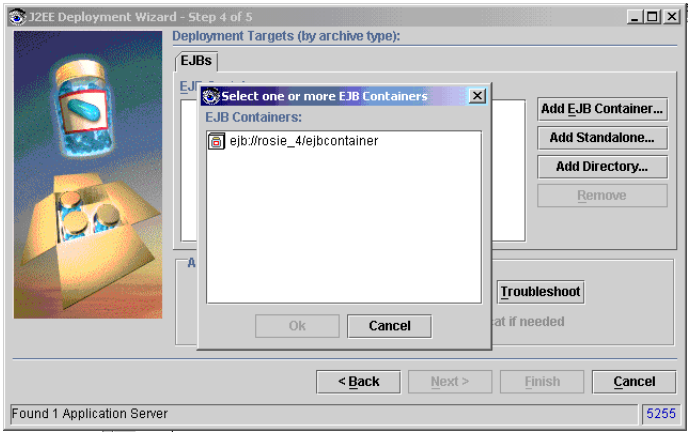
마법사는 JAR를 분석하여 결과를 보고합니다.

5 보고된 오류가 없을 경우에는 Next를 클릭하고 오류가 있을 경우에는 오류를 수정하고 다시 시작합니다.

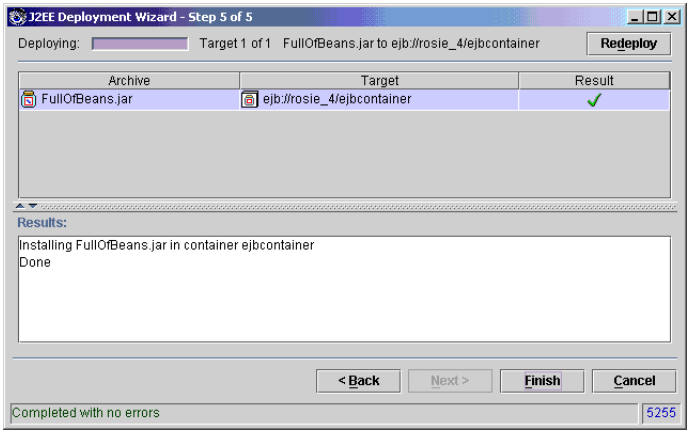


6 이 페이지를 사용하여 사용자의 EJB를 배포하려는 컨테이너를 선택합니다.

이 예제는 사용자가 Add EJB Container 버튼을 클릭하여 컨테이너를 선택한 것입니다.



컨테이너를 지정했으면 Next를 클릭합니다.



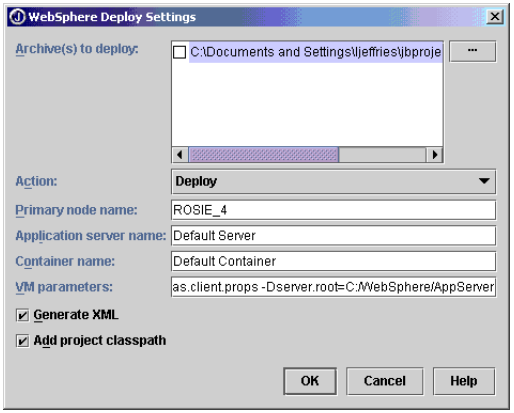
마법사는 EJB를 배포하고 그 결과를 보고합니다.

7 Finish를 클릭하여 마법사를 닫습니다.

WebLogic 또는 WebSphere 서버에 배포

WebLogic 및 WebSphere 개발자는 Tools|EJB Deployment 명령을 사용하여 EJB를 배포할 수도 있습니다. 현재 프로젝트에 대해 선택한 애플리케이션 서버가 WebLogic 또는 WebSphere인 경우, 이 명령은 서버에만

적용되는 Deploy Settings 대화 상자를 나타냅니다. 다음은 이러한 WebSphere Deploy Settings 대화 상자의 예입니다.



필요한 필드에 내용을 입력하고 OK를 선택합니다. 자세한 내용은 Deploy Settings 대화 상자에 있는 Help 버튼을 클릭하여 참조하십시오.

Properties 대화 상자에서 배포 옵션 설정

Tools|EJB Deployment를 사용하여 프로젝트의 현재 애플리케이션 서버에 배포하기 위한 옵션을 설정할 수도 있고 Properties 대화 상자를 사용할 수도 있습니다. 속성이 설정되는 특정 노드를 위해 이러한 속성을 저장합니다.

다음과 같은 방법으로 Properties 대화 상자를 사용하여 배포 옵션을 설정합니다.

1 EJB 그룹 노드 또는 이 노드의 자식 JAR를 마우스 오른쪽 버튼으로 클릭하여 팝업 메뉴를 표시합니다.

2 Properties를 선택하여 Properties 대화 상자를 표시합니다. EJB 그룹 노드를 마우스 오른쪽 버튼으로 클릭했을 경우에는 Properties 대화 상자의 Deployment 탭을 클릭한 다음 BAS 4.5와 같은 애플리케이션 서버 특정적인 페이지를 클릭해야 합니다.

3 옵션을 설정합니다. BAS 4.5의 경우에는 VM 매개변수만 설정하면 됩니다. IAS 4.1의 경우에는 호스트 이름과 VM 매개변수를 설정할 수 있습니다. WebLogic 사용자는 유닛 이름, 배포 옵션, 암호와 VM 매개변수를 설정할 수 있습니다. WebSphere 사용자는 기본 노드 이름, 애플리케이션 서버 이름, 컨테이너 이름, VM 매개변수와 XML을 생성하는 옵션을 설정할 수 있습니다. 다른 배포 옵션을 갖는 여러 노드가 선택된 경우에는 기본값을 사용합니다.

4 대상 애플리케이션 서버가 WebSphere이고 WebSphere XMLConfig 유틸리티에 입력하여 XML 파일을 생성하기 원할 경우에는 Generate XML 체크 박스를 선택 표시합니다. 이 옵션을 선택 표시하지 않으면 파일이 생성되지 않습니다. `deploy_<selectednode>.xml`이라는 프로젝트 노

드 아래에 나타나는 생성된 XML 파일을 사용자가 수정할 경우에는 이 옵션을 선택 해제하여 변경 사항을 잃지 않도록 합니다.

애플리케이션 서버에 빠르게 배포

사용자는 개발 주기 동안 이미 실행 중인 컨테이너에 엔터프라이즈 빈을 신속하게 배포, 재배포 및 배포 해제할 수도 있습니다. 프로젝트 창에서 EJB 그룹 노드나 EJB 그룹 노드의 자식 노드를 마우스 오른쪽 버튼으로 클릭하여 배포 명령 목록을 보십시오.

- Deploy –현재 실행 중인 프로젝트 애플리케이션 서버의 컨테이너에 JAR를 배포합니다.
- Redeploy – 현재 실행 중인 컨테이너에 JAR를 다시 배포합니다.
- Undeploy – 이미 배포되어 있는 JAR를 실행 중인 컨테이너에서 배포 해제합니다.
- List Deployments – 실행 중인 컨테이너에 배포된 모든 JAR를 나열합니다.
- Stop Container – 컨테이너를 중단합니다. 이 옵션은 WebSphere의 경우에만 나타납니다. 배포된 EJB가 변경될 경우 컨테이너를 중단한 다음 다시 시작해야 등록에 대한 변경사항이 적용됩니다.
- Start Container – 컨테이너를 시작합니다. 이 옵션은 WebSphere의 경우에만 나타납니다.

Chapter

14

Deployment Descriptor 에디터 사용

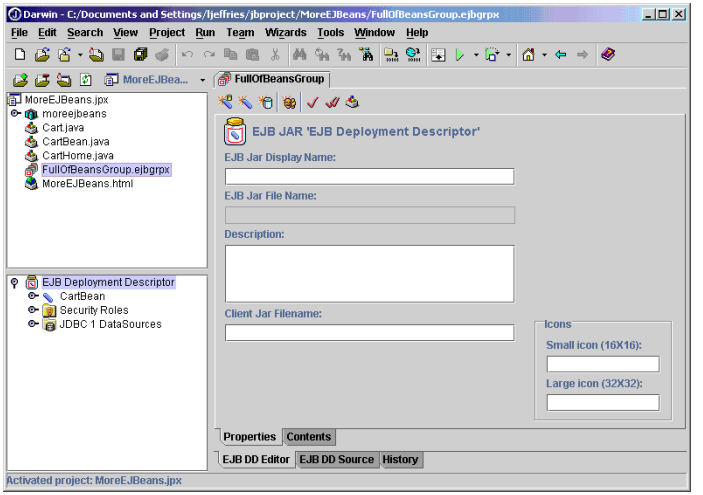
JBuilder에는 EJB deployment descriptor 파일의 트랜잭션 정책과 보안 규칙 같은 Borland 배포 정보를 변경하는 데 사용할 수 있는 Deployment Descriptor 에디터가 포함되어 있습니다. 또한 엔터프라이즈 빈을 지속하는 메소드를 변경할 수 있습니다.

JBuilder의 Deployment Descriptor 에디터를 통해 기존 Borland Deployment Descriptor를 수정할 수 있습니다. Deployment Descriptor에 대한 일반적인 내용은 Chapter 13 "엔터프라이즈 빈 배포"를 참조하십시오.

WebLogic과 WebSphere 서버에 고유한 몇 가지 속성을 보고 편집할 수도 있습니다. 자세한 내용은 14- 26페이지의 "WebLogic 및 WebSphere 속성 보기 및 편집"를 참조하십시오.

Deployment Descriptor 에디터 표시

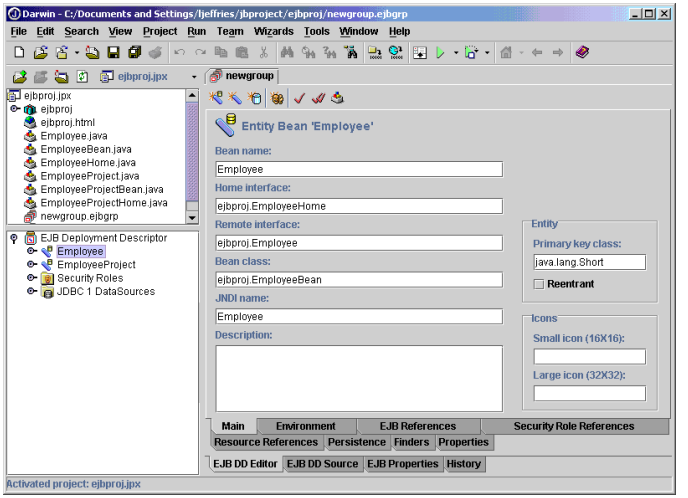
Deployment Descriptor 에디터를 표시하려면 프로젝트 창에서 EJB 그룹을 더블 클릭합니다. Deployment Descriptor 에디터가 나타납니다.



구조 창에 EJB 그룹 트리가 나타납니다.

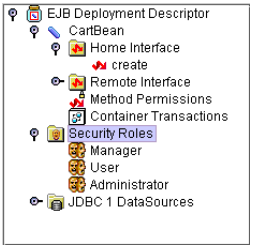
엔터프라이즈 빈의 Deployment Descriptor 보기

Deployment Descriptor 에디터에서 엔터프라이즈 빈에 관한 정보를 보려면 구조 창에서 해당 빈을 클릭합니다. 또는 Contents 탭을 클릭하여 EJB 그룹의 컨텐트를 표시한 다음 보려는 빈의 아이콘을 더블 클릭합니다. 그렇게 하면 Deployment Descriptor 에디터에 다음과 같은 몇 가지 탭이 더 나타납니다.



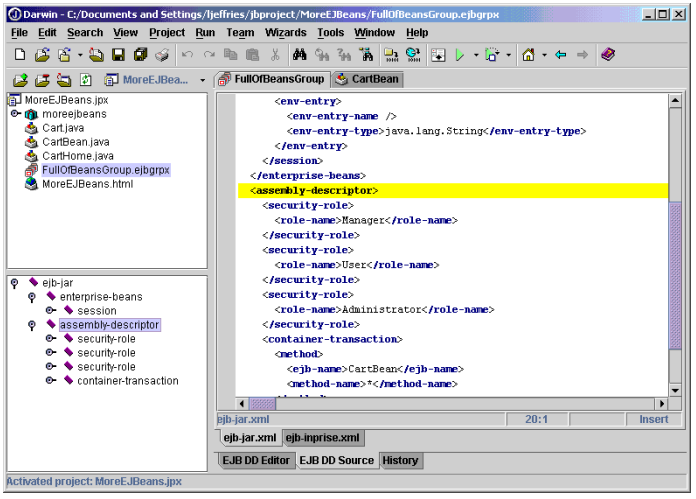
Deployment Descriptor 에디터의 아래쪽에 있는 이러한 탭 중 하나를 클릭하여 다른 페이지를 볼 수 있습니다. 에디터를 사용하여 빈의 배포 정보를 원하는 대로 변경합니다.

구조 창에서 Deployment Descriptor의 추가 내용을 볼 수 있습니다.



구조 창에서 관심 있는 엔터프라이즈 빈의 노드를 확장합니다. Home Interface, Remote Interface, Method Permission, Container Transaction 같은 추가 노드의 일부 또는 전부가 나타납니다. 이러한 노드 중 하나를 선택하면 빈에 대한 추가 정보가 표시됩니다. 예를 들어, Home Interface를 선택하면 홈 인터페이스의 메소드 이름을 갖는 컨텐트 창 아이콘이 표시되고 Remote Interface를 선택하면 원격 인터페이스의 메소드 이름을 갖는 컨텐트 창 아이콘이 표시됩니다. 또한 구조 창에서 Security Roles와 Data Sources 노드를 사용하여 보안 역할과 데이터 소스에 대한 정보를 표시하고 편집할 수 있습니다.

각 Descriptor의 소스 코드를 보려면 EJB DD Source 탭을 클릭합니다. EJB 그룹에 있는 각 Deployment Descriptor의 경우 해당 탭에 파일의 이름이 함께 나타납니다. 보고자 하는 파일의 탭을 선택합니다. Deployment Descriptor의 소스 코드를 보면서 구조 창에서 요소를 클릭하여 소스 코드의 해당 요소로 highlight bar를 이동할 수 있습니다. 소스 코드를 직접 편집할 수 있습니다.



새 엔터프라이즈 빈에 대한 정보 추가

다음과 같은 방법으로 Deployment Descriptor 에디터를 사용하여 엔터프라이즈 빈을 Deployment Descriptor에 추가합니다.

1 구조 창에서 EJB Deployment Descriptor 노드를 선택합니다.

2 추가하려는 빈의 타입을 선택합니다.

- 세션 빈을 추가하려면 Deployment Descriptor 에디터 툴바에 있는  아이콘을 클릭합니다.
- 엔티티 빈을 추가하려면 Deployment Descriptor 에디터 툴바에 있는  아이콘을 클릭합니다.

이름 대화 상자가 나타납니다.

3 대화 상자에서 추가하려는 엔터프라이즈 빈의 이름을 입력하고 OK를 선택합니다.

구조 창에 새 빈이 나타납니다.

4 구조 창에서 빈을 선택합니다.

엔터프라이즈 빈에 대한 일련의 패널들이 컨텐트 창에 나타납니다. 아래에 있는 탭을 사용하여 새 빈에 대한 정보를 입력하는 데 사용할 패널을 표시합니다.

빈 정보 변경

빈 정보를 변경하려면 구조 창에서 빈을 선택합니다. 엔터프라이즈 빈에 대한 일련의 패널들이 컨텐트 창에 나타납니다. 아래에 있는 탭을 사용하여 새 빈에 대한 기존 정보를 수정하는 데 사용할 패널을 표시합니다.

XML 및 Deployment Descriptor 소스 코드 사용에 익숙한 경우 EJB DD Source 탭을 선택하고 소스 코드에서 직접 변경할 수 있습니다.

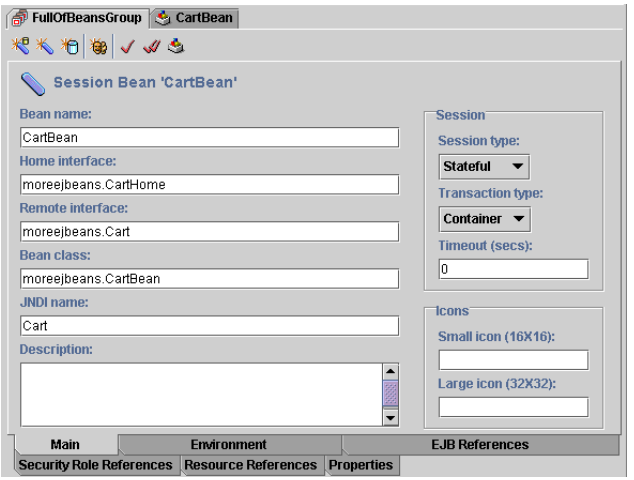
엔터프라이즈 빈 정보

이 단원에서는 Deployment Descriptor에서 엔터프라이즈 빈을 생성하고 저장할 수 있는 정보의 타입에 대해 설명합니다.

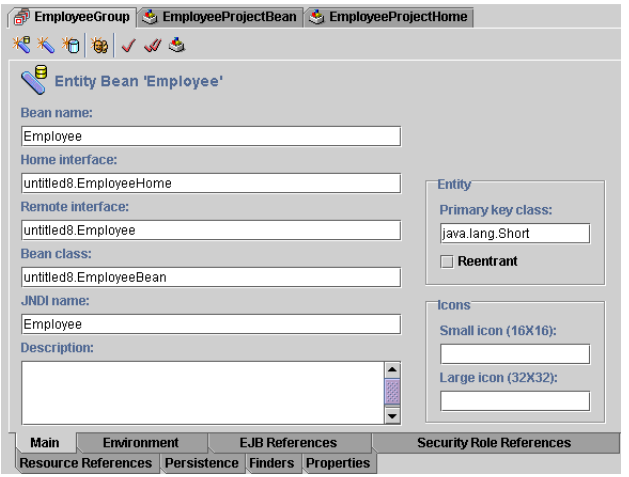
Main 패널

Main 패널을 사용하여 엔터프라이즈 빈에 대한 일반적인 정보를 입력하거나 변경합니다.

다음은 세션 빈의 Main 패널입니다.



다음은 엔티티 빈의 Main 패널입니다.



Main 패널은 다음 정보를 포함합니다.

- **Description:** 빈의 용도와 기능에 대한 요약 내용.이 정보는 옵션입니다.
- **Bean Name:** 빈 프로바이더가 엔터프라이즈 빈에 할당한 논리적 이름. 각 엔터프라이즈 빈은 논리적 이름을 가집니다. 빈의 논리적 이름과 빈에 할당된 JNDI 이름 사이에는 구조적인 관계가 없습니다. 빈 배포자가 빈의 논리적 이름을 변경할 수도 있습니다.

- Home Interface: 엔터프라이즈 빈의 홈 인터페이스의 전체 이름.이 정보는 지정되어야 합니다.
- Remote Interface: 엔터프라이즈 빈의 원격 인터페이스의 전체 이름.이 정보는 지정되어야 합니다.
- Bean Class: 빈의 비즈니스 메소드를 구현하는 Java 클래스의 전체 이름. 이 정보는 지정되어야 합니다.
- JNDI Name: 엔터프라이즈 빈의 홈 인터페이스의 JNDI 이름.
- Small icon: 빈을 표현하는데 사용되는 16 X 16 픽셀 아이콘 파일의 이름.
- Large icon: 빈을 표현하는데 사용되는 32 X 32 픽셀 아이콘 파일의 이름.

Session 빈의 Main 패널에는 다음이 포함됩니다.

- Session Type: 엔터프라이즈 빈이 상태 없음(Stateless)인지 상태 있음(Stateful)인지 여부를 지정합니다.
- Transaction Type: 트랜잭션 정책이 빈에 의해 설정되는지 컨테이너에 의해 설정되는지 여부를 지정합니다.

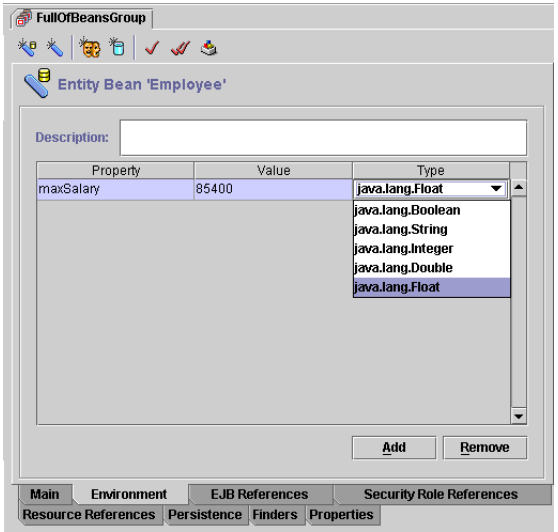
Entity 빈의 Main 패널에는 다음이 포함됩니다.

- Primary Key Class: Entity 빈의 기본 키 클래스의 전체 이름. 기본 키 클래스를 지정해야 합니다.
- Reentrant: 빈이 재진입임을 나타냅니다. 빈 재진입은 만들지 않는 것이 좋습니다.

Environment 패널

Environment 패널은 엔터프라이즈 빈의 환경 엔트리를 모두 나열합니다. 빈을 어셈블하거나 배포할 때 환경 엔트리를 통해 빈의 비즈니스 로직을 사용자 지정할 수 있습니다. 환경을 통해서 빈의 소스 코드에 액세스하거나 변경하지 않고 빈을 사용자 지정할 수 있습니다.

각 엔터프라이즈 빈은 자신의 환경 엔트리 집합을 정의합니다. 엔터프라이즈 빈의 모든 인스턴스는 동일한 환경 엔트리를 공유합니다. 엔터프라이즈 빈 인스턴스는 런타임 시에 빈의 환경을 수정할 수 없습니다.



환경 엔트리를 추가하려면, 다음과 같이 하십시오.

1 Add를 클릭하여 새 엔트리를 만듭니다.

비어 있는 새 행이 나타납니다.

2 Property 열에 속성을, Value 열에 속성 값을 입력합니다.

3 Type 드롭다운 리스트에서 속성 타입을 선택합니다.

4 원하는 만큼 환경 엔트리를 계속 추가합니다.

행을 제거하려면, 다음과 같이 하십시오.

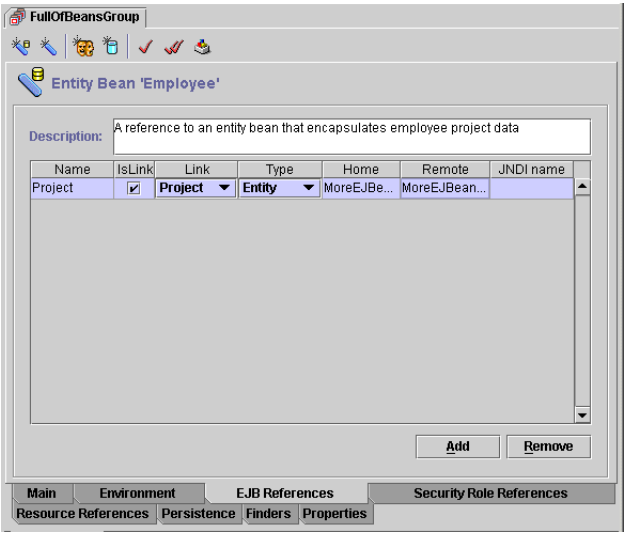
1 행을 선택합니다.
2 Remove 버튼을 클릭합니다.

다음은 환경 엔트리에 대한 중요한 사항입니다.

- 빈 프로바이더는 엔터프라이즈 빈의 코드에서 액세스하는 모든 환경 엔트리를 선언해야 합니다.
- 빈 프로바이더가 환경 엔트리의 값을 포함하는 경우 어셈블리 또는 배포 동안에 값을 변경할 수 있습니다.
- 어셈블러는 빈 프로바이더에 의해 설정된 환경 엔트리의 값을 수정할 수 있습니다.
- 배포자는 모든 환경 엔트리의 값이 의미있는 값으로 설정되었는지 확인해야 합니다.

EJB References 패널

EJB References 패널은 빈이 요구하는 다른 엔터프라이즈 빈의 홈에 대한 모든 엔터프라이즈 빈 참조를 나열합니다.



각 EJB 참조는 참조하는 엔터프라이즈 빈이 참조되는 빈에 대해 가지는 인터페이스 요구 사항을 설명합니다. 동일한 JAR 파일 내의 빈과 빈 사이의 참조 또는 엔티티 빈에 대한 세션 빈과 같은 외부 엔터프라이즈 빈(JAR 파일 외부에 있는 빈)으로부터 참조를 정의할 수 있습니다. 각 참조는 다음 필드를 포함합니다.

- **Description:** 참조되는 빈에 대한 간략한 설명.이 정보는 옵션입니다.
- **Name:** 참조된 빈의 이름.
- **IsLink:** IsLink가 선택 표시되어 있으면 참조는 JAR 안에 있는 빈에 대한 것이므로 JNDI Name 값은 관련이 없습니다. IsLink가 선택 표시되어 있지 않으면 빈을 찾기 위해 JNDI 이름이 사용됩니다. 이 옵션을 선택 표시하는 경우 Link 드롭다운 리스트에서 참조되는 빈을 선택하십시오.
- **Link:** EJB 참조를 대상 엔터프라이즈 빈에 연결합니다. Link 값은 대상 빈의 이름입니다. 이 정보는 옵션입니다.
- **Type:** 참조되는 빈의 예상 타입.
- **Home:** 참조되는 빈의 홈 인터페이스의 예상 Java 타입.
- **Remote:** 참조되는 빈의 원격 인터페이스의 예상 Java 타입.
- **JNDI Name:** 참조되는 빈의 JNDI 이름.

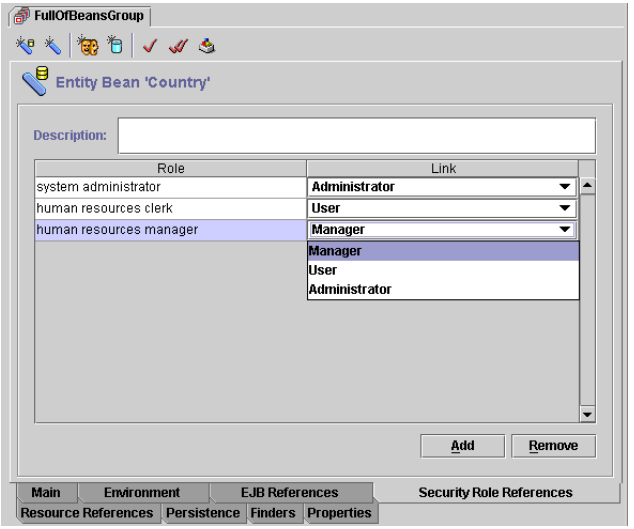
다음은 EJB 참조에 대한 중요한 사항입니다.

- 대상 엔터프라이즈 빈은 선언된 EJB 참조와 호환 가능한 타입이어야 합니다.

- 선언된 EJB 참조는 모두 운영 환경에 있는 엔터프라이즈 빈의 홈에 바인딩되어야 합니다.
- Link 값이 지정되어 있으면 엔터프라이즈 빈 참조는 대상 엔터프라이즈 빈의 홈에 바인딩되어야 합니다.

Security Role References 패널

Security Role References 패널은 보안 역할에 대한 모든 엔터프라이즈 빈의 참조를 나열합니다. 패널은 빈 개발자가 사용하는 보안 역할 참조를 애플리케이션 어셈블러 또는 배포자가 정의하는 특정 보안 역할에 연결합니다.



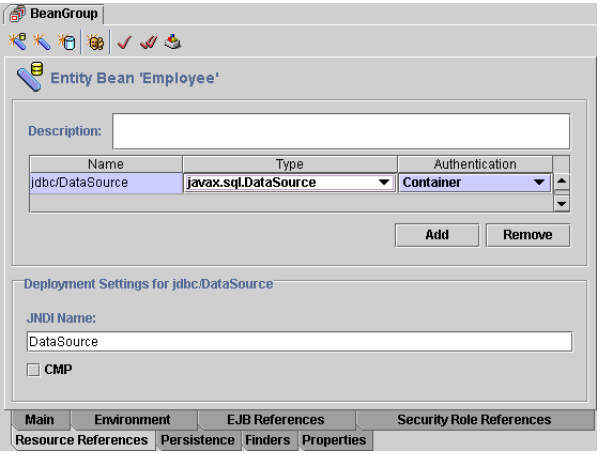
보안 역할 참조를 추가하려면 보안 역할이 하나 이상 정의되어 있고 이 패널의 Add 버튼이 비활성화되어 있어야 합니다. 애플리케이션 배포의 보안 역할 생성 및 지정에 대한 내용은 14- 24페이지의 "보안 역할과 메소드 권한 추가"를 참조하십시오.

역할을 추가하려면 Add 버튼을 클릭하고 다음 세 가지 필드에 내용을 입력합니다.

- **Description:** 보안 역할에 대해 설명하는 옵션 필드입니다.
- **Role:** 빈 개발자가 지정하는 보안 역할의 이름입니다.
- **Link:** 애플리케이션 배포 시 사용되는 보안 역할의 이름입니다. 보통 이 역할은 특정 운영 환경에서 작동하도록 애플리케이션 어셈블러 또는 배포자에 의해 정의됩니다.

Resource References 패널

Resource References 패널은 모든 엔터프라이즈 빈의 리소스 팩토리 참조를 나열합니다. 이 패널을 통해 애플리케이션 어셈블러 및 빈 개발자는 엔터프라이즈 빈에 의해 사용되는 모든 참조를 찾을 수 있습니다. Container 관리 방식을 사용하는 각 엔티티는 리소스 참조를 가져야 합니다.



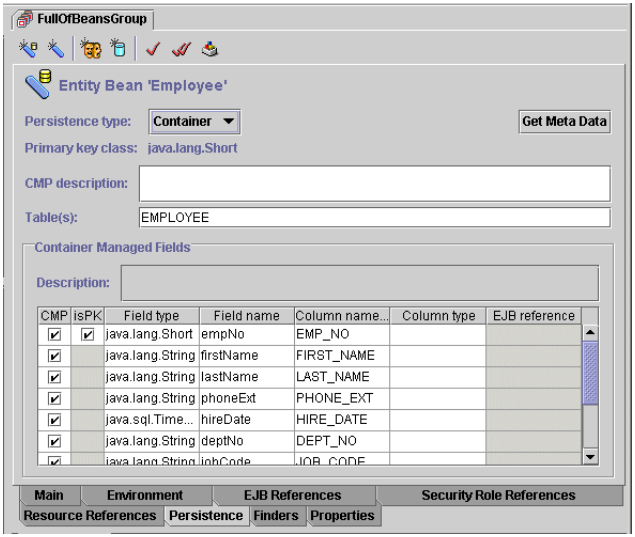
리소스 참조를 추가하려면 Add 버튼을 클릭하고 다음 필드에 내용을 입력합니다.

- **Description:** 리소스 참조에 대한 설명.이 정보는 옵션입니다.
- **Name:** 엔터프라이즈 빈의 코드에 사용된 환경 엔트리의 이름.
- **Type:** 엔터프라이즈 빈의 코드에 의해 예상되는 리소스 팩토리의 Java 타입. (이는 리소스의 Java 타입이 아니라 리소스 *팩토리*의 Java 타입입니다.)
- **Authentication:** Application 인증은 엔터프라이즈 빈이 프로그램에서 리소스 사용승인을 수행하는 것을 나타냅니다. Container 인증은 컨테이너가 배포자에 의해 제공되는 주요 매핑 정보에 기반하는 리소스 사용을 승인하는 것을 나타냅니다.
- **JNDI Name:** 리소스 참조의 JNDI 이름.
- **CMP(container-managed persistence):** 빈에 대해 Container 관리 방식을 사용하는 경우 CMP 필드가 선택 표시되어 있는 리소스 참조가 하나 있어야 합니다.

Persistence 패널

Persistence 패널은 엔티티 빈에만 나타납니다. Persistence 패널은 엔터프라이즈 빈의 퍼시스턴스가 관리되는 방식을 지정합니다. 패널에 표시되

는 정보는 퍼시스턴스가 Bean 관리 방식인지 Container 관리 방식인지의 여부에 따라 다양합니다.



Persistence 패널에는 다음 필드가 포함됩니다.

- Persistence Type: 퍼시스턴스가 빈에 의해 관리되는지 컨테이너에 의해 관리되는지 여부를 나타냅니다.
- Primary Key Class: 엔티티 빈의 기본 키 클래스의 전체 이름. 기본 키는 지정되어야 합니다.
- Get Meta Data button:

  이 버튼을 클릭하면 테이블의 메타데이터가 검색되고 각 Column Name 셀의 드롭다운 리스트가 구성됩니다. 드롭다운 리스트의 각 요소는 열 이름/열 타입 쌍입니다. 드롭다운 리스트에서 선택하면 열 이름과 열 타입 셀이 모두 입력됩니다. 드롭다운 리스트에는 패널에서 아직 사용되지 않은 열 이름만 포함되어 있습니다.

Container 관리 방식의 빈의 경우, 빈의 필드를 데이터베이스 테이블의 열에 매핑할 수 있도록 다음 추가 정보가 제공됩니다.

- **Table(s):** 빈에 의해 참조되는 데이터베이스 테이블의 이름.
- **CMP:** 선택 표시는 필드가 Container-managed임을 나타냅니다.
- **isPK:** 선택 표시는 필드가 기본 키임을 나타냅니다.
- **Field Type:** 필드의 데이터 타입.
- **Field Name:** 필드의 이름. Field Name 열에는 엔티티 빈의 모든 필드가 나열됩니다.
- **Column Name:** 빈의 복합 필드(예: location.street)를 데이터베이스 테이블의 열에 매핑할 수 있습니다. 루트 필드(예: location) 또는 하위 필드(예: location.street)를 매핑하거나 둘 다 매핑하지 않을 수도 있습니다.
- **Column Type:** 필드의 데이터 타입.
- **EJB Reference:** 필드 타입이 EJB 클래스인 경우 선택할 EJB 참조의 목록과 함께 메뉴가 나타납니다. 이러한 참조는 EJB 참조 패널에 설정됩니다.

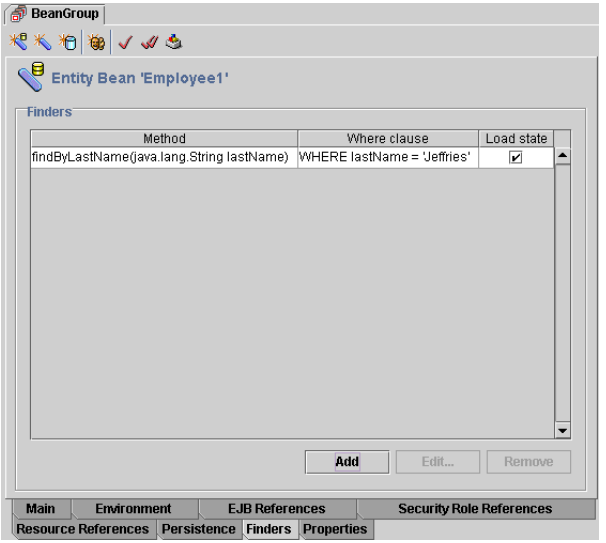
Container 관리 방식의 엔티티 빈을 설정하려면 Container as the Persistence Type을 선택합니다. EJB Entity Modeler를 사용하여 빈을 생성하는 경우 Container가 선택됩니다. 패널은 기본 키 클래스 이름을 표시하며 이 이름은 변경할 수 없습니다. CMP Description 필드에서 빈에 대해 설명하는 옵션 텍스트를 입력할 수 있습니다.

Container 관리 방식인 엔티티 빈의 각 필드에는 CMP 필드가 선택 표시되어 있습니다. 각 필드마다 매핑하는 열의 이름을 입력합니다. EJB Entity Modeler를 사용했다면 JBuilder에서 이미 이러한 열을 매핑했습니다. 원한다면 Column Name과 Column Type을 편집할 수 있습니다. Description 필드에 각 필드를 설명하는 텍스트를 입력할 수 있지만 필수 사항은 아닙니다.

Deployment Descriptor 에디터는 JDBC를 사용하여 기존 테이블에 대한 메타데이터를 얻습니다. 기존 엔티티 빈을 기존 테이블에 편리하게 연결할 수 있습니다. 예를 들어, 협력 업체 엔터프라이즈 빈을 구매하여 사용자 데이터베이스의 테이블에서 사용할 수 있습니다. Column Name과 Column Type 필드를 구성하기 위해 Get Meta Data 버튼을 클릭하면 메타데이터가 검색되어 표시됩니다.

Finders 패널

Finders 패널은 Container-managed 빈이 사용하는 "where" 절을 지정하여 빈이 지정한 finder 메소드를 실행합니다.



finders 패널에 다음 정보가 나타납니다.

- **Method:** finder 메소드 이름과 모든 매개변수 목록.
- **Where Clause:** 컨테이너에 의해 사용되는 SQL "where" 절을 지정하여 데이터베이스의 레코드를 검색합니다. 모든 SQL 문이 WebLogic 쿼리 로직으로 변환될 수 있지는 않습니다.
- **Load State:** 이 속성을 선택하면 find 작업이 발생할 때마다 컨테이너에서 모든 Container-managed 필드를 사전 로드(preload)할 수 있습니다.

다음과 같은 방법으로 finder 메소드를 지정합니다.

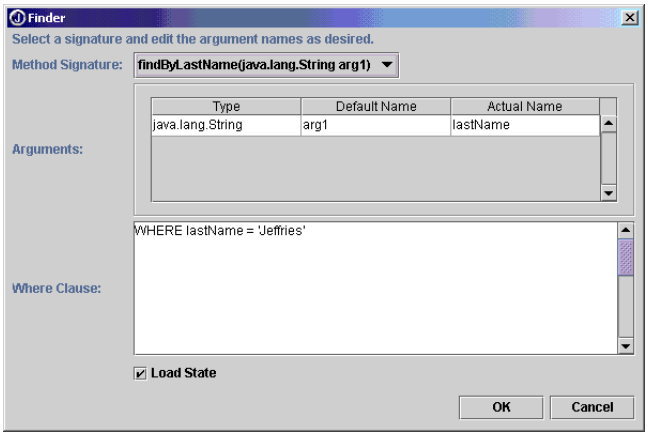
1. Add 버튼을 클릭합니다.

   Finder 대화 상자가 나타납니다.

2. 드롭다운 리스트에서 원하는 finder 메소드에 대한 메소드 시그너처를 선택합니다.

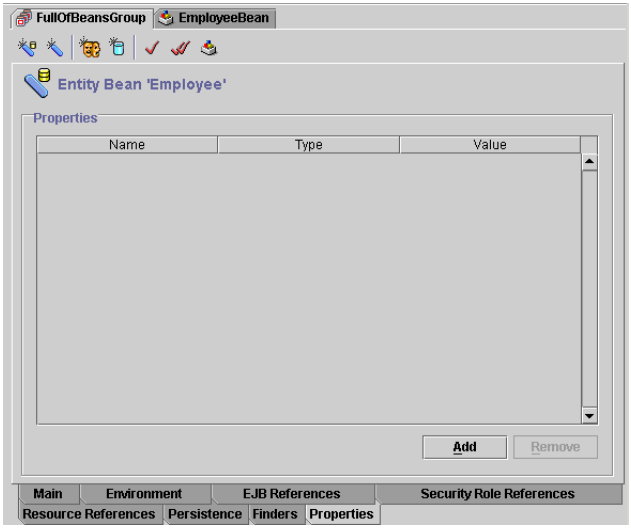
3. 원하는 경우 인수 이름을 수정하고 find 작업에 적합한 Where 절을 지정합니다.

다음은 예제입니다.



Properties 패널

Properties 패널은 컨텍스트에 따라 다양한 정보를 포함합니다. 구조 창에서 엔터프라이즈 빈을 선택하면 다음과 같은 Properties 창이 나타납니다.



다음과 같은 방법으로 구조 창에 선택되어 있는 엔터프라이즈 빈에 속성을 추가합니다.

1 Add 버튼을 클릭하여 패널에 행을 추가합니다.

2 Name 드롭다운 리스트에서 추가할 속성을 선택합니다.

3 Value 필드에 값을 지정합니다.

어떤 값은 드롭다운 리스트에서 선택하여 값을 지정할 수 있고 다른 값은 문자열 또는 정수 값과 같은 적절한 값을 입력해야 합니다. 어떤 값은 값이 참인지 여부를 나타내는 체크 박스를 선택하는 부울 값에 대한 체크 박스가 제공됩니다. 다음은 가능한 값 및 그에 대한 설명입니다.

ejb.cmp.primaryKeyGenerator

`com.inprise.ejb.cmp.PrimaryKeyGenerator` 인터페이스를 구현하고 기본 키를 생성하는 사용자에 의해 작성된 클래스를 지정합니다. 기본 키 생성에 대한 자세한 내용은 `/Borland/AppServer/examples/ejb/pkgen`의 예제를 참조하십시오.

ejb.cmp.getPrimaryKeyBeforeInsertSql

SQL, CMP 엔진 실행 파일을 지정하여 다음 INSERT 발생 시 기본 키를 생성합니다. CMP 엔진은 기본 키 값으로 엔티티 빈을 업데이트합니다. 이 속성은 보통 Oracle Sequences와 함께 사용됩니다. 기본 키 생성에 대한 자세한 내용은 `/Borland/AppServer/examples/ejb/pkgen`의 예제를 참조하십시오.

ejb.cmp.getPrimaryKeyAfterInsertSql

SQL. CMP 엔진 실행 파일을 지정하여 다음 INSERT 이후에 기본 키를 생성합니다. CMP 엔진은 기본 키 값으로 엔티티 빈을 업데이트합니다. 이 속성을 지정할 때 `ejb.cmp.ignoreColumnsOnInsert` 속성도 지정해야 합니다. 기본 키 생성에 대한 자세한 내용은 `/Borland/AppServer/examples/ejb/pkgen`의 예제를 참조하십시오.

ejb.cmp.ignoreColumnsOnInsert

INSERT 중에 CMP가 설정하지 않아야 할 열의 이름을 지정합니다. 이 속성은 `ejb.cmp.getPrimaryKeyAfterInsertSql` 속성과 함께 사용됩니다. 기본 키 생성에 대한 자세한 내용은 `/Borland/AppServer/examples/ejb/pkgen`의 예제를 참조하십시오.

ejb.cmp.checkExistenceBeforeCreate

새 엔티티 빈을 생성하기 전에 발생하는 존재 확인을 하지 않습니다. EJB 1.1 사양은 컨테이너에서 먼저 엔티티 빈의 존재를 확인(즉, 테이블에 행이 있는지 확인)하고 해당 엔티티가 있는 경우 `javax.ejb.DuplicateKeyException`을 발생시킵니다. 더 나은 성능을 위해 데이터베이스에 대한 이러한 별도 액세스를 제거하고 데이터베이스에서 중복 값이 삽입되지 않도록 할 수 있습니다.

ejb.cmp.jdbcAccesserFactory

`com.inprise.ejb.cmp.JdbcAccessor` 인터페이스의 사용자 구현 인스턴스에 대한 팩토리를 지정합니다. 이 인터페이스를 통해 특정 코드를 작성하여 `java.sqlResultSet`의 값을 얻거나 값을 `java.sql.PreparedStatement`로 설정할 수 있습니다. 기본값은 none입니다.

ejb.cmp.manager

com.inprise.ejb.cmp.Manager 인터페이스를 구현하는 클래스의 이름을 지정합니다. 이 클래스의 인스턴스는 Container-managed persistence(CMP)를 수행하는 데 사용됩니다.

ejb.maxBeansInPool

ready pool의 최대 빈 수를 지정합니다. ready pool이 이 제한을 초과하는 경우 unsetEntityContext()를 호출함으로써 컨테이너에서 엔티티가 제거됩니다. 기본 설정은 1000입니다.

ejb.maxBeansInCache

Option A 캐시의 최대 빈 수를 지정합니다(다음에 나오는 ejb.transactionCommitMode 참조). 캐시가 이 제한을 초과하는 경우 ejbPassivate()를 호출함으로써 엔티티가 ready pool로 옮겨 집니다. 기본 설정은 1000입니다.

ejb.transactionCommitMode

트랜잭션에 대한 엔티티 빈의 처리 방식을 나타냅니다. 값은 다음과 같습니다.

- A 또는 Exclusive-이 엔티티는 데이터베이스의 특정 테이블에 대한 배타 액세스를 갖습니다. 따라서, 마지막으로 커밋된 트랜잭션 끝의 빈 상태를 다음 트랜잭션 시작의 빈 상태로 가정할 수 있습니다. 빈은 트랜잭션에 걸쳐 캐시됩니다.
- B 또는 Shared-이 엔티티는 데이터베이스의 특정 테이블에 대한 액세스를 공유합니다. 하지만 더 나은 성능을 위해 특정 빈은 트랜잭션 간의 특정 기본 키와 연결된 채로 유지되어 트랜잭션 간의 ejbActivate()와 ejbPassivate()에 대한 중복 호출이 방지됩니다. 빈은 active pool에 머무릅니다. 이 설정은 기본값입니다.
- C 또는 None-이 엔티티는 데이터베이스의 특정 테이블에 대한 액세스를 공유합니다. 특정 빈은 트랜잭션 간의 특정 기본 키와의 연결을 유지하지 않지만 모든 트랜잭션 이후 ready pool로 돌아갑니다. 이것은 그다지 유용한 설정이 아닙니다.

ejb.cmp.optimisticConcurrencyBehavior

다음 중 하나를 지정할 수 있습니다.

- UpdateAllFields
- UpdateModifiedFields
- VerifyModifiedFields
- VerifyAllFields

- UpdateAllFields – 수정 여부와 상관 없이 모든 필드에 대해 업데이트를 실행합니다. CMP 빈이 "MyTable"이라는 테이블에 저장된 "key", "value1", "value2"의 세 개 필드를 갖는다고 가정합니다. 다음 업데이트는 빈의 수정 여부와 상관 없이 모든 트랜잭션의 끝에 실행됩니다.

  ```
  UPDATE MyTable SET (value1 = <value1>, value2 = <value2>)
       WHERE key = <key>
  ```

- UpdateModifiedFields 이것은 기본 설정 값입니다. 수정된 필드에만 업데이트를 실행하거나, 빈이 수정되지 않은 경우 업데이트를 모두 하지 않습니다. 위의 빈에서 "value1"만 수정된 경우 다음과 같은 업데이트가 실행됩니다.

  ```
  UPDATE MyTable SET (value1 = <value1>)
       WHERE key = <key>
  ```

  이를 통해 다음과 같은 이유로 성능을 크게 향상시킬 수 있습니다.

  1. 많은 경우에 데이터 액세스는 읽기 전용입니다. 읽기 전용인 경우에는 데이터베이스에 업데이트를 보내지 않는 것이 좋습니다. Borland는 이 단일 최적화를 통해 상당한 성능 향상을 이루었습니다.
  2. 많은 데이터베이스는 열의 수정 여부에 따라 로그를 작성합니다. 예를 들어, SQL 서버는 TEXT 또는 IMAGE 필드가 업데이트되는 경우 열의 값이 실제로 변경되었는지 여부에 상관 없이 업데이트를 로그합니다. 데이터베이스는 이전의 값과 동일한 값을 가지도록 업데이트하는 것("UpdateAllFields"를 통해 발생)과 실제로 열의 값을 수정하는 것의 차이를 구별하지 못합니다. 값이 실제로 변경되지 않은 경우 업데이트를 표시하지 않으면 DBMS 사용 시 성능에 매우 많은 영향을 미칠 수 있습니다.
  3. 데이터베이스로의 JDBC 기반 네트워크 소통량이 감소하고 JDBC 드라이버로의 작업이 줄어집니다. 일반적으로 네트워크 문제는 심각하지 않지만 JDBC 드라이버 문제는 중요합니다. 성능을 측정해 본 결과 대규모 EJB 애플리케이션에서 CPU 시간의 최대 70%가 JDBC 드라이버에 소비되는 것으로 나타났습니다. 경우에 따라 이러한 문제는 많은 상용 JDBC 드라이버가 충분히 성능이 조정되지 않은 사실에 기인합니다. 성능이 잘 조정된 드라이버에서도 수행해야 할 작업이 적을 때 더 잘 작동합니다.

- VerifyModifiedFields - 이 모드에서 CMP 엔진은 업데이트 중인 필드가 이전에 읽어들인 값과 일치하는지 확인하면서 조정된 업데이트를 실행합니다. 따라서 "value1"만이 수정된 이전 예제에서는 다음과 같은 업데이트가 실행됩니다.

```
UPDATE MyTable SET (value1 = <value1>)
     WHERE key = <key> AND value1 = <old-value1>
```

- VerifyAllFields - 이 모드는 모든 필드를 확인한다는 점을 제외하고는 VerifyModifiedFields와 유사합니다. 따라서 업데이트는 다음과 같습니다.

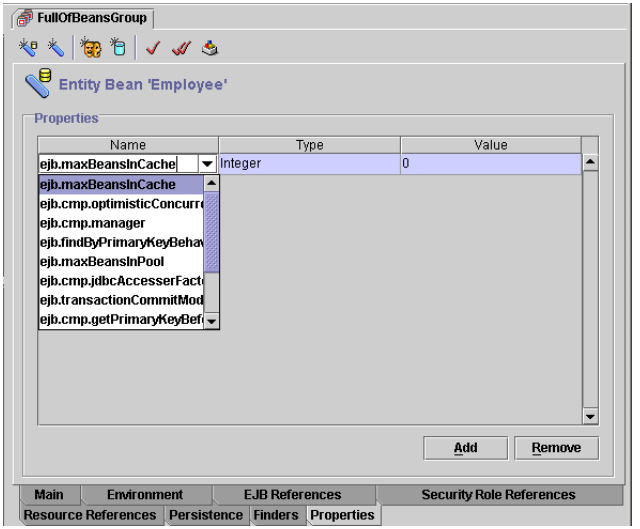
```
UPDATE MyTable SET (value1 = <value1>)
  WHERE key = <key> AND value1 = <old-value1> AND value2 = <old-value2>
```

이러한 두 가지 설정을 통해 컨테이너에서 SERIALIZABLE 독립 레벨을 복제할 수 있습니다. 종종 애플리케이션에서 직렬화 가능한 독립적 의미론을 필요로 합니다. 하지만 데이터베이스에서 이를 구현하도록 요구할 경우 성능에 크게 영향을 줄 수 있습니다. 테스트한 결과는 덜 제한된 독립 레벨 대신 Oracle의 SERIALIZABLE을 사용하면 애플리케이션 속도가 50% 이상 저하될 수 있음을 나타냅니다. 이러한 속도 저하의 주된 이유는 Oracle에서 열 레벨 잠금 모델을 사용하는 낙관적 동시성을 제공하기 때문입니다. 위의 두 가지 설정에서 사용자는 기본적으로 CMP 엔진이 필드 레벨 잠금을 사용하는 낙관적 동시성을 구현하도록 요구합니다. 병행 시스템에서 잠금의 단위 정도가 작을수록 동시성은 더 좋아집니다.

ejb.findByPrimaryKeyBehavior

findByPrimaryKey() 메소드의 원하는 동작을 표시합니다. 다음과 같은 값이 있습니다.

- Verify - `findByPrimaryKey()`의 표준 동작은 지정된 기본 키가 데이터베이스에 존재하는지 확인하는 것입니다.
- Load - finder 호출이 활성 트랜잭션에서 실행 중인 경우 이 동작은 `findByPrimaryKey()`가 호출될 때 빈 상태가 컨테이너에 로드되도록 합니다. 발견된 객체가 일반적으로 사용되며 find 타임 시 객체 상태를 로드하는 것이 좋다고 가정합니다. 이 설정은 기본값입니다.
- None - 이 동작은 `findByPrimaryKey()`가 no-op여야 함을 나타냅니다. 기본적으로 이 동작은 객체를 실제로 사용할 때까지 빈의 확인을 지연시킵니다. find를 호출하고 객체를 실제로 사용하는 사이에 객체를 제거할 수 있으므로 대부분의 프로그램에서 이 최적화는 클라이언트 로직에서 변경을 야기하지 않습니다.



다른 컨텍스트의 Properties 패널에 대한 내용은 14-22페이지의 "데이터 소스 속성 설정"과 14-25페이지의 "메소드 권한 지정"을 참조하십시오.

컨테이너 트랜잭션

Container-managed 트랜잭션을 사용하는 엔터프라이즈 빈에는 컨테이너에서 설정한 트랜잭션 정책이 있어야 합니다. Deployment Descriptor 에디터를 통해 Container-managed 트랜잭션 정책을 설정한 다음 이러한 정책을 엔터프라이즈 빈의 홈과 원격 인터페이스의 메소드에 연결할 수 있습니다.

Container-managed 트랜잭션 추가

다음과 같은 방법으로 컨테이너 트랜잭션을 추가합니다.

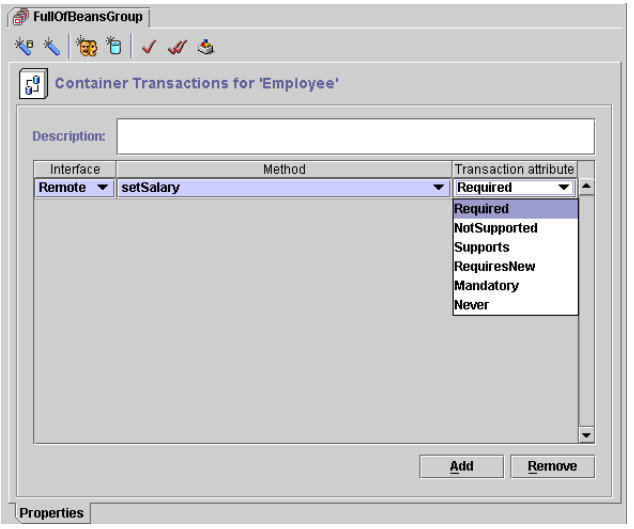
1. 구조 창에서 엔터프라이즈 빈을 더블 클릭하여 빈의 트리를 확장합니다.
2. 구조 창에서 Container Transactions을 클릭합니다.
3. Add 버튼을 클릭하여 그리드에 행을 추가합니다.
4. 행에서 홈 또는 원격 인터페이스 메소드를 노출할 Interface를 선택하거나 *를 선택하여 홈과 원격 인터페이스를 모두 나타냅니다.

5 사용 가능한 Method의 드롭다운 리스트에서 메소드를 선택하거나 *를 선택하여 이 트랜잭션이 속하는 모든 메소드를 선택합니다.

6 Transaction Attributes의 드롭다운 리스트에서 트랜잭션이 가질 속성을 선택합니다.

사용 가능한 트랜잭션 속성에 대한 설명은 20- 4페이지의 "트랜잭션 속성"을 참조하십시오.

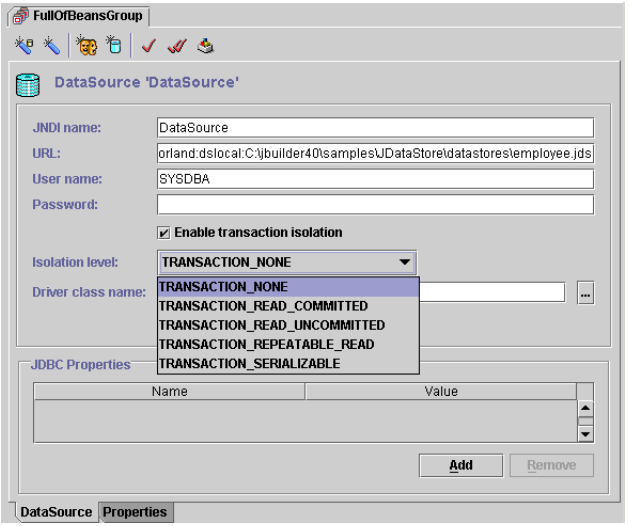
7 트랜잭션에 대해 설명하는 Description 필드에 설명할 내용을 입력합니다. 이 정보는 옵션입니다.



데이터 소스 사용

Deployment Descriptor에서 데이터 소스에 대한 내용을 보려면, 구조 창에서 Data Sources 노드를 확장하고 데이터 소스 중 하나를 선택합니다. Deployment Descriptor 에디터에서 Data Source 패널을 표시합니다. 패널을 사용하여 선택된 데이터 소스에 대한 정보를 수정할 수 있습니다. 엔티티 빈의 경우만 데이터 소스를 갖습니다.

Deployment Descriptor 에디터를 통해 엔티티 빈에 대한 새 데이터 소스를 지정하고 데이터 트랜잭션의 독립 레벨을 설정할 수 있습니다.



다음과 같은 방법으로 새 데이터 소스를 Deployment Descriptor에 추가합니다.

1 Deployment Descriptor 툴바에서  아이콘을 클릭합니다.

   New DataSource 대화 상자가 나타납니다.

2 새 데이터 소스의 이름을 입력하고 OK를 선택합니다.

   새 데이터 소스가 구조 창의 트리에 추가됩니다.

3 트리에서 데이터 소스를 선택합니다.

4 새 데이터 소스에 대한 정보를 입력합니다.

   데이터 소스는 데이터 소스 이름, 데이터 소스의 URL 위치 및 필요한 경우 소스에 액세스하기 위한 사용자 이름과 암호에 의해 정의됩니다. 패널에는 JDBC 드라이버의 클래스 이름과 JDBC 속성도 들어 있습니다.

5 데이터 소스 연결을 지정했으면 Test Connection 버튼을 선택할 수 있습니다.

   Deployment Descriptor 에디터는 지정된 데이터 소스와의 연결을 시도합니다. 메시지 로그에 그 결과가 게시됩니다.

독립 레벨 설정

*독립 레벨*이란 용어는 다중 사용자 데이터베이스에서 서로 방해하지 않도록 되어 있는 다중 인터리브 트랜잭션에 대한 정도를 나타냅니다. 가능한 트랜잭션 위반은 다음과 같습니다.

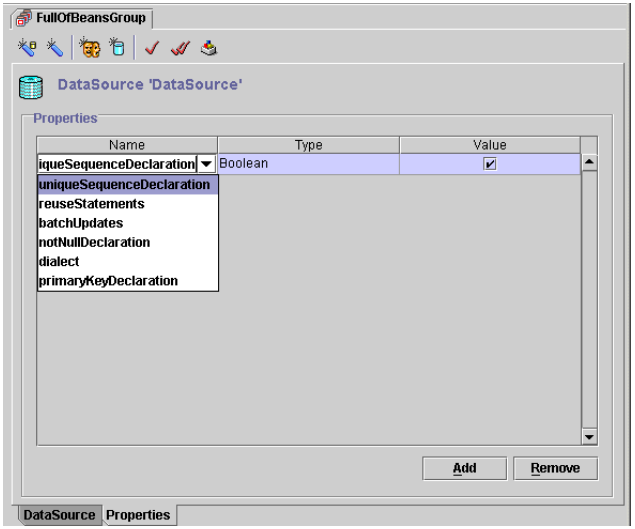
- **Dirty read:** Transaction t1은 행을 수정하고 Transaction t2는 행을 읽습니다. 이제 t1은 롤백을 수행하고 t2는 이전에 존재하지 않았던 행을 발견합니다.
- **Non-repeatable read:** Transaction t1은 행을 검색합니다. 그런 다음 transaction t2는 이 행을 업데이트하고 t1은 동일한 행을 다시 검색합니다. Transaction t1은 이제 동일한 행을 두 번 검색하고 그에 대한 서로 다른 두 값을 발견합니다.
- **Phantoms:** Transaction t1은 특정 검색 조건을 만족시키는 행 집합을 읽습니다. 그런 다음 transaction t2는 동일한 검색 조건을 만족시키는 행을 하나 이상 삽입합니다. transaction t1이 읽기를 반복하면 이전에 존재하지 않았던 행이 나타납니다. 이러한 행을 phantoms라고 합니다.

데이터 소스의 트랜잭션 독립 레벨을 설정하거나 변경하려면 Isolation Level 드롭다운 리스트의 독립 레벨 중 하나를 선택합니다.

Attribute	Syntax	Description
Uncommitted	TRANSACTION_READ_UNCOMMITTED	세 가지 위반을 모두 허용합니다.
Committed	TRANSACTION_READ_COMMITTED	non-repeatable 읽기와 phantoms을 허용하지만 dirty read를 허용하지 않습니다.
Repeatable	TRANSACTION_REPEATABLE_READ	phantoms을 허용하지만 다른 두 가지 위반은 허용하지 않습니다.
Serializable	TRANSACTION_SERIALIZABLE	세 가지 위반을 모두 허용하지 않습니다.

데이터 소스 속성 설정

Deployment Descriptor 에디터에서 데이터 소스를 선택할 때 Properties 탭이 Data Source 탭과 함께 나타납니다. Properties 패널을 통해 Borland CMP(Container-Managed Persistence) 엔진에 영향을 주는 속성을 설정할 수 있습니다.



다음과 같은 방법으로 데이터 소스의 속성을 수정합니다.

1 구조 창에서 데이터 소스를 선택합니다.

2 Properties 탭을 클릭합니다.

3 Properties 패널에 있는 드롭다운 리스트에서 설정할 속성을 선택합니다.

   Type 값은 Name 목록에서의 선택에 따라 자동으로 설정됩니다.

4 속성의 Value 열에서 값을 선택합니다.

5 Add 버튼을 클릭하여 새 행을 추가함으로써 추가 속성을 더한 다음 해당 새 속성의 Name 및 Value 항목을 선택합니다.

다음은 가능한 속성입니다.

Property	Description
uniqueSequence	CMP 엔진이 기본 키 열의 고유 순서를 선언할지 여부를 결정합니다. 보통 적절한 열을 기본 키로 선언하여 이 작업을 수행합니다(primaryKeyDeclaration 참조).
batchUpdates	CMP 엔진에서 데이터베이스에 대한 업데이트를 일괄 처리할지 여부를 나타냅니다. 이렇게 하면 많은 엔티티를 업데이트하는 트랜잭션의 성능이 크게 향상되며 드라이버가 일괄 처리 업데이트를 지원하는 경우에만 사용합니다. 안타깝게도 아직까지는 대부분의 드라이버가 일괄 처리 업데이트를 지원하지 않습니다. 기본값은 False입니다.

Property	Description
reuseStatements	CMP 엔진에서 트랜잭션 간에 준비된 문장을 재사용할지 여부를 결정합니다. 준비된 문장을 재사용하면 성능에 많은 영향을 주므로 JDBC 드라이버에서 재사용하도록 명시된 경우에만 준비된 문장을 사용합니다. 기본값은 True입니다.
notNullDeclaration	Null일 수 없는 Java 필드(예: `int` 또는 `float`)를 Non-Null 열에 매핑해야 할지 여부를 결정합니다. 기본값은 True입니다.
dialect	JDataStore, Oracle, Informix 또는 기타 데이터 소스 등 데이터 소스의 타입을 결정합니다. Value 드롭다운 리스트에서 dialect 값을 선택합니다. 이 필드를 설정하지 않으면 CMP 엔진은 JDataStore에 대해서만 테이블을 생성합니다. 기본값은 none입니다.
primaryKeyDeclaration	CMP 엔진에서 테이블의 기본 키 열을 기본 키가 되도록 선언할지 여부를 결정합니다. 일부 데이터베이스는 기본 키 선언을 지원하지 않습니다. 기본값은 True입니다.

보안 역할과 메소드 권한 추가

Deployment Descriptor 에디터를 통해 배포 디스크립터의 보안 역할을 생성 또는 편집할 수 있습니다. 보안 역할을 생성한 다음 엔터프라이즈 빈의 홈과 원격 인터페이스의 메소드를 이러한 역할과 연결하여 애플리케이션의 보안 뷰를 정의할 수 있습니다.

이 단원에서는 Deployment Description 에디터를 사용하여 보안 역할을 생성하고 엔터프라이즈 빈 메소드를 역할에 할당하는 방법에 대해 설명합니다. 14- 9페이지의 "Security Role References 패널" 단원에서는 Roles 패널을 사용하여 사용자 그룹 및 사용자 계정을 역할에 할당하는 방법에 대해 설명합니다.

Deployment Descriptor에서의 보안 역할 정의는 옵션입니다.

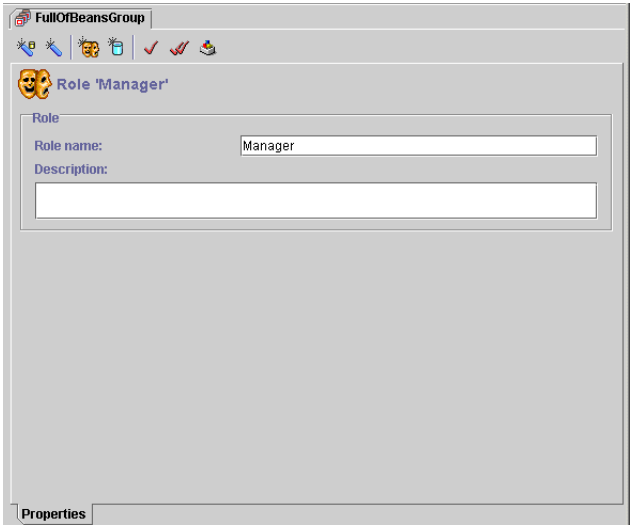
보안 역할 생성

다음과 같은 방법으로 Deployment Descriptor에서 보안 역할을 생성합니다.

1 Deployment Descriptor 툴바에서  아이콘을 클릭합니다.

2 나타나는 대화 상자에서 새 보안 역할의 이름을 입력하고 OK를 선택합니다.

구조 창의 Security Roles 노드 아래에 새 역할이 나타납니다. 새 역할이 보이지 않으면 Security Roles 노드를 확장합니다. 또한, 역할의 Properties 패널이 나타납니다.



Properties 패널에서 새 역할에 대한 설명을 입력할 수 있습니다. 이 설명은 옵션입니다.

메소드 권한 지정

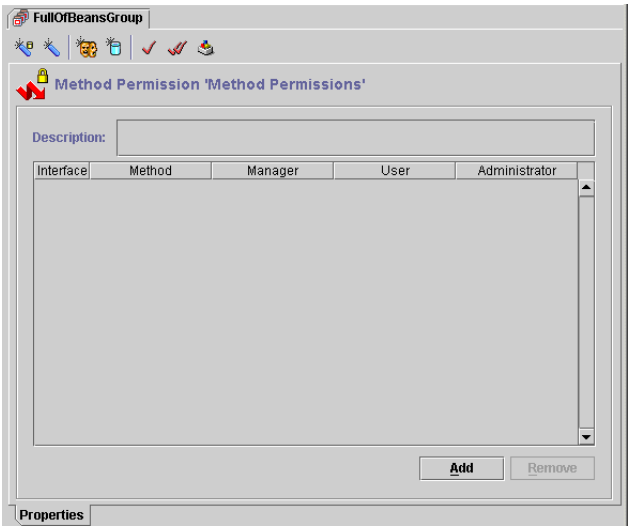
보안 역할을 정의했으면 엔터프라이즈 빈의 홈과 원격 인터페이스에 보안 역할을 호출할 수 있는 메소드를 지정할 수 있습니다.

보안 역할을 빈의 홈 또는 원격 인터페이스의 메소드에 연결하지 않아도 됩니다. 이런 경우, Deployment Descriptor에 정의된 보안 역할 중 어떤 것도 이러한 메소드를 호출하도록 허용되지 않습니다.

다음과 같은 방법으로 메소드 권한을 할당합니다.

1 구조 창의 빈 노드를 확장하여 Method Permissions 하위 노드를 나타냅니다.

2 Method Permission 노드를 선택하여 Properties 패널을 표시합니다.



각각의 정의된 보안 역할은 열 제목으로 나타납니다.

3 Add 버튼을 클릭하여 패널에 행을 추가합니다.

4 새 행에서 Home, Remote 또는 *를 선택하여 Interface 필드의 드롭다운 리스트로부터 모두 표시합니다.

5 Method 드롭다운 리스트에서 호출할 권한을 부여할 메소드를 선택하거나, 모든 메소드를 호출하는 권한을 나타내는 *를 선택합니다.

6 지정된 메소드를 호출하는 권한을 제공할 각 보안 역할의 체크 박스를 선택 표시합니다.

7 최종 단계로서 Description 필드에 옵션 설명을 입력하여 행이 정의하는 권한에 대해 설명할 수 있습니다.

WebLogic 및 WebSphere 속성 보기 및 편집

Deployment Descriptor 에디터를 사용하여 WebLogic과 WebSphere에 고유한 공급업체 지정 요소 일부를 보고 편집할 수 있습니다. WebLogic 또는 WebSphere 서버 중 하나가 대상 애플리케이션 서버인 경우 Deployment Descriptor 에디터는 컨텐트 창의 아래에 EJB Properties 탭을 표시합니다.

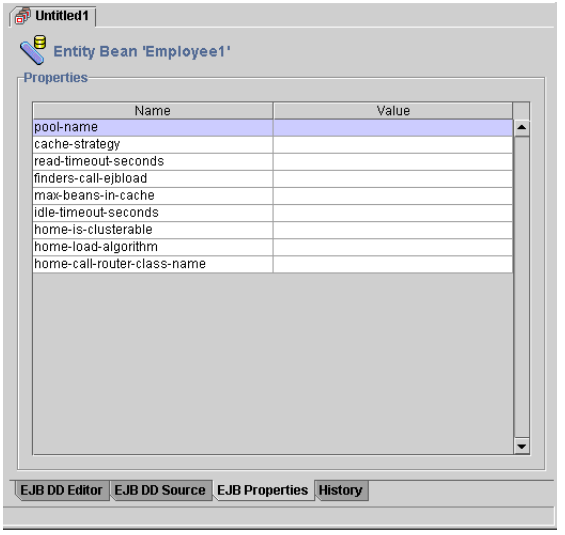
다음과 같은 방법으로 WebLogic 또는 WebSphere 특정 속성을 표시합니다.

1 프로젝트 창에서 EJB 그룹 노드를 더블 클릭합니다.

2 구조 창에서 사용할 엔터프라이즈 빈을 클릭합니다.

3 EJB Properties 탭을 클릭합니다.

EJB Properties 패널은 WebLogic 또는 WebSphere 서버 특정 속성의 테이블을 표시합니다. 예를 들어, WebLogic Server 6.0 애플리케이션 서버를 선택했을 때 EJB Properties 패널은 다음과 같이 나타납니다.



이 테이블을 사용하여 속성 값을 보거나 편집할 수 있습니다. 왼쪽에 있는 열을 사용하여 수정할 속성의 값을 입력합니다. 속성 값이 애플리케이션 서버 특정 Deployment Descriptor에 저장됩니다.

Descriptor 내용 확인

디스크립터 파일의 편집 작업을 완료한 후 정보 내용이 올바른 형식인지와 필요한 빈 클래스 파일이 존재하는지 등을 확인할 수 있습니다.

디스크립터 내용을 확인하려면 Deployment Descriptor 에디터 툴바에서 Verify 아이콘()을 클릭합니다. Verify를 선택하기 전에  아이콘을 클릭하여 Verify 옵션을 설정할 수 있습니다.

다음 사항을 확인합니다.

- 디스크립터가 EJB 1.1 사양에 부합되는지 확인합니다.
- Deployment Descriptor에서 참조하는 클래스가 EJB 1.1 사양에 부합되는지 확인합니다.

Set Class Path는 소스 코드 리플렉션(reflection)이 아닌 바이트 코드 리플렉션에 사용됩니다. 바이트 코드 리플렉션을 사용하려면 `Borland/AppServer/console/plugins/iaspt.ini` 파일의 Preferences 섹션에 다음과 같은 문장을 추가합니다.

```
UseSourceReflection=false
```

클래스 경로를 설정하려면  아이콘을 클릭하여 나타나는 대화 상자에서 새 클래스 경로를 설정합니다.

Chapter

15

EJB 컴포넌트용 DataExpress 사용

JBuilder에 들어 있는 몇 가지 컴포넌트를 통해 사용자는 엔티티 빈에서 DataExpress DataSet으로 데이터를 가져오고(providing) DataExpress DataSet으로부터 데이터를 다시 엔티티 빈에 저장(resolving)할 수 있습니다. 이러한 EJB 컴포넌트용 DataExpress를 통해 세션 빈 랩 엔티티 빈 디자인 패턴을 쉽게 구현할 수 있습니다. 일반적으로 클라이언트는 디자인 패턴을 사용하여 엔티티 빈에 직접 액세스하지 않고 세션 빈을 사용하여 엔티티 빈에 액세스합니다. 엔티티 빈과 함께 위치해 있는 세션 빈은 단일 트랜잭션 내의 엔티티 빈에 모든 호출을 만든 다음 한꺼번에 모든 데이터를 반환합니다. DataSet은 세션 빈에서 클라이언트로 그리고 클라이언트에서 세션 빈으로 다시 데이터를 전송하는 방법을 제공합니다. 데이터는 유선(wire)으로 단 한 번 클라이언트에 공급(provide)한 다음 서버에 있는 엔티티 빈의 변경 사항을 해결(resolve)하기 위해 다시 한번 전송되기 때문에 성능이 향상됩니다.

또한 이러한 컴포넌트를 사용하면 dbSwing 또는 InternetBeans Express 같은 DataExpress-aware 비주얼 컴포넌트를 사용하여 클라이언트 애플리케이션을 보다 쉽게 구축할 수 있습니다. DataExpress에 대한 전반적인 설명은 *Database Application Developer' Guide*의 "JBuilder 데이터베이스 애플리케이션 이해"를 참조하십시오.

이 장에서는 이러한 컴포넌트를 사용하여 서버에 배포된 엔티티 빈에서 사용자의 클라이언트 애플리케이션으로 데이터를 전송하고 다시 클라이언트 애플리케이션에서 엔티티 빈으로 데이터를 전송하는 방법에 대해 설명합니다. 이 장에서 사용하는 코드의 출처는 `/JBuilder5/samples/Ejb/ejb.jpx` 샘플 프로젝트입니다. 샘플이 액세스하는 데이터는 `Employee`의 데이터 저장소에 저장되어 있습니다. 이 샘플은 `Employee` 데이터를 가진 엔티티 빈을 만듭니다. 이 샘플은 또한 `Employee`에서 데이터를 가져오는 `Personnel` 세션 빈을 만든 다음 이를 클라이언트로 보냅니다. 클라이언트는 데이터를

Employee 엔티티 빈 인스턴스의 데이터를 해결(resolve)하는 Personnel로 다시 보냅니다.

DataExpress EJB 컴포넌트

DataExpress EJB 컴포넌트 4개는 컴포넌트 팔레트의 EJB 페이지에 있습니다. 사용자는 Inspector를 사용하여 속성 및 이벤트를 설정할 때 UI 디자이너에서 이러한 컴포넌트를 사용할 수 있습니다. 비주얼하게 작업할 수 없는 추가 클래스를 사용자의 코드에서 호출할 수 있습니다. 클래스에 관한 모든 정보는 API 참조를 참고하십시오.

서버용 컴포넌트

서버에 배포된 세션 빈에서 사용되는 컴포넌트 팔레트의 EJB 페이지에 있는 두 가지 컴포넌트는 EntityBeanProvider와 EntityBeanResolver 컴포넌트입니다. EntityBeanProvider는 서버에 배포된 엔티티 빈에서 데이터를 공급(provide)하고 EntityBeanResolver는 이 엔티티 빈의 데이터를 해결(resolve)합니다. 사용자가 만든 세션 빈에 이 컴포넌트를 추가하면 엔티티 빈으로부터 데이터를 공급(provide)하고 엔티티 빈의 데이터를 해결(resolve)할 수 있습니다.

클라이언트용 컴포넌트

클라이언트측에서 사용되는 EJB 페이지의 컴포넌트 두 가지는 EjbClientDataSet과 SessionBeanConnection입니다. EjbClientDataSet는 SessionBeanConnection에서 참조되는 세션 빈으로부터 나온 데이터를 공급(provide)하고 세션 빈의 변경 내용을 해결(resolve)합니다. SessionBeanConnection은 서버의 세션 빈에 참조를 보유하며 그 세션 빈에서 나온 데이터셋(dataset)을 공급(provide)하고 해결(resolve)하는 메소드 이름을 갖고 있습니다.

엔티티 빈 생성

EJB Entity Modeler를 사용하여 원하는 데이터에 액세스하는 엔티티 빈을 만들기 시작하십시오. 샘플 프로젝트는 Employee와 Department 엔티티 빈을 모두 만들지만 이 장에서는 Employee만을 다룹니다.

서버측 세션 빈 생성

서버에 상주하게 될 세션 빈을 만듭니다. Enterprise JavaBean 마법사를 사용하여 상태 없는(stateless) 세션 빈을 만들어 보십시오. 이후의 다음 단원에서는 이 세션 빈에 EntityBeanProvider와 EntityBeanResolver 클래스를

추가하게 됩니다. 이 클래스는 직렬화할 수 없으므로 상태 없는 세션 빈에 두는 것이 더 쉽습니다. Borland Application Server에서 상태 없는 세션 빈은 부동화(passivated)되지 않기 때문입니다. 애플리케이션용 상태 있는(stateful) 세션 빈이 필요한 경우, 상태 있는 세션 빈이 상태 없는(stateless) 세션 빈을 참조하게 하거나 상태 있는 세션 빈이 활성화되었을 때 EntityBeanProvider와 EntityBeanResolver를 다시 인스턴스화해야 합니다.

다음은 PersonnelBean이라는 결과적인 빈 클래스입니다.

```
public class PersonnelBean implements SessionBean {
  private SessionContext sessionContext;
  public void ejbCreate() {
  }
  public void ejbRemove() throws RemoteException {
  }
  public void ejbActivate() throws RemoteException {
  }
  public void ejbPassivate() throws RemoteException {
  }
  public void setSessionContext(SessionContext sessionContext) throws RemoteException {
    this.sessionContext = sessionContext;
  }
}
```

Design 탭을 클릭하여 UI 디자이너를 표시합니다.

세션 빈에 provider 컴포넌트 및 resolver 컴포넌트 추가

컴포넌트 팔레트의 EJB 페이지에서 EntityBeanProvider와 EntityBeanResolver를 세션 빈에 추가합니다. 또한 데이터셋 컴포넌트를 추가하여 엔티티 빈에서 모아진 데이터가 클라이언트로 전송되기 전에 이를 저장하고 클라이언트에서 다시 전송한 데이터를 저장하여야 합니다. 컴포넌트 팔레트의 DataExpress 페이지로부터 TableDataSet 컴포넌트를 추가하고 TableDataSet를 적당한 이름으로 지정합니다.

다음은 PersonnelBean 상단부의 모습입니다. TableDataSet은 employeeDataSet으로 이름이 재지정되었습니다.

```
public class PersonnelBean implements SessionBean {
  private SessionContext sessionContext;
  EntityBeanProvider entityBeanProvider = new EntityBeanProvider();
  EntityBeanResolver entityBeanResolver = new EntityBeanResolver();
  TableDataSet employeeDataSet = new TableDataSet();
  ...
```

Inspector를 사용하여 TableDataSet의 provider 및 resolver 속성을 새로 추가된EntityBeanProvider 및 EntityBeanResolver 컴포넌트에 각각 설정합니다. 그러면 jbInit()메소드에 두 개의 메소드가 새로 만들어 집니다.

```
employeeDataSet.setProvider(entityBeanProvider);
employeeDataSet.setResolver(entityBeanResolver);
```

샘플 프로젝트는 대신 setSessionContext()메소드에서 이러한 메소드를 보여 줍니다. 샘플 프로젝트를 그대로 모방하여 사용하고 싶으면 사용자가

직접 setSessionContext()에 메소드 호출을 추가할 수 있습니다. 어떠한 방식을 써도 좋습니다.

이 클래스의 멤버에 대해, 액세스하려는 데이터를 포함하는 엔티티 빈의 홈 인터페이스에 참조를 추가합니다. 다음 예제에서 참조는 굵게 표시된 Employee 엔티티 빈의 홈 인터페이스에 대한 것입니다.

```
public class PersonnelBean implements SessionBean {
  private SessionContext sessionContext;
  EntityBeanProvider entityBeanProvider = new EntityBeanProvider();
  EntityBeanResolver entityBeanResolver = new EntityBeanResolver();
  TableDataSet employeeDataSet = new TableDataSet();
  EmployeeHome employeeHome;
  ...
```

setSessionContext() 메소드 작성

세션 빈의 sessionContext() 메소드에 try 블록을 추가합니다. 메소드를 다음과 같이 수정합니다.

```
public void setSessionContext(SessionContext sessionContext)
  throws RemoteException {
    this.sessionContext = sessionContext;
  try {
    Context context = new InitialContext();
    Object object = context.lookup("java:comp/env/ejb/Employee");
    employeeHome = (EmployeeHome) PortableRemoteObject.narrow(object,
          EmployeeHome.class);
    entityBeanProvider.setEjbHome(employeeHome);
    entityBeanResolver.setEjbHome(employeeHome);
  }
  catch (Exception ex) {
    throw new EJBException(ex);
  }
}
```

setSessionContext()가 EntityBeanProvider 및 EntityBeanResolver 컴포넌트 내의 ejbHome 속성 값을 Employee 엔티티 빈의 홈 인터페이스 이름으로 설정한다는 점에 유의하십시오.

배포 디스크립터에 EJB 참조 추가

검색이 작동하도록 하려면 배포 디스크립터의 Personnel에 EJB 참조를 추가해야 합니다. Deployment Descriptor 에디터를 사용할 수 있습니다.

1. 프로젝트 창에서 EJB 그룹 노드를 더블 클릭하십시오. 샘플 프로젝트의 경우에는 personnel.ejbgrpx에 해당합니다.

   Deployment Descriptor 에디터가 나타납니다.

2. 구조 창의 Personnel 빈을 클릭합니다.

3. Deployment Descriptor 에디터의 EJB References 탭을 클릭합니다.

4 Add 버튼을 클릭하여 찾고자 하는 데이터가 들어 있는 엔티티 빈에 참조를 추가합니다.

5 참조 이름을 입력합니다. 샘플 프로젝트의 경우, 이 이름은 ejb/Employee 입니다.

6 IsLink 체크 박스를 선택 표시합니다.

7 Link 드롭다운 목록에서 엔티티 빈을 지정합니다.

나머지 데이터는 자동으로 채워질 것입니다.

provider 메소드 및 resolver 메소드 추가

세션 빈에 provider 및 resolver라는 두 가지 메소드를 추가해야 합니다. 이 두 가지 메소드의 이름은 사용자가 EjbClientDataSet 컴포넌트의 methodName 속성 값으로 지정한 값을 사용합니다. 그러므로 PersonnelBean에 대한 provider는 provideEmployee()가 되고 resolver는 resolveEmployee()가 됩니다.

Provider는 유선(wire)으로 전송할 수 있는 데이터를 엔티티 빈에서 데이터셋으로 공급(provide)하는 EntityBeanConnection 클래스의 메소드를 호출합니다. (provideEmployee() 메소드는 다음과 같습니다.

```
public DataSetData [] provideEmployee(RowData [] parameterArray,
  RowData [] masterArray) {
    return EntityBeanConnection.provideDataSets(new StorageDataSet []
     {employeeDataSet}, parameterArray, masterArray);
}
```

Resolver는 엔티티 빈의 업데이트를 해결(resolve)하는 EntityBeanConnection 클래스의 메소드를 호출합니다. 다음은 resolveEmployee()가 표시되는 방법입니다.

```
public DataSetData [] resolveEmployee(DataSetData[] dataSetDataArray) {
  return EntityBeanConnection.saveChanges(dataSetDataArray,
        new DataSet [] {employeeDataSet});
}
```

그런 다음 이 메소드를 원격 인터페이스에 추가합니다. 가장 간단한 방법은 BeansExpress를 사용하는 것입니다. 에디터에서 빈(bean) 소스 파일을 열어 놓은 상태에서 Bean 탭, Methods 탭을 차례로 클릭한 다음 사용자가 추가한 두 메소드의 이름 옆에 있는 체크 박스를 선택 표시합니다. 이제 세션 빈의 원격 인터페이스를 점검하여 두 메소드가 정의되었는지 확인할 수 있습니다.

```
public interface Personnel extends EJBObject {
  public com.borland.dx.dataset.DataSetData[]
     providePersonnel(com.borland.dx.ejb.RowData[] parameterArray,
         com.borland.dx.ejb.RowData[] masterArray) throws RemoteException;
  public com.borland.dx.dataset.DataSetData[]
     resolvePersonnel(com.borland.dx.dataset.DataSetData[] dataSetDataArray)
         throws RemoteException;
}
```

Finder 메소드 호출

사용자는 EntityBeanProvider에 어떤 엔티티 빈을 공급(provide)할지 알려 주어야 합니다. 이렇게 하려면 EntityBeanProvider에 이벤트를 추가합니다.

1. UI 디자이너에서 구조 창의 EntityBeanProvider를 선택합니다.
2. Inspector의 Events 탭을 클릭하고 findEntityBeans 이벤트 옆의 비어 있는 열을 더블 클릭합니다. 새 이벤트가 추가됩니다.

   다음과 같은 이벤트 코드가 ejb.jpx 프로젝트에 나타나는 것과 똑같이 나타납니다.

```
entityBeanProvider1.addEntityBeanFindListener(new
   com.borland.dx.ejb.EntityBeanFindListener() {
      public void findEntityBeans(EntityBeanFindEvent e) {
        entityBeanProvider1_findEntityBeans(e);
      }
   });

...

void entityBeanProvider_findEntityBeans(EntityBeanFindEvent e) {

}
```

3. 새 이벤트 핸들러에 finder 메소드를 추가하여 원하는 엔티티 빈을 반환합니다. 아래에서 굵게 표시된 것이 추가된 코드입니다.

```
void entityBeanProvider_findEntityBeans(EntityBeanFindEvent e) {
  try {
     e.setEntityBeanCollection(employeeHome.findAll());
   }
  catch (Exception ex) {
    throw new EJBException(ex);
   }
}
```

위의 예에서 이벤트 핸들러는 findAll() 메소드를 호출하여 모든 엔티티 빈을 반환합니다. 원하는 finder는 무엇이든지 호출할 수 있습니다. EntityBeanProvider의 parameterRow 속성을 사용하여 호출할 finder 및/또는 전달할 매개변수를 동적으로 결정할 수 있습니다.

해결(resolve)을 위해서는 기본적으로 EntityBeanResolver가 업데이트와 삭제의 적용 방법을 결정합니다. 그러나 어떤 create() 메소드를 호출하고 어떤 매개변수를 전달할 것인지 결정할 방법이 없으므로 EntityBeanResolver는 새로운 엔티티 빈을 생성하는 방법을 자동으로 결정하지 못합니다. 따라서 데이터 소스에 행을 추가하려면 직접 create 이벤트를 추가하여 필요한 로직을 제공해야 합니다. Inspector를 사용하여 뼈대 create 이벤트 코

드를 사용자의 세션 빈에 추가할 수 있습니다. ejb.jpx 샘플 프로젝트에서 create 이벤트의 예제를 볼 수 있습니다. 또한 EntityBeanResolver에서 사용할 수 있는 다른 이벤트를 사용하여 기본 동작을 오버라이드할 수 있습니다.

애플리케이션 서버에 세션과 엔티티 빈을 배포합니다. 빈 배포에 대한 자세한 내용은 13- 5페이지의 "애플리케이션 서버에 배포"를 참조하십시오.

클라이언트측 구축

엔티티 빈과 엔티티 빈에 액세스하는 세션 빈을 만들고 대상 애플리케이션 서버에 배포했으므로 이제 클라이언트를 구축할 준비가 되었습니다.

다음 단계를 따릅니다.

1. 데이터 모듈을 만듭니다. File|New|Data Module을 선택합니다.
2. 컴포넌트 팔레트에서 EjbClientDataSet을 선택하고 EjbClientDataSet을 데이터 모듈에 추가합니다.
3. 컴포넌트 팔레트에서 SessionConnectionBean을 선택하고 데이터 모듈에 추가합니다.
4. Inspector에서 EjbClientDataSet의 sessionBeanConnection 속성을 SessionBeanConnection 컴포넌트의 이름으로 설정합니다.
5. Inspector에서 EjbClientDataSet 컴포넌트의 methodName 속성에 대한 이름을 지정합니다.

   methodName 속성은 데이터를 공급(provide)하고 해결(resolve)하는 메소드의 이름 지정 방식을 결정합니다. 예를 들어, methodName에 대한 Employee의 값을 지정하는 경우 데이터를 공급(provide)하고 해결(resolve)하는 세션 빈 메소드는 provideEmployee()와 resolveEmployee()가 됩니다. 나중에 provideEmployee()와 resolveEmployee()를 사용자가 만든 세션 빈에 추가해야 합니다.
6. Inspector 또는 소스 코드에서 직접 SessionBeanConnection 컴포넌트의 jndiName 속성을 설정합니다. 또는 사용자가 만들 세션 빈의 원격 인터페이스 이름을 sessionBeanRemote 속성 값으로 대신 지정할 수 있습니다.

또한 Inspector를 사용하여 creating 이벤트를 SessionBeanConnection에 추가할 수 있습니다. 이벤트 핸들러에 추가한 코드에서 JNDI 검색이 발생한 후에 세션 빈이 생성되는 것을 제어할 수 있습니다. 매개변수를 필요로 하는 홈 인터페이스에 create() 메소드를 호출하려면 일반적으로 creating 이벤트를 추가해야 합니다.

다음은 /JBuilder5/samples/Ejb/ejb.jpx 샘플 프로젝트에서 볼 수 있는 결과 소스 코드입니다.

```
import com.borland.dx.dataset.*;
import com.borland.dx.ejb.*;

public class PersonnelDataModule implements DataModule {
  private static PersonnelDataModule myDM;
  SessionBeanConnection sessionBeanConnection = new SessionBeanConnection();
  EjbClientDataSet personnelDataSet = new EjbClientDataSet();

  public PersonnelDataModule() {
  try {
    jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
  private void jbInit() throws Exception {
  try {
      sessionBeanConnection.setJndiName("Personnel");
      sessionBeanConnection.addCreateSessionBeanListener(new
            com.borland.dx.ejb.CreateSessionBeanListener() {
        public void creating(CreateSessionBeanEvent e) {
          sessionBeanConnection_creating(e);
        }
    });
      personnelDataSet.setSessionBeanConnection(sessionBeanConnection);
      personnelDataSet.setMethodName("Personnel");
    }
  catch (Exception ex) {

    }
  }
  public static PersonnelDataModule getDataModule() {
    if (myDM == null) {
      myDM = new PersonnelDataModule();
    }
    return myDM;
  }
  public com.borland.dx.ejb.SessionBeanConnection getSessionBeanConnection() {
    return sessionBeanConnection;
  }
  public com.borland.dx.ejb.EjbClientDataSet getPersonnelDataSet() {
    return personnelDataSet;
  }

  void sessionBeanConnection_creating(CreateSessionBeanEvent e) {

  }
}
```

관계 처리

EntityBeanProvider는 자동으로 관계를 평평하게(flatten) 합니다. 예를 들어 Dept를 반환하는 getDept() 메소드가 들어 있는 Employee 엔티티 빈을 갖고 있고 Dept가 엔티티 빈 원격인 경우, Employee 엔티티 빈의 모든 필드와 Dept 엔티티 빈의 모든 필드(각 엔티티 빈의 기본 키를 가진 숨김 열 포함)가 들어 있는 DataSet 이 만들어 집니다. Dept.ejbPrimaryKey를 제외한 그 외의 다른 Dept 필드는 읽기 전용입니다.

EntityBeanProvider가 관련된 엔티티 빈의 홈을 동적으로 결정할 수 없으므로 일대일 관계일 경우에는 이벤트 리스너(event listener)를 EntityBeanProvider에 추가하여 변경 내용을 해결(resolve)해야 합니다. 샘플 ejb.jpx 프로젝트는 관계 처리에 대한 예제는 보여 주지 않습니다.

샘플 프로젝트

이제까지 클라이언트와 서버 간에 데이터를 효과적으로 상호 전송하는 방법에 대해 살펴 보았습니다. 샘플 /JBuilder5/samples/Ejb/ejb.jpx 프로젝트는 dbSwing 컨트롤을 사용하는 Java 클라이언트와 JSP 기술이 결합된 InternetBeansExpress를 사용하는 Web 클라이언트에서 위에서 설명한 기술을 사용하는 방법을 보여 줍니다. 사용자는 실제 데이터를 가지고 작업할 수 있게 됩니다. 샘플 프로젝트 실행에 대한 전체적인 지침은 프로젝트의 Ejb.html 페이지에서 확인하십시오.

Chapter

16

세션 빈 개발

JBuilder의 EJB 도구를 이용하여 enterprise beans 및 그 지원 인터페이스를 매우 간단하게 만들 수 있습니다. 그러나 이러한 클래스와 인터페이스에 대한 요구 사항을 이해해야 JBuilder가 만드는 파일을 수정할 수 있고 JBuilder가 수행하는 작업을 이해할 수 있습니다. 다음에 이어지는 장을 통해 이러한 작업을 이해하게 될 것입니다.

세션 빈은 보통 단일 클라이언트 세션의 수명 동안 존재합니다. 세션 빈의 메소드는 빈을 사용하는 클라이언트에 대한 작업이나 프로세스를 수행합니다. 세션 빈은 클라이언트와 연결되어 있는 동안에만 지속됩니다. 어떤 의미로 보면 세션 빈은 EJB 서버의 클라이언트를 나타냅니다. 세션 빈은 보통 클라이언트에게 서비스를 제공합니다. Data store에 있는 영구 데이터를 사용해야 할 필요가 없는 경우 통상적으로 세션 빈을 사용하고 있는 것입니다.

세션 빈 타입

세션 빈에는 두 가지 타입이 있는데, 메소드 호출 사이의 상태 정보를 유지하는 세션 빈은 *상태 있는(stateful)* 빈이라고 하고, 그렇지 못한 것은 *상태 없는(stateless)* 빈이라고 합니다.

상태 있는(stateful) 세션 빈

상태 있는 세션 빈은 단일 클라이언트에 의해 사용되는 객체이고 클라이언트를 대신하여 상태를 유지합니다. 쇼핑 카트 세션 빈을 예로 들어 봅니다. 온라인 상점에서 고객이 구입하려는 항목을 선택하면 쇼핑 카트 세션 빈 객체 안의 목록에 선택한 항목을 저장하여 "쇼핑 카트"에 추가합니다. 고객이 항목을 구입할 준비가 되면 이 목록을 사용하여 전체 금액을 계산합니다.

상태 없는(stateless) 세션 빈

상태 없는 세션 빈은 특정 클라이언트의 상태를 유지하지 않습니다. 그러므로 이 세션 빈을 여러 클라이언트에서 사용할 수 있습니다. sortList() business 메소드를 포함하는 sort 세션 빈을 예로 들어 봅니다. 클라이언트는 sortList()를 호출하여 정렬되지 않은 항목 목록을 전달합니다. 그러면 sortList()는 정렬된 목록을 클라이언트에게 다시 전달합니다.

세션 빈 클래스 작성

다음과 같이 세션 빈 클래스를 만듭니다.

- javax.ejb.SessionBean 인터페이스를 구현하는 클래스를 만듭니다.
- 하나 이상의 ejbCreate() 메소드를 구현합니다. 상태 없는 세션 빈을 만드는 경우, 클래스는 매개변수 없는 단 한 개의 ejbCreate() 메소드를 구현합니다. 이미 빈의 홈 인터페이스를 생성했다면 홈 인터페이스 안에 있는 각 create() 메소드와 시그너처가 동일한 ejbCreate() 메소드가 빈에 지정되어야 합니다.
- 빈에 지정할 business 메소드를 정의하고 구현합니다. 빈에 대한 원격 인터페이스를 이미 만들었다면 메소드는 원격 인터페이스에서 지정한 것과 똑같이 정의되어야 합니다.

JBuilder의 Enterprise JavaBean 마법사는 홈 및 원격 인터페이스를 생성하는 것을 비롯하여 위와 같은 작업을 시작할 수 있습니다. 이 마법사는 SessionBean 인터페이스를 확장하는 클래스를 생성하고 SessionBean 메소드의 비어 있는 구현을 작성합니다. 빈이 구현을 요청할 경우 구현을 작성해야 합니다. 다음 단원은 메소드와 그 사용법을 설명합니다.

SessionBean 인터페이스 구현

SessionBean 인터페이스는 모든 세션 빈이 구현해야 하는 메소드를 정의합니다. 이 인터페이스는 EnterpriseBean 인터페이스를 확장합니다.

```
package javax.ejb;
public interface SessionBean extends EnterpriseBean {
    void setSessionContext(SessionContext sessionContext)
        throws EJBException, RemoteException;
    void ejbRemove() throws EJBException, RemoteException;
    void ejbActivate() throws EJBException, RemoteException;
    void ejbPassivate() throws EJBException, RemoteException;
}
```

SessionBean 인터페이스의 메소드는 세션 빈의 수명 주기와 밀접하게 관련되어 있습니다. 다음 표에서 그 용도를 설명합니다.

메소드	설명
setSessionContext()	세션 컨텍스트를 설정합니다. 빈의 컨테이너는 이 메소드를 호출하여 세션 빈 인스턴스와 그 컨텍스트를 연결합니다. 세션 컨텍스트 인터페이스는 세션이 실행되는 컨텍스트의 런타임 속성에 액세스하는 메소드를 제공합니다. 보통 세션 빈은 데이터 필드에 그 컨텍스트를 보관합니다.
ejbRemove()	세션 객체가 곧 제거될 것임을 알립니다. 컨테이너는 원격 또는 홈 인터페이스의 remove() 메소드를 호출한 클라이언트의 결과인 상태 있는 세션 빈을 제거할 때마다 이 메소드를 호출합니다.
ejbActivate()	상태 있는 세션 객체가 활성화되었음을 알립니다.
ejbPassivate()	상태 있는 세션 객체가 컨테이너에 의해 곧 비활성화될 것임을 알립니다.

ejbActivate() 메소드와 ejbPassivate() 메소드를 통해 상태 있는 세션 빈은 리소스를 관리할 수 있습니다. 자세한 내용은 16- 5페이지의 "상태 있는 (stateful) 빈"를 참조하십시오.

business 메소드 작성

Enterprise bean 클래스 안에서 JBuilder의 코드 에디터를 사용하여 빈에게 필요한 비즈니스 메소드의 전체적인 구현을 작성합니다. 클라이언트에서 이러한 메소드를 사용할 수 있게 하려면 빈 클래스에서 선언한 것과 똑같이 빈의 원격 인터페이스에서 메소드를 선언해야 합니다. JBuilder의 Bean 디자이너를 사용하면 이 작업을 수행할 수 있습니다. 10- 9페이지의 "원격 인터페이스를 통한 비즈니스 메소드 노출"를 참조하십시오.

하나 이상의 ejbCreate() 메소드 추가

Enterprise JavaBean 마법사를 사용하여 enterprise bean을 시작하면 마법사가 매개변수를 가지지 않는 클래스 빈에 ejbCreate() 메소드를 추가합니다. 그리고 매개변수를 가지는 ejbCreate() 메소드를 추가할 수 있습니다. 상태 없는 세션 빈은 상태를 유지하지 않으므로 매개변수 없는 ejbCreate() 메소드가 하나 이상 필요하지 않지만, 상태 있는 세션 빈은 매개변수를 가지는 ejbCreate() 메소드가 하나 이상 필요합니다. 추가로 매개변수를 가진 ejbCreate() 메소드를 작성할 때, 다음 규칙에 유의하십시오.

- 각 ejbCreate()는 public으로 선언되어야 합니다.
- 각 메소드는 void를 반환해야 합니다.
- ejbCreate() 메소드의 매개변수는 빈의 원격 인터페이스에서 해당 create() 메소드와 동일한 개수와 형식을 가져야 합니다. 상태 없는 세션 빈의 경우 매개변수 없는 ejbCreate()가 하나만 존재할 수 있습니다.

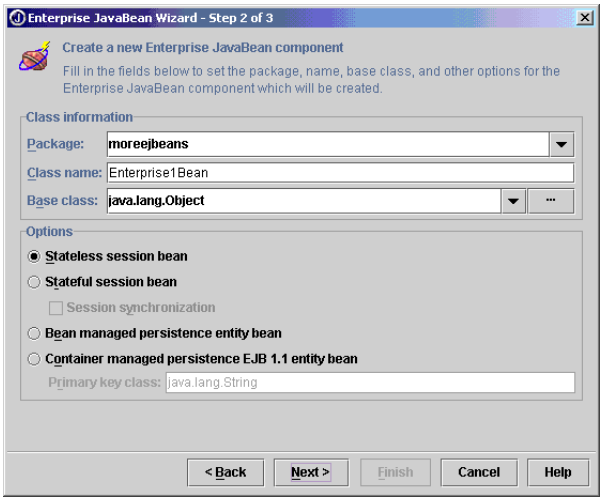
다음은 세션 빈의 모든 ejbCreate() 메소드에 대한 시그너처입니다.

```
public void ejbCreate( <zero or more parameters> ) {
// implementation
}
```

ejbCreate() 메소드는 javax.ejb.CreateException과 같은 애플리케이션 특정 예외 및 기타 예외를 발생시킬 수는 있지만 반드시 예외를 발생시킬 필요는 없습니다. Enterprise JavaBean 마법사는 javax.ejb.CreateException을 발생시키는 ejbCreate() 메소드를 생성합니다.

JBuilder를 사용하여 세션 빈을 만드는 방법

JBuilder의 Enterprise JavaBean 마법사를 사용하면 다음과 같이 마법사의 두 번째 페이지에서 Stateless Session Bean 옵션 아니면 Stateful Session Bean 옵션을 선택하여 세션 빈을 만들기 시작할 수 있습니다.



Enterprise JavaBean 마법사는 enterprise bean 클래스를 만드는 것은 물론 만들면서 동시에 빈의 홈 및 원격 인터페이스도 만듭니다. 이런 방법으로 ejbCreate() 메소드가 항상 void를 반환하는 동안 홈 인터페이스의 create() 메소드는 원격 인터페이스를 반환하는 것을 확인할 수 있습니다.

빈 클래스에서 비즈니스 메소드를 작성한 후 Bean 디자이너를 사용하여 빈의 원격 인터페이스에서 정의된 것들 중에서 원하는 것을 지정할 수 있습니다. 클라이언트 애플리케이션은 원격 인터페이스에서 정의된 메소드만 액세스할 수 있습니다. 일단 클라이언트가 호출하려는 메소드를 지정하면 Bean 디자이너는 원격 인터페이스에서 메소드를 정의합니다.

이미 enterprise bean 클래스를 모두 가지고 있지만 홈과 원격 인터페이스가 없는 경우에는 JBuilder의 EJB Interfaces 마법사를 사용하여 그 인터페이스를 만들 수 습니다. 메소드 시그너처는 홈 및 원격 인터페이스의 EJB 1.1 사양과 부합하므로 잘못 만들어지는 것을 염려하지 않아도 됩니다.

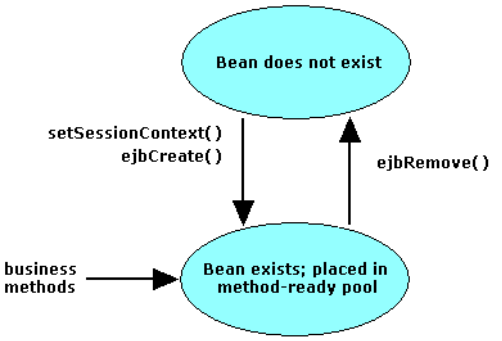
JBuilder의 EJB 도구에 대한 내용은 Chapter 16 "세션 빈 개발"을 참조하십시오

세션 빈의 수명

상태 있는 세션 빈과 상태 없는 세션 빈은 서로 다른 수명 주기를 가집니다. 사용자는 각 세션 빈의 수명 주기 동안 어떤 일이 발생하는 지 이해해야 합니다.

상태 없는(stateless) 빈

상태 없는 세션 빈의 수명은 클라이언트가 세션 빈의 홈 인터페이스의 `create()` 메소드를 호출할 때 시작됩니다. 컨테이너는 세션 빈의 새 인스턴스를 만들고 객체 참조를 클라이언트에 반환합니다.



생성 프로세스 동안 컨테이너는 `SessionBean` 인터페이스의 `setSessionContext()` 메소드를 호출하고 세션 빈 구현의 `ejbCreate()` 메소드를 호출합니다. 새로운 빈 객체는 클라이언트가 사용할 준비가 된 상태 없는 빈 객체의 풀에 합류합니다. 상태 없는 세션 객체는 클라이언트 특정 상태를 유지하지 않기 때문에 컨테이너는 아무 빈 객체나 지정하여 들어오는 메소드 호출을 처리할 수 있습니다. 컨테이너가 세션 빈 객체 풀에서 객체를 제거하면 컨테이너는 빈 객체의 `ejbRemove()` 메소드를 호출합니다.

홈/원격 인터페이스의 `create()` 메소드 또는 `remove()` 메소드를 호출하면 상태 없는 세션 빈 풀에서 상태 없는 세션 빈 객체를 추가하거나 제거하지 않는다는 점에 유의하십시오. 컨테이너는 상태 없는(stateless) 빈의 수명 주기를 조절합니다.

상태 있는(stateful) 빈

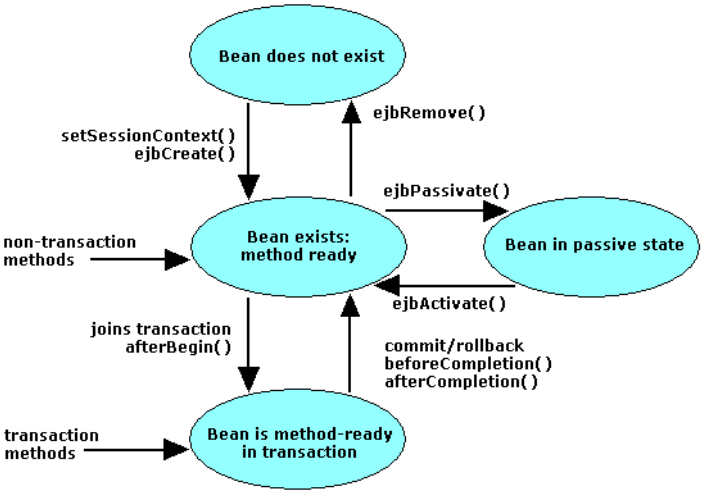
클라이언트가 세션 빈의 홈 인터페이스의 `create()` 메소드를 호출하면 상태 있는 세션 빈의 수명이 시작됩니다. 컨테이너는 세션 빈의 새로운 인스턴스를 만들어 초기화하고 객체 참조를 클라이언트에 반환합니다.

생성 프로세스 동안 컨테이너는 SessionBean 인터페이스의 setSessionContext() 메소드를 호출하고 세션 빈 구현의 ejbCreate() 메소드를 호출합니다. 빈 제공자는 이러한 메소드를 사용하여 세션 빈을 초기화할 수 있습니다.

이제 세션 빈의 상태는 메소드 준비(method-ready) 상태로서, 비트랜잭션 작업을 수행할 수 있다는 것을 의미하거나 트랜잭션 작업을 위해 트랜잭션에 포함될 수 있다는 것을 의미합니다. 빈은 다음 세 가지가 발생할 때까지 메소드 준비(method-ready) 상태로 남아 있습니다.

- 빈이 트랜잭션을 입력합니다.
- 빈이 제거됩니다.
- 빈이 부동화(passivate)됩니다.

클라이언트가 원격 또는 홈 인터페이스의 remove() 메소드를 호출하면 컨테이너는 세션 빈 객체에 해당하는 ejbRemove() 메소드를 호출합니다. 빈 제공자는 애플리케이션 특정 클린업 코드를 이 메소드에 넣습니다. ejbRemove()가 완료되면 빈 객체를 더 이상 사용할 수 없습니다. 클라이언트가 빈 객체에서 메소드를 호출하려고 하면 컨테이너는 java.rmi.NoSuchObjectException을 발생시킵니다.



컨테이너는 세션 빈 인스턴스를 비활성화시킬 수 있습니다. 세션 객체가 잠시 idle 상태이거나 컨테이너에 메모리가 더 필요한 경우와 같이 리소스 관리 상의 이유에서 일반적으로 비활성화됩니다. 컨테이너는 빈의 `ejbPassivate()` 메소드를 호출하여 빈을 비활성화시킵니다. 빈 인스턴스가 비활성화되는 것을 부동화(passivation)라고 하는데 이 때 컨테이너는 참조 정보와 세션 객체의 상태를 디스크에 저장하고 빈에 할당된 메모리를 해제합니다. 부동화가 발생하기 직전에 실행하려는 작업이 몇 가지 있을 경우 `ejbPassivate()`에 코드를 추가할 수 있습니다.

컨테이너는 `ejbActivate()` 메소드를 호출하여 빈 객체를 다시 활성화합니다. 이것은 클라이언트가 부동화된 세션 빈에 메소드를 호출할 때 발생합니다. 활성화되어 있는 동안 컨테이너는 세션 객체를 메모리에 다시 만들고 그 상태를 저장합니다. 빈이 다시 활성화된 직후에 어떤 작업을 하려면 `ejbActivate()` 메소드에 코드를 추가합니다.

트랜잭션에서의 메소드 준비(method-ready) 상태

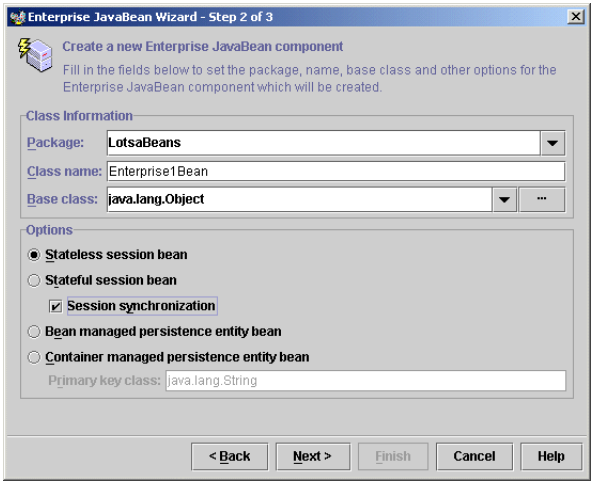
클라이언트가 트랜잭션 컨텍스트에서 세션 빈 객체에 메소드를 호출하면 컨테이너는 새로운 트랜잭션을 시작하거나 빈 객체를 기존 객체에 포함시킵니다. 빈은 트랜잭션에서의 메소드 준비(method-ready) 상태를 입력합니다. 트랜잭션의 수명 주기 중에는 트랜잭션 동기화 포인트라고 하는 시기가 있는데, 이 때 세션 빈 객체에게 다가오는 트랜잭션 이벤트를 공지하고 객체는 그 전에 필요한 작업을 수행합니다.

SessionSynchronization 인터페이스

세션 빈이 포함된 트랜잭션의 상태에 대한 공지를 받으려는 경우 세션 빈은 `SessionSynchronization` 인터페이스를 구현할 수 있습니다. 컨테이너 관리 트랜잭션을 사용하는 상태 있는 세션 빈에서만 `SessionSynchronization`을 구현할 수 있는데 그 사용 여부는 옵션입니다. `SessionSynchronization`의 메소드는 컨테이너가 빈에 만든 콜백으로서 트랜잭션 안에 포인트를 표시합니다. 다음은 `SessionSynchronization` 인터페이스입니다.

```
public interface javax.ejb.SessionSynchronization
{
    public abstract void afterBegin() throws RemoteException;
    public abstract void beforeCompletion() throws RemoteException;
    public abstract void afterCompletion(boolean completionStatus) throws
      RemoteException;
}
```

Enterprise JavaBean 마법사는 이러한 메소드를 빈 클래스에 추가할 수 있습니다. 마법사를 사용하여 Session Synchronization 체크 박스를 선택하면 마법사는 빈 클래스에서 내용이 없는 세 가지 메소드를 선언합니다.



다음 표는 각 메소드들을 간단하게 설명한 것입니다.

메소드	설명
afterBegin()	빈 인스턴스가 트랜잭션에서 곧 사용될 것임을 알립니다. `afterBegin()`에서 작성한 모든 코드는 트랜잭션 범위 안에서 발생합니다.
beforeCompletion()	빈 인스턴스에 트랜잭션이 곧 커밋될 것임을 알립니다. 빈에 캐시된 값이 있으면 `beforeCompletion()`를 사용하여 그 값을 데이터베이스에 기록합니다. 필요하면 세션 빈에서 `SessionContext` 인터페이스의 `setRollbackOnly()` 메소드를 호출하여 트랜잭션을 강제로 롤백시키기 위해 `beforeCompletion()` 메소드를 사용할 수 있습니다.
afterCompletion()	빈 인스턴스에 트랜잭션이 완료되었음을 알립니다. 트랜잭션이 커밋되면 매개변수 `completionStatus`는 true로 설정됩니다. 트랜잭션이 롤백되면 매개변수는 false로 설정됩니다.

`SessionSynchronization` 메소드를 사용하는 방법은 다음과 같습니다. 클라이언트는 원격 인터페이스에서 정의된 트랜잭션 비즈니스 메소드를 호출하는데 이 메소드는 빈 객체를 트랜잭션 준비(transaction-ready) 상태로 둡니다. 컨테이너는 빈 객체에서 `afterBegin()` 메소드를 호출합니다. 나중에 트랜잭션이 커밋되면 컨테이너는 `beforeCompletion()`을 호출한 다음 커밋이 성공하면 `afterCompletion(true)` 메소드를 호출합니다. 트랜잭션이 롤백되거나 커밋에 실패하면 컨테이너는 `afterCompletion(false)`을 호출합니다. 그러면 세션 빈 객체는 다시 메소드 준비(method-ready) 상태가 됩니다.

트랜잭션에서 세션 빈 사용에 대한 자세한 내용은 Chapter 20 "트랜잭션 관리"를 참조하십시오

쇼핑 카트 세션 빈

이 예제에서는 온라인 상점에서 가상 쇼핑 카트가 되는 상태 있는 세션 enterprise bean Cart의 사용에 대해 설명합니다. 고객은 항목을 골라 가상 쇼핑 카트에 넣습니다. 잠시 이 사이트를 나갔다가 돌아올 수도 있으며 더 많은 상품을 쇼핑 카트에 추가할 수 있습니다. 고객은 원하면 언제든지 쇼핑 카트에 담긴 항목을 볼 수 있습니다. 구입할 준비가 되면 쇼핑 카트에 있는 항목을 구입합니다.

/Borland/AppServer/examples/ejb/cart 디렉토리에서 쇼핑 카트 예제에 대한 전체 코드를 볼 수 있습니다.

쇼핑 카트 예제 파일 검사

쇼핑 카트 예제는 여러 가지 다른 파일로 구성됩니다. 이 단원에서는 스스로 작성하거나 세션 빈에 관한 흥미로운 것에 대하여 설명하는 파일에 초점을 맞춥니다. 쇼핑 카트 디렉토리에서 여기서 설명하지 않는 일부 파일은 저절로 생성된 파일(스텁, 뼈대 코드, 기타 CORBA 코드)입니다.

주요 파일들은 다음과 같습니다.

- CartHome.java. Cart 세션 빈에 대한 홈 인터페이스를 정의하는 파일입니다.
- Cart.java. Cart 세션 빈에 대한 원격 인터페이스를 정의하는 파일입니다.
- CartBean.java. 세션 빈 클래스입니다.
- Item.java. CartBean에 의해 사용되는 항목 파일입니다. 이 항목은 가상 쇼핑 카트에 있는 항목의 가격과 제목을 알 수 있는 메소드를 제공합니다.
- Cart.xml. 배포 정보 파일입니다. EJB 1.1에서 XML 파일은 enterprise bean의 배포 정보를 포함합니다.
- 예외 파일. 이 파일은 CartBean에 의해 발생된 애플리케이션 특정 예외를 정의합니다. 예외는 다음 세 가지이며 각 예외들은 각각의 해당 파일에 정의되어 있습니다.
  - ItemNotFoundException
  - PurchaseProblemException
  - CardExpiredException
- CartClient.java. 클라이언트 애플리케이션입니다.

Cart 세션 빈

이 단원에서는 Cart 세션 빈을 구현하는 방법에 대해 자세히 설명합니다. Enterprise JavaBean 마법사를 사용하여 시작할 수 있습니다. 마법사의 두 번째 페이지에서 클래스 이름에 CartBean을 입력하고 Stateful Session

Bean 옵션을 선택한 다음 Next를 클릭합니다. 제시된 홈 인터페이스 이름, 원격 인터페이스, 빈 홈 이름을 확인하고 Finish를 선택합니다. 마법사는 다음과 같이 뼈대 빈 클래스를 만듭니다.

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

public class CartBean implements SessionBean {

  private SessionContext sessionContext;

  public void ejbCreate() throws CreateException {
  }

  public void ejbRemove() throws RemoteException {
  }

  public void ejbActivate() throws RemoteException {
  }

  public void ejbPassivate() throws RemoteException {
  }

  public void setSessionContext(SessionContext context) throws RemoteException {
    sessionContext = context;
  }
}
```

세션 빈 클래스는 public으로 정의해야 합니다. 세션 빈 클래스를 final이나 abstract로 정의할 수 없습니다. 빈 클래스는 `SessionBean` 인터페이스를 구현해야 합니다.

세션 빈은 Java 객체이므로 인스턴스 변수를 가질 수 있습니다. `CartBean`은 클래스에 추가한 네 개의 인스턴스 변수를 가집니다. 네 개의 변수는 다음과 같이 private로 선언됩니다.

```
private Vector _items = new Vector();
private String _cardHolderName;
private String _creditCardNumber;
private Date _expirationDate;
```

이러한 선언을 Enterprise JavaBean 마법사가 클래스에 추가한 `sessionContext` 변수 뒤에 놓습니다.

`_items` 변수는 쇼핑 카트 객체가 소유한 항목을 유지합니다. 이 변수는 벡터, 즉 항목의 컬렉션입니다. 나머지 세 가지 인스턴스 변수는 온라인 고객의 신용 카드 정보를 저장합니다.

필요한 메소드 추가

세션 빈은 `SessionBean` 인터페이스에 의해 정의된 네 가지 메소드를 구현해야 합니다. EJB 컨테이너는 세션 빈의 수명 주기에서 특정 포인트에 있는 빈 인스턴스에 이 메소드들을 호출합니다. 빈 제공자는 최소한 내용이 없

는 메소드를 구현해야 합니다. 빈 제공자는 필요하면 이러한 메소드에 코드를 추가할 수 있습니다. 그러나 CartBean 세션 빈은 이러한 메소드에 코드를 추가하지 않습니다. 네 가지 메소드는 다음과 같습니다.

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

Enterprise JavaBean 마법사는 이러한 네 개의 메소드를 전부 추가합니다. setSessionContext() 메소드에서 마법사는 sessionContext 인스턴스 변수에 컨텍스트 매개변수의 값을 할당합니다.

컨테이너는 setSessionContext() 메소드를 호출하여 빈 인스턴스와 그 세션 컨텍스트를 연결합니다. 빈은 이러한 세션 컨텍스트 참조를 대화 상태의 일부로 유지하도록 선택할 수 있지만 반드시 선택해야 하는 것은 아닙니다. Enterprise JavaBean 마법사를 사용한 경우 세션 컨텍스트 참조는 유지됩니다. 세션 빈은 세션 컨텍스트를 사용하여 환경 변수 또는 홈 인터페이스와 같은 자체 정보를 얻을 수 있습니다.

컨테이너는 빈 인스턴스를 수동 상태로 놓아야 할 때 빈 인스턴스에 ejbPassivate() 메소드를 호출합니다. 컨테이너는 빈을 부동화할 때 빈의 현재 상태를 보조 저장소에 기록합니다. 컨테이너는 나중에 빈을 활성화시킬 때 이 상태를 복원합니다. 컨테이너가 실제로 빈 인스턴스를 부동화시키기 직전에 ejbPassivate() 메소드를 호출하기 때문에 빈 제공자는 이 메소드에 코드를 추가하여 원하는 특별한 변수 캐싱을 수행할 수 있습니다. 컨테이너는 같은 방법으로 빈 인스턴스를 수동 상태에서 활성 상태로 돌리기 직전에 빈 인스턴트에 ejbActivate() 메소드를 호출합니다. 컨테이너가 빈을 활성화하면 계속 지속되는 모든 상태 값이 복원됩니다. 여기서 ejbActivate() 메소드에 코드를 추가하도록 선택할 수 있습니다. CartBean은 이러한 구현을 비워 둘 수 있습니다.

세션 빈이 생성자를 구현할 필요는 없지만 최소한 하나의 ejbCreate() 메소드를 구현해야 합니다. 이 메소드는 새로운 빈 인스턴스를 만드는 생성자 역할을 합니다. 상태 있는 세션은 하나 이상의 ejbCreate() 메소드를 구현할 수 있습니다. 각 ejbCreate() 메소드는 그 매개변수에서만 다릅니다.

CartBean 예제는 다음과 같이 세 개의 매개변수를 가지는 ejbCreate() 메소드를 선언합니다.

```
public void ejbCreate(String cardHolderName, String creditCardNumber,
      Date expirationDate) throws CreateException {
      _cardHolderName = cardHolderName;
      _creditCardNumber = creditCardNumber;
}
```

컨테이너는 빈 인스턴스를 제거하기 직전에 ejbRemove() 메소드를 호출합니다. 빈을 제거하기 전에 실행할 애플리케이션 특정 코드를 추가할 수는 있지만 반드시 필요한 것은 아닙니다. 앞에 나온 CartBean 예제는 ejbRemove() 메소드의 몸체를 비워 둡니다.

비즈니스 메소드 추가

비즈니스 메소드가 따라야 하는 규칙에는 다음과 같이 몇 가지가 있습니다.

- 메소드 이름은 EJB 아키텍처에 의해 예약된 이름과 충돌을 피하기 위해 접두사 **ejb**로 시작하지 않아야 합니다.
- 각 메소드는 public으로 선언되어야 합니다.
- 메소드는 final 또는 static으로 선언될 수 없습니다 .
- 매개변수 및 반환 타입은 적합한 RMI-IIOP 타입이어야 합니다.
- throws 절에는 `javax.ejb.EJBException` 예외를 포함할 수 있고 임의의 애플리케이션 특정 예외를 정의할 수 있습니다.

`Cart` 예제에서는 다섯 가지 비즈니스 메소드가 구현됩니다. 세션 빈 클래스 메소드의 시그너처(메소드 이름, 매개변수 개수, 매개변수 타입, 반환 타입)은 원격 인터페이스와 일치해야 합니다. 발생한 것을 확인하려면 Bean 디자이너를 사용하여 빈 클래스에 추가한 비즈니스 메소드를 빈의 원격 인터페이스로 이동할 수 있습니다.

각 비즈니스 메소드는 프로그램의 흐름을 따라가도록 돕기 위해 메소드 이름과 현재 수행되는 작업을 표시하는 코드 한 줄을 포함합니다.

`addItem()` 메소드는 다음과 같이 쇼핑 카트에 항목 목록을 유지하는 벡터에 항목을 추가합니다.

```
public void addItem(Item item) {
    System.out.println("\taddItem(" + item.getTitle() + "):" + this);
    _items.addElement(item);
}
```

`removeItem()` 메소드는 더 복잡합니다. 메소드는 항목 목록에 있는 요소들을 둘러본 다음 제거할 항목의 클래스와 제목이 목록에 있는 항목의 클래스와 제목과 일치하는지 확인합니다. 이 메소드는 정말 제거하려는 항목을 제거하고 있는 지 확인합니다. 일치하는 항목을 찾지 못하면 메소드는 `ItemNotFoundException`을 발생시킵니다.

```
public void removeItem(Item item) throws ItemNotFoundException {
    System.out.println("\tremoveItem(" + item.getTitle() + "):" + this);
    Enumeration elements = _items.elements();
    while(elements.hasMoreElements()) {
      Item current = (Item) elements.nextElement();
      // items are equal if they have the same class and title
      if(item.getClass().equals(current.getClass()) &&
         item.getTitle().equals(current.getTitle())) {
         _items.removeElement(current);
    return;
    }
}
```

`getTotalPrice()` 메소드는 전체 가격을 0으로 초기화한 다음 항목 목록을 둘러보고 각 요소의 가격을 전체 가격에 추가합니다. 이 메소드는 전체 가격을 페니 단위로 반올림합니다.

```
public float getTotalPrice() {
    System.out.println("\tgetTotalPrice():" + this);
```

```
    float totalPrice = 0f;
    Enumeration elements = _items.elements();
    while(elements.hasMoreElements()) {
      Item current = (Item) elements.nextElement();
      totalPrice += current.getPrice();
    }
    // round to the nearest lower penny
    return (long) (totalPrice * 100) / 100f;
}
```

클라이언트와 서버 간에 전달된 모든 데이터 타입은 직렬화되어야 합니다. 즉, 모든 데이터 타입은 java.io.Serializable 인터페이스를 구현해야 합니다. Cart 예제에서 빈은 항목 목록을 클라이언트로 반환합니다. 직렬화할 수 있는 제약 조건이 없는 경우 _items.elements()를 사용하여 항목 벡터의 컨텐트를 반환할 수 있습니다. 그러나 _items.elements()는 직렬화할 수 없는 Enumeration 객체를 반환합니다. 프로그램은 이러한 문제를 피하기 위해 com.inprise.ejb.util.VectorEnumeration(_items)이라는 클래스를 구현합니다. 이 클래스는 벡터를 사용하여 벡터의 컨텐트에 대해 직렬화할 수 있는 실제적인 열거를 반환합니다. CartBean 객체는 직렬화할 수 있는 벡터를 클라이언트에게 전달하고 전달된 직렬화할 수 있는 벡터를 클라이언트측으로부터 받습니다. getContents() 메소드는 Java Enumeration과 직렬화할 수 있는 VectorEnumeration 간에 대화를 수행합니다.

```
public java.util.Enumeration getContents() {
    System.out.println("\tgetContents():" + this);
    return new com.inprise.ejb.util.VectorEnumeration(_items);
}
```

purchase() 메소드는 다음 작업을 수행해야 합니다.

1. 현재 시간을 알려 줍니다.
2. 신용 카드의 만료일과 현재 날짜를 비교합니다. 만료일이 현재 날짜보다 이전일 경우 메소드는 CardExpiredException 애플리케이션 예외를 발생시킵니다.
3. 목록 업데이트를 비롯하여 요금을 신용 카드 회사에 알려 주고 항목의 배송을 초기화하여 구입 절차를 완료합니다. (Cart 예제에서는 실제로 이 중에서 어떤 것도 구현되지 않습니다.) 어느 시점에서 오류가 발생하면 구입 절차가 완료되지 않으며 메소드는 PurchaseProblemException 예외를 발생시킵니다.

```
public void purchase() throws PurchaseProblemException {
    System.out.println("\tpurchase():" + this);
    // make sure the credit card has not expired
    Date today = Calendar.getInstance().getTime();
    if(_expirationDate.before(today)) {
       throw new CardExpiredException("Expiration date: " + _expirationDate);
    }
    // complete purchasing process
    // throw PurchaseProblemException if an error occurs
}
```

CartBean에는 CartBean과 카드 소지자 이름을 인쇄하는 toString() 메소드가 들어 있습니다.

```
// method to print out CartBean and the name of card holder
  public String toString() {
```

```
    return "CartBean[name=" + _cardHolderName + "]";
  }
```

Item 클래스

CartBean 예제는 Item 클래스를 사용합니다. Item은 public이고 java.io.Serializable를 확장하며, 직렬화된 데이터는 다음과 같이 유선(wire)으로 전달됩니다.

```
package shoppingcart;
public class Item implements java.io.Serializable {
  private String _title;
  private float _price;
  public Item(String title, float price) {
    _title = title;
    _price = price;
  }
  public String getTitle() {
    return _title;
  }
  public float getPrice() {
    return _price;
  }
}
```

예외

Cart 예제에는 다음과 같이 세 가지 예외 클래스가 있습니다. 모두 Java 클래스 Exception을 확장한 것입니다.

```
public class ItemNotFoundException extends Exception {
  public ItemNotFoundException(String message) {
      super(message);
  }
}
public class PurchaseProblemException extends Exception {
  public PurchaseProblemException(String message) {
      super(message);
  }
}
public class CardExpiredException extends Exception {
  public CardExpiredException(String message) {
      super(message);
  }
}
```

필요한 인터페이스

Enterprise beans는 항상 홈 인터페이스와 원격 인터페이스라는 두 개의 인터페이스를 가집니다. 이 예제에서 Cart 세션 빈은 Cart라는 public EJB 원격 인터페이스와 CartHome이라는 홈 인터페이스를 가집니다.

Enterprise JavaBean 마법사를 사용하면 홈 및 원격 인터페이스와 빈 클래스가 동시에 만들어집니다. 이미 세션 빈을 가졌지만 인터페이스를 가지지 않은 경우 EJB Interfaces 마법사를 사용하십시오. 마법사를 사용하려면 코드 에디터에서 enterprise bean의 소스 코드를 표시하고 Wizards|EJB Interfaces를 선택합니다. 마법사의 프롬프트에 입력을 마치면 마법사는 enterprise bean에 대한 홈 및 원격 인터페이스를 만듭니다.

Enterprise JavaBean 마법사와 EJB Interfaces 마법사의 사용에 대한 더 자세한 내용은 Chapter 16 "세션 빈 개발"을 참조하십시오

홈 인터페이스

다른 모든 홈 인터페이스처럼 CartHome은 EJBHome 인터페이스를 확장합니다. 홈 인터페이스가 빈 인스턴스를 만들고 찾는 두 가지 작업을 잠재적으로 수행할 수 있는 반면, 세션 빈은 빈 인스턴스만 만들면 됩니다. 세션 빈은 클라이언트 세션이 끝나면 항상 사라집니다. 따라서 고객이 온라인 상점에 들어올 때는 예를 들어 CartBean 인스턴스가 없다는 이유로 CartBean 인스턴스를 찾을 필요가 없습니다. 엔티티 빈이 여러 클라이언트에 의해 사용되고 데이터 엔트리가 존재하는 한 지속하기 때문에 엔티티 빈에 대한 홈 인터페이스에서만 find 작업을 수행합니다. 다음은 CartHome 인터페이스입니다.

```
// CartHome.java
public interface CartHome extends javax.ejb.EJBHome {
  Cart create(String cardHolderName, String creditCardNumber,
              java.util.Date expirationDate)
    throws java.rmi.RemoteException,javax.ejb.CreateException;
}
```

CartHome 인터페이스는 매우 간단하므로 create() 메소드 하나만 정의합니다. 이 인터페이스는 상태 있는 세션 빈이기 때문에 하나 이상의 create() 메소드가 존재할 수 있습니다. 이 예제에서 CartHome 인터페이스의 create() 메소드는 cardHolderName, creditCardNumber, expirationDate이라는 세 개의 매개변수를 가집니다.

클라이언트가 create() 메소드를 호출하여 쇼핑 카트를 요청하면 컨테이너는 그 사용자 전용 쇼핑 카트 하나를 만들어 줍니다. 클라이언트는 쇼핑 카트를 간헐적으로 사용할 수 있지만 세션 빈은 고객이 종료하고 세션 빈 인스턴스가 제거될 때까지 그 클라이언트에 대해 활성화 상태에 남아 있습니다.

이러한 메소드가 트랜잭션의 컨텍스트 안에 있는지 여부에 상관 없이 상태 있는 세션 빈은 메소드 호출을 통해 상태를 유지합니다. 이 상태는 빈 객체의 의해 전달되는 데이터입니다. 이 데이터는 객체의 수명 동안 빈 객체와

연결된 상태로 남아 있습니다. 세션이 끝나면 컨테이너는 메모리에서 세션 빈의 상태를 플러시합니다.

create() 메소드는 EJB 사양에 정의되어 있는 규칙을 따릅니다. 즉, RMI 원격 예외인 java.rmi.RemoteException을 발생시키고 EJB create 예외인 javax.ejb.CreateException을 발생시킵니다. create() 메소드의 시그너처는 매개변수의 개수와 타입에 있어서 세션 빈 클래스에 있는 ejbCreate() 메소드의 시그너처와 일치합니다. create()의 반환 값은 Cart 원격 인터페이스입니다. 이것은 CartHome 인터페이스가 CartBean에 대한 팩토리로 기능하기 때문입니다. (CartBean 클래스에서 일치하는 ejbCreate() 메소드에 대한 반환 값은 void입니다.)

원격 인터페이스

Cart 세션 빈은 EJBObject 인터페이스를 확장하는 원격 인터페이스 Cart를 가집니다. EJBObject는 모든 원격 인터페이스에 대한 기본 인터페이스입니다. 이 인터페이스는 다음 작업을 수행할 수 있는 메소드를 정의합니다.

- 세션 빈에 대한 정보를 얻습니다.

  빈 객체가 다른 enterprise bean 객체와 동일한지 검사합니다. (또한 엔티티 빈에 대한 기본 키를 얻을 수 있지만 세션 빈에 적용되지 않습니다.)

- 세션 빈에 대한 참조 또는 핸들을 얻습니다.

  빈 인스턴스에 대한 직렬화 핸들의 빈 홈 인터페이스에 대한 참조를 얻을 수 있습니다. 핸들을 저장하여 나중에 검색한 다음 빈 인스턴스에 대한 참조를 다시 얻기 위해 사용할 수 있습니다.

- 빈 인스턴스를 제거합니다.

  EJBObject 인터페이스는 빈 인스턴스를 제거하는 remove() 메소드를 정의합니다.

Cart 원격 인터페이스는 EJBObject에서 상속한 메소드 외에 다섯 개의 비즈니스 메소드를 추가로 정의합니다. 이러한 비즈니스 메소드는 CartBean 세션 빈 클래스에서 구현된 메소드입니다. Cart 원격 인터페이스는 이러한 메소드를 클라이언트에게 보여 주기만 합니다. 클라이언트는 원격 인터페이스가 보여 주는 enterprise bean의 메소드만 호출할 수 있습니다. 보여진 비즈니스 메소드는 다음과 같습니다.

- addItem()은 항목을 쇼핑 카트에 추가합니다.
- removeItem()은 항목을 쇼핑 카트에서 제거합니다.
- getTotalPrice()는 모든 항목의 가격을 더하여 전체 가격을 반환합니다.
- getContents()는 쇼핑 카트에 있는 모든 항목을 모은 다음 볼 수 있거나 인쇄할 수 있는 목록으로 반환합니다.
- purchaseItems()는 항목을 구입하려고 시도합니다.

Cart 배포 디스크립터

EJB 1.1 사양에 따르면 배포 디스크립터는 XML 파일이어야 합니다. XML 파일은 Sun Microsystems에서 승인한 Document Type Definition(DTD)을 따릅니다. 배포 디스크립터에는 컨테이너의 enterprise bean 또는 애플리케이션 배포 방법을 설명하는 속성 집합이 들어 있습니다.

JBuilder의 Enterprise JavaBean 마법사를 사용하여 세션 빈을 만들 때 JBuilder는 또한 배포 디스크립터도 만듭니다. 그러면 Deployment Descriptor 에디터를 사용하여 필요에 따라 사용자 지정할 수 있습니다.

배포 디스크립터에는 빈의 속성을 나타내는 값을 가진 태그와 속성의 집합이 들어 있습니다. 예를 들어, `Cart` 예제에 대한 몇 가지 태그는 다음과 같습니다.

- <session> 태그는 enterprise bean이 세션 빈이라는 것을 지정합니다. <session> 태그 외에도 다음과 같은 태그가 사용됩니다.
  - <ejb-class> - 빈을 구현하는 세션 빈 클래스의 이름.
  - <home> - 홈 인터페이스 이름.
  - <remote> - 원격 인터페이스 이름.
  - <session-type> - 세션 빈이 상태있음(stateful) 또는 상태 없음(stateless)인지 여부.
  - <transaction-type> - 지속성(persistence)을 컨테이너가 관리(container-managed)하는지 빈이 관리(bean-managed)하는지 여부.
- <trans attribute> - 각 메소드에 대한 트랜잭션 속성.
- <timeout> - 세션 빈에 대한 타임 아웃 값.

Cart 세션 빈에 대한 배포 디스크립터 파일은 다음과 같습니다.

```
<ejb-jar>
  <øfÝÕ«¡¹Û¿Ã¡Ó ½Û>
    <ºººº«>
      <ºŠ³Ì>
        XML deployment descriptor cart session bean
      <ºŠ³Ì>
      <ejb-name>cart</ejb-name>
     <home>CartHome</home>
       <remote>Cart</remote>
       <ejb-class>CartBean</ejb-class>
       <session-type>Stateful</session-type>
       <transaction-type>Container</transaction-type>
    <ºººº«>
  <øfÝÕ«¡¹Û¿Ã¡Ó ½Û>
  <assembly-descriptor>
    <container-transaction>
      <³?º"µÂ>
        <ejb-name>cart</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
```

```
    </container-transaction>
    <container-transaction>
      <³?º"µÂ>
        <ejb-name>cart</ejb-name>
        <method-name>purchase</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

CartClient 프로그램은 Cart enterprise bean을 사용하는 클라이언트 애플리케이션입니다. 이 프로그램의 main() 루틴에는 모든 enterprise bean 클라이언트 애플리케이션이 구현해야 하는 요소들이 들어 있습니다. 이 프로그램은 다음과 같은 작업을 수행하는 방법을 보여줍니다.

- JNDI를 사용하여 빈의 홈 인터페이스를 찾습니다.
- 홈 인터페이스의 create() 메소드를 사용하여 새로운 원격 Cart 객체를 만듭니다.
- Cart 객체에 선언된 메소드를 호출합니다.

```
public static void main(String] args) throws Exception {
// get a JNDI context using the Naming service
javax.naming.Context context = new javax.naming.InitialContext();

Object objref = context.lookup("cart");
CartHome home = (CartHome)javax.rmi.PortableRemoteObject.narrow(objref, CartHome.class);
Cart cart;
{
  String cardHolderName = "Jack B. Quick";
  String creditCardNumber = "1234-5678-9012-3456";
    Date expirationDate = new GregorianCalendar(2001, Calendar.JULY,1).getTime();
    cart = home.create(cardHolderName, creditCardNumber, expirationDate);
}
Book knuthBook = new Book("The Art of Computer Programming", 49.95f);
cart.addItem(knuthBook);
   ... // create compact disc and add it to cart, then list cart contents
summarize(cart)
cart.removeItem(knuthBook);
   ... // add a different book and summarize cart contents
   try  {
    cart.purchase();
    }
    catch(PurchaseProblemException e) {
        System.out.println("Could not purchase the items:\n\t" + e);
    }
cart.remove();
}
```

main() 루틴은 객체를 조회하는 JNDI 컨텍스트에 의해 시작됩니다. 이 루틴은 초기 컨텍스트(Java 이름 지정 컨텍스트)를 생성합니다. 이 루틴은 표준 JNDI 코드입니다.

main()은 cart라는 CartHome 객체를 조회합니다. CosNaming에서 JNDI를 포함한 이름을 조회하도록 클라이언트에서 CosNaming 서비스를 호출하여 이름을 조회합니다. CosNaming 서비스는 클라이언트에 대한 객체 참조를 반환합니다. 이 예제에서 CosNaming 서비스는 COBRA 객체 참조를 반환합니다. 이 프로그램은 객체 참조에 대한

PortableRemoteObject.narrow() 작업을 수행하고 반환된 객체를 CartHome 타입으로 변환한 다음 변수 홈에 지정합니다. 이러한 호출이 분산 애플리케이션에서는 일반적입니다. 이 호출은 CORBA와 IIOP를 사용하여 다음 작업을 수행합니다.

- 서버와 대화합니다.
- CosNaming 조회를 수행합니다.
- CORBA 객체 참조를 얻습니다.
- 객체 참조를 클라이언트에게 반환합니다.

이 프로그램은 원격 cart 객체에 대한 참조를 선언하고 세 개의 create() 매개변수를 초기화한 다음 새로운 원격 cart 객체를 만듭니다.

이 프로그램은 책과 CD라는 두 개의 쇼핑 카트 항목을 만들고 이 항목들을 cart의 addItem() 메소드를 사용하여 쇼핑 카트에 추가합니다.

그러면 루틴은 summarize() 함수를 호출하여 현재 쇼핑 카트에 들어 있는 항목을 나열합니다. summarize() 함수는 Java의 Enumeration을 반환하는 카트의 getContents() 메소드를 사용하여 쇼핑 카트에 있는 요소 또는 항목을 검색합니다. 이 때 루틴은 Java Enumeration 인터페이스 메소드를 사용하여 Enumeration에 있는 각 요소를 읽어 들여 각 요소의 제목 및 가격을 가져 옵니다. 다음은 summarize() 함수 코드입니다.

```
static void summarize(Cart cart) throws Exception {
    System.out.println("======== Cart Summary =========")'
    Enumeration elements = cart.getContents();
    while(elements.hasMoreElements()) {
        Item current = (Item) elements.nextElement():
        System.out.println("Price: $" + current.getPrice() + "\t" +
          current.getClass().getName() + " title: " + current.getTitle());
    }
    System.out.println("Total: $" + cart.getTotalPrice());
    System.out.println("==============================");
}
```

그리고 나면 프로그램은 카트의 removeItem() 메소드를 호출하여 쇼핑 카트에서 항목을 제거합니다. 다른 항목을 추가하고 나서 쇼핑 카트 내용을 다시 요약합니다.

마지막으로 프로그램은 항목을 구입하려고 시도합니다. 구입 작업은 서버에서 구현되지 않았기 때문에 실패하고 PurchaseProblemException이 발생합니다.

사용자가 쇼핑 세션을 마쳤기 때문에 프로그램은 쇼핑 카트를 제거합니다. 쇼핑 카트를 제거하는 것이 프로그램 연습에는 바람직하지만 반드시 제거해야 하는 것은 아닙니다. 세션 빈은 세션 빈을 만든 클라이언트만을 위해서 존재합니다. 클라이언트가 세션을 마치면 컨터이너는 자동적으로 세션 빈 객체를 제거합니다. 컨테이너 또한 타임 아웃되었을 때 즉각적이지는 않지만 세션 빈 객체를 제거합니다.

CartClient 또한 책과 CD라는 두 가지 타입의 항목을 가진 일반적인 Item 클래스를 확장하는 코드를 포함합니다. 이 예제에서 사용된 클래스는 Book과 CompactDisc입니다.

Chapter

17

엔티티 빈 개발

엔티티 빈은 데이터베이스와 같은 영구적인 저장소에 저장된 데이터를 직접 나타냅니다. 엔티티 빈은 관계형 데이터베이스의 테이블 내에 있는 하나 이상의 행이나 객체 지향 데이터베이스의 엔티티 객체를 나타냅니다. 또한 엔티티 빈은 여러 테이블에 걸친 하나 이상의 행을 나타낼 수 있습니다. 데이터베이스에서 기본 키는 테이블의 행을 고유하게 식별합니다. 기본 키는 또한 유사한 방법으로 특정 엔티티 빈 인스턴스를 식별합니다. 관계형 데이터베이스 테이블의 각 열은 엔티티 빈의 인스턴스 변수를 나타냅니다.

엔티티 빈은 일반적으로 데이터베이스에 저장된 데이터를 나타내기 때문에 데이터가 있는 한 존재합니다. 엔티티 빈이 비활성 상태로 남아 있는 기간에 상관 없이 컨테이너는 영구적인 저장소에서 제거하지 않습니다.

엔티티 빈을 제거하는 유일한 방법은 명시적으로 제거하는 것입니다. 엔티티 빈은 데이터베이스에서 원본 데이터를 제거하는 자신의 `remove()` 메소드를 호출하여 제거됩니다. 또는 기존의 엔터프라이즈 애플리케이션은 데이터베이스에서 데이터를 제거할 수 있습니다.

지속성과 엔티티 빈

모든 엔티티 엔터프라이즈 빈은 지속적인 것이며 그 상태가 세션과 클라이언트 사이에 저장됩니다. 빈 제공자로서 엔티티 빈의 지속성이 구현되는 방법을 선택할 수 있습니다.

빈 자체가 지속성을 유지하게 만들도록 엔티티 빈 클래스에서 직접 빈의 지속성을 구현할 수 있습니다. 이것을 *Bean 관리방식(Bean Managed Persistence, BMP)*이라고 합니다.

또는 엔티티 빈의 지속성을 EJB 컨테이너가 처리하도록 위임할 수 있습니다. 이것을 *컨테이너 관리방식(Container Managed Persistence, CMP)*이라고 합니다.

Bean 관리방식

Bean 관리방식(Bean Managed Persistence, BMP)을 사용하는 엔티티 빈은 데이터베이스에 액세스하고 업데이트하는 코드를 포함합니다. 즉, 빈 제공자로서 엔티티 빈 또는 관련 클래스에서 데이터베이스 액세스 호출을 직접 작성합니다. 일반적으로 JDBC를 사용하여 이러한 호출을 작성합니다.

데이터베이스 액세스 호출은 엔티티 빈의 비즈니스 메소드 또는 엔티티 빈 인터페이스 메소드 중 하나에서 나타낼 수 있습니다. (곧 엔티티 빈 인터페이스에 대하여 더 자세히 알아볼 것입니다.)

일반적으로 BMP를 사용하는 빈은 추가적으로 데이터 액세스 코드를 작성해야 하기 때문에 작성하기가 더 어렵습니다. 또한 빈의 메소드에 데이터 액세스 코드를 포함하도록 선택할 수도 있으므로 엔티티 빈을 다른 데이터베이스나 다른 스키마에 사용하기는 더 어렵습니다.

컨테이너 관리방식

컨테이너 관리 방식(Container Managed Persistence, CMP)을 사용하는 엔티티 빈의 경우, 데이터베이스에 액세스하고 업데이트하는 코드를 작성할 필요가 없습니다. 그 대신 빈은 컨테이너에 의존하여 데이터베이스에 액세스하고 업데이트합니다.

CMP는 BMP에 비해 많은 장점을 갖습니다.

- CMP를 사용하는 빈은 코딩하기가 더 쉽습니다.
- 지속성 세부 사항은 엔티티 빈을 수정하거나 다시 컴파일하지 않고 변경될 수 있습니다. 그 대신 배포자나 애플리케이션 어셈블러는 Deployment Descriptor를 수정할 수 있습니다.
- 오류의 가능성이 적어지므로 코드의 복잡성도 줄어듭니다.
- 빈 제공자로서 개발자는 기본으로 사용하는 시스템 문제가 아닌 빈의 비즈니스 로직에 초점을 맞출 수 있습니다.

하지만 CMP에는 몇 가지 제한 사항이 있습니다. 예를 들어, 컨테이너는 `ejbLoad()` 메소드를 호출하기 전에 엔티티 객체의 전체 상태를 빈 인스턴스의 필드로 로드할 수도 있습니다. 이 때 빈이 많은 필드를 가진 경우라면 성능 문제를 야기시킬 수 있습니다.

엔티티 빈의 기본 키

엔티티 빈 인스턴스마다 기본 키를 가져야 합니다. 기본 키는 인스턴스를 고유하게 식별하는 값 또는 값의 조합입니다. 예를 들어, 직원 레코드를 포함하는 데이터베이스 테이블은 직원의 사회 보장 번호를 기본 키로 사용할 수 있습니다. 이 직원 테이블을 모델링하는 엔티티 빈 또한 사회 보장 번호를 기본 키로 사용합니다.

엔터프라이즈 빈의 경우, 기본 키는 문자열(String) 타입 또는 정수(Integer) 타입이나 고유한 데이터를 포함하는 Java 클래스에 의해 표현됩니다. 이러한 기본 키 클래스는 해당 클래스가 RMI_IIOP에서 적합한 값 타입인 클래스일 수 있습니다. 이는 클래스가 `java.io.Serializable` 인터페이스를 확장해야 하고, 모든 Java 클래스가 상속하는 `Object.hashCode()` 메소드 및 `Object.equals(Other other)` 메소드를 구현해야 함을 의미합니다.

기본 키 클래스는 특정 엔티티 빈 클래스에 고유할 수 있습니다. 즉, 각 엔티티 빈은 자신의 고유한 기본 키 클래스를 정의할 수 있습니다. 또는 여러 엔티티 빈은 동일한 기본 키 클래스를 공유할 수 있습니다.

엔티티 빈 클래스 작성

다음과 같은 방법으로 엔티티 빈 클래스를 만듭니다.

- `javax.ejb.EntityBean` 인터페이스를 구현하는 클래스를 만듭니다.
- 하나 이상의 `ejbCreate()` 메소드를 구현합니다. 이미 빈의 홈 인터페이스를 만들었다면 홈 인터페이스에 있는 각각의 `create()` 메소드와 시그너처가 동일한 `ejbCreate()` 메소드가 빈에 있어야 합니다.
- 빈에 지정할 비즈니스 메소드를 정의하고 구현합니다. 이미 빈의 원격 인터페이스를 만들었다면 메소드는 원격 인터페이스에서와 똑같이 정의되어야 합니다.
- BMP를 사용하는 엔티티 빈의 경우 finder 메소드를 구현합니다.

JBuilder의 Enterprise JavaBean 마법사에서 이러한 작업을 시작할 수 있습니다. 이 마법사는 `EntityBean` 인터페이스를 확장하고 `EntityBean` 메소드의 비어 있는 구현을 작성하는 클래스를 만듭니다. 빈에서 필요한 경우 구현을 채워야 합니다. 다음 단원에서는 이러한 메소드와 사용되는 방법을 설명합니다.

기존 데이터베이스 테이블을 사용하여 엔티티 빈을 만들려면 JBuilder의 EJB Entity Modeler를 사용하십시오. 자세한 내용은 11- 1페이지의 "EJB Entity Bean Modeler를 사용하여 엔티티 빈 생성"을 참조하십시오.

EntityBean 인터페이스 구현

`EntityBean` 인터페이스는 모든 엔티티 빈이 구현해야 하는 메소드를 정의합니다. 이 인터페이스는 `EnterpriseBean` 인터페이스를 확장합니다.

```
public void EntityBean extends EnterpriseBean {
    public void setEntityContext(EntityContext ctx) throws EJBException,
      RemoteException;
    public void unsetEntitycontext() throws EJBException, RemoteException;
    void ejbRemove() throws RemoveException, EJBException, RemoteException;
    void ejbActivate() throws EJBException, RemoteException;
    void ejbPassivate() throws EJBException, RemoteException;
    void ejbLoad() throws EJBException, RemoteException;
    public void ejbStore() throws EJBException, RemoteException;
}
```

EntityBean 인터페이스의 메소드는 엔티티 빈의 수명 주기와 밀접하게 관련되어 있습니다. 다음 표에서 그 용도를 설명합니다.

메소드	설명
setEntityContext()	엔티티 컨텍스트를 설정합니다. 컨테이너는 이 메소드를 사용하여 EntityContext 인터페이스에 대한 참조를 빈 인스턴스에 전달합니다. EntityContext 인터페이스는 엔티티 빈에 대한 런타임 컨텍스트의 속성에 액세스하는 메소드를 제공합니다. 이 컨텍스트를 사용하는 엔티티 빈 인스턴스는 인스턴스 변수에 저장해야 합니다.
unsetEntityContext()	setEntityContext() 메소드 호출 동안 할당되었던 리소스를 해제합니다. 컨테이너는 엔티티 빈의 현재 인스턴스 수명이 끝나기 전에 이 메소드를 호출합니다.
ejbRemove()	이러한 특정 엔티티 빈과 관련된 하나 이상의 데이터베이스 항목을 제거합니다. 클라이언트가 remove() 메소드를 호출할 때 컨테이너는 이 메소드를 호출합니다.
ejbActivate	활성화된 엔티티 빈을 공지합니다. 컨테이너는 사용 가능한 인스턴스 풀(pool)에서 선택되고 특정 엔티티 객체 ID에 지정된 인스턴스에 이 메소드를 호출합니다. 빈 인스턴스가 활성화될 때 필요한 추가 리소스를 얻을 수 있는 기회를 가집니다.
ejbPassivate()	곧 비활성화될 엔티티를 공지합니다. 즉, 인스턴스와 엔티티 객체 ID의 연결이 곧 끊어지고 인스턴스는 사용 가능한 인스턴스의 풀(pool)로 돌아간다는 것을 공지합니다. 이 때 인스턴스는 풀에 있는 동안 유지하지 않으려는 ejbActivate() 메소드에 할당된 모든 리소스를 해제할 수 있습니다.
ejbLoad()	엔티티 객체가 데이터베이스로부터 나타내는 데이터를 새로 고칩니다. 컨테이너는 엔티티 빈 인스턴스에 이 메소드를 호출하므로 인스턴스는 인스턴스의 변수에서 캐시된 엔티티 상태를 데이터베이스의 엔티티 상태와 동기화합니다.
ejbStore()	엔티티 객체가 데이터베이스로부터 나타내는 데이터를 저장합니다. 컨테이너는 엔티티 빈 인스턴스에 이 메소드를 호출하므로 인스턴스는 데이터베이스 상태를 인스턴스 변수에서 캐시된 엔티티 상태와 동기화합니다.

엔티티 빈 메소드의 선언 및 구현

엔티티 빈은 다음과 같은 세 가지 메소드 타입을 가질 수 있습니다.

- Create 메소드
- Finder 메소드

- 비즈니스 메소드

Create 메소드 생성

Enterprise JavaBean 마법사를 사용하여 엔터프라이즈 빈을 시작하는 경우, 마법사가 매개변수를 가지지 않은 빈 클래스에 ejbCreate() 메소드와 ejbPostCreate() 메소드를 추가하는 것을 알 수 있습니다. 빈에서 필요한 경우 추가 create 메소드를 작성할 수 있습니다.

엔티티 빈은 create 메소드를 가질 필요가 없다는 것을 명심하십시오. 엔티티 빈의 create 메소드를 호출하면 데이터베이스에 새로운 데이터를 삽입합니다. 엔티티 객체의 새 인스턴스가 DBMS 업데이트 또는 레거시 애플리케이션을 통해서만 데이터베이스에 추가되어야 하는 경우 create 메소드 없는 엔티티 빈을 가질 수 있습니다.

ejbCreate() 메소드

매개변수를 포함하는 ejbCreate() 메소드를 추가하려고 선택한 경우 다음과 같은 규칙을 기억하십시오.

- 각 ejbCreate()는 public으로 선언되어야 합니다.
- 컨테이너에 의해 관리되는 엔티티 빈의 경우 ejbCreate() 메소드는 null을 반환해야 합니다.

  컨테이너는 컨테이너에 의해 관리되는 엔티티 빈을 만드는 것에 대한 전체적인 책임을 가집니다.
- 빈 관리방식 엔티티 빈의 경우 ejbCreate() 메소드는 새 엔티티 객체에 대한 기본 키 클래스의 인스턴스를 반환해야 합니다.

  컨테이너는 이 기본 키를 사용하여 실제 엔티티 참조를 만듭니다.
- ejbCreate() 메소드의 매개변수는 빈의 원격 인터페이스에 있는 해당 create() 메소드와 동일한 개수와 타입을 가져야 합니다.
- 각 ejbCreate() 메소드는 매개변수의 수에 있어서 ejbCreate() 메소드와 일치하는 해당 ejbPostCreate() 메소드를 가져야 합니다.

ejbCreate() 메소드의 시그너처는 빈이 CMP 또는 BMP를 사용하는지 여부에 상관 없이 동일합니다. 다음은 엔티티 빈의 모든 ejbCreate() 메소드의 시그너처입니다.

```
public <PrimaryKeyClass> ejbCreate( <zero or more parameters> )
// implementation
}
```

클라이언트가 create() 메소드를 호출하면 컨테이너는 응답하여 ejbCreate() 메소드를 실행하고 엔티티 객체를 나타내는 레코드를 데이터베이스에 삽입합니다. ejbCreate() 메소드는 대개 일부 엔티티 상태를 초기화합니다. 따라서 이 메소드는 경우에 따라 하나 이상의 매개변수를 가지고 그 구현은 엔티티 상태를 매개변수 값으로 설정하는 코드를 포함합니다. 예를 들어, 이 장에서 나중에 설명할 은행 예제는 당좌 예금 계좌 엔티티 빈을 갖는데, 여기서 엔티티 빈의 ejbCreate() 메소드는 문자열과 float 값

을 갖는 두 개의 매개변수를 취합니다. 메소드는 다음과 같이 계좌 이름을 문자열 값으로 초기화하고 계좌 잔액을 float 값으로 초기화합니다.

```
public AccountPK ejbCreate(String name, float balance) {
    this.name = name;
    this.balance = balance;
  return null;
}
```

ejbPostCreate() method

ejbCreate() 메소드가 실행을 마치면, 컨테이너는 일치하는 ejbPostCreate() 메소드를 호출하여 인스턴스가 초기화를 완료하도록 합니다. 다음과 같은 ejbPostCreate() 메소드는 매개변수에 있어서 ejbCreate() 메소드와 일치하지만 void를 반환합니다.

```
public void ejbPostCreate( <zero or more parameters> )
// implementation
}
```

ejbPostCreate() 메소드를 정의할 때 다음 규칙을 따르십시오.

- public으로 선언되어야 합니다.
- 메소드는 final 또는 static로 선언될 수 없습니다.
- 메소드의 반환 타입은 void여야 합니다.
- 매개변수 목록은 해당 ejbCreate() 메소드의 목록과 일치해야 합니다.

ejbPostCreate() 메소드를 사용하여 클라이언트에게 사용 가능하기 전에 빈에서 필요로 하는 특별한 처리를 수행합니다. 빈이 특별한 처리를 수행해야 할 필요가 없는 경우에는 메소드 몸체를 비워 두지만, BMP를 사용하는 엔티티 빈의 경우 각 ejbCreate() 메소드마다 하나의 ejbPostCreate() 메소드를 포함해야 합니다.

finder 메소드 생성

모든 엔티티 빈은 하나 이상의 finder 메소드를 가져야 합니다. Finder 메소드는 엔티티 빈을 찾는 클라이언트에 의해 사용됩니다. 빈 관리방식의 각 엔티티 빈은 빈의 홈 인터페이스에 해당 findByPrimaryKey() 메소드를 가지는 ejbFindByPrimaryKey() 메소드를 가져야 합니다. 다음은 ejbFindByPrimaryKey() 메소드의 시그너처입니다.

```
public <PrimaryKeyClass> ejbFindByPrimaryKey(<PrimaryKeyClass primaryKey>) {
// implementation
}
```

빈에 대하여 추가 finder 메소드를 정의할 수 있습니다. 예를 들어, ejbFindByLastName() 메소드를 가졌을 수도 있습니다. 각 finder 메소드는 다음 규칙을 따라야 합니다.

- public으로 선언되어야 합니다.
- 메소드 이름은 접두사 **ejbFind**로 시작되어야 합니다.

- 메소드는 static 또는 final로 선언될 수 없습니다.
- 메소드는 기본 키 또는 기본 키의 컬렉션(모음) 아니면 기본 키의 열거(Enumeration)를 반환해야 합니다.
- 메소드의 매개변수와 반환 타입은 유효한 Java RMI 타입이어야 합니다.

BMP를 사용하는 엔티티 빈의 경우, 빈 클래스에서 선언된 각 finder 메소드는 동일한 매개변수를 갖는 빈의 홈 인터페이스에서 해당 finder 메소드를 가져야 하지만 엔티티 빈의 원격 인터페이스를 반환합니다. 클라이언트는 홈 인터페이스의 finder 메소드를 호출하여 원하는 엔티티 빈을 찾은 다음 컨테이너는 빈 클래스에서 해당 finder 메소드를 호출합니다. 18- 4 페이지의 "엔티티 빈의 finder 메소드"를 참조하십시오.

비즈니스 메소드 작성

엔터프라이즈 빈 클래스 내에서 빈에 필요한 비즈니스 메소드의 전체 구현을 작성합니다. 이러한 메소드를 클라이언트가 사용할 수 있게 하려면 똑같은 시그너처를 사용하여 빈의 원격 인터페이스에서 이 메소드를 선언해야 합니다.

JBuilder 마법사를 사용하여 엔티티 빈 생성

JBuilder의 Enterprise JavaBeans 마법사를 사용하여 엔티티 빈의 생성을 빨리 시작할 수 있습니다. 기존 데이터베이스 테이블을 사용하여 엔터프라이즈 빈을 만들려는 경우 JBuilder의 EJB Entity Modeler를 사용하여 시작합니다. 기존 엔티티 빈을 가지고 있지만 그것의 홈 인터페이스 및 원격 인터페이스를 가지지 않은 경우, EJB 인터페이스 마법사를 사용하여 만듭니다. 그런 다음 코드 에디터와 Bean 디자이너를 사용하여 Jbuilder에서 코드를 수정할 수 있습니다. JBuilder의 Deployment Descriptor 에디터를 사용하여 배치 디스크립터를 편집합니다. 새로운 엔터프라이즈 빈을 테스트하기 위해 EJB Test Client Application 마법사를 사용하여 테스트 클라이언트 애플리케이션을 만듭니다. 마지막으로, EJB Deployment 마법사를 사용하여 엔터프라이즈 빈을 배포하는 프로세스를 단순화합니다.

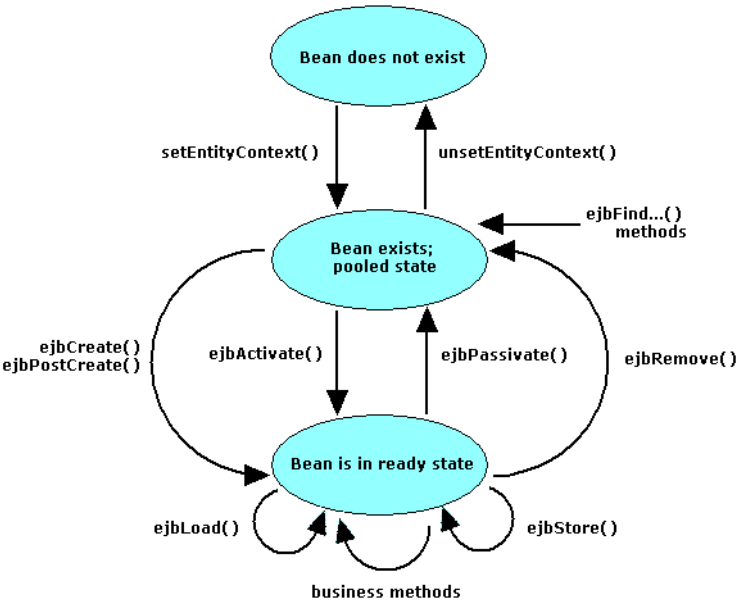
JBuilder의 EJB 개발 도구를 사용하여 엔티티 빈을 만드는 것에 대한 자세한 내용은 Chapter 3, "Developing enterprise beans" Chapter 14 "Deployment Descriptor 에디터 사용" 및 Chapter 13 "엔터프라이즈 빈 배포"를 참조하십시오

엔티티 빈의 수명

엔티티 엔터프라이즈 빈의 수명 주기에는 다음과 같은 세 가지 다른 상태가 있습니다.

- Nonexistent
- Pooled
- Ready

다음 다이어그램은 엔티티 빈 인스턴스의 수명 주기에 대해 설명한 것입니다.



존재하지 않는 상태

처음에 엔티티 빈 인스턴스는 존재하지 않습니다. EJB 컨테이너는 엔티티 빈의 인스턴스를 만든 다음 엔티티 빈에 `setEntityContext()` 메소드를 호출하여 해당 컨텍스트에 대한 참조, 즉 `EntityContext` 인터페이스에 대한 참조를 인스턴스에 전달합니다. `EntityContext` 인터페이스는 컨테이너 제공 서비스에 인스턴스 액세스를 제공하고 클라이언트에 대한 정보를 얻도록 합니다. 엔티티 빈은 이제 공용 상태에 있습니다.

풀링 상태

각 타입의 엔티티 빈은 자신의 풀(pool)을 가집니다. 풀에 있는 인스턴스는 데이터와 연결되지 않습니다. 인스턴스 변수 모두 설정되지 않았기 때문에 ID를 가지지 않고 모두 동일합니다. 컨테이너는 그러한 엔티티 빈을 요청하는 클라이언트에 어떠한 인스턴스도 할당할 수 있습니다.

클라이언트 애플리케이션이 엔티티 빈의 finder 메소드 중 하나를 호출할 때, 컨테이너는 풀에 있는 임의의 인스턴스에 해당 `ejbFind()` 메소드를 실행합니다. 인스턴스는 finder 메소드가 실행되는 동안 풀링 상태를 유지합니다.

컨테이너가 클라이언트의 요청을 엔티티 객체에 서비스하는 인스턴스를 선택할 때, 인스턴스는 풀링 상태에서 준비 상태로 바뀝니다. 엔티티 인스턴스가 풀링 상태에서 준비 상태로 바뀌는 방법은 다음 두 가지입니다.

- `ejbCreate()` 메소드와 `ejbPostCreate()` 메소드를 통한 방법.
- `ejbActivate()` 메소드를 통한 방법.

컨테이너는 빈의 홈 인터페이스에 대한 클라이언트의 `create()` 요청을 처리할 인스턴스를 선택합니다. `create()` 호출에 응답하여 컨테이너는 엔티티 객체를 만들고, 인스턴스가 엔티티 객체에 할당될 때 `ejbCreate()` 메소드와 `ejbPostCreate()` 메소드를 호출합니다.

컨테이너는 `ejbActivate()` 메소드를 호출하여 인스턴스를 활성화하므로 기존 엔티티 객체에 대한 호출에 응답할 수 있습니다. 클라이언트 호출을 처리할 수 있는 준비 상태에 있는 적당한 인스턴스가 없을 때 일반적으로 컨테이너는 `ejbActivate()` 메소드를 호출합니다.

준비 상태

인스턴스가 준비 상태에 있을 때 특정 기본 키에 연결됩니다. 클라이언트는 엔티티 빈에 애플리케이션 특정 메소드를 호출할 수 있습니다. 컨테이너는 `ejbLoad()` 메소드와 `ejbStore()` 메소드를 호출하여 빈에게 데이터 로드와 저장을 알립니다. 또한 이러한 메소드를 통해 빈 인스턴스가 자신의 상태와 원본으로 사용하는 데이터 엔티티의 상태를 동기화할 수 있습니다.

풀링 상태로 돌아가기

엔티티 빈 인스턴스가 풀링 상태로 돌아가면 인스턴스는 엔티티에 의해 나타내는 데이터에서 분리됩니다. 이제 컨테이너는 동일한 엔티티 빈 홈 내에서 엔티티 객체의 인스턴스를 할당할 수 있습니다. 다음은 엔티티 빈 인스턴스가 준비 상태에서 풀링 상태로 다시 돌아가는 두 가지 방법입니다.

- 컨테이너는`ejbPassivate()` 메소드를 호출하여 원본으로 사용하는 엔티티 객체를 제거하지 않고 기본 키에서 인스턴스를 분리합니다.
- 컨테이너는 `ejbRemove()` 메소드를 호출하여 엔티티 객체를 제거합니다. 클라이언트 애플리케이션이 빈의 홈 `remove()` 메소드 또는 원격 `remove()` 메소드를 호출할 때 컨테이너는 `ejbRemove()` 메소드를 호출합니다.

풀에서 분리된 인스턴스를 제거하려면 컨테이너는 인스턴스의 `unsetEntityContext()` 메소드를 호출합니다.

은행 엔티티 빈 예제

은행 예제는 엔티티 빈 사용 방법을 보여 줍니다. 이 예제에서는 동일한 `Account` 원격 인터페이스를 두 가지 구현합니다. 한 구현에서는 BMP를 사용하고 다른 구현에서는 CMP를 사용합니다.

BMP를 사용하는 SavingsAccount 엔티티 빈은 저축 예금 계좌를 모델링합니다. 엔티티 빈 코드를 보면 직접적인 JDBC 호출을 포함한다는 것을 알 수 있습니다.

CMP를 사용하는 CheckingAccount 엔티티 빈은 당좌 예금 계좌를 모델링합니다. 엔티티 빈은 지속성을 구현하는 데 빈 개발자가 아닌 컨테이너에 의존합니다.

Teller라는 세 번째 엔터프라이즈 빈은 한 계좌에서 다른 계좌로 자금을 이체합니다. 이 엔터프라이즈 빈은 단일 컨테이너 방식의 트랜잭션 내에서 여러 엔티티 빈에 대한 호출이 그룹화될 수 있는 방법을 보여 주는 상태 없는 세션 빈입니다. 차변 기입에 실패하면 이체 작업에서 차변 기입 전에 대변 기입이 발생하더라도 컨테이너는 트랜잭션을 롤백하고 대변 기입과 차변 기입 모두 발생하지 않습니다.

은행 예제의 전체 코드는 /Borland/AppServer/examples/ejb/bank 디렉토리에서 찾을 수 있습니다.

엔티티 빈 홈 인터페이스

한 엔티티 빈이 CMP를 사용하고 다른 엔티티 빈이 BMP를 사용하더라도, 여러 엔티티 빈이 동일한 홈 인터페이스 및 원격 인터페이스를 공유할 수 있습니다. SavingsAccount 엔티티 빈과 CheckingAccount 엔티티 빈은 모두 동일한 홈 인터페이스인 AccountHome을 사용합니다. 이 엔티티 빈들은 또한 동일한 Account 원격 인터페이스를 사용합니다.

엔티티 빈의 홈 인터페이스는 세션 빈의 홈 인터페이스와 매우 유사합니다. 이 빈들은 동일한 javax.ejb.EJBHome 인터페이스를 확장합니다. 엔티티 빈의 홈 인터페이스는 최소한 하나의 finder 메소드를 포함해야 합니다. create() 메소드는 옵션입니다.

다음은 AccountHome 인터페이스의 코드입니다.

```
public interface AccountHome extends javax.ejb.EJBHome {
  Account create(String name, float balance)
    throws java.rmi.RemoteException, javax.ejb.CreateException;
  Account findByPrimaryKey(AccountPK primaryKey)
    throws java.rmi.RemoteException, javax.ejb.FinderException;
  java.util.Enumeration findAccountsLargerThan(float balance)
    throws java.rmi.RemoteException, javax.ejb.FinderException:
}
```

AccountHome 홈 인터페이스는 세 가지 메소드를 구현합니다. 엔티티 빈에 대한 create() 메소드는 필요하지 않지만 은행 예제에서는 하나의 create() 메소드를 구현합니다. create() 메소드는 새 엔티티 빈 객체를 원본으로 사용하는 데이터베이스에 삽입합니다. 새 엔티티 객체의 생성을 DBMS나 다른 애플리케이션에게 맡길 수 있는데, 이 경우에는 create() 메소드를 정의하지 않습니다.

create() 메소드에는 두 가지 매개변수, 계좌 이름 문자열과 잔액이 필요합니다. 엔티티 빈 클래스에 있는 이 메소드의 구현은 이러한 두 가지 매개변수 값을 사용하여 새 객체를 만들 때 엔티티 객체 상태(계좌 이름과 첫 잔

액)를 초기화합니다. create() 메소드의 throws 절에 java.rmi.RemoteException와 java.ejb.CreateException을 포함해야 합니다. 애플리케이션 특정 예외를 추가적으로 포함시킬 수 있습니다.

엔티티 빈은 findByPrimaryKey() 메소드를 가져야 하므로, AccountHome 인터페이스는 이 메소드를 선언합니다. 이 메소드는 하나의 매개변수인 AccountPK 기본 키 클래스를 취하고 Account 원격 인터페이스에 대한 참조를 반환합니다. 이 메소드는 하나의 특정한 계좌 엔티티를 찾아서 그것에 대한 참조를 반환합니다.

또한 홈 인터페이스는 필수는 아니지만 두 번째 finder 메소드인 findAccountsLargerThan()을 선언합니다. 이 메소드는 잔액이 일정액 이상인 모든 계좌를 포함하는 Java Enumeration을 반환합니다.

엔티티 빈 원격 인터페이스

빈들이 다른 지속성 관리 전략을 사용할 때에도 둘 이상의 엔티티 빈은 동일한 원격 인터페이스를 사용할 수 있습니다. 은행 예제의 두 엔티티 빈 모두 동일한 Account 원격 인터페이스를 사용합니다.

원격 인터페이스는 javax.ejb.EJBObject 인터페이스를 확장하고 클라이언트에 사용 가능한 비즈니스 메소드를 노출합니다. 코드는 다음과 같습니다.

```
public interface Account extends javax.ejb.EJBObject {
  public float getBalance() throws java.rmi.RemoteException;
  public void credit(float amount) throws java.rmi.RemoteException;
  public void debit(float amount) throws java.rmi.RemoteException;
}
```

컨테이너 관리방식을 사용하는 엔티티 빈

은행 예제는 CMP를 사용하는 기본적인 사항에 대해 설명하는 CheckingAccount 엔티티 빈을 구현합니다. 여러 가지 면에서 이 구현은 세션 빈 구현과 유사합니다. 하지만 CMP를 사용하는 엔티티 빈의 구현에서 주의해야 할 몇 가지 중요한 사항이 있습니다.

- 엔티티 빈에는 finder 메소드에 대한 구현이 없습니다. EJB 컨테이너는 CMP를 사용하는 엔티티 빈에 finder 메소드 구현을 제공합니다 . 빈의 클래스에서 finder 메소드에 대한 구현을 제공하는 대신, Deployment Descriptor에는 컨테이터가 이러한 finder 메소드를 구현할 수 있는 정보를 포함합니다.
- 엔티티 빈은 빈에 대한 컨테이너에 의해 관리되는 모든 필드를 public으로 선언합니다. CheckingAccount 빈은 name과 balance를 public 필드로 선언합니다.
- 엔티티 빈 클래스는 ejbActivate(), ejbPassivate(), ejbLoad(), ejbStore(), ejbRemove(), setEntityContext(), unsetEntityContext() 등과 같이 EntityBean 인터페이스에서 선언된 여러 가지 메소드를 구현합니다. 하

지만 엔티티 빈은 적당한 곳에 애플리케이션 특정 코드를 추가할 수 있지만, 이러한 메소드에만 뼈대 구현을 제공해야 합니다. CheckingAccount 빈은 setEntityContext()에 의해 반환되는 컨텍스트를 저장하고 unsetEntityContext()에서 참조를 해제합니다. 그렇지 않은 경우 EntityBean 인터페이스 메소드에 코드를 추가하지 않습니다.

- CheckingAccount 엔터프라이즈 빈을 통해 호출자가 새로운 당좌 예금 계좌를 만들 수 있기 때문에 이 엔터프라이즈 빈에는 ejbCreate() 메소드의 구현을 포함합니다. 또한 구현에서 인스턴스의 두 변수인 name과 balance를 매개변수 값으로 초기화합니다. 컨테이너가 클라이언트로 돌아가기 위해 CMP를 사용하여 적당한 참조를 만들기 때문에 ejbCreate()는 null 값을 반환합니다.
- ejbPostCreate() 메소드는 필요하면 추가적인 초기화를 수행할 수 있지만, CheckingAccount는 이 메소드의 최소 구현을 제공합니다. CMP를 사용하는 빈의 경우, 공지 콜백으로 작용하기 때문에 ejbPostCreate()의 최소 구현만 필요합니다.

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

public class CheckingAccount implements EntityBean {
  private javax.ejb.EntityContext _context;
  public String name;
  public float balance;


  public float getBalance() {
    return balance;
  }

  public void debit(float amount) {
    if(amount > balance) {
      // mark the current transaction for rollback ...
     _context.setRollbackOnly();
    }
else {
      balance = balance - amount;
    }
  }

  public void credit(float amount) {
    balance = balance + amount;
  }

public AccountPK ejbCreate(String name, float balance) {
    this.name = name;
    this.balance = balance;
  return null;
  }

  public void ejbCreate(String name, float balance) {}
  public void ejbRemove() {}
```

```
  public void setEntityContext(EntityContext context) {
    _context = context;
  }

  public void unsetEntityContext() {
    context = null;
  }

  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void ejbLoad() {}
  public void ejbStore() {}
  public String toString() {
    return"CheckingAccount[name=" + name + ",balance=" + balance +"]";
  }
}
```

빈 관리방식을 사용하는 엔티티 빈

은행 예제는 또한 BMP를 사용하는 엔티티 빈인 SavingsAccount 빈을 구현합니다. SavingsAccount 빈은 CheckingAccount 빈과는 다른 계좌 테이블에 액세스합니다. 이 두 가지 엔티티 빈이 다른 지속성 관리 방법을 사용하지만 두 가지 모두 동일한 홈 인터페이스 및 원격 인테페이스를 사용합니다. 하지만 두 구현에는 차이가 있습니다.

BMP를 사용하는 엔티티 빈 구현은 다음을 수행합니다.

- 인스턴스 변수를 public 대신 private로 선언할 수 있습니다.

  빈은 이러한 변수에 액세스하고 데이터베이스 값을 이러한 인스턴스 변수에 로드하고 해당 변경 사항을 데이터베이스에 저장하는 코드를 포함합니다. 이와 같이 빈은 이러한 변수에 대한 액세스를 적절하게 제한할 수 있습니다. 이는 컨테이너 관리방식의 모든 변수를 public으로 선언해야 컨테이너가 이러한 변수에 액세스할 수 있는 CMP를 사용하는 빈과 다릅니다.

- ejbCreate() 메소드는 기본 키 클래스를 반환합니다.

  SavingsAccount 빈에서 클래스는 AccountPK입니다. 컨테이너는 반환된 기본 키 클래스를 취하고 이를 엔티티 빈 인스턴스에 대한 원격 참조를 생성하는 데 사용합니다.

- CMP를 사용하는 빈처럼, BMP를 사용하는 빈은 ejbCreate() 메소드의 비어 있는 구현보다 더 많이 옵션으로 제공할 수도 있습니다.

  SavingsAccount 빈은 이 메소드에 추가 초기화 코드를 포함할 필요가 없습니다.

- 이 빈은 ejbLoad() 메소드와 ejbStore() 메소드를 구현합니다.

  컨테이너가 지속성을 처리하기 때문에 CMP를 사용하는 빈은 보통 이러한 메소드의 비어 있는 구현만 제공합니다. BMP를 사용하는 엔티티 빈은 자신의 코드를 제공하여 데이터베이스 값을 ejbLoad() 메소드에 있

는 자신의 인스턴스 변수로 읽어 들이고, ejbStore() 메소드에서 변경된 값을 데이터베이스에 기록합니다.

- 이 엔티티 빈은 모든 finder 메소드를 구현합니다.

  SavingsAccount 엔티티 빈은 필수인 ejbFindByPrimaryKey() 메소드와 옵션인 ejbFindAccountsLargerThan() 메소드인 두 가지 finder 메소드를 구현합니다.

- BMP를 사용하는 엔티티 빈은 ejbRemove() 메소드를 구현해야 합니다.

  빈이 원본으로 사용하는 데이터베이스 엔티티 객체를 관리하고 있기 때문에, 빈은 데이터베이스로부터 엔티티 객체를 제거할 수 있도록 이 메소드를 구현해야 합니다. 컨테이너가 데이터베이스 관리를 맡고 있기 때문에 CMP를 사용하는 빈은 이 메소드의 구현을 생략합니다.

- 원본으로 사용하는 데이터베이스 객체에 액세스하는 각 메소드는 올바른 데이터베이스 액세스 코드를 포함합니다.

  이러한 메소드에는 ejbCreate(), ejbRemove(), ejbLoad(), ejbStore, ejbFindByPrimaryKey() 및 다른 모든 finder 메소드와 비즈니스 메소드가 있습니다. 각 메소드는 데이터베이스에 연결하는 코드를 포함하고, 그 다음에 빌드할 코드가 나온 다음 메소드에 의해 처리되는 기능을 수행하는 SQL 문을 실행합니다. SQL 문이 완전하면 메소드는 반환하기 전에 문장과 데이터베이스 연결을 닫습니다.

다음 코드 예제는 SavingsAccount 구현 클래스에서 흥미로운 코드 부분을 보여 줍니다. 이 예제에서는 ejbActivate(), ejbPassivate() 등과 같은 EntityBean 인터페이스 메소드의 단지 비어 있는 구현인 코드를 제거합니다.

먼저 데이터베이스 엔티티 객체에 액세스하는 ejbLoad() 메소드를 주목하여 BMP를 사용하는 빈이 데이터베이스 액세스를 구현하는 방법을 봅니다. SavingsAccount 클래스에서 구현된 모든 메소드가 ejbLoad() 메소드가 사용하는 것과 동일한 방법을 따라야 합니다. ejbLoad() 메소드는 데이터베이스에 연결을 만들면서 시작합니다. 이 메소드는 JDBC 연결 풀로부터 데이터베이스에 JDBC 연결을 얻기 위해 DataSource를 사용하는 내부 getConnection() 메소드를 호출합니다. 일단 연결이 이루어지면 ejbLoad()는 PreparedStatement 객체를 만들고 해당 SQL 데이터베이스 액세스 문을 만듭니다. ejbLoad() 메소드가 엔티티 객체 값을 엔티티 빈의 인스턴스 변수로 읽어 들이기 때문에, 이 메소드는 패턴과 일치하는 이름을 가진 저축 예금 계좌의 잔액을 선택하는 쿼리에 대한 SQL SELECT 문을 만듭니다. 그런 다음 메소드는 쿼리를 실행합니다. 쿼리가 결과를 반환하면 메소드는 잔액을 추출합니다. ejbLoad() 메소드는 PreparedStatement 객체를 닫은 다음 데이터베이스 연결을 닫음으로써 완료됩니다. ejbLoad() 메소드는 연결을 실제로 닫지는 않습니다. 그 대신 메소드는 단순히 연결 풀에 대한 연결을 반환합니다.

```
import.java.sql.*;
import javax.ejb.*;
import java.util.*;
import java.rmi.RemoteException;
```

```
public class SavingsAccount implements EntityBean {
  private entitycontext _constext;
  private String _name;
  private float _balance;

  public float getBalance() {
    return _balance;
  }

  public void debit(float amount) {
    if(amount > balance) {
      // mark the current transaction for rollback...
     _context.setRollbackOnly();
    } else {
        _balance = _balance - amount;
    }
  }
  public void credit(float amount) {
    _balance = _balance + amount;
  }

  // setEntitycontext(), unsetEntityContext(), ejbActivate(), ejbPassivate(),
  // ejbPostCreate() skeleton implementations are not shown here
  ...

  public AccountPK ejbCreate(String name, float balance)
      throws RemoteException, CreateException {
    _name = name;
    _balance = balance;
   try  {
      Connection connection = getConnection();
      PreparedStatement statement = connection.prepareStatement
        ("INSERT INTO Savings_Accounts (name, balance) VALUES (?,?)*);
      statement.setString(1, _name);
      statement.setFloat(2, _balance);
      if(statement.executeUpdate() != 1) {
        throw new CreateException("Could not create: " + name);
      }
      statement.close();
      connection.close();
      return new AccountPK(name);
    } catch(SQLException e) {
        throw new RemoteException("Could not remove: " + name, 3);
    }
  }
  ...
  public void ejbRemote() throws RemoteException, RemoveException {
   try  {
      Connection connection = getConnection();
      PreparedStatement statement = connection.prepareStatement
        ("DELETE FROM Savings Account WHERE name = ?");
      statement.setString(1, _name);
      if(statement.executeUpdate() != 1) {
        throw new RemoteException("Could not remove:" + _name, e);
      }
      statement.close();
      connection.close();
    } catch(SQLException e) {
        throw new RemoteException("Could not remove:" + _name, e);
    }
  }
  public AccountPK ejbFindByPrimaryKey(AccountPK key) throws RemoteException,
        FinderException {
```

```
    try  {
       Connection connection = getConnection();
       PreparedStatement statement = connection.prepareStatement
         ("SELECT name FROM Savings_Accounts WHERE name = ?");
       statement.setString(1, key.name);
       ResultSet resultSet = statement.executeQuery();
       if(!resultSet.next()) {
         throw new FinderException("Could not find: " + key
       statement.close();
       connection.close();
       return key;
     } catch(SQLException e) {
         throw new RemoteException("Could not: " + key, e);
     }
   }

   public java.util.Enumeration ejbFindAccountsLargerThan(float balance)
       throws RemoteException, FinderException {
    try  {
       Connection connection = getConnection();
       PreparedStatement statement = connection.prepareStatement
         ("SELECT name FROM Savings_Account WHERE balance > ?");
       statement.setFloat(1, balance);
       ResultSet resultSet = statement.executeQuery();
       Vector keys = new Vector();
       while(resultSet.next()) {
         String name = resultSet.getString(1);
         keys.addElement(new AccountPK(name));
       }
       statement.close();
       connection.close();
       return keys.elements();
     } catch(SQLException 3) {
         throw new RemoteException("Could not findAccountsLargerThan: " + balance,
e);
     }
   }

   public void ejbLoad() throws RemoteException {
     // get the name from the primary key
     _name = (AccountPK) _context.getPrimaryKey()).name;
    try  {
       Connection connection = getConnection();
       PreparedStatement statement = connection.prepareStatement
         ("SELECT balance FROM Savings_Account WHERE name = ?");
       statement.setString(1, _name);
       ResultSet resultSet = statement.executeQuery();
       if(!resultSet.next()) {
         throw new RemoteException("Account not found: " + _name);
       }
       _balance = resultSet.getFloat(1);
       statement.close();
       connection.close();
     } catch(SQLException e) {
         throw new RemoteException("Could not load:" + _name, e);
     }
   }

   public void ejbStore() throw RemoteException {
    try  {
       Connection connection = getConnection();
       PreparedStatement statement = connection.prepareStatement
         ("UPDATE Savings_Accounts SET balance = ? WHERE name = ?");
```

```
        statement.setFloat(1, _balance);
        statement.setString(2, _name);
        statement.executeUpdate();
        statement.close();
        connection.close();
      } catch(SQLException e) {
        throw new RemoteException("Could not store:" + _name, e);
      }
    }

    private connection getconnection() throws SQLException {
      Properties properties = _context.getEnvironment();
      String url = properties.getProperty("db.url");
      String username = properties.getProperty("db.username");
      String password = properties.getProperty("db.password");
      if(username != null) {
        return DriverManager.getConnection(url, username, password);
      } else {
        return DriverManager.getConnection(url);
      }
    }

    public String toString() {
      return "SavingsAccount[name=" + _name + ",balance=" + _balance +"]";
    }
  }
```

기본 키 클래스

CheckingAccount와 SavingsAccount는 모두 동일한 필드를 사용하여 특정 계좌 레코드를 고유하게 식별합니다. 이 경우 두 빈 모두 다음과 같이 동일한 기본 키 클래스, AccountPK를 사용하여 둘 중 하나 타입의 계좌에 대한 고유한 식별자를 나타냅니다.

```
public class AccountPK implements java.io.Serializable {
  public String name;
  public AccountPK() {}
  public AccountPK(String name) {
    this.name = name;
  }
}
```

Deployment Descriptor

은행 예제에 대한 Deployment Descriptor는 세 가지 종류의 빈, Teller 세션 빈, CMP를 사용하는 CheckingAccount 엔티티 빈, BMP를 사용하는 SavingsAccount 엔티티 빈을 배포합니다.

Deployment Descriptor에 있는 속성을 사용하여 세션 빈에서처럼 엔티티 빈의 인터페이스, 트랜잭션 속성 등에 대한 정보를 지정합니다. 그러나 엔티티 빈에 고유한 정보도 추가할 수 있습니다.

Bean 관리방식 XML 코드 예제는 BMP를 사용하는 엔티티 빈에 대한 전형적인 Deployment Descriptor 속성 태그를 보여 줍니다. 컨테이너 관리방식 XML 코드 예제는 CMP를 사용하는 엔티티 빈에 대한 전형적인

Deployment Descriptor 태그를 보여 줍니다. 두 가지 타입의 엔티티 빈에 대한 디스크립터 태그를 비교할 때, CMP를 사용하는 엔티티 빈에 대한 Deployment Descriptor가 더 복잡하다는 것을 알 수 있습니다.

빈의 Deployment Descriptor 타입은 <entity>로 설정됩니다. 두 코드 예제 모두에서 <enterprise-beans> 섹션 내에 있는 첫 번째 태그는 그 빈이 엔티티 빈임을 지정합니다.

엔티티 빈 Deployment Descriptor는 다음과 같은 타입의 정보를 지정합니다.

- 관련된 인터페이스(홈과 원격)와 빈 구현 클래스의 이름.

  각 엔터프라이즈 빈은 <home> 태그를 사용하여 해당 홈 인터페이스를 지정하고, <remote> 태그를 사용하여 해당 원격 인터페이스를 지정하고, <ejb-class> 태그를 사용하여 구현 클래스 이름을 지정합니다.

- 엔티티 빈이 등록되고 클라이언트가 빈을 찾는 JNDI 이름.

- 빈의 트랜잭션 속성 및 해당 트랜잭션 분리 레벨.

  이는 대개 Deployment Descriptor의 <assembly-descriptor> 섹션에 나타납니다.

- 엔티티 빈의 기본 키 클래스 이름.

  이 예제에서 기본 키 클래스는 `AccountPK`이고 <prim-key-class> 태그 내에 나타납니다.

- 빈에 의해 사용되는 지속성.

  `CheckingAccount` 빈은 CMP를 사용하므로, Deployment Descriptor는 <persistence-type> 태그를 Container로 설정합니다.

- 빈 클래스가 재진입인지 여부.

  `SavingsAccount` 빈과 `CheckingAccount` 빈 모두 재진입이 아니므로 <reentrant> 태그를 False로 설정합니다.

- 빈이 CMP를 사용하는 경우 컨테이너가 관리하는 필드.
  BMP를 사용하는 빈은 컨테이너 관리방식의 필드를 지정하지 않습니다. 따라서, `SavingsAccount` 빈에 대한 Deployment Descriptor는 컨테이너 관리방식의 필드를 지정하지 않습니다. CMP를 사용하는 엔티티 빈은 해당 필드의 이름 또는 컨테이너가 관리해야 하는 인스턴스 변수를 지정해야 합니다. 여기에 <cmp field> 태그와 <field name> 태그의 조합을 사용합니다. 첫 번째 태그인 <cmp field>는 필드가 컨테이너 관리방식이라는 것을 가리킵니다. 첫 번째 태그 내에서 <fields name> 태그는 필드 자체의 이름을 지정합니다. 예를 들어, `CheckingAccount` 빈 Deployment Descriptor는 잔액 필드가 다음과 같이 컨테이너 관리방식이라는 것을 가리킵니다.

  ```
  <cmp field><field name>balance</field name></cmp field>
  ```

컨테이너에 의해 관리되는 빈의 컨테이너 관리방식의 필드에 대한 정보. 컨테이너는 이 정보를 사용하여 이러한 필드에 대한 finder 메소드를 생성합니다.

BMP를 사용하는 엔티티 빈에 대한 Deployment Descriptor

다음 코드 예제는 BMP를 사용하는 엔티티 빈에 대한 Deployment Descriptor의 주요 부분을 보여 줍니다. 컨테이너가 아닌 빈이 데이터베이스 엔티티 값으로부터 자신의 고유한 페치를 처리하고 이 값으로 업데이트하기 때문에, 디스크립터는 컨테이너가 관리하는 필드를 지정하지 않습니다. 또는 빈의 구현이 제공하기 때문에, Deployment Descriptor에서 해당 finder 메소드를 구현하는 방법을 컨테이너에게 알려 주지 않습니다.

```
<enterprise-beans>
<entity>
  <description>This entity bean is an example of bean-managed persistence</description>
  <ejb-name>savings</ejb-name>
  <home>AccountHome</home>
  <remote>Account</remote>
  <ejb-class>SavingsAccount</ejb-class>
  <persistence-type>Bean</persistence-type>
  <prim-key-class>AccountPK</prim-key-class>
  <reentrant>False</reentrant>
</entity>
  ...
</enterprise-beans>
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>savings</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
```

컨테이너 관리방식을 사용하는 엔티티 빈에 대한 Deployment Descriptor

다음 코드 예제는 CMP를 사용하는 엔티티 빈에 대한 Deployment Descriptor의 주요 부분을 보여 줍니다. 빈은 컨테이너가 데이터베이스 엔티티 값을 로드하고 이러한 값을 업데이트하도록 하기 때문에, 디스크립터는 컨테이너가 관리하는 필드를 지정합니다.

```
<enterprise-beans>
<entity>
  <description>This entity bean is an example of container-managed persistence
    </description>
  <ejb-name>checking</ejb-name>
  ><home>AccountHome</home>
  <remote>Account</remote>
  <ejb-class>chkingAccount</ejb-class>
  <persistence-type>Container</persistence-type>
  >prim-key-class>AccountPK</prim-key-class>
  <reentrant>False</reentrant>
  <cmp-field>
    <field-name>name</field-name>
  <cmp-field>
</entity>
</enterprise-beans>
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>checking</ejb-name>
```

```
        <method-name>*</method-name>
      </method>
      <trans-attribute-transaction>
    </container-transaction>
  </assembly-descriptor>
```

Chapter

18

홈 인터페이스 및 원격 인터페이스 생성

엔터프라이즈 빈 프로바이더는 각 빈에 대해 두 가지 인터페이스, 홈 인터페이스와 원격 인터페이스를 생성해야 합니다. 홈 인터페이스는 클라이언트 애플리케이션이 엔터프라이즈 빈의 인스턴스를 생성하고 찾고 제거하는 데 사용하는 메소드를 정의합니다. 원격 인터페이스는 빈에 구현되는 business 메소드를 정의합니다. 클라이언트는 원격 인터페이스를 통해 이러한 메소드에 액세스합니다.

홈 인터페이스 생성

엔터프라이즈 빈의 홈 인터페이스는 빈의 수명 주기를 제어합니다. 홈 인터페이스에는 엔터프라이즈 빈의 인스턴스를 생성하고 찾으며 제거하는 메소드에 대한 정의가 들어 있습니다.

빈 프로바이더로서 홈 인터페이스를 정의해야 하지만 구현하지는 않습니다. EJB 컨테이너는 홈 인터페이스를 정의하여 빈에 대한 참조를 반환하는 홈 객체를 생성합니다.

홈 인터페이스의 이름은 빈 클래스와 동일한 이름과 함께 **Home** 접미사를 갖는 것이 통례적입니다. 예를 들어, 엔터프라이즈 빈의 이름이 `Account`이면 `Account`의 홈 인터페이스는 `AccountHome`이 됩니다. 기본적으로, JBuilder의 EJB 마법사에서 생성한 홈 인터페이스는 이 규칙을 따르지만 경우에 따라 마법사를 사용하는 동안 인터페이스의 이름을 수정할 수 있습니다.

EJBHome 기본 클래스

각각의 홈 인터페이스는 `javax.ejb.EJBHome` 인터페이스를 확장합니다. 다음은 `EJBHome`의 완전한 정의입니다.

```
package javax.ejb
public interface EJBHome extends java.rmi.Remote {
    void remove(Handle handle) throws java.rmi.RemoteException, RemoveException;
    void remove(Object primaryKey) throws java.rmi.RemoteException, RemoveException;
    EJBMetaData getEJBMetaData() throws RemoteException;
    HomeHandle getHomeHandle() throws RemoteException;
}
```

EJBHome에는 엔터프라이즈 빈 인스턴스를 제거하는 두 가지 remove() 메소드가 있습니다. 첫 번째 remove() 메소드는 핸들로 인스턴스를 식별하고 두 번째 메소드는 기본 키로 인스턴스를 식별합니다.

순차적인 빈 객체 식별자인 핸들은 핸들이 참조하는 엔터프라이즈 빈 객체와 수명이 같습니다. 세션 빈의 경우 핸들은 세션이 있는 경우에만 존재합니다. 엔티티 빈의 경우 핸들은 계속 지속될 수 있고 클라이언트는 일련화된 핸들을 사용하여 핸들이 식별하는 엔티티 객체에 대한 참조를 재설정할 수 있습니다.

클라이언트가 두 번째 remove() 메소드를 사용하면 기본 키를 사용하여 엔티티 빈 인스턴스를 제거합니다.

getEJBMetaData()는 엔터프라이즈 빈 객체의 EJBMetaData 인터페이스를 반환합니다. 이 인터페이스를 통해 클라이언트는 빈에 대한 메타데이터 정보를 얻을 수 있으며, 배포된 엔터프라이즈 빈을 사용하는 애플리케이션을 구축하는 개발 도구에 의해 사용됩니다.

EJBHome 인터페이스에는 엔터프라이즈 빈의 인스턴스를 생성하거나 찾는 메소드가 없습니다. 그 대신 빈을 위해 개발한 홈 인터페이스에 이러한 메소드를 추가해야 합니다. 세션 빈과 엔티티 빈의 생명 주기가 서로 다르기 때문에 각각의 홈 인터페이스에서 정의된 메소드도 다릅니다.

세션 빈에 대한 홈 인터페이스 생성

상태 없는(stateless) 세션 빈의 경우를 제외하면 세션 빈은 거의 항상 단일 클라이언트를 갖습니다. 클라이언트가 세션 빈을 생성하면 그 세션 빈 인스턴스는 해당 클라이언트를 사용할 때만 존재합니다.

다음과 같은 방법으로 세션 빈에 대한 홈 인터페이스를 생성합니다.

- javax.ejb.EJBHome을 확장하는 홈 인터페이스를 선언합니다.
- 빈의 각 ejbCreate() 메소드에 대한 create() 메소드 시그너처를 추가하여 인수의 개수 및 타입을 정확하게 일치시킵니다.

JBuilder의 Enterprise JavaBean 마법사를 사용하면 JBuilder는 엔터프라이즈 빈 클래스를 생성하는 동시에 정의된 create() 메소드가 하나인 홈 인터페이스를 생성합니다. 그런 다음 사용자의 빈에 ejbCreate() 메소드를 추가하면 홈 인터페이스에 다른 create() 메소드를 추가할 수 있습니다. 또는 기존 엔터프라이즈 빈 클래스를 가지는 경우 JBuilder의 EJB Interfaces 마법사를 사용하여 빈 클래스에 있는 것과 적절히 일치하는 시그너처와 함께 홈 인터페이스 및 원격 인터페이스를 만드십시오. 자세한 내용은 Chapter 10 " JBuilder를 사용한 엔터프라이즈 빈 생성"을 참조하십시오.

원격 인터페이스의 생성을 시작으로 EJB를 개발하기로 한 경우에는 EJB Bean Generator를 사용하여 뼈대 빈(bean) 클래스 및 홈 인터페이스를 생성할 수 있습니다. EJB Bean Generator 사용에 관한 자세한 내용은 10- 11페이지의 "원격 인터페이스에서 빈 클래스 생성"을 참조하십시오.

세션 빈의 create() 메소드

세션 홈 인터페이스는 하나 이상의 create() 메소드를 정의해야 하므로 세션 빈 팩토리로서 기능합니다. 클라이언트가 create()를 호출하면 새로운 빈 인스턴스가 생성됩니다. EJB 사양에 따라 홈 인터페이스에서 정의된 각 create() 메소드는 다음을 수행해야 합니다.

- 세션 빈의 원격 인터페이스 타입을 반환해야 합니다.
- create()로 이름이 지정되어야 합니다.
- 세션 빈 클래스의 ejbCreate() 메소드와 일치해야 합니다. 각 create() 메소드에 대한 인수의 개수 및 타입은 세션 빈 클래스의 해당 ejbCreate() 메소드와 일치해야 합니다.
- java.rmi.RemoteException 예외를 발생시킵니다.
- javax.ejb.CreateException 예외를 발생시킵니다.
- 매개변수가 있는 경우 그 매개변수를 사용하여 새로운 세션 빈 객체를 초기화합니다.

EJB 마법사를 사용하면 이러한 규칙을 따르는지 확인할 수 있습니다.

다음 코드 예제는 세션 홈 인터페이스의 가능한 두 가지 create() 메소드를 보여 줍니다. 굵게 표시된 부분은 필수입니다.

```
public interface AtmHome extends javax.ejb.EJBHome {
  Atm create()
    throws java.rmi.RemoteException, javax.ejb.CreateException;
  Atm create(Profile preferredProfile)
    throws java.rmi.RemoteException, javax.ejb.CreateException;
}
```

엔티티 빈에 대한 홈 인터페이스 생성

엔티티 빈은 여러 클라이언트를 사용하도록 디자인되었습니다. 한 클라이언트에서 엔티티 빈 인스턴스를 생성하면 다른 클라이언트에서 사용할 수 있습니다.

다음과 같은 방법으로 엔티티 빈에 대한 홈 인터페이스를 생성합니다.

- javax.ejb.EJBHome을 확장하는 인터페이스를 선언합니다.
- 빈의 각 ejbCreate() 메소드에 대해 create() 메소드 시그너처를 추가하여 시그너처를 정확하게 일치시킵니다.
- 빈의 각 finder 메소드에 대해 finder 메소드 시그너처를 추가하여 시그너처를 정확하게 일치시킵니다.

JBuilder의 Enterprise JavaBean 마법사나 EJB Entity Modeler를 사용하면 JBuilder는 엔터프라이즈 빈 클래스를 생성하는 동시에 정의된 create() 메소드가 하나인 홈 인터페이스를 생성합니다. 그런 다음 사용자의 빈에 ejbCreate() 메소드를 추가하는 경우 홈 인터페이스에 다른 create() 메소드를 추가할 수 있습니다. 또는 기존 엔터프라이즈 빈 클래스를 가지고 있는 경우 JBuilder의 EJB Interfaces 마법사를 사용하여 빈 클래스에 있는 것과 적절히 일치하는 시그너처로 홈 인터페이스 및 원격 인터페이스를 만듭니다. 자세한 내용은 Chapter 10 " JBuilder를 사용한 엔터프라이즈 빈 생성"을 참조하십시오.

원격 인터페이스의 생성을 시작으로 EJB를 개발하기로 한 경우에는 EJB Bean Generator를 사용하여 뼈대 빈(bean) 클래스 및 홈 인터페이스를 생성하십시오. EJB Bean Generator 사용에 관한 자세한 내용은 10- 11페이지의 "원격 인터페이스에서 빈 클래스 생성"을 참조하십시오.

엔티티 빈의 create() 메소드

세션 빈에 대한 홈 인터페이스처럼 엔티티 빈에 대한 홈 인터페이스도 하나 이상의 create() 메소드를 정의해야 합니다. EJB 사양에 따라 정의된 각 create() 메소드는 다음을 수행해야 합니다.

- java.rmi.RemoteException 예외를 발생시켜야 합니다.
- javax.ejb.CreateException 예외를 발생시켜야 합니다.
- 엔티티 빈의 원격 인터페이스 타입을 반환해야 합니다.
- create()로 이름이 지정되어야 합니다.
- 세션 빈 클래스의 ejbCreate() 메소드와 일치해야 합니다. 각 create() 메소드에 대한 인수의 개수 및 타입은 세션 빈 클래스의 해당 ejbCreate() 메소드와 일치해야 합니다.
- 엔티티 빈 클래스에서 해당 ejbCreate() 및 ejbPostCreate() 메소드에 의해 발생된 모든 예외를 throws 절의 예외에 포함시켜야 합니다. 달리 말하면, create() 메소드에 대한 예외의 집합은 ejbCreate()와 ejbPostCreate()의 두 메소드에 대한 예외 합집합의 수퍼셋이 되어야 합니다. ejbCreate() 메소드의 반환 타입은 기본 키 클래스입니다.
- 매개변수를 갖는 경우, 해당 매개변수를 사용하여 새 엔티티 빈 객체를 초기화합니다.

엔티티 빈의 finder 메소드

일반적으로 엔티티 빈은 수명이 길고 여러 클라이언트에 의해 사용될 수 있으므로, 클라이언트 애플리케이션이 엔티티 빈 인스턴스를 필요로 할 때 대개는 이미 존재합니다. 엔티티 빈 인스턴스가 이미 존재하는 경우, 클라이언트는 엔티티 빈 인스턴스를 생성할 필요가 없지만 적절한 기존 인스턴스를 찾아야 합니다. 이는 엔티티 빈의 홈 인터페이스가 하나 이상의 finder 메소드를 정의하는 이유입니다.

세션 빈은 빈을 생성한 애플리케이션인 클라이언트를 하나만 사용하기 때문에 finder 메소드가 필요하지 않습니다. 클라이언트는 인스턴스가 있는 위치를 이미 알고 있기 때문에 세션 빈 인스턴스를 찾을 필요가 없습니다.

각각의 엔티티 빈 홈 인터페이스는 기본 finder 메소드인 `findByPrimaryKey()`를 정의해야 합니다. 이 메소드를 통해 클라이언트는 다음과 같이 기본 키를 사용하여 엔티티 객체를 찾을 수 있습니다.

```
<entity bean's remote interface> findByPrimaryKey(<primary key type> key)
    throws java.rmi.RemoteException, FinderException;
```

`findByPrimaryKey()`는 단일 인수, 기본 키를 가집니다. 이 메소드의 반환 타입은 엔티티 빈의 원격 인터페이스입니다. 빈의 배포 디스크립터에서 기본 키의 타입을 컨테이너에 알립니다. `findByPrimaryKey()`는 항상 단일 엔티티 객체를 반환합니다.

홈 인터페이스에서 추가 finder 메소드를 정의할 수 있습니다. 각 finder 메소드에는 빈에 의해 관리되는 지속성을 위한 엔티티 빈 클래스의 해당 구현이 있어야 합니다. 컨테이너에 의해 관리되는 지속성을 위해 컨테이너는 finder 메소드를 구현합니다. 각 finder 메소드는 다음 규칙을 따라야 합니다.

- finder 메소드의 반환 타입은 원격 인터페이스 타입이며 둘 이상의 엔티티 객체를 반환하는 finder 메소드의 경우에는 컨텐트 타입으로서 원격 인터페이스 타입을 갖는 컬렉션 타입입니다. 유효한 Java 컬렉션 타입은 `java.util.Enumeration`과 `java.util.Collection`입니다.
- finder 메소드는 항상 접두사 **find**로 시작합니다. 엔티티 빈 클래스에 있는 해당 finder 메소드는 접두사 **ejbFind**로 시작합니다.
- 메소드는 `java.rmi.RemoteException` 예외를 발생시켜야 합니다.
- 메소드는 `javax.ejb.FinderException` 예외를 발생시켜야 합니다.
- 홈 인터페이스에 있는 finder 메소드의 throws 절은 엔티티 빈 클래스에 있는 해당 ejbFind<xxx> 메소드의 throws 절과 일치해야 합니다.

다음 예제의 홈 인터페이스에는 두 가지 `create()` 메소드와 두 가지 finder 메소드가 들어 있습니다. 굵게 표시된 부분은 필수입니다.

```
public interface AccountHome extends javax.ejb.EJBHome {

  Account create(String accountID)
    throws java.rmi.RemoteException, javax.ejb.CreateException;

  Account create(String accountID, float initialBalance)
    throws java.rmi.RemoteException, javax.ejb.CreateException;

  Account findByPrimaryKey(String key)
    throws java.rmi.RemoteException, javax.ejb.FinderException;

  Account findBySocialSecurity(String socialSecurityNumber)
    throws java.rmi.RemoteException, javax.ejb.FinderException;
}
```

원격 인터페이스 생성

엔터프라이즈 빈에 대해 생성한 원격 인터페이스는 클라이언트 애플리케이션이 호출할 수 있는 business 메소드를 기술합니다. 원격 인터페이스에서 메소드를 정의하고 엔터프라이즈 빈 클래스에서 이와 동일한 메소드를 구현합니다. 엔터프라이즈 빈의 클라이언트는 빈에 직접 액세스할 수 없습니다. 원격 인터페이스를 통해 메소드에 액세스합니다.

다음과 같은 방법으로 원격 인터페이스를 생성합니다.

- javax.ejb.EJBObject를 확장하는 인터페이스를 선언합니다.
- 클라이언트 애플리케이션이 엔터프라이즈 빈에서 호출할 수 있는 모든 business 메소드를 원격 인터페이스에서 선언하여 시그너처를 해당 빈 클래스의 시그너처와 정확하게 일치시킵니다.

JBuilder의 EJB 마법사를 사용하면 JBuilder는 EJBObject를 확장하는 원격 인터페이스를 생성합니다. 그런 다음 빈 디자이너를 사용하여 원격 인터페이스에 나타내려는 빈의 business 메소드를 지정할 수 있습니다.

원격 인터페이스에 정의된 각 메소드는 세션 빈 및 엔티티 빈 모두에 대해 동일한 다음 규칙을 따라야 합니다.

- 메소드는 public이어야 합니다.
- java.rmi.RemoteException 예외를 발생시켜야 합니다.
- 반환 값과 모든 인수는 유효한 타입의 RMI-IIOP가 되어야 합니다.
- 메소드는 클라이언트가 호출할 수 있는 엔터프라이즈 빈의 클래스에 있는 각 메소드에 대한 원격 인터페이스에 있어야 합니다. 원격 인터페이스의 메소드 및 빈 자체의 메소드는 이름, 인수의 개수 및 타입, 반환 타입이 같아야 하고 동일한 예외를 발생시키거나 원격 인터페이스 메소드의 예외에 대한 서브셋을 발생시켜야 합니다.

다음 코드 예제에서는 ATM 세션 빈의 Atm이라는 샘플 원격 인터페이스에 대한 코드를 보여 줍니다. Atm 원격 인터페이스는 transfer()라는 business 메소드를 정의합니다. 굵게 표시된 부분은 필수입니다.

```
public interface Atm extends javax.ejb.EJBObject{

   public void transfer(String source, String target, float amount)
     throws java.rmi.RemoteException, InsufficientFundsException;
}
```

Atm 인터페이스에 선언된 transfer() 메소드는 두 가지 예외, 필수 예외인 java.rmi.RemoteException과 애플리케이션 특정적인 예외인 InsufficientFundsException를 발생시킵니다.

EJBObject 인터페이스

원격 인터페이스는 javax.ejb.EJBObject 인터페이스를 확장합니다. 다음은 EJBObject의 소스 코드입니다.

```
package javax.ejb;
public interface EJBObject extends java.rmi.Remote (
  public EJBHome getEJBHome() throws java.rmi.RemoteException;
  public Object getPrimaryKey() throws java.rmi.RemoteException;
  public void remove() throws java.rmi.RemoteException, java.rmi.RemoveException;
  public Handle getHandle() throws java.rmi.RemoteException;
  boolean isIdentical (EJBObject other) throws java.rmi.RemoteException;
}
```

getEJBHome() 메소드를 통해 애플리케이션은 빈의 홈 인터페이스를 얻을 수 있습니다. 빈이 엔티티 빈인 경우 getPrimaryKey() 메소드는 빈에 대한 기본 키를 반환합니다. remove() 메소드는 엔터프라이즈 빈을 삭제하고 getHandle()은 빈 인스턴스에 영구적인 핸들을 반환합니다. isIdentical()을 사용하여 두 엔터프라이즈 빈을 비교합니다.

EJB 컨테이너는 엔터프라이즈 빈에 대한 EJBObject를 생성합니다. 원격 인터페이스는 EJBObject 인터페이스를 확장하기 때문에 컨테이너가 생성하는 EJBObject는 원격 인터페이스에서 정의한 모든 business 메소드 뿐만 아니라 EJBObject 인터페이스의 모든 메소드에 대한 구현을 포함합니다. 인스턴스화된 EJBObject는 네트워크 상에 표시되고 빈의 프록시 역할을 하며 스텁(stub)과 뼈대(skeleton)를 가집니다. 빈 자체는 네트워크 상에 표시되지 않습니다.

Chapter

19

엔터프라이즈 빈 클라이언트 개발

엔터프라이즈 빈의 클라이언트는 애플리케이션 독립형 애플리케이션, 또 다른 엔터프라이즈 빈인 서블릿 또는 애플릿입니다. 모든 경우에 있어서 클라이언트는 엔터프라이즈 빈을 사용하기 위해 다음 사항을 수행해야 합니다.

- 빈의 홈 인터페이스를 찾습니다. EJB 사양은 클라이언트가 JNDI(Java Naming 및 Directory Interface) API를 사용하여 홈 인터페이스를 찾도록 규정하고 있습니다.
- 엔터프라이즈 빈 객체의 원격 인터페이스에 참조를 얻습니다. 여기에는 빈의 홈 인터페이스에 정의된 메소드를 사용하는 것도 포함됩니다. 세션 빈을 생성하거나 엔티티 빈을 생성 또는 찾을 수 있습니다.
- 엔터프라이즈 빈에서 정의한 하나 이상의 메소드를 호출합니다. 클라이언트는 엔터프라이즈 빈에서 정의한 메소드를 직접 호출하지 않습니다. 그 대신 클라이언트는 엔터프라이즈 빈 객체의 원격 인터페이스에 있는 메소드를 호출합니다. 원격 인터페이스에 정의된 메소드는 엔터프라이즈 빈이 클라이언트에 노출시킨 메소드입니다.

다음 단원에서는 예제 `SortBean` 세션 빈을 호출하는 클라이언트 애플리케이션인 `SortClient.java`에 대해 설명합니다. `SortBean`은 통합/정렬 알고리즘을 구현하는 상태 없는(stateless) 세션 빈입니다. 다음은 `SortClient`:의 코드입니다.

```
// SortClient.java
...

public class SortClient {
  ...
  public static void main(String[] args) throws Exception {
    javax.naming.Context context;
```

```
    { // get a JNDI context using the Naming service
      context = new javax.naming.InitialContext();
    }
    Object objref = context.lookup("sort");
    SortHome home = (SortHome) javax.rmi.PortableRemoteObject.narrow(objref,
      SortHome.class);
    Sort sort = home.create();
    ... //do the sort and merge work
    sort.remove();
  }
}
```

홈 인터페이스 찾기

SortClient는 필요한 JNDI 클래스와 SortBean 홈 인터페이스 및 원격 인터페이스를 import합니다. 클라이언트는 JNDI API를 사용하여 엔터프라이즈 빈의 홈 인터페이스를 찾습니다. 먼저 클라이언트는 JNDI 초기 이름 지정 컨텍스트를 획득해야 합니다. SortClient의 코드는 InitialContext라는 새로운 javax.naming.Context 객체를 인스턴스화합니다. 그런 다음 클라이언트는 컨텍스트 lookup() 메소드를 사용하여 이름을 홈 인터페이스로 변경합니다.

컨텍스트의 lookup() 메소드는 java.lang.Object 타입의 객체를 반환합니다. 사용자의 코드는 반환된 객체를 원하는 타입으로 변환해야 합니다. SortClient 코드는 정렬 예제에 대한 클라이언트 코드의 일부분을 보여 줍니다. main() 루틴은 JNDI 이름 지정 서비스와 해당 컨텍스트 lookup() 메소드를 사용하여 홈 인터페이스를 찾기 시작합니다. 이 경우 sort인 원격 인터페이스의 이름을 context.lookup() 메소드에 전달합니다. 프로그램은 결과적으로 context.lookup() 메소드의 결과를 홈 인터페이스의 타입인 SortHome로 변환한다는 것에 유의하십시오.

원격 인터페이스 얻기

일단 엔터프라이즈 빈의 홈 인터페이스를 가지면 빈의 원격 인터페이스에 대한 참조가 필요합니다. 빈의 원격 인터페이스에 대한 참조를 얻으려면 홈 인터페이스의 create 메소드나 finder 메소드를 사용합니다. 사용할 메소드는 엔터프라이즈 빈 타입과 빈 provider가 홈 인터페이스에 정의한 메소드에 따라 다릅니다.

세션 빈

엔터프라이즈 빈이 세션 빈인 경우 클라이언트는 create 메소드를 사용하여 원격 인터페이스를 반환합니다. 세션 빈에는 finder 메소드가 없습니다. 세션 빈이 상태 없음(stateless)일 경우 세션 빈은 단 한 개의 create() 메소드를 갖게 되므로 이 메소드가 클라이언트가 원격 인터페이스를 얻기 위해 호출해야 하는 메소드가 됩니다. 기본 create() 메소드는 매개변수를 가지

지 않습니다. 그러므로 SortClient 코드 예제의 경우 원격 인터페이스를 얻기 위한 호출은 다음과 같습니다.

```
Sort sort = home.create();
```

반면 Chapter 16, □세션 빈 개발,淡【□ 논의된 카트 예제(cart example)는 상태 있는(stateful) 세션 빈을 사용하고 카트 예제의 홈 인터페이스인 CartHome은 둘 이상의 create() 메소드를 구현합니다. create() 메소드 중 하나는 3개의 매개변수를 갖는데 이들은 카트 내용의 구매자를 식별하고 Cart 원격 인터페이스에 참조를 반환합니다. CartClient는 cardHolderName, creditCardNumber와 expirationDate, 3개의 매개변수의 값을 설정한 다음 create() 메소드를 호출합니다. 코드는 다음과 같습니다.

```
Cart cart;
{
  String cardHolderName = "Jack B. Quick";
  String creditCardNumber = "1234-5678-9012-3456";
  Date expirationDate = new GregorianCalendar(2001, Calendar.JULY, 1).getTime();
  cart = home.create(cardHolderName, creditCardNumber, expirationDate);
}
```

엔티티 빈

엔티티 빈의 경우 create 메소드나 finder 메소드를 사용하여 원격 인터페이스를 얻습니다. 엔티티 객체는 데이터베이스에 저장되어 있는 일부 데이터를 나타내고 이 데이터는 지속적이기 때문에 엔티티 빈은 일반적으로 오랜 시간 동안 존재합니다. 따라서 클라이언트는 새로운 데이터를 생성하여 원본으로 사용하는 데이터베이스에 저장하는 새 엔티티 객체를 만들지 않고 이러한 데이터를 나타내는 엔티티 빈을 단순히 찾기만 하면 됩니다.

클라이언트는 찾기 작업을 사용하여 관계형 데이터베이스 테이블 안에 있는 특정 행과 같은 기존 엔티티 객체를 찾습니다. 다시 말해서 찾기 작업(find operation)으로 데이터 저장소에 이미 삽입되어 있는 데이터 엔티티를 찾습니다. 데이터는 엔티티 빈에 의해 데이터 저장소에 추가되었거나, EJB 컨텍스트의 외부, 예를 들면 데이터베이스 관리 시스템(DBMS)에서 직접 추가되었을 수도 있습니다. 또는 레거시(legacy) 시스템의 경우, EJB 컨테이너를 설치하기 이전에 데이터가 존재했었을 수도 있습니다.

클라이언트는 엔티티 빈 객체의 create() 메소드를 사용하여 원본으로 사용하는 데이터베이스에 저장될 새로운 데이터 엔티티를 생성합니다. 엔티티 빈의 create() 메소드는 엔티티 상태를 데이터베이스에 삽입하여 create() 메소드의 매개변수 내의 값에 따라 엔티티의 변수를 초기화합니다

각각의 엔티티 빈 인스턴스는 고유하게 식별하는 기본 키를 가져야 합니다. 엔티티 빈 인스턴스는 또한 특정 엔티티 객체를 찾는 데 사용될 수 있는 보조 키를 가질 수 있습니다.

Finder 메소드 및 기본 키 클래스

엔티티 빈용 기본 finder 메소드는 기본 키 값을 사용하여 엔티티 객체를 찾는 findByPrimaryKey()입니다. 다음은 findByPrimaryKey()의 시그너처입니다.

```
<remote interface> findByPrimaryKey(<key type> primaryKey)
```

각각의 엔티티 빈은 findByPrimaryKey() 메소드를 구현해야 합니다. primaryKey 매개변수는 배포 디스크립터에 정의되어 있는 별도의 기본 키 클래스입니다. 키 타입은 기본 키 에 대한 타입이고 RMI-IIOP에 적합한 값 타입(value type)이어야 합니다. 기본 키 클래스는 자바 클래스나 사용자가 작성한 클래스 같은 모든 클래스가 될 수 있습니다.

예를 들어, 기본 키 클래스 AccountPK를 정의한 Account 엔티티 빈을 가진다고 가정합니다. String 타입인 AccountPK는 Account 빈에 대한 식별자를 갖습니다. 특정 Account 엔티티 빈 인스턴스에 대한 참조를 얻으려면 AccountPK를 계정 식별자(account identifier)에 설정하고 다음과 같이 findByPrimaryKey() 메소드를 호출합니다.

```
AccountPK accountPK = new AccountPK("1234-56-789");
Account source = accountHome.findByPrimaryKey(accountPK);
```

빈 provider는 클라이언트가 사용할 수 있는 finder 메소드를 추가로 정의할 수 있습니다.

Create 메소드 및 Remove 메소드

클라이언트는 홈 인터페이스에 정의된 create 메소드를 사용하여 엔티티 빈을 생성할 수도 있습니다. 클라이언트가 엔티티 빈에 대해 create 메소드를 호출하면 엔티티 객체의 새 인스턴스가 데이터 저장소에 저장됩니다. 새 엔티티 객체는 자신의 식별자인 기본 키 값을 항상 갖습니다. 엔티티 객체의 상태는 create 메소드에 매개변수로서 전달된 값으로 초기화될 수 있습니다.

데이터가 데이터베이스에 있는 한 엔티티 빈은 존재합니다. 엔티티 빈의 수명은 클라이언트의 세션에 의해 제한되지 않습니다. 빈의 remove 메소드 중 하나를 호출하여 엔티티 빈을 제거할 수 있습니다. 이러한 메소드는 빈을 제거하고 원본으로 사용하는 데이터베이스에서 엔티티 데이터의 표현을 제거합니다. DBMS나 레거시 애플리케이션을 사용하여 데이터베이스 레코드를 제거하는 것처럼 엔티티 객체를 직접 삭제할 수도 있습니다.

메소드 호출

빈의 원격 인터페이스에 대한 참조를 가지게 되면 클라이언트는 빈의 원격 인터페이스에 정의되어 있는 메소드를 호출할 수 있습니다. 클라이언트에서는 빈의 비즈니스 로직을 구체화하는 메소드가 가장 중요합니다.

예를 들어, 다음은 카트 세션 빈에 액세스하는 클라이언트의 일부 코드입니다. 다음 코드는 카드 홀더를 위한 새로운 세션 빈 인스턴스를 생성하고 원

격 인터페이스에 Cart 참조를 재시도했던 지점으로부터 시작합니다. 클라이언트는 빈 메소드를 호출할 준비가 된 상태입니다.

```
...
Cart cart;
{
  ...
  // obtain a reference to the bean's remote interface
  cart = home.create(cardHolderName, creditCardNumber, expirationDate);
}

// create a new book object
Book knuthBook = new Book("The Art of Computer Programming", 49.95f);

// add the new book item to the cart
cart.addItem(knuthBook);
...

// list the items currently in the cart
summarize(cart);
cart.removeItem(knuthBook);
...
```

우선 클라이언트는 새로운 예약(book) 객체를 생성하여 title과 price 매개변수를 설정합니다. 그런 다음 클라이언트는 엔터프라이즈 빈 비즈니스 메소드인 addItem()을 호출하여 쇼핑 카트에 예약 객체를 추가합니다. Cart 세션 빈은 addItem() 메소드를 정의하고 Cart 원격 인터페이스는 이 메소드를 public으로 만듭니다. 클라이언트는 다른 항목(위의 코드에서는 보이지 않습니다)을 추가한 다음 자신의 summarize() 메소드를 호출하여 쇼핑 카트 내의 항목을 나열합니다. 그런 다음, 빈 인스턴스를 제거하는 remove() 메소드가 옵니다. 클라이언트는 자신의 summarize() 메소드를 호출하는 것과 동일한 방식으로 엔터프라이즈 빈 메소드를 호출합니다.

빈 인스턴스 제거

remove() 메소드는 엔티티 빈과 세션 빈에 대해 각기 다르게 작동합니다. 하나의 클라이언트에는 하나의 세션 객체만이 존재하고 이 세션 객체는 지속적이지 않기 때문에 세션 빈의 클라이언트는 세션 객체에 대한 작업이 완료되었을 때 remove() 메소드를 호출해야 합니다. 두 개의 remove() 메소드를 클라이언트에 사용될 수 있습니다: 클라이언트는 javax.ejb.EJBObject.remove() 메소드를 사용하여 세션 객체를 제거할 수 있고 javax.ejb.EJBHome.remove(Handle handle) 메소드를 사용하여 세션 핸들을 제거할 수 있습니다. 빈 핸들에 대한 자세한 내용은 19- 6페이지의 "핸들을 사용하여 빈 참조"를 참조하십시오.

클라이언트가 세션 객체를 반드시 제거해야 되는 것은 아니지만 이는 좋은 프로그래밍 습관이라 할 수 있습니다. 클라이언트가 상태 있는 세션 빈 객체를 제거하지 않은 경우 시간 초과 값에 의해 지정된 일정 시간이 경과하면 결국 컨테이너가 세션 빈 객체를 제거합니다. 시간 초과 값은 배포 속성

에 해당합니다. 하지만 클라이언트는 나중에 참조할 것을 대비하여 핸들을 세션에 보관할 수도 있습니다.

엔티티 빈은 트랜잭션 기간 동안에만 클라이언트와 연결되어 있고 컨테이너가 클라이언트의 활성화 및 부동화와 같은 수명 주기에 대한 책임을 맡고 있으므로 엔티티 빈의 클라이언트에는 이러한 문제가 발생하지 않습니다. 엔티티 빈의 클라이언트는 엔티티 객체가 원본으로 사용하는 데이터베이스에서 지워질 경우에만 빈의 `remove()` 메소드를 호출합니다.

핸들을 사용하여 빈 참조

핸들은 엔터프라이즈 빈을 참조하는 또다른 방법입니다. 핸들은 빈에 대해 직렬화할 수 있는 참조입니다. 빈의 원격 인터페이스로부터 핸들을 얻을 수 있습니다. 일단 핸들을 얻으면 파일이나 지속적인 다른 저장소에 핸들을 쓸 수 있습니다. 나중에 저장소에서 핸들을 가져와서 핸들을 사용하여 엔터프라이즈 빈에 대한 참조를 다시 만들수 있습니다.

원격 인터페이스 핸들을 사용하여 빈에 대한 참조만을 다시 생성 할 수는 있지만 원격 인터페이스 핸들을 사용하여 빈 자체를 다시 생성할 수는 없습니다. 또다른 프로세스를 통해 빈이 제거되었거나 시스템이 빈 인스턴스를 제거했으면 클라이언트가 핸들을 사용해 빈에 대한 참조를 다시 만들고자 할 경우 예외가 발생합니다.

빈 인스턴스가 여전히 존재하는지 확신할 수 없을 경우에는 원격 인터페이스에 핸들을 사용하는 대신 빈의 홈 핸들을 저장하고 후에 빈의 creat 메소드나 finder메소드를 호출함으로써 빈 객체를 다시 만들 수 있습니다.

빈 인스턴스를 만든 후 클라이언트는 `getHandle()` 메소드를 사용하여 이 인스턴스에 대한 핸들을 얻을 수 있습니다. 일단 핸들을 얻으면 클라이언트는 직렬화된 파일에 핸들을 쓸 수 있습니다. 나중에 클라이언트 프로그램은 직렬화된 파일을 읽어, 읽은 객체를 `Handle` 타입으로 변환할 수 있습니다. 그런 다음 클라이언트 프로그램은 핸들에 있는 `getEJBObject()` 메소드를 호출하여 빈 참조를 얻어 `getEJBObject()`의 결과를 빈에 올바른 타입으로 좁힙니다.

예를 들어, `CartClient` 프로그램은 다음과 같은 코드를 실행하여 `Cart` 세션 빈에 핸들을 사용할 수도 있습니다.

```
import java.io;
import javax.ejb.Handle;
...
Cart cart;
...
  cart = home.create(cardHolderName, creditCardNumber, expirationDate);

// call getHandle() on the cart object to get its handle
cartHandle = cart.getHandle();
```

```
// write the handle to serialized file
FileOutputStream f = new FileOutputStream ("carthandle.ser");
ObjectOutputStream o = new ObjectOutputStream(f);
o.writeObject(myHandle);
o.flush();
o.close();
...

// read handle from file at later time
FileInputStream fi = new FileInputStream ("carthandle.ser");
ObjectInputStream oi = new ObjectInputStream(fi);

//read the object from the file and cast it to a Handle
cartHandle = (Handle)oi.readObject();
oi.close();
...

// Use the handle to reference the bean instance
Cart cart = (Cart)
javax.rmi.PortableRemoteObject.narrow(cartHandle.getEJBObject(),
    Cart class);
...
```

세션 빈 핸들 작업을 마쳤을 경우 클라이언트는 `javax.ejb.EJBHome.remove(Handle handle)` 메소드를 호출하여 세션 핸들 빈을 제거할 수 있습니다.

트랜잭션 관리

클라이언트 프로그램은 엔터프라이즈 빈이나 컨테이너가 트랜잭션을 관리하게 하는 대신 자신의 트랜잭션을 관리할 수 있습니다. 클라이언트는 자신의 트랜잭션을 관리하는 세션 빈과 동일한 방식으로 자신의 트랜잭션을 관리합니다.

자신의 트랜잭션을 관리할 경우 트랜잭션 범위를 구분할 책임은 클라이언트에게 있습니다. 즉, 클라이언트는 명시적으로 트랜잭션을 시작하고 종료(커밋 또는 롤백)해야 합니다.

클라이언트는 `javax.transaction.UserTransaction` 인터페이스를 사용하여 자신의 트랜잭션을 관리합니다. 그렇게 하려면 JNDI를 사용하여 먼저 `UserTransaction` 인터페이스에 대한 참조를 얻어야 합니다. 일단 `UserTransaction` 컨텍스트를 가질 경우에는 `UserTransaction.begin()` 메소드를 사용하여 트랜잭션을 시작한 후 `UserTransaction.commit()` 메소드를 사용하여 트랜잭션을 커밋하고 종료하거나 `UserTransaction.rollback()`을 사용하여 트랜잭션을 롤백하고 종료합니다. 이 사이에서 클라이언트는 EJB 객체 등에 액세스합니다.

다음 코드는 클라이언트가 자신의 트랜잭션을 관리하는 방법을 보여 줍니다. 특별히 클라이언트에 의해 관리되는 트랜잭션에 속하는 코드는 굵게 표시되어 있습니다.

```
...
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
...
public class clientTransaction {
  public static void main (String[] argv) {
    InitialContext initContext = new InitialContext();
    UserTransaction ut = null;
    ut = (UserTransaction)initContext.lookup("java:comp/UserTransaction");

    // start a transaction
    ut.begin();

    // do some transaction work
    ...

    // commit or rollback the transaction
    ut.commit(); // or ut.rollback();
    ...
  }
}
```

트랜잭션에 대한 자세한 내용은 Chapter 20 "트랜잭션 관리"를 참조하십시오.

빈 정보 발견

엔터프라이즈 빈에 대한 정보를 메타데이터라고 부릅니다 클라이언트는 엔터프라이즈 빈의 홈 인터페이스인 `getMetaData()` 메소드를 사용하여 빈에 대한 메타데이터를 얻을 수 있습니다.

`getMetaData()` 메소드는 이미 설치된 빈을 함께 연결하기 위해서 엔터프라이즈 빈에 대한 정보를 필요로 하는 툴 빌더와 개발 환경에 의해 가장 많이 사용됩니다. 클라이언트 스크립팅 또한 빈에 대한 메타데이터를 얻으려 할 수도 있습니다.

일단 클라이언트가 홈 인터페이스 참조를 가져오면 클라이언트는 자신의 `getEJBMetaData()` 메소드를 호출할 수 있습니다. 그런 다음 클라이언트는 `EJBMetaData` 인터페이스 메소드를 호출하여 다음과 같은 정보를 추출할 수 있습니다.

- `EJBMetaData.getEJBHome()` 메소드를 사용하여 빈의 `EJBHome` 홈 인터페이스에 대한 정보 추출.
- `EJBMetaData.getHomeInterfaceClass()` 메소드를 사용하여 인터페이스, 클래스, 필드 및 메소드를 포함하는 빈의 홈 인터페이스 클래스 객체에 대한 정보 추출.

- EJBMetaData.getRemoteInterfaceClass() 메소드를 사용하여 클래스 정보를 포함하는 빈의 원격 인터페이스 클래스 객체에 대한 정보 추출.
- EJBMetaData.getPrimaryKeyClass() 메소드를 사용하여 빈의 기본 키 클래스 객체에 대한 정보 추출.
- EJBMetaData.isSession() 메소드를 사용하여 빈이 세션 빈인지 엔티티 빈인지 여부에 관한 정보 추출. 세션 빈일 경우 메소드는 true를 반환합니다.
- EJBMetaData.isStatelessSession() 메소드를 사용하여 세션 빈이 상태 없음(stateless)인지 상태 있음(stateful)인지 여부에 대한 정보 추출. 세션 빈이 상태 없음(stateless)일 경우 메소드는 true를 반환합니다.

다음은 EJBMetaData 인터페이스의 전체 코드입니다.

```
package javax.ejb;

public interface EJBMetaData {
  EJBHome getEJBHome();
  Class getHome InterfaceClass();
  Class getRemoteInterfaceClass();
  Class getPrimaryKeyClass();
  boolean isSession();
  boolean isStatelessSession();
}
```

JBuilder로 클라이언트 생성

JBuilder를 사용하여 클라이언트 생성을 시작할 수 있습니다. JBuilder에는 엔터프라이즈 빈을 테스트하기 위한 단일 클라이언트 애플리케이션을 만드는 EJB Test Client 마법사가 있습니다. EJB Test Client 마법사를 사용하여 실제 클라이언트 애플리케이션 구축을 시작할 수도 있습니다. 마법사에 클라이언트가 액세스 할 엔터프라이즈 빈 중 하나의이름을 알리면 마법사는 이름 지정 컨텍스트가 얻는 코드를 작성하고 빈의 홈 인터페이스를 찾으며 원격 인터페이스에 대한 참조를 얻습니다.

하지만 클라이언트가 여러 개의 빈을 호출할 수도 있으므로 클라이언트가 액세스하는 다른 빈에 대한 클라이언트 코드에서 이러한 단계를 수행해야 합니다. 그런 다음 엔터프라이즈 빈의 비즈니스 로직에 액세스하는 호출을 클라이언트 코드에 직접 추가해야 합니다.

EJB Test Client 마법사 사용에 대한 자세한 내용은 12- 1페이지의 "테스트 클라이언트 생성"을 참조하십시오.

Chapter

20

트랜잭션 관리

트랜잭션을 지원하는 Java 2 Enterprise Edition(J2EE) 같은 플랫폼에서 애플리케이션을 개발하는 것이 좋습니다. 트랜잭션 기반 시스템은 오류 복구나 멀티 유저 프로그래밍과 같은 복잡한 문제로부터 개발자를 해방시켜 주기 때문에 애플리케이션 개발을 쉽게 만듭니다. 트랜잭션은 단일 데이터베이스나 단일 사이트로 제한되지않습니다. 분산 트랜잭션은 여러 사이트에 있는 여러 데이터베이스를 동시에 업데이트할 수 있습니다.

개발자들은 대개 애플리케이션의 전체 작업을 연속된 유닛으로 나눕니다. 작업의 각 유닛은 개별적인 트랜잭션입니다. 애플리케이션이 진행될 때 기본으로 사용하는 시스템에서는 각각의 작업 유닛인 각 트랜잭션이 다른 프로세스로부터 간섭 받지 않고 완료하는지 확인니다. 그렇지 않을 경우 시스템은 트랜잭션을 롤백하고 트랜잭션이 수행했던 작업을 되돌려 애플리케이션을 트랜잭션 시작 이전 상태가 되도록 합니다.

트랜잭션의 특성

일반적으로 트랜잭션은 데이터베이스에 액세스하는 작업을 말합니다. 데이터베이스에 대한 모든 액세스는 트랜잭션 컨텍스트에서 발생합니다. 모든 트랜잭션은 머리글자를 따서 ACID로 나타내는 다음과 같은 특성을 공유합니다.

- 원자성(Atomicity)

  일반적으로 트랜잭션은 하나 이상의 작업으로 구성됩니다. 원자성은 완료될 트랜잭션의 모든 작업이 성공적으로 수행되었음을 요구합니다. 트랜잭션의 모든 작업이 수행되지 않은 경우에는 어떠한 트랜잭션 작업도 수행될 수 없습니다.

- 일관성(Consistency)

  일관성은 데이터베이스의 일관성을 말합니다. 트랜잭션은 하나의 일관성 상태에서 또 다른 일관성 상태로 데이터베이스를 이동시켜야 하고 데이터베이스의 의미적 및 물리적 무결성을 유지해야 합니다.

- 독립성(Isolation)

  독립성은 각 트랜잭션이 데이터베이스에서 현재 데이터를 처리하고 있는 유일한 트랜잭션이어야 함을 요구합니다. 다른 트랜잭션을 동시에 실행할 수 있다 하더라도 작업을 성공적으로 완료하고 커밋하지 않는 한 트랜잭션은 이러한 처리를 볼 수 없습니다. 업데이트 간의 상호 의존성 때문에 트랜잭션이 다른 트랜잭션의 업데이트 서브셋을 보는 것만으로도 데이터를 비일관된 것으로 간주할 수 있습니다. 독립성은 이러한 종류의 데이터 비일관성으로부터 트랜잭션을 보호합니다.

  독립성은 트랜잭션 동시성에 관련됩니다. 독립 레벨이 있습니다. 높은 레벨의 독립성은 동시성의 범위를 제한합니다. 가장 높은 레벨의 독립성은 모든 트랜잭션이 직렬화될 수 있을 때 발생합니다. 즉, 데이터베이스 내용은 각 트랜잭션이 스스로 실행하여 다음 트랜잭션이 시작하기 전에 완료될 경우 나타납니다. 하지만 일부 애플리케이션은 더 높은 레벨의 동시성을 위해 독립성의 레벨이 감소되는 것을 허용하기도 합니다. 대개 이러한 애플리케이션은 트랜잭션이 부분적으로 업데이트되고 비일관된 데이터를 읽는다고 할지라도 상당히 많은 동시성 트랜잭션을 실행합니다.

- 지속성(Durability)

  지속성은 커밋된 트랜잭션에 의해 만들어진 업데이트가 오류 상황에 관계 없이 데이터베이스에서 지속된다는 것을 의미합니다. 지속성은 커밋 작업 후 발생한 오류에도 불구하고 커밋된 업데이트가 데이터베이스에 남아 있도록 하며 데이터베이스가 시스템이나 미디어 오류 후에 복구될 수 있도록 합니다.

컨테이너에서의 트랜잭션 지원

EJB 컨테이너는 일반 트랜잭션을 지원하지만 중첩된 트랜잭션을 지원하지는 않습니다. EJB 컨테이너는 또한 트랜잭션을 암시적으로 전달합니다. 이는 컨테이너가 클라이언트에 대한 작업을 투명하게 처리하기 때문에 트랜잭션 컨텍스트를 매개변수로서 명시적으로 전달할 필요가 없다는 것을 의미합니다.

JSP와 서블릿은 클라이언트로서 동작할 수 있는 반면 트랜잭션 컴포넌트로 디자인되지 않았습니다. 엔터프라이즈 빈을 사용하여 트랜잭션 작업을 수행하십시오. 엔터프라이즈 빈을 호출하여 트랜잭션 작업을 수행할 경우 빈과 컨테이너는 트랜잭션을 적절하게 설정합니다.

엔터프라이즈 빈 및 트랜잭션

엔터프라이즈 빈과 EJB 컨테이너는 트랜잭션 관리를 매우 쉽게 만듭니다. 엔터프라이즈 빈을 통해 애플리케이션은 단일 트랜잭션으로 여러 데이터베이스에 있는 데이터를 업데이트할 수 있고 이러한 데이터베이스는 여러 EJB 서버에 상주할 수 있습니다.

일반적으로 트랜잭션 관리를 책임지는 애플리케이션은 다음 작업을 수행해야 합니다.

- 트랜잭션 객체 생성
- 명시적으로 트랜잭션 시작
- 트랜잭션 컨텍스트 추적
- 모든 업데이트가 완료되었을 경우 트랜잭션 커밋

이러한 애플리케이션은 매우 숙련된 개발자를 필요로 했을 뿐 아니라 오류가 발생하기도 쉬웠습니다.

엔터프라이즈 빈을 사용하면 컨네이너가 모든 트랜잭션은 아니지만 대부분의 트랜잭션을 관리합니다. 컨테이너는 트랜잭션을 시작하고 종료하며 트랜잭션 객체의 수명 동안 트랜잭션 컨텍스트를 유지 관리합니다. 분산 환경에서의 트랜잭션에 대한 사용자의 책임이 급격히 줄어듭니다.

엔터프라이즈 빈의 트랜잭션 속성은 배포 시에 선언됩니다. 이러한 트랜잭션 속성은 컨테이너가 빈의 트랜잭션을 관리하는지, 빈이 자신의 트랜잭션을 관리하는지 여부 및 관리 범위는 어디까지인지를 나타냅니다.

빈에 의해 관리되는 트랜잭션과 컨테이너에 의해 관리되는 트랜잭션 비교

엔터프라이즈 빈이 business 메소드의 일부로서 자신의 트랜잭션 구분을 프로그램에서 수행할 경우 빈에 의해 관리되는 트랜잭션을 사용하는 것으로 간주됩니다. (트랜잭션을 구분한다는 것은 트랜잭션이 시작하고 종료하는 곳을 지정한다는 의미입니다.) 빈이 모든 트랜잭션 구분을 자신의 EJB 컨테이너에 맡길 경우 컨테이너는 애플리케이션 어셈블러의 배포 지침에 따라 트랜잭션을 구분합니다. 이를 컨테이너에 의해 관리되는 트랜잭션을 사용한다고 말합니다.

상태 있는(stateful) 세션 빈과 상태 없는(stateless) 세션 빈은 두 가지 타입의 트랜잭션 중 어느 것이나 사용할 수 있습니다. 하지만 빈은 동시에 두 가지 타입의 트랜잭션 관리를 모두 사용할 수는 없습니다. 빈 provider는 세션 빈이 사용할 타입을 결정합니다. 엔티티 빈은 컨테이너에 의해 관리되는 트랜잭션만 사용할 수 있습니다.

한 작업의 일부로서 트랜잭션을 시작한 다음 다른 작업의 일부로서 트랜잭션을 완료하고자 할 경우, 빈이 자신의 트랜잭션을 관리하기를 원할 수도 있습니다. 하지만 하나의 작업이 메소드를 시작하는 트랜잭션을 호출하였지만 메소드를 종료하는 트랜잭션을 호출하는 작업이 없을 경우에는 문제가 발생합니다.

가능하면, 컨테이너에 의해 관리되는 트랜잭션을 사용하는 엔터프라이즈 빈을 작성해야 합니다. 엔터프라이즈 빈은 사용자측의 작업을 덜 필요로 하며 오류를 덜 발생시킵니다. 또한 컨테이너에 의해 관리되는 트랜잭션으로 빈을 사용자 지정하고 빈을 구성하는것이 더 쉽습니다.

트랜잭션 속성

컨테이너에 의해 관리되는 트랜잭션을 사용하는 엔터프라이즈 빈은 빈의 각 메소드나 전체 빈과 연결된 트랜잭션 속성을 갖습니다. 속성 값은 빈을 포함하는 트랜잭션을 관리하는 방법을 컨테이너에 알려 줍니다. 트랜잭션 속성에는 애플리케이션 어셈블러 또는 배포자(deployer)가 빈의 각 메소드에 연결할 수 있는 다음의 6가지 속성이 있습니다.

- Required

  연결된 메소드에 의해 수행된 작업이 전역 트랜잭션 컨텍스트 안에 있음을 보장합니다. 호출자가 이미 트랜잭션 컨텍스트를 가지고 있을 경우 컨테이너는 동일한 컨텍스트를 사용합니다. 호출자가 트랜잭션 컨텍스트를 가지고 있지 않을 경우 컨테이너는 자동으로 새로운 트랜잭션을 시작합니다. 이러한 속성을 사용하여 여러 개의 빈을 쉽게 구성할 수 있고 동일한 전역 트랜잭션을 사용하여 모든 빈의 작업을 쉽게 통합할 수 있습니다.

- RequiresNew

  기존의 트랜잭션에 연결된 메소드를 사용하지 않고자 할 때 사용됩니다. 컨테이너가 항상 새로운 트랜잭션을 시작하도록 합니다.

- Supports

  메소드가 전역 트랜잭션 사용을 피할 수 있도록 합니다. 빈의 메소드가 하나의 트랜잭션 리소스에만 액세스하거나 어떠한 트랜잭션 리소스에도 액세스하지 않을 경우와 메소드가 다른 엔터프라이즈 빈을 호출하지 않을 경우에만 이 속성을 사용하십시오. 이 속성을 사용하면 전역 트랜잭션과 연관된 비용이 들지 않기 때문에 빈을 최적화합니다. 이 속성이 설정되고 전역 트랜잭션이 이미 존재할 경우, 컨테이너는 메소드를 호출하고 기존의 전역 트랜잭션을 조인합니다. 전역 트랜잭션이 없을 경우 컨테이너는 메소드의 지역 트랜잭션을 시작하며 트랜잭션은 메소드의 끝에서 완료됩니다.

- NotSupported

  빈이 전역 트랜잭션 사용을 피할 수 있도록 해줍니다. 클라이언트가 트랜잭션 컨텍스트를 사용하여 메소드를 호출할 경우 컨테이너는 메소드를 일시 중지시킵니다. 메소드의 끝에서 전역 트랜잭션이 다시 실행됩니다.

- Mandatory

  이 트랜잭션 속성을 사용하여 메소드를 호출하는 클라이언트는 미리 연관된 트랜잭션을 가지고 있어야 합니다. 그렇지 않을 경우 컨테이너는 `javax.transaction.TransactionRequiredException`을 발생시킵니다. 이 속성

을 사용하면 빈은 호출자의 트랜잭션에 대한 가정을 해야되기 때문에 구성에 대하여 덜 유연해집니다.

- Never

  이 트랜잭션 속성을 사용하여 메소드를 호출하는 클라이언트는 트랜잭션 컨텍스트를 가져서는 안됩니다. 그렇지 않을 경우 컨테이너는 예외를 발생시킵니다.

지역 트랜잭션 및 전역 트랜잭션

데이터베이스에 하나의 연결만이 존재할 경우 엔터프라이즈 빈은 연결 상에서 `commit()` 또는 `rollback()`을 호출하여 트랜잭션을 직접 제어 할 수 있습니다. 이러한 타입의 트랜잭션을 지역 트랜잭션이라 합니다. 전역 트랜잭션을 사용하면 트랜잭션을 처리하는 전역 트랜잭션 서비스에 모든 데이터베이스 연결이 등록됩니다. 전역 트랜잭션의 경우 엔터프라이즈 빈은 데이터베이스 연결 자체를 직접 호출하지 않습니다.

빈에 의해 관리되는 트랜잭션 구분을 사용하는 빈은 `javax.transaction.UserTransaction` 인터페이스를 사용하여 전역 트랜잭션의 범위를 식별합니다. 빈이 컨테이너에 의해 관리되는 구분을 사용할 경우 컨테이너는 애플리케이션 어셈블러에 의해 빈의 배포 디스크립터에 설정된 트랜잭션 속성을 사용하여 각각의 클라이언트 호출을 인터럽트하여 트랜잭션 구분을 제어합니다. 트랜잭션 속성은 또한 트랜잭션이 지역인지 전역인지를 결정합니다.

컨테이너에 의해 관리되는 트랜잭션의 경우 컨테이너는 지역 트랜잭션 및 전역 트랜잭션을 실행해야 하는 시기를 결정하기 위한 특정 규칙을 따릅니다. 일반적으로 컨테이너는 전역 트랜잭션이 이미 존재하지 않음을 확인한 후 지역 트랜잭션 내에서 메소드를 호출합니다. 또한 새로운 전역 트랜잭션을 시작하지 않는다는 것과 트랜잭션 속성이 컨테이너에 의해 관리되는 트랜잭션을 위해 설정된 것을 확인합니다. 다음 조건 중 하나가 true일 경우 컨테이너는 자동으로 지역 트랜잭션 내에서 메소드 호출을 래핑합니다.

- 트랜잭션 속성이 NotSupported로 설정되어 있고 컨테이너는 데이터베이스 리소스가 액세스되어 있음을 감지합니다.
- 트랜잭션 속성이 Supports로 설정되어 있고 컨테이너는 메소드가 전역 트랜잭션 내에서 호출되지 않았음을 감지합니다.
- 트랜잭션 속성이 Never로 설정되어 있고 컨테이너는 데이터베이스 리소스가 액세스되어 있음을 감지합니다.

트랜잭션 API 사용

모든 트랜잭션은 Java Transaction API(JTA)를 사용합니다. 트랜잭션이 컨테이너에 의해 관리될 경우, 플랫폼은 트랜잭션 범위 구분을 처리하고

컨테이너는 JTA API를 사용합니다. 빈 코드에서는 이 API를 사용할 필요가 없습니다.

빈이 자신의 트랜잭션을 관리할 경우에는 JTA `javax.transaction.UserTransaction` 인터페이스를 사용해야 합니다. 이 인터페이스를 통해 클라이언트 또는 컴포넌트는 트랜잭션 범위를 구분할 수 있습니다. 빈에 의해 관리되는 트랜잭션을 사용하는 엔터프라이즈 빈은 `EJBContext.getUserTransaction()` 메소드를 사용합니다.

또한 모든 트랜잭션 클라이언트는 JNDI를 사용하여 `UserTransaction` 인터페이스를 찾습니다. JNDI 이름 지정 서비스를 사용하여 다음과 같이 JNDI `InitialContext`를 생성해서 인터페이스를 찾습니다.

```
javax.naming.Context context = new javax.naming.InitialContext();
```

일단 빈에 `InitialContext`가 있으면 빈은 JNDI `lookup()` 작업을 사용하여 `UserTransaction` 인터페이스를 얻을 수 있습니다:

```
javax.transaction.UserTransaction utx = (javax.transaction.UserTransaction)
    context.lookup("java:comp/UserTransaction")
```

엔터프라이즈 빈은 `EJBContext` 객체로부터 `UserTransaction` 인터페이스에 대한 참조를 얻을 수 있습니다. 빈은 `InitialContext` 객체를 얻은 다음 JNDI `lookup()` 메소드를 사용해야 하는 대신 간단하게 `EJBContext.getUserTransaction()` 메소드를 사용할 수 있습니다. 하지만 엔터프라이즈 빈이 아닌 트랜잭션 클라이언트는 JNDI 조회 방법을 사용해야 합니다.

빈 또는 클라이언트가 `UserTransaction` 인터페이스에 대한 참조를 가지고 있는 경우에는 자신의 트랜잭션을 초기화하고 관리할 수 있습니다. 즉, `UserTransaction` 인터페이스 메소드를 사용하여 트랜잭션을 시작하고 커밋이나 롤백할 수 있습니다. `begin()` 메소드를 사용하여 트랜잭션을 시작한 다음 `commit()` 메소드를 사용하여 데이트베이스에 변경 내용을 커밋합니다. 또는 `rollback()` 메소드를 사용하여 트랜잭션 내에서 만들어진 모든 변경사항을 취소하고 트랙잭션 시작 이전 상태로 데이터베이스를 복원합니다. `begin()`과 `commit()` 메소드 사이에 코드를 포함시켜 트랜잭션의 비즈니스를 수행합니다. 다음과 같은 예제가 있습니다.

```
public class NewSessionBean implements SessionBean {
    EJBContext ejbContext;

    public void doSomething(...) {
        javax.transaction.UserTransaction utx;
        javax.sql.DataSource dataSource1;
        javax.sql.DataSource dataSource2;
        java.sql.Connection firstConnection;
        java.sql.Connection secondConnection;
        java.sql.Statement firstStatement;
        java.sql Statement secondStatement;

        java.naming.Context context = new javax.naming.InitialContext();

        dataSource1 = (javax.sql.DataSource)
        context.lookup("java:comp/env/jdbcDatabase1");
```

```
        firstConnection = dataSource1.getConnection();

        firstStatement = firstConnection.createStatement();

        dataSource2 = (javax.sql.DataSource)
        context.lookup("java:comp/env/jdbcDatabase2");
        secondConnection = dataSource2.getConnection();

        secondStatement = secondConnection.createStatement();

        utx = ejbContext.getUserTransaction();

        utx.begin();

        firstStatement.executeQuery(...);
        firstStatement.executeUpdate(...);
        secondStatement.executeQuery(...);
        secondStatement.executeUpdate(...);

        utx.commit();

        firstStatement.close;
        secondStatement.close
        firstConnection.close();
        secondConnection.close();
    }
    ...
```

트랜잭션 예외 처리

엔터프라이즈 빈은 트랜잭션을 처리하는 동안 오류가 생길 경우 애플리케이션 및 시스템 레벨의 예외를 발생시킬 수 있습니다. 애플리케이션 레벨의 예외는 비즈니스 로직의 오류에서 기인합니다. 호출 애플리케이션에서 이를 처리해야 합니다. 런타임 오류와 같은 시스템 레벨의 예외는 애플리케이션 자체를 넘어서며 애플리케이션이나 엔터프라이즈 빈 또는 빈 컨테이너에 의해 처리될 수 있습니다.

엔터프라이즈 빈은 `throws` 절에서 자신의 `Home` 인터페이스와 `Remote` 인터페이스에서 애플리케이션 레벨의 예외 및 시스템 레벨의 예외를 선언합니다. 빈 메소드를 호출할 경우에는 클라이언트 애플리케이션의 try/catch 블록에서 선택 표시된 예외를 확인해야 합니다.

시스템 레벨의 예외

엔터프라이즈 빈은 시스템 레벨의 예외(일반적으로 `java.ejb.EJBException` 이지만 `java.rmi.RemoteException`도 가능합니다)를 발생시켜 예상치 못한 시스템 레벨의 오류를 표시합니다. 예를 들어, 엔터프라이즈 빈은 데이터베이스를 연결할 수 없는 경우에 예외를 발생시킵니다. `java.ejb.EJBException`

은 런타임 예외이며 빈의 business 메소드의 throws 절에 반드시 나열될 필요는 없습니다.

시스템 레벨의 예외는 일반적으로 트랜잭션이 롤백되는 것을 요구합니다. 빈에 의해 관리되는 컨테이너는 경우에 따라 롤백을 실행합니다. 하지만 특별히 트랜잭션이 빈에 의해 관리되는 경우, 클라이언트는 트랜잭션을 롤백해야 합니다.

애플리케이션 레벨의 예외

빈은 애플리케이션 레벨의 예외를 발생시켜 애플리케이션 특정의 오류 조건을 표시합니다. 이는 비즈니스 로직 오류일뿐 시스템 문제는 아닙니다. 애플리케이션 레벨의 예외는 java.ejb.EJBException 이외의 예외입니다. 애플리케이션 레벨의 예외는 선택 표시된 예외인데 이는 이러한 예외를 잠정적으로 발생시킬 수 있는 메소드를 호출 할 때 이들 예외를 확인해야함을 의미합니다.

빈의 business 메소드는 애플리케이션 예외를 사용하여 수용 불가능한 입력 값이나 수용 가능한 한계를 초과하는 양과 같은 비정상적인 애플리케이션 조건에 대해 보고합니다. 예를 들어, 계좌 잔고의 차변에 기입하는 빈 메소드는 애플리케이션 예외를 발생시켜 계좌 잔고가 특정 차변 기입 작업을 허용할 정도로 충분하지 않다는 것을 보고할 수 있습니다. 클라이언트는 경우에 따라 전체 트랜잭션을 롤백하지 않고 이러한 애플리케이션 레벨의 오류로부터 복구할 수 있습니다.

애플리케이션 또는 호출 프로그램은 이전에 발생된 동일한 예외를 다시 불러 호출 프로그램이 정확한 문제의 원인을 알 수 있게 합니다. 애플리케이션 레벨의 예외가 발생할 경우 엔터프라이즈 빈 인스턴스는 클라이언트의 트랜잭션을 자동으로 롤백할 수 없습니다. 클라이언트는이제 오류 메시지를 평가할 수 있는 지식과 기회를 가지고 이러한 상황을 교정하는 데 필요한 조치를 취하여 트랜잭션을 복구합니다. 또는 트랜잭션을 취소할 수 있습니다.

애플리케이션 예외 처리

애플리케이션 레벨의 예외는 비즈니스 로직 오류를 보고하기 때문에 클라이언트는 이러한 예외를 처리해야 합니다. 이러한 예외는 트랜잭션 롤백을 필요로 할 수도 있지만 롤백을 위한 트랜잭션을 자동으로 표시하지는 않습니다. 클라이언트는 트랜잭션을 자주 취소하고 롤백해야 하지만 트랜잭션을 재시도할 수 있습니다.

빈 provider로서 사용자는 클라이언트가 트랜잭션을 계속할 경우 빈의 상태가 데이터 무결성의 손실이 없도록 해야 합니다. 이렇게 할 수 없을 경우에는 롤백을 위한 트랜잭션을 표시해야 합니다.

트랜잭션 롤백

클라이언트가 애플리케이션 예외를 가질 경우에는 먼저 현재 트랜잭션이 롤백만을 위해 표시되어 있었는지 확인합니다 . 예를 들어 , 클라이언트는 javax.transaction.TransactionRolledbackException을 받을 수도 있습니다. 이 예외는 helper 엔터프라이즈 빈이 실패했음과 트랜잭션이 취소되었거나 " 롤백 전용 " 표시되었음을 나타냅니다 . 일반적으로 클라이언트는 엔터프라이즈 빈이 작동하는 트랜잭션 컨텍스트를 알지 못합니다 . 빈은 호출 프로그램의 컨텍스트에서 작동하거나 호출 프로그램의 컨텍스트와 별도인 자신의 트랜잭션 컨텍스트에서 작동했을 수도 있습니다 .

엔터프라이즈 빈이 호출 프로그램과 동일한 트랜잭션 컨텍스트에서 작동했다면, 빈 자체나 빈의 컨테이너에서 이미 롤백을 위한 트랜잭션을 표시한 상태입니다. EJB 컨테이너가 롤백을 위한 트랜잭션을 표시할 경우에는 클라이언트는 트랜잭션 상의 모든 작업을 중지해야 합니다. 선언(declarative) 트랜잭션을 사용하는 클라이언트는 일반적으로 javax.transaction.TransactionRolledbackException 같은 적절한 예외를 가집니다. 선언(declarative) 트랜잭션은 컨테이너가 트랜잭션 세부 사항을 관리하는 트랜잭션입니다.

자체가 엔터프라이즈 빈인 클라이언트는 javax.ejbEJBContext.getRollbackOnly() 메소드를 호출하여 자신의 트랜잭션이 롤백을 위해 표시되어 있었는지 알아 봅니다.

빈에 의해 관리되는 트랜잭션(클라이언트에 의해 명시적으로 관리되는 트랜잭션)의 경우, 클라이언트는 java.transaction.userTransaction 인터페이스로부터 rollback() 메소드를 호출하여 트랜잭션을 롤백합니다.

트랜잭션을 계속하기 위한 옵션

트랜잭션이 롤백 표시되지 않았을 경우 클라이언트는 다음과 같은 옵션을 가집니다 .

- 트랜잭션을 롤백합니다.

  클라이언트가 롤백 표시가 없는 트랜잭션을 위해 선택 표시된 예외를 받은 경우, 가장 안전한 방법은 트랜잭션을 롤백하는 것입니다. 롤백 전용으로 트랜잭션을 표시하거나 또는 클라이언트가 사실상 트랜잭션을 시작하였을 경우에는 rollback() 메소드를 호출함으로써 클라이언트는 트랜잭션을 롤백할 수 있습니다.

- 선택 표시된 예외를 발생시키거나 원래 예외를 다시 발생시킴으로써 책임을 전가합니다.

  클라이언트는 또한 자신의 선택 표시된 예외를 발생시키거나 원래 예외를 다시 발생시킬 수 있습니다. 예외를 발생시킴으로써 클라이언트는 트랜잭션 체인 훨씬 위에 있는 다른 프로그램으로 하여금 트랜잭션 취소 여부를 결정하도록 합니다. 하지만 일반적으로 문제 발생에 가장 가까운 코드나 프로그램이 트랜잭션을 계속할 것인지 여부를 결정하도록 하고 있습니다.

- 트랜잭션을 재시도하고 계속합니다. 트랜잭션의 부분을 재시도할 수도 있습니다.

  클라이언트는 트랜잭션을 계속할 수 있습니다. 클라이언트는 예외 메시지를 평가하고 다른 매개변수로 메소드를 다시 호출하여 성공할 수 있을 지 여부를 결정할 수 있습니다. 하지만 트랜잭션을 다시 시도한다는 것은 잠재적인 위험을 내포합니다. 코드에서는 엔터프라이즈 빈이 적절하게 자신의 상태를 클린업했는 지 여부를 알 수 없습니다.

  하지만 상태 없는(stateless) 세션 빈을 호출하는 클라이언트는 발생된 예외에서 기인하는 문제를 알 수 있는 경우 트랜잭션을 재시도할 수 있습니다. 호출된 빈이 상태 없는 빈이므로 부적절한 상태에 대해 염려할 필요는 없습니다.

Borland AppServer를 사용할 경우, 트랜잭션과 Borland 컨테이너에 대한 자세한 내용은 Borland AppServer's Enterprise JavaBeans Programmer Guide의 "Transaction Management" 장을 참조하십시오.

Part

II

Distributed Application Developer's Guide

Chapter

21

분산 애플리케이션 개발

이것은 JBuilder 기업용 버전의 기능입니다.

JBuilder는 분산 애플리케이션 개발을 적극 지원합니다. JBuilder 개발 환경은 분산 애플리케이션의 생성 방법을 극히 간소화 하여 다계층 애플리케이션에 필요한 파일들을 생성합니다. 일단 이러한 파일들이 생성되면 생성된 코드에 필요한 비즈니스 로직을 추가할 수 있습니다.

이 안내서는 분산 애플리케이션과 분산 애플리케이션 용어에 친숙한 개발자를 위해 만들어 졌습니다. 분산 애플리케이션과 분산 애플리케이션 용어에 친숙하지 않다면 분산 애플리케이션의 개념과 기술에 대해 설명하고 있는 다른 설명서를 보는 것이 좋습니다. Borland 웹 사이트 http://www.borland.com/visibroker/books/에서 추천 도서를 보실 수 있습니다.

본 안내서에서는 다음과 같은 기술을 다룹니다.

- CORBA(Common Object Request Broker Architecture)

  CORBA는 분산 컴퓨팅 환경을 위한 개방형 표준 기반 솔루션입니다. OMG(Object Management Group)에서는 CORBA에 대한 사양을 개발하고 ORB(Object Request Brokers) 간의 표준 통신 프로토콜을 지정했습니다. 해당 사양을 보려면 http://www.omg.org/의 OMG 웹 사이트를 방문하십시오. 이 사이트는 CORBA를 처음 사용하는 이들을 위한 정보도 제공합니다.

서버와 클라이언트를 모든 프로그램 언어로 쓸 수 있다는 것이 CORBA의 주요 이점입니다. 이는 IDL(Interface Definition Language)로 객체가 정의되어 있고 객체, 클라이언트 그리고 서버 간의 통신이 ORB(Object Request Brokers)를 통해 처리되기 때문에 가능합니다.

JBuilder는 *VisiBroker for Java*와 *OrbixWeb*과 관련된 CORBA 애플리케이션 개발을 적극 지원합니다. JBuilder 개발 환경은 IDL 모듈을 사용하고 필요한 모든 인터페이스 파일, 클라이언트 스텁 및 서버 뼈대를 생성하므로 CORBA 애플리케이션을 매우 쉽게 만들 수 있습니다. 마법사는 CORBA 서버 및 클라이언트 생성을 돕습니다. 그러면 생성된 코드에 필요한 비즈니스 로직을 추가할 수 있습니다.

JBuilder와 VisiBroker(OrbixWeb을 사용하기 위해 쉽게 수정될 수도 있음)를 사용한 CORBA 애플리케이션 생성 단계에 대해 개괄적으로 설명하는 자습서는 Chapter 23 " JBuilder의 CORBA 기반 분산 애플리케이션 살펴보기"를 참조하십시오. Borland AppServer 설치 디렉토리 `Borland/AppServer/examples/vbj` 내부의 IDL에서 생성된 인터페이스가 있는 CORBA 샘플 애플리케이션을 볼 수 있습니다.

- RMI(Remote Method Invocation)

  RMI(Remote Method Invocation)를 사용하면 분산된 Java 간 애플리케이션을 만들 수 있으므로, 다른 호스트 상에서도 다른 Java 가상 머신이 원격 Java 객체의 메소드를 호출할 수 있습니다. Java 프로그램은 원격 객체에 대한 참조를 얻고 나면 RMI에서 제공하는 부트스트랩(bootstrap) 이름 지정 서비스에서 원격 객체를 조회하거나 인수 또는 반환값으로 참조를 수신하여 원격 객체를 호출할 수 있습니다. 클라이언트는 서버에서 원격 객체를 호출할 수 있습니다. 이 서버는 동시에 다른 원격 객체의 클라이언트가 되기도 합니다. RMI는 순수 객체 지향 다형성을 지원하므로 객체 직렬화를 사용하여 매개변수를 마샬링(marshaling)하거나 마샬링 해제하며 타입을 자르지 않습니다.

  "Java Remote Method Invocation - Distributed Computing for Java (Java용 분배 컴퓨팅 환경) (백서)" 조항은 Java RMI의 사용 방법을 보다 잘 이해할 수 있도록 해줍니다. 위 조항이 있는 페이지에 액세스 하려면 http://java.sun.com/marketing/collateral/javarmi.html로 브라우저를 지정합니다.

  RMI 애플리케이션 생성을 위해 JBuilder를 사용하는 자습서에 대해서는 Chapter 25 " JBuilder의 Java RMI 기반분산 애플리케이션 살펴보기"을 참조하십시오. JBuilder는 또한 Java RMI을 사용하는 여러 샘플 애플리케이션을 제공합니다. Chapter 25 " JBuilder의 Java RMI 기반분산 애플리케이션 살펴보기"는 JBuilder 설치 디렉토리 `samples/RMI`에 있습니다. 또 다른 샘플은 JBuilder 설치 디렉토리 `samples/DataExpress/StreamableDataSets`에 있습니다.

- Java에서의 인터페이스 정의

  VisiBroker는 Java환경에서의 CORBA 개발을 용이하게 만드는 두 가지 컴파일러와 결합합니다. 이 컴파일러들은 자바 언어를 사용하여 CORBA 인터페이스를 정의할 수 있게 해줍니다.

  - `java2iiop` 컴파일러를 사용하면 전체적인 자바 환경에서 작업할 수 있습니다. `java2iiop` 컴파일러는 Java 인터페이스를 사용하고 IIOP 호환 스텁 및 뼈대를 생성합니다. 이 컴파일러를 통해 확장 가능한 구조를 사용하여 값에 따라 Java 객체를 직렬화 할 수 있습니다.
  - `java2idl` 컴파일러는 Java 코드를 IDL로 변환하므로 원하는 프로그램 언어에서 클라이언트 스텁을 생성할 수 있습니다. 뿐만 아니라 이 컴파일러는 사용자의 Java 인터페이스를 IDL로 매핑하므로 사용자가 동일한 IDL을 지원하는 다른 프로그램 언어로 Java 객체를 다시 구현할 수 있습니다.

  JBuilder에서의 컴파일러 사용에 대한 자세한 내용은 Chapter 24 "Java에서의 인터페이스 정의"를 참조하십시오.

다계층 애플리케이션을 구축하는 데 도움을 줄 그 밖의 주제는 다음과 같습니다.

- Chapter 22 "CORBA 애플리케이션용 JBuilder 설정"

  JBuilder의 분산 애플리케이션 기능을 가능하게 해줄 VisiBroker 또는 OrbixWeb를 사용하여 JBuilder를 설정하는 데 필요한 단계를 설명합니다.

- Chapter 26 "분산 애플리케이션 디버깅"

  JBuilder의 디버깅 서버를 사용하여 원격으로 애플리케이션을 디버깅하는 방법을 설명합니다.

- Chapter 27 "원격 디버깅 자습서"

  원격 컴퓨터에서 이미 실행되고 있는 프로그램에 애플리케이션을 추가하고 디버깅하는 과정을 자세히 설명합니다.

분산 애플리케이션에 관해 업데이트된 새로운 설명서와 자습서는 Borland 웹 사이트 http://www.borland.com/techpubs/jbuilder/의 JBuilder Technical Publications 페이지에서 주기적으로 확인하십시오. JBuilder 도움말 메뉴에서 사용할 수 있는 온라인 도움말 시스템은 본 설명서가 발행된 시기에 이용한 항목보다 좀더 업데이트된 정보를 갖고 있습니다.

분산 애플리케이션에 대해 궁금한 내용은 http://www.borland.com/newsgroups/의 CORBA-RMI newsgroup, borland.public.jbuilder.corba-rmi로 문의하십시오.

Chapter

22

CORBA 애플리케이션용 JBuilder 설정

이것은 JBuilder 기업용 버전의 기능입니다.

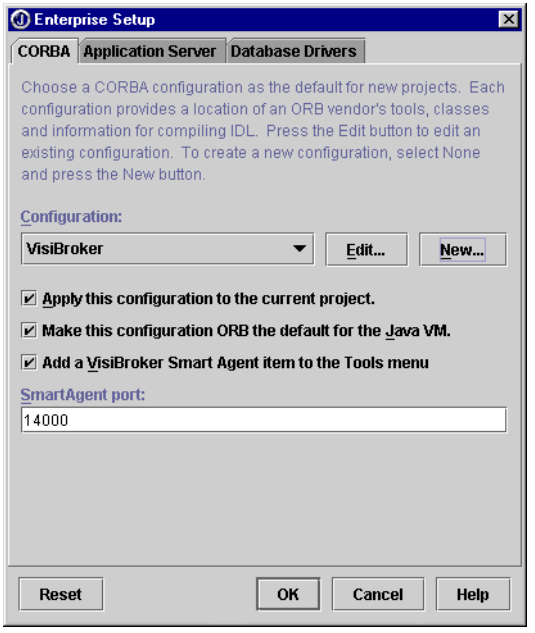
이 장에서는 ORB(VisiBroker or OrbixWeb)를 사용하여 JBuilder를 설정하는 방법에 대해 설명함으로써 사용자가 CORBA 애플리케이션을 생성, 실행 및 배포할 수 있도록 합니다. 본 설명서에 수록된 자습서는 JBuilder 기업용 버전에 포함되어 있는 VisiBroker ORB를 사용합니다. ORB의 특정 기능에 대해서는 OrbixWeb 설명서를 참조하십시오.

다음과 같은 방법으로 시스템에 다음 항목들을 설치합니다.

1. JBuilder 기업용 에디션을 설치합니다.
2. VisiBroker나 OrbixWeb과 같은 ORB를 설치합니다.

   VisiBroker는 JBuilder 기업용 CD에 포함되어 있습니다. VisiBroker는 Borland AppServer의 한 부분입니다.
3. JBuilder에서 Tools|Enterprise Setup을 실행시켜 Enterprise Setup 대화 상자를 표시합니다. CORBA 탭을 선택합니다. 이 대화 상자에 매개변수가 있기 때문에 JBuilder가 ORB를 볼 수 있습니다.

4 구성 드롭다운 리스트에서 구성을 선택할 수 있습니다. VisiBroker와 OrbixWeb을 먼저 선택합니다. Enterprise Setup 대화 상자는 다음과 같습니다.



5 VisiBroker나 OrbixWeb을 사용하는 경우에는 Edit 버튼을 클릭하여 Edit Configuration 대화 상자를 표시합니다. New 버튼을 클릭하여 새로운 구성을 정의합니다.

6 Edit Configuration 대화 상자에서 Path For ORB Tools 필드 옆에 있는 생략 버튼을 클릭하여 JBuilder를 ORB 툴에 액세스시킵니다. VisiBroker를 사용할 경우 `osagent.exe` 파일이 들어 있는 디렉토리를 지정합니다. 그 디렉토리는 `Borland/AppServer/bin`입니다.

7 Library For Projects 필드옆에 있는 생략 버튼을 클릭하여 ORB용 라이브러리를 프로젝트에 추가합니다. 생략 버튼은 Select A Different Library 대화 상자를 표시합니다. 라이브러리는 생성된 스텁과 뼈대를 컴파일하고 애플리케이션을 실행하는 데 필요합니다.

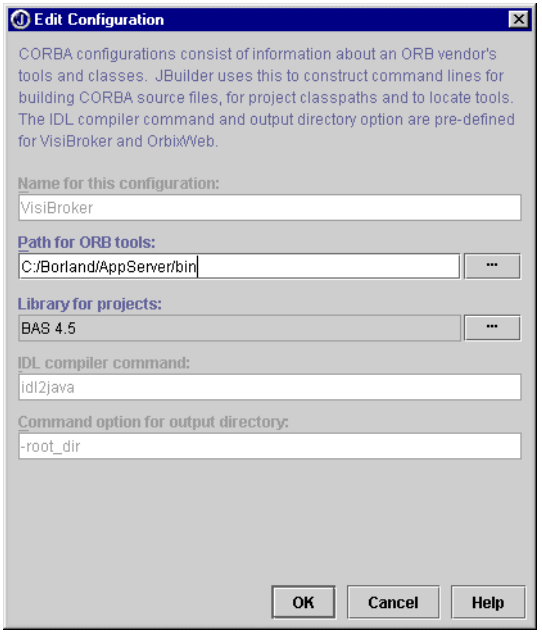
   1 라이브러리를 추가하려면 Select A Different Library 대화 상자에 있는 New를 클릭하여 New Library 마법사를 표시합니다.

   2 New Library 마법사에서 Name 필드(또는 예를 들어 VisiBroker나 OrbixWeb)에 있는 새 라이브러리 이름을 입력하십시오.

   3 라이브러리 위치를 JBuilder 디렉토리, 프로젝트 디렉토리 또는 사용자의 홈 디렉토리 중에서 선택합니다.

   4 Add를 클릭하여 라이브러리(JAR) 파일을 찾습니다. VisiBroker를 사용할 경우 라이브러리 파일 `vbjorb.jar`는 `Borland/AppServer/lib` 디렉토리에 있습니다.

5 Edit Configuration 대화 상자가 다시 나올 때까지 OK를 클릭합니다. Edit Configuration 대화는 다음과 같습니다.



6 Edit Configuration 대화 상자를 닫으려면 OK를 클릭합니다.

8 Enterprise Setup 대화 상자에서 기본 프로젝트와 같은 ORB 구성으로 기존 프로젝트가 설정되어야 할 경우 Apply This Configuration To The Current Project 옵션을 선택된 상태로 둡니다.

9 Make This Configuration's ORB The Default For The Java VM 옵션을 선택된 상태로 두고 위의 ORB를 Java VM용 기본 ORB로 사용합니다. 오류 메시지를 받았을 경우에는 JBuilder 설치의 `jdk1.3/jre/lib` 디렉토리에 있는 `orb.properties` 파일을 수정해서는 안됩니다. 사용자는 수동으로 `orb.properties` 파일을 생성하거나 편집하여 다음 정보를 지정할 수 있습니다. 샘플 파일은 VisiBroker ORB를 참조합니다.

```
# Make VisiBroker for Java the default ORB
org.omg.CORBA.ORBClass=com.borland.vbroker.orb.ORB
org.omg.CORBA.ORBSingletonClass=com.borland.vbroker.orb.ORB
```

10 VisiBroker의 경우 Add A VisiBroker SmartAgent Item To The Tools Menu 옵션을 선택하여 개발 기간 동안 SmartAgent가 JBuilder 서비스로 실행되도록 합니다. SmartAgent가 필요 없는 경우에는 이 옵션을 사용할 수 없습니다.

11 포트를 선택하여 SmartAgent를 실행합니다. 14000개의 기본 포트가 대부분의 시스템에서 작동됩니다.

12 OK를 클릭합니다. 이러한 설정은 `orb.properties` 파일에 기록됩니다.

이제 JBuilder CORBA를 사용하기 위한 시스템 설정이 모두 끝났습니다. CORBA 애플리케이션 생성하기 위한 기능들을 사용한 자습서는 Chapter 23 " JBuilder의 CORBA 기반 분산 애플리케이션 살펴보기"를 참조하십시오. JBuilder의 VisiBroker 기능을 사용하는 방법은 Chapter 24 "Java에서의 인터페이스 정의"를 참조하십시오.

Chapter

23

JBuilder의 CORBA 기반 분산 애플리케이션 살펴보기

이것은 JBuilder 기업용 버전의 기능입니다.

이 장에서는 JBuilder, VisiBroker, CORBA(Common Object Request Broker Architecture)를 사용한 분산 애플리케이션의 생성에 대해 설명합니다.

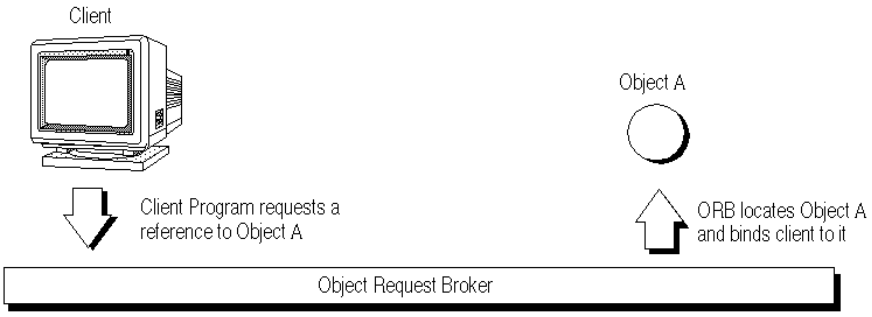
CORBA란?

CORBA(Common Object Request Broker Architecture)를 통해 분산 애플리케이션은 작성된 언어 또는 이러한 애플리케이션의 위치에 관계 없이 상호 운용(애플리케이션 간 통신)이 가능합니다.

CORBA 사양은 분산 객체 애플리케이션 개발의 복잡성 및 높은 비용을 해소하기 위해 Object Management Group에서 채택한 것입니다. CORBA는 애플리케이션 간에 재사용하고 공유할 수 있는 소프트웨어 컴포넌트 생성을 위해 객체 지향 접근 방법을 사용합니다. 각 객체는 내부 작업의 상세 사항을 캡슐화하여 애플리케이션의 복잡성을 줄이는 잘 정의된 인터페이스를 표시합니다. 객체를 일단 구현하여 테스트하면 계속 반복해서 사용할 수 있기 때문에 애플리케이션 개발 비용이 줄어듭니다.

Figure 23.1의 ORB(Object Request Broker)는 클라이언트 애플리케이션을 사용하려는 객체와 연결합니다. 클라이언트 프로그램은 통신하는 객체가 동일한 컴퓨터에 구현되어 있는지 또는 네트워크 어딘가의 원격 컴퓨터에 구현되어 있는지 알 필요가 없습니다. 클라이언트 프로그램은 개체의 이름을 알고 객체의 인터페이스 사용 방법만 이해하면 됩니다. ORB는 객체 위치 지정, 요청 라우팅, 결과 반환에 관한 상세 사항을 관리합니다.

Figure 23.1객체에서 작동하는 클라이언트 프로그램

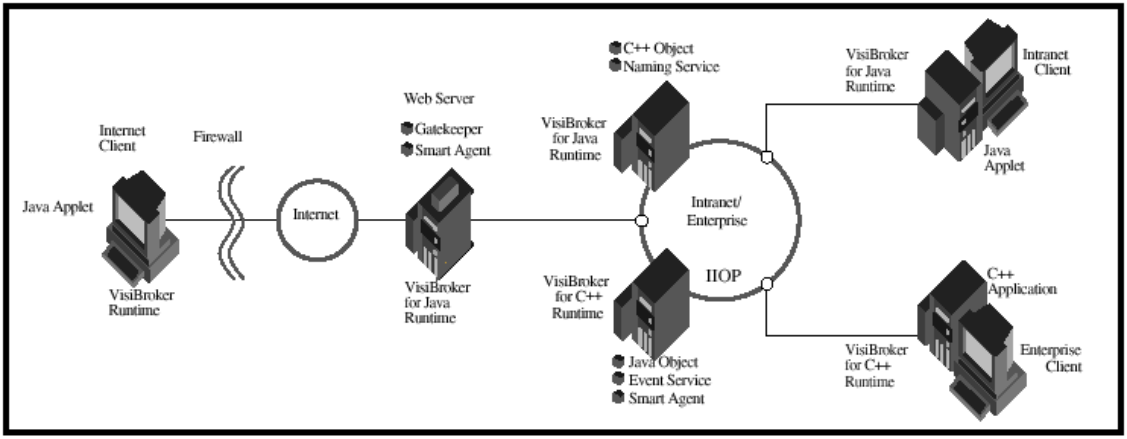


참고 ORB 자체는 별도의 프로세스가 아닙니다. ORB는 최종 사용자의 애플리케이션 내에서 통합되는 라이브러리 및 네트워크 자원의 집합으로 클라이언트 애플리케이션이 객체를 찾아 사용할 수 있도록 합니다.

VisiBroker란?

Java용 VisiBroker는 완벽한 CORBA ORB 런타임을 제공하며, 개방형의 유연하고 상호 운용 가능한 분산형 Java 애플리케이션의 개발, 배포, 관리를 위한 지원 개발 환경을 제공합니다. Java용 VisiBroker로 개발한 객체는 인터넷 또는 로컬 인트라넷을 통해 분산 객체 사이에서 통신하기 위한 OMG의 IIOP(Internet Inter-ORB Protocol) 표준을 사용하여 통신하는 웹 기반 애플리케이션에서 쉽게 액세스할 수 있습니다. VisiBroker는 IIOP의 원시 구현이 적용되어 높은 성능과 상호 운용성을 보장합니다.

Figure 23.2 VisiBroker 아키텍처



JBuilder와 VisiBroker가 연동하는 방법

JBuilder는 일련의 마법사 및 기타 툴을 제공하여 CORBA 애플리케이션의 개발을 수월하게 합니다.

이 장에서는 Java 개발 툴용 JBuilder 및 VisiBroker 일부를 숙지하도록 도와주는 자습서를 다루며 Java용 JBuilder 및 VisiBroker를 사용하는 분산형 객체 기반 애플리케이션 개발에 관해 설명합니다. 23- 4페이지의 "자습서: JBuilder에서 CORBA 애플리케이션 생성"에 나타나 있듯이 이 프로세스의 각 단계를 설명하는 데 동일한 애플리케이션이 사용됩니다.

JBuilder 및 VisiBroker를 사용하여 분산 애플리케이션을 개발하려면 먼저 애플리케이션이 필요로 하는 객체를 확인해야 합니다. 그런 다음 일반적으로 다음 단계들을 따릅니다.

1 IDL(Interface Definition Language) 파일을 생성하는 IDL 마법사를 사용하여 각 객체에 대한 사양을 작성합니다.

IDL은 객체가 제공하는 연산 및 연산 호출 방법을 지정하기 위해 구현자가 사용하는 언어입니다. 이 예제에서 보면, IDL에서 `Account` 인터페이스를 `balance()` 메소드로, `AccountManager` 인터페이스를 `open()` 메소드로 정의합니다.

2 IDL 컴파일러를 사용하여 클라이언트 스텁 코드 및 서버 POA(Portable Object Adapter) 종속 코드를 생성합니다. POA에 관한 자세한 내용은 페이지 3- 13의 "뱒OA란?" 을 참조하십시오.

`idl2java` 컴파일러를 사용하면 클라이언트측 스텁(`Account` 및 `AccountManager` 객체의 메소드에 인터페이스 제공)과 서버측 클래스(원격 객체의 구현을 위한 클래스 제공)가 제공됩니다.

3 클라이언트 프로그램 코드를 작성합니다.

클라이언트 프로그램의 구현을 완료하려면 ORB를 초기화하고, `Account` 및 `AccountManager` 객체로 바인딩한 다음 이들 객체의 메소드를 호출하고 잔고를 출력합니다.

4 서버 객체 코드를 작성합니다.

서버 객체 코드의 구현을 완료하려면 `AccountPOA` 및 `AccountManagerPOA` 클래스에서 파생되어 `interface`의 메소드 구현을 제공하고 서버의 `main` 루틴을 구현해야 합니다.

5 클라이언트 및 서버 코드를 컴파일합니다.

클라이언트 프로그램을 생성하려면 클라이언트 프로그램 코드를 클라이언트 스텁으로 컴파일합니다. `Account` 서버를 생성하려면 서버 객체 코드를 서버 종속 코드로 컴파일합니다.

6 서버를 시작합니다.

7 클라이언트 프로그램을 실행합니다.

분산 애플리케이션 생성을 위한 JBuilder 및 VisiBroker 사용에 관한 자세한 내용은 `Borland/AppServer/doc/books/vbj/vbj45` 디렉토리의 *VisiBroker for Java Programmer's Guide* 및 *VisiBroker for Java Reference Guide*를 참조하십시오.

CORBA(Common Object Request Broker Architecture)에 관한 자세한 내용은 다음과 같은 몇 가지 유용한 링크를 참조하십시오.

- http://www.borland.com/books Borland사의 제품 및 기술에 관한 서적
- http://www.omg.org/news/whitepapers/wpjava.htm - *"A White Paper: Java, RMI, and CORBA"*
- http://www.omg.org/gettingstarted/specintro.htm - OMG의 사양에 대한 소개 중 *CORBA/IIOP Specification*
- http://www.omg.org/gettingstarted/corbafaq.htmhttp://www.omg.org/gettingstarted/corbafaq.htm - *CORBA Basics*

자습서: JBuilder에서 CORBA 애플리케이션 생성

이 자습서에서 은행 계좌의 잔고를 질의할 수 있는 간단한 예제 클라이언트 애플리케이션을 생성할 수 있습니다. 이 자습서는 간단한 CORBA 기반 분산 애플리케이션을 실행하고 배포하는 데 필요한 기본 파일의 생성 단계를 설명합니다.

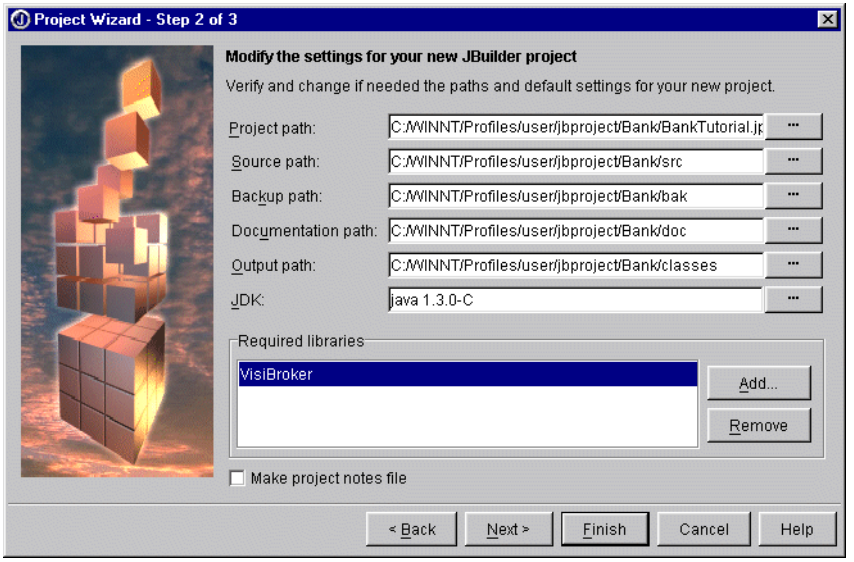
이 예제에서 다음과 같은 방법을 배우게 됩니다.

- IDL로 객체 인터페이스 정의
- 클라이언트 스텁 루틴 및 서버 종속 코드 생성
- 클라이언트 프로그램 구현: ORB를 초기화하고 `AccountManager` 객체로 바인딩하여 잔고를 알아내서 출력하고 예외 사항을 처리합니다.
- 서버 객체 구현 방법: ORB를 초기화하고 POA(Portable Object Adapter)를 생성한 다음 계좌 관리자 종속 객체를 생성하고 종속 객체를 활성화합니다. POA 관리자 및 POA를 활성화하고 요청 수신을 준비합니다.
- 예제 만들기
- Java 클라이언트로 예제 실행: Smart Agent, `AccountManager` 서버를 시작하고 클라이언트 프로그램을 실행합니다.

1 단계: 프로젝트 설정

이 자습서는 사용자가 JBuilder 개발 환경과 친숙하다고 가정합니다. JBuilder를 처음 사용하는 경우 *JBuilder를 이용한 애플리케이션 구축*의 "JBuilder 환경"에서 개요를 참조하십시오.

1. Chapter 22 "CORBA 애플리케이션용 JBuilder 설정"에 설명되어 있는 대로 서로 "마주보도록" JBuilder와 VisiBroker를 설정합니다.
2. File|New Project를 선택하여 새 프로젝트를 생성합니다.
3. Project Name 필드의 이름을 `BankTutorial`로 변경합니다.
4. Project Directory 필드의 이름을 `Bank`로 변경합니다. 마법사의 2 단계로 가려면 Next를 클릭합니다.
5. VisiBroker가 2 단계의 Required Libraries 목록에 나열되어 있지 않으면 Add 버튼을 클릭합니다. Select One or More Libraries 대화상자에서 VisiBroker를 선택하고 OK를 클릭합니다. Application 마법사의 2 단계는 다음과 같이 보입니다.



6. 또는 Next 버튼을 클릭하여 Title, Author, Company, Description 필드를 입력할 수 있는 Project Wizard의 3 단계로 갑니다.
7. Finish를 클릭합니다.
8. VisiBroker가 기본 IDL 컴파일러로 설정되어 있는지 확인합니다. 다음과 같은 방법으로 이 작업을 수행합니다.
   1. Project|Project Properties를 선택하여 Project Properties 대화상자를 표시합니다.
   2. Build 탭을 선택합니다. Build 페이지의 IDL 탭을 선택합니다.
   3. VisiBroker가 IDL Compiler 드롭다운 목록에서 선택되어 있어야 합니다.
   4. 대화 상자를 닫으려면 OK를 클릭합니다.

BankTutorial 프로젝트가 프로젝트 창에 나타납니다. 다음으로 계정 인터페이스를 IDL로 작성하여 객체 인터페이스를 정의합니다.

2 단계: IDL로 CORBA 객체용 인터페이스 정의

CORBA 애플리케이션을 생성하는 첫 번째 단계는 OMG의 **IDL(Interface Definition Language)**을 사용하여 모든 객체 및 인터페이스를 지정하는 것입니다. IDL은 C++과 유사한 구문을 가지고 있어 모듈, 인터페이스 데이터 구조 등을 정의하는 데 사용할 수 있습니다. IDL은 여러 가지 프로그래밍 언어로 매핑할 수 있습니다. Java용 IDL 매핑은 *VisiBroker for Java Reference Guide*에 설명되어 있습니다.

이 항목에서는 이 예제에서 `BankTutorial.idl` 파일을 생성하는 방법을 보여줍니다. `Account` 인터페이스는 현재 잔고를 알 수 있는 방법 한 가지를 제공합니다. `AccountManager` 인터페이스는 계좌가 들어 있지 않은 경우 사용자의 계좌를 만듭니다.

다음과 같은 방법으로 IDL 파일을 생성하고 객체 인터페이스를 정의합니다.

1. File|New를 선택한 다음 객체 갤러리의 CORBA 페이지에서 Sample IDL을 선택합니다.
2. File Name 필드에 `BankTutorial.idl`을 입력합니다. OK를 클릭합니다. 예제 IDL 파일이 생성되어 프로젝트에 추가됩니다.
3. 예제 코드를 제거하고 다음 IDL 코드를 삽입합니다. IDL 구문 및 의미에 관한 내용을 다운로드하려면 브라우저를 http://www.omg.org/technology/documents/formal/corba_omg_idl_text_file.htm으로 이동하십시오.

```
module Bank {
  interface Account {
    float balance();
  };
  interface AccountManager {
    Account open(in string name);
  };
};
```

4. File|Save All을 선택합니다. 컴파일하기 전에 IDL 파일을 저장해야 합니다.

3 단계에서 VisiBroker IDL 컴파일러를 사용하여 IDL 사양의 스텁 루틴 및 종속 코드를 생성합니다.

3 단계: 클라이언트 스텁 및 서버 종속 코드 생성

이 단계에서는 `idl2java` 컴파일러라고도 하는 VisiBroker IDL 컴파일러를 사용하여 IDL 사양의 스텁 루틴 및 종속 코드를 생성합니다. 스텁 루틴은 클라이언트 프로그램이 객체의 연산을 호출하는 데 사용됩니다. 종속 코드는 사용자가 작성한 코드와 함께 객체를 구현하는 서버를 생성합니다. 클라이언트 프로그램 및 서버 객체에 대한 코드는 클라이언트 및 서버의 실행 가능한 클래스를 제공하는 Java 컴파일러에 대한 입력에 사용됩니다.

다음과 같은 방법으로 클라이언트 스텁 및 서버 종속 코드를 생성합니다.

1 프로젝트 창에서 BankTutorial.idl 파일을 마우스 오른쪽 버튼으로 클릭합니다.

2 Make를 선택하여 idl2java 컴파일러를 호출합니다.

파일이 특별한 처리를 필요로 하는 경우 프로젝트 창에서 IDL 파일을 마우스 오른쪽 버튼으로 클릭하여 Properties를 선택함으로써 IDL 속성을 설정할 수 있습니다. 이렇게 하면 Build 페이지의 IDL 탭이 표시되는데 IDL로 정의된 원격 인터페이스 컴파일용 옵션을 지정할 수 있습니다.

생성되는 파일

Java가 파일 당 하나의 공용 인터페이스 또는 클래스를 허용하기 때문에 IDL 파일을 컴파일하면 여러 .java 파일이 생성됩니다. 이러한 파일은 Generated Source라는 이름으로 생성된 디렉토리에 저장되는데 이 이름은 IDL로 지정된 모듈 이름으로서, 생성된 파일이 속하는 패키지입니다. BankTutorial.idl 옆의 확장 glyph를 클릭하여 생성된 파일을 볼 수 있습니다.

다음 파일들이 생성됩니다.

- _AccountManagerStub.java – 클라이언트측의 AccountManager 객체에 대한 스텁 코드
- _AccountStub.java – 클라이언트측의 Account 객체에 대한 스텁 코드
- Account.java – Account 인터페이스 선언
- AccountHelper.java – 유용한 유틸리티 메소드를 정의하는 AccountHelper 클래스 선언
- AccountHolder.java – Account 객체 전달용 홀더를 제공하는 AccountHolder 클래스 선언
- AccountManager.java – AccountManager 인터페이스 선언
- AccountManagerHelper.java – 유용한 유틸리티 메소드를 정의하는 AccountManagerHelper 클래스 선언
- AccountManagerHolder.java – AccountManager 객체 전달용 홀더를 제공하는 AccountManagerHolder 클래스 선언
- AccountManagerOperations.java – AccountManager 인터페이스에 정의되어 있는 메소드 서명을 Bank.idl 파일에 선언
- AccountManagerPOA.java – 서버측에 AccountManager 객체를 구현하기 위한 POA 종속 코드(구현 기반 코드)
- AccountManagerPOATie.java – 타이 메커니즘(tie mechanism)을 사용하여 서버측에 AccountManager 객체를 구현하는 데 사용되는 클래스. 타이 메커니즘(tie mechanism)에 관한 자세한 내용은 *VisiBroker for Java Programmer's Guide*의 9장 "Using the tie mechanism"을 참조하십시오.

- AccountOperations.java – Account 인터페이스에 정의되어 있는 메소드 서명을 Bank.idl 파일에 선언
- AccountPOA.java – 서버측에 Account 객체를 구현하기 위한 POA 종속 코드(구현 기반 코드)
- AccountPOATie.java – 타이 메커니즘을 사용하여 서버측에 Account 객체를 구현하는 데 사용되는 클래스 타이 메커니즘에 관한 자세한 내용은 *VisiBroker for Java Programmer's Guide*의 9장 "Using the tie mechanism"을 참조하십시오.

생성된 파일에 관한 자세한 내용은 *VisiBroker for Java Programmer's Guide*의 4장 "Developing an example application with Visibroker"를 참조하십시오.

다음 단계에서 클라이언트 프로그램을 구현합니다.

4 단계: 클라이언트 구현

CORBA 서버 애플리케이션용 클라이언트는 이 자습서에 설명되어 있는 것처럼 로컬에서 실행되는 다른 애플리케이션이 될 수 있습니다. 또한 웹 브라우저에서 실행되는 HTML 클라이언트가 될 수도 있습니다. JSP(JavaServer Page) 애플릿 또는 서블릿을 생성할 수 있습니다. 또한 InternetBeans를 사용하는 JSP 또는 서블릿을 생성할 수 있습니다. JBuilder를 사용한 웹 애플리케이션 개발에 관한 자세한 내용은 *Web Application Developer's Guide*를 참조하십시오.

idl2java에서 생성된 Bank 패키지에는 은행 클라이언트 구현에 사용되는 많은 클래스가 들어 있습니다. 다음에 이어지는 일련의 단계에서는 Java 클라이언트를 생성하여 ORB를 초기화하고, AccountManager 객체에 바인딩한 다음, 잔고를 알아내어 출력하고 예외 사항을 처리합니다.

다음과 같은 방법으로 클라이언트를 생성합니다.

1 사용자 인터페이스에 사용되는 새 애플리케이션을 생성합니다. 다음과 같은 방법으로 이 작업을 수행합니다.

  1 File|New를 선택한 다음 객체 갤러리의 New 페이지에서 Application을 선택하여 Application 마법사를 시작합니다.

  2 1 단계 및 2 단계에서 모든 기본값을 적용하고 Finish를 클릭합니다.

2 AccountManager 인터페이스를 생성합니다.

  1 Wizards|Use CORBA Interface를 선택합니다. 마법사의 1 단계에 있는 옵션을 선택하지 마십시오.

  2 2 단계로 가려면 Next를 클릭합니다.

  3 IDL 파일로 BankTutorial.idl을 선택합니다.

  4 AccountManagerClientImpl1에서 클래스 이름을 변경하지 마십시오.

5 Interface 목록에 banktutorial.Bank.AccountManager가 선택되어 있어야 합니다.

6 3 단계로 가려면 Next를 클릭합니다.

7 myAccountManager의 기본 필드 이름을 적용합니다. Finish를 클릭합니다. AccountManagerClientImpl1.java 파일이 생성되어 프로젝트 창에 표시됩니다.

3 동일한 단계를 반복하여 Account 인터페이스를 생성합니다.

1 Wizards|Use CORBA Interface를 선택합니다. 마법사의 1 단계에 있는 옵션을 선택하지 마십시오.

2 2 단계로 가려면 Next를 클릭합니다.

3 IDL 파일로 BankTutorial.idl을 선택합니다.

4 클래스 이름을 AccountClientImpl1로 변경합니다.

5 Interface 목록에서 banktutorial.Bank.Account를 선택합니다.

6 3 단계로 가려면 Next를 클릭합니다.

7 myAccount의 기본 필드 이름을 적용합니다. Finish를 클릭합니다. AccountClientImpl1.java 파일이 생성되어 프로젝트 창에 표시됩니다.

4 File|Save All을 선택합니다.

AccountManager 객체에 바인딩

다음 단계는 출하 시의 인터페이스(AccountManager)를 ORB(Object Request Broker)에 연결하는 것입니다. 다음과 같은 방법으로 이 작업을 수행합니다.

1 프로젝트 창에서 Frame1.java를 더블 클릭합니다.

2 Design 탭을 클릭하여 파일을 UI 디자이너에 엽니다.

3 컴포넌트 팔레트의 CORBA 탭을 선택합니다.

4 OrbConnect 객체를 선택합니다.

5 컴포넌트 트리를 클릭하여 OrbConnect 객체를 Frame 파일에 추가합니다.

6 컴포넌트 트리에서 myAccountManager 객체를 선택합니다.

7 Inspector에서 ORBConnect 속성을 선택합니다. 드롭다운 목록에서 orbConnect1을 선택합니다.

8 컴포넌트 트리에서 OrbConnect1 객체를 선택합니다.

9 Inspector에서 initialize 속성을 **true**로 설정합니다.

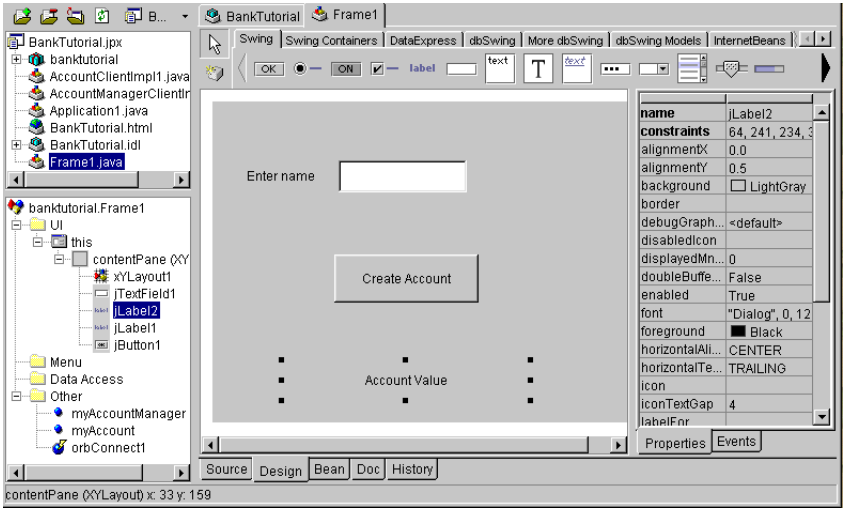
10 File|Save All을 선택합니다.

런타임 시 랩퍼 클래스 바인딩

다음과 같은 방법으로 런타임 시 랩퍼 클래스를 바인딩합니다.

1. 컴포넌트 트리에서 contentPane 객체를 선택합니다.
2. layout 속성을 XYLayout으로 변경합니다.
3. 컴포넌트 팔레트의 Swing 탭에서 JButton 컨트롤을 선택합니다. 프레임의 중앙을 클릭하여 JButton 컨트롤을 디자인 면에 추가합니다. 이 버튼은 새 계정을 생성하여 AccountManager 객체에 바인딩하는 데 사용됩니다.
4. Inspector에서 JButton의 text 속성을 선택합니다. Create account를 입력합니다. 컨트롤의 크기를 조정해야 할 수도 있습니다.
5. 컴포넌트 팔레트의 Swing 탭에서 JLabel 컨트롤을 선택합니다. 디자인 면의 상단 왼쪽에 놓습니다.
6. Inspector에서 text 속성을 Enter name으로 변경합니다. 컨트롤의 크기를 텍스트에 맞도록 조정해야 합니다.
7. Swing 탭에서 JTextField 컨트롤을 선택합니다. 이 컨트롤을 방금 추가한 JLabel 컨트롤 오른쪽에 놓습니다. 이 컨트롤은 새 계정의 이름을 입력하는 데 사용됩니다.
8. Inspector에서 text 속성의 jTextField1을 제거하고 공백으로 남겨 놓습니다.
9. Swing 탭에서 다른 JLabel 컨트롤을 선택합니다. 이 컨트롤을 3 단계에서 추가한 JButton 컨트롤 아래에 놓습니다. 이 레이블은 계좌의 잔고를 표시합니다. 이 콘트롤의 크기를 조정하여 프레임만큼 커지도록 합니다.

10 Inspector에서 text 속성을 Account value로 변경합니다. 또한 수평 정렬 속성을 변경하여 텍스트가 컨트롤의 중앙에 정렬되도록 할 수 있습니다. UI 디자이너는 다음과 같이 표시됩니다.



중요 이 프로그램을 실제 환경에 배포하는 경우 XYLayout은 디자인 전용이므로 사용하지 않아야 합니다. 그 대신 다른 플랫폼의 표시를 조정하는 GridBagLayout과 같은 다른 레이아웃으로 바꾸어야 합니다. 레이아웃을 바꾸려면 Inspector에서 contentPane 객체를 선택하고 layout 속성을 변경합니다.

11 JButton 컨트롤을 선택합니다. Inspector의 Event 탭을 클릭합니다. actionPerformed 이벤트를 선택한 다음 값 상자를 더블 클릭하여 이벤트를 생성합니다. 포커스가 편집기로 이동합니다.

12 jButton1_actionPerformed 이벤트에 다음 코드를 입력합니다.

```
myAccount.setCorbaInterface(myAccountManager.open(jTextField1.getText()));
jLabel2.setText("The account balance is " + myAccount.balance());
```

13 File|Save All을 선택합니다.

클라이언트 클래스, 즉Application1.java 및 Frame1.java에서 다음을 수행합니다.

1 OrbConnect 속성에 지정된 ORB를 사용하여 ORB를 초기화합니다.

2 AccountManager 객체로 바인딩합니다.

클라이언트 프로그램이 balance() 메소드를 호출하도록 하기에 앞서 먼저 bind() 메소드를 사용하여 AccountManager 객체를 구현하는 서버에 대한 연결을 설정해야 합니다. bind() 메소드는 idl2java 컴파일러에서 자동으로 구현됩니다. bind() 메소드는 ORB에서 서버에 대한 연결을 찾아 설정할 것을 요청합니다. 서버를 검색하여 연결 설정에 성공하는 경우 프록시 객체가 생성되어 서버의 AccountManagerPOA 객체를 나타냅니다.

AccountManager 객체에 대한 객체 참조가 클라이언트 프로그램에 반환됩니다.

그런 다음 클라이언트 클래스에서 AccountManager 객체의 open() 메소드를 호출하여 지정된 고객 이름에 대한 Account 객체의 객체 참조를 얻어야 합니다.

3 bind()가 반환하는 객체 참조를 사용하여 계좌의 잔고를 알아냅니다.

클라이언트 프로그램에서 Account 객체 연결을 설정하면 balance() 메소드를 사용하여 잔고를 알아낼 수 있습니다. 클라이언트측의 balance() 메소드는 실제로 데이터를 서버 객체에 요청하고 전송하는 데 필요한 모든 데이터를 수집하는 idl2java 컴파일러가 생성하는 스텁입니다.

4 잔고를 표시합니다.

다음 단계에서는 서버를 구현합니다.

5 단계: 서버 구현

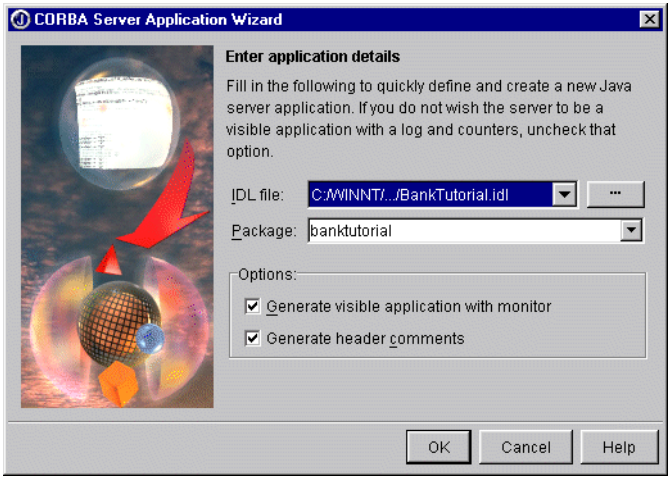
이 단계에서는 서버 구현 방법을 설명합니다. ORB를 초기화하고 POA(Portable Object Adapter)를 생성하며 계좌 관리자 종속 객체를 생성하고 POA 관리자 및 POA를 활성화하며 요청 수신을 준비하게 됩니다. IDL 파일이 컴파일될 때 생성된 BankTutorial 패키지에는 은행 서버 구현에 사용되는 많은 파일들이 들어 있습니다.

다음과 같은 방법으로 서버를 만듭니다.

1 File|New를 선택한 다음 객체 갤러리의 CORBA 페이지에서 CORBA Server Application을 선택합니다. OK를 클릭하여 CORBA Server Application 마법사를 표시합니다.

2 IDL File 필드에 BankTutorial.idl이 선택되어 있어야 합니다.

3 Generate Visible Application With Monitor를 선택하여 서버 모니터를 생성합니다. 마법사는 다음과 같이 나타납니다.



4 OK를 클릭하여 애플리케이션을 생성하고 마법사를 종료합니다.

`banktutorial.Bank.server` 패키지 및 `BankServerApp.java` 파일이 생성됩니다. 또한 `BankAppGenFileList.html` 파일이 프로젝트에 추가되며, 생성된 파일 목록을 포함합니다.

`BankServerApp.java`라는 서버 프로그램에서 다음을 수행합니다.

- Object Request Broker를 초기화합니다.
- 요구 방침에 따라 POA(Portable Object Adaptor)를 생성합니다.
- 계좌 관리자 종속 객체를 생성합니다.
- 종속 객체를 활성화합니다.
- POA 관리자와 POA를 활성화합니다.
- 수신 요청을 대기합니다.

생성된 파일에 관한 설명은 *VisiBroker for Java Programmer's Guide*를 참조하십시오.

POA란?

Portable Object Adaptor 또는 POA는 클라이언트 요청을 인터셉트하여 클라이언트 요청에 부합하는 객체를 확인합니다. 그런 다음 객체가 호출되고 클라이언트로 응답이 반환됩니다. POA는 BOA(Basic Object Adaptor)를 대체한 것으로 서버측에 이식성을 제공하기 위한 목적으로 개발되었습니다.

POA는 다음과 같이 수행하도록 디자인되었습니다.

- 프로그래머가 상이한 ORB 제품에서 이식 가능한 개체 구현을 구축할 수 있도록 합니다.
- 객체에 영구적인 ID를 지원합니다.

- 객체의 명확한 활성화를 지원합니다.
- 단일 종속 객체가 동시에 여러 객체 ID를 지원하도록 합니다.
- POA의 여러 개별 인스턴스가 서버에 존재할 수 있도록 합니다.
- 프로그래밍 작업 및 오버헤드를 최소화하는 투명한 객체를 지원합니다.
- POA 할당 객체 ID를 통해 종속 객체의 암시적인 활성화를 지원합니다.
- 객체 구현에서 객체의 동작에 대해 최대한 책임질 수 있도록 합니다.
- 개별 객체, 객체 ID, 객체 상태 저장 위치, 이전에 특정 ID 값이 사용되었는지 여부, 객체가 존재 또는 존재하지 않도록 중지했는지 등을 설명하는 지속적인 상태를 관리하도록 ORB에 요청할 필요가 없습니다.
- POA에 구현된 정책 정보 객체를 연결하기 위한 확장 메커니즘을 제공합니다.
- 프로그래머가 OMG IDL 컴파일러 또는 DSI 구현에 의해 생성된 정적 뼈대 클래스에서 상속 받는 객체를 구현하도록 설정할 수 있습니다.

POA에 관한 자세한 내용은 `Borland/AppServer/doc/books/vbj/vbj45` 디렉토리에 있는 *VisiBroker for Java Programmer's Guide*의 7장 "Using POAs"를 참조하십시오.

6 단계: CORBA 인터페이스 구현 제공

애플리케이션은 이제 잔고가 0인 `Account` 객체들을 생성하는 사용자 인터페이스를 갖게 되었습니다. 예제를 좀 더 흥미롭게 만들려면 고객의 데이터베이스, 계좌 번호를 추가할 수 있으며 또 조회도 가능하고 출금 및 예금을 통해 계좌에 차변, 대변을 작성하는 데 사용할 수 있는 잔고를 추가할 수 있습니다. 그러나 이 간단한 예제에서는 한 가지 구현만 추가하여 이름의 글자 수를 기준으로 주어진 계좌 이름에 대한 잔고를 생성하기로 하겠습니다.

다음과 같은 방법으로 CORBA 인터페이스를 구현합니다.

1 `banktutorial.Bank.server` 패키지를 프로젝트 창에 확장하여 모든 파일을 표시합니다.

2 `AccountImpl.java` 파일을 더블 클릭하여 컨텐트 창에 엽니다.

3 `balance()` 메소드를 찾아봅니다(***Ctrl+F***). `return 0`을

```
return _name.length()*100¿³²Œ ¼ŸðŸ?¥œ¥Ÿ.
```

4 File|Save All을 선택합니다.

다음 단계에서 Project|Make 명령을 사용하여 애플리케이션을 컴파일합니다.

7 단계: 애플리케이션 컴파일

프로젝트를 컴파일하려면 Project|Make Project "BankTutorial.jpx"를 선택합니다.

메시지 창에 오류가 표시됩니다. 다음 단계로 계속하기 전에 구문 오류를 수정하십시오.

컴파일 이전 작업

SmartAgent를 14000이 아닌 다른 포트에서 실행하도록 설정한 경우 다음 VM 매개변수를 Project Properties 대화상자(Tools|Project Properties)의 Run 페이지에 있는 Application 탭의 VM Parameters 필드에 입력해야 합니다.

```
-Dvbroker.agent.port=port number
```

8 단계: Java 애플리케이션 실행

다음과 같은 방법으로 자습서 애플리케이션을 실행합니다.

1. Smart Agent를 시작합니다.
2. 서버 구현을 시작합니다.
3. 클라이언트 애플리케이션을 실행합니다.
4. 애플리케이션을 테스트하고 배포합니다.

이 단계들은 다음 단원에서 자세히 설명합니다.

Smart Agent 시작

Smart Agent 또는 `OsAgent`는 객체 위치 서비스입니다. 클라이언트는 OsAgent를 사용하여 서버를 검색하고 서버는 OsAgent를 사용하여 서비스를 알려줍니다. 이것이 서로를 찾는 방법입니다. 일반적으로 사용자는 Smart Agent가 최소한 로컬 네트워크의 호스트 한 대에서 실행되기를 바랍니다. Smart Agent는 *VisiBroker for Java Programmer's Guide*의 16장 "Using the Smart Agent"에 설명되어 있습니다.

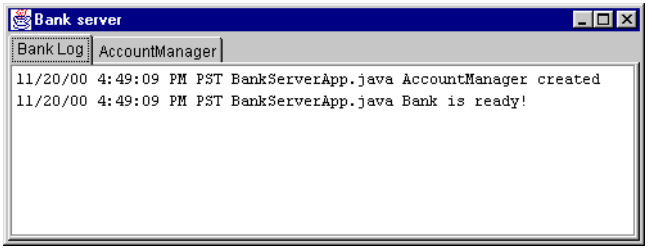
Smart Agent를 시작하려면 Tools|VisiBroker Smart Agent를 선택합니다. Smart Agent는 Tools 메뉴에서 선택 또는 선택 해제하여 토글할 수 있습니다.

Windows NT를 실행하고 Smart Agent를 NT Service로 시작하려는 경우 설치 시 ORB Services를 NT Services로 등록해야 합니다. 자세한 설명은 VisiBroker for Java *Installation Guide*를 참조하십시오. Service가 등록되어 있는 경우 Services Control Panel을 통해 NT Service로서 Smart Agent를 시작할 수 있습니다. Tools 메뉴에서 시작할 필요는 없습니다.

또는 `Borland/AppServer/bin` 디렉토리에서 `osagent.exe`를 실행하여 명령줄에서 Smart Agent를 실행할 수 있습니다.

서버 시작

서버 구현을 시작하려면 프로젝트 창에서 BankServerApp.java 파일을 마우스 오른쪽 버튼으로 클릭합니다. Run을 선택합니다. Bank 서버 창이 표시됩니다. 다음과 같이 나타납니다.

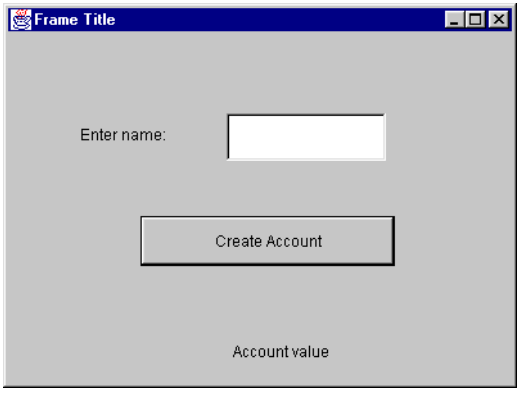


BankServerApp.java가 실행되면 virtual machine이 호출되어 다른 특수 기능을 제공합니다. 명령줄 옵션 및 자세한 내용은 *VisiBroker for Java Reference Manual*의 "Programmer Tools"를 참조하십시오.

클라이언트 실행

Java 클라이언트를 실행하려면 프로젝트 창에서 Application1.java를 마우스 오른쪽 버튼으로 클릭합니다. Run을 선택합니다.

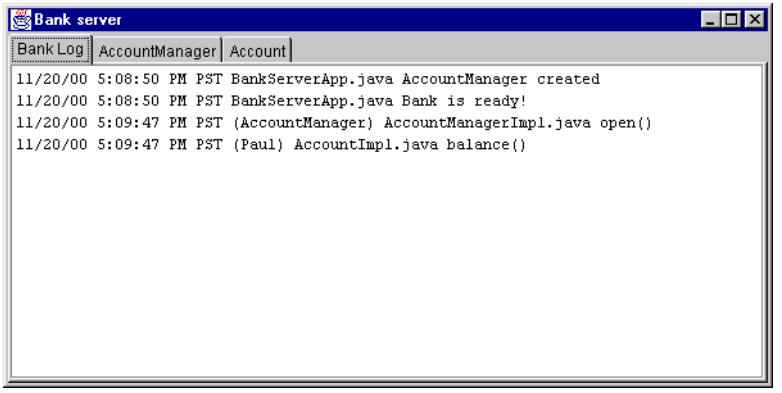
실행 애플리케이션이 다음과 같이 나타납니다.



텍스트 필드에 이름(예: Paul)을 입력하고 Create Account 버튼을 클릭합니다. 생성된 잔고가 아래 그림처럼 레이블에 표시됩니다.

계좌 잔고는 이름의 글자 수에 100을 곱한 것을 기본으로 합니다.

서버 모니터는 새 `Account`를 생성할 때마다 메시지를 출력합니다. 서버측에 다음과 같은 출력이 나타납니다.



AccountManager 페이지에 생성된 `AccountManager` 객체의 수가 표시됩니다. 이 인터페이스는 계좌가 없는 경우 사용자를 위해 계좌를 생성합니다. 이 예제에서는 하나의 `AccountManager` 인터페이스 객체만 생성됩니다. Account 페이지에 생성된 `Account` 객체의 수가 표시됩니다. `balance()` 메소드가 호출될 때마다 `Account` 객체가 생성됩니다.

애플리케이션 배포

VisiBroker는 배포 단계에서도 사용됩니다. 개발자가 테스트하여 실제 준비가 된 클라이언트 프로그램이나 서버 애플리케이션을 생성했을 때 이 단계를 따릅니다. 이 시점에서 시스템 관리자는 최종 사용자의 데스크탑에 클라이언트 프로그램을 배포하거나 서버 클래스 시스템에 서버 애플리케이션을 배포할 준비를 갖추게 됩니다.

배포 단계에서 VisiBroker ORB는 프런트 엔드에서 클라이언트 프로그램을 지원합니다. 사용자는 클라이언트 프로그램을 실행하는 시스템마다 ORB를 설치해야 합니다. 동일한 호스트에서 ORB를 사용하는 클라이언트들은 ORB를 공유합니다. VisiBroker ORB는 중간 계층의 서버 애플리케이션도 지원합니다. 사용자는 서버 애플리케이션을 실행하는 시스템마다 ORB 전체를 설치해야 합니다. 동일한 서버 시스템에서 ORB를 사용하는 서버 애플리케이션 또는 객체는 ORB를 공유합니다. 클라이언트는 GUI 프런트 엔드, 애플릿, 클라이언트 프로그램이 될 수 있습니다. 서버 구현은 중간 계층의 비지니스 로직을 포함합니다.

JBuilder는 VisiBroker for Java "Developer Version"의 `DEVELOPMENT LICENSE` 하나를 포함합니다. 이 라이센스는 JBuilder와 함께 사용되어야 합니다. 다른 개발자들은 VisiBroker 개발 라이센스가 별도로 있어야 합니다. 배포된 프로그램은 각 서버 시스템마다 별도의 VisiBroker 개발 라이센스를 필요로 합니다.

기타 예제 애플리케이션

`Borland/AppServer` 설치 시 `examples` 디렉토리에는 VisiBroker와 함께 몇 개의 예제 CORBA 애플리케이션이 제공됩니다. 각 예제들은 *VisiBroker for Java Programmer's Guide*에 설명되어 있는 기능과의 통합을 시도합니다.

Chapter

24

Java에서의 인터페이스 정의

이것은 JBuilder 기업용 버전의 기능입니다.

이 장에서는 VisiBroker 컴파일러, java2iiop 및 java2idl을 사용하여 IDL 대신 Java로 작성된 인터페이스 정의로부터 클라이언트 스텁(stub)과 servant를 생성하는 방법에 대해 설명합니다. Java용 VisiBroker는 JBuilder Enterprise와 함께 제공되며 Borland AppServer의 일부입니다.

이 장에는 다음 항목들이 포함됩니다.

- "개요"-이러한 컴파일러에 대한 옵션 및 사용에 대해 설명합니다.
- "java2iiop 컴파일러 작업"-모든 Java, IIOP 호환 환경에서 분산 애플리케이션을 만드는 방법에 대해 설명합니다.
- "java2idl 컴파일러 작업"-원하는 프로그램 언어로 클라이언트 스텁을 생성할 수 있도록 자바 코드를 IDL 인터페이스로 전환하는 방법에 대해 설명합니다.

*VisiBroker for Java Programmer's Guide*에 이러한 컴파일러에 대한 자세한 내용이 나와 있습니다. 이 안내서는 Borland AppServer 설치의 Borland/AppServer/doc/books/vbj/vbj45/programmers-guide 디렉토리에 들어 있습니다. intro.html 파일을 열어 안내서의 소개로 이동합니다.

개요

VisiBroker는 제품을 웹에 친숙하고 자바 환경에서 사용하기 쉽게 만들어 주는 다음과 같은 컴파일러들을 통합합니다. 이러한 기능은 다음을 포함합니다.

- **java2iiop**

  java2iiop 컴파일러를 통해 사용자가 원하는 경우 모든 자바 환경에 머무를 수 있습니다. java2iiop 컴파일러는 Java 인터페이스를 사용하고 IIOP 호환 스텁 및 뼈대를 생성합니다. java2iiop 컴파일러를 사용 상의 이점 중 하나는 확장 가능한 구조를 사용함으로써 Java 직렬화된 객체

를 값에 따라 전달할 수 있다는 점입니다. 이 기능은 Java용 VisiBroker에 유일합니다. 자세한 내용은 *VisiBroker for Java Programmer's Guide*의 26장 "Using RMI over OOP"에서 24- 4페이지의 "java2iiop 컴파일러 작업"을 참조하십시오.

- java2idl

  java2idl 컴파일러는 Java 코드를 IDL로 변환하므로 원하는 프로그램 언어에서 클라이언트 스텁을 생성할 수 있습니다. 뿐만 아니라 이 컴파일러는 사용자의 Java 인터페이스를 IDL로 매핑하므로 사용자가 동일한 IDL을 지원하는 다른 프로그램 언어로 Java 객체를 다시 구현할 수 있습니다. 이 컴파일러에 대한 자세한 내용은 *VisiBroker for Java Reference*의 2장 "Programmer Tools"를 참조하고 24- 9페이지의 "java2idl 컴파일러 작업"을 읽어 보십시오.

- Web Naming

  Web Naming을 통해 URL을 객체에 연결하여 URL 지정을 통한 객체 참조를 얻을 수 있습니다. 자세한 내용은 *VisiBroker for Java Programmer's Guide*를 참조하십시오.

Java로 CORBA 인터페이스를 설계할 경우 java2iiop를 컴파일러로 사용하여 클라이언트 스텁(stub)과 servant를 만들어야 합니다. Java로 CORBA 인터페이스를 설계할 경우 IDL 파일을 만들지 않아도 됩니다. 사실 생성된 IDL 파일은 수정되거나 컴파일될 수 없습니다. 다음의 경우에만 Java 인터페이스에서 IDL 파일을 생성합니다.

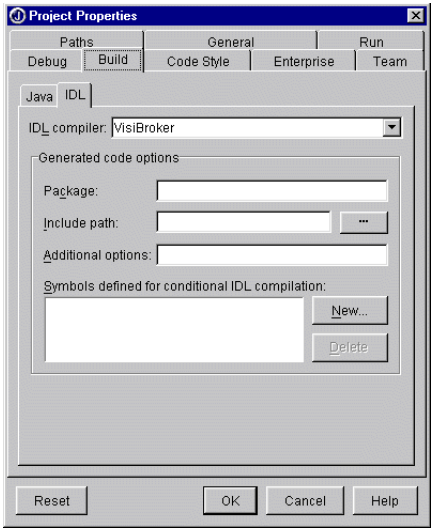
- 인터페이스 레포지토리를 채웁니다.
- 크로스 랭귀지 애플리케이션을 만듭니다.

JBuilder에서 java2iiop 및 java2idl 컴파일러에 액세스

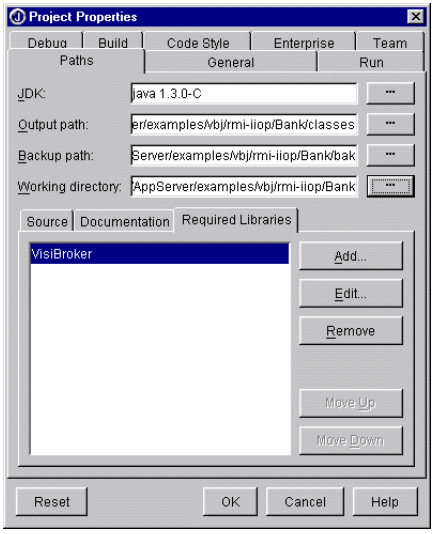
다음과 같은 방법으로 JBuilder 개발 환경에서 java2iiop 및 java2idl 컴파일러에 액세스합니다.

1 Chapter 22 "CORBA 애플리케이션용 JBuilder 설정"에서 설명하는 단계에 따라 JBuilder를 올바르게 설정해야 합니다.

2 Project|Project Properties를 선택하여 Project Properties 대화 상자를 표시합니다. Build 페이지의 IDL 탭에서 IDL Compiler 필드를 VisiBroker로 설정합니다.



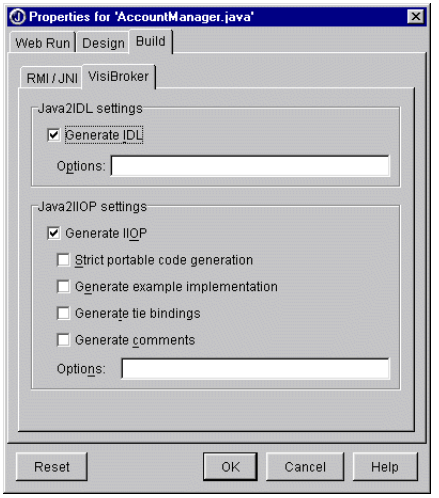
3 Paths 페이지에서 Required Libraries 목록에 VisiBroker가 있는 지 확인합니다. 대화 상자는 다음과 같습니다.



4 Project Properties 대화 상자를 닫으려면 OK를 클릭합니다.

5 JBuilder 프로젝트에서 Java 인터페이스 파일을 만들었으면 프로젝트 창에서 파일을 마우스 오른쪽 버튼으로 클릭합니다. Properties를 선택합니다.

6 Build 페이지의 Visibroker 탭에서 Generate IDL과 Generate IIOP 옵션을 선택합니다. 대화 상자는 다음과 같습니다.



7 대화 상자를 닫으려면 OK를 클릭합니다.

8 프로젝트 창에서 Java 인터페이스 파일을 마우스 오른쪽 버튼으로 클릭한 후 Make를 선택하여 인터페이스 정의를 컴파일하고 IIOP 또는 IDL 인터페이스 파일을 생성합니다.

RMI

원격 Java 객체에서 메소드를 호출하는 또다른 방법은 RMI(Remote Method Invocation)를 사용하는 것입니다. Java용 VisiBroker는 RMI와 동일한 기능을 제공할 뿐만 아니라 Java로 작성되지 않았더라도 CORBA와 호환되는 다른 모든 객체와 통신하는 Java 객체를 만들 수 있게 해줍니다. RMI 원격 인터페이스를 설명하는 Java 인터페이스의 예제는 `Borland/AppServer/examples/vbj/rmi-iiop/Bank` 디렉토리의 `Account.java` 예제 파일을 참조하십시오. RMI 개발에 대한 자세한 내용은 Chapter 25 " JBuilder의 Java RMI 기반분산 애플리케이션 살펴보기"을 참조하십시오.

java2iiop 컴파일러 작업

`java2iiop` 컴파일러를 통해 CORBA에서 인터페이스와 데이터 타입으로 사용할 수 있는 인터페이스와 데이터 타입을 정의할 수 있습니다. IDL이 아닌 Java로 인터페이스와 데이터 타입을 정의할 수 있다는 것이 이점입니다. 컴파일러는 자바 바이트 코드를 읽고 소스 코드는 읽지 않지만 클래스 파일은 읽습니다. 그런 다음 컴파일러는 CORBA에 필요한 모든 마샬링과 통신을 수행하는 데 필요한 IIOP 호환 스텁과 뼈대를 생성합니다.

java2iiop 컴파일러를 실행하면 IDL로 인터페이스를 작성한 것과 동일한 파일이 생성됩니다. 수치 타입(short, int, long, float, double), 문자열, CORBA 객체 또는 인터페이스 객체, 임의의 타입 및 타입 코드 등 모든 기본(primitive) 데이터 타입은 java2iiop 컴파일러에서 인식되며 IDL의 해당 타입에 매핑됩니다.

자바 인터페이스에서 java2iiop 컴파일러를 사용 시 이 컴파일러를 원격 호출에 사용되는 인터페이스로 정의하고 표시합니다. 이 컴파일러에서 org.omg.CORBA.Object 인터페이스를 확장하여 표시할 수 있습니다. (또한 인터페이스는 원격 호출에서 사용할 수 있는 메소드를 정의해야 합니다.) 컴파일러를 실행할 때 이러한 특수 CORBA 인터페이스를 검색합니다. 이러한 인터페이스가 있는 경우 원격 호출에서 인터페이스를 사용할 수 있게 해주는 마샬링 요소, readers 및 writers가 모두 생성됩니다. 자바 인터페이스 정의 파일의 예제인 AccountManager.java는 Borland/AppServer/examples/vbj/rmi-oop/Bank 디렉토리에 들어 있습니다.

참고 RMI(Remote Method Invocations)에 숙련된 개발자에게는 이것이 java.rmi.Remote 인터페이스를 확장하는 클래스와 같습니다.

클래스의 경우, 컴파일러는 다른 규칙을 따르며 클래스를 IDL 구조 또는 확장 가능한 구조로 매핑합니다. 복잡한 데이터 타입에 대한 자세한 내용은 24- 7페이지의 "복잡한 데이터 타입의 매핑"을 참조하십시오.

java2iiop를 실행하는 IIOP 인터페이스 생성

Borland/AppServer/examples/vbj/rmi-oop 디렉토리에 있는 예제 파일은 Java에서 인터페이스를 정의하고 java2iiop 컴파일러를 사용하여 이러한 인터페이스를 IIOP 호환 스텁(stub)과 servant로 컴파일하는 방법을 보여 줍니다.

이 예제의 파일은 다음과 같습니다.

- Bank/Account.java – Account RMI 원격 인터페이스를 설명하는 인터페이스.
- Bank/AccountData.java – 데이터를 전달하여 계좌를 만드는 데 사용하는 클래스.
- Bank/AccountManager.java – AccountManager CORBA 인터페이스에 대해 설명하는 인터페이스.
- Bank/AccountManagerOperations.java – AccountManager.java.로 정의된 메소드 시그너처를 선언하는 인터페이스.
- AccountImpl.java – Account 인터페이스를 구현하는 servant
- AccountManagerImpl.java – AccountManager 인터페이스를 구현하는 servant.
- Client.java – 클라이언트 메인 라인.
- Server.java – 서버 메인 라인.

AccountManager 인터페이스는 계좌를 생성하고 계좌를 열거나 다시 열며 계좌의 목록을 얻습니다. 생성된 계좌는 Account 이름을 얻고 계좌의 잔액을 설정하고 얻는 연산을 포함합니다. AccountData는 계좌를 생성하는 데 사용되는 정보를 나타냅니다.

이 예제는 IIOP를 통한 RMI 사용과 원격 메소스 호출에서 IIOP를 통한 RMI를 사용하여 (IDL로부터 매핑되지 않은) 원시 Java 클래스를 전달하는 방법을 설명합니다. 또한 CORBA 서버에서 RMI 스타일 원격 인터페이스를 사용하고 CORBA 서버에 액세스하기 위해 RMI 클라이언트를 사용하는 방법에 대해 설명합니다. 예제를 실행하고 보기 위해서는 Borland/AppServer/examples/vbj/rmi-oop 디렉토리의 rmi-iiop_java.html 파일을 참조하십시오.

JBuilder에서 IIOP 인터페이스를 생성하려면 Properties 대화 상자에서 java2iiop 컴파일러를 선택해야 합니다. 다음과 같은 방법으로 이 작업을 수행합니다.

1. 프로젝트 창에서 Java 인터페이스 파일인 AccountManager를 마우스 오른쪽 버튼으로 클릭합니다. 컨텍스트 메뉴에서 Properties를 선택합니다.
2. Build 페이지의 VisiBroker 탭에서 Generate IIOP 옵션을 선택한 다음 OK를 클릭합니다.
3. Java 인터페이스 파일을 마우스 오른쪽 버튼으로 클릭한 다음 컨텍스트 메뉴에서 Make를 선택합니다.

그러면 java2iiop 컴파일러는 Helper 및 Holder 클래스 같은 보조 파일을 모두 생성합니다. 생성된 파일에 대한 자세한 내용은 *VisiBroker for Java Programmer's Guide*의 4장 "Developing an example application with Visibroker"를 참조하십시오.

참고 java2iiop 컴파일러에 의해 생성된 파일은 idl2java 컴파일러에 의해 생성된 파일과 동일합니다.

개발 프로세스 완료

*VisiBroker for Java Programmer's Guide*에는 Java용 VisiBroker를 사용하는 이 예제의 개발 프로세스를 완료하는 예제 코드와 예제 애플리케이션을 포함한 자세한 내용이 들어 있습니다. 일반적으로 스텁과 뼈대를 생성한 후에 클라이언트와 서버 클래스를 생성합니다. 그런 다음 다음 단계를 따릅니다.

1. 뼈대 클래스를 사용하여 서버 객체의 구현을 생성합니다. 이에 대한 예제는 Borland/AppServer/examples/vbj/rmi-oop 디렉토리의 Server.java를 참조하십시오.
2. 프로젝트 창에서 서버 파일을 마우스 오른쪽 버튼으로 클릭하고 Make를 선택하여 서버 클래스를 컴파일합니다.

참고 AppServer/examples/vbj/rmi-oop/ 디렉토리에서 Server.java를 사용하는 경우 해당 디렉토리의 Makefile과 Makefile.java의 이름을 재지정해야 합니다. 왜냐하면 JBuilder의 컴파일러와 충돌할 수도 있기 때문입니다.

3. 클라이언트를 위한 코드를 작성합니다. 이에 대한 예제는 Borland/AppServer/examples/vbj/rmi-oop 디렉토리의 Client.java를 참조하십시오.
4. 프로젝트 창에서 클라이언트 파일을 마우스 오른쪽 버튼으로 클릭하고 Make를 선택하여 클라이언트 코드를 컴파일합니다.

참고 AppServer/examples/vbj/rmi-oop/ 디렉토리에서 Client.java를 사용하는 경우 해당 디렉토리의 Makefile과 Makefile.java의 이름을 재지정해야 합니다. 왜냐하면 JBuilder의 컴파일러와 충돌할 수도 있기 때문입니다.

5. Tools 메뉴에서 VisiBroker Smart Agent를 선택하여 시작합니다. 이 옵션을 사용하려면 Chapter 22 "CORBA 애플리케이션용 JBuilder 설정"에서 설명한 대로 JBuilder를 설정해야 합니다.
6. 프로젝트 창에서 서버 파일을 마우스 오른쪽 버튼으로 클릭하고 Run을 선택하여 서버 객체를 시작합니다.
7. 프로젝트 창에서 클라이언트 파일을 마우스 오른쪽 버튼으로 클릭하고 Run을 선택하여 클라이언트를 시작합니다.

중요 이 장에 있는 예제는 VisiBroker Smart Agent가 실행 중이라고 가정합니다. Tools|VisiBroker Smart Agent를 선택하여 Smart Agent를 실행해야 합니다. Smart Agent에 대한 자세한 내용은 *VisiBroker for Java Programmer's Guide*의 16장 "Using the Smart Agent"를 참조하십시오.

기본 데이터 타입을 IDL로 매핑

java2iiop에 의해 생성된 클라이언트 스텁(stub)은 객체 서버에 전송될 수도 있도록 작업 요청을 나타내는 Java 기본 데이터 타입의 마샬링을 처리합니다. Java 기본 데이터 타입을 마샬링할 때 IIOP 호환 형식으로 변환해야 합니다. 자세한 내용은 *VisiBroker for Java Programmer's Guide*의 26장 "RMI over OOP"를 참조하십시오.

복잡한 데이터 타입의 매핑

이 단원에서는 java2iiop 컴파일러를 사용하여 인터페이스, 배열, Java 클래스 및 확장 가능한 구조체 등의 복잡한 데이터 타입을 처리하는 방법에 대해 설명합니다. 자세한 내용은 *VisiBroker for Java Programmer's Guide*의 26장 "RMI over OOP"를 참조하십시오.

인터페이스

Java 인터페이스는 IDL에서 CORBA 인터페이스로 표시됩니다. Java 인터페이스는 org.omg.CORBA.Object 인터페이스에서 상속해야 합니다. 이러한 인터페이스를 구현하는 객체를 전달하는 경우 참조에 의해 전달됩니다.

참고 java2iiop 컴파일러는 CORBA 인터페이스에서 오버로드된 메소드를 지원하지 않습니다.

org.omg.CORBA.Object를 확장하지 않는 인터페이스는 확장 가능한 구조체로 매핑됩니다.

배열

클래스에서 정의될 수도 있는 또 다른 복잡한 데이터 타입은 배열입니다. 배열을 사용하는 인터페이스 또는 정의가 있는 경우 배열은 CORBA 언바운드 시퀀스로 매핑됩니다.

Java 클래스 매핑

Java 클래스에서 임의 데이터 타입을 정의할 수 있습니다. 일부 임의 데이터 타입은 *구조체*라고도 하는 IDL 구조체와 유사합니다. 특정 요구 사항에 맞도록 Java 클래스를 정의하는 경우 java2iiop 컴파일러는 그 클래스를 IDL 구조체로 매핑합니다. 클래스가 다음과 같은 요구 사항에 ³šµŒ 맞는 경우 Java 클래스를 IDL 구조체로 매핑할 수 있습니다.

- 클래스는 final입니다.
- 클래스는 public입니다.
- 클래스는 구현 상속을 사용하지 않습니다.
- 클래스의 데이터 멤버는 public입니다.

java2iiop 컴파일러는 클래스를 검사하여 이러한 요구 사항을 모두 충족하는지 확인합니다. 클래스가 요구 사항을 모두 충족하면 컴파일러는 이 클래스를 IDL 구조체로 매핑하고, 요구 사항을 모두 충족하지 못하면 클래스를 확장 가능한 구조체로 매핑합니다.

확장 가능한 구조체

위에 나열한 요구 사항을 모두 충족하지 않는 Java 클래스는 확장 가능한 구조체로 매핑됩니다. 확장 가능한 구조체는 CORBA 구조체의 상향 호환 확장입니다. 확장 가능한 구조체를 사용할 때 객체는 값에 의해 전달됩니다.

값에 의한 전달은 객체 상태를 다른 Java 프로그램으로 전달하는 기능입니다. Java 객체의 클래스 정의가 서버 측에 있다고 가정할 경우 Java 프로그램은 원래 객체와 동일한 상태를 갖는 복제된(cloned) 객체에서 메소드를 호출할 수 있습니다.

확장 가능한 구조체의 사용은 OMG IDL에 대한 VisiBroker 확장입니다. 다음과 같은 추가 키워드 extensible을 제공합니다. 순수 CORBA 내에 머무르거나 코드를 다른 ORB로 이식하려는 경우에는 확장 가능한 구조체가 아닌 IDL 구조체를 사용해야 합니다. 확장 가능한 구조체를 통해 Java로 정의될 수 있지만 CORBA 제한 때문에 IDL로 정의될 수 없는 클래스를 사용할 수 있습니다.

VisiBroker는 Java 직렬화를 이용하여 확장 가능한 구조체의 형식으로 클래스를 전달합니다. Java 직렬화는 Java 객체 상태를 요청의 일부로서 유선 상에 전달될 수 있는 8진수의 순차적인 스트림으로 압축합니다.

참고 Java 직렬화 때문에, 전달되는 모든 데이터 타입은 순차 가능하도록 java.io.Serializable을 구현해야 합니다.

확장 가능한 구조체를 통해 포인터 의미를 사용하는 데이터를 성공적으로 전달할 수 있습니다. 예를 들어, 애플리케이션 코드로 연결 목록을 정의하는 것은 표준 작업이지만 표준 IDL로 연결 목록을 정의할 수는 없습니다. 확장 가능한 구조체로 연결 목록을 정의하여 이 문제를 해결할 수 있습니다. 확장 가능한 구조체를 주소 영역에 전달할 때 포인터는 유지됩니다.

java2idl 컴파일러 작업

IDL은 CORBA 애플리케이션의 기반입니다. IDL 인터페이스는 컴포넌트의 범위를 정의합니다. IDL을 사용하여 컴포넌트의 속성, 발생하는 예외, 인터페이스가 지원하는 메소드를 정의할 수 있습니다.

java2idl 컴파일러를 사용하여 .java 파일에 정의된 Java 인터페이스로부터 IDL 인터페이스를 생성할 수 있습니다. Java 인터페이스를 사용하여 CORBA 객체를 나타내는 데에는 몇 가지 제한이 있습니다. 모든 Java 기본 타입을 사용할 수 있습니다. 하지만 객체가 java.io.Serializable을 구현하는 경우에만 인터페이스를 정의하는 데 Java 객체를 사용할 수 있습니다.

다음 단계는 Java 인터페이스 정의로부터 IDL 인터페이스를 생성하는 방법입니다.

1. 프로젝트 창에서 Java 인터페이스 파일을 마우스 오른쪽 버튼으로 클릭합니다. 컨텍스트 메뉴에서 Properties를 선택합니다.
2. Build 페이지의 VisiBroker 탭에서 Generate IDL 옵션을 선택한 다음 OK를 클릭합니다.
3. Java 인터페이스 파일을 마우스 오른쪽 버튼으로 클릭한 다음 컨텍스트 메뉴에서 Make를 선택합니다.

그러면 java2idl 컴파일러는 Helper 및 Holder 클래스 같은 보조 파일을 모두 생성합니다. 프로젝트 창에서 Java 인터페이스 파일의 왼쪽에 있는 더하기(+) 기호를 클릭하여 생성된 파일을 표시합니다. 생성된 파일에 관한 자세한 내용은 *VisiBroker for Java Programmer's Guide*의 4장 "Developing an example application with Visibroker"를 참조하십시오.

Chapter

25

JBuilder의 Java RMI 기반 분산 애플리케이션 살펴보기

이것은 JBuilder 전문 가용 및 기업용 버전의 기능입니다.

RMI(Remote Method Invocation)를 사용하면 분산된 Java 간 애플리케이션을 만들 수 있으므로, 다른 호스트 상에서도 다른 Java 가상 머신이 원격 Java 객체의 메소드를 호출할 수 있습니다. Java 프로그램은 원격 객체에 대한 참조를 얻고 나면 RMI에서 제공하는 부트스트랩(bootstrap) 이름 지정 서비스에서 원격 객체를 조회하거나 인수 또는 반환값으로 참조를 수신하여 원격 객체를 호출할 수 있습니다. 클라이언트는 서버에 있는 원격 객체를 호출할 수 있고 이 서버 또한 다른 원격 객체의 클라이언트가 될 수 있습니다. RMI는 순수 객체 지향 다형성을 지원하므로 객체 직렬화를 사용하여 매개변수를 마샬링(marshaling)하거나 마샬링 해제하며 타입을 자르지 않습니다.

Java에서 인터페이스를 정의하는 또 다른 방법으로 `java2iiop` 컴파일러를 사용할 수 있습니다. 이 컴파일러를 사용하면 모든 Java환경에서 작업이 가능합니다. 이 컴파일러는 Java 인터페이스를 사용하여 IIOP에 순응하는 스텁(stub)과 뼈대를 생성합니다. `java2iiop` 컴파일러 사용 상의 이점 중 하나는 확장 가능한 구조를 사용함으로써 Java 직렬화된 객체를 값에 따라 전달할 수 있다는 점입니다. 이 컴파일러에 대한 자세한 내용은 24- 1페이지의 "Java에서의 인터페이스 정의"을 참조하십시오. 이것은 JBuilder 기업용 버전의 기능입니다.

예제 RMI 애플리케이션인 `SimpleRMI.jpr`는 JBuilder 설치 시 `samples/Rmi` 디렉토리에 설치됩니다. 예제 애플리케이션에 대한 설명은 HTML 프로젝트 파일인 `SimpleRMI.html`을 참조하십시오.

RMI 및 `DataSetData`를 사용하는 분산 데이터베이스 애플리케이션을 작성한 예는 JBuilder 설치의 `samples/DataExpress/StreamableDataSets` 디렉토리에서 보실 수 있습니다. 사원 샘플 테이블에서 데이터를 가져와서 RMI을 통해 `DataSetData` 형태로 이 데이터를 송신하는 서버 애플리케이션이 이 예제에

포함되어 있습니다. 클라이언트는 사용자 지정 Provider와 사용자 지정 Resolver를 통해 서버와 통신하고 그리드에 데이터를 보여 줍니다.

참고 RMI 애플리케이션을 만드는데 필요한 도구가 Java와 함께 제공되므로 모든 JBuilder 버전에서 RMI 기능을 사용할 수 있습니다. 그러나 본 자습서에 나온 각 단계들은 JBuilder IDE에 있는 도구의 사용 방법을 보여 줍니다. JBuilder Foundation을 가지고 있을 경우에는 RMI 명령줄 도구 사용 시 이 단계들을 수정해야 합니다.

자습서: JBuilder에서 Java RMI 기반 분산 애플리케이션 생성

본 자습서는 Java Remote Method Invocation(RMI)을 사용하여 종래 "Hello World" 프로그램의 배포용 버전을 생성할 때 따라야 하는 단계를 보여줍니다. 배포용 "Hello World" 예제는 Tomcat(기본 웹 서버)에서 실행되는 애플릿을 사용하여 애플릿이 다운로드된 서버로 원격 메소드를 호출합니다. 그런 다음 서버로부터 "Hello World!"라는 메시지를 가져옵니다. 애플릿이 실행될 경우 "Hello World!"가 클라이언트 브라우저에 나타납니다.

본 자습서는 다음과 같이 3개 과정으로 나뉘어 있습니다.

- 1과:Java 소스 및 HTML 파일 생성
- 2과: 클래스 파일과 HTML 파일 컴파일
- 3과:RMI 레지스트리, 서버 및 애플릿 시작

본 자습서에서 사용하는 소스 코드는 JBuilder 내의 해당 파일에 복사할 수 있습니다. 이 파일들은 다음과 같습니다.

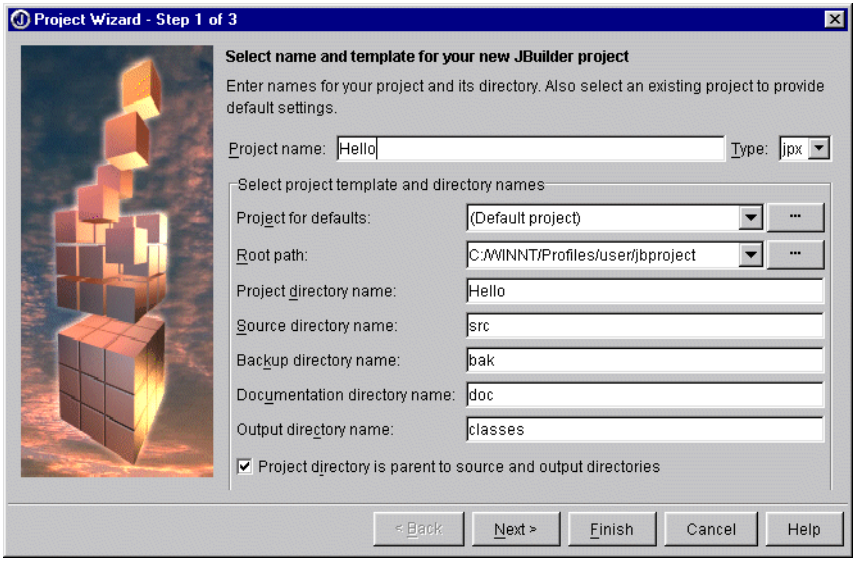
- Hello.java – 원격 인터페이스.
- HelloImpl.java – Hello.java를 구현하는 원격 객체 구현입니다.
- HelloApplet.java – 원격 메소드 sayHello를 호출하는 애플릿입니다.
- HelloApplet.html – 웹 페이지용 HTML 코드입니다.
- rmi.policy – 정책 파일입니다.

1과:Java 소스 및 HTML 파일 생성

시작하려면 JBuilder 프로젝터를 생성해야 합니다. 다음과 같은 방법으로 프로젝트를 생성합니다.

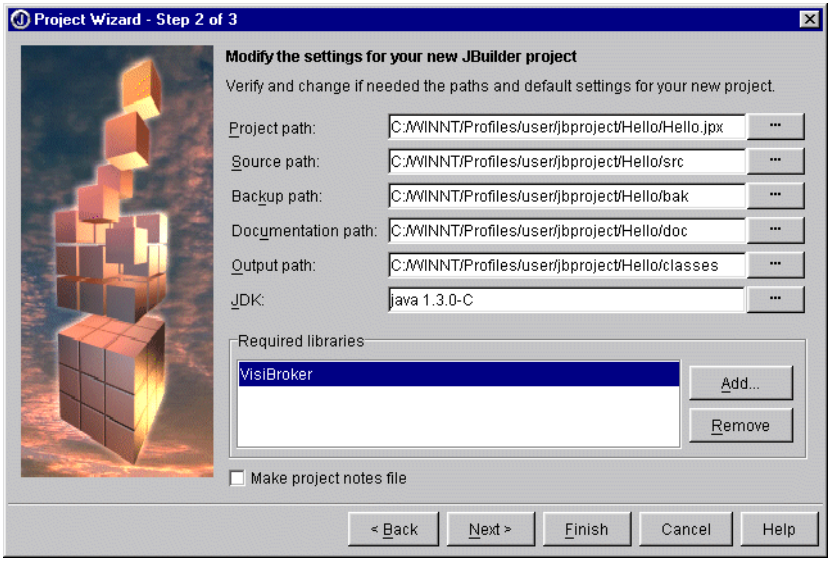
1 File|New Project를 선택합니다. Project 마법사가 나타납니다.

2 Project Name을 `Hello`로 변경합니다. Project Directory Is Parent To Source And Output Directories 옵션이 선택 표시되도록 합니다. Finish를 클릭합니다. 마법사의 1 단계는 다음과 같이 보입니다.



3 마법사의 2 단계로 가려면 Next를 클릭합니다.

4 2 단계 페이지에서 VisiBroker 라이브러리가 Required Libraries 목록에 표시되도록 합니다. VisiBroker 라이브러리가 표시되지 않을 경우에는 Add를 클릭하고 Select One Or More Libraries 대화 상자에서 VisiBroker 라이브러리를 선택합니다. 마법사의 2 단계는 다음과 같이 보입니다.



프로젝트 파일 Hello.jpx와 프로젝트 노트 파일 Hello.html이 프로젝트 창에 표시됩니다.

1과에서는 적용 가능한 인터페이스와 구현을 단계적으로 생성합니다.

- 1 단계: 원격 인터페이스 함수 정의
- 2 단계: 구현과 서버 클래스 작성
- 3 단계: 원격 서비스를 사용하는 클라이언트 프로그램 작성
- 4 단계: 규칙 파일 쓰기(작성)

1 단계: 원격 인터페이스의 함수 정의

원격 객체는 원격 인터페이스를 구현하는 클래스 인스턴스입니다. 원격 인터페이스는 사용자가 원격으로 호출하려는 모든 메소드를 선언합니다. 원격 인터페이스에는 다음과 같은 특징이 있습니다.

- 원격 인터페이스는 public으로 선언되어야 합니다. 그렇지 않으면 클라이언트가 원격 인터페이스와 같은 패키지에 들어 있지 않을 경우 원격 인터페이스를 구현하는 원격 객체를 로드하려 할 때 클라이언트에 오류가 발생할 수 있습니다.
- 원격 인터페이스는 java.rmi.Remote 인터페이스를 확장해야 합니다.
- 각각의 메소드는 애플리케이션 특정 예외 이외에도 자신의 throws 절에 있는 java.rmi.RemoteException이나 RemoteException의 수퍼클래스를 선언해야 합니다.
- 인수 또는 반환값(직접 또는 로컬 객체에 간접적으로 포함됨)으로 전달된 원격 객체의 데이터 타입은 HelloImpl과 같은 구현 클래스가 아닌 Hello와 같은 원격 인터페이스 타입으로서 선언되어야 합니다.

다음과 같은 방법으로 원격 인터페이스에 대한 인터페이스 정의를 생성합니다.

1 File|New Class를 선택합니다.

New Class 마법사가 표시됩니다.

2 Class Name 필드에 Hello를 입력합니다.

3 모든 옵션을 선택 해제합니다.

4 OK를 클릭합니다.

Hello.java 파일이 에디터에 표시됩니다.

5 기존 코드를 삭제하고 다음 코드를 추가합니다.

```
package hello;

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

  public interface Hello extends Remote {
    String sayHello() throws RemoteException;
  }
```

인터페이스에는 문자열을 호출자에게 반환하는 sayHello 메소드만 포함됩니다.

6 File|Save All을 선택합니다.

네트워크와 관련된 통신 문제와 서버상의 문제로 인해 원격 메소드 호출이 로컬 메소드 호출의 경우와 매우 상이한 방법으로 실패할 수 있으므로 원격 메소드는 java.rmi.RemoteException을 통해 통신 실패를 보고합니다. 분산 시스템에서의 실패와 복구에 대한 자세한 내용은 http://www.sunlabs.com/techrep/1994/abstract-29.html의 *A Note on Distributed Computing*을 참조하십시오.

2 단계: 구현 및 서버 클래스 작성

원격 객체 구현 클래스는 최소한 다음과 같은 작업을 수행합니다.

- 최소한 하나의 원격 인터페이스 구현을 선언합니다
- 원격 객체 생성자를 정의합니다
- 원격으로 호출될 수 있는 메소드를 구현합니다.

이 컨텍스트에서 "서버" 클래스는 원격 객체 구현 인스턴스를 생성하는 main() 메소드를 갖는 클래스이며 이 인스턴스를 RMI 리지스트리에 있는 이름으로 바인딩합니다. 이 main() 메소드를 포함하고 있는 클래스는 구현 클래스이거나 완전히 다른 클래스일 수도 있습니다.

다음 예제에서 main() 메소드는 HelloImpl.java의 일부입니다. 이러한 서버 클래스는 다음과 같은 작업을 수행해야합니다.

- 보안 관리자 생성 및 관리
- 두 개 이상의 원격 객체 인스턴스를 생성합니다.
- 부트스트랩(bootstrapping)하기 위해 RMI 원격 객체 리지스트리를 사용하여 적어도 하나의 원격 객체를 등록합니다

다음과 같은 방법으로 "Hello World" 서버용 코드를 포함하는 HelloImpl.java 파일을 생성합니다.

1 File|New Class를 선택합니다.

New Class 마법사가 표시됩니다.

2 Class Name 필드에 HelloImpl을 입력합니다.

3 모든 옵션을 선택 해제하고 OK를 클릭합니다.

HelloImpl.java 파일이 에디터에 표시됩니다.

4 기존 코드를 삭제하고 다음 코드를 추가합니다.

```
package hello;

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements Hello {

  public HelloImpl() throws RemoteException {
    super();
  }

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

  public static void main(String args[]) {

    // Create and install a security manager
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
  try {
      HelloImpl obj = new HelloImpl();
      // Bind this object instance to the name "HelloServer"
         Naming.rebind("//localhost/HelloServer", obj);

      System.out.println("HelloServer bound in registry");
    } catch (Exception e) {
      System.out.println("HelloImpl err: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

5 File|Save All을 선택합니다.

원격 인터페이스 구현

클래스가 인터페이스를 구현을 선언하는 경우에는 클래스와 컴파일러 간에 일종의 계약이 체결됩니다. 이 계약을 체결함으로써 클래스는 자신이 구현하고 있는 인터페이스에서 선언한 각각의 메소드 시그너처에 메소드 몸체나 정의를 제공합니다. 인터페이스 메소드는 암시적으로 public이고 abstract이므로 구현 클래스가 계약을 수행하지 않으면 정의에 의해 abstract 클래스가 되며 클래스가 abstract으로 선언되지 않으면 컴파일러에서 이러한 사실을 지적합니다.

다음 예제에서 구현 클래스는 `HelloImpl.java`입니다. 구현 클래스는 어떠한 원격 인터페이스를 구현하고 있는지 선언합니다. `HelloImpl` 클래스 선언은 다음과 같습니다.

```
public class HelloImpl extends UnicastRemoteObject implements Hello {
```

편의에 따라 구현 클래스는 본 예제에서는 java.rmi.server.UnicastRemoteObject인 원격 클래스를 확장할 수 있습니다. UnicastRemoteObject를 확장하면 HelloImpl 클래스를 다음과 같은 원격 객체를 생성하는데 사용할 수 있습니다.

- 통신하는 데에 RMI의 기본 소켓 기반 전송을 사용합니다.
- 항상 실행합니다.

원격 객체 생성자 정의

원격 클래스 생성자는 비원격(non-remote) 클래스 생성자가 제공하는 것과 동일한은 기능을 제공합니다. 원격 클래스 생성자는 새로 생성된 클래스의 각 인스턴스 변수를 초기화한 다음 생성자를 호출한 클래스로 클래스 인스턴스를 돌려 보냅니다.

원격 인터페이스는 "export" 되어야 합니다. 원격 객체를 export하면 특정 포트의 원격 객체로 수신되는 호출을 수신 대기함으로써 수신되는 원격 메소드 요청을 승인할 수 있습니다. java.rmi.server.UnicastRemoteObject 또는 java.rmi.activation.Activatable을 확장할 경우 클래스는 생성 시 자동으로 export됩니다.

UnicastRemoteObject 또는 Activatable 이외의 클래스로부터 원격 객체를 확장하려 할 경우에는 클래스의 생성자나 해당되는 초기화 메소드로부터 UnicastRemoteObject.exportObject() 메소드나 Activatable.exportObject() 메소드 호출을 통해 원격 객체를 명시적으로 export해야 합니다.

객체를 export하여 잠재적으로 java.rmi.RemoteException을 발생시킬 수 있으므로 생성자가 다른 일을 하지 않더라도 사용자는 RemoteException을 발생시키는 생성자를 정의해야 합니다. 생성자를 정의하지 않으면 컴파일은 다음과 같은 오류 메시지를 나타냅니다.

```
HelloImpl.java:13: Exception java.rmi.RemoteException¿Ã ¼?x?µ«æÓæ? «'¥œ¥Ÿ.
±Þ²³¡^ æÞ¿ª xÊøÏøº¥¬ Exception java.rmi.RemoteException¿½ ¿Ã ³?º"µÂ¿« throws
¿"øºº¦ º±æšµ«æÓæ? «'¥œ¥Ÿ.

       super();
       ^
1 error
```

HelloImpl 클래스의 생성자는 다음과 같습니다.

```
public HelloImpl() throws RemoteException {
    super();
}
```

다음 사항에 유의하십시오.

- super() 메소드 호출은 원격 객체를 export하는 java.rmi.server.UnicastRemoteObject,의 비인수(no-augument) 생성자를 호출합니다.
- 통신 리소스를 사용할 수 없을 경우 생성 기간 동안 원격 객체를 export하고자 하는 RMI 시도가 실패할 수도 있으므로 생성자는 java.rmi.RemoteException을 발생시켜야 합니다.

수퍼클래스의 비인수 생성자(no-argument constructor)인 super()에 대한 호출이 기본 발생할지라도(심지어 호출이 없는 경우에도) 이 예제에서는 이 생성자를 포함하여 Java 가상 머신(VM)에서 클래스보다 먼저 수퍼클래스를 만든다는 사실을 분명하게 보여 줍니다.

각 원격 메소드에 대한 구현 제공

원격 객체를 위한 구현 클래스는 원격 인터페이스에 지정된 각각의 원격 메소드를 구현하는 코드를 포함합니다. 예를 들어, 호출자에게 "Hello World!" 문자열을 돌려 보내는 sayHello() 메소드에 대한 구현은 다음과 같습니다.

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

객체들이 인터페이스 java.io.Serializable을 구현하는 한 원격 메소드로의 인수값이나 원격 메소드로부터의 반환값은 이 객체들을 포함하는 Java 플랫폼용 데이터 타입이 될 수 있습니다. java.lang과 java.util에 들어 있는 대부분의 주요 클래스는 Serializable 인터페이스를 구현합니다. RMI에서 다음과 같은 작업을 수행합니다.

- 기본적으로 로컬 객체는 복사에 의해 전달되는데 이는 static이나 transient를 제외한 모든 객체의 데이터 멤버나 필드가 복사된다는 것을 의미합니다. 기본 직렬화 동작을 변경하는 방법에 대한 자세한 내용은 http://java.sun.com/j2se/1.3/docs/guide/serialization/spec/serialTOC.doc.html의 Java Object Serialization Specification을 참조하십시오.
- 원격 객체는 참조에 의해 전달됩니다. 원격 객체에 대한 참조는 사실상 원격 객체를 위한 클라이언트측의 프락시(proxy)인 스텁에 대한 참조입니다. 스텁(stub)에 대한 자세한 내용은 http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html.에 있는 *RMI Specification*에 설명되어 있습니다. 본 자습서의 뒷부분에에 있는 25- 14페이지의 "Java 소스 파일 컴파일" 단원에서 스텁을 만들게 됩니다.

클래스는 원격 인터페이스에 지정되지 않은 메소드를 정의하는데 이 메소드들은 서비스를 실행하는 가상 머신안에서만 호출될 수 있을 뿐 원격으로는 호출될 수 없습니다.

보안 관리자 생성 및 설치

서버의 main() 메소드에서는 우선 다음과 같은 보안 관리자를 생성하여 설치해야 합니다. RMISecurityManager나 사용자가 정의한 보안 관리자예를 들면, 다음과 같습니다.

```
if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
}
```

로드된 클래스가 수행해서는 안되는 작업을 수행하지 않도록 하기 위해 보안 관리자는 항상 작동해야 합니다. 보안 관리자가 지정되어 있지 않을 경

우에는 로컬 CLASSPATH의 클래스를 제외하고 RMI 클라이언트나 서버에 의해 로드되는 어떠한 클래스도 허용되지 않습니다.

두 개 이상의 원격 객체 인스턴스 생성

서버의 main() 메소드는 서버를 제공하는 두 개 이상의 원격 객체 구현 인스턴스를 생성해야 합니다. 예를 들면, 다음과 같습니다.

```
HelloImpl obj = new HelloImpl();
```

생성자는 원격 객체를 export하는데 이것은 원격 객체가 일단 생성되면 이와 동시에 수신되는 호출을 승인할 준비가 된다는 것을 의미합니다.

원격 객체 등록

클라이언트나 피어(peer) 또는 애플릿과 같은 호출자가 원격 객체의 메소드를 호출할 수 있도록 하려면 호출자는 우선 원격 객체에 대한 참조를 얻어야 합니다.

부트스트랩을 위해 RMI 시스템은 원격 객체에 "//host/objectname" 형식의 URL 포맷 이름(URL-formatted name)을 바인딩할 수 있도록 해주는 원격 객체 레지스트리를 제공하는데, 여기서 objectname은 단순한 문자열 이름입니다.

RMI 레지스트리는 원격 클라이언트가 원격 객체에 대한 참조를 얻을 수 있는 간단한 서버측 네임 서버입니다. RMI 레지스트리는 일반적으로 RMI 클라이언트가 말해야 하는 첫 번째 원격 객체를 찾기 위해 사용됩니다. 그러면 첫 번째 객체는 다른 객체를 찾는 데 사용되는 애플리케이션 특정 지원을 제공합니다.

예를 들어 참조는 다른 원격 메소드 호출로의 매개변수 또는 다른 원격 메소드 호출로부터의 반환값으로서 얻을 수 있습니다. 이 객체가 어떻게 작동하는지에 대해서는 http://java.sun.com/j2se/1.3/docs/guide/rmi/Factory.html의 *Applying the Factory Pattern to RMI*를 참조하십시오.

일단 원격 객체가 서버에 등록되면 호출자는 이름별로 객체를 조회하고 원격 객체 참조를 얻은 다음 객체의 메소드를 원격으로 호출합니다.

예를 들어 다음 코드는 "HelloServer" 이름을 원격 객체의 참조에 바인딩합니다.

```
Naming.rebind("//localhost/HelloServer", obj);
```

rebind() 메소드 호출 인수에 대한 다음 사항에 유의하십시오.

- 첫 번째 매개변수는 원격 객체의 위치와 이름을 나타내는 URL 포맷(URL-formatted) java.lang.String입니다.
  - 다음 예제에서 localhost는 개발 테스트에 사용됩니다. localhost 값을 서버 머신의 이름이나 IP 주소로 변경해야 합니다. 그렇지 않으면 호스트가 URL로부터 생략될 경우 기존의 호스트를 기본값으로 설정합니다. 어떠한 프로토콜도 URL에 지정할 필요가 없습니다. 그 예제는 다음과 같습니다. HelloServer.

- 선택적으로 포트 번호는 URL에서 부여할 수 있습니다. 예제는 다음과 같습니다. "//myhost:1234/HelloServer".포트는 1099를 기본값으로 합니다. 서버가 기본값 대신 포트에 레지스트리를 생성할 경우에만 포트 번호를 지정해야 합니다.

- 두 번째 매개변수는 원격 메소드를 호출하는 객체 구현 참조입니다.
- RMI 런타임은 원격 객체의 스텁에 대한 참조를 obj 인수가 지정하는 사실상의 원격 객체 참조로 대체합니다. HelloImpl의 인스턴스와 같은 원격 구현 객체는 가상 머신을 생성된 장소에 남겨두지 않으므로 클라이언트가 서버의 원격 객체 레지스트리에서 조회 작업을 수행할 경우에는 구현용 스텁을 가진 객체는 반환됩니다.

보안상의 이유로 애플리케이션은 같은 호스트에서 작동하는 레지스트리만을 바인딩하거나 바인딩 해제할 수 있습니다. 이로 인해 클라이언트는 서버의 원격 레지스트리의 어떠한 항목도 제거하거나 겹쳐쓸 수 없습니다. 그러나 조회는 어떤 호스트에서도 실행될 수 있습니다.

3 단계: 원격 서비스를 사용하는 클라이언트 프로그램 작성

배포용 "Hello World" 예제의 애플릿 부분은 원격으로 애플릿이 실행될 때 표시되는 "Hello World!" 문자열을 얻기 위해 HelloServer의 sayHello 메소드를 호출합니다. 다음과 같은 방법으로 애플릿을 생성합니다.

1 File|New를 선택합니다. 객체 갤러리의 Web 페이지에서 Applet 아이콘을 선택합니다. OK를 클릭하여 Applet 마법사를 표시합니다.

2 Class 필드를 HelloApplet으로 변경합니다.

3 Finish를 클릭합니다.

HelloApplet.java 파일과 HelloApplet.html 파일은 프로젝트 창에 표시됩니다. HelloApplet.java는 에디터에 표시됩니다.

4 **굵게** 표시된 코드를 HelloApplet.java에 추가합니다.

```
package hello;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloApplet extends Applet {
  boolean isStandalone = false;
  String message = "blank";
  //"obj" is the identifier that we'll use to refer
  //to the remote object that implements the "Hello"
  //interface
  Hello obj = null;
```

```
  /**Get a parameter value*/
  public String getParameter(String key, String def) {
    return isStandalone ? System.getProperty(key, def) :
      (getParameter(key) != null ? getParameter(key) : def);
  }

  /**Construct the applet*/
  public HelloApplet() {
  }

  /**Initialize the applet*/
 public void init() {
  try {
    jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  /**Component initialization*/
  private void jbInit() throws Exception {
  obj = (Hello)Naming.lookup("//" + getCodeBase().getHost()
      + "/HelloServer");
    message = obj.sayHello();
  }

  public void paint(Graphics g){
    g.drawString(message, 25, 50);
 }
  /**Get Applet information*/
  public String getAppletInfo() {
    return "Applet Information";
  }

  /**Get parameter info*/
  public String[][] getParameterInfo() {
  return null;
  }
}
```

5 File|Save All을 선택합니다.

애플릿은 다음 작업을 수행합니다.

1 먼저 애플릿은 서버 호스트의 RMI 레지스트리로부터 HelloServer로 알려진 원격 객체 구현에 대한 참조를 얻습니다. Naming.rebind() 메소드와 같이 Naming.lookup() 메소드는 URL 포맷(URL-formatted) java.lang.String을 얻습니다. 다음 예제에서 애플릿은 getHost() 메소드와 getCodeBase() 메소드를 함께 사용하여 URL 문자열을 만들 수 있습니다. Naming.lookup()은 다음과 같은 작업을 관리합니다.

- Naming.lookup()에 대한 인수로 제공된 포트 번호와 호스트 이름을 사용하여 서버의 레지스트리와 연락하기 위해 레지스트리 스텁 인스턴스를 만듭니다.

- 레지스트리 스텁을 사용하여 레지스트리의 URL의 이름 컴포넌트(HelloServer)를 사용하는 원격 lookup() 메소드를 호출합니다.
  - 레지스트리는 HelloImpl_Stub 인스턴스 바운드를 해당 이름으로 돌려 보냅니다.
  - 그러면 레지스트리는 원격 객체 구현(HelloImpl) 스텁 인스턴스를 수신하여 CLASSPATH나 스텁의 코드베이스(codebase)로부터 스텁 클래스(HelloImpl_Stub)를 로드합니다.
- Naming.lookup()은 스텁을 스텁 호출자(HelloApplet)에게 돌려 보냅니다.

2 애플릿은 서버의 원격 객체에 있는 원격 sayHello() 메소드를 호출합니다.

- RMI은 "Hello World!" 응답 문자열을 직렬화하여 반환합니다.
- RMI는 문자열을 직렬화 해제(deserialize)하고 해당 문자열을 message라는 변수에 저장합니다.

3 애플릿은 paint() 메소드를 호출하여 애플릿의 그리기 영역에 "Hello World!" 문자열이 표시되도록 합니다.

매개변수로서 Naming.lookup() 메소드로 전달된 생성된 URL 문자열에는 서버의 호스트 이름이 포함되어 있어야 합니다. 그러지 않을 경우 애플릿의 조회 시도는 클라이언트를 기본값으로 설정하고 애플릿이 로컬 시스템에 액세스 할 수 없는 대신 애플릿 호스트만을 갖는 통신에 제한되기 때문에 AppletSecurityManager는 예외를 발생시킵니다.

애플릿을 포함하는 HTML 파일의 이해

애플릿을 포함하는 HTML 파일은 전 단계에서 Applet 마법사에 의해 생성되었습니다. 이 파일을 HelloApplet.html이라고 합니다. 생성된 코드는 다음과 같습니다.

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>
HTML Test Page
</title>
</head>
<body>
hello.HelloApplet will appear below in a Java enabled browser.<br>
<applet
  codebase = "."
  code     = "hello.HelloApplet.class"
  name     = "TestApplet"
  width    = 400
  height   = 300
  hspace   = 0
  vspace   = 0
  align    = middle
>
</applet>
```

```
</body>
</html>
```

다음 사항에 유의하십시오.

- 클래스를 다운로드하려는 시스템에서 실행될 HTTP 서버가 있어야 합니다.
- 다음 예제에서 코드베이스는 웹 페이지가 스스로 로드된 디렉토리 아래의 디랙토리를 지정합니다. 이러한 상대적 경로를 사용하는 것이 좋습니다. 예를 들어 애플릿의 HTML가 참조하는 애플릿의 클래스 파일이 존재하는 codebase 디렉토리가 HTML 디렉토리 위의 디렉토리에 있다면 "../"와 같은 상대적인 경로를 사용하고자 할 것 있습니다.
- 애플릿의 code 속성은 애플릿의 완벽한 패키지 이름을 지정합니다. 위의 예제에서 code 속성은 다음과 같습니다. code="hello.HelloApplet.class"

4 단계: 정책 파일 작성

시스템에서 이 코드를 실행하려면 루트 프로젝트 디렉토리를 생성해야 합니다.

참고 다음 예제에서는 편의를 위해 모든 사람에게 전역적인 허용(global permission)를 제공하는 정책 파일을 사용하게 됩니다. **실제 환경에서는 이 정책 파일을 사용하지 마십시오.** . java.security.policy 파일을 사용하여 허가를 얻을 수 있는 적절한 방법에 대한 자세한 내용은 다음 문서를 참조하십시오.

- http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html의 *Default Policy Implementation 및 Policy File Syntax*
- http://java.sun.com/j2se/1.3/docs/guide/security/permissions.html의 *Permissions in JDK1.2*

다음과 같은 방법으로 정책 파일을 만듭니다.

1. 프로젝트 파일 Hello.jpx.를 마우스 오른쪽 버튼으로 클릭합니다.
2. Add Files/Packages를 선택합니다.
3. Add Files Or Packages To Project 대화 상자에서 프로젝트 (jbproject/Hello)의 루트를 선택합니다. (Project 버튼을 클릭하고 이 디렉토리로 빠르게 이동합니다.)
4. File Name 필드에 rmi.policy를 입력합니다.
5. OK를 클릭합니다. 대화 상자에서 파일 생성 여부를 묻습니다. OK를 다시 클릭합니다.
6. 프로젝트 창에서 rmi.policy 파일을 더블 클릭합니다. 에디터에 비어 있는 파일이 나타납니다.

7 다음의 코드를 입력합니다.

```
grant {
  // Allow everything for now
  permission java.net.SocketPermission  "*:1024-65535", "accept, connect, listen";
};
```

8 File|Save All을 선택합니다.

2과: 클래스 파일 및 HTML 파일 컴파일

"Hello World" 예제의 소스 파일이 이제 완성되어 프로젝트에 다음과 같은 파일들이 포함됩니다.

- `Hello.java` – 원격 인터페이스에 대한 소스 코드.
- `HelloImpl.java` – 원격 객체 구현에 대한 소스 코드. 이 프로젝트는 "Hello World" 서버에 대한 코드도 포함합니다.
- `HelloApplet.java` – 애플릿에 대한 소스 코드.
- `HelloApplet.html` – "Hello World" 애플릿을 참조하는 웹 페이지. Applet 마법사는 배포용 파일을 보다 쉽게 패키지하기 위해 이 파일을 `Hello/classes` 디렉토리에 둡니다.
- `rmi.policy` – 허가 파일

2과에서는 `.java` 소스 파일을 컴파일하여 `.class` 파일과 RMI 스텁을 생성합니다. 스텁은 원격 객체용 클라이언트측 프락시로서 실제 원격 객체 구현으로 호출을 번갈아 전달하는 서버측 디스패처(뼈대)로 호출을 전달합니다.

Java 소스 파일 컴파일

다음과 같은 방법으로 Java 소스 파일을 컴파일합니다.

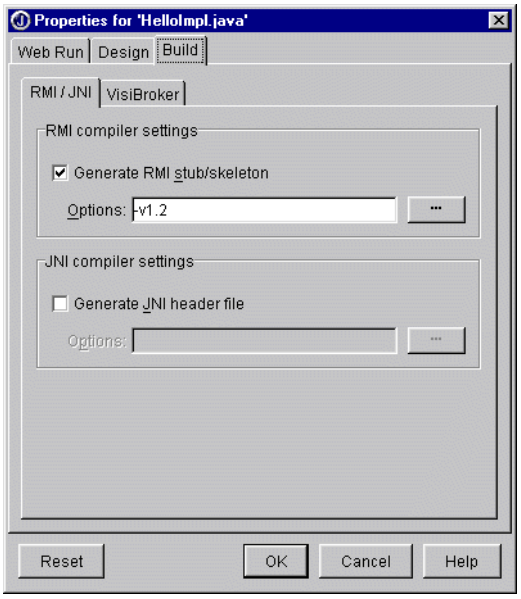
1 프로젝트 창에서 `HelloImpl.java`를 마우스 오른쪽 버튼으로 클릭합니다. 컨텍스트 메뉴에서 Properties를 선택합니다. Build 페이지를 선택합니다.

Properties 대화 상자가 표시됩니다.

2 Build 페이지의 RMI/JNI 탭에 있는 Generate RMI Stub/Skeleton 필드를 선택 표시합니다.

3 Option 필드에 `-v1.2`를 입력하고 OK를 클릭합니다. (Option 필드에 `-v1.2` 를 입력하지 않을 경우 컴파일 도중 경고가 나타납니다.)

Properties 대화 상자는 다음과 같습니다.



4 대화 상자를 닫으려면 OK를 클릭합니다.

5 `HelloImpl.java`를 마우스 오른쪽 버튼으로 클릭하고 Make를 선택합니다.

파일은 RMI 컴파일러인 `rmic`으로 컴파일됩니다. 생성된 파일 (`HelloImpl_Stub.java`)을 `classes/Generated Source` 디렉토리에 둡니다.

6 서버용 런타임 구성을 생성합니다.

1 Run|Configurations를 선택합니다.

Runtime Configurations 대화 상자가 표시됩니다.

2 New를 클릭합니다.

3 Runtime Properties 대화 상자의 Application 페이지의 Configuration Name 필드에 `Hello World Server`를 입력합니다.

4 Main 클래스 필드 옆에 있는 생략 버튼을 클릭합니다.

Select Main Class For Project 대화 상자가 표시됩니다.

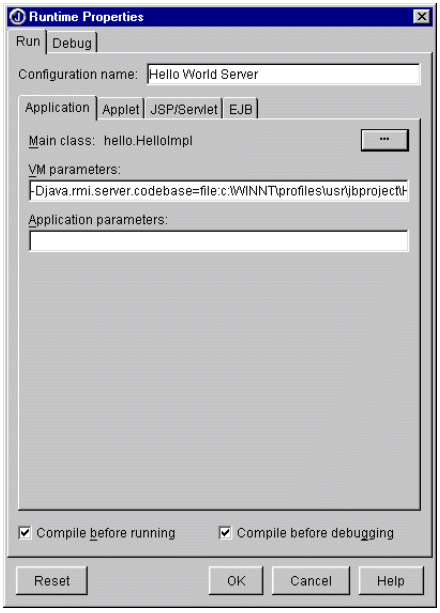
5 `hello` 패키지를 확장하고 `HelloImpl`을 선택합니다. OK를 클릭합니다.

6 프로젝트 디렉토리를 사용하여 다음과 유사한 매개변수를 Runtime Properties 대화 상자의 VM 매개변수 필드에 입력합니다.

```
-Djava.rmi.server.codebase=file:c:\WINNT\profiles\UserName\jbproject\
Hello\classes\-Djava.security.policy=file:c:\WINNT\profiles\UserName\
jbproject\Hello\rmi.policy
```

8.3 경로 이름이 있어야 한다는 것을 유의하십시오. 뿐만 아니라 `classes` 옵션과 `-Djava.security.policy` 사이에 백슬래시가 있어야 합

니다. java.rmi.server.codebase 매개변수는 서버의 클래스 파일 위치를 식별합니다. java.security.policy 매개변수는 보안 정책 파일 위치를 식별합니다. 위의 작업을 마치면 Runtime Configurations 대화 상자가 다음과 같이 보입니다.



7 OK를 두 번 클릭하여 Runtime Configuration과 Runtime Properties 대화 상자를 종료합니다.

7 File|Save All을 선택합니다.

8 Project|Make Project "Hello.jpx"를 선택합니다.

이전 단계는 파일에서 javac와 rmic 컴파일러를 실행하는 것과 유사합니다. 1.1 클라이언트의 Activatable 이외의 Unicast 원격 객체와 1.2 클라이언트의 모든 원격 객체로의 액세스를 지원하는 스텁과 뼈대를 만들어야 할 경우에는 명령줄의 rmic -vcompat를 실행하십시오.

rmic 옵션에 대한 설명은 http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/rmic.html의 Solaris rmic 매뉴얼 페이지 또는 http://java.sun.com/j2se/1.3/docs/tooldocs/win32/rmic.html의 Win32 rmic 매뉴얼 페이지를 참조하십시오.

생성된 스텁은 원격 객체와 똑같은 원격 인터페이스의 집합을 구현합니다. 이것은 클라이언트가 타입 변환 및 타입 선택 표시를 위해 자바 언어에 내장된 연산자를 사용할 수 있다는 것을 뜻합니다. 또 Java 플랫폼을 위해 쓰여진 원격 객체가 객체 지향 다형성을 지원한다는 것을 의미합니다.

3과: RMI 레지스트리, 서버 및 애플릿 시작

3과에서는 JBuilder에서 RMI 레지스트리를 시작하고 JBuilder에서 서버를 시작하며 기본 웹 서버인 Tomcat을 사용하여 애플릿을 실행합니다.

3과는 다음 단계를 설명합니다.

- RMI 레지스트리 시작
- 서버 시작
- 애플릿 실행

1 단계: RMI 레지스트리 시작

RMI 레지스트리는 원격 클라이언트가 원격 객체에 대한 참조를 얻을 수 있는 간단한 서버측 네임 서버입니다. RMI 레지스트리는 일반적으로 애플리케이션이 대화해야 하는 첫 번째 원격 객체를 찾는데만 사용됩니다. 이 객체는 다른 객체를 찾기 위한 애플리케이션 특정 지원을 교대로 제공합니다.

JBuilder가 Windows NT를 실행하는 기본 디렉토리가 아닌 디렉토리에 설치될 경우에는 Control Panel의 System Environment 속성을 편집함으로써 CLASSPATH를 수정해야 할 수도 있습니다.

다음과 같은 방법으로 서버의 레지스트리를 시작합니다.

1 Tools|RMIRegistry를 선택합니다.

이 단계에서는 어떠한 것도 출력하지 않으며 일반적으로 배경(background)에서 실행됩니다.

중요 원격 인터페이스를 변경할 때는 언제나 레지스트리를 중단하고 다시 시작하거나 원격 객체 구현에서 변경/추가된 원격 인터페이스를 사용해야 합니다. 그렇지 않을 경우 레지스트리의 클래스 바운드는 변경된 클래스와 일치하지 않게 됩니다.

다음과 같은 방법으로 레지스트리를 중단합니다.

1 Tools|RMIRegistry를 선택 해제합니다.

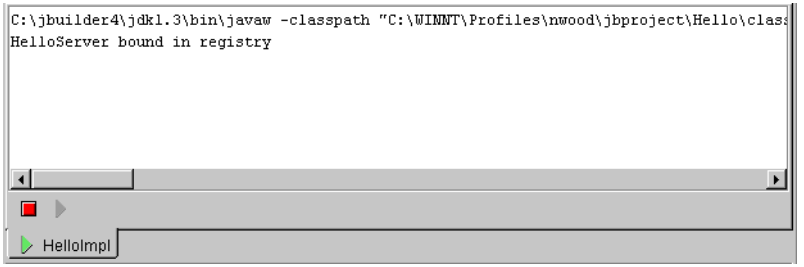
2 단계: 서버 시작

다음 단계를 따라 `HelloImpl` 서버를 시작합니다.

1 JBuilder 툴바의 Run 버튼 옆에 있는 아래쪽 화살표를 클릭합니다.

2 `Hello World Server`를 선택합니다

서버가 시작됩니다. "HelloServer bound in registry"가 메시지 창에 나타납니다.



참고 클래스를 아직 로컬로 사용할 수 없는 경우 **그리고** 클래스 파일이 서버에 상주하는 곳에 `java.rmi.server.codebase` 속성이 적절하게 설정된 경우에만 스텁 클래스가 라이언트의 가상 머신에 동적으로 다운로드됩니다. 서버가 작동할 경우에 JBuilder는 코드베이스 속성을 설정합니다.

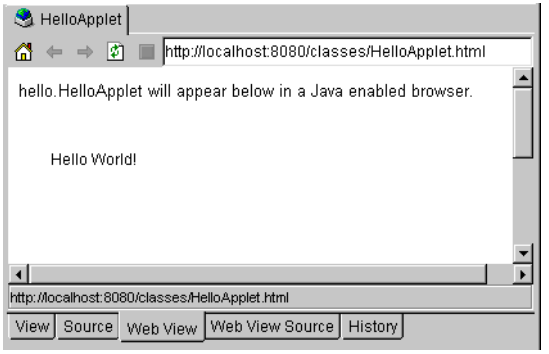
3 단계: 애플릿 실행

일단 레지스트리와 서버가 작동하면 애플릿도 실행될 수 있습니다. 애플릿은 Web 페이지(HTML 파일)를 브라우저로 로드하여 실행됩니다.

1 마우스 오른쪽 버튼으로 `HelloApplet.html`을 클릭합니다.

2 Web Run을 선택합니다.

"Hello World!" 애플릿이 Web View 창 안에서 실행되고 있는 상태에서 Tomcat Web 서버가 시작됩니다. "Hello World!" 애플릿이 올바르게 작동하는 경우에는 화면에 다음과 비슷하게 나타날 것입니다.



자습서를 모두 마쳤습니다.

Java RMI에 대해 더 자세히 알아보려면 다음 사이트를 참고하십시오.

- http://java.sun.com:80/products/jdk/rmi/의 *Java Remote Method Invocation*.
- http://java.sun.com:80/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html의 *Remote Method Invocations Specification*.

- http://java.sun.com:80/j2se/1.3/docs/guide/rmi/spec/rmi-arch.html#5498의 *RMI System Overview*.
- http://java.sun.com:80/marketing/collateral/javarmi.html의 *Java Remote Method Invocation - Distributed Computing for Java(a White Paper)*와 http://java.sun.com:80/marketing/collateral/javarmi.html#2의 *Advantages*.
- http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html에 있는 업데이트된 RMI에 관한 색인, *RMI Enhancements(for JDK 1.2)*.

Chapter

26

분산 애플리케이션 디버깅

이것은 JBuilder 기업용 버전의 기능입니다.

JBuilder에는 분산 애플리케이션을 디버깅하는 여러 가지 디버거 기능이 있습니다. 특히 크로스 프로세스 디버깅 및 원격 디버깅에 대한 지원이 포함됩니다.

이러한 지원은 JBuilder의 기본 디버깅 기능에 추가됩니다. JBuilder를 처음 사용하는 경우 JBuilder 디버거 환경에 대한 자세한 내용은 온라인 도움말 설명서 *JBuilder를 이용한 애플리케이션 구축*의 "Java 프로그램 디버깅"을 참조하십시오.

원격 디버깅

원격 디버깅은 한 컴퓨터에서 실행되는 코드를 다른 컴퓨터로부터 디버깅하는 프로세스입니다. 예를 들어, 이 기능은 다른 컴퓨터에 복제되지 않은 네트워크로 연결된 한 컴퓨터의 애플리케이션에 문제가 발생하는 경우에 적합합니다.

이 장에서 "클라이언트 컴퓨터"는 JBuilder를 실행하는 컴퓨터입니다. 즉, 디버깅을 실행하는 컴퓨터를 가리킵니다. "원격 컴퓨터"는 디버깅하려는 애플리케이션을 실행합니다.

원격으로 디버깅하는 방법은 두 가지입니다. 다음 두 가지 중 하나를 수행할 수 있습니다.

- 클라이언트 컴퓨터에서 원격 컴퓨터의 프로그램을 시작하고 클라이언트 컴퓨터에서 JBuilder를 사용하여 프로그램을 디버깅합니다. 자세한 내용은 26- 2페이지의 "원격 컴퓨터에서 프로그램 시작 및 디버깅"을 참조하십시오. 이 경우에는 원격 컴퓨터에서 JBuilder의 Debug Server를 실행해야 합니다.
- 원격 컴퓨터에서 이미 실행되고 있는 프로그램에 추가하고 클라이언트 컴퓨터에서 JBuilder를 사용하여 프로그램을 디버깅합니다. 자세한 내용은 26- 5페이지의 "원격 컴퓨터에서 이미 실행되고 있는 프로그램 디

버깅"을 참조하십시오. 이 경우에는 Debug Server를 실행할 필요가 없습니다.

참고 클라이언트 컴퓨터와 원격 컴퓨터 모두 JDK 1.3이나 JDK 1.2.2가 설치되어 있거나 JPDA 디버깅 API를 지원하는 JDK가 설치되어 있어야 합니다. 두 컴퓨터의 JDK 버전은 일치하지 않아도 됩니다. JBuilder를 설치하면 JDK 1.3은 자동으로 설치됩니다.

JBuilder가 설치되어 있는 같은 컴퓨터에서 별도의 프로세스로 실행되는 로컬 코드를 디버깅할 수도 있습니다. 이렇게 하려면 디버그 모드로 프로세스를 시작하고 JBuilder를 추가합니다. 자세한 내용은 26- 8페이지의 "별도의 프로세스로 실행되는 로컬 코드 디버깅"을 참조하십시오.

또한 클라이언트/서버 애플리케이션을 디버깅하는 데 이상적인 크로스 프로세스 브레이크포인트를 설정할 수 있습니다. 자세한 내용은 26- 9페이지의 "크로스 프로세스 브레이크포인트로 디버깅"을 참조하십시오.

Runtime Configurations 대화 상자를 사용하여 구성에 대해 개별적으로 원격 디버깅 옵션을 설정할 수도 있습니다. 자세한 내용은 *Building Applications with JBuilder*의 "런타임 구성 설정" 및 "디버거 구성 생성"을 참조하십시오.

원격 컴퓨터에서 프로그램 시작 및 디버깅

이 단원에서는 원격 컴퓨터에서 프로그램을 시작하고 클라이언트 컴퓨터에서 JBuilder를 사용하여 디버깅하는 방법에 대해 설명합니다. 즉, 다음과 같은 작업을 수행합니다.

1. 원격 컴퓨터에 Debug Server를 설치하고 실행합니다.
2. 원격 컴퓨터에서 애플리케이션을 컴파일하거나 애플리케이션의 .class 파일을 원격 컴퓨터로 복사합니다.
3. 클라이언트 컴퓨터의 JBuilder를 사용하여 원격 컴퓨터에서 애플리케이션을 시작하고 디버깅합니다.

중요 디버깅할 애플리케이션의 소스 파일을 클라이언트 컴퓨터에서 사용할 수 있어야 합니다. 컴파일된 .class 파일을 원격 컴퓨터에서 사용할 수 있어야 합니다. 파일들이 서로 일치해야 합니다. 그렇지 않으면 잘못된 오류가 생성되거나 잘못된 소스 코드 라인에서 디버거가 중지하는 등 예기치 못한 결과가 일어날 수도 있습니다. 소스 코드를 수정할 때마다 원격 컴퓨터에서 .class 파일을 업데이트해야 합니다.

먼저 원격 컴퓨터에 Debug Server를 설치하고 실행합니다. 원격 컴퓨터에 JBuilder가 이미 설치되어 있으면 아래의 4 단계부터 시작합니다.

1. JBuilder 설치 시의 `/remote` 디렉토리에 있는 `debugserver.jar` 파일을 원격 컴퓨터로 복사합니다. 다음 단계에서 필요하므로 파일을 복사한 디렉토리 위치는 기록해 두십시오.

2 Debug Server 셸 스크립트, DebugServer(Unix) 또는 배치 파일, DebugServer.bat(Windows)를 원격 컴퓨터의 같은 디렉토리로 복사합니다.

3 JDK 1.3이나 JDK 1.2.2가 원격 컴퓨터에 설치되어 있어야 합니다. JDK 1.3 다운로드 및 설치 지침을 보려면 http://java.sun.com/products/OV_jdkProduct.html의 드롭다운 목록에서 Java 2 SDK, Standard Edition, V 1.3을 선택합니다.

4 Debug Server 파일이 설치되어 있는 원격 컴퓨터의 디렉토리로 이동합니다. DebugServer를 실행하여 원격 Debug Server의 환경 변수를 사용자 지정하고 실행합니다.

   Unix 시스템의 경우에는 다음 명령을 사용하십시오.

   ```
   ./DebugServer [debugserver.jar_dir] [jdk_home_dir]
   [-port=<portnumber>]  [-timeout=<milliseconds>]
   ```

   Windows 시스템의 경우에는 다음 명령을 사용하십시오.

   ```
   DebugServer [debugserver.jar_dir] [jdk_home_dir]
   [-port=<portnumber>]  [-timeout=<milliseconds>]
   ```

   여기서,

   - debugserver.jar_dir–Debug Server JAR 파일이 있는 원격 컴퓨터의 디렉토리입니다. Windows 시스템의 경우 드라이브 문자가 필요합니다.
   - jdk_home_dir–원격 컴퓨터에 있는 JDK 설치의 홈 디렉토리입니다. Windows 시스템의 경우 드라이브 문자가 필요합니다.
   - -port–기본값 18699이 아닌, 다른 포트에서 디버그 서버를 시작하는 선택 매개변수입니다. 기본값을 사용하고 있는 경우에만 이 값을 변경하십시오. 유효한 값은 1025에서 65535 사이입니다. 이 값은 클라이언트 컴퓨터의 Project Properties 대화 상자에서 Debug 페이지의 Port Number 필드에 입력한 값과 일치해야 합니다. 아래의 6 단계를 참조하십시오.
   - -timeout–클라이언트 컴퓨터에 원격 컴퓨터를 연결하려고 시도하는 수를 밀리초 단위로 설정하는 선택 매개변수입니다. 이 숫자에 도달하면 프로세스는 중단됩니다. 기본 설정은 60,000밀리초입니다.

   다음은 Windows 환경에서 사용되는 이 명령의 예입니다.

   ```
   DebugServer d:\remote d:\jdk1.3 -port=1234 -timeout=20000
   ```

Debug Server를 로드하면 시작 모드로 JBuilder 원격 디버깅 기능을 사용합니다. Debug Server가 실행 중이면 애플리케이션을 컴파일하고 .class 파일을 원격 컴퓨터로 복사해야 합니다. 애플리케이션을 원격으로 컴파일할 수도 있습니다. 그런 다음 클라이언트 컴퓨터에서 실행되는 JBuilder를 사용하여 원격 컴퓨터에서 프로그램을 시작하고 디버깅합니다.

1 애플리케이션 컴파일 클라이언트 컴퓨터에서 JBuilder를 사용하여 애플리케이션을 컴파일한 다음 FTP(File Transfer Protocol)를 복사하거나 사용하여 .class 파일을 원격 컴퓨터로 전달합니다. javac 컴파일러

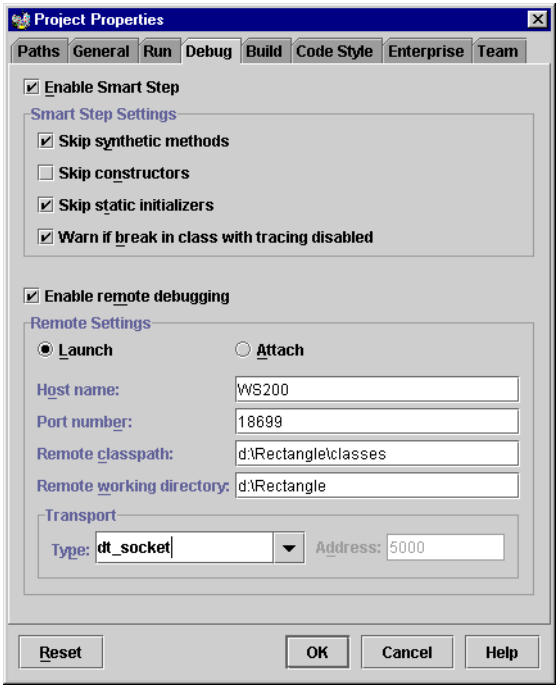
를 호출할 때 -g 옵션을 사용하여 원격 컴퓨터에서 직접 애플리케이션을 컴파일할 수도 있습니다.

2 클라이언트 컴퓨터에서 JBuilder를 엽니다.

3 애플리케이션이 디버깅할 프로젝트를 엽니다.

4 Project|Project Properties를 선택하여 Project Properties 대화 상자를 연 다음 Debug 탭을 선택합니다.

5 Enable Remote Debugging 체크 박스를 선택합니다. Launch 옵션을 선택합니다.



6 다음 필드에 입력합니다.

- Host Name-원격 컴퓨터의 이름입니다. localhost가 기본값입니다. 호스트 이름을 찾으려면 원격 컴퓨터에서 네트워크 설정을 확인해야 합니다.
- Port Number-통신하는 원격 컴퓨터의 포트 번호입니다. 기본 포트 번호인 18699를 사용합니다. 기본값을 사용하고 있는 경우에만 이 값을 변경하십시오. 유효한 값은 1024에서 65535 사이입니다. 이 값은 원격 컴퓨터에 있는 Debug Server의 -port 매개변수와 일치해야 합니다. 앞 단원의 4 단계를 참조하십시오.
- Remote Classpath-원격으로 디버깅하는 애플리케이션의 컴파일된 .class 파일이 있는 classpath입니다. 이 필드는 다른 classpath 필드처럼 작동합니다. 클래스가 패키지에 있으면 클래스를 포함하는 디렉토리가 아닌 패키지의 루트를 지정합니다. Windows 시스템을

사용하는 경우 드라이브 문자를 지정하십시오. 이 원격 classpath는 이 디버깅 세션에만 적용됩니다.

- Remote Working Directory-원격 컴퓨터의 작업 디렉토리입니다. Windows 시스템을 사용하는 경우 드라이브 문자를 지정하십시오. 이 원격 작업 디렉토리는 이 디버깅 세션에만 적용됩니다.

경고 작업 디렉토리는 JDK 1.2.2에서 지원되지 않습니다. 원격 컴퓨터가 JDK 1.2.2에서 실행되는 경우 원격 작업 디렉토리를 입력하면 디버거는 콘솔 출력, 입력 및 오류 보기에 경고를 표시합니다.

- Transport-전송 유형은 dt_shmem(Unix 시스템에서는 사용되지 않는 공유 메모리 전송)와 dt_socket(소켓 전송) 중 하나입니다. 전송 유형에 관한 자세한 내용은http://java.sun.com/products/jpda/doc/conninv.html#Transports에서 "JPDA:Connection and Invocation Details - Transports"를 참조하십시오.

7 OK를 클릭합니다.

8 다음 옵션 중 하나를 선택하여 디버깅 세션을 시작합니다.

명령	단축키	설명
Run|Debug Project	*Shift + F9*	디버거에서 프로그램을 시작하며, 프로그램을 실행하여 완료하거나 사용자 입력이 필요한 코드의 첫 줄이나 브레이크포인트에서 실행을 일시 중지(suspend)합니다.
Run|Step Over	*F8*	실행 코드의 첫 번째 줄에서 실행을 일시 중지(suspend)합니다.
Run|Step Into	*F7*	실행 코드의 첫 번째 줄에서 실행을 일시 중지(suspend)합니다.

디버거를 시작하면 Remote Classpath 설정을 기반으로 디버깅하려는 애플리케이션이 원격 컴퓨터에서 시작됩니다. 디버거는 클라이언트 컴퓨터에서 실행되는 JBuilder에 표시되지만 원격 컴퓨터에서 실행되는 .class 파일을 디버깅합니다.

참고 애플리케이션이 원격 컴퓨터에서 이미 실행되고 있으면 Debug Server는 새 인스턴스를 시작합니다. 이미 실행 중인 애플리케이션을 디버깅하려면 26- 5페이지의 "원격 컴퓨터에서 이미 실행되고 있는 프로그램 디버깅"을 참조하십시오.

1 원격 컴퓨터에서 애플리케이션을 종료하려면 JBuilder의 프로세스를 중단합니다. 원격 컴퓨터에서 Debug Server를 닫으려면 Debug Server의 File|Exit 명령을 선택합니다.

원격 컴퓨터에서 이미 실행되고 있는 프로그램 디버깅

이 단원에서는 원격 컴퓨터에서 이미 실행되고 있는 프로그램에 추가하고 클라이언트 컴퓨터에서 JBuilder를 사용하여 디버깅하는 방법에 대해 설명합니다. 즉, 다음 작업을 수행해야 합니다.

1. 원격 컴퓨터에서 VM 디버그 옵션으로 애플리케이션을 실행합니다.
2. 클라이언트 컴퓨터에서 JBuilder를 사용하여 실행 중인 애플리케이션에 추가하고 디버깅합니다.

중요 디버깅할는 애플리케이션의 소스 파일을 클라이언트 컴퓨터에서 사용할 수 있어야 합니다. 컴파일된 .class 파일을 원격 컴퓨터에서 사용할 수 있어야 합니다. 파일들이 서로 일치해야 합니다. 그렇지 않으면 잘못된 오류가 생성되거나 잘못된 소스 코드 라인에서 디버거가 중지하는 등 예기치 못한 결과가 일어날 수도 있습니다. 소스 코드를 수정할 때마다 원격 컴퓨터에서 .class 파일을 업데이트해야 합니다.

이미 실행 중인 프로그램에 추가하는 단계를 설명한 자습서는 Chapter 27 "원격 디버깅 자습서"를 참조하십시오.

다음과 같은 방법으로 원격 컴퓨터에서 프로그램을 시작하고 추가합니다.

1. 원격 컴퓨터에서 애플리케이션을 컴파일합니다. 클라이언트 컴퓨터에서 JBuilder를 사용하여 애플리케이션을 컴파일한 다음 FTP(File Transfer Protocol)를 복사하거나 사용하여 .class 파일을 원격 컴퓨터로 전달합니다.
2. 다음 VM 옵션을 사용하여 원격 컴퓨터에서 애플리케이션을 실행합니다.
   - 원격 컴퓨터에 JBuilder가 있으면 JBuilder 내에서 프로그램을 실행할 수 있습니다. 프로젝트를 연 다음 Project Properties 대화 상자의 Run 페이지를 엽니다. VM Parameters 필드에 다음 매개변수를 입력합니다.

     ```
     -classic -Xdebug -Xnoagent -Djava.compiler=NONE
     -Xrunjdwp:transport=dt_socket, server=y,address=5000,suspend=y
     ```

     `address` 및 `suspend`는 옵션입니다. 자세한 내용은 아래를 참조하십시오.
   - 원격 컴퓨터에 JBuilder가 없으면 명령줄에서 프로그램을 실행해야 합니다. Java 명령줄에 다음 VM 옵션을 추가합니다.

     ```
     -Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:transport=dt_socket,
     server=y,address=5000,suspend=y
     ```

     `address` 및 `suspend` 매개변수는 옵션입니다. 두 매개변수는 `server` 매개변수 다음에 오고 쉼표로 구분됩니다. 매개변수 사이에 공백은 허용되지 않습니다.
     - 선택한 전송을 기반으로 하는 `address` 매개변수는 디버거가 원격 컴퓨터와 통신하는 포트 번호/주소를 저장합니다. 이 매개변수를 사용하면 더 쉽게 구성할 수 있습니다. Project Properties 대화 상자의 Debug 페이지에서 Address 필드를 계속해서 수정하지 않아도 됩니다. Transport Type을 dt_socket으로 설정하면 `address` 매개변수는 포트 번호를 저장합니다. dt_shmem으로 설정하면 이 매개변수는 고유한 주소 이름으로 설정됩니다.

- suspend 매개변수는 프로그램이 시작되는 즉시 일시 중지할지 묻습니다. suspend=n으로 지정하면 이 옵션을 해제할 수 있습니다. suspend=n 및 브레이크포인트 없음으로 설정하면 프로그램을 시작하여 중지하지 않고 실행 완료됩니다.

참고 애플리케이션을 실행하려면 /jre/bin 디렉토리의 java.exe가 아닌 JDK의 /bin 설치 디렉토리에서 java 실행 파일을 사용합니다. 이렇게 하면 Java VM이 디버깅에 필요한 디버거 파일(Unix의 경우 libjdwp.so, Windows의 경우 jdwp.dll)을 로드할 수 있습니다.

3 클라이언트 컴퓨터에서 JBuilder를 엽니다.

4 원격 컴퓨터에서 이미 실행되고 있는 애플리케이션의 프로젝트를 엽니다.

5 Project|Project Properties를 선택하여 Project Properties 대화 상자를 연 다음 Debug 탭을 선택합니다.

6 Enable Remote Debugging 체크 박스를 선택합니다. 체크 박스 아래에 있는 Attach 옵션을 선택합니다.

7 다음 필드에 입력합니다.

- Host Name-원격 컴퓨터의 이름입니다. localhost가 기본값입니다. 호스트 이름을 찾으려면 원격 컴퓨터에서 네트워크 설정을 확인해야 합니다.
- Transport-전송 방식 옵션입니다.

- 타입: dt_socket(소켓 전송)과 dt_shmem(Unix 시스템에서는 사용되지 않는 공유 메모리 전송) 중 하나입니다. 전송 방식에 관한 자세한 내용은http://java.sun.com/products/jpda/doc/conninv.html#Transports에서 "JPDA:Connection and Invocation Details - Transports"를 참조하십시오.
- Address
  - Transport Type을 dt_socket으로 설정하면 이 매개변수는 통신하는 원격 컴퓨터의 포트 번호를 저장합니다. 기본 포트 번호인 5000를 사용합니다. 기본값을 사용하고 있는 경우에만 이 값을 변경하십시오. 이 값은 원격 컴퓨터에서 프로그램을 시작하는 Java VM의 `address` 매개변수와 일치해야 합니다. 이 단원 앞 부분의 2 단계를 참조하십시오.
  - Transport Type으로 dt_shmem을 선택하면 통신하는 원격 컴퓨터의 고유한 이름으로 `address` 매개변수를 설정합니다. 기본값은 javadebug입니다.

8 대화 상자를 닫으려면 OK를 클릭합니다.

9 Run|Step Over 또는 Run|Step Into를 선택하여 디버거를 시작합니다.

10 원격 컴퓨터에 있는 VM의 `suspend` 매개변수를 `y`(이 단원 앞 부분의 2 단계 참조)로 설정하면 디버그 툴바의 Resume Program 버튼을 클릭하여 디버깅을 계속합니다.

11 애플리케이션을 종료하려면 원격 컴퓨터에서 애플리케이션을 닫습니다.

12 원격 컴퓨터에서 분리하려면 JBuilder의 프로세스를 중단합니다.

참고 원격 컴퓨터에서 애플리케이션을 시작하고 디버깅하려면 26- 2페이지의 "원격 컴퓨터에서 프로그램 시작 및 디버깅" 단원을 참조하십시오.

별도의 프로세스로 실행되는 로컬 코드 디버깅

JBuilder가 설치되어 있는 같은 컴퓨터에서 별도의 프로세스로 실행되는 로컬 코드를 디버깅하려면 앞에서 설명한 2 단계부터 시작합니다. Project Properties 대화 상자에 있는 Debug 페이지의 Attach 옵션에 다음 설정을 사용합니다.

Host Name	기본값, localhost로 설정합니다.
Transport Type	dt_socket(소켓 전송)이나 dt_shmem(Unix 시스템에서는 사용되지 않는 공유 메모리 전송)으로 설정합니다.
Transport Address	Transport Type이 dt_socket이면 5000으로 설정하고 dt_shmem이면 javadebug로 설정합니다.

크로스 프로세스 브레이크포인트로 디버깅

크로스 프로세스 브레이크포인트는 별도의 프로세스에 지정된 클래스의 모든 메소드나 지정된 메소드를 한 단계씩 실행할 때 디버거가 중단되게 합니다. 이렇게 하면 클라이언트와 서버에 브레이크포인트를 설정하지 않고 클라이언트 프로세스에서 서버 프로세스를 한 단계씩 실행할 수 있습니다. 일반적으로 클라이언트에는 라인 브레이크포인트를 설정하고 서버에는 크로스 프로세스 브레이크포인트를 설정합니다. 크로스 프로세스 단계를 설명하는 자습서는 Chapter 27 "원격 디버깅 자습서"를 참조하십시오.

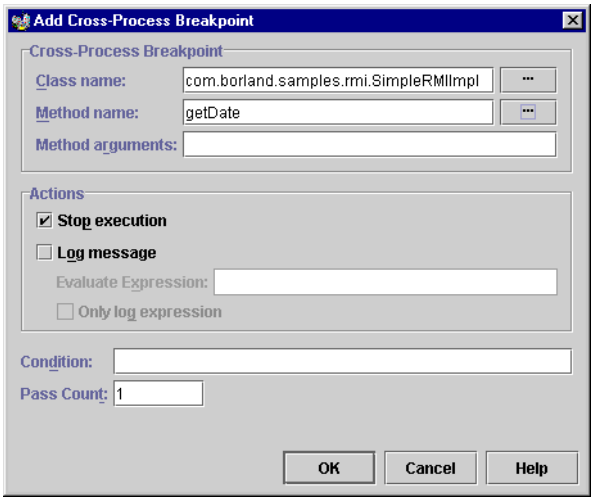
다음과 같은 방법으로 서버 프로세스에 설정된 크로스 프로세스 브레이크포인트를 활성화합니다.

1. 원격 컴퓨터에서 디버그 모드로 서버 프로세스를 시작합니다. 26- 5페이지의 "원격 컴퓨터에서 이미 실행되고 있는 프로그램 디버깅" 단원의 2 단계를 참조하십시오.
2. 원격 컴퓨터에서 이미 실행되고 있는 서버에 클라이언트 컴퓨터에 의 JBuilder 내에서 추가합니다. 26- 5페이지의 "원격 컴퓨터에서 이미 실행되고 있는 프로그램 디버깅" 단원의 4-8 단계를 참조하십시오.
3. 클라이언트 코드에 라인 브레이크포인트를 설정하고 클라이언트 디버깅을 시작합니다. 브레이크포인트에서 서버 코드를 한 단계씩 실행합니다. Step Over는 사용하지 마십시오. 프로시저 단위로 실행하면 크로스 프로세스 브레이크포인트에서 중단되지 않습니다.

크로스 프로세스 브레이크포인트를 설정하려면 Add Cross-Process Breakpoint 대화 상자를 사용합니다. Add Cross-Process Breakpoint 대화 상자를 열려면 다음 중 하나를 수행합니다.

- 디버깅 세션 이전이나 도중에 Run|Add Breakpoint를 선택하고 Add Cross Process Breakpoint를 선택합니다.
- 디버깅 세션에 있으면 디버거 툴바의 Add Breakpoint 버튼()의 오른쪽에 있는 아래쪽 화살표를 클릭하여 Add Cross-Process Breakpoint를 선택합니다.
- 디버깅 세션에 있으면 Data의 빈 영역과 코드 브레이크포인트 보기를 마우스 오른쪽 버튼으로 클릭하고 Add Cross-Process Breakpoint를 선택합니다.

Add Cross-Process Breakpoint 대화 상자가 표시됩니다.



다음과 같은 방법으로 크로스 프로세스 브레이크포인트를 설정합니다.

1 Class Name 필드에 디버거가 중단시킬 메소드를 포함하고 있는 서버 측 클래스의 이름을 입력합니다. Browse 버튼을 사용하여 클래스를 찾아봅니다.

2 Method 필드에 디버거가 중단시킬 메소드 이름을 입력합니다. Browse 버튼을 사용하여 선택한 클래스에 사용 가능한 메소드를 찾아볼 수 있는 Select Method 대화 상자를 표시합니다. 메소드 이름은 필요하지 않습니다. 메소드 이름을 지정하지 않으면 디버거는 지정한 클래스에서 모든 메소드 호출을 중단시킵니다.

참고 선택한 클래스에 구문 오류나 컴파일러 오류가 포함되어 있으면 메소드를 선택할 수 없습니다.

3 Method Arguments 필드에 쉼표로 구분된 메소드 인수 목록을 입력합니다. 디버거는 메소드 이름 및 인수 목록이 일치할 때 중단됩니다. 이것은 오버로드된 메소드에 대해 유용합니다.

- 인수를 지정하지 않으면 디버거는 메소드 이름이 지정된 모든 메소드에서 중단됩니다.
- Select Method 대화 상자에서 메소드 이름을 선택하면 Methods Argument 필드는 자동으로 입력됩니다.

4 디버거에 대한 Actions를 선택합니다. 디버거는 브레이크포인트에서 실행을 중단시킬 수 있으며 메시지를 표시하거나 표현식을 계산할 수 있습니다. 자세한 내용은 *JBuilder를 이용한 애플리케이션 구축*의 "Java 프로그램 디버깅"장에서 "브레이크포인 동작 설정"을 참조하십시오.

5. Condition 필드에 브레이트포인트의 조건이 있으면 그 조건을 설정합니다. 자세한 내용은 *JBuilder를 이용한 애플리케이션 구축*의 "Java 프로그램 디버깅"장에서 "브레이크포인트 생성"을 참조하십시오.
6. Pass Count 필드에서 브레이크포인트를 활성화하기 위해 브레이크포인트를 통과해야 하는 횟수를 설정합니다. 자세한 내용은 *JBuilder를 이용한 애플리케이션 구축*의 "Java 프로그램 디버깅"장에서 "횟수 브레이크포인트 사용"을 참조하십시오.
7. 대화 상자를 닫으려면 OK를 클릭합니다.
8. 크로스 프로세스 브레이크포인트를 호출하는 메소드의 클라이언트에 라인 브레이크포인트를 설정합니다.
9.  라인 브레이크포인트에서 중단하면 디버거 툴바에서 Step Into 버튼을 클릭하여 서버측 브레이크포인트 메소드를 한 단계씩 실행합니다. Step Over를 사용하면 디버거는 중단되지 않습니다.

Chapter

27

원격 디버깅 자습서

자습서 정보

이 자습서는 JBuilder 기업용 버전의 기능 중 하나입니다.

이 단계별 자습서는 다음과 같은 작업 수행 방법을 보여 줍니다.

- 원격 디버깅 기능을 사용하여 원격 컴퓨터에서 이미 실행되고 있는 프로그램에 추가합니다.
- 크로스 프로세스 단계를 사용하여 디버깅합니다.
- 미리 설정된 구성을 사용하여 클라이언트와 서버 프로세스를 모두 디버깅합니다.

자습서는 JBuilder `samples\RMI` 설치 폴더에 제공된 예제 프로젝트를 사용합니다. 예제는 JBuilder로 만든 RMI 애플리케이션입니다. 이 자습서를 실행하기 전에, `samples` 폴더를 설치했는지 확인합니다.

이 자습서는 다음과 같이 가정합니다.

- Windows 컴퓨터를 사용한다고 가정합니다.
- 컴파일, 실행 및 디버깅에 익숙하다고 가정합니다. 그렇지 않은 경우 "컴파일, 실행 및 디버깅" 자습서를 학습해야 합니다. 또 온라인 도움말 설명서, *JBuilder를 이용한 애플리케이션 구축*에서 다음 장을 읽어보시면 됩니다.
  - Java 프로그램 컴파일
  - Java 프로그램 실행
  - Java 프로그램 디버깅
- JBuilder의 클라이언트/서버 프로세스에 익숙하다고 가정합니다.
- Chapter 26 "분산 애플리케이션 디버깅"을 읽은 것으로 가정합니다.
- DOS 창 및 명령줄에서의 명령 실행에 익숙하다고 가정합니다.

이 자습서를 실행하려면 다음 사항을 갖추어야 합니다.

- 네트워크에서 실행되는 컴퓨터 두 대. 한 대에는 JBuilder가 설치되고 다른 한 대에는 JDK 1.3이 설치되어 있어야 합니다. 이 자습서에서 JBuilder가 설치된 컴퓨터는 "클라이언트" 컴퓨터라고 하고, JDK만 설치된 컴퓨터는 "원격" 컴퓨터라고 합니다. 이 컴퓨터는 서버를 실행합니다.
- 원격 컴퓨터의 네트워크 ID. 일반적으로 이 ID는 네트워크 관리자가 설정합니다.
- 클라이언트 컴퓨터에서 원격 컴퓨터로 파일을 전송하는 방법.

참고 디버깅하지 않고 예제를 실행하려면 프로젝트의 HTML 파일, `SimpleRMI.html`의 지침을 수행합니다.

팁 이 자습서를 인쇄하려면 Help Viewer에서 자습서를 열고 File|Print를 선택합니다. JBuilder CD에 있는 `pdf/enterprise.pdf/`의 3부와 JBuilder 웹 사이트, http://www.borland.com/techpubs/jbuilder/에 있는 PDF *Distributed Application Developer's Guide*에서도 보실 수 있습니다.

1 단계: 예제 프로젝트 열기

이 자습서는 JBuilder 설치의 `samples/RMI` 폴더에 제공되는 예제 프로젝트를 사용합니다. 이 자습서를 실행하기 전에, `samples` 폴더를 설치했는지 확인합니다.

이 단계에서 프로젝트 파일을 엽니다. 다음과 같은 방법으로 예제 프로젝트를 엽니다.

1. File|Open Project를 선택합니다. Open Project 대화 상자가 표시됩니다.
2. JBuilder 설치의 `samples/RMI` 폴더로 이동합니다.
3. `SimpleRMI.jpr`을 더블 클릭합니다. 프로젝트 창에 프로젝트가 열립니다. 프로젝트의 파일이 프로젝트 창에 나열됩니다. 이 프로젝트는 다음과 같은 여섯 개의 파일로 구성됩니다.
   - `SimpleRMI.html` – 프로젝트에 대한 개요를 제공하는 HTML 파일입니다. 이 파일은 JBuilder에서 RMI 애플리케이션을 만들고 실행하는 것에 관한 지침을 제공합니다.
   - `SimpleRMI.policy` – 보안 정책 파일입니다. 이 파일은 네트워크를 통해 클라이언트 요청을 수신 대기하고 받아들이는 RMI 서버의 권한을 지정합니다.
   - `SimpleRMIClient.java` – 서버 객체에 연결되는 클라이언트 클래스입니다.

- SimpleRMIImpl.java – RMI 서버 인터페이스를 구현하는 클래스입니다.
- SimpleRMIInterface.java – RMI 인터페이스입니다.
- SimpleRMIServer.java – Impl 클래스의 인스턴스를 만드는 서버 클래스입니다.

2 단계에서는 클라이언트와 서버 런타임 및 디버그 구성을 설정합니다.

2 단계: 런타임 및 디버그 구성 설정

이 단계에서는 클라이언트와 서버의 런타임 및 디버그 구성을 설정합니다. 매개변수는 한 번만 설정해야 하므로 미리 설정된 매개변수를 사용하여 실행하고 디버깅하는 시간을 저장합니다. 미리 설정된 구성으로 애플리케이션을 실행하거나 디버깅할 때마다 원하는 구성을 선택합니다. 이 자습서에 사용할 구성을 설정하려면 다음 표에 나열된 대화 상자 페이지를 사용합니다.

Table 27.1 클라이언트와 서버 런타임 및 디버그 구성 설정에 사용하는 대화 상자 페이지

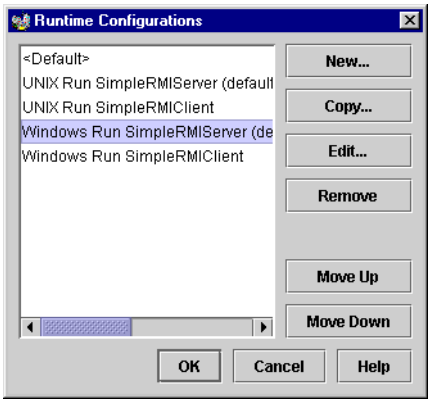
Runtime Properties 대화 상자 페이지	적용 대상	설명
Run 페이지	원격 컴퓨터에서 실행되는 서버	RMI 서버의 run 매개변수를 구성합니다.
Debug 페이지	원격 컴퓨터에서 실행되는 서버	클라이언트 컴퓨터의 서버가 원격 컴퓨터 프로세스에 추가되는 방법을 구성합니다.
Run 페이지	JBuilder를 이용한 컴퓨터에서 실행되는 클라이언트	RMI 클라이언트의 run 매개변수를 구성합니다.

참고 Project Properties 대화 상자에서 Run 또는 Debug 페이지의 설정은 변경하지 않아도 됩니다. Runtime Properties 대화 상자에서 Run 및 Debug 페이지의 설정만 변경합니다(Run|Configurations).

다음과 같은 방법으로 서버의 런타임 구성을 설정합니다.

1 Run|Configurations를 선택합니다. Runtime Configurations 대화 상자가 표시됩니다.

2 Windows Run SimpleRMIServer라는 구성을 선택합니다.



3 Edit를 눌러 Runtime Properties 대화 상자의 Run 페이지를 표시합니다.

4 Application 탭에서 Java VM으로 전달되는 매개변수를 조정합니다. VM Parameters 필드의 시작 부분에 다음 명령을 입력하고 그 다음에 공백을 입력합니다.

```
-classic
```

이 인수는 디버거가 HotSpot VM 대신 Classic VM을 사용하도록 지시합니다.

5 VM Parameters codebase 인수는 서버 클래스 파일의 위치를 가리켜야 합니다. Windows에서는 일반적으로 설치하는 위치가 samples\RMI 폴더의 classes 폴더입니다.

```
-Djava.rmi.server.codebase=file:C:\JBuilder\samples\RMI\classes\
```

참고 인수에서 classes 항목 다음에 오는 마지막 백슬래시는 필수입니다.

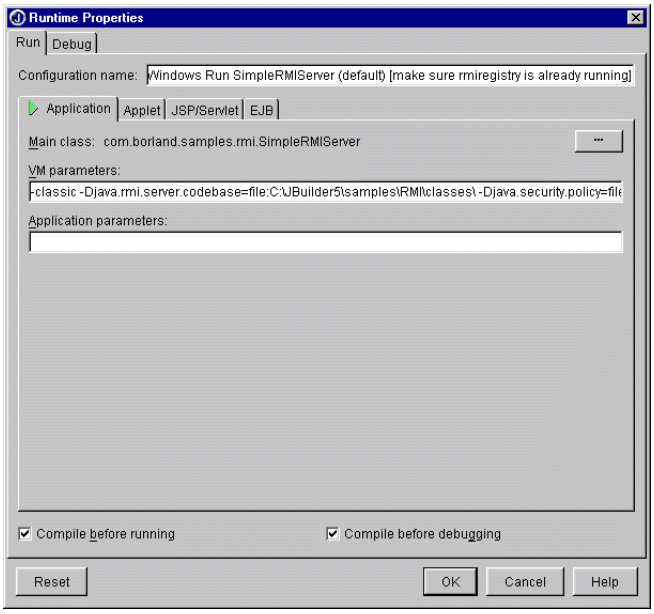
6 VM Parameters 필드의 보안 정책 인수는 보안 정책 파일의 위치를 가리켜야 합니다. 정책 파일은 네트워크를 통해 RMI 클라이언트 요청을 수신 대기하고 받아 들이는 RMI 서버의 권한을 지정합니다. Windows에서는 일반적으로 설치하는 위치가 samples\RMI 폴더입니다.

```
-Djava.security.policy=file:C:\JBuilder\samples\RMI\SimpleRMI.policy
```

7 메인 클래스가 다음과 같이 설정되었는지 확인합니다.

```
com.borland.samples.rmi.SimpleRMIServer
```

8 완료되면 서버의 Run 페이지는 다음과 같이 나타납니다.



다음과 같은 방법으로 서버의 원격 디버깅 구성을 설정합니다.

1 Debug 탭을 클릭합니다.

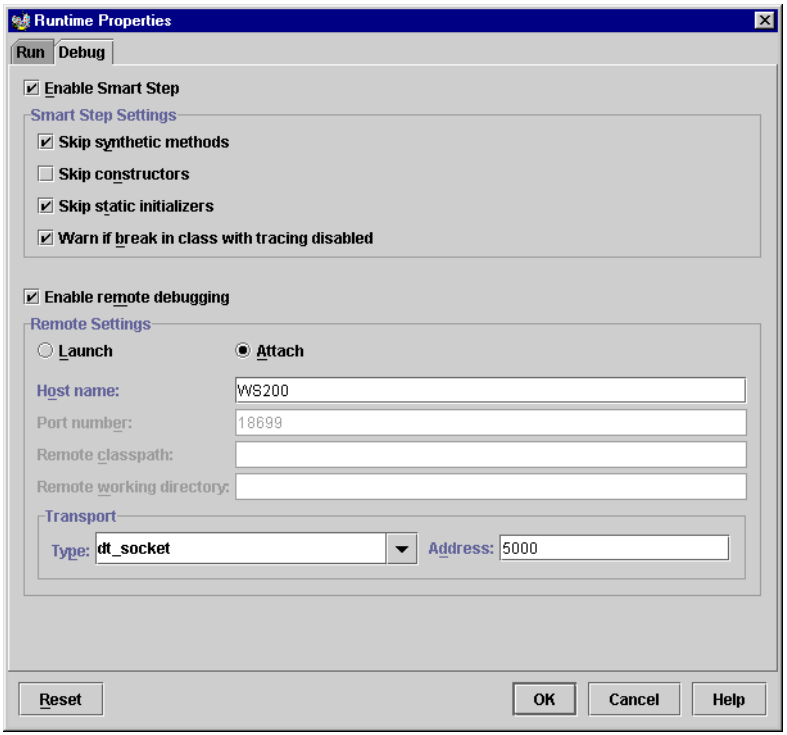
2 Enable Remote Debugging 옵션을 클릭한 다음 Attach 옵션을 클릭합니다.

3 Host Name 필드에 서버가 실행될 컴퓨터 이름을 입력합니다.

4 Transport Type을 `dt_socket`으로 그대로 둡니다.

5 Address 필드에 원격 컴퓨터의 주소를 입력합니다. 원격 컴퓨터에서 서버를 실행할 때 이 번호를 다시 사용하게 됩니다. 27- 11페이지의 "5 단계: 원격 컴퓨터에서 RMI Registry 및 서버 시작"을 참조하십시오. 이 자습서의 용도를 위해 설정을 `5000`으로 그대로 둡니다.

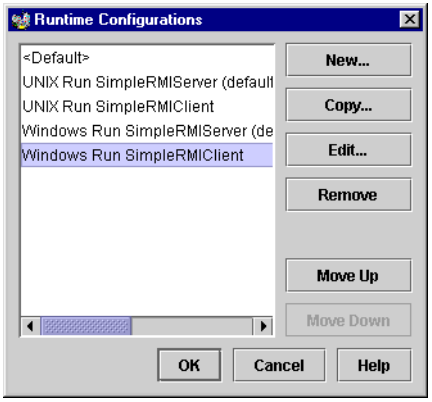
6 완료되면 서버의 Debug 페이지는 다음과 같이 나타납니다.



7 OK를 클릭하여 서버의 Runtime Properties 대화 상자를 닫습니다.

다음으로 클라이언트의 런타임 구성을 설정합니다.

1 Runtime Configurations 대화 상자에서 `Windows Run SimpleRMIClient`라는 구성을 선택합니다.



2 Edit를 눌러 Runtime Properties 대화 상자의 Run 페이지를 표시합니다.

3 Application 페이지에서 Java VM으로 전달되는 매개변수를 조정합니다. VM Parameters 필드의 시작 부분에 다음 명령을 입력합니다.

```
-classic
```

이 인수는 디버거가 classic VM을 사용하도록 지시합니다.

4 VM Parameters 필드의 나머지 인수는 보안 정책 파일의 위치를 가리켜야 합니다. Windows에서는 일반적으로 설치하는 위치가 `samples\RMI` 폴더입니다.

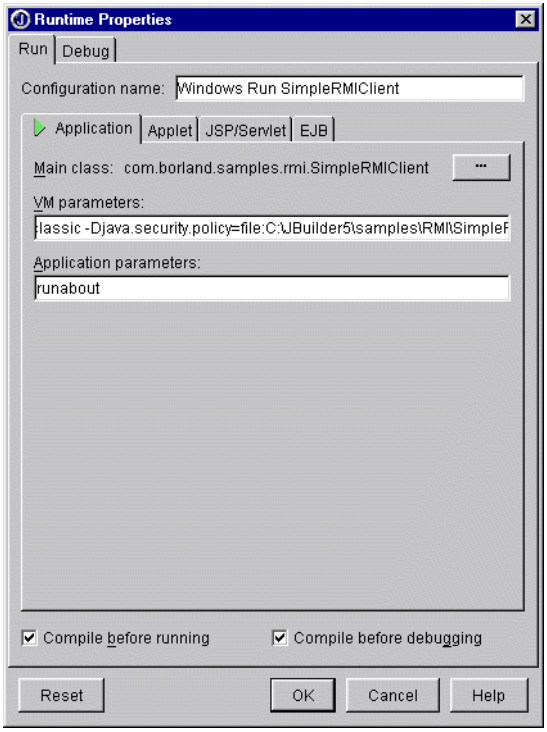
```
-Djava.security.policy=file:C:\JBuilder\samples\RMI\SimpleRMI.policy
```

5 메인 클래스가 다음과 같이 설정되었는지 확인합니다.

```
com.borland.samples.rmi.SimpleRMIClient
```

6 Application Parameters 필드에 원격 컴퓨터의 이름을 입력합니다. 이 이름은 서버의 Runtime Properties 대화 상자에서 Debug 페이지의 Host Name 필드에 입력한 이름입니다(이전 단원 참조).

7 완료되면 클라이언트의 Run 페이지가 다음과 같이 나타납니다.



8 Item Properties 대화 상자를 닫으려면 OK를 클릭합니다.

9 OK를 다시 클릭하여 Runtime Configurations 대화 상자를 닫습니다.

다음 단계에서는 클라이언트와 서버의 브레이크포인트를 설정합니다.

3 단계: 브레이크포인트 설정

이 단계에서는 클라이언트 프르세스에 라인 브레이크포인트를 설정하고 서버 프로세스에 크로스 프로세스 브레이크포인트를 설정합니다. 라인 브레이크포인트는 크로스 프로세스 브레이크포인트를 호출하려고 할 때 클라이언트가 실행 정지(pause)되게 합니다. 크로스 프로세스 브레이크포인트는 서버를 실행 정지합니다. 이러한 기술을 사용하면 클라이언트 프로세스에서 서버 프로세스를 한 단계씩 실행할 수 있습니다.

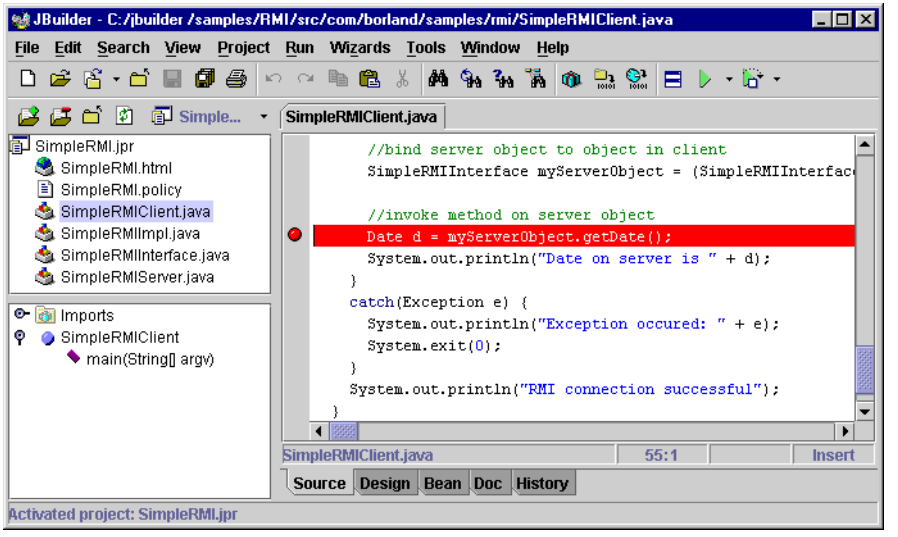
클라이언트 프로세스에 라인 브레이크포인트를 설정하려면

1 프로젝트 창에서 SimpleRMIClient.java를 더블 클릭합니다. 그러면 에디터에서 열립니다.

2 Search|Go To Line 명령을 사용하여 줄 55로 이동합니다. 그 줄은 다음과 같습니다.

```
Date d = myServerObject.getDate();
```

3 코드 줄의 왼쪽에 있는 회색 부분인 여백을 클릭하고 라인에 브레이크포인트를 설정합니다.

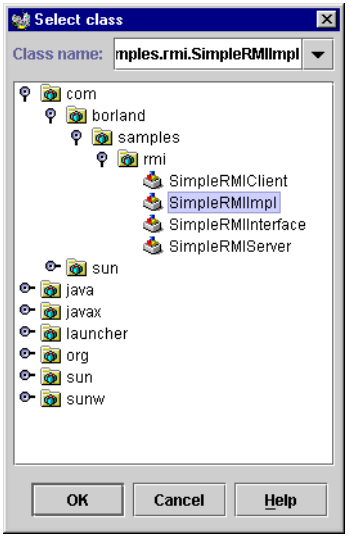


다음과 같은 방법으로 서버 프로세스에 크로스 프로세스 브레이크포인트를 설정합니다.

1 Run|Add Breakpoint|Add Cross Process Breakpoint를 선택합니다. Add Cross-Process Breakpoint 대화 상자가 표시됩니다.

2 Class Name 필드의 오른쪽에 있는 생략 버튼을 선택합니다.

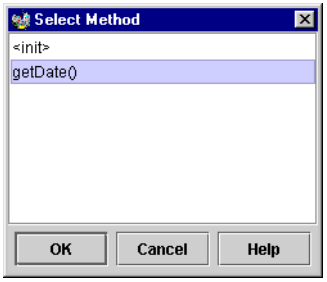
3 Select Class 대화 상자에서 com 폴더를 확장하고 서버 구현 클래스, com.borland.samples.rmi.SimpleRMIImpl을 찾아봅니다.



4 대화 상자를 닫으려면 OK를 클릭합니다.

5 Method Name 필드의 오른쪽에 있는 생략 버튼을 선택합니다.

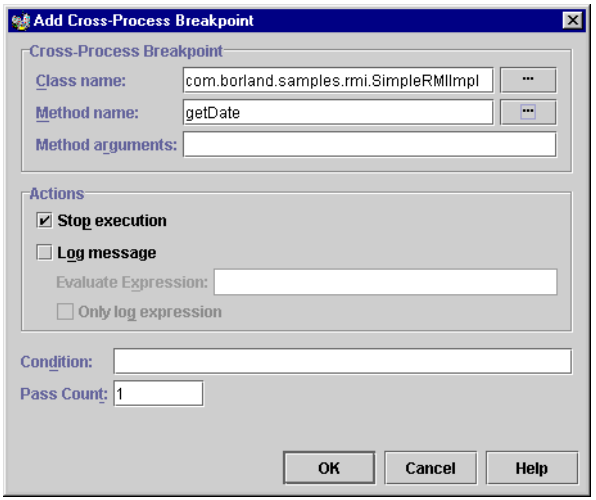
6 Select Method 대화 상자에서 getDate()를 선택합니다.



7 대화 상자를 닫으려면 OK를 클릭합니다.

8 Add Cross-Process Breakpoint 대화 상자의 Actions 옵션을 Stop Execution으로 설정합니다.

9 Add Cross-Process Breakpoint 대화 상자는 다음과 같이 나타납니다.



10 대화 상자를 닫으려면 OK를 클릭합니다.

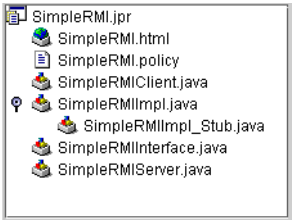
다음 단계에서는 서버를 컴파일하고 서버 클래스 파일을 원격 컴퓨터로 복사합니다.

4 단계: 서버 컴파일 및 원격 컴퓨터로 서버 클래스 파일 복사

이 단계에서는 서버를 컴파일하고 서버 클래스 파일을 원격 컴퓨터로 복사하는 방법에 대해 알려 줍니다.

JBuilder로 서버 파일을 컴파일하려면 Project|Make Project "SimpleRMI.jpr"을 선택합니다. 상태 표시줄은 프로젝트가 구축되었음을 보여 줍니다.

확장/축소 아이콘은 프로젝트 창의 `SimpleRMIImpl.java`로 표시되어야 합니다. RMI 컴파일러는 스텁 클래스, `SimpleRMIImpl_Stub.java`를 만듭니다. 이 파일은 자동 생성되므로 편집하지 마십시오.



DOS 창으로 이동하여 `samples\RMI` 폴더를 찾습니다. 이 폴더에는 `classes` 폴더가 들어 있습니다. `classes` 폴더에는 패키지 구조를 따르는 폴더 계층이 포함됩니다. 서버 클래스 파일은 `com\borland\samples\rmi` 폴더에 저장됩

니다. classes 폴더에는 dependency cache 및 Generated Source 폴더도 포함됩니다.

원격 컴퓨터로 서버 클래스 파일을 복사해야 합니다. 이 자습서의 용도에 따라 원격 컴퓨터의 RMI라는 새 폴더로 전체 RMI 폴더를 복사할 수 있습니다. 이렇게 하려면 다음 중 하나를 수행할 수 있습니다.

- 네트워크로 파일을 복사한 다음 원격 컴퓨터로 복사할 수 있습니다.
- 디스켓으로 파일을 복사한 다음 원격 컴퓨터로 복사할 수 있습니다.
- 파일을 원격 컴퓨터로 FTP 전송합니다.

중요 앞으로 JBuilder를 실행하는 클라이언트 컴퓨터의 소스 파일을 업데이트하면 .class 파일을 원격 컴퓨터로 다시 복사해야 합니다. 그렇게 하지 않으면 소스 파일과 컴파일된 파일이 일치하지 않아 오류가 발생합니다.

다음 단계에서는 원격 컴퓨터에서 RMI Registry 및 서버를 시작합니다.

5 단계: 원격 컴퓨터에서 RMI Registry 및 서버 시작

이 단계에서는 원격 컴퓨터에서 RMI 레지스트리를 시작하는 방법과 원격 컴퓨터에서 디버그 모드로 서버를 시작하는 방법에 대해 알려 줍니다. 서버를 시작하는 Java 명령줄의 디버그 설정뿐만 아니라 RMI 설정을 인식해야 합니다.

다음과 같은 방법으로 원격 컴퓨터에서 RMI Registry를 시작합니다.

1 4DOS 또는 4NT 창을 엽니다.

2 Jdk1.3\bin 폴더로 변경합니다.

3 다음 명령을 입력하여 RMI Registry를 시작합니다.

```
start rmiregistry
```

RMI Registry는 별도의 프로세스로 시작됩니다. 레지스트리가 시작되지 않으면 사용 가능한 메모리가 부족할 수 있습니다. 실행 중인 다른 애플리케이션을 종료한 다음 DOS 창을 닫고 다시 시도하십시오.

다음과 같은 방법으로 원격 컴퓨터에서 서버를 시작합니다.

1 Command 창을 시작합니다. Java 명령줄은 256자 이상입니다. 표준 4DOS나 4NT 창에서는 실행할 수 없습니다. Start 메뉴에서 Command 창을 시작합니다. Run을 클릭하고 command를 입력합니다. NT 컴퓨터인 경우에는 cmd를 입력합니다.

2 Jdk1.3\bin 폴더가 경로에 있는지 확인합니다.

3 RMI 예제를 포함하는 폴더의 루트로 이동합니다.

4 프롬프트에서 다음 명령을 입력합니다. 이 명령은 디버그 모드로 서버를 시작하고 실행을 일시 중지(suspend)시킵니다. 배치 파일이나 셸 스크립트에 명령을 저장하려고 할 수 있습니다. 이렇게 하려면 명령에 줄 바꿈이 없는지 확인합니다.

```
java -classic -Xdebug -Xnoagent -Djava.compiler=NONE -
Djava.rmi.server.codebase=file:\rmi\classes\ -Djava.security.policy=file:\
rmi\SimpleRMI.policy -
Xrunjdwp:transport=dt_socket,server=y,address=5000,suspend=y -classpath d:\
rmi\classes com.borland.samples.rmi.SimpleRMIServer
```

서버를 실행하기 위해 입력한 명령줄은 RMI와 디버거 인수를 모두 사용합니다. 각 매개변수에 대한 설명은 다음과 같습니다.

Table 27.2 명령줄 RMI 및 디버거 인수

매개변수	설명
java	Java VM을 실행하는 명령입니다.
-classic	HotSpot VM 대신에 classic VM을 사용하도록 디버거에 지시합니다.
-Xdebug	디버그 모드로 VM을 실행합니다.
-Xnoagent	디버그 에이전트를 사용하지 않습니다.
-Djava.compiler=NONE	JIT를 사용하지 않습니다.
-Djava.rmi.server.codebase= file:\rmi\classes\	서버 클래스 파일의 위치를 식별합니다.
-Djava.security.policy= file:\rmi\SimpleRMI.policy	자바 보안 정책 파일의 위치를 식별합니다.
-Xrunjdwp:transport=dt_socket,server=y,address=5000,suspend=y	디버거 옵션은 다음과 같습니다. • transport: transport 메소드입니다. 서버의 Runtime Properties Debug 페이지에 설정된 것과 일치해야 합니다 (27- 3페이지의 "2 단계: 런타임 및 디버그 구성 설정" 참조). • server: 서버 모드로 VM을 실행합니다. • address: 디버거가 원격 컴퓨터와 통신하는 포트 번호입니다. 서버의 Runtime Properties Debug 페이지에 설정된 것과 일치해야 합니다(27- 3페이지의 "2 단계: 런타임 및 디버그 구성 설정" 참조). • suspend: 프로그램이 시작되는 즉시 일지 중지할지를 지정합니다.
-classpath d:\rmi\classes	클래스 경로입니다.
com.borland.samples.rmi.SimpleRMIServer	실행 가능한 서버 파일이며 패키지 이름을 포함합니다.

다음 단계에서는 디버거를 사용하여 실행 중인 서버에 추가하고 크로스 프로세스 브레이크포인트가 설정된 서버의 `getDate()` 메소드를 한 단계씩 실행합니다.

6 단계: 디버그 모드에서 서버 프로세스와 클라이언트 시작 및 크로스 프로세스 브레이크포인트 한 단계씩 실행

이 단계에서는 JBuilder의 디버그 모드에서 서버 프로세스와 클라이언트를 모두 시작한 다음 크로스 프로세스 브레이크포인트를 한 단계씩 실행하는 방법을 알려 줍니다. 단계를 시작했으면 JBuilder를 사용하여 클라이언트와 서버 사이를 단계별로 실행할 수 있습니다. 다음을 수행합니다.

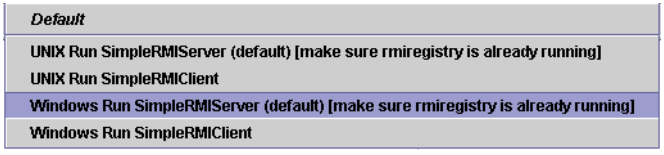
- 디버그 모드로 클라이언트 컴퓨터에서 서버 프로세스를 시작합니다.
- 디버그 모드로 클라이언트 컴퓨터에서 클라이언트를 시작합니다.
- 원격 컴퓨터에서 실행되는 서버에서 크로스 프로세스 브레이크포인트를 한 단계씩 실행합니다.

6 단계 : 디버그 모드에서 서버 프로세스 및 클라이언트 시작

다음과 같은 방법으로 JBuilder를 실행하는 클라이언트 컴퓨터에서 디버그 모드로 서버 프로세스를 시작합니다.

1 메인 툴바의 Debug Program 버튼 오른쪽에 있는 아래쪽 화살표를 클릭합니다.

2 `Windows Run SimpleRMIServer` 구성을 선택합니다.



참고 클라이언트 컴퓨터에서 RMI Registry를 시작하지 않아도 됩니다. 원격 컴퓨터에서 이미 실행되고 있습니다.

3 디버거가 시작되고 실행을 일시 정지합니다.

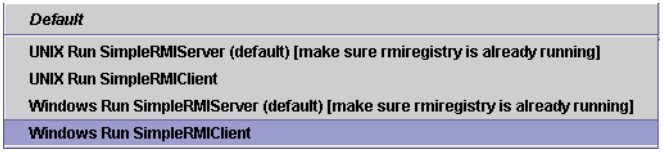
4 디버거 툴바에서 Resume Program 버튼을 클릭합니다.

원격 컴퓨터에 `SimpleRMIImpl ready` 메시지가 표시됩니다. 컴퓨터 이름과 주소가 JBuilder AppBrowser 창의 아래쪽에 있는 서버 프로세스 탭에 표시됩니다.

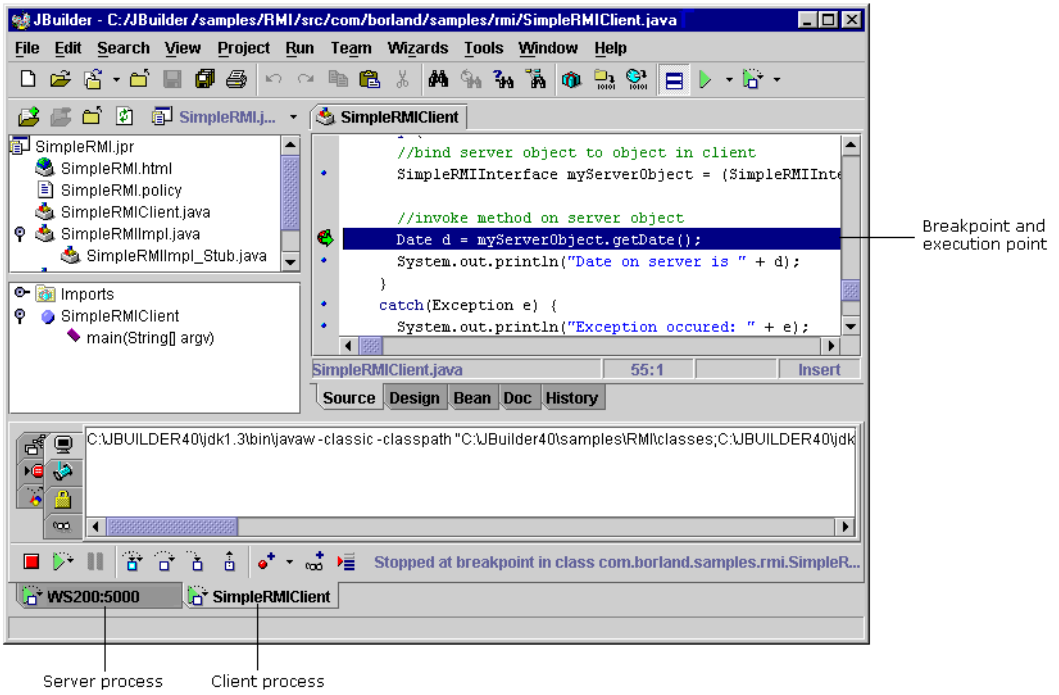
다음과 같은 방법으로 JBuilder를 실행하는 클라이언트 컴퓨터에서 디버그 모드로 클라이언트를 시작합니다.

1 메인 툴바의 Debug Program 버튼 오른쪽에 있는 아래쪽 화살표를 마우스 오른쪽 버튼으로 클릭합니다.

2 `Windows Run SimpleRMIClient` 구성을 선택합니다.

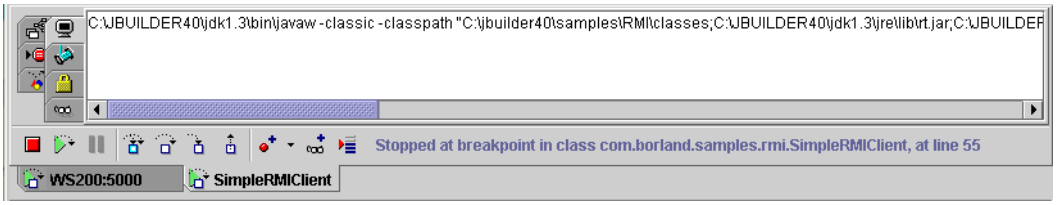


3 디버거가 시작되고 서버의 `getDate()` 메소드 호출에서 실행을 중단합니다.



다음과 같은 방법으로 크로스 프로세스 브레이크포인트를 한 단계씩 실행합니다.

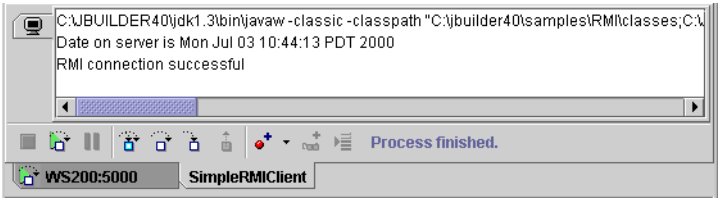
1 `SimpleRMIClient` 프로세스의 디버거 탭을 클릭합니다.



2 클라이언트의 디버거 툴바에서 Step Into 아이콘을 클릭하여 서버쪽 브레이크포인트 메소드를 한 단계씩 실행합니다. Step Over를 사용하면 디버거가 중단되지 않습니다.

3 Step Into 버튼을 세 번 더 클릭합니다. 원격 컴퓨터에 `SimpleRMIImpl getDate()` 메시지가 표시됩니다.

4 클라이언트가 실행되어 완료될 때까지 Step Into를 계속 클릭합니다. 디버거의 SimpleRMIClient 프로세스는 다음과 같이 나타납니다.



원격 컴퓨터에서 실행되는 서버의 출력은 다음과 같이 나타납니다.

```
SimpleRMIImpl ready
SimpleRMIImpl.getDate()
```

5 원격 컴퓨터에서 서버를 종료하려면 Command 창에서 ***Ctrl + C***를 누릅니다. RMIRegistry를 닫으려면 RMIRegistry 창에서 닫기 버튼을 클릭합니다.

JBuilder의 디버그 모드에서 서버나 클라이언트를 시작하면 다음 오류 메시지 중 하나가 표시될 수 있습니다.

Table 27.3 RMI 클라이언트 / 서버 오류 메시지

오류 메시지	설명
connection refused	원격 컴퓨터에서 RMI Registry가 실행되지 않습니다. 모든 프로세스를 중단하고 명령줄에서 start rmiregistry를 입력하여 원격 컴퓨터에서 RMI Registry를 실행합니다. Jdk1.3\bin 폴더는 경로에 있어야 합니다. 원격 서버를 다시 시작하고 디버그 프로세스를 다시 시작합니다.
Java exception: java.rmi.NotBoundException SimpleRMIImpl Instance	아직 서버 디버그 프로세스를 시작하지 않았습니다. 서버의 디버거 툴바에서 Resume Program 버튼을 클릭합니다. 디버그 모드에서 클라이언트를 다시 시작합니다.

축하합니다! 자습서를 완료했습니다. 미리 설정된 런타임 구성을 사용하여 원격 컴퓨터에서 RMI 서버를 실행했습니다. 그런 다음 JBuilder의 원격 디버깅 기능을 사용하여 프로그램을 디버깅했습니다.