제 5 장

원격 객체들

원격 메소스 호출
원격 메소드 호출 설정
원격 메소드 전달인자 전달
애플릿에서 RMI 사용법
자바 IDL과 CORBA

주기적으로 프로그래밍 공동체는 모든 문제들의 해결 방안으로써 "모든 곳에 객체들(Object everywhere)" 라는 생각을 한다. 이 생각은 협동하는 일련의 객체들은 어디에든지 위치할 수 있다는 것이다. 물론 이런 객체들은 표준 프로토콜을 가진 네트웍을 통하여 통신이 가능하다는 것을 가정한다.
예를 들어 여러분은 클라이언트에 객체를 갖을 수 있다, 이것은 데이터에 대한 요청을 채워서 보낸다. 클라이언트 객체는 서버가 가지고 있는 상세 요청정보에 대하여 메시지를 보낸다. 서버 객체는 데이터베이스에 접근하거나 부가적인 객체들과 통신하여 요청 정보를 모은다. 서버 객체는 클라이언트의 요청에 답을 한다. 이것은 클라이언트에게 답을 전송하는 것이다. 이처럼 대부분의 프로그래밍에서, 이런 방법은 여러가지 모호한 개념을 포함하고 있다. 이 장에서는 아래와 같은 목적을 다룬다.

통신이 가능한 객체간의 모델에 대하여 설명한다.
분산된 객체들의 유용한 상황들에 대하여 설명한다.
분산 객체들을 사용하는 방법과 자바가 수행되고 있는 2대의 컴퓨터 사이의 통신을 위해 원격 메소드 호출을 사용하는 방법을 보여준다.
여러분에게 CORBA를 소개한다, CORBA(Common Object Request Broker Architecture)는 다른 언어(자바,C++등)로 작성된 객체들 간의 통신을 가능하게 한다.

원격 객체에 대한 소개 : 클라이언트와 서버의 역할

여러분이 클라이언트 컴퓨터 상에서 지역적으로 정보를 수집하고 이 정보를 네트웍을 통해서 서버로 보낸다고 가정하자. 우리는 사용자가 지역적인 기계상에서 요청 정보 폼을 작성한다고 가정한다.이 폼 데이터는 서버로 보내지게 되며 서버는 요청에 대하여 작업을 한다. 반대로 클라이언트가 볼 수 있는 형태의 정보로 되돌려 보내진다. 이 과정은 아래의 그림5-1과 같다.

그림5-1 : 클라이언트와 서버사이에서 객체들의 전송

전통적인 C/S 모델에서 요청은 특정한 형태로 전송된다(가령, 변수와 값을 쌍 또는 XML 데이터의 형태로). 서버는 요청형태를 파싱하고,응답을 계산한 후 클라이언트에게 응답을 전송한다. 클라이언트는 응답을 파싱하고 사용자에게 보여준다.

그러나 여러분의 데이터가 구조화되었다면, 여기에는 중요한 코딩상의 어려움이 존재한다.:
여러분은 데이터를 변환하고 전송 형태를 적용해야 만 하는 상황을 맞게 될 것이다.

노트: 다른 방법으로는 HTML 폼으로 서버에게 요청정보를 보낸다. 클라이언트는 단순한 부라우저이고 서버는 정보에 대한 요청을 모으는 것이 필요하며 HTML과 같은 형태이다. 이런 모델일지라도 데이터를 모으는데 있어 폼 정보를 분리해야 한다. 일반적으로 폼 처리는 웹서버 상에서 일어난다, 데이터 처리는 다른 서버상에서 일어나는데 이를 응용 프로그램 서버라 부른다. 여러분은 다시 두가지 통신객체를 가진다, 웹 서버상의 클라이언트 객체와 응용 프로그램 서버상의 서버 객체이다. 이렇게 매우 일반적인 모델은 사실적이다. 이 장에서는 우리는 좀더 전통적인 클라이언트/서버 예제에 대하여 다룬다 이것은 클라이언트와 서버의 역할을 좀더 직관적으로 할 수 있기 때문이다.

해결 가능한 방법은 무엇일까? 이제, 객체들이 다른 객체들에게 요청을 보내는 것을 OOP의 개념으로 생각을 하자. 우리는 다른 기계상에 객체를 위치시키고 이들 객체들간에 직접 메시지를 보낼 수 있다. 클라이언트 객체는 자바언어로 작성되었다고 가정하고 그래서 이것은 어느 곳에서도 작동이 가능하다. 서버 객체들은 두가지 가능성을 가진다:

서버 객체들은 자바 언어로 작성되지 않았다.(기존 객체이거나 누군가 원하지 않기 때문이다.)
서버 객체들은 자바 언어로 작성된다.

첫번째 상황은 객체들이 어떤 언어로 작성되었든 간에 상관없이 이들 객체들간의 대화방법을 필요로 한다. 여러분이 이러한 것에 대하여 생각한다면, 이 방법에 대한 이론적인 가능성이 놀라운 성과라는 데 동의할 것이다.

궁극적으로 임의 언어로 작성된 일련의 바이트들이 어떻게 할 수 있는지, 이런 지식에 대하여 우리는 알지 못하기에, 어떤 서비스를 제공하고 무슨 메시지를 제공할 것인지를 말해주어야 한다. 물론 이런 작업은 사실 쉽지가 않다 그러나 이 개념은 매우 우수하다. "Object Management Group" 혹은 OMG(www.omg.org)에 의한 "common object request broker architecture", 혹은 CORBA 표준은 데이터의 교환과 서비스들을 찾는 데 있어 필요한 공통의 메커니즘을 정의한다.

기본 아이디어는 이들 정보를 찾기 위한 작업과 어떤 요청된 서비스의 구동을 ORB(Object Request Broker)라고 불리는 것에 위임하는 것이다.여러분은 ORB를 객체간 통신을 위한 공통의 중개자중의 하나로 생각할 수 있다.객체들은 직접적으로 다른 객체들과 통신을 할 수 없다. 그들은 항상 그들 사이를 중재하는 객체 브로커를 사용해야만 한다. ORB들은 네트웍상에 존재하며, 이것은 객체들간의 통신을 할 수 있게 하는 매우 중요한 것이다. 대부분의 ORB들은 OMG에 의해 정립된 ORB간의 통신을 위한 명세를 따른다. 우리는 이런 명세를 Internet Inter-ORB Protocol 혹은 IIOP라 부른다.

노트: 마이크로소프트는 객체간 통신에 대하여 하위 차원의 프로토콜을 사용한다.(COM+라고 현재 불려진다) ORB와 유사한 서비스가 윈도우 운영 체제에 포함되어 있다. 비록 몇몇 업체들이 유닉스 플랫폼상에서 일정 수준의 상호연동을 제공하기는 하지만 COM+은 윈도우에 주된 솔루션이다. 반대로 CORBA는 객체간 통신에 대한 다중언어와 다중 플랫폼을 지원한다.

CORBA는 완벽하게 언어에 중립을 지킨다. 클라이언트와 서버 프로그램은 C++,자바언어, 혹은 CORBA 바인딩을 지원하는 모든 다른 언어로 작성될 수 있다.
여러분은 IDL(Interface Definition Language)를 사용하여 메세지의 사인과 여러분의 객체가 전송할 수 있고 이해할 수 있는 데이터 타입을 지정할 수 있다(IDL 명세는 자바언어의 인터페이스와 비슷하다, 사실 여러분은 통신 객체들이 지원해야 하는 인터페이스를 정의하는 것으로 IDL을 생각할 수 있다. 이 모델의 한가지 멋진 기능은 존재하는 기존의 객체에 대한 IDL 명세를 제공할 수 있고 이들의 서비스에 접근할 수 있다는 것이다.) 매우 많은 사람들이 CORBA가 곧 매우 중요하게 될 것이며 자바언어가 CORBA 클라이언트와 서버를 구현하는데 가장 좋은 선택이라고 믿고 있다. 그러나 솔직히 말하면, CORBA는 느린 성능과 복잡한 구현, 상호연동 문제가 있다고 평가 받는다.

만일 두 통신 객체들이 자바 언어로 작성되었다면 완전한 일반성과 CORBA의 복잡성을 필요로 하지 않는다. Sun사는 자바 응용 프로그램 간의 통신을 위한 RMI라고 불리는 CORBA

와 비슷한 구조를 개발하였다,

노트: CORBA 지원자들은 초기에 CORBR 표준을 무시하는 RMI를 좋아하지 않았다. 그러나 그들은 CORBA와 RMI 간의 상호 연동을 위해 노력하였다. 현재 RMI 버전은 객체간 통신을 위해 기존의 RMI 프로토콜 대신 IIOP 프로토콜을 사용한다.

CORBA가 미래에 관심을 갖는 반면에 RMI는 보다 이해하기 쉽고 사용하기 쉽다. 이런 이유로 우리는 이 장에서 RMI를 먼저 시작한다. 물론 RMI는 자바 객체들간의 통신에 대하여서만 사용한다. 이 다음 절에서는 우리는 CORBA 프로그래밍과 C++ 클라이언트와 자바언어로 작성된 서버 그리고 자바 언어로 작성된 클라이언트와 C++로 작성된 서버의 사용 방법을 소개할 것이다.

원격 메소드 호출

원격 메소드 호출 구조는 매우 간단한 것 처럼 보인다. 만일 여러분이 다른 기계상에 존재하는 객체에 접근한다면 여러분은 원격 객체의 메소드를 호출할 수 있다. 물론 메소드의 전달인자들은 다른 기계에 전송돼야하고 , 이 객체는 실행할 메소드를 알려주고 반환 값이 다시 전송돼야 한다. RMI는 이러한 상세한 작업들을 관리한다

예를 들어 제품정보를 찾고 있는 클라이언트는 서버상의 Warehouse 객체에 질의를 할 수 있다. Customer객체는 하나의 인자를 가지고 Find 원격 메소드를 호출한다.Find 메소드는 클라이언트에게 객체를 반환한다.:제품 정보를 가진 객체(그림 5-2 참조)

그림5-2 : 서버 객체상의 원격 메소드 호출

RMI 용어로, 원격 호출을 하는 메소드는 *클라이언트 객체*이고 원격 객체를 *서버 객체*라 한다. 클라이언트/서버 용어는 오직 단일 메소드 호출에 적용된다는 것을 상기하는 것이 중요하다. 원격 메소드를 호출하는 자바 코드를 수행하는 컴퓨터는 해당 호출에 대한 클라이언트이다. 호출을 처리하는 객체가 위치하고 있는 컴퓨터는 해당 호출에 대한 역할이 반대로 바뀌는 경우도 가능하다. 앞선 호출에 대한 서버가 다른 컴퓨터상에 존재하는 객체상의 원격 메소드를 호출할 경우 클라이언트가 된다.

스터브들과 전달인자 마샬링

클라이언트 코드가 원격 객체상의 원격 메소드 호출을 원할 때, 실제적으로 스터브(stub)라는 대표 객체내에 갭슐화되어 있는 일반 자바 메소드를 호출한다. 스터브는 서버가 아니라 클라이언트상에 존재한다. 스터브는 원격 메소드 내에서 사용되는 전달인자들을 바이트들의 블록으로 포장한다. 이 패키지는 각 전달인자에 대하여 장치 독립적인 인코딩 방법을 사용한다. 예를 들면, 숫자는 항상 빅 인디언(big-endian) 형태로 전송된다. 객체들은 제1편의 12장에 서술된 연속화 기법으로 인코딩된다. 전달인자 마샬링은 하나의 가상 머신에서 다른 가상 머신으로 안정한 형태의 전송을 위해서 사용된다.

요약하면 : 클라이언트상의 스터브는 다음과 같은 정보 블럭으로 구성된다.

- **사용되어질 원격 객체의 식별자**
- **호출될 메소드의 설명**
- **마샬링 전달인자**

그런 다음, 이 정보가 서버로 전송되다. 서버측에서는 모든 원격 메소드 호출에 대한 행동은 아래와 같은 기능을 수행한다.

- **전달인자를 언마샬(unmarshall)한다.**
- **호출될 객체를 위치시킨다.**
- **희망하는 메소드를 호출한다.**
- **반환 값에 대한 먀샬링과 호출에 대한 예외상황을 감지하다.**
- **클라이언트상의 스터브를 통해 마샬링된 반환 데이터로 구성된 패키지를 돌려보낸다.**

스터브는 서버로부터 반환된 값 혹은 예외에 대하여 언마샬한다. 이 값은 스터브의 호출에 대한 결과 값이 된다. 혹은 원격 메소드가 예외를 발생하면 스터브는 호출자의 프로세스 공간에 재투구(rethrows) 한다. 그림5-3은 원격 메소드 호출에 대한 정보의 흐름을 보여준다.

그림 5-3 : 전달인자 마샬링

이 과정은 매우 복잡하다, 그러나 좋은 소식은 이 과정이 완전하게 자동적이며 확장이 가능하고 프로그래머에게 투명하다는 것이다. 더욱 이 원격 자바 객체를 설계한 사람들은 원격지의 서버 객체를 지역 객체처럼 동일하게 보고 사용(look and feel) 할 수 있도록 노력하고 있다.

원격 메소드 호출 구문은 지역 호출과 동일하다. 만약 원격 기계상에 존재하는 중앙 창고객체에 위한 스터브 객체가 centralWarehouse 이고 호출하고자 하는 메소드가 getQuantity 메소드이면 전형적인 호출방법은 다음과 같다.

```
centralWarehouse.getQuantity("SuperSucker 100 Vacuum Cleaner");
```

클라이언트 코드는 항상 원격 객체에 접근하기 위한 인터페이스 타입의 객체변수를 사용한다. 예를 들면 이런 호출에 관련된 인터페이스는 아래와 같다.

```
interface Warehouse
{   public  int getQuantity(String description)
            throws RemoteException;
   . . .
}
```

그리고 인터페이스를 구현할 변수에 대한 객체 선언이다.

```
Warehouse  centralWarehouse = . . . ;
```

물론 인터페이스들은 단지 무슨 메소드가 어떤 전달인자를 통해 호출될 수 있는지 명시해 놓은 추상화 객체들이다. 인터페이스 타입의 변수들은 반드시 항상 실제 객체의 타입들에 연계되어 있어야 한다. 원격 메소드가 호출될 때, 객체 변수는 스터브 객체를 참조한다. 클라이언트 프로그램은 실제로 이들 객체의 타입을 알지 못한다. 스터브 클래스와 관련된 객체들은 자동적으로 생성된다.
자바 디자이너들이 프로그래머로부터 원격 메소드 호출에 대한 많은 상세들을 감추어 주는 것은 훌륭한 작업을 수행한 반면에 여전히 정복되어야 할 많은 기술들이 남아 있다.
이런 프로그래밍 작업을 이장의 나머지 주제가 될 것이다.

노트: 원격 객체들은 지역 객체들처럼 자동적으로 쓰레기 수집이 된다. 그러나 현재 분산된 수집자는 참조 계수를 사용하고 참조되지 않은 객체들이 생명주기를 감지하지 못한다. 생명주기는 프로그래머에 의해서 극명하게 제거되어야 한다.

동적 클래스 로딩

여러분이 원격 객체를 전달인자나 원격 메소드의 반환 값으로 다른 자바 프로그램에 전달할 때, 해당 프로그램은 관련된 스터브 객체를 다룰 수 있어야 한다. 이것은 스터브 클래스에 대한 코드를 가지고 있어야 한다. 스터브 메소드는 흥미로운 작업을 수행하지는 못한다.
그들은 단지 전달인자에 대한 마샬과 언마샬 그리고 이 정보를 서버에 전달한다. 물론 이러한 작업은 프로그래머에게 투명하게 수행된다.

더구나 전달인자,반환 값 그리고 예외 객체들에 대한 클래스 또한 로드돼야 한다. 이것은 여러분이 생각하는 것 보다 좀더 복잡할 수 있다. 가령, 원격 메소드는 클라이언트에게 알려진 특정 반환 타입으로 선언되어 진다. 그러나 실제 클라이언트에게 알려진 것이 아니라 상속된 클래스의 객체를 반환한다. 클래스 로더는 상속된 해당 클래스를 로드할 것이다.

해당 클래스를 로딩하는 동안 스터브 클래스들은 수행중인 클라이언트 프로그램을 이용할 수 있어야 한다. 이를 위한 한가지 명백한 방법은 이들 클래스들을 로컬 파일 시스템상에서 이용할 수 있게 만드는 것이다. 그러나 만약 서버 프로그램이 확장되고 새로운 클래스에 대한 반환형, 그리고 예외 상황이 추가된다면 클라이언트에 대한 지속적인 갱신이 혼란스럽게 될 것이다. 이런 이유로 인하여 RMI 클라이언트는 다른 장소로부터 스터브 클래스를 자동적으로 로딩할 수 있다. 이 과정은 부라우저상에서 작동하는 애플릿의 로딩 과정과 비슷하다.

다른 네트웍상으로부터 새로운 프로그램 코드가 로딩될 때 마다 보안 문제가 발생한다. 따라서 여러분은 RMI 클라이언트 응용 프로그램에서 보안 관리자를 사용할 필요가 있다. 이것은 스터브 코드의 바이러스로부터 프로그램을 보호하는 안전한 기법이다.
특별한 응용 프로그램에 대하여 프로그래머는 그들의 클래스 로더와 보안 관리자를 사용하여 대체할 수 있다. 그러나 RMI에서 제공되는 클래스 로더와 보안 관리자는 일반적인 사용을 위해 충분한 기능을 제공한다.

원격 메소드 호출 설정

비록 간단한 원격 객체 예제를 사용한다 하더라도 독립된 자바 응용 프로그램이나 애플릿을 수행하는 것보다 좀더 많은 설정을 요구한다.
여러분들은 서버와 클라이언트 컴퓨터에서 자바 프로그램을 실행해야 한다. 객체와 관련하여 필요한 모든 정보는 클라이언트측 인터페이스와 서버측 구현부분에 분리되어 있어야 한다. 또한 클라이언트는 서버측에 원하는 객체를 위치시키기 위한 특별한 질의 메커니즘이 있어야 한다.

실제 코딩을 시작하기 위해, 우리는 간단한 예제를 사용하여 이러한 각각의 요구 사항들을

여러분이 직접 경험하기 원한다. 첫번째 예제는 서버 컴퓨터상의 제품형태의 객체를 두개 생성할 것이다. 그리고 나서 클라이언트 컴퓨터에 위치시키고 질의를 수행하는 프로그램이 작동될 것이다.

노트: 여러분은 이 예제를 단일 컴퓨터상이나 네트웍으로 연결된 두 대의 컴퓨터 상에서 수행할 수 있다. 우리는 여러분에게 두가지 방법 모두를 제공한다. 만약 이 코드를 단일 컴퓨터 상에서 수행한다면 여러분은 네트웍 서비스가 가능해야 한다. 특히 TCP/IP가 수행되고 있어야 한다는 것을 확인해야 하며 만약 여러분이 네트웍 카드가 없다면 다이얼-업 네트워킹을 사용할 수가 있다.

인터페이스들과 구현들

여러분의 클라이언트 프로그램은 서버객체를 조작할 필요가 있다. 그러나 실제로 서버 객체의 복사본을 가지고 있지 않다. 이 객체들은 서버에 존재한다. 클라이언트 코드는 여전히 이 객체들을 갖고 할 수 있는 것들이 무엇인지 여전히 알고 있어야 한다.
이러한 능력들은 클라이언트와 서버가 서로 공유하는 인터페이스내에서 표현되고 양쪽 컴퓨터상에 동시에 존재해야 한다.

```
interface Product // 클라이언트와 서버에 의해 공유된다.
   extends Remote
{  public String getDescription() throws RemoteException;
}
```

이 예제에서 처럼 원격 객체를 위한 모든 인터페이스는 java.rmi 패키지내에 정의된 Remote 인터페이스를 확장해야 한다. 또한 이 인터페이스내에 정의된 모든 메소드들은 RemoteException을 투구해야 한다.– 항상 원격 호출이 실패할 가능성이 존재한다. 가령, 서버 또는 네트웍 통신이 일시적으로 사용이 불가능할 수 있다.
여러분의 클라이언트 코드는 이런 가능성에 대한 준비를 해야 한다. 이러한 이유들로 인해 자바 언어는 강제적으로 모든 원격 메소드 호출마다 RemoteException을 감지하도록 하고 호출이 성공하지 못했을 때 취해야 할 적합한 행동을 명시하도록 한다.

스터브를 통해 서버 객체에 접근하는 클라이언트는 다음의 인터페이스로 구현된다.

```
Product  p = . . . ;
```

```
// 원격 객체에 대한 참조를 클라이언트가 어떻게 사용하는지를 보라.
String d = p.getDescription();
System.out.println(d);
```

다음 절에서, 우리는 클라이언트가 원격 객체에 대한 참조를 얻는 방법들을 살펴볼 것이다.

하부에서 여러분은 실제 원격 인터페이스내에서 알려진 메소드를 수행하는 클래스를 구현해야 한다.

```
public class ProductImpl
    implements UnicastRemoteObject
{   public ProductImpl(String d)
     throws RemoteException
    {   descr = d;
     }

     public String getDescription()
            throws RemoteException
   { return "I am a " + descr + ". Buy me!";
     }
    private String descr;
}
```

이 클래스는 getDescription이라는 원격 클라이언트로부터 호출될 수 있는 단일 메소드를 가지고 있다.
위의 것은 서버 클래스이다. 왜냐하면 이 클래스는 원격지에서 접근 가능한 객체를 만들어 주는 구체화(concrete) 자바 클래스인 UnicastRemoteServer를 확장했기 때문이다.

노트: ProductImpl 클래스는 전형적인 서버 클래스가 아니다, 그 이유는 아주 단순한 작업만을 수행하기 때문이다. 일반적으로 여러분은 클라이언트에서 지역적으로 수행할 수 없는 많은 작업들을 서버 클래스가 가지기를 원한다. 우리는 단지 원격 메소드 호출에 대한 기법에 대하여 설명하기 위하여 이 예제를 사용한다.

모든 서버 클래스는 반드시 java.rmi.server 패기지로부터 RemoteServer 클래스를 확장해야만

한다. 그러나 RemoteServer 클래스는 추상 클래스로서 서버 객체와 그들의 클라이언트 스터브 사이의 통신에 대한 기본 메커니즘만이 정의 되어 있다.
UnicastRemoteObject 클래스는 RemoteServer 클래스의 확장이고 구체화 클래스이다. 따라서 이 클래스에 어떤 추가적인 코드를 작성할 필요없이 사용할 수 있다. 서버 클래스가 UnicastRemotServer 를 상속받는 것은 서버 클래스에 대한 가장 작은 저항 경로를 얻는 것이다. 그리고 이 장에서 소개되는 모든 서버 클래스들은 이렇게 상속받을 것이다. 그림 5-4는 이들 클래스들의 상속 관계를 보여준다.

그림5-4 : 상속 다이어그램

UnicastRemoteObject는 서버상에 존재한다. 이 객체는 서비스의 요청을 받았을 때 존재하고 TCP/IP 프로토콜을 통해 도달할 수 있어야 한다.
이것은 이 책에 있는 모든 서버 클래스들을 확장하는 데 사용하는 클래스이자 현재 버전의 RMI 패키지에서 이용할 수 있는 유일한 서버 클래스이다. 머지않아 Sun사와 협력 업체들은 RMI 서버들이 사용할 수 있는 다른 클래스를 디자인 할 것이다. 예를 들면, Sun사는 이미 객체들이 복수 개의 서버들 상에서 중복되는 객체를 위해 MulticastRemoteServer에 대하여 논의하고 있다. 가능한 다른 것들은 요구에 따라 활동하는 객체나 UDP같은 다른 통신 프로토콜을 사용할 수 있는 객체를 지원하는 것이다.

노트: 여러분이 RMI를 사용할 때(혹은 다른 분산 객체 메커니즘을 사용할 때) 다소 어리둥절하게 만드는 클래스들의 이름들이 존재한다. 우리는 모든 예제에 대하여 단일 이름 변경을 사용한다. 다행스럽게도 이것은 각각의 클래스의 목적을 좀더 쉽게 인식하게 해준다.(표 5-1)

표 5-1 RMI 이름 규칙

접미사가 없다(예, Product)	원격 인터페이스
Impl 접미사 (예, ProductImpl)	해당 인터페이스의 서버 클래스 구현
Server 접미사(예, ProductServer)	서버 객체들을 생성하는 서버 프로그램

Client 접미사(예, ProductClient)	원격 메소드들을 호출하는 클라이언트 프로그램
_Stub 접미사(예, ProductImp_Stub)	rmic 프로그램에 의해서 자동적으로 생성되는 스터브 클래스
_Skel 접미사(예, ProductImpl_Skel)	생성되는 스케로턴 클래스

여러분은 ProductImpl 클래스 객체에 대한 스터브를 생성해야 한다.
스터브들은 마샬된 전달인자들의 클래스이고 네트웍으로부터 전송된 메소드 호출의 결과에 대한 마샬이라는 것을 상기해야 한다.
프로그래머는 결코 이들의 클래스를 직접적으로 사용할 수 없다. 더욱이 이들 클래스들은 결코 손으로 작성될 수 없으며 rmic 에 의하여 아래와 같이 자동적으로 생성된다.

rmic 툴에 의한 작업으로서 ProductImpl_Stub.class 클래스 파일이 생성된다. 만약 여러분의 클래스가 패키지 안에 있다면 여러분은 반드시 완전한 패기지명을 사용하여 작업해야 한다.
만약 여러분의 클라이언트가 JDK 1.1이면 아래와 같이 한다.
스터브 파일과 ProductImpl_Skel.class라는 파일이 생성된다. 이 스케로턴 파일은 더 이상 자바 2 플랫폼에서는 필요하지 않다.

노트: rmic을 실행하기 전에 반드시 소스파일을 javac을 이용하여 컴파일해야 한다. 만약 여러분이 패키지 파일안의 클래스에 대한 스터브를 생성하려면 반드시 완전한 패키지명을 적어야 한다.

서버 객체들의 위치

서버에 존재하는 원격 객체에 접근하기 위해서 클라이언트는 지역 스터브 객체를 필요로 한다. 어떻게 스터브 객체가 요청을 할 수 있을까? 가장 일반적인 방법은 서버 객체를 반환하는 원격 메소드를 호출하는 것이다. 그러나 여기에서 "닭이 먼저냐 달걀이 먼저냐" 라는 문제가 발생한다. 첫번째로 서버 객체는 몇몇 다른 방법으로 위치될 필요가 있다. Sun RMI 라이브러리는 부트스트랩 레지스트리(bootstrap registry) 서비스를 제공한다.

서버 프로그램은 부트스트랩 서비스를 가지고 객체를 등록하고, 클라이언트는 이들 객체를 가지고 온다. 서버 객체는 객체에 대한 참조와 이름을 부트스트랩 레지스트리 서비스에 제공함으로서 등록되고 이름은 중복되지 않은 (희망 사항) 유일한 문자열이다.

```
// 서버
ProductImpl p1 = new ProductImpl("Blackwell Toaster");
Naming.bind("toaster",p1);
```

클라이언트 코드는 아래와 같은 방식으로 명시된 서버 이름과 객체 이름에 의해서 서버 객체에 접근하는 스터브 코드를 얻는다.

```
// 클라이언트
Product p
   = (Product)Naming.lookup("rmi://yourserver.com/toaster");
```

RMI URL은 서버에 의하여 rmi:/ 로 시작하며 포트 번호 그리고 슬래쉬(/), 객체 이름으로 작성된다. 다른 예제는 아래와 같다.

```
rmi://localhost:99/central_warehouse
```

기본 포트는 1099이다.

노트: 전체 레지스트리내에서 이름들을 중복되지 않고 유일하게 유지하는 것은 아주 어렵기 때문에 여러분은 서버상에 객체들을 위치시키기 위해 일반적인 메소드 같은 이름을 사용하지 않는 편이 낳다. 대신에 부트스트랩 서비스에 등록된 상대적으로 적은 수의 명명된 서버 객체를 이용하는 편이 낳다. 우리의 예제에서 일시적으로 이 규칙을 위반하고 상대적으로 평범한 객체들을 등록하였다. 이렇게 한 이유는 여러분에게 객체들을 등록하고 위치시키는 방법들을 소개할 필요가 있었기 때문이다.

보안 이유들 때문에, 응용 프로그램이 레지스트리로서 같은 호스트 상에서 수행된다면 오직 등록된 객체들만을 바인드,언바인드 혹은 재바인드(rebind)를 할 수 있다. 이것은 악의에 찬 클라이언트가 레지스트리 정보를 변경하는 것을 막을 수 있다. 그러나, 모든 클라이언트는 객체를 탐색(look-up)할 수 있다.

팁: 만일 여러분이 우리의 서버를 튜토리얼 문서내의 서버 예제와 비교한다면 우리가 서버내에 보안 관리자를 설치하지 않았다는 점에 주목할 것이다. 튜토리얼에서 언급한 것과 달리 보안 관리자는 필요하지 않을 뿐만 아니라 RMI서버들에 대한 이득도 없다. 보안 관리자

를 추가하는 것은 설정 에러들에 대한 다른 많은 문제를 양산해내기 때문에 우리는 여러분에게 오직 서버 구현시 특정 행동을 제어하길 원하거나 서버가 다른 서버의 RMI 클라이언트 그 자체일 경우에만 보안 관리자를 설치하라고 추천한다.

예제 5-1은 toaster와 microwave라는 이름으로 두개의 Product 객체들을 등록하는 완전한 프로그램이다.

예제 5-1: ProductServer.java

```
import java.rmi.*;
import java.rmi.server.*;
import sun.applet.*;

public class ProductServer
{  public static void main(String args[])
   {  try
      {  System.out.println
            ("Constructing server implementations...");

         ProductImpl p1
            = new ProductImpl("Blackwell Toaster");
         ProductImpl p2
            = new ProductImpl("ZapXpress Microwave Oven");

         System.out.println
            ("Binding server implementations to registry...");

         Naming.rebind("toaster", p1);
         Naming.rebind("microwave", p2);

         System.out.println
            ("Waiting for invo경고s from clients...");
      }
      catch(Exception e)
      {  System.out.println("Error: " + e);
      }
```

```
    }
}
```

예제 5-2: ProductImpl.java

```
import java.rmi.*;
import java.rmi.server.*;

public class ProductImpl
    extends UnicastRemoteObject
    implements Product
{   public ProductImpl(String n)
        throws RemoteException
    {   name = n;
    }

    public String getDescription()
        throws RemoteException
   {  return "I am a " + name + ". Buy me!";
    }

    private String name;
}
```

예제 5-3: Product.java

```
import java.rmi.*;

public interface Product extends Remote
{   String getDescription()
        throws RemoteException;
}
```

서버의 구동

우리의 서버프로그램은 아직 구동될 준비가 되지않았다. 왜냐하면 부트스트랩 레지스트리 서비스를 이용할 수 있어야 하고 등록하는 동안 작동하고 있어야 하기 때문이다.

윈도우 95나 NT하에서, 여러분은 도스 프롬프트나 실행 다이얼로그 박스상에서 명령문을 실행하여 이를 수행할 수 있다.(start는 윈도우에서 프로그램을 시작하라는 명령어이다.)

```
start  rmiregistry
```

아래는 유닉스에서의 사용법이다.

```
rmiregistry &
```

이제 여러분은 서버를 구동시킬 준비가 되었다. 윈도우 상에서 아래와 같은 명령어를 사용한다.

```
start java ProductServer
```

유닉스 상에서의 명령은 다음과 같다.

```
java ProductServer &
```

만약 서버 프로그램을 아래와 같이 작동시키면

```
java ProductServer
```

프로그램은 정상적으로 종료하지 못한다. 결국 이것은 아주 낯설어 보인다. 프로그램은 단지 두개의 객체를 만들고 이를 등록한다. 실제 main함수는 여러분이 기대했던 것처럼 등록한 후에 즉시 종료한다. 그러나 여러분이 UnicastRemoteServer를 확장한 클래스의 객체를 생성할 때, 자바는 프로그램이 무한정 수행 되도록 분리된 쓰레드를 시작한다. 그리하여 프로그램은 클라이언트들이 연결되도록 하기 위해 계속 머물러 있는다.

팁: JDK는 분리된 윈도우 프로세스로서 자바 인터프리터를 시작하고 계속 수행하게 해주는 javaw명령을 포함하고 있다. 이전 버전의 문서는 여러분들이 이 RMI를 위해 윈도우에서 백그라운드로 자바 세션을 수행하기 위해서 start java가 아니라 javaw를 사용하기를 권하고 있다. 그러나 이것은 두가지 이유에서 좋은 아이디어가 아니다. 윈도우는 javaw 백그라운드 프로세스를 종료하기 위한 도구를 갖고 있지 않다. 이것은 작업목록에 나타나지 않는다. 여러분은 등록된 클래스의 스터브를 변경할 때 부트스트랩 레지스트리 서비스를 종료하고 재시작할 필요가 있다. Start 명령어로 시작한 프로세스를 종료하기 위해 여러분이 할 수 있는 모든 일은 윈도우를 클릭하고 Ctrl + C 를 누르는 것이다.

여기에는 또 다른 중요한 이유가 있다. 서버 프로세스가 javaw를 사용하여 수행될 때, 출력이나 에러 메시지가 버려진다. 특히 그 메시지들은 어는 곳에서도 보여지지 않는다. 만일 여러분들이 에러 메시지나 출력을 보고자 하면, Start 메소드를 사용하라. 그러면 출력 결과가 콘솔에 보여진다. 우리가 그랬던 것처럼 여러분은 이러한 메시지들을 보길 원할 것이다.

여러분이 RMI를 사용할 때 잘못될 수 있는 많은 것들이 존재한다. 가장 일반적인 에러는 아마 rmic를 수행하는 것을 기억하지 못하는 것이다. 그러면 서버는 스터브를 잃어버렸다고 불평한다. 만일 여러분이 javaw를 사용한다면 에러 메시지를 보지 못하기 때문에 왜 클라이

언트가 서버 객체들을 발견하지 못하는지 궁금해 질 것이다.

클라이언트 프로그램을 작성하기 전에, 원격 객체의 등록이 성공적으로 이루어 졌는지 검사해 보자. Naming 클래스는 현재 등록되어 있는 모든 이름의 목록을 반환하는 list 메소드를 가지고 있다. 다음은 레지스트리 내의 이름의 리스트를 반환하는 프로그램이다. (예제 5-4)
우리의 경우 출력결과는 다음과 같다.

rmi:/toaster
rmi:/microwave

예제 5-4: ShowBindings.java

```
import java.rmi.*;
import java.rmi.server.*;

public class ShowBindings
{  public static void main(String[] args)
   {   try
       {   String[] bindings = Naming.list("");
           for (int i = 0; i < bindings.length; i++)
                System.out.println(bindings[i]);
        }
       catch(Exception e)
       {   System.out.println("Error: " + e);
        }
   }
}
```

클라이언트 측면

이제 우리는 새롭게 등록된 Product 객체들 각각에게 해당 내용의 출력을 요청하는 클라이언트 프로그램을 작성할 수 있다.

RMI를 사용하는 클라이언트 프로그램은 네트웍상에서 동적인 로드 과정을 조정하는 보안 관리자를 설치할 필요가 있다. 자바는 StubSecurityManager와 같은 보안 관리자를 제공하며 다음과 같이 설치할 수 있다.

System.setSecurityManager (new RMISecurityManager());

노트: 이 경우에, 우리는 실제로 동적 클래스 로딩을 필요로 하지 않는다. 모든 클래스들은 로컬 파일 시스템에서 이용할 수 있다. 그럼에도 불구하고, 자바는 보안관리자가 있어야 한다고 주장한다. 이미 보안 관리자는 애플릿들에 대해 해당 애플릿 코드가 어떤 해로운 일을 수행하지 못하도록 보장해 준다. 응용 프로그램들을 다룰 경우 기본 보안 관리자가 존재하지 않는다. 그래서 우리는 항상 RMI 를 사용할 때 보안 관리자를 설정할 필요가 있다.

예제5-5는 완벽한 클라이언트 프로그램을 보여준다. 클라이언트는 RMI 레지스트리에 등록된 두 Product 객체에 대한 참조를 갖고 있으며 이 두 객체에 대한 getDescription 메소드를 호출한다.

예제 5-5: ProductClient.java

```
import java.rmi.*;
import java.rmi.server.*;

public class ProductClient
{  public static void main(String[] args)
   {   System.setSecurityManager(new RMISecurityManager());
       String url = "rmi://localhost/";
          // change to "rmi://yourserver.com/"
         // when server runs on remote machine
           // yourserver.com
       try
       {   Product c1 = (Product)Naming.lookup(url + "toaster");
          Product c2 = (Product)Naming.lookup(url + "microwave");
            System.out.println(c1.getDescription());
            System.out.println(c2.getDescription());
       }
       catch(Exception e)
       {   System.out.println("Error " + e);
       }
       System.exit(0);
   }
}
```

클라이언트 수행

기본적으로, RMISecurityManager는 네트웍 통신의 설정에 의한 모든 프로그램 코드를 제한하다. 그러나 프로그램은 다음 이유에서 네트웍 통신이 가능해야 하다.

RMI 레지스트리에 도달하기 위해.

서버 객체에 접속하기 위해.

노트: 클라이언트 프로그램이 배포될 때 스터브 클래스를 로드하기 위한 권한이 필요하다. 이 문제에 대해서는 배포에 대하여 설명할 때로 미룬다.

클라이언트가 RMI 레지스트리와 서버 객체에 연결되도록 하려면 정책 파일(policy file)이 필요하다, 정책 파일에 대해서는 9장에서 상세하게 다룬다. 지금은 단지 예제 파일을 수정하는 정도이다. 이 파일은 응용 프로그램이 포트 번호가 1024인 네트웍을 통한 접근을 허용하기 위한 예이다.(RMI 포트는 기본적으로 1099이고 1024 이상의 포트 번호를 사용한다.)

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

여러분이 자바 프로그램을 수행할 때 여러분은 반드시 정책 파일에 대하여 아래와 같이 기술해야 한다.

```
java ProductClient -Djava.security.policy=client.policy
```

노트: JDK1.1에서는 RMI 클라이언트에 대한 정책 파일이 필요하지 않다.

만약 해당 RMI 레지스트리와 서버가 수행중이면, 여러분은 클라이언트를 작동시킬 수가 있다. 또는 프로그램을 종료하려면 RMI 레지스트리와 서버를 종료시키면 된다.

다음은 원격 메소드 호출 작업을 위해 필요한 단계들의 요약이다.

1. 서버와 클라이언트측에 인터페이스와 구현에 대한 소스파일을 컴파일 한다.
   javac Product*.java
2. 구현 클래스에 대한 rmic를 수행한다.
   rmic -v1.2 ProductImpl
3. RMI 레지스트리를 시작한다.
   start rmiregistry
   혹은
   rmiregistry &
4. 서버를 시작한다.
   start java ProductServer
5. 클라이언트를 시작한다.
   java -Djava.security.policy =client.policy ProductClient

다음의 내용이 출력된다.

I am a Blackwell Toaster. Buy me!

I am a ZapXpress Microwave Oven. Buy me!

모든게 완벽한 것 같지 않다. 그러나 자바가 getDescription 메소드를 호출할 때 그 내부에서 무슨 일이 일어나는지 생각해보자. 클라이언트 프로그램은 lookup 메소드를 통해 얻은 스터브 객체에 대한 참조를 가지고 있다.
클라이언트는 서버측에 있는 스케로턴 객체에게 네트웍 메시지를 보내는 getDescription 메소드를 호출한다. 스케로턴 객체는 서버상에 위치한 ProductImpl 객체의 getDescription 메소드를 호출한다. 해당 메소드는 문자열을 계산해낸다. 문자열은 스케로턴에게 반환되고 네트웍을 통해 보내진다. 그리고 스터브에 의해서 받아들여지고 결과로서 반환된다. (그림 5-5를 참조)

그림 5-5 원격 getDescription 메소드 호출

java.rmi.server.Naming

static Remote lookup(String url)
해당 URL에 대한 원격 객체를 반환한다. 만일 해당 이름이 일치하지 않으면 NotBound 예외상황을 발생시킨다.

static void bind(String name, Remote obj)
해당 name을 obj 원격 객체에 바인드시킨다. 만일 객체가 이미 바운드되었으면 Already

-BoundException을 발생시킨다.

static void unbind(String name)
name의 바운드를 해제한다. 만일 name이 현재 바운드 되어있지 않으면 NotBound 예외 상황을 발생시킨다.

static void rebind(String name, Remote obj)
name을 원격 객체에 바인딩시킨다. 기존의 바인딩을 대체한다.

static String[] list(String url)
주어진 URL에 위치한 레지스트리에 있는 URL들의 문자열 배열을 반환한다.
배열은 레지스트리에 있는 이름들을 담고 있다.

배포를 위한 준비

RMI를 이용한 응용 프로그램에서의 배치는 많은 위험 요소들을 가지고 있다. 우리는 지역적인 객체에 대한 손실에 대하여 알아본다. 3개의 디렉토리로 분리하였다.

server
download
client

서버 디렉토리는 서버가 작동하기 위한 모든 파일들을 가지고 있다. 여러분들은 나중에 이 파일들을 서버 작업을 하는 기계에 옮길 수 있다. 우리의 예제에서는 서버 디렉토리에 다음의 파일들이 있다.

server:
ProductServer.class
ProductImpl.class
Product.class
ProductImpl_Stub.class

주의: 서버 디렉토리에 스터브 클래스를 추가하라. 스터브 클래스들은 서버가 구현된 객체를 등록할 때 필요하다. 반대로 서버는 다운로드 디렉토리에 위치할 필요가 없다. 여러분이 codebase를 설정했을지라도…

---

download 디렉토리에는 클라이언트에 로드되어야 할 파일들이 있으며 또한 그 클래스에 의존하는 파일들도 위치하고 있다. 우리의 예제에서 여러분은 다운로드 디렉토리에 아래의 파일들을 추가해야 한다.

Download:

ProductImpl_Stub.class

Product.class

만약 여러분의 프로그램이 JDK1.1로 작성되었다면 여러분은 또한 스케로턴 클래스도 필요하다. 여러분은 나중에 웹서버에 이 파일을 위치시켜야 한다.

주의: 추가적으로 인터페이스 스터브에 대하여 여러분은 다운로드 디렉토리에 반드시 원격 인터페이스에 대한 클래스 파일들을 포함해야 한다. 만약 여러분이 그렇지 않으면 서버는 서버 클래스 등록시에 비정상적으로 다운될 것이다.

마지막으로 클라이언트 디렉토리는 클라이언트를 구동시키기 위해 필요한 파일들을 가지고 있다.

Client:

ProductClient.class

Product.class

Client.policy

여러분은 클라이언트 컴퓨터에 이 파일들을 배치해야 한다.

주의: 여러분은 클라이언트에 대한 원격 인터페이스를 위한 클래스 파일들을 배치해야 한다. 만약 그렇지 않으면 클라이언트는 로드되지 않는다. 또한 이 클래스 파일들은 다운로드 되지않는다.

이제 여러분은 정확히 모든 파일들을 배치하였다. 이제 여러분은 로딩되고 테스트 할 수 있다.

우리는 여러분의 컴퓨터에서 웹서버가 작동한다고 가정한다. 그렇지 않으면 여러분은 경량

서버를 ftp://java.sun.com/pub/jdk1.1/rmi/class-server.zip 으로 부터 구할 수 있다.
이 웹서버는 완전한 기능을 가지고 있지 않다. 그러나 쉽게 설치가 가능하고 클래스 파일에 대한 서비스를 지원하기에 충분하다.

첫째로 웹서버의 웹문서 디렉토리의 다운로드 디렉토리로 이동한다.
다음으로 client.policy 파일을 편집한다. 이것은 클라이언트에 대한 권한을 주어야 한다.

RMI 레지스트리에 접근과 서버 구현에 대한 1024 포트 번호 이상으로 연결한다.
HTTP포트(기본적으로 80번)로 스터브 클래스를 로드하기 위하여 연결한다.

아래와 같이 파일을 수정한다.

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

끝으로 여러분은 여러분 자신의 설정으로 테스트를 준비한다.

1. 웹서버를 구동시킨다.
2. 새로운 쉘을 실행시킨다. 현재 클래스 패스가 아무것도 설정되어 있지 않은지 확인한다. 클래스 파일을 포함하지 않은 디렉토리로 이동한다. 그리고 나서 RMI 레지스트리를 구동한다.

주의: 만약 여러분이 단지 여러분의 단일 디렉토리내에서 프로그램과 클라이언트,서버와 스터브 클래스 파일을 테스트 하기를 원한다면 여러분은 이 디렉토리 안에서 RMI 레지스트리를 구동시킬 수 있다. 그러나 배치를 위하여 클래스 패스없이, 그리고 클래스 파일을 가지고 있지 않은 디렉토리에서 등록을 확인해야 한다.

---

3. 새로운 쉘상에서 서버 디렉토리로 이동한다. 서버를 다운로드 URL을 주어 구동시킨다.

java.rmi.server.codebase property:

start java

-Djava.rmi.server.codebase = http://localhost/download/ProuductServer

---

주의: URL**의 끝이 슬래쉬**(/)**로 끝나는 것을 확인하는 매우 중요하다**.

---

4. **클라이언트 디렉토리로 이동하여 클라이언트를** java.security.policy **프로퍼티 파일에 주어진 값으로 구동한다**.

   java -Djava.security.policy=client.policy ProductClient

만약 클라이언트와 서버가 문제없이 구동되었다면, **그러면 여러분은 다음 단계로 이동할 수 있으며 클라이언트와 서버에 분리된 클래스를 배치할 준비가 되었다**. **만약 그렇지 않다면 여러분은 몇가지 조작이 필요하다**.

팁: **만약 여러분이 웹서버를 로컬에 설치하기를 원하지 않는다면**, **여러분은 클래스 로딩을 테스트하기 위하여** URL**파일을 이용할 수가 있다**. **다른 한편으로 약간의 트릭으로 설정할 수가 있다**.

```
Permission java.io.FilePermission
  "downloadDirectory", "read";
```

여러분의 정책 파일에서 다운로드 디렉토리는 인용부호로 표기된 완전한 경로(path)**로**, **마지막에 마이너스 기호까지 포함하여**(**하위 디렉토리까지 포함**) **작성되어야 한다**.
윈도우에서는 여러분은 이중 역슬래쉬로 파일명을 작성해야 한다. **아래처럼**

```
permission java.io.FilePermission
  "c:\\home\\test\\download\\-", "read";
```

혹은 유닉스에서

```
permission java.io.FilePermission
  "/home/test/download/-", "read";
```

RMI **레지스트리를 시작한 후 서버와 클라이언트를 구동시킨다**.

```
start java
   -Djava.rmi.server.codebase=file:/c:\home\test\download\ProductServer
```

혹은

```
java –Djava.rmi.server.codebase=file://home/test/download\ProductServer &
```

(URL **마지막에 슬래쉬가 있다는 것을 명심해야 한다**.)

프로그램 배포

이제 여러분은 여러분의 배포된 프로그램을 테스트 했으며, 여러분은 실제의 클라이언트와 서버에 배포할 준비가 되었다.

웹서버의 다운로드 디렉토리로 클래스들을 이동시킨다. 서버를 구동시킬 때 URL**을 확인해야 한다. 여러분의 서버에서** RMI **레지스트리와 서버가 구동되기 위한 클래스들을 서버의 디렉토리로 옮긴다.**

여러분의 서버 설정은 이제 끝났다. 그러나 여러분은 클라이언트에서 두가지를 변경해야 한다. 첫번째로, 정책파일에서 여러분의 서버 이름을 수정한다. 아래와 같이:

```
grant
{   permission java.net.SocketPermission
       “yourserver.com:1024-65535”, “connect”;
    permission java.net.SocketPermission
       “yourserver.com:80”, “connect”;
};
```

마지막으로 클라이언트 프로그램에서 실제 서버의 RMI URL**을 수정한다**.

```
String url = “rmi://yourser.com/”;
Product c1 = (Product)Naming.lookup(url + “toaster”);
…
```

그리고 나서 클라이언트를 재컴파일 한다. 만약 문제가 있다면 아래의 체크 목록에 따라서 설정을 확인한다. 아래의 체크리스트는 RMI **작동에 있어서 일반적인 문제들에 대하여 설명한다.**

팁: **사실, 여러분은 여러분의 프로그램에서** RMI URL**을 하드 코딩하길 원하지 않는다. 대신에 여러분은 이를 프로퍼티 파일에 저장할 수 있다. 우리는 이 기술을 예제** 5-13**에서 사용**

할 것이다.

RMI 배치 확인 사항

RMI 배치는 많은 모호한 에러 메시지에 의한 실패가 많기 때문에 매우 어렵다.

http://archives.java.sun.com/archives/rmi -users.html 의 RMI 토론 리스트를 보면 많은 프로그래머들이 초기 실행에 많은 실패를 한다. 만약 여러분도 같다면 여러분은 아래의 항목들에 대하여 확인을 하는 것이 많은 도움이 될 것이다. 우리는 적어도 테스트에서 한번은 잘못되는 것에 대하여 관리하고 있다.

서버 디렉토리에 스터브 클래스 파일을 넣었는가?

클라이언트 디렉토리와 다운로드 디렉토리에 인터페이스 클래스를 넣어 두었는가?

rmiregistry 를 구동할 때 클래스 패스의 설정을 해제 했는가?, 클래스 파일로부터 현재 디렉토리는 자유로운가?

클라이언트를 구동시킬 때 정책 파일을 사용하는가?, 정책 파일은 정확한 서버 명을 가지고 있는가(혹은

만약 파일을 사용할 때 : 테스트를 위한 URL에서 정책 파일에 정확히 기록되었는지? 마지막을 / 혹은 \\- 를 정확히 작성하였는지, 윈도우에서의 파일명 작성시 \\ 를 사용하는 것을 기억하는지?

여러분의 codebase에서 URL의 마지막에 / 를 넣었는가?

마지막으로 RMI 레지스트리는 모든 클래스 파일을 찾을 수 있도록 나타내어야 하며, 만약 여러분이 테스트를 위하여 변화된 클래스 파일을 제거했다면, 항상 rmiregistry를 재실행해 주어야 한다.

원격 메소드들에 전달인자 전달

여러분은 종종 원격 객체들에게 전달인자들을 전달하기를 원한다. 이번 절에서는 몇몇 함정들을 살펴보면서 이러한 일을 수행하기 위한 몇 가지 기술들을 설명한다.

비원격 객체들의 전달

자바가 서버에서 클라이언트로 원격 객체를 전달할 때, 클라이언트는 스터브를 받는다.

스터브를 사용하여 원격 메소드들의 호출을 제공하는 서버 객체를 조작할 수 있다. 그러나 객체는 서버상에 존재한다. 또한 원격 메소드 호출을 통해 단지 Remote 인터페이스의 구현들 뿐만 아니라 모든 객체를 전달하고 반환할 수 있다. 예를 들면, 앞 절에서 살펴본

getDescription 메소드는 String 객체를 반환한다. 이 문자열 객체는 서버 상에서 만들어 지고 클라이언트에게 전달돼야 한다. 왜냐하면 String 은 Remote 인터페이스의 구현이 아니기 때문에 클라이언트는 문자열 스터브 객체를 반환할 수 없다. 대신에 클라이언트는 문자열의 복사본을 얻는다. 결국, 그 다음에 클라이언트는 자신의 String 객체를 가지고 작업을 하게 된다. 이 말의 의미는 해당 문자열을 다루기 위해 서버상의 어떤 객체와도 더 이상 연결할 필요가 없다는 말이다.

원격 객체가 아닌 객체가 하나의 자바 가상 머쉰에서 다른 머쉰으로 전송될 필요가 있을 때마다, 자바 가상 머쉰은 복사본을 만들고 이를 네트웍을 가로 질러 전달한다. 이것은 로컬 메소드에서 전달인자를 전달하는 것과는 아주 다르다. 객체들을 로컬 메소드에 전달하거나 메소드 결과들로서 반환할 때, 오직 객체의 참조자만이 전달된다. 그러나 이 때 객체 참조자는 로컬 자바 가상 머쉰내에 있는 객체의 메모리 주소이다. 따라서 이 정보는 다른 자바 가상 머쉰에서는 의미가 없다.

어떻게 문자열의 복사본이 네트웍을 통해 전달되는 지를 상상하는 것은 어려운 일이 아니다. 또한 자바는 1권의 12장에서 설명한 연속화 메커니즘을 사용하여 보다 일반적인 클래스들을 일반화한 객체들의 복사본을 만들 수 있다. 현재 모든 클래스들은 기본적으로 연속화가 될 수 있다. 따라서 이 말은 프로그래머가 극명하게 복사를 제한하거나 자바 언어 그 자체에서 연속화를 금지한 클래스들 간의 정보를 복사할 수 없다는 것을 의미한다.
다음의 프로그램은 전달인자들을 복사하고 실행 중에 값들을 반환하는 것을 보여준다. 이 프로그램은 선물을 구매하게 해주는 간단한 프로그램이다. 사용자는 클라이언트상에서 프로그램을 수행시키고 선물 수령인에 대한 정보를 수집한다. 이 경우, 나이, 성별, 취미 등의 정보가 여기에 해당된다.(그림 5-6을 참조)

그림5-6: 서버로부터 제안된 제품 얻기

그런 다음 Customer 타입의 객체가 서버에게 보내진다. Customer가 원격 객체가 아니기 때문에 객체의 복사는 서버상에서 만들어진다. 서버 프로그램은 제품들의 벡터를 되돌려 보낸다. 벡터는 고객의 프로파일과 일치하는 제품들을 포함하고 있다. 그리고 항상 코아 자바 책의 복사인 한 아이템을 서버에서 해당 클라이언트로 복사한다. 1권의 제12장에서 설명한 것처럼 연속화 메커니즘은 복사된 객체내에서 참조되는 모든 객체들의 복사를 만든다. 우리의 경우, 모든 벡터 항목들의 복사를 만든다. 우리는 별도의 복잡한 과정을 추가하였다. 항목들은 실제 원격 Product 객체들이다. 따라서 수령인은 서버상의 제품들에 대한 스터브 객체들로 채워진 벡터의 복사본을 얻는다.(그림 5-7 참조)

그림 5-7 : 로컬 전달인자와 결과 객체들의 복사

요약하면, 원격 객체들은 스터브로서 네트웍을 통해 전달된다. 비원격 객체들은 복사된다. 이 모든 것은 자동적으로 이루어지며 프로그래머의 간섭을 필요로 하지 않는다. 원격 메소드가 호출될 때 마다, 스터브는 모든 전달인자 값들의 복사본을 포함하는 패키지를 만들고 전달인자들을 마샬링해주는 객체 연속화 메커니즘을 사용하여 이를 서버에게 전달한다. 서버 스케로턴은 해당 전달인자들을 언마샬링한다. 따라서 특히 이 과정은 전달인자 객체들이 큰 경우에 상당히 느리다.

자! 지금부터 완전한 프로그램을 살펴보자. 먼저 우리는 product**와** warehouse **서비스에 대한 인터페이스를 가지고 있다. 아래의 예제** 5-6**과** 5-7**을 보자**

예제 5-6: Product.java

```
import java.rmi.*;
import java.awt.*;

public interface Product
        extends Remote
{       String getDescription() throws RemoteException;

        Static final int MALE = 1;
        Static final int FEMALE = 2;
        Static final int BOTH = MALE + FEMALE }
```

예제 5-7: Warehouse.java

```
import java.rmi.*;
import java.util.*;

public interface Warehouse
        extends Remote
{       public Vector find(Customer c)
          throws RemoteException;
}
```

예제5-8은 제품 서비스를 위한 구현을 보여준다. 제품들은 설명서와 가격범위, 공급자의 성별(남,여,모두) 그리고 취미를 저장하고 있다. 이 클래스가 Product 인터페이스 내부에서 알려진 getDescription 메소드를 구현했다는 사실을 기억하라. 또한 이 클래스에는 Product 인터페이스의 일부가 아닌 match 라는 다른 메소드가 구현되어 있다. match 메소드는 로컬 메소드의 예이다. 이 메소드는 원격지에서가 아니라 오직 로컬 프로그램에서만 호출될 수 있다. 따라서 match 메소드가 로컬이기 때문에 RemoteException을 준비할 필요가 없다.

예제5-9는 Customer 클래스에 대한 코드를 보여준다. Customer 클래스는 원격 클래스가 아니기 때문에 원격 실행을 할 수가 없다. 그러나, 이 클래스는 연속화 객체이기 때문에 이 클래스의 객체들은 하나의 자바 가상 머쉰에서 다른 가상 머쉰으로 전송이 가능하다.

예제 5-11은 Warehouse 서비스의 구현을 보여준다. ProductImpl 클래스처럼 WarehouseImpl 클래스는 로컬과 원격 메소드를 가지고 있다. add 메소드는 로컬이다. 이 메소드는 서버에 의해서 사용되고 제품들을 창고에 넣는다. find 메소드는 원격 메소드이다. 이 메소드는 창고내에 있는 아이템들을 찾기 위해 사용된다.

Customer 객체가 실제로 어떻게 복사되는 보여주기 위해, warehouseImpl 클래스의 find 메소드는 실제로 전달받은 고객 객체를 제거한다. 원격 메소드가 반환할 때, WarehouseClient 는 서버에게 전달되는 고객 객체를 출력해 준다. 여러분이 보게 되는 것처럼, 해당 객체는 변하지 않는다. 서버는 오직 해당 복사본만을 제거한다. 이 경우 clear 기능은 이 목적을 위해서는 유용하지 않다. 이것은 오직 전달인자로서 로컬 객체들이 전달될 때 이를 복사하는 예를 보여준다.

일반적으로, ProductImpl과 WarehouseImpl 같은 서버 클래스들의 메소드는 동기화 되어야 한다. 그래야 복수 개의 클라이언트 스터브들이 동시에 서버 객체에게 호출할 수 있다. 비록 몇몇 메소드들은 서버의 상태를 변경한다. 메소드들의 동기화에 대한 자세한 정보는 2장을 보라. 예제 5-11에서, 우리는 WarehouseImpl 클래스의 메소드들을 동기화 한다. 왜냐하면 로컬 add 와 원격 find 메소드들은 동시에 호출되기 때문이다. 우리는 ProductImpl 클래스의 메소드들을 동기화 시키지 않는다. 왜냐하면 제품 서버 객체들은 그들의 상태를 변경하지 않기 때문이다.
예제 5-12는 객체를 생성하고 이를 부트스트랩 레지스트리 서비스에 등록하는 서버 프로그램을 보여준다.

노트: 여러분은 반드시 레지스트리와 서버 프로그램을 시작해야 하고 클라이언트를 시작하

기 전에 이 두 개가 수행되고 있도록 유지해야 한다는 사실을 명심해야 한다.

예제 5-13은 클라이언트에 대한 코드를 보여준다. 사용자가 submit 버튼을 눌렀을 때 새로운 고객 객체가 생성되고 원격 find 메소드에게 전달된다. 그 다음(서버 객체의 clear 호출이 고객 레코드에 영향을 끼치지 못하도록 하기위해) 고객 레코드가 텍스트 영역에 출력된다. 마지막으로, 벡터내에 반환된 제품들의 설명서가 텍스트 영역에 보태진다. 각 getDescription 호출은 원격 메소드 호출이라는 것을 명심하라.

팁: 만약 여러분이 아래와 같이 서버를 구동시키면

java **–Djava.rmi.server.logCalls=ture** WarehouseServer

모든 원격 호출에 대한 서버 로그를 콘솔에 남긴다. RMI 통신에 대한 좋은 경험을 위해서 시도해 보라

예제 5-8: ProductImpl.java

```
import java.rmi.*;
import java.rmi.server.*;

public class ProductImpl
   extends UnicastRemoteObject
   implements Product
{  public ProductImpl(String n, int s, int age1, int age2,
      String h)
      throws RemoteException
   {  name = n;
      ageLow = age1;
      ageHigh = age2;
      sex = s;
      hobby = h;
   }

   public boolean match(Customer c) // local method
```

```
   {  if (c.getAge() < ageLow || c.getAge() > ageHigh)
         return false;
      if (!c.hasHobby(hobby)) return false;
      if ((sex & c.getSex()) == 0) return false;
      return true;
   }

   public String getDescription()
      throws RemoteException
   {  return "I am a " + name + ". Buy me!";
   }

   private String name;
   private int ageLow;
   private int ageHigh;
   private int sex;
   private String hobby;
}
```

예제 5-9:Customer.java

```
import java.io.*;

public class Customer implements Serializable
{  public Customer(int theAge, int theSex, String[] theHobbies)
   {  age = theAge;
      sex = theSex;
      hobbies = theHobbies;
   }

   public int getAge() { return age; }

   public int getSex() { return sex; }

   public boolean hasHobby(String aHobby)
   {  if (aHobby == "") return true;
```

```
      for (int i = 0; i < hobbies.length; i++)
         if (hobbies[i].equals(aHobby)) return true;

      return false;
   }

   public void reset()
   {  age = 0;
      sex = 0;
      hobbies = null;
   }

   public String toString()
   {  String result = "Age: " + age + ", Sex: ";
      if (sex == Product.MALE) result += "Male";
      else if (sex == Product.FEMALE) result += "Female";
      else result += "Male or Female";
      result += ", Hobbies:";
      for (int i = 0; i < hobbies.length; i++)
         result += " " + hobbies[i];
      return result;
   }

   private int age;
   private int sex;
   private String[] hobbies;
}
```

예제 5-10: Warehouse.java

```
import java.rmi.*;
import java.util.*;

public interface Warehouse
   extends Remote
```

```
{  public Vector find(Customer c)
      throws RemoteException;
}
```

예제 5-11: WarehouseImpl.java

```
import java.rmi.*;
import java.util.*;
import java.rmi.server.*;


public class WarehouseImpl
   extends UnicastRemoteObject
   implements Warehouse
{  public WarehouseImpl()
      throws RemoteException
   {  products = new Vector();
   }

   public synchronized void add(ProductImpl p) // local method
   {  products.add(p);
   }

   public synchronized Vector find(Customer c)
      throws RemoteException
   {  Vector result = new Vector();
      for (int i = 0; i < products.size(); i++)
      {  ProductImpl p = (ProductImpl)products.get(i);
         if (p.match(c)) result.add(p);
      }
      result.add(new ProductImpl("Core Java Book",
         0, 200, Product.BOTH, ""));
      c.reset();
      return result;
   }
```

```
   private Vector products;
}
```

예제 5-12: WarehouseServer.java

```
import java.rmi.*;
import java.rmi.server.*;

public class WarehouseServer
{  public static void main(String[] args)
   {  try
      {  System.out.println
            ("Constructing server implementations...");

         WarehouseImpl w = new WarehouseImpl();
         fillWarehouse(w);

         System.out.println
            ("Binding server implementations to registry...");

         Naming.rebind("central_warehouse", w);

         System.out.println
            ("Waiting for invocations from clients...");
      }
      catch(Exception e)
      {  System.out.println("Error: " + e);
      }
   }

   public static void fillWarehouse(WarehouseImpl w)
      throws RemoteException
   {  w.add(new ProductImpl("Blackwell Toaster",
         Product.BOTH, 18, 200, "Household"));
```

```
        w.add(new ProductImpl("ZapXpress Microwave Oven",
           Product.BOTH, 18, 200, "Household"));
        w.add(new ProductImpl("Jimbo After Shave",
           Product.MALE, 18, 200, "Beauty"));
        w.add(new ProductImpl("Handy Hand Grenade",
           Product.MALE, 20, 60, "Gardening"));
        w.add(new ProductImpl("DirtDigger Steam Shovel",
           Product.MALE, 20, 60, "Gardening"));
        w.add(new ProductImpl("U238 Weed Killer",
           Product.BOTH, 20, 200, "Gardening"));
        w.add(new ProductImpl("Van Hope Cosmetic Set",
           Product.FEMALE, 15, 45, "Beauty"));
        w.add(new ProductImpl("Persistent Java Fragrance",
           Product.FEMALE, 15, 45, "Beauty"));
        w.add(new ProductImpl("Rabid Rodent Computer Mouse",
           Product.BOTH, 6, 40, "Computers"));
        w.add(new ProductImpl
           ("Learn Bad Java Habits in 21 Days Book",
           Product.BOTH, 20, 200, "Computers"));
        w.add(new ProductImpl("My first Espresso Maker",
           Product.FEMALE, 6, 10, "Household"));
        w.add(new ProductImpl("JavaJungle Eau de Cologne",
           Product.FEMALE, 20, 200, "Beauty"));
        w.add(new ProductImpl("Fast/Wide SCSI Coffee Maker",
           Product.MALE, 20, 50, "Computers"));
        w.add(new ProductImpl("ClueLess Network Computer",
           Product.BOTH,6, 200, "Computers"));
    }
}
```

예제 5-13: WarehouseClient.java

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

```
import java.awt.event.*;
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import javax.swing.*;

public class WarehouseClient
{  public static void main(String[] args)
   {  JFrame frame = new WarehouseClientFrame();
      frame.show();
   }
}

class WarehouseClientFrame extends JFrame
   implements ActionListener
{  public WarehouseClientFrame()
   {  initUI();

      System.setSecurityManager(new RMISecurityManager());
      try
      {  Properties props = new Properties();
         String fileName = "WarehouseClient.properties";
         FileInputStream in = new FileInputStream(fileName);
         props.load(in);
         String url = props.getProperty("warehouse.url");
         if (url == null)
            url = "rmi://localhost/central_warehouse";

         centralWarehouse = (Warehouse)Naming.lookup(url);
      }
      catch(Exception e)
      {  System.out.println("Error: Can't connect to warehouse. " + e);
      }
   }
```

```
private void callWarehouse(Customer c)
{  try
   {  Vector recommendations = centralWarehouse.find(c);
      result.setText(c + "\n");
      for (int i = 0; i < recommendations.size(); i++)
      {  Product p = (Product)recommendations.get(i);
         String t = p.getDescription() + "\n";
         result.append(t);
      }
   }
   catch(Exception e)
   {  result.setText("Error: " + e);
   }
}


public void actionPerformed(ActionEvent evt)
{  Object[] hobbyObjects = hobbies.getSelectedValues();
   String[] hobbyStrings = new String[hobbyObjects.length];
   System.arraycopy(hobbyObjects, 0, hobbyStrings, 0,
      hobbyObjects.length);
   Customer c = new Customer(Integer.parseInt(age.getText()),
      (male.isSelected() ? Product.MALE : 0)
         + (female.isSelected() ? Product.FEMALE : 0),
      hobbyStrings);
   callWarehouse(c);
}

private void initUI()
{  setTitle("WarehouseClient");
   setSize(300, 300);
   addWindowListener(new WindowAdapter()
      {  public void windowClosing(WindowEvent e)
         {  System.exit(0);
         }
      } );
```

```
getContentPane().setLayout(new GridBagLayout());

GridBagConstraints gbc = new GridBagConstraints();
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.weightx = 100;
gbc.weighty = 0;

add(new JLabel("Age:"), gbc, 0, 0, 1, 1);
age = new JTextField(4);
age.setText("20");
add(age, gbc, 1, 0, 1, 1);

male = new JCheckBox("Male", true);
female = new JCheckBox("Female", true);
add(male, gbc, 0, 1, 1, 1);
add(female, gbc, 1, 1, 1, 1);

gbc.weighty = 100;
add(new JLabel("Hobbies"), gbc, 0, 2, 1, 1);
String[] choices = { "Gardening", "Beauty",
   "Computers", "Household", "Sports" };
gbc.fill = GridBagConstraints.BOTH;
hobbies = new JList(choices);
add(new JScrollPane(hobbies), gbc, 1, 2, 1, 1);

gbc.weighty = 0;
gbc.fill = GridBagConstraints.NONE;
JButton submitButton = new JButton("Submit");
add(submitButton, gbc, 0, 3, 2, 1);
submitButton.addActionListener(this);

gbc.weighty = 100;
gbc.fill = GridBagConstraints.BOTH;
result = new JTextArea(4, 40);
result.setEditable(false);
```

```
      add(result, gbc, 0, 4, 2, 1);
   }

   private void add(Component c, GridBagConstraints gbc,
      int x, int y, int w, int h)
   {  gbc.gridx = x;
      gbc.gridy = y;
      gbc.gridwidth = w;
      gbc.gridheight = h;
      getContentPane().add(c, gbc);
   }

   private Warehouse centralWarehouse;
   private JTextField age;
   private JCheckBox male;
   private JCheckBox female;
   private JList hobbies;
   private JTextArea result;
}
```

원격 객체들 전달하기

서버에서 클라이언트로 원격 객체들을 전달하는 것은 간단하다. 클라이언트는 스터브 객체를 받고 이것을 원격 인터페이스와 같은 타입을 갖는 객체 변수에 저장한다. 지금부터 이 변수를 통해 서버상에 존재하는 실제 객체에 접근할 수 있다. 클라이언트는 이 변수를 자신의 로컬 머쉰내에 복사할 수 있다- 이 모든 복사본들은 같은 스터브에 대해 손쉽게 참조하게 해준다.

오직 *원격 인터페이스들만이* 스터브를 통해 접근할 수 있다는 사실은 중요하다. 원격 인터페이스는 Remote 를 확장한 모든 인터페이스이다. 모든 로컬 메소드들은 스터브를 통해 접근할 수 없다. 로컬 메소드는 원격 인터페이스에 정의되지 않은 모든 메소드이다. 로컬 메소드들은 오직 실제 객체를 포함하고 있는 가상 머쉰상에서만 수행된다.

스터브들은 오직 원격 인터페이스를 구현한 클래스들로 부터 생성된다. 그리고 인터페이스내에 명시된 메소드들 만이 스터브 클래스들에 의해서 제공된다. 만일 유추된 클래스가 원

격 인터페이스를 구현하지 않고 기본 클래스가 이를 구현하고 유추된 클래스의 객체가 원격 메소드에게 전달되면, 오직 기본 클래스의 메소드들만이 접근할 수 있다. 보다 확실히 이해하기 위해, 다음의 예제를 고려하자. 우리는 ProductImpl에서 BookImpl 클래스를 유추하였다:

```
class BookImpl extends productImpl
{       public BookImpl(String title , String theISBN,
                int sex, int age1, int age2, String hobby)
        {       super( title + "Book" , sex, age1, age2, hobby);
                IDBN = theISBN;
        }
        public String getStockCode() { return ISBN; }
        String ISBN;
}
```

지금부터 우리는 전달인자나 반환 값으로 책 객체를 원격 메소드에게 전달한다고 가정하자. 수령인은 스터브 객체를 얻는다. 그러나 해당 스터브는 책 스터브가 아니다. 대신에 이것은 ProductImpl 기본 클래스에 대한 스터브이다. 왜냐하면 ProductImpl 클래스가 원격 인터페이스를 구현한 유일한 클래스이기 때문이다(그림 5-8 참조)

그림 5-8 : 상속 다이어그램

원격 클래스는 복수 개의 인터페이스들을 구현할 수 있다. 예를 들면, BookImpl 클래스는 Product외에 두 번째 인터페이스를 구현할 수 있다. 여기서 우리는 StockUnit 이라는 원격 인터페이스를 정의하고 이 인터페이스를 구현하는 BookImpl 클래스를 작성할 것이다.

```
interface StockUnit extends Remote
{       public String getStockCode() throws RemoteException;
}

class BookImpl extends ProductImpl implements StockUnit
{       public BookImpl(String titile , String theISBN,
                int sex, int age1, int age2, String hobby)
                throws RemoteException
        {       super(title + " Book " , sex, age1, age2, hobby);
                ISBN = theISBN;
```

```
        }
        public String getStockCode() throws RemoteException
        {       return ISBN;        }
        private String ISBN;
}
```

그림 5-9 는 상속 다이어그램을 보여준다.

그림 5-9 : 상속 다이어그램

자바가 책 객체를 원격 메소드에 전달할 때, 수령인은 Product와 Stockunit 클래스 모두에 포함되어 원격 메소드들에게 접근하는 스터브를 얻는다. 사실, 여러분은 instanceof 메소드를 사용하여 어떤 원격 객체가 인터페이스를 구현한 것인지 발견할 수 있다.

다음은 아주 전형적인 상황이다. 여러분이 Product 타입의 변수를 통해 원격 객체를 전달받는다고 가정하자.

```
Vector result = centralWarehouse.find();
for( int i = 0; i < result.size(); i++)
{       Product p = (Product) result.elementAt(i);
        ....
}
```

현재 원격 객체는 책일 수도 아닐 수도 있다. 우리는 해당 객체인지 아닌지를 알아내기 위해 instanceof 를 사용하고자 한다. 그러나 우리는 테스트를 할 수 없다.

```
if ( p instanceof BookImpl) // 잘못됐음
{       BookImpl b = (BookImpl) p;
        ...
}
```

객체 p 는 스터브 객체를 참조한다. 그리고 BookImpl은 서버 객체의 클래스이다. 우리는 스터브 객체를 BookImpl_Stub로 형변환한다.

```
if ( p instanceof BookImpl_Stub) // 잘못됐음
{       BookImpl_Stub b = (BookImpl_Stub) p;  // 유용하지 않음
        ...
}
```

그러나 이것은 우리에게 아주 좋아보이지 않는다. 스터브들은 RMI 메커니즘에 의해서 내부적으로 사용하는 rmic 프로그램에 의해서 기계적으로 생성된다. 그리고 클라이언트들은 이것들에 대해 생각할 필요가 없다. 대신에 우리는 두번째 인터페이스에 대해 형변환을 한다:

```
if ( p instanceof  StockUnit)
{       StockUnit s = (StockUnit)p;
        String c = s.getStockCode();
        ...
}
```

이 코드는 StockUnit 인터페이스 구현을 참조하는 p가 스터브 객체인지 아닌지를 검사한다. 만일 그렇다면 스터브 객체는 해당 인터페이스의 getStockCode 원격 메소드를 호출한다.

요약하면:

만일 원격 인터페이스를 구현한 클래스에 속하는 객체가 원격 메소드에 전달되면, 원격 메소드는 스터브 객체를 받는다.

여러분은 스터브 객체를 원격 인터페이스를 구현한 어떤 클래스로도 형변환 할 수 있다.

여러분은 이러한 인터페이스내에 정의된 모든 원격 메소드들을 호출할 수 있다. 그러나 여러분은 스터브를 통해 어떠한 로컬 메소드도 호출할 수 없다.

셋에서 원격 객체들 사용하기

2장에서 살펴본 것처럼 셋에 삽입된 객체들은 equals 메소드를 오버라이드 해야 한다. 해쉬 셋이나 해쉬 테이블의 경우, hashCode 메소드 또한 정의돼야 한다. 그러나 원격 객체를 비교하려 할 때 문제가 발생한다. 만일 두개의 원격 객체들이 같은 내용을 가지고 있는지 알고자 한다면, 객체들을 포함하고 있는 서버들에 접속하고 Object 클래스의 equals 메소드는 RemoteException을 투구하도록 선언되지 않았다. 반면에, 원격 인터페이스내에 있는 모든 메소드들은 RemoteException을 선언해야 한다. 하위 클래스의 메소드는 상위 클래스에 정의된 대체가능한 메소드들 보다 적은 수의 예외상황을 투구 할 수 없기 때문에, 원격 인터페이스내에서 equals 메소드를 정의할 수 없다. 마찬가지의 상황이 hashCode에도 적용된다.

대신에, 여러분은 모든 스터브(stub)와 서버(server)객체들에 대한 기본 클래스인 RemoteObject 클래스내에서 equals과 hashCode를 재정의하는 것에 의존해야 하다. 이러한 메소드들은 객체의 실제 내용을 살펴보지 않고 단지 서버 객체들의 위치를 검사한다. 같은 서버 객체를 참조하는 두개의 스터브들은 equals 메소드에 의해서 동일하게 인식된다. 다른 서버 객체들을 참조하는 두개의 스터브들은 비록 이들 객체들이 동일한 내용을 갖고 있더라도 결코 동일하지 않다. 이와 유사하게, 해쉬 코드는 오직 객체 식별자를 통해 계산되어 진다. 다른 서버 객체들을 참조하는 스터브들은 비록 서버 객체들이 동일한 내용을 갖더라도 다른 해쉬 코드들을 가질 것이다.

이러한 제한은 오직 스터브들과 관련되어 있다. 여러분은 서버 객체 클래스들을 위해 equals과 hashCode를 재정의 할 수 있다. 이러한 메소드들은 서버 객체들을 서버상의 해쉬 테이블내에 삽입할 때 호출된다. 그러나 이러한 메소드들은 결코 스터브들을 비교하거나 해싱할 때 호출되지 않는다.
클라이언트 행동과 서버 행동(behavior) 사이의 차이점을 분명히 이해하기 위해 그림 5-10의 상속 다이어그램을 참조하라.

그림 5-10: equals와 hashCode 메소드들의 상속

RemoteObject 클래스는 스터브와 서버 클래스들 모두의 기본 클래스이다. 스터브상에서 equals과 hashCode 메소드를 오버라이드 할 수 없다. 왜냐하면 스터브들은 자동적으로 생성되기 때문이다. 서버상에서, 여러분은 구현 클래스들에 대한 메소드를 오버라이드 할 수 있다. 그러나 이들 메소드들은 오직 서버상에서 로컬로만 사용된다. 만일 이러한 메소들을 오버라이드 하면 구현 객체들과 스터브 객체들은 더 이상 동일하게 고려되지 않는다.

요약: 여러분은 해쉬 테이블내에서 스터브 객체들을 사용할 수 있다. 그러나 동등 검사와 해싱(hashing)은 원격 객체의 내용을 고려하지 않는다는 점을 명심해야 한다.

원격 객체 복제
스터브들은 clone 메소드들 사용할 수 없다, 그래서 여러분들은 스터브상의 복제 호출에 의한 원격 객체의 복제를 할 수 없다. 그 이유는 매우 기술적이다. 만약 clone 메소드가 서버의 구현 객체에 대해 복제가 가능하다면 clone 메소드는 RemoteException 을 투구해야 한다.
그러나 상위 클래스 객체의 clone 메소드는 CloneNotSupportedException 예외를 발생하지 않

는다.
이러한 제한은 이전 절에서와 동일하다. equals와 hashCode 메소드는 원격 객체에 대한 값을 전혀 찾을 수 없으며 단지 스터브의 참조를 비교할 수 있다. 만약 여러분이 원격 객체에 대한 다른 참조를 원한다면 여러분은 단지 스터브 변수를 복사할 수가 있다. 그러므로 clone 메소드는 간단하지만 스터브에 대한 정의는 아니다.

만약 여러분이 원격 객체에 대한 복제를 원한다면 다른 방법을 사용해야 한다. 말하자면 remoteClone 메소드 이다. 원격 객체 서비스에 정의된 인터페이스내에 삽입한다. 물론 이 메소드는 RemoteException 예외를 투구한다. 구현 클래스상에서, 간단하게 정의된 remoteClone 메소드로 clone를 호출하고 구현 객체의 복제를 반환한다.

```
interface Product extends Remote
{  public Object remoteClone()
       throws  RemoteException,CloneNotSupportedException;
       ….
}

class ProductImpl extends UnicastRemoteObject
            implements Product
{   publc Object remoteClone()
            throws CloneNotSupportedException
    {   return clone(); }
}
```

부적합한 원격 전달인자들
우리의 쇼핑 응용 프로그램에 각 선물의 그림을 보여주는 기능을 추가한다고 가정하자. 왜 Product 인터페이스에 다음의 메소드를 추가하는 것이 간단하지 않는가?

```
void paint(Graphics g) throws RemoteException
```

불운하게도 이 코드는 제대로 동작하지 않는다. 그리고 그 이유가 무엇인지 이해하는 것이 아주 중요하다. Graphics 클래스는 원격 인터페이스를 구현하지 않는다. 그러므로 Graphics 타입의 객체들은 복사에 의해서 전달된다. 실제로, Graphics는 추상화 클래스이다. paint 메소드는 전달인자나 Component 클래스의 getGraphics 메소드의 반환 값으로서 얻어지는 Graphics 객체는 실제 특정 플랫폼의 그래픽 컨텍스를 구현한 몇몇 하위 클래스에 속한다. 다시 말하면 이러한 객체들은 네이티브 그래픽 코드와 상호작동 할 필요가 있다. 이들 객체는 네이티

브 그래픽 메소드들에 의해서 필요로 하는 메모리 블럭들에 대한 포인터들을 저장한다. 자바에는 포인터가 없다. 그래서 이들 포인터는 그래픽 객체내에 정수로 저장되고 다시 네이티브 메소드들에서 포인터로 형변환된다.

그러므로 그래픽 객체를 복사하는 것은 아무 의미가 없다. 무엇보다도, 모든 목표 시스템이 다를 수가 있다는 데 있다. 예를 들면, 클라이언트가 윈도우상에서 수행되고 서버가 유닉스상에서 X11을 수행하면, 서버는 윈도우 그래픽을 렌더링 할 때 필요한 네이티브 메소드를 가지고 있지 않다. 그러나 비록 서버와 클라이언트가 같은 그래픽 시스템을 갖고 있더라도, 포인터 값들은 서버상에서 합법적일 수 없다.

대신에, 만일 서버가 이미지를 클라이언트로 보내려고 한다면, 네트웍상에서 데이터를 전송하기위한 몇가지 다른 메커니즘을 익혀야 만 한다. 그러나 이 메커니즘들이 알려지면서, 실제로 이미지를 다루는 기술은 상당히 어려운 것이라고 판명되었다. Image 클래스는 단지 Graphics 클래스로서 디바이스 독립적이다. 우리는 JPEG 형태의 연속된 바이트들로서 이미지 데이터를 전송하였다. 그러나 AWT 패키지내에는 JPEG 데이터 블럭을 이미지로 바꾸는 메소드가 없다(현제, 이것은 오직 sun.awt.image 패키지에서 아직 공개되지 않는 클래스를 사용해서 처리할 수 있다.) 다음 절에서, 우리는 어떻게 이 문제를 보다 현실적인 방법으로 해결하는지 소개할 것이다. 이 현실적인 방법은 클라이언트에게 URL을 보내고 URL을 통해 이미지를 읽어오는 Applet 클래스의 메소드를 사용하는 것이다.

애플릿에서 RMI 사용하기

애플릿에서 RMI를 사용하는 경우에 특별한 주의를 기울여야 한다. 애플릿들은 부라우저 내부에서 수행되기 때문에 그들 자신의 보안 관리자를 가지고 있다. 그래서 우리는 클라이언트상에서 StubSecurityManager를 사용하지 않는다.

우리는 스터브와 서버 파일들을 위치시켜야 할 곳이 어디인지 주의를 기울여야 한다. APPLET 태그를 사용하는 웹페이지를 개방하려는 부라우저를 생각해보자. 부라우저는 해당 태크내에서 참조되는 클래스 파일과 실행되는 동안 필요로 하는 다른 모든 클래스 파일들을 로드한다. 클래스 파일들은 웹페이지를 전송받은 호스트와 같은 호스트에서 로드된다. 애플릿의 보안 제약때문에, 애플릿은 오직 최초 다운로드된 호스트에서만 네트웍 연결을 만들 수 있다. 그러므로 서버 객체들은 웹 페이지와 같은 호스트에 존재해야 한다. 다시 말해, 같은 호스트는 다음을 갖고 있어야 한다:

- 웹 페이지
- 애플릿 코트
- 스터브 코드
- 스케로턴과 서버 객체들

- **부트스트랩 레지스트리**

다음은 우리의 쇼핑 프로그램을 보다 확장한 애플릿이다, 앞서 살펴본 응용 프로그램처럼, 애플릿은 고객 정보를 얻은 후 구매하고자 하는 것과 일치하는 것을 제안해 준다, 사용자가 제안된 것들 중에 하나를 선택할 때, 애플릿은 아이템의 이미지를 출력한다. 앞서 언급한 것처럼, 이미지 객체들이 로컬 그래픽 시스템에서 사용하는 형태로 저장되어 있기 때문에 이미지 객체를 서버에서 클라이언트로 전송하는 것은 불가능하다. 대신에 서버는 간단하게 이미지 파일의 이름을 문자열 형태로 클라이언트에게 전송한다. 그리고 우리는 이미지를 가져오기 위해 Applet 클래스의 getImage 메소드를 사용한다.(그림 5-11 참조)

그림 5-11 : 창고 애플릿

다음은 이러한 종류의 상황에서 코드를 분산시키는 방법이다.

- java.rmi.regisry.RegistryImpl: 호스트상의 모든 곳(애플릿이 시작하기 전에 수행하고 있어야 한다.)
- WarehouseServer: 호스트상의 모든 곳(애플릿이 시작하기 전에 수행하고 있어야 한다.).
- WarehouseImpl: 스케로턴들(WarehouseServer가 발견할 수 있는 한, 호스트 상의 어느 곳이든 가능하다).
- WarehouseApplet: Applet 태그 내에서 참조되는 디렉토리
- 스터브들: WarehouseApplet과 같은 디렉토리내에 존재해야 한다.

애플릿은 애플릿을 포함하는 동일한 호스트 상의 RMI 레지스트리를 찾는다. 먼저 호스트를 찾기 위해서는 getCodeBase와 getHost 메소드를 사용한다.

```
String url = getCodeBase().getHost();
url = "rmi://" + url;
centralWarehouse = (Warehouse)Naming.lookup(url + "/central_warehouse");
```

예제 5-14는 애플릿에 대한 코드이다. 애플릿은 보안 관리자를 설치하지 않고 애플릿을 포함하고 있는 동일한 호스트상의 부트스트랩 레지스트리를 참조한다는 사실에 주목하라. 예제 5-15는 서버의 소스 코드이다.

예제 5-14 : WarehouseApplet.java

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

```
import java.awt.event.*;
import java.applet.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import javax.swing.*;

public class WarehouseApplet extends JApplet
   implements ActionListener
{  public void init()
   {  initUI();
      String url = getCodeBase().getHost();
      url = "rmi://" + url;
      try
      {  centralWarehouse = (Warehouse)Naming.lookup(url
            + "/central_warehouse");
      }
      catch(Exception e)
      {  showStatus("Error: Can't connect to warehouse. " + e);
      }
   }

   private void callWarehouse(Customer c)
   {  try
      {  products = centralWarehouse.find(c);
         descriptionListModel.clear();
         for (int i = 0; i < products.size(); i++)
         {  Product p = (Product)products.get(i);
            Image productImage
               = getImage(getCodeBase(), p.getImageFile());
            Icon productIcon = new ImageIcon(productImage);
            descriptionListModel.addElement(productIcon);
         }
      }
      catch(Exception e)
      {  showStatus("Error: " + e);
```

```
   }
}

private void initUI()
{  getContentPane().setLayout(new GridBagLayout());

   GridBagConstraints gbc = new GridBagConstraints();
   gbc.fill = GridBagConstraints.HORIZONTAL;
   gbc.weightx = 100;
   gbc.weighty = 0;

   add(new JLabel("Age:"), gbc, 0, 0, 1, 1);
   age = new JTextField(4);
   age.setText("20");
   add(age, gbc, 1, 0, 1, 1);

   male = new JCheckBox("Male", true);
   female = new JCheckBox("Female", true);
   add(male, gbc, 0, 1, 1, 1);
   add(female, gbc, 1, 1, 1, 1);

   gbc.weighty = 100;
   add(new JLabel("Hobbies"), gbc, 0, 2, 1, 1);
   String[] choices = { "Gardening", "Beauty",
      "Computers", "Household", "Sports" };
   gbc.fill = GridBagConstraints.BOTH;
   hobbies = new JList(choices);
   add(new JScrollPane(hobbies), gbc, 1, 2, 1, 1);

   gbc.weighty = 0;
   gbc.fill = GridBagConstraints.NONE;
   submitButton = new JButton("Submit");
   add(submitButton, gbc, 0, 3, 2, 1);
   submitButton.addActionListener(this);
```

```
      gbc.weighty = 100;
      gbc.fill = GridBagConstraints.BOTH;
      descriptionListModel = new DefaultListModel();
      descriptions = new JList(descriptionListModel);
      add(new JScrollPane(descriptions), gbc, 0, 4, 2, 1);
   }

   private void add(Component c, GridBagConstraints gbc,
      int x, int y, int w, int h)
   {  gbc.gridx = x;
      gbc.gridy = y;
      gbc.gridwidth = w;
      gbc.gridheight = h;
      getContentPane().add(c, gbc);
   }

   public void actionPerformed(ActionEvent event)
   {  Object[] hobbyObjects = hobbies.getSelectedValues();
      String[] hobbyStrings = new String[hobbyObjects.length];
      System.arraycopy(hobbyObjects, 0, hobbyStrings, 0,
         hobbyObjects.length);
      Customer c = new Customer(Integer.parseInt(age.getText()),
         (male.isSelected() ? Product.MALE : 0)
         + (female.isSelected() ? Product.FEMALE : 0),
         hobbyStrings);
      callWarehouse(c);
   }

   private Warehouse centralWarehouse;
   private JTextField age;
   private JCheckBox male;
   private JCheckBox female;
   private JList hobbies;
   private JButton submitButton;
   private JList descriptions;
   private DefaultListModel descriptionListModel;
```

```
   private Vector products;
}
```

예제5-15 : WareHouseServer.java

```
import java.rmi.*;
import java.rmi.server.*;

public class WarehouseServer
{  public static void main(String[] args)
   {  try
      {  System.out.println
            ("Constructing server implementations...");

         WarehouseImpl w = new WarehouseImpl();
         fillWarehouse(w);

         System.out.println
            ("Binding server implementations to registry...");

         Naming.rebind("central_warehouse", w);

         System.out.println
            ("Waiting for invocations from clients...");
      }
      catch(Exception e)
      {  System.out.println("Error: " + e);
      }
   }

   public static void fillWarehouse(WarehouseImpl w)
      throws RemoteException
   {  w.add(new ProductImpl("Blackwell Toaster",
         Product.BOTH, 18, 200, "Household", "toaster.jpg"));
```

```
        w.add(new ProductImpl("Jimbo After Shave",
           Product.MALE, 18, 200, "Beauty", "shave.jpg"));
        w.add(new ProductImpl("U238 Weed Killer",
           Product.BOTH, 20, 200, "Gardening", "weed.jpg"));
        w.add(new ProductImpl("Rabid Rodent Computer Mouse",
           Product.BOTH, 6, 40, "Computers", "rodent.jpg"));
        w.add(new ProductImpl
           ("Learn Bad Java Habits in 21 Days Book",
           Product.BOTH, 20, 200, "Computers", "book.jpg"));
        w.add(new ProductImpl("JavaJungle Eau de Cologne",
           Product.FEMALE, 20, 200, "Beauty", "cologne.jpg"));
        w.add(new ProductImpl("Fast/Wide SCSI Coffee Maker",
           Product.MALE, 20, 50, "Computers", "coffee.jpg"));
        w.add(new ProductImpl("ClueLess Network Computer",
           Product.BOTH, 6, 200, "Computers", "computer.jpg"));
        w.add(new ProductImpl("Digging Dinosaur",
           Product.BOTH, 6, 200, "Gardening", "dino.jpg"));
        w.add(new ProductImpl("Fantastic Fan",
           Product.BOTH, 6, 200, "Household", "fan.jpg"));
        w.add(new ProductImpl("Japanese Cat",
           Product.BOTH, 6, 200, "Gardening", "cat.jpg"));
        w.add(new ProductImpl("Ms. Frizzle Curling Iron",
           Product.FEMALE, 6, 200, "Beauty", "curl.jpg"));
    }
}
```

Java IDL 과 CORBA

RMI와 다르게 CORBA는 여러분이 자바 객체와 다른 언어로 작성된 객체들 간에 객체 호출을 할 수 있게 해준다. CORBA는 두 클라이언트와 서버간의 사용할 수 있는 ORB(Object Request Broker)에 의존적이다. ORB는 객체사이의 CORBA 통신에 대한 전반적인 번역을 담당한다. CORBA 2 는 보다 많은 서비스가 추가되었으며 ORB는 다양한 종류의 관리 작업을 수행할 수 있다. 이 범위는 서비스의 구동으로부터 프로세스 진행, 생명 주기 서비스(life cycle service), 명명 서비스(naming service)에 의한 객체의 생성,복사,이동 또는 소멸을 허용하며 객체 검색에 의한 그들의 이름을 알 수가 있다.

자바 2 플랫폼은 CORBA 2 표준에 근거한 ORB를 완전하게 구현하였다. 그래서 모든 응용프로그램과 애플릿은 자바2 플랫폼에서 원격 CORBA 객체 접속이 가능하게 배포할 수 있

다.

노트: Sun사는 자바2 플랫폼에서 “Java IDL” 을 지원한다고 한다. 이 말은 정확하게는 틀린 말이다. IDL은 인터페이스 정의 언어이다, 정의 언어는 인터페이스 클래스에 대한 서술이다. 기술에서 가장 중요한 것은 단지 IDL 지원이 아니라 CORBA와의 연결이다. (주: 하지만 CORBA에서 가장 중요한 것은 구현 언어의 종류에 무관하게 객체의 인터페이스를 정의할 수 있게 해주는 인터페이스 정의 언어이다. ORB는 기본의 RPC 통신에 객체 개념을 포함시킨 것이라고 이해해도 크게 틀리지 않다.)

아래는 CORBA 객체 구현을 위한 단계이다.

1. 객체 작업 방법에 대한 상세의 인터페이스를 IDL을 사용하여 작성한다. 인터페이스 정의 언어는 CORBA 인터페이스에 대한 정의이다. IDL은 언어 중립적인 형태의 고유한 인터페이스 언어이다.
2. IDL 컴파일러를 사용하여 대상언어로 변환한다. 스터브와 헬퍼(helper) 클래스를 생성한다.
3. 서버 객체에 대해 구현 코드를 추가한다. 언어는 여러분이 선택할 수 있다. (IDL 컴파일러에 의해 생성된 스케로턴은 단지 산술 코드일 뿐이다. 여전히 여러분은 서버 메소드에 대하여 실제적인 구현 코드를 제공할 필요가 있다.) 구현 코드를 컴파일 한다.
4. 서버 객체를 생성하고 등록하기 위한 프로그램을 작성한다. 가장 편리한 등록방법은 rmiregistry 와 비슷한 CORBA 명명 서비스(naming service)를 사용하는 것이다.
5. 객체의 위치와 그들의 서비스 호출에 대한 클라이언트 프로그램을 작성한다.
6. 명명 서비스, 서버상의 서버 프로그램 그리고 클라이언트상의 클라이언트 프로그램을 구동시킨다.

이 단계들은 RMI를 사용한 응용 프로그램의 배치와 비슷한 단계이며, 두가지 중요한 차이가 있다.

여러분은 클라이언트와 서버를 구현하기 위한 다양한 언어를 사용할 수 있다.
여러분은 고유한 인터페이스를 위해 IDL을 사용한다.

다음 절에서는 여러분은 어떻게 IDL을 이용하여 CORBA 인터페이스를 구현하는지, 그리고 어떻게 자바로 구현된 클라이언트와 C++로 구현된 서버, C++로 구현된 클라이언트와 자바로 구현된 서버간의 연결을 보게 될 것이다.

그러나 CORBA는 매우 어려운 주제이다. 우리는 단지 여러분에게 기본적인 예제를 보여줄 뿐이다. 좀더 많은 정보를 원한다면 "Client/Server Programming with Java and Corba by Rebert Orfali and Dan Harkey[John Wiley & Sons 1998]" 와 "Michi Henning과 Steve Vinoski가 쓴 Advanced CORBA Programming with C++[Addison-Wesley 1999]"를 추천한다.

인터페이스 정의 언어(Interface Definition Language)

IDL의 구문을 소개하기 위하여, RMI를 이용하여 작성한 것과 같은 예제를 실행해 보자. RMI에서는 자바 언어에 의한 인터페이스로 시작한다. CORBA의 IDL 구문에 따른 인터페이스의 시작점은 아래와 같다.

```
interface Product
{   string getDescription();
};
```

IDL 과 자바 언어에 사이에는 여러가지 미묘한 차이가 있다. IDL에서 인터페이스 정의의 끝은 세미콜론으로 끝난다. 또한 string이 소문자로 작성된다는 것을 알아야 한다. 사실, string 클래스는 문자열에 대한 CORBA에서의 표기법이며 자바 언어의 String과는 다르다. 자바언어에서는 문자열이 16-bit 유니코드이다. CORBA에서는 단지 8-bit 문자이다. 만약 여러분이 16-bit 문자열을 ORB를 통해서 전송한다면 문자열 캐릭터는 nonzero 상위 바이트를 가지므로 예외를 발생한다. 이런 종류의 타입의 부적합한 문제는 언어들 간의 상호운용성을 확보하기 위한 부분에서 대가를 지불해야 한다.

노트: CORBA는 또한 "wide" 문자들에 대한 wchar 형을 가지고 있다. 그러나 이 형은 유니코드 인코딩에 대한 아무런 보장을 하지 못한다.

"IDL to Java" 컴파일러는 인터페이스에 대한 IDL 정의를 자바 언어로 변환해 준다. 예를 들어, Product.idl로 작성된 IDL Product를 idltojava로 수행하면 Product.java 파일이 아래의 내용으로 반환한다.

```
interface Product
{   String getDescription();
}
```

노트: JDK 1.3에서는 idltojava가 idlj 프로그램으로 대체되었다.

또한 IDL 컴파일러는 몇가지 다른 소스 파일들 - ORB와의 통신을 위한 스터브 클래스들과 세 개의 헬퍼 클래스들이 생성된다.

노트: 여러분은 IDL로 모든 프로그래밍을 할 수 는 없다. IDL은 단지 인터페이스에 대한 표현일 뿐이다. IDL로 서술된 CORBA 객체는 C++ 이나 자바 언어로 구현되어 진다.

IDL로 부터 주어진 룰에 의해서 구현되어진 자바 프로그램 언어를 자바 프로그램으로 생성되었다고 부른다. 언어 생성은 OMG에 의해서 표준화가 되어 있다. 모든 CORBA 벤더들은 특정한 프로그램 언어를 구성하기 위해 동일한 규칙을 준수해야 한다.

우리는 모든 측면의 IDL 또는 자바 프로그래밍 언어 바인딩에 대한 것을 다루지 않는다. - Object Management Group(www.omg.org)의 웹사이트에서 CORBA 관련 문서를 참조하라 그러나, 몇 가지 모든 IDL 사용자들이 필요로 하는 중요한 개념들이 있다.

메소드를 정의할 때, 여러분은 전달인자 전송시 자바 프로그래밍 언어에서 제공하는 것보다 많은 선택을 갖을 수 있다. 모든 전달인자들은 in,out 혹은 inout 으로 선언할 수 있다. in 전달인자는 메소드에 간단하게 전달된다. - 이것은 자바 프로그래밍 언어에서 전달인자를 전송하는 방식과 동일하다. 그러나 자바 프로그래밍 언어는 out 전달인자에 대해서는 다르다. 메소드의 out 전달인자는 반환되기 전에 저장소에 값이 저장된다. 호출자는 out 전달인자 안에 저장된 값을 검색할 수 있다.

예를 들어, find 메소드는 product 객체에 저장된 값을 찾을 수 있다.

```
interface Warehouse
{  boolean locate(in String descr, out Product p);
    .....
};
```

만약 전달인자가 단지 out 전달인자만으로 정의 되어 있다면, 그 메소드는 전달인자의 초기화를 기대할 수 없다. 그러나, 만약 inout 전달인자로 정의되어 있다면 호출자는 메소드에 대한 값을 제공해야 만 한다. 그리고 그 메소드는 호출자가 검색하여 값을 변환하기 위하여 값을 변환시킬 수 있다. 자바 프로그래밍 언어에서 이런 전달인자들은 자바 IDL 컴파일러에 의해서 생성된 특별한 holder 클래스와 같다.

IDL 컴파일러는 모든 인터페이스에 대하여 추가적인 Holder를 갖는 클래스를 생성한다. 예를 들면, Product 인터페이스를 컴파일할 때 자동적으로 ProductHolder 클래스를 생성한다. 모든 Holder 클래스는 value라 불리는 공용 인스턴스를 생성한다.

메소드가 out 전달인자를 가지고 있을 때 IDL 컴파일러는 holder를 사용하여 메소드의 서명을 변경할 수 있다. 예를 들어

```
interface Warehouse
{    boolean locate(String descr, ProductHolder p);
     . . .
};
```

메소드가 호출될 때 여러분은 holder 객체 전달을 필요로 한다. 메소드가 반환된 후에, 여러분은 holder 객체로 부터 out 전달인자에 대한 값을 검색할 수 있다. 다음은 여러분이 어떻게 locate 메소드를 호출하는 가를 보여준다.

```
Warehouse w = ... ;
String descr = ...;
Product p;
ProductHolder  pHolder = new ProductHolder();
if( w.locate(descr, pHolder))
  p = pHolder.value;
```

기본적인 타입에 대한(IntHolder, DoubleHolder와 같이) holder 클래스를 미리 정의 한다.

노트: IDL은 오버로드된 메소드를 지원하지 않는다. 여러분은 각 오버로드가 필요한 메소드에 대해 각기 다른 이름을 제공해야 한다.

IDL에서, 여러분은 배열의 변수 크기를 정의 하기 위해서 sequence 생성자를 사용한다. 여러분은 반드시 순차적 전달인자나 반환된 값들을 정의하기 위해서 먼저 타입을 정의해야 한다. 예를 들어, 다음은 "sequence of products" 타입의 정의이다.

```
typedef sequence<Product> ProductSeq;
```

그러면 여러분은 이 타입을 메소드에서 선언할 수 있다.

```
interface Warehouse
{  ProductSeq find(in Customer c);
    ...
};
```

자바 프로그래밍 언어에서는 sequence들은 배열과 같다. 예를 들어 find 메소드는 아래와 같이 매핑된다.

```
Product[] find(Customer c)
```

만약 메소드가 예외를 발생하면, 여러분은 먼저 예외 타입을 정의한 다음 raises 선언을 사용한다. 아래의 예제와 같이 find 메소드는 BadCustomer 예외를 발생시킬 수 있다.

```
interface Warehouse
{    exception BadCustomer  {   string reason; };
     ProductSeq find(in Customer c) raises BadCustomer;
      ...
};
```

IDL 컴파일러는 예외 타입을 클래스로 변환한다.

```
final public class BadCustomer extends org.omg.CORBA.UserException
{    public BadCustomer()  {}
    public BadCustomer(String __reason) { reason = __reason; }
   public String reason;
}
```

만약 여러분이 이런 예외를 감지하면, 여러분은 이것의 공용 인스턴스 변수들을 조사할 수 있다. raises 구분자는 자바 메소드의 throws 구분자로 변환된다.

```
ProductSeq find(Customer c) throws BadCustomer
```

인터페이스는 상수를 포함할 수 있다. 예를 들어

```
interface Warehouse
{   const int SOLD_OUT = 404;
    . . .
};
```

또한 인터페이스는 속성들을 포함할 수 있다. 속성은 인스턴스 변수와 같다. 그러나 이것들은 실제적으로 접근자와 변경자 메소드의 쌍으로 되어있다. 예를 들어 다음은 Book 인터페이스에 대한 isbn 속성이다.

```
interface Book
{   attribute string isbn;
    ...
};
```

같은 자바 프로그래밍 언어에서 메소드의 쌍과 같다, 두개의 isbn 이름을 갖는다.

```
String isbn()  // 접근자
void isbn(String __isbn)  // 변경자
```

만약 속성이 읽기만 가능하게 선언되었다면, 변경자는 생성되지 않는다.
여러분은 CORBA 인터페이스에 특별한 변수를 지정할 수 없다. - 객체에 대한 데이터의 표현은 구현 전략의 부분이고, IDL은 모든 구현을 지정하지 않는다.
CORBA는 인터페이스 상속을 지원한다. 예를 들면,

```
interface Book : Product { /* ... ... */ };
```

여러분은 콜론(:)으로 상속을 표시한다. 인터페이스는 다른 인터페이스를 상속할 수 있다.
IDL에서는 인터페이스의 정의, 타입,상수 그리고 예외를 module로 그룹화 할 수 있다.

```
module corejava
{   interface Product
    { ...
     };

    interface Warehouse
    { ...
     };
};
```

모듈은 자바 프로그래밍 언어의 패키지로 변환된다.

여러분이 IDL 파일을 가지고 있으면, 여러분이 여러분의 ORB 벤더가 제공하는 IDL 컴파일러를 수행하면, 여러분의 대상 언어(자바 또는 C++언어 같은)에 대한 스터브들과 helper 클래스들을 얻는다.

예를 들면, IDL 파일을 자바 언어로 변환하기 위해서 idltojava 프로그램을 실행한다. 명령줄에 IDL 파일 이름을 적어야 한다.

```
idltojava Product.idl
```

프로그램은 다섯 개의 파일을 생성한다.

Product.java : 인터페이스 정의이다.

ProductHolder.java : out 전달인자에 대한 Holder 클래스이다.

ProductHelper.java : 다음 절에서 사용될 Helper 클래스이다.

_ProductImplBase.java : 구현 클래스의 상위 클래스이다.

_ProductStub.java : ORB와의 통신을 위한 스터브 클래스이다.

같은 IDL 파일을 C++로 컴파일하였다. 우리는 자유롭게 사용할 수 있는 공개 CORBA제품인 omniORB를 사용하여 예제를 작성하였다. omniORB 패키지는 IDL을 C++로 컴파일해주는 omniidl2를 포함하고 있다. 컴파일러를 실행하면 두개의 C++ 파일을 생성한다.

Product.hh : Product,ProductHelper 클래스와 _sk_Product(서버 구현 클래스에 대한 기본 클래스) 클래스의 헤더 파일이다.

ProductSK.cpp : 이들 클래스에 대한 C++ 소스 코드이다.

노트: 언어 바인딩이 표준화되는 동안 각 벤더들은 이를 어떻게 생성하고 현실적인 바인딩을 위한 코드 패키지가 무엇인가에 대해 결정해야 한다. 다른 벤더의 IDL-to-C++ 컴파일러는 다른 파일 구성으로 파일을 생성한다.

CORBA 예제

우리의 첫번째 예제는 CORBA를 지원하는 자바 2 플랫폼에서 자바 언어로 작성된 클라이언트가 C++로 작성된 서버 객체를 어떻게 호출하는 지를 보여준다. 서버 측면에서는 우리는 omniORB를 사용하였다.

omniORB는 http://www.uk.research.att.com/omniORB/index.html 에서 다운로드 받을 수 있다

노트: omniORB 제품은 공개이나 윈도우 상에서는 Microsoft C++ 컴파일러가 필요하다. 리눅스 상에서는 GNU의 C++ 컴파일러를 사용할 수 있다.

노트: 물론 원칙적으로 여러분은 서버상에서 임의의 CORBA 2의 완전한 ORB를 사용할 수

있다. 그러나 여러분이 만약 다른 ORB를 사용한다면 C++ 코드는 변경할 필요가 있다. 만약 우리의 경험을 필요로 한다면 이 절의 마지막인 bootstrapping 이슈를 살펴 보라. 우리는 다른 ORB에 대한 정보를 제공한다.

우리의 예제 C++ 서버 객체는 서버의 환경 변수를 간단하게 보고한다. 인터페이스는 다음과 같다.

```
interface Env
{    string getenv(in string name);
};
```

예를 들어, 아래의 자바 프로그래밍 언어는 서버 객체의 수행에 의한 PATH 환경 정보를 처리하는 것이다.

```
Env env = ...;
String value = env.getenv("PATH")
```

C++ 로 구현된 코드는 아래에 있으며 우리는 단순히 C 라이브러리에 있는 getenv 메소드를 호출하였다.

```
class EnvImpl : public virtual _sk_Env
{
public:
  virtual char* getenv(const char *name)
  {   char* value = ::getenv(name);
      return CORBA::string_dup(value);
   }
};
```

여러분은 이 절에서는 C++ 코드에 대해서 이해할 필요가 없다. - 단지 CORBA객체 안에 캡슐화된 것을 자바 프로그래밍 언어로 작성된 프로그램으로 부터 호출이 가능하다는 큰 줄기만 알고 있으면 된다.

서버 측면에서, 여러분은 지금 아래와 같이 C++ 프로그램을 작성할 필요가 있다.

1. ORB를 시작하고;
2. EnvImpl 클래스의 객체를 생성하고 ORB에 등록한다.
3. 객체의 이름에 대한 명명 서버에 바인드 시킨다.
4. 클라이언트로 부터의 호출을 기다린다.

여러분은 이 절의 마지막 부분에 있는 예제 5-17을 찾을 수 있다. 우리는 C++ 코드에 대하여 상세히 다루지 않는다. 만약 여러분이 관심이 있다면 omniORB문서를 참고하라. 그 문서는 매우 좋은 자습서들을 단계별로 상세히 설명하고 있다.

이제 클라이언트 코드로 가보자. 여러분은 이미 원격 객체에 대한 참조로 서버 객체에 대한 호출 방법을 보았다. 그러나 참조를 얻기 위하여, 여러분은 RMI와는 다른 모호함을 보게 될 것이다.

먼저 여러분은 ORB를 초기화 한다. ORB는 다른 ORB간의 대화와 전달인자의 마샬링과 언마샬링의 방법을 알고 있는 단순한 코드 라이브러리이다.

```
ORB orb = ORB.init(args,null);
```

다음으로 여러분의 다른 객체들에 대한 위치를 찾기 위한 명명 서비스를 위치 시켜야 한다.
그러나 CORBA에서 명명 서비스는 단지 다른 CORBA 객체 일 뿐이다. 명명 서비스를 호출하기 위해 여러분은 이것을 위치시켜야 한다. CORBA 1에서는 이것에 대한 참조 방법이 표준화 되어 있지 않았기 때문에 아주 큰 문제였다. 그러나 CORBA 2에서는 이름에 의한 표준화된 서비스로 위치할 수 있게 해준다.

```
String[] services = orb.list_initial_services();
```

ORB는 표준화된 서비스의 이름 리스트로 이들을 연결할 수 있다. 명명 서비스는 표준화된 NameService라는 이름을 가지고 있다. 몇몇 ORB들은 추가적으로 초기화 서비스를 가지고 있다. 자바 2 플랫폼에서는 단지 이름 리스트에 대한 초기화만 가능하다.

객체 참조를 포함하는 서비스는 resolve_initial_references 메소드를 사용한다. 이것은 일반적인 org.omg.corba.Object 클래스의 인스턴스인 CORBA 객체를 반환한다.
여러분은 패키지 앞부분에 이를 사용할 필요가 있다. 만약 단지 Object만 사용한다면 컴파일러는 java.lang.Object 로 가정한다.

```
org.omg.CORBA.Object object
  = orb.resolve_initial_references("NameService");
```

다음으로, 여러분이 Naming Context 인터페이스의 메소드를 호출할 수 있도록 Naming Context 참조에 대한 참조를 변환해야 한다. RMI에서는 여러분은 다른 타입에 대한 참조를 간단하게 변환할 수 있다. 그러나 CORBA에서는 간단하게 참조를 변환할 수 없다.

```
Naming Context naming Context
  = (Naming Context)object; // 에러
```

대신에 여러분은 대상 인터페이스의 헬퍼 클래스의 narrow 메소드를 사용해야 한다.

```
Naming Context nameContext
  = Naming ContextHelper.narrow(object);
```

주의 : CORBA 객체 참조를 하위 타입으로 형변환하는 것은 간혹 성공할 수 있다. 많은 org.omg.CORBA.Object 참조는 이미 적합한 인터페이스를 구현한 객체에 대한 포인터이다. 그러나 객체 참조는 실제로 인터페이스를 구현한 다른 객체에 대한 참조를 유지할 수 있다. 여러분은 어떻게 스터브 객체들이 생성되는지 알 수 있는 방법이 없기 때문에 항상 CORBA 객체 참조를 다른 하위 타입으로 변환하기 위해 narrow 메소드를 사용해야 한다.

이제 여러분은 naming context를 가지게 되었다. 서버에 위치한 naming context에 대한 객체를 위치시키고 사용할 수 있다. naming context는 서버 객체들의 이름으로 구성된다.
이름들은 일련의 이름 컴포넌드로 중첩된다. 구조적 이름 구성에 의한 중첩 레벨을 사용할 수 있다 - 이것은 파일 시스템 상이 디렉토리와 비슷하게 사용된다.

이름 컴포넌트는 ID와 종류로 구성된다. 컴포넌트에 대한 ID는 같은 부모 컴포넌트에 대한 모든 이름에 대하여 유일하다. 종류는 컴포넌트의 여러가지 타입을 말한다. 이런 종류는 표준화 되어 있지 않다. 우리는 중첩된 이름을 가지는, 그리고 객체 이름들에 대한 "Object"들의 컴포넌트 이름에 대한 "Context"를 사용한다.

우리의 예제에서 서버 프로그램은 순차적인 이름의 표현에 EnvImpl 객체를 대체한다.

```
(id = "corejava", kind = "Context"), (id=Env", kind = "Object")
```

우리는 컴포넌트 이름으로 구성된 배열과 Naming Context 인터페이스의 resolve 메소드의 전달에 의해 원격 참조를 검색할 수 이다.

```
NameComponent[] path =
{   new NameComponent("corejava","Context"),
    new NameComponent("Env", "Object")
};
org.omg.CORBA.Object envObj = nameimgContext.resolve(path);
```

다시 한번 객체 참조에 대한 결과에 대해 반드시 narrow 메소드를 사용한다.

```
Env env = EnvHelper.narrow(envObj);
```

이제 우리는 원격 메소드를 호출할 준비가 되었다.

```
String value = env.getenv("PATH");
```

예제 5-16은 완전한 예제이다.

아래는 클라이언트 프로그램의 표준화된 과정의 예를 보여준다.

1. ORB를 구동한다.
2. 초기화된 Naming Service와 좁혀진(narrowing된) Naming Context 참조에 대한 검색 결과에 의한 명명 서비스를 위치시킨다.
3. 이름들의 조합에 의한 호출과 Naming Context의 resolve 메소드의 호출을 원하는 메소드를 가진 객체를 위치시킨다.
4. 정확한 타입과 여러분의 메소드의 호출에 대한 결과를 좁힌다.

실제적인 테스트 프로그램은 아래와 같다.

1. IDL 파일을 C++과 자바 IDL 컴파일러로 컴파일한다.
2. C++ 서버 프로그램을 컴파일 한다.(make 파일은 CD-ROM에 제공되어 있다)
3. 클라이언트 프로그램을 컴파일 한다.
4. 서버의 명명 서비스를 구동시킨다.(예를 들면 여러분이 omniORB를 사용한다면 omniNames이다) 명명 서비스는 여러분이 중지시키기 전 까지 구동된다.
5. 서버를 구동시킨다(EnvServer). 서버 또한 중지 전까지 구동된다.
6. 클라이언트를 실행한다(java EnvClient). 서버 프로세스의 PATH 정보를 출력한다.

만약 서버가 원격지에 있거나 서버 ORB의 초기 포트가 자바 IDL의 기본 포트인 900과 동일하지 않다면 여러분은 ORBInitialHost와 ORBInitialPort 속성을 설정해야 한다.
여러분은 시스템 프로퍼티를 설정할 수 있다.

```
org.omg.CORBA.ORBInitialHost
org.omg.CORBA.ORBInitialPort
```

예를 들면 -D 옵션으로 바이트코드 변환을 사용하거나 명령어 라인상에서 설정할 수 가 있다.

```
java EnvClient -ORBInitialHost warthog  -ORBInitialPort 1050
```

명령어 라인에 의한 전달인자는 호출에 의해 ORB에게 전달된다.

```
ORB orb =ORB.Init(args,null);
```

원칙적으로, 여러분의 ORB 벤더들은 여러분들이 확실하게 독립적인 처리 작업을 할 수 있도록 해주어야 한다.

만약 여러분의 클라이언트가 명명 서비스를 찾지 못한다면, 서버에 대한 클라이언트의 포트 번호를 확인해 보라

팁: 만약 여러분이 명명 서비스 연결에 문제가 있다면 여러분의 ORB에 위치한 초기화 된 서비스의 목록을 출력해 보라.

```
public class ListServices
{   public static void main(String args[]) throws Exception
    { ORB orb = ORB.init(args,null);
      String[] services = orb.list_initial_services();
     for(int i = 0; i < services.length; i++)
           System.out.println(services[i]);
     }
}
```

몇몇 ORB의 NameServices는 리스팅 서비스를 지원하지 않지만 설정을 변경하는 데는 큰 문제가 없다. 여러분은 적절한 객체 참조에 대한 서버 객체의 위치를 변경할 수 있거나, 혹은 IOR을 변경할 수 있다. IOR은 아래에 좀더 자세한 정보가 잇다.

IOR을 통해 객체 위치시키기

만약 여러분이 클라이언트가 호출하기 위한 서버 ORB와 명명 서비스를 설정할 수 없다면 여러분은 공통의 객체 참조를 이용하여 CORBA 객체를 위치시킬 수 있거나, IOR을 사용할 수 있다. IOR은 "IOR:"로 시작하는 긴 문자열이며 16진수로 표시된다.

```
IOR:012020201000000000490000000000002343420342823073a23300002
0000004e00000002020f342000003e0000f000e8989a989898398d98999899
000000000000020340200000000000023420234398e0000000000000dfd00
e7828fs98d7887e89e00000000000000000000000d0000000010102203030
```

IOR은 객체에 대한 유일한 표현이다(주:실세계에서 개개인을 유일하게 구별해주는 주민번호와 같다). 많은 서버 클래스는 등록된 모든 객체에 대한 IOR을 표현하고 있다. 클라이언트는 그들의 위치를 지정할 수 있다. 여러분은 서버의 IOR을 클라이언트 프로그램에 다음과 같이 붙여 넣을 수 있다.

```
String ref = "IOR:012003000000000000000342.....";
// 서버로부터 IOR를 붙인다.
org.omg.CORBA.Object object = orb.string_to_object(ref);
```

그러면 반환된 객체를 적절한 타입으로 변환할 수 있다. 예를 들면,

```
Env env = EnvHelper.narrow(object);
```

이 책의 코드들 테스트 할 때, 우리는 Visibroker로 클라이언트를 연결할 때 이 메소드를 이용하여 성공하였다.

이 절에서 여러분들은 C++로 구현된 서버에 어떻게 연결하는 지를 보았다. 여러분은 CORBA 객체와 자바 프로그래밍 언어로 작성된 임의의 프로그램에 대한 접근을 지원하기 위한 서비스를 제공할 수 있으며 클라이언트 상의 추가적인 배포도 가능하다.
다음 절에서는 반대되는 내용을 보게 될 것이다. 여기에서 서버는 자바로 구현되었으며 클라이언트는 C++로 구현되었다.

org.omg.CORBA.ORB

static ORB init(String[] args, Properties props)
새로운 ORB를 생성하고 이것을 초기화 시킨다.
전달인자: args ORB에 대한 명령라인의 인자들
props ORB 설정에 대한 프로퍼티의 테이블

String[] list_initial_services()
NameService에 대한 사용가능한 초기화된 목록을 반환한다.

org.omg.CORBA.Object resolve_initial_references(String name)
초기화된 서비스중에서 전송된 객체를 반환한다.
전달인자 : name 초기화된 서비스의 이름

org.omg.CORBA.Object string_to_object(String ior)
주어진 IOR에 대한 객체의 위치를 지정한다(동일한 객체 참조).

예제 5-16 : EnvClient.java

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class EnvClient
{  public static void main(String args[])
   {  try
      {  ORB orb = ORB.init(args, null);
         org.omg.CORBA.Object namingContextObj
            = orb.resolve_initial_references("NameService");
         NamingContext namingContext
            = NamingContextHelper.narrow(namingContextObj);

         NameComponent[] path = { new NameComponent("corejava", "Context"),
            new NameComponent("Env", "Object") };
         org.omg.CORBA.Object envObj = namingContext.resolve(path);
         Env env = EnvHelper.narrow(envObj);
         System.out.println(env.getenv("PATH"));
      }
      catch(Exception e)
      {   System.out.println("Error: " + e);
          e.printStackTrace(System.out);
      }
   }
}
```

예제 5-17 : EnvServer.cpp

```
#include <iostream.h>
#include "omnithread.h"

#include "Env.hh"

class EnvImpl : public virtual _sk_Env
```

```
{
public:
  virtual ~EnvImpl() {}
  virtual char* getenv(const char *name);
};

char* EnvImpl::getenv(const char *name)
{  char* value = ::getenv(name);
   return CORBA::string_dup(value);
}

void bindObjectToName(CORBA::ORB_ptr orb,
   const char name[], CORBA::Object_ptr obj)
{  CosNaming::NamingContext_var rootContext;

   try
   {  // Obtain a reference to the root context of the Name service:
      CORBA::Object_var initServ
         = orb->resolve_initial_references("NameService");

      rootContext = CosNaming::NamingContext::_narrow(initServ);

      if (CORBA::is_nil(rootContext))
      {  cerr << "Failed to narrow naming context." << endl;
         return;
      }
   }
   catch(CORBA::ORB::InvalidName& ex)
   {  cerr << "Service does not exist." << endl;
      return;
   }

   try
   {  // Bind a context called "corejava" to the root context:

      CosNaming::Name contextName;
```

```
      contextName.length(1);
      contextName[0].id   = "corejava";
      contextName[0].kind = "Context";

      CosNaming::NamingContext_var corejavaContext;
      try
      {  // Bind the context to root, and assign testContext to it:
         corejavaContext = rootContext->bind_new_context(contextName);
      }
      catch(CosNaming::NamingContext::AlreadyBound&)
      {  // If the context already exists, this exception will be raised.
         // In this case, just resolve the name and assign testContext
         // to the object returned:
         CORBA::Object_var contextObj
            = rootContext->resolve(contextName);
         corejavaContext = CosNaming::NamingContext::_narrow(contextObj);
         if (CORBA::is_nil(corejavaContext))
         {  cerr << "Failed to narrow corejava context." << endl;
            return;
         }
      }

      // Bind obj to name in the corejava context
      CosNaming::Name objectName;
      objectName.length(1);
      objectName[0].id   = name;
      objectName[0].kind = "Object";

      corejavaContext->rebind(objectName, obj);
   }
   catch (CORBA::COMM_FAILURE&)
   {  cerr << "COMM_FAILURE" << endl;
   }
}

int main(int argc, char *argv[])
```

```
{  cout << "Creating and initializing the ORB..." << endl;

   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv, "omniORB2");
   CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

   cout << "Registering server implementation with the ORB..." << endl;

   EnvImpl* impl = new EnvImpl();
   impl->_obj_is_ready(boa);

   CORBA::String_var s = orb->object_to_string(impl);
   cout << s << endl;

   cout << "Binding server implementation to name service..." << endl;

   Env_var env = impl->_this();
   bindObjectToName(orb, "Env", env);

   cout << "Waiting for invocations from clients..." << endl;

   boa->impl_is_ready();

   return 0;
}
```

org.omg.ConNaming.NamingContext

org.omg.CORBA.Object resolve(NameComponent[] name)

주어진 이름에 대한 바운드된 객체를 반환한다.

org.omg.CosNaming.NameComponent

NameComponent(String id, String kind)

새로운 컴포넌트를 생성한다.

전달인자 : id **생성될 컴포넌트의 지시자를 표현한 문자열**

kind **생성될 컴포넌트의 타입을 표현한 문자열**

CORBA 서버의 구현

만약 CORBA 기반 구조를 배포하려면 CORBA 서버 객체에 대해서 자바 프로그래밍 언어가 가장 좋다는 것을 알게 될 것이다. 언어 바인딩이 중립적이며, C++ 보다 서버 소프트웨어가 안전하고 쉽게 생성될 수 있다. 이 절에서는 자바 프로그래밍 언어로 CORBA 서버를 구현하는 방법을 기술한다.
이 절의 예제는 앞 절의 예제와 비슷하다. 우리는 자바 가상 머쉰에 의한 시스템 속성을 찾는 기능을 제공한다. 다음은 이 기능을 제공하기 위한 IDL 이다.:

```
interface SysProp
{   string getProperty(in string name);
}
```

예를 들어 우리의 클라이언트 테스트 프로그램은 다음의 서버를 호출한다.

```
CORBA::String_var key = "java.vendor";
CORBA::String_var value = sysProp->getProperty(key);
```

반환 값은 서버 머쉰에서 실행되는 자바 가상 머신의 벤더들을 설명하는 문자열이다. 우리는 C++ 로 작성된 클라이언트 프로그램에 대해서는 상세히 다루지 않는다. 예제는 5-19의 코드들이다.

서버 구현을 위해서, 여러분은 IDL 파일로 부터 idltojava 컴파일러로 생성된 _SysPropImplBase을 확장해서 사용한다. 아래와 같이 구현된다.

```
class SysPropImpl extedns _SysPropImplBase
{  public String getProperty(String key)
    {   return System.getProperty(key);
    }
}
```

노트: 여러분은 구현 클래스에 대하여 여러분이 원하는 이름을 붙일 수 있다. 이 책에서 우리는 RMI 관례처럼 Impl 접미사를 사용한다. 다른 프로그램에서는 Servant 혹은 _i를 사용하는 경우도 있다.

노트 : 만약 여러분이 다른 클래스를 먼저 확장하여 구현 하였다면 여러분은 기본 클래스 구현을 확장할 수 없다. 이런 경우에는 idltojava 컴파일러로 오퍼레이션 인터페이스와 tie 클래스를 생성하여 지시할 수 있다. 좀더 자세한 내용은

http://java.sun.com/products/jdk/1.2/doc/guide/idl/.

다음으로, 여러분은 아래의 작업으로 서버 프로그램을 작성해야 한다.

1. ORB를 구동시킨다.
2. 구현 서버를 생성하고 ORB에 등록한다.
3. IOR를 출력한다.(변경된 클라이언트에 대한 명명 서비스)
4. 명명 서비스에 대한 구현 서버를 바인딩 시킨다.
5. 클라이언트의 호출을 기다린다.

완벽한 소스는 예제 5-18이다. 여기에는 중요한 내용만 기술하였다. 클라이언트 프로그램을 위해 ORB를 구동시킨다.

```
ORB orb = ORB.init(args,null);
```

connection 메소드를 사용하여 객체를 ORB에 등록한다.

```
SysPropImpl impl = new SysPropImpl();
orb.connect(impl);
```

다음은 object_to_string 메소드로 IOR을 가져와서 출력한다.

```
System.out.println(orb.object_to_string(impl));
```

여러분은 같은 방법으로 클라이언트 프로그램에서 정확한 명명 서비스에 대한 참조를 얻을 수 있다.

```
org.omg.CORBA.Object namimgContextObj =
   orb.resolve_initial_references("NameService");
NamingContext namingContext =
   NamingContextHelper.narrow(namingContextObj);
```

그런 다음 객체에 대한 이름을 서술하여 생성한다. 다음은 SysProp 객체 호출을 보여준다.:

```
NameComponent[] paht =
{   new NameComponent("SysProp", "Object")
}
```

경고: 우리는 여기서 중첩된 이름의 사용을 하지 않는다. 현재 JDK에서 tnameserv 명명 서비스는 중첩 이름을 지원하지 않는다.

여러분은 이름에 대한 객체 바인드를 위해 rebind 메소드를 사용한다.

```
namingContext.rebind(path, impl);
```

마지막으로 클라이언트의 호출이 있을 때 까지 대기하고 있는다. 병목을 피하기 위해 결코 notify 하지 않는 객체의 wait를 호출한다.

```
java.lang.Object sync = new java.lang.Object();
synchronized(sync)
{   sync.wait();
}
```

이 프로그램의 테스트는 아래와 같다.

1. IDL 파일을 C++과 자바 IDL 컴파일러로 컴파일한다.
2. 서버 프로그램을 컴파일 한다.
3. C++ 클라이언트 프로그램을 컴파일한다.(make 파일은 CD-ROM에서 제공한다.)
4. 서버 상의 tnameserv 서비스를 시작한다. 이 프로그램은 JDK에 포함되어 있다. 명명 서비스는 여러분이 종료시키기 전까지 종료되지 않는다. 만약 필요하다면 여러분은 ORBInitPort 전달인자를 사용하여 기본 포트인 900을 변경할 수도 있다.
5. 서버를 구동시킨다.(java SysPropServer). 서버는 여러분이 종료시키지 않는 한 종료하지 않는다.
6. 클라이언트를 구동시킨다(java EnvClient). 서버의 자바 가상 머쉰의 벤더를 출력한다.

여러분은 다른 프로그래밍 언어로 작성된 클라이언트와 서버의 연결을 CORBA를 이용한 방법을 보았다. 이것으로 우리의 의견을 결론짓는다.
CORBA 표준에는 동적 메소드 호출과 트랜잭션 핸들링과 지속성에 관련된 표준화된 서비스 등 아주 흥미있는 내용들이 많이 있다. 우리는 Client/Server Programming with java and CORBA by Robert Offali and Dan Harkey[John Wiley & Sons 1998]을 추천한다.

예제 5-18 : SysPropS erver.java

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
```

```
class SysPropImpl extends _SysPropImplBase
{  public String getProperty(String key)
   {  return System.getProperty(key);
   }
}

public class SysPropServer
{  public static void main(String args[])
   {  try
      {  System.out.println("Creating and initializing the ORB...");

         ORB orb = ORB.init(args, null);

         System.out.println
            ("Registering server implementation with the ORB...");

         SysPropImpl impl = new SysPropImpl();
         orb.connect(impl);
         System.out.println(orb.object_to_string(impl));

         org.omg.CORBA.Object namingContextObj =
             orb.resolve_initial_references("NameService");
         NamingContext namingContext
            = NamingContextHelper.narrow(namingContextObj);

         NameComponent[] path =
            {  new NameComponent("SysProp", "Object")
            };

         System.out.println
            ("Binding server implementation to name service...");

         namingContext.rebind(path, impl);

         System.out.println
```

```
            ("Waiting for invo경고s from clients...");

         java.lang.Object sync = new java.lang.Object();
         synchronized (sync)
         {  sync.wait();
         }
      }
      catch (Exception e)
      {   System.err.println("Error: " + e);
          e.printStackTrace(System.out);
      }
   }
}
```

예제 5-19 : SysPropClient.cpp

```
#include <iostream.h>
#include "SysProp.hh"

CORBA::Object_ptr getObjectReference(CORBA::ORB_ptr orb,
   const char serviceName[])
{  CosNaming::NamingContext_var rootContext;

   try
   {  // Obtain a reference to the root context of the Name service:
      CORBA::Object_var initServ;
      initServ = orb->resolve_initial_references("NameService");

      // Narrow the object returned by resolve_initial_references()
      // to a CosNaming::NamingContext object:
      rootContext = CosNaming::NamingContext::_narrow(initServ);
      if (CORBA::is_nil(rootContext))
      {  cerr << "Failed to narrow naming context." << endl;
         return CORBA::Object::_nil();
      }
```

```
   }
   catch(CORBA::ORB::InvalidName&)
   {  cerr << "Name service does not exist." << endl;
      return CORBA::Object::_nil();
   }

   // Create a name object, containing the name corejava/SysProp:
   CosNaming::Name name;
   name.length(1);

   name[0].id   = serviceName;
   name[0].kind = "Object";

   CORBA::Object_ptr obj;
   try
   {  // Resolve the name to an object reference, and assign the reference
      // returned to a CORBA::Object:
      obj = rootContext->resolve(name);
   }
   catch(CosNaming::NamingContext::NotFound&)
   {  // This exception is thrown if any of the components of the
      // path [contexts or the object] aren't found:
      cerr << "Context not found." << endl;
      return CORBA::Object::_nil();
   }
   return obj;
}

void callServer(CORBA::Object_ptr obj)
{  SysProp_var sysProp = SysProp::_narrow(obj);

  if (CORBA::is_nil(sysProp))
  {  cerr << "hello: cannot invoke on a nil object reference.\n" << endl;
     return;
  }
```

```
  CORBA::String_var key = "java.vendor";
  CORBA::String_var value = sysProp->getProperty(key);

  cerr << key << "=" << value << endl;
}

int main (int argc, char *argv[])
{  CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv, "omniORB2");
   CORBA::BOA_ptr boa = orb->BOA_init(argc, argv, "omniORB2_BOA");

   try
   {  CORBA::Object_var obj = getObjectReference(orb, "SysProp");
      callServer(obj);
   }
   catch (CORBA::COMM_FAILURE&)
   {  cerr << "COMM_FAILURE" << endl;
   }

   return 0;
}
```

org.omg.CORBA.ORB

void connect(org.omg.CORBA.Object obj)

해당 ORB에 구현된 객체를 연결한다. 사용가능한 ORB는 객체 메소드로 호출이 가능하다.

String object_to_string(org.omg.CORBA.Object obj)

주어진 객체의 IOR(공용 객체 참조)을 문자열로 반환한다.

org.omg.CosNamingContext

void bind(NameComponent[] name, org.omg.CORBA.Object obj)

void rebind(NameComponent[] name, org.omg.CORBA.Object obj)

객체의 이름을 바인드 한다. bind **메소드는 만약 먼저 바인드 되어 있다면** AlreadyBound **예외를 투구한다.** rebind **메소드는 먼저 바인딩된 객체를 다시 바인드 한다.**

전달인자: name **바운드될 객체의 이름**

obj **명명 컨텍스드 안에 저장된 객체**