특집 1 | 프로그래밍, 무엇을 어떻게 배워야 하나

프로그래밍 언어의 숨겨진 진실과 거짓

이 글에서는 주로 자바를 예로 들어 프로그래밍 언어와 프로그래밍에 대한 진실을 말하고자 한다. 처음에는 자바에 관해 알려진 헛소문을 바로 잡는 것부터 시작하겠다. 그렇다고 자바를 헐뜯는다고 생각하지 않았으면 좋겠다. 필자도 자바가 꽤 쓸만한 도구라는 것을 잘 알고 있다. 또한 자바가 산업과 기술에 미친 영향이 크다는 걸 모르는 것도 아니다. 다만 더도 말고 덜도 말고, 자바(C#을 포함해서)를 '그럭저럭 쓸만한 언어' 정도로만 봐줬으면 한다. 잘못된 환상에 젖어 눈이 멀면 우리는 항상 되풀이되는 질문, 예를 들어 "프로그래밍을 어떻게 배워야 하며, 프로그램을 어떻게 만들어야 잘 만드는 것이고, 새로운 기술을 어디서부터 배워나가야 하는가?"에 올바르게 답할 수 없다.

김재우 kizoo@bluette.com
블루엣에서 개발 환경과 온라인 교육 시스템을 결합한 소프트웨어를 설계하고 있다. 소프트웨어 공학 기술이나 관련 이론을 실천하도록 만드는 것이 개발자로서의 목표. 현재 정보기술원과 함께 분야별 표준 교육 과정, 전문 개발자 양성 및 인증을 위한 교육 시스템 'Theory Into Practice'를 설계하고 있으며, 인도 Vinayaka 대학 IT Parks 소프트웨어 개발팀과 함께 기업형 솔루션 교육과정 및 소프트웨어 개발 기술을 연구하고 있다.

이 글을 쓰기 얼마 전에 서울에 계신 은사님을 뵈러 갔다(참고로 필자는 부산에 살고 있다). 언제나 그렇지만, 인사를 나누는 대신에 만나자마자 기술에 대한 얘기를 건네기에 바쁘다. 자바에 Generic Type을 더하자는 제안이 Public Review 단계에 와 있다는 얘기부터 시작해서, 한참 동안 프로그래밍과 소프트웨어에 관한 이런 저런 얘기를 나눴다. 가기 전에 이미 이 글을 어떻게 써내려 가야할지 틀을 잡아 놓은 상태였지만, 내가 옳어 대는 말이 '오해'나 사지 않을까 걱정도 적잖이 해왔던 터라, 글을 시작하지 못하고 우물쭈물하고 있었던 게 사실이다. 그래서인지 은사와의 대화는 큰 도움이 됐다. 이 글은 그 대화로부터 다시 틀을 잡은 것이다. 그러나 내용은 처음에 생각했던 것과 크게 다르지 않다.

은사와 필자가 별 다른 얘기를 나눈 것은 아니다. 그동안 꾸준히 이 분야에 있던 사람이라면 다 들어봤음직한 얘기다. GJ(Generic Java) 얘기만 해도 전혀 새로운 개념이라 할 수 없다. 자바를 이렇게 저렇게 부르자는 연구는 자바가 나올 때부터 꾸준히 얘기됐던 것이다. 그렇게 치자면 이제와 그걸 받아들여 볼까 하고 눈치보고 있는 셈이니, 정말 속된 말로 '느려 터졌다'고 한 번 비꼬아주는 것이 더 옳겠다.

이와 같이 눈에 띄지 않는 세상은 생각보다 빨리 움직이고 있다. C#(닷넷)을 보면 벌써 같은 연구를 하고 있다. 변변한 책 한 권도 없는데 무슨 소리냐 하는 사람은 http://research.microsoft.com/~dsyme/publications.html을 한번 들러 보기 바란다. 이렇듯 2~3년 후를 내다보기는 쉽지 않아도, 길게 5~10년 후를 내다보기는 어려운 일만은 아니다. 더구나 이 분야는 모르는 사람들이 말하는 것처럼 그렇게 정신없이 바뀌지 않는다. 빨리 바뀌는 것은 작고 덜 중요한 것들이고, 실제 그 알맹이는 수십 년 동안 별로 달라지지 않았다.

자바와 C#에 관한 헛소문

먼저 자바나 C#에 대한 헛소문 몇 가지를 가져와 바로 잡아보자. 물론 이제는 많은 사람이 그 진실을 알 만큼은 안다. 그래서 이 글이 '뒷북치기'나 되지 않을까 좀 걱정도 된다. 하지만 아직 그 헛소문이 꼬리를 내리지 않은 것이 사실이고, 그 꼬리를 밟아 피해를 보는 사람도 여전히 많다. 게다가 설령 진실을 알게 된다 해도 그런 사실이 진정 무엇을 뜻하는지 모른다면 진실을 모르고 살 때와 다를 바 없다는 게 더 큰 문제다.

헛소문 1, 자바는 쉽다

워낙 자바가 유행하다보니, 자바를 프로그래밍 언어로 처음 배우는 사람이 많다. 심지어 일부 대학에서도 신입생에게 자바를 가르친다고 들었다. 하지만 함부로 그런 결정을 내리기 전에 배우는 이들이 어떤 영향을 받게 될지 진지하게 고민해 볼 필요가 있다. 만일 졸업하기 전, 한 5~6개월 정도 '실무 적응 훈련' 식으로 자바나 C#을 가르치겠다면 말릴 이유가 없다. 그리고 그런 연습은 학생 스스로 할 수 있어야 한다. 그것이 안 된다면 잘못 가르치거나 제대로 공부를 안한 것이다. 정말 자바(여기서 자바라고 하면 J2EE 등 관련 기술을 아우른다)나 C#(물론 닷넷을 아우른다) 외에는 공부할 게 없다면 몰라도 말이다. 운 좋게도 '뭘 배워야 할지' 또는 '뭘 가르쳐야 할지'를 아직 정하지 못했다면 이 글에서 실마리를 찾을 수 있었으면 좋겠다. 먼저 다음의 코드를 살펴보자.

```
import java.lang.*;

public class Test {
   public static void main(String[] args) {
      System.out.println("Hello, World");
   }
}
```

지겹도록 유명한 'Hello, World' 프로그램이다. C만큼이나 질기게도 버텨온 '불후의 예제'다. Hello, World 프로그램은 보통 처음 언어를 배우는 사람에게 가장 간단한 프로그램 구조를 보여줄 때 사용한다. 말하자면 앞의 코드가 '가장 간단한' 자바 애플리케이션인 셈이다. 그런데 수백 명이 모인 강단에서 이 가장 간단한 예제를 제대로 설명해야 한다고 하자. 모인 사람들은 처음으로 프로그래밍을 배워보겠다고 눈을 말똥거리며 앉아있다. 한 번 해보면 알겠지만 생각처럼 그리 쉽지 않다. 이 짧은 코드에는 정말 설명해야 할 것이 많다. 알다시피 이 예제에 나오는 모든 언어 구성요소를 정확히 알고 있다면(그 때문에 이 예제를 쓰겠지만), 적게 잡아도 자바 언어의 3분의 1은 익힌 것이다(JDK는 또 다른 문제이므로 여기서 JDK는 뺀다).

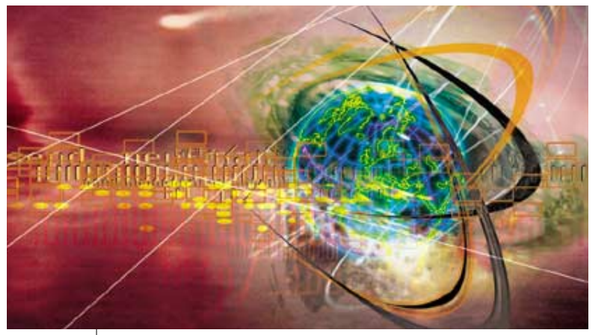
물론 편하게 넘어가고 싶다면 아주 좋은 방법이 있다. "자 이 예제는 이런 저런 글자를 화면에 찍습니다. 그러니 그런 줄 알고 외워 두도록 합시다". 이렇게 간단하게 말하고 시험삼아 컴파일이나 한 번 해주는 것이다. 대부분 첫 수업은 이런 예제로 별다른 설명 없이 지나가는 경우가 대부분일 것이다. 물론 몇 마디 설명을 덧붙이겠지만 필자가 말하는 '가장 편한 방법'과 크게 다르지는 않을 것이다. 어떤 방법을 쓴다고 해도 어려운 건 마찬가지다.

사실 지금까지 필자도 바로 이 예제를 그렇게나 잘 설명해보려고 여러 방법을 다 써봤다. 아니, 여태까지 새로운 언어를 가르칠 때마다 비슷한 예제로 설명했으니 'Hello, World 전문가'라 불릴 만하다. 예를 들어 import는 package와 함께 '전국을 왜 시 · 도 · 군 · 면으로 나누는가(Namespace Partition)'로, System.out은 '전라도.돌쇠(Qualified Naming)'를 부르는 것과 같다는 식으로 부족한 유머를 섞어가며 설명하곤 했다('전라도.돌쇠'는 처음부터 거기 있었다(static object)'라고 설명해야 겠다). 어쨌거나 배우는 사람들은 무척 재미있어 했다. 그런데 문제는 그 다음이다. 재미있다고 문제가 해결되는 것이 아니란 말이다. 나중에 알고 보면, 자바만 수개월 공부한 사람도 처음에 배운 이 예제 속에 들어있던 '언어 구성요소(Language Cons tructs)'의 의미(Semantics)를 제대로 모르는 경우가 아주 많았다. 그렇게나 쉽게 설명했는데 왜 잘 이해하지 못하는 걸까. 배우는 이가 노력을 덜한 탓일까. 물론 그럴 수도 있다. 그러나 꼭 그 때문만은 아니다. 자바가 정말 처음 배우는 프로그래밍 언어라면 절대 저 예제를 쉽게 이해할 수 없다. 그냥 글자 몇 자 찍는데도 저 난리 법석을 떨어야하는데 어떻게 쉬울 수 있을까. 세상에 저렇게 어렵고 가혹한 첫 번째 예제는 없다. 다시 말하지만, 그냥 배우는 사람의 지능이나 노력이 모자라서가 아니다. 이

유는 딱 하나, 자바가 어렵기 때문이다. 초보자에게는 자바가 자바로 짜야할 프로그램보다 훨씬 어려운 언어다.

은사의 말씀대로 '자바는 쉽다(Java is easier)' 뒤에 'C++보다(than C++)'라는 말이 숨어 있는 것일지도 모르겠다. 그러나 자바는 지금까지 쏟아져 나온 다른 객체지향 언어와 비교해봐도 절대 쉽지 않다. 그 '의미의 부모'라 할 수 있는 스몰토크에 비한다면 정말 복잡하다. 이제 이 예제를 다시보자. 그리고 뭘 얼마나 알아야 이해할 수 있는지 살펴보자. 다음은 Hello, World 예제를 이해하기 위해 알아야 하는 사항들이다.

- ◆ 먼저 import를 이해해야 한다. import는 package로 갈라놓은 name space를 다시 뭉치는 기능이라, 이 둘을 한꺼번에 알아야 한다. 그리고 하나 하나의 정확한 기능이 뭔지를 제대로 알려면 namespace가 뭔지, 왜 필요한지, 어떻게 써야 하는지 아는 것이 중요하다.
- ◆ java.lang/java.io 같은 namespace가 왜 있고 어떤 것들이 그곳에 살고 있는지, 적어도 동네 관광 정도는 해야 하지 않을까.
- ◆ public과 같은 access modifier를 알아야 한다. 이때, private/protected까지 건드려야 하는 것인지 한참 고민하게 된다.
- ◆ class와 static 사이에 알 수 없는 관계를 파헤쳐야 한다. 특히 static member 눈에 class는 그저 namespace일 뿐이다. 그 유명한 객체지향과는 아무런 상관이 없다. 동시에 System.out이란 녀석도 static object라는 걸 설명해야 한다. 더 나아가서는 그게 singleton instance라는 것과 그 이유도 살펴봐야 할 지 모르며, System 안에 있는 그 형제들도 알아야 할 필요가 있다.
- ◆ void main(String[] args)를 외우던지 아니면 이해해야 한다. C라도 다뤄본 적이 있으면 그나마 다행인데 아니라면 정말 힘들다. 아마 예제를 몇 개는 더 만들어 설명해야 한다. 배보다 배꼽이 크다.



눈에 보이는 것 위주로 몇 가지만 짚었다. 정말 처음 배우는 사람에게 이 예제를 설명하려면 이 정도에서 끝나지 않는다. 그래서 이 예제를 가르치려면 숨이 막힌다. 왜 이런 언어를 갓 들어온 신입생에게 배우라고 떠미는지 도저히 이해할 수 없다.

언젠가 이런 얘기를 동료에게 했더니 "이제는 import는 없어도 된다"고, "뭐 그런거야 차차 배우면 되지 않냐"고 한다. 그러나 필자 경험에 비춰볼 때 그건 잘못된 생각이다. 자바로 이런 저런 과제를 잘 해내던 사람들도 이 예제를 제대로 설명하지 못하는 경우가 많다. 그건 잘 모르고 쓴다는 얘기다. 물론, 잘 몰라도 프로그램을 만들 수는 있다. 또, 언어의 기능을 하나 하나 깊이 안다고 프로그래밍을 잘 하는 것은 더욱 아니다. 그러나 중요한 기능을 제대로 모르고 프로그램을 만들면 그만한 대가를 치르게 된다. 다시 말하지만, 프로그래밍 언어의 기능을 모두 알고 프로그래밍하는 사람은 없다. 그리고 그럴 필요도 없다. 그러나 프로그램을 만들 때 반드시 써야 할 기능이라면 정말 똑바로 알아야 한다.

간단히 말해 프로그래밍 언어란 '프로그램'을 만들려고 쓰는 도구다. 그런데 자바처럼 어려운 언어라면(C#, C++도 마찬가지다), 프로그램을 '만드는 방법'을 가르치는 것보다 '프로그래밍 언어' 그 자체를 배우다가 시간을 허비할 수밖에 없다. 처음으로 프로그래밍을 배우는 사람들에게 프로그래밍에 대한 '진지한 재미'를 가르치기보다 '언어의 난해함'을 일러줘야 할 필요가 있을까. 수개월을 공부한 다음에 남의 코드를 베껴쓰지 않고 제대로 만들 수 있는 것이 무엇인지를 보면 정확히 알 수 있다. 제발 이제 언어의 환상에서 벗어나 프로그램 짜는 법을 배우고 가르쳐야 할 때까 아닐까. 우리가 눈에 띄는 소프트웨어를 못 만드는 원인을 교육에서 찾는다면, 반드시 가르쳐야 할 것을 못 가르친 탓이지 어찌 그게 자바나 C#, 비주얼 베이직 같은 도구나 언어를 몰라서 일까. 그리고 이런 언어를 배울 수 있는 자격이 고등학교 졸업장은 가져야 한다고 치면, 이들은 그 시절에 그 어려운 미적분을 풀어내던 사람들이다. 선형대수와 기하를 공부한 사람들이고, 연립방정식을 풀려고 끙끙대던 학생들이다. 지나칠 정도로 많이 배우고 온 학생들이 자바나 C#을 배운 다음 만든 프로그램을 보라. 과연 누구의 잘못인가. 그들에게 가르쳐야 할 것은 새로운 것이 아니다. 이미 배웠던 지식을 제대로 활용해 컴퓨터를 이용해 문제를 제대로, 그리고 재밌게 푸는 방법을 알려주고 터득할 수 있도록 도와줘야 한다. 정녕 그렇다면 그 도구가 꼭 자바나 C#이어야 할 필요가 있을까.

헛소문 2, 자바는 플랫폼 독립적 개발 환경이다

자바가 처음부터 지금까지 고집을 꺽지 않고 주장하는 것이 'WORE(Write Once Run Everywhere)'다. 그렇다면 답이 뻔한 질문을 하나 해보자. 'JVM이 없어도 클래스 파일이 돌아가는가'. 물론 아니다. 그게 아니라면 어떻게 'Run Everywhere'인가. 반드시 JVM이 있어야 한다면, JVM은 또 하나의 플랫폼이 아니라고 할 수 있는가. 필자가 보기에 자바는 플랫폼 독립형 언어가 절대 아니다. 자바는 JVM이란 플랫폼에 아주 강하게 묶여 있는 언어다. 더욱이 그 지겹도록 쓸데없는 '순수성' 논쟁을 떠올리면 자유롭기보다 '폐쇄적'이라고 해야 옳다.

다른 사람의 얘기를 먼저 들어보고 얘기를 계속 하자. 자바 덕에 많은 질문을 받아서 시달렸을(?) 법한 사람을 꼽자면, 아무래도 C++를 만든 'Bjarne Stroustrup'을 뺄 수 없다. 아예 그의 홈페이지(http://www.research.att.com/~bs)에 가보면 FAQ에 이런 질문과 답이 있다.

질문 : 가령, C++를 만들 때 C와 소스 호환성을 신경쓰지 않았다면 자바와 비슷한 언어를 만들었을까요(프로그래밍 언어를 좀 안다면, 이런 주제로 한두 번은 토론해 봤을 것이다).

답 : 아니요. 자바는 그 근처에도 오지 못합니다. 사람들이 계속 C++와 자바를 비교하겠다면 일단 제가 쓴 'The Design and Evolution of C++(D&E)'를 보라고 권하고 싶습니다(정말 재미있는 책이다). 그 책을 보면 왜 C++가 그리 됐는지 알게 될 겁니다. 그런 다음 제가 C++를 만들 때 세운 설계 기준을 두고 두 언어를 비교해 보는 것이 옳습니다. 그 기준은 썬의 자바 팀이 세운 기준과 명백하게 다릅니다. 문법은 비슷할지 몰라도, C++와 자바는 아주 다른 언어입니다. 여러 면에서 볼 때, 자바는 C++보다 스몰토크와 더 비슷하다고 할 수 있습니다(MIT 대학의 B.Liskov도 교재에서 같은 말을 한다). 자바가 비교적 단순한데 새로 만든 언어는 대개 그렇습니다만, 그 단순함은 약간 허풍이라 할 수 있고 또 기능상에 결함이 있는 탓입니다(B.S는 'Template - Generic Programming' 자원이 빠진 것을 꼬집고 있다). 시간이 흐르면, 자바는 덩치가 커지고 더 복잡하게 변할 겁니다(이미 지금도 그렇다). 나중에 두세 배로 불어날 거고, 플랫폼을 타는 확장이 아닌 그런 라이브러리도 많아질 겁니다. 돈으로 성공하는 언어는 바로 그렇게 진화합니다. 크게 봐서, 성공을 했다고 볼 수 있는 언어 중 그 어떤 것이라도 예외없이 그렇게 흘러갔습니다. 그리고 이런 현상에는 그만한 이유가 있습니다. 자바는 플랫폼 독립형이 아닙니다. 자바가 바로 플랫폼입니다. 윈도우처럼, 분명 자바는 특정 회사가 소유한 상용 플랫폼입니다. 즉, 윈텔이나 자바/JVM에서 프로그램을 짠다는 말은 한 기업이 소유하는, 그리고 그 회사가 이익을 챙기려고 만든 플랫폼과 코드를 쓰고 있는 겁니다. 물론 자바가 아니라도, 아무 언어로나 JVM과 관련 운영체제 기능을 써서 프로그램을 짤 수 있다고 이미 말한 바 있습니다. 그렇지만 JVM 등은 정말 자바를 쓰면 더 유리하도록 만들어 놓았습니다. JVM은 진짜 '언어 독립적인 VM/OS'가 될 수 없습니다. 그 근처에도 오지 못합니다. 저는 이후로도 충분히 이식성있는 C++를 주 개발 언어로 계속 쓸 생각이고, 다른 일에는 여러 가지 언어를 골고루 섞어 쓸 겁니다.

이왕 나온 김에 C#에 대한 얘기도 들어보자.

질문 : C#에 대해서는 어떻게 생각하십니까.

답 : 언어로서 C#에 대한 얘기는 하지 않겠습니다. 이 세상에 또 다른 상용 언어가 있어야 한다고 줄기차게 설득합니다만, 특정 운영체제에 들러붙은 언어가 또 필요하다면 저를 설득하기 어려울 겁니다. 잘라 말하지만, 전 상용 언어를 그렇게 좋아하는 사람이 아닙니다. 저는 공개해서 표준이 된 기술을 좋아합니다.

자바가 성공한 뒤에는 분명 C++가 있다. 바로 얼마 전까지만 해도 C++는 자바와 마찬가지로 피해갈 수 없는 관문이었다. 앞의 얘기가 C++ 편을 들 수밖에 없는 사람의 얘기라고 할 수 있다. 절대 C++가 쉽고 결함이 없다는 얘기는 아니다. 말할 필요도 없이 C++는 어렵고 자바만큼이나 결함이 많은 언어다. 그러나 자바도 그에 못지 않게 어렵고, C#은 자바보다 더 어렵다. 상자 안에 갇혀 바퀴를 돌리는 다람쥐 입장에서 보면 아주 먼 거리를 열심히 뛰어다니지만, 자유롭게 앞으로 내달리는 다람쥐에게 그건 달리고 있는 것이 아니다. C++에서 자바, 그리고 C#으로 이어지는 고리, 바퀴를 돌리고 있는 것은 아닌지 진지하게 생각해 볼 일이다.

다시 플랫폼 독립성 얘기로 넘어가자. '플랫폼 독립성'이란 그 실상이 어떤지를 따져서 판단할 문제다. 원칙과 이론이 그렇다고 주장해봐야 별 의미가 없다. 다시 말하자면 플랫폼 독립적인 기술이라고 강조하는 것보다 바로 지금 얼마나 많은 플랫폼에서 잘 돌아가며 앞으로는 어떻게 될 것인가라는 예측 가능한 근거가 더 중요하다. 그리고 플랫폼 독립성은 쓰는 사람의 관점에 따라 달리 볼 수 있는 문제이기 때문에 그 목적을 이루는 방법도 여러 가지고, 자바 방식만이 옳다고 할 수는 없다.

플랫폼 독립성은 개발자에게 좋은 무기다. 소스가 됐든 바이너리가 됐든 맨날 하던 일을 또 다시 하지 않도록 해주겠다는 것이

나쁠 리 없다. 사용자 쪽에서도 이 플랫폼에서 쓰던 프로그램을 저 플랫폼에서 똑같이 쓸 수 있다면 정말 좋은 일이다. 그런데 그런 목적을 두고 보자면, 솔직히 자바보다 GNU C/C++가 더 많은 플랫폼에 퍼져 있지 않은가. 그리고 바이너리 호환의 가치도 크지만 소스 코드 호환은 나쁠 것이 뭔가. 게다가 속도가 느려지는 대가를 따지면 선뜻 뭐가 낫다고 하기 힘들다. 알다시피 자바는 그 목적을 이루는 방법(JVM이란 가상 머신이 있어야 한다는 것) 때문에 지금까지도 느리다는 불평을 많이 들어왔다. 물론 다시 케케묵은 성능 논쟁을 할 필요는 없다. 지금 와서 그래봐야 의미 없는 일이다. 적어도 공평하지는 않다. 어떤 언어로 만든 프로그램이라도 VM에서 작동하면 느려지는 것이 당연하다. 자바 이전에도 VM/OS 식으로 돌아가는 언어 환경은 많이 있었고, 그런 언어로 만든 프로그램도 느리긴 마찬가지였다. 더구나 어느 책에서 하는 말처럼 자바로 작성한 프로그램이 느린 이유가 프로그래머들이 프로그램을 제대로 잘못 작성해서 그런 것이지, 모든 것이 자바 탓은 아닐 것이다. 필자는 그런 작은 것을 놓고 따지자는 것이 아니다. 실제로 자바식 '플랫폼 독립'으로 뭐가 이득인지 따져봐야 한다는 말이다. 그리고 그 얘기는 뒤에 다시 하겠다.

아무튼 정리하자면, 자바만이 '플랫폼 독립'의 유일 무이한 해결책은 아니라는 것이다. 그리고 '독립성'은 그 실제 효과로 따져야 옳다는 걸 덧붙인다. 그래야 말이 된다. 그리고 요새 C#도 비슷하게 '언어 독립성'을 강조하지만(또 BSD 플랫폼에 닷넷 프레임워크를 옮겨 심는다는 소리도 들린다), 그 결과는 두고봐야 알 일이다. 자바랑 뭔가 크게 다른 방식이라고 하지만 크게 보면 다를 수가 없다. 자바도 처음에는 비슷하게 시작했다는 걸 잊지 말아야 할 것이다. 다행스럽게 두 해결책이 모두 처음에 한 약속을 제대로 지키지 못해도(아마 그럴 거다), 그 대안은 있다. 아니, 얼마든지 있다. 그렇다면 어떻게 될 지 모를 결과를 기다리느니, 지금 나와있는 걸 쓰는 게 더 낫다. 더구나 그 대안은 대부분 공개 기술이다.

그런 기술이 없어서 문제가 되는 것은 아니다. 모두가 그 대안에 별로 진지한 관심을 쏟지 않는다는 것이 문제다. 그리고 이런 현상을 책임져야 할 곳은 대학을 비롯한 교육 기관이 아닐지... 사실 학원은 어쩔 수 없다 치더라도 대학에서는 이론보다 바로 써먹을 수 있는 것을 강조하는 우를 범하지 말아야 한다. 학생들을 무슨 특정 회사의 홍보 요원이나 전용 기술자로 만들어 절름발이 기술자로 키우는 것과 별 다를 게 없다. 부산에 있는 모 대학은 특정 회사의 이런 저런 소프트웨어를 사용하기로 공식 계약했다고, 그걸 무슨 큰 자랑인 듯 광고까지 하는 걸 봤다. 광고에 앞서 정말 부끄러운 일이 아닌지 생각해 볼 일이다.

이 글을 쓰다보니 오래 전에 봤던 Alexander Stepanov와의 인터뷰 기사가 생각난다. A. Stepanov는 아주 오랫동안 '포괄 프로그래밍(Generic Programming)'을 연구해온 사람이다. C++의 STL이 그의 연구 결과이자 작품이다. 엔지니어라면 한 번쯤 읽어볼 만한 글이라서 '이유'를 설명하는 대신에 일부만 인용해 본다. 독자 여러분 스스로 그 답을 찾기 바란다(그는 객체지향 프로그래밍을 별로 미덥지 않게 여기기 때문에 약간 치우친다는 느낌이 들지만 근거 없이 하는 말은 아니다. 뒤에서 필자도 비슷한 얘기를 할 것이다). 물론 원문(http://www.stlport.org/resources/StepanovUSA.html)을 읽어봐도 좋다.

질문 : 자바는 새로 나온 언어인데, 아직 템플릿 같은 것이 없습니다. 그래서 포괄 프로그래밍을 할 방법이 없습니다. 모든 것을 클래스로만 표현해야 합니다. 자바를 어떻게 생각하십니까(뭐 생각하고 말고 할 것이 있겠는가).

답 : 자바로 몇 달 프로그래밍을 해봤습니다. 만든 사람이 믿는 것과 달리 별로 끌리는 데가 없더군요. 새로운 깨달음이랄까, 뭐 그런 걸 찾을 수 없었습니다(그렇게 새로운 뭔가를 발견할 수 없는). 그런 언어는 내 평생 처음입니다. 게다가 내가 C++로 프로그램을 만들 때 절대 쓰지도 않던 기능, 즉 상속이나 메쏘드 재정의 등 뭐 그런 객체지향 특징은 다 갖고 있으면서 꼭 필요하고 쓸모 있던 기능은 아예 다 빼버렸더군요. 자바는 분명 성공했다고 할 수 있습니다. 그리고 결국 MS-DOS가 그랬듯이 모든 사람에게 자바를 배우게 만들어 수익을 챙길 수는 있겠지만, 지적 가치는 전혀 없습니다. 해시 테이블을 어떻게 만들었는지 보십시오. 또, 그 좋다는 애플릿을 줄 때 덤으로 따라오는 정렬 기능을 보세요. AWT도 써봐야 합니다. 그래서 언어가 좋은지 나쁜지 제대로 알고 싶을 때는 그 언어를 좋아하는 프로그래머가 어떻게 코드를 썼는지 보는 것이 가장 좋은 방법입니다. 그리고 자바는 돈 지향 프로그래밍 언어(Money-oriented MOP)가 뭔지를 확실하게 보여줍니다. 언젠가 SGI에서 자바를 가장 열렬하게 지지하는 사람이(Alex는 STL을 발표할 당시만 해도 HP에 있었다. 그러나 HP는 연구실을 닫았고 이후 SGI로 옮겼다) "알렉스, 당신도 돈이 되는 쪽으로 연구를 해야 합니다"라고 하더군요. 그렇지만 나는 꼭 돈이 되는 쪽으로 가고 싶지는 않습니다. 보통 그리로 가면 냄새가 좋질 않아요.

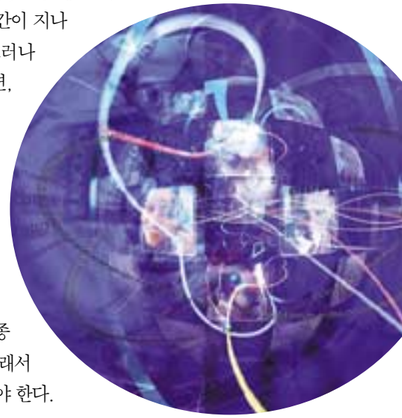
필자는 Stepanov처럼 '기술의 순수함'을 고집하는 사람은 아니다(사실 아직 그럴 자격도 갖추지 못했다). 여기다 필자가 버는 돈이 거의 자바 때문이라 투덜거릴 자격이나 있는지 모르겠다. 너무 자바나 C#만이 전부인 것 처럼 여기는 것 같아 조심스럽게 건내본 얘기다. 더구나 둘 중에 뭐가 더 좋다고 벌이는 논쟁은 발전적이라기 보다 소모적이라는 느낌을 줄 때가 많다. 필자는 올해만 해도 정부가 지원하는 개발자 교육 선정에 여러 번 참가했다. 그런데 거의 모든 교육과정에 자바와 닷넷이 빠지는 것을 보지 못했다. 물론 자바를 잘 쓰면 좋은 일이고, 그걸 배워서 돈을 벌 수 있다면 말릴 일이 아니다. 그런데 배우고 쓸 프로그래밍 도구가 꼭 자바와 닷넷밖에 없는지, 그리고 사용자에게 꼭 자바나 닷넷 상품만 쓰라고 강조해야겠는가.

헛소문 3, 자바는 '순수' 객체지향 언어다

수년 전, 연구실 후배들과 모 회사의 자바 프로그래머가 뉴스그룹에서 한 바탕 설전(그것은 토론이라고 할 수가 없다)을 한 적이 있다. 서로 한참 억지를 쓰며 우겨대다 투덜거리며 끝이 났는데, 본래 이런 싸움에는 승패도 없고 기분만 나빠지게 마련이다. 한 마디로 가치 없는 싸움이다. 어처구니없지만, 그 주제는 대충 '자바의 순수성'에 대한 얘기였다. 아마 그 때 썬과 MS가 델리게이트(delegate) 예약어를 갖고 한참 자존심 대결을 한 뒤가 아닐까 싶다. 그 사건은 필자의 후배 중 누가 J++가 더 낫다는 분위기로 글을 올려서 시작된 모양이었다(자바로 윈도우용 프로그램을 짜고 싶다고 한 걸로 알고 있다). 정말 그랬다면 아예 드러내놓고 한번 싸워보자고 나선 것이다. 아니라면 한참 자바 바람이 불던 그 때에 자바 뉴스그룹에다 그런 글을 올리는 무모한 짓을 할 리 없다.

그런데 이 분야에서 일하다 보면 의외로 이런 논쟁을 많이 하는 걸 볼 수 있다. 참 이해 못할 일이다. 일 잘하는 목수가 제 망치를 아끼는 것은 이상하지 않다. 그리고 망치가 좋다고 자랑할 수도 있다. 그러나 다른 목수의 망치와 견주어 자기 것이 더 낫다고 우길 필요가 있을까. 더구나 망치보고 '이건 순수한 낫이다'이라고 우기는 건 또 뭔가. 그건 더 우스운 일이다. 한 마디로 말해 자바는 그렇게 '순수'하지 않다고 할 수 있다. 누가 스몰토크나 BETA을 두고 그리 얘기했다면 조금 고개를 끄덕여 줄지 모를 일이다. 그러나 자바는 그렇게 깨끗하지 않다. 그리고 프로그래밍 언어가 순수하다는 것은 자랑이 아니다. 그건 되려 엄청난 약점이다.

다들 알다시피 자바는 서로 너무 다른 두 세계를 '억지로' 붙여 놓았다. 스몰토크 등으로부터 빌어온 세계에다, C/C++에서 살짝 때어온 귀퉁이를 Wrapper 클래스로 기워놓은 셈이다. 두 동네가 너무나 다른 나머지, 서로 완전히 다른 법칙으로 다스리기 때문에 처음 자바를 배우는 사람들을 정말 괴롭힌다. 그래도 이 문제는 좀 시간이 지나면 그런 데로 참을 만하다. 그러나 value type을 새로 만들어 보려면, 그걸 object type으로 만들어야 하고, 서로 잘 섞어 쓸 수도 없는 이상한 클래스를 만들어 써야 한다(그래서 그런지 이런 것을 실제 만들어 쓰는 걸 별로 보지 못했다). 예를 들어 보자. 자바로 유리수를 만든다고 하자. '값' 세계에선 새로운 종(Type)을 만들 방법이 없다. 그래서 무조건 Object Type으로 만들어야 한다.



```
class Rational extends Number {
   public Rational(int n, int d) { ... } // (n/d)
   ...
   // for Number compliance
   public double doubleValue() {...}
   ...
}
...
Rational u = new Rational(3,2); // 3/2
Rational v = new Rational(6,4); // 6/4
```

코드를 보자. 여기서 u, v값을 ==로 비교해 봐야 답은 false

다. Rational은 Class(Object) Type 객체라서, u. v 변수는 객체를 가리키는 참조(Reference, C언어의 포인터) 값이다. 다시 말해, 객체 세상에서 ==은 두 포인터가 같으냐 다르냐를 비교할 뿐이다. 그러므로 Rational과 같은 클래스를 이렇게 놔두면 안된다(왜 이런 문제가 생겼을까?). 이 문제를 해결하려면 u, v가 가리키는 객체의 값을 비교하도록 새 메쏘드를 더해야 한다.

```
class Rational {
   ...
   public boolean isSame(Rational r) { /* 최대공약수로 나누어 분자,
   분모를 비교하라. */ }
   ...
}
```

이제 값으로 비교하고 싶을 때 isSame을 쓸 수 있다. 그러나 이것만으로 문제가 완전히 해결되지 않는다. 남아 있는 문제를 설명하기 위해서 다음처럼 집합을 만들어 insert로 원소를 집어넣는 예를 들어보겠다.

```
class Set {
   ...
   public void insert(Object e) { /* 같은 객체가 없다면 집어 넣는다. */ }
   ...
}
```

Rational 객체만 집어넣을 수 있는 집합을 만들어 봐야 별다른 의미가 없기 때문에, 꼴이 다른 객체도 넣을 수 있도록 집합을 만드는 것이 보통이다. insert는 java.lang.Object를 받는다. 이 때 다음과 같이 Rational 객체를 집어 넣는다.

```
Set s = new Set();
...
s.insert(u); s.insert(v);
...
```

앞에서 u와 v는 같은 값이므로 들어가지 않아야 옳다. 이걸 검사하는 데 ==를 쓸 수 없기 때문에, 앞에서 만든 isSame을 쓸 수 있어야 한다. 그래서 java.lang.Object에 isSame과 같은 기능이 있어야 하고, Object 클래스에 equals(Object)이 바로 그런 역할을 한다. 종합하면 Rational은 대충 다음과 같이 고쳐 써야 옳다. 물론 isSame 대신에 java.lang.Object.equals를 덮어쓴다.

```
class Rational extends java.lang.Number { // Rational도 Number다.
   ...
   public int denom();
   public int numer();
   public boolean equals(Object e) {
      if (this == e) return true;
          if (obj != null && e instanceof Number)
               if (e instanceof Rational) {
                 return this.numer() == e.numer() && this.denom()
                    == e.denom();//  ❶
                 }
          return this.doubleValue() == e.doubleValue();
          }
      else return false;
  }
          ...
}
```

자, equals 안의 코드를 눈여겨보자. 코드 안의 두 유리수를 비교하는 데 꼭 필요한 코드는 ❶뿐이다. 나머지 코드도 어떤 기능인지 쉽게 알 수 있다. 간단한 코드지만, '어떤 언어가 얼마나 괜찮은지' 단번에 알 수 있는 예제이기 때문에 그 판단 기준은 뚜렷하다. 모자란 언어는 모자란 만큼 사용자(프로그래머)에게 그것을 떠넘긴다. 문제는 이렇게 복잡한 과정을 거쳤지만 이것도 완벽한 해결책이 아니라는 것이다.

첫째, 유리수는 수니까 다른 종의 수와 값을 비교하는 것이 당연하다. 객체 마을에서 모든 '수 객체'를 대표하는 종은 Number이므로 Number와 비교할 수 있도록 만들어야 한다. 그런데 본래 '같다는 관계'는 서로 바꿀 수 있어야 한다(symmetric relation). 자바에서 이 문제를 해결할 깔끔한 방법은 없다. 그럴 때 필자는 가끔씩 다음과 같이 해결하기도 한다. 물론 전혀 마음에 들지 않는다.

```
class NumberUtil {
   public static boolean equals(Number left, Number right) {
      if (left != null && right != null)
         return left.doubleValue() == right.doubleValue();
      return false;
      }
...
   private NumberUtil() {}
}
```


둘째(이건 좀 다른 문제인데) equals에다 어울리지 않는 객체를 집어넣어도 답은 언제나 false라는 점이다. 예를 들어 다음과 같이 이상한 코드를 써도 답은 나온다(이것을 이상하게 생각하지 않는 것이 더 이상하다).

```
class Rabbit {...}
...
u.equals(new Rabbit()); // 토끼와 유리수가 달라서 false?
```

이런 경우에 Exception으로 처리할 수도 있겠지만(원칙으로 따지자면 이 방법이 더 옳다), 좋은 방법이라 할 수 없다. 메쏘드 equals를 쓸 때마다 매번 try-catch로 둘러싸거나 아니면, '언제 Exception이 튀어나올까?' 불안에 떨어야 하기 때문이다. 제대로 하자면 이런 오류는 컴파일할 때 잡아낼 수 있어야 한다.

물론 자바만 이런 문제로 시달리는 것은 아니다. 본래 객체지향 언어에서 수를 섞어 쓰는 일(Mixed Arithmetic problem)은 골치 아픈 문제고, 그럴듯한 해결책이 없다. 아마도 이 문제 때문에 자바는 울며 겨자 먹기로 value type을 끌어들였을 것이다. 다시 말하자면 여러 꼴의 수 객체끼리 섞어 쓰는 문제를 서브타이핑(Subtyping)으로 해결하지 않고, C/C++로부터 이어져온 이전 방식을 그대로 사용했다고 할 수 있다.

필자는 이것이 도대체 객체지향과 무슨 관계가 있는지 묻고 싶어 오래된 얘기를 다시 들춰냈다. 그래도 순수하다고 생각한다면, ==이 메시지가 아닌 이유는 무엇일까. '순수' 객체지향 언어라면서 객체가 아닌 뭔가가 '버젓이' 있고, 메시지도 아닌 기호들이 있다는 것을 어떻게 받아들여야 하는가. 순수 객체지향 언어라면 3+4도 순전히 객체간의 메시지 전달(Message-passing)이어야 한다는 말이다. 3도 객체고 4도 물론 객체여야 하고, +는 메시지여야 말이 된다. 그리고 메시지는 당연히 객체마다 해석을 달리 할 수 있어야 하므로, 클래스마다 다시 정의할 수 있어야 한다(그렇게 하라고 메쏘드가 있고, Method Dynamic Binding이 있다). 그러면 equals같은 것이 꼭대기 Type인 Object에 포함돼 덩치를 불릴 이유가 없다. 변명할 수는 있다. '객체를 만드는 대가가 커서 효율성 때문에 어쩔 수 없었다' 라고. 물론 Heap Object를 쓰는 대가가 만만치 않다는 건 잘 아는 사실이다.

다시 Rational을 보자. Rational로 객체를 수백 개 만들면 대가는 여전하다. 정말 효율성이 걱정됐다면, Rational을 Heap 객체로 만들도록 방치한 이유를 어떻게 설명할 것인가. Value Type을 줄여 쓰려면, 그 스스로 새로운 Type을 만들 수 있도록 해줘야 했다. 그러면 괴상한 객체도 아니고 수도 아닌 Wrapper 클래스는 없어도 좋았다. 거기다 객체와 수를 섞어 쓰면 지저분해지는 코드는 더 말할 필요도 없다. C#에서는 이 문제를 struct라는 기능으로 해결한다. 그리고 Wrapper 클래스를 두는 대신에 Boxing을 쓴다. 물론 이도 전혀 새로운 해결책이라 할 수 없다. 잘 쓰는 말로 '꽁수'다. 결국, C# 방식은 C++의 해결책과 다를 게 없다. C#은 참말로 '오직 닷넷만을 위한 C++ 지향적 자바(C++-oriented Java only for .NET)'이다. C#은 자바가 '단순함'을 외치며 버렸던 C++의 유산을 다시 끌어 담았다. 그리고 지금 와서 새로운 언어라고 강조하고 있다. 이런 것을 두고 '다람쥐 쳇바퀴 돈다' 라고 표현한다. 이렇게 돌아올 거면 '참신한 언어(Brand new language)' 란 말을 써서는 안된다.

이제 정리해 보자. 이유야 어쨌든, C#이나 자바 모두 객체지향이 아닌 기능도 필요하다고 인정하고 있다. 그렇다면, 더 이상 '순수' 란 말을 쓰지 않아야 옳다. 또, 자바의 '순수성' 이라는 애매한 말을 놓고 다툴 필요도 없다. 자바나 C#에 비해 훨씬 '순수' 한 스몰토크는 같은 문제를 저렇게 처리하지 않기 때문이다. 분명한 것은 객체지향 언어는 쓸 만하지만 '순수' 객체지향 언어는 결코 좋은 게 아니라는 것이다.

헛소문 4, 자바는 '새로운' 객체지향 언어다

'자바 2'라고 부르는 커다란 변화를 말하자면, 가장 먼저 떠오르는 것이 anonymous object다. 아마 다른 언어에서 비슷한 기능을 즐겨 사용하다 보니 자바에 그런 것이 없어 갑갑했던 모양이

다. 물론 컴포넌트 방식의 프로그래밍이라는 큰 변화에서 보면 anonymous object는 그저 속임수를 써서 가져다 붙인 아주 작은 기능일 뿐이다. 하지만 '그걸 왜 넣었을까?' 하고 따라 올라가보면 뭔가 색다른 시도라는 것을 알 수 있고, 앞으로 프로그래밍이 어떻게 달라질지 미루어 짐작할 수도 있다. 언제나 그렇듯이 겉으로 보기에는 작지만, 그것이 큰 변화를 암시하는 증거가 되는 경우가 있다. Anonymous object가 바로 그런 증거라 할 수 있다. 이런 얘기가 더 재미있긴 하겠지만 다음 기회로 넘기고, 여기서는 자바 2의 변화와 객체지향에 대한 의미만 살펴보겠다. 그동안 귀가 따갑도록 들어온 객체지향 개념을 다른 시각에서 보고자 하는 것이다.

자바 2에서 시작된 변화, 즉 부품 결합 방식으로 프로그래밍하자는 것은 그냥 언어 기능을 확장한 정도가 아니라, '기법이 크게 변했다' 고 봐야 옳다. 이걸 쓸 수도 있고 쓰지 않아도 되는 '부가 기능' 정도로 받아들이면 설계에 미치는 중요함을 되새겨볼 수 없기 때문에, 제대로 쓰는 방법을 익힐 수 없다. 적어도 '객체' 로 프로그래밍하는 세계에서 만큼은 '객체를 어떻게 묶어야 좋을까' 하는 생각의 틀을 바꿔놓은 중요한 도약이다. 그러므로 객체지향을 기본까지 거슬러 올라가서 한 번쯤 그 의미를 되새겨봐야 참된 변화의 깊이를 잴 수 있다.

먼저 이것도 '예고된 진화' 였을 뿐이며, 당연히 그랬어야 하는 것을 뒤늦게 실천으로 옮긴 것 일뿐이라는 걸 알아두자. 자바 안에서만 보자면 '혁신' 이랄 수 있지만, 크게 보면 그저 '답습' 일 뿐이다. 사실 컴포넌트 결합 방식이 괜찮다는 얘기나, 그래서 생산성을 크게 올릴 수 있었다는 성과보고는 90년 중반에 이르러 거의 마무리됐다. 많은 사람들이 쓰고 있는 비주얼 베이직과 델파이 등이 그 성공의 흔적이라 할 수 있는데, 그 덕에 개발이 얼마나 쉬워졌는지는 다시 거론할 필요도 없다. 그런데 왜 자바에서는 이 중요한 경험을 반복 답습했을까. 결국 이런 방식을 받아들일 거라면, 아예 처음부터 그렇게 갔어야 옳지 않았을까. 특정 기업이 기술 변화를 주도하면서 이익을 목적으로 이런 변화를 악용한 셈인데, 90년대에 들어서는 이런 현상이 아주 심하게 나타났다. 설사 마음먹고 한 짓이 아니라 하더라도 결국 피해를 입는 것도 개발자고, 이익을 보는 쪽도 개발자다. 응용 프로그램을 파는 회사에서 프로그램의 버전을 부풀리는 것은 어쩔 수 없다. 그러나 아무리 상품이라고 해도 '바탕 기술' 을 그렇게 다루는 것은 우려스럽다. 응용 프로그램의 버전이 바뀌는 것과는 그 무게와 깊이가 다르기 때문에 그 순수하지 못한 변화로부터 오는 충격은 생각보다 크다. 하물며 덜 만든 기술을 빨리 상품으로 내놓은 것도 모자라서 당연히 그랬어야 하는 것을 무슨 대단한 업적인 것처럼 자랑하는 것은 사용자를 우습게 본 것이라 할 수 있다. 그 덕분에 이제는 이런 방식이 공개로 개발하는 소프트웨어나 이익을 목적으로 하지 않는 기술 사회에도 뿌리를 내리고 말았다. 어디를 돌아봐도 다음 버전이 며칠, 몇 달만에 툭툭 튀어나오는 바람에 사람들은 그 속도를 따라 잡느라 정신없다. 새로 나오는 기술을 금방 금방 따라 잡지 않으면 큰 병에 걸리는 것처럼, 이미 나온 것을 채 받아들이기도 전에 새로 나온 것에 매달린다. 전형이랄 수 있는 예로, 모 회사에서 나온 게임 개발 툴킷은 단 1년만에 서너 판을 훌쩍 뛰어 넘었고, 책방에 가면 새 판에 대해 소개한 책과 이전 버전에 대해 소개한 책이 나란히 걸려있기까지 했다. 이런 식이니 모르는 사람들은 자고 나면 바뀌는 분야라고들 할 수밖에 없는 것이다.

예전에는 엔지니어들은 이렇게 조급증에 걸려있지 않았다. 그런데 요새는 그 기술의 무게는 재보지도 않고, 그저 사탕가게에 가 새로운 과자를 기다리는 어린아이 같이 조바심을 내며 줄을 서는 판국이다. 잠시 보던 책을 덮고 차분하게 생각을 해보자. 바뀌는 것 중에 정말 '바뀐 것' 이 얼마나 있던가. 바탕이 바뀌는 거야 놓치면 후회할 일이지만, 그런 큰 변화를 주도하는 이치는 바로 나오는 것도 아니고, 자바 2와 같이 대부분 예고된 변화라고 할 수 있다. 그러므로 그런 '진화' 는 정말 기다려왔던 것이며 좀 더 빨리 받아들이는 게 맞다. 하지만 그렇지 못한 것에 대해서는 귀를 닫고 지내는 것이 더 낫다. 필요하면 그때가서 배워도 늦지 않다.

파이썬과 QT로부터 배워야 할 교훈

앞에서 WORE에 대해 얘기를 풀어놓았으니, 길은 다르지만 목적을 같이하는 다른 기술을 살펴보는 것이 좋겠다. 비슷한 듯 하지만 다른 것을 서로 견주어 보면서 참뜻을 잡아나가는 것도 좋은 방법이다. 이미 말한 바와 같이 플랫폼 독립성이 추구하는 목적은 '이식성(Portability)' 이다. 이식성이란 가치는 '이론' 에서 벗어나 실무에서 확인돼야 의미가 있다. 잘 알듯이 인터프리터에서 돌아가는 언어는 대부분 VM/OS 방식을 써왔고, 컴파일러 방식을 쓰는 C/C++ 같은 언어라도 이식성이 중요하면, 컴파일러 자체를 VM/OS와 동일한 구조로 만든다. 앞서 소개했지만 자바의 VM/OS 구조는 정말 오래된 기법이다. 굳이 자바에서 한 일이라면 이 고전 기법을 JVM이라는 상품으로 만들어내고, 우연히 웹 덕분에 시장에서 성공을 거두었다는 것밖에 없다. 지금이라도 JVM에서 돌아가는 클래스 바이너리만 만들어내면, 꼭 자바 언어를 써야할 이유는 어디에도 없는 것이다(http://www.jython.org에 들러보자). 더욱이 이미 비슷한 방식을 쓰는 언어라면 JVM에서 돌아가게 만들기가 쉬울 것이다. 실제 이런 시도는 아주 많았고, 지금도 계속되고 있다. VM/OS 방식을 쓴다는 사실은 이미 여러 개의 언어를 섞어 쓴다는 말인데 이론으로 볼 때 아무런 하자가 없다. 즉 어떤 다른 의도가 없다면 언어 독립성은 거저 얻을 수 있는 것이다. 닷넷도 비슷하다. 아니 몇 가지 군더더기를 걸러내면 VM/OS 방식과 거의 다를 게 없다. 그리고 플랫폼 독립성 대신 언어 독립성을 말하지만, 앞서 내린 결론처럼 자바라고 언어 독립성이 없는 게 아니라서 그런 주장은 '조삼모사' 와 같은 것이다. 그리고 닷넷도 자바처럼 플랫폼 독립성을 얻지 못할 아무런 걸림돌이 없다. 오히려 문제는 플랫폼 독립성을 얻기 전에 일단 각 플랫폼에 어떻게든 이식하고 봐야 한다는 것이다. 거기다 언어 독립성을 얻으려면 각 플랫폼에서 돌아가는 바이트코드 또는 그와 비슷한 코드를 내놓도록 컴파일러를 새로 만들어야 한다. 그런데 정말 그들의 주장대로 목적을 이룰 수 있을까.

빛 바랜 'WORE'

일단 결과부터 보자. 현재 WORE는 생각만큼 빛을 보지 못하고 있다. 겉으로만 보면 정말 수많은 플랫폼으로 JVM을 옮겨 심었거나, 옮기는 중이다(http://www.geocities.com/marcoschmidt.geo/java.html). 그런데 실제 얼마나 많은 플랫폼에서 JVM으로 프로덕션 코드를 만들고 있는가. 우리가 쓰는 응용 프로그램 중에 이른바 '순수' 자바로 만든 프로그램은 몇 개나 되고, 현재 자기가 쓰는 플랫폼에서 믿을 만한 코드를 만들어 내고 있는가. 결국 VM/OS 방식이 실제 성공을 거두려면 각 운영체제마다 이식하는 VM이 튼튼해야 한다. 그리고 각 운영체제별로 꾸준한 지원이 있어야 하는 것도 당연하다. 그러나 기업은 봉사단체가 아니다. 그러므로 이익을 낼 수 없는 운영체제에 VM을 심자고 돈과 노력을 들여야할 이유가 없다. 하물며 사후 지원을 기대하겠는가. 설사 그런 의지가 있다고 해도 그 많은 운영체제에 어떻게 JVM을 이식하고 지원할 수 있겠는가. 리눅스만 하더라도 JVM을 옮겨 심고, 공식 지원하기 시작한 것은 얼마되지 않았다. 게다가 그 작업마저도 남들이 한 것을 가져다가 시작했다(J2EE까지 예를 들어 한마디 하고 싶지만 JVM 얘기는 일단 접어두자).

이제 닷넷을 빌어 '과대 광고' 에 대한 비판을 마무리지어 보겠다. 오라일리와 Hejlsberg의 인터뷰 기사를 살펴보면서 그 실상을 이해하면 도움이 될 것이다(http://windows.oreilly.com/news/hejlsberg_0800.html). 이 대화에서 알 수 있듯이 닷넷 IL이나 자바 바이트코드와 같은 기술은 정말 오래된 기술(p-code Engine)이다. 물론 IL이 자바와 크게 다르다고 열심히 설명하지만 그저 닷넷용 JIT를 상품으로 만든 정도라고 봐도 무리가 아니다. 물론 자바 바이트코드 처리 방식보다 여러 면에서 나은 점이 있을 수 있겠지만, 같은 일을 한참 뒤에 하면서 더 잘 하지 못한다면 그게 더 이상한 일이다. C#도 마찬가지라고 할 수 있다. 써보면 자바보다 이런 저런 면에서 더 편한 면이 분명히 있다. 그런데 이미 시장에 나와 이리 저리 불평을 들어온 언어를 실컷 보고, 거의 같은 언어를 한참 뒤에 만들면서 더 잘 만들지 못한다면 그게 더 이상한 일 아니겠는가. 앞의 글에서 인용한 Alex와 B.S의 말을 다시 떠올려 보자. 그리고 이 모순의 근원이 어디서부터 왔는지 가만히 생각해보자. 굳이 여기서 중언 부언하지 않더라도 그 답은 바로 나와야 한다.

올해 닷넷 프레임워크 시험판이 나오면서 Python.NET과 Haskell.NET 등을 지원할 지 모른다는 소문이 떠돌았다. 아직은 구경해본 적이 없지만 실제 그런 컴파일러가 있을 지도 모른다. 그러나 소문처럼 그런 이상은 쉽게 실현되지 않을 것이며, 실효를 거두기란 더욱 어렵다. 그리고 닷넷의 주장과 달리 여러 언어가 같은 런타임을 함께 쓰는 것은 정말 이루기 힘든 목표다. 프로그래밍 언어마다 얼마나 많은 차이가 있는지 알고 하는 얘기라

면, 한 마디로 용감하다고 할 수밖에 없다. 마치 에스페란토어로 모든 나라 사람들이 막힘 없이 자기 감정과 철학을 옮겨 담을 수 있다는 주장과 다를 게 없다. 그리고 실제 그런 컴파일러가 공개 배포된다고 해도 유행하지 않는 언어를 적극 가르치고 홍보하는 교육 기관은 거의 없을 것이다(사실 이게 목표일 지도 모른다). 그리고 조만간에 자바처럼 이 분야에 갓 들어온 이들에게 C#을 가르쳐야 한다고 주장할지도 모른다.

파이썬과 Qt의 이식성

우리는 눈을 딴 곳으로 돌릴 필요가 있다. 그리고 파이썬과 Qt로부터 우리가 놓치고 있는 뭔가를 좀 배워야 한다. 하나는 프로그래밍 언어로서 그 이식성을 배울 만하고, 다른 하나는 툴킷으로서 그것이 실현한 이식성에 주목할 필요가 있다.

먼저 파이썬을 보자. 파이썬이 대단한 언어는 아니지만, 이미 만들어둔 부품을 얼마나 쉽게 끌어들이는 지에 주목할 필요가 있다. 근 1년 동안 썩 쓸만하다 싶은 라이브러리 앞에 Py-자가 붙어있지 않는 경우를 찾아보기 힘들 정도였다(참된 재사용이란 바로 이런 것이다). 그리고 정말 많은 플랫폼에서 잘 돌아간다. 이런 면에서는 펄과 같은 대부분의 스크립트 언어들이 비슷하다고 할 수 있지만, 프로그래밍 언어로서도 파이썬 만큼이나 표현력이 뛰어난 언어는 드물다. 더욱이 객체지향 프로그래밍도 할 수 있다. 더 좋은 것은 모든 것을 저 혼자 다 하겠다고 욕심부리기보다 이미 잘 만든 것을 그대로 쉽게 갖다 붙일 수 있도록 유연한 구조를 선택했다는 데 있다.

한편, Qt는 C++ 언어로 만든 툴킷이다. 이 툴킷은 잘 만든 소프트웨어일 뿐이지만 C++의 언어적 결함을 잘 메워 마치 자바로 C++를 쓰는 느낌이 들 때가 있다. 첫째는 각 부품마다 자원을 반자동으로 관리한다. 잘만 쓴다면 가비지 컬렉션(Garbage Collection)이 부럽지 않다. 둘째는 형 안전성을 보장하며(signal과 slot을 connect로 연결하는) 위임(Delegation) 방식을 채택해 부품성과 유연성이 뛰어난 이벤트 처리기능을 제공한다. 더 설명할 필요 없이 비주얼 C++의 MFC 같은 다른 C++ 툴킷과 비교해보면, 예로 든 두 기능이 얼마만큼 진보인지 잘 알 수 있다. 마지막으로 여러 플랫폼에서 소스 호환성을 보장한다. 특히 X11과 Win32 API와 같이 크게 다른 두 플랫폼에서 소스 호환성을 보장하면서도 세련된 인터페이스를 갖추고 있다는 점은 높이 살만 하다. 자바와 마찬가지로 Qt는 특정 회사의 상품이다. 그리고 둘 다 호환성을 무기로 시장에 뛰어들었다. 그러나 실제 효과는 사뭇 다르다.

"도구보다 방법을…"

여기서 서로 다른 두 기술을 소개하는 이유는 이 두 기술이 호환성이란 목표를 어떻게 이루어냈는가를 말하고 싶어서다. 즉, 바이너리나 소스 호환성 중에 어떤 게 더 가치가 있냐고 따지는 게 아니라, 각 기술이 이루어낸 '진보'의 자세를 보자는 것이다. 파이썬을 보면 버릴 것(C/C++같은 예전 언어들의 문법이나 기능)은 과감하게 버리거나 고치되, 정말 다시 써야할 것(이미 다른 언어로 잘 만들어 놓은 라이브러리)을 현명하게 챙겼다. Qt는 어렵지만 포기하기 아까운 기술(C++와 그 동안 쌓인 프로그래밍 기법)에서 가장 골치 아픈 허점을 메우고 안정성과 실용성을 제공했으며, 더 높은 수준의 응용 프로그램 소스 호환성을 가능하게 했다.

앞에서 든 두 가지 말고도 비슷한 예는 얼마든지 찾아볼 수 있다. 남은 것은 배우는 사람들과 가르치는 사람들 그리고 쓰는 사람들의 문제다. 특히 가르치는 곳에서 현재의 기술과 이론을 배우는 사람들은 앞으로도 이 분야에서 적어도 10년도 넘게 일할 사람들이다. 그리고 그들이 일할 때는 어떤 기술을 주로 쓰고, 어떤 환경에서 일하게 될 건지 아무도 알 수 없다. 더구나 지금 가르치는 기술에서 껍질에 해당하는 부분은 그 때쯤이면 거의 사라질 가능성이 높다. 그러므로 교육 기관은 이들이 기술에 대한 공평한 시각을 갖출 수 있도록, 다양한 기술을 접할 수 있는 기회를 마련해줘야 한다. 그리고 일을 하는 도구 그 자체가 아닌 '일을 하는 방법'을 스스로 찾아낼 수 있도록 조심스럽게 길을 인도해야 한다.

객체지향의 가치

지나간 프로그래밍 언어의 역사를 다른 관점에서 돌아보자. 스몰토크 때부터 내려오는 '객체지향'은 '상속(inheritance)'이라고 할 수 있을 만큼, 상속이란 언어 기능을 크게 강조했다. 그리고 한동안, 객체지향 프로그래밍 서적에서는 상속으로 얼마나 재사용성이 높아지는 지를 열심히 설명했다. 물론, 상속은 객체지향의 두드러진 특징임에 분명하고, 재사용성의 기본이다. 그러나 그렇게 말하는 게 지금도 과연 옳을까.

헷갈리는 '상속'의 개념

80년대부터 90년대에 쏟아져 나온 언어의 흐름을 보면, 상속을 그저 '상속'해서는 안되는 이유가 있었다. 스몰토크로 대표할 수 있는 고전 객체지향 언어에서는 형(type)이란 것이 별로 하는 일이 없었지만, 객체지향이 다시 살아났을 때는 상황이 달랐다. 현대 프로그래밍 언어의 특징 중 하나는 "가능한 빨리 숨어있는 오류를 많이 솎아내야, 더 튼튼한 코드를 쓸 수 있다"는 믿음을 바탕에 깔고, 어떻게든 형과 '형 검사(type checking)' 기능을 집어넣었다(Applicative(Functional) Programming 언어도 비슷하지만 처리하는 방식이 크게 다르므로 여기서는 빼고 생각하자). 그런데 상속이 형 검사와 만나서 어떤 일이 벌어질 지를 심각하게 생각하지 않았다.

먼저 상속이 뭐길래 그렇게들 좋아하는지 간단히 살펴보자. 예를 들어 클래스 B가 클래스 A를 상속한다고 하자. 클래스 B는 클래스 A에서 '모든 걸' 다 내려받는다. 그래서 클래스 B의 객체는 클래스 A의 객체와 구조도 같고, A의 객체가 받을 수 있는 메시지는 B의 객체도 받아낼 수 있다. 그러므로 클래스 A 객체가 들어갈 수 있는 자리라면, 클래스 B 객체로 바꾸어도 좋다. 이걸 대치(substitution)라 하는데, 이게 객체지향 재사용의 핵심이다. 이른바 상속의 모든 것이다. 즉 나중에 클래스 C를 만들어도 A를 상속하면 마찬가지로 대치가 가능하기 때문에 A로 두고 만든 코드를 그대로 쓸 수 있다.

이 설명에서 알 수 있듯이 상속과 상속의 이점은 꽤 이해하기 쉽다. 그러나 쓰기도 쉽다고 생각하면 곤란하다. 알고 보면 상속이란 그렇게 쉽게 다룰 문제가 아니다. 앞에서 설명한 것처럼 클래스 B가 클래스 A로부터 정말 모든 걸 다 내려받아야 할 때는 문제가 없다. 그러나 이 '상속'에는 서로 성질이 다른 두 기능이 섞여있다. 모든 문제는 여기서부터 시작된다.

클래스 B를 클래스 A의 서브 클래스로 만들어도 된다는 얘기는, B가 A와 뭔가 비슷한 점이 있다는 말이다. 그런데 B가 A로부터 뭘 내려받는지 애매하다. 다시 말해 B의 객체가 A의 객체와 자료구조가 비슷하기 때문에 인스턴스 변수와 코드를 받겠다는 건지, 아니면 A의 객체를 B의 객체로 바꿔치기할 수 있도록 A가 정하는 메시지 규칙을 따르겠다는 건지 알 수 없다.

객체지향 프로그래밍을 공부한 사람이라면, 비슷한 말을 자주 들었을 것이다(이미 눈치챘겠지만). 지금 서브클래싱(implementation inheritance)과 서브타이핑(interface inheritance)의 차이점을 말하고 있다. 상속에는 두 가지 서로 다른 기능이 묶여있고, 이 둘이 그렇게나 다르다는 것을 예전에는 심각하게 다루지 않았다. 그리고 이해할 수 없는 일이긴 하지만 가장 최근에 나왔다는 자바와 C#도 반쯤 문제 있는 상속 기능을 제공한다. 실제로 크게 걱정할 일은 아니지만, 객체지향을 그렇게나 주장하는 언어치고는 자격미달이라 하겠다. 비슷한 얘기를 GoF의 책 'Design Patterns'에서 찾아보자.

"이 두 개념은 헷갈리기 쉽다. 왜냐하면 많은 언어에서 이 둘을 달리 보지 않기 때문이다. C++나 Eiffel 같은 언어에서도 상속은 한꺼번에 인터페이스와 구현을 내려받는 기능이다. ...(중략)... 대부분의 프로그래밍 언어에는 인터페이스 상속과 구현 상속을 나눠 쓸 수 있는 기능이 없지만 실제로는 둘을 구분한다. 스몰토크 프로그래머는 서브클래스와 서브타입이 같은 걸로 본다(그렇지 않은 예가 있다는 게 잘 알려져 있는데도 불구하고). C++ 프로그래머는 형을 추상 클래스로 선언하고, 이것으로 객체를 쓰고 있다."

왜 전에는 이런 얘기를 들어볼 수 없었을까. 적어도 80년대에

는 이런 얘기를 들어보지 못했다. 그 때 본 예제 중 볼랜드 C++에 클래스 라이브러리가 있었다. 필자는 C++를 공부한다고 오랫동안 그 코드를 읽어 내려갔다. 이제 와서 보면, 서브클래스와 서브타입을 구분했다는 기억이 없다. IBM 연구원이 만든 C++ 관련 논문에서 Bag과 Set을 다루며 비슷한 얘기를 하고 있었다는 것이 기억난다.

서브타입과 서브클래스를 구분해야

필자는 그 문제가 '상속'과 '형 검사'가 부딪히면서 관심을 끌게 된 것이라고 본다. 어쨌거나 이브가 사과를 깨물게 된 다음부터는 '객체지향'적으로 사는 게 힘들어졌다. 눈을 뜨고 보니 상속은 정말 어려운 기능이었다. 모르긴 해도 학자들이 하는 얘기를 다 들어주자면, 재사용이고 뭐고 다 포기하고 상속 없이 사는 길을 찾고 싶을 지도 모른다. 그래도 객체지향을 버릴 수 없다면(우리도 모르는 사이에 상속받았을 뿐이다) 이 둘의 차이점은 반드시 알아야 한다. 한번이라도 코드를 제대로 설계해 프로그램을 만들고 싶은 욕심이 있다면, 그리고 모르고 썼을 때 그 피해가 생각보다 크다는 걸 알게 된 다음에는 무시하고 살기가 그리 쉽지 않을 것이다.

먼저, 서브타이핑에 대해서 얘기해보자. 형 검사와 상속은 서로 가는 방향이 반대다. 하나는 잘못을 빨리 막기 위해서 프로그래밍에 엄격한 규칙을 두는 것이고, 다른 하나는 차차 뭔가를 고쳐만들 수 있도록 엄격함을 누그러뜨리자는 것이다. 이 둘을 합치려다 보니 대충 생각했던 상속이 떡 하니 걸린 것이고, 성질이 크게 다른 두 기능을 함께 부릴 수 있도록 적당히 엄격한 규칙을 정의할 필요가 생긴 것이다. 그러자면 먼저 객체지향이 주장하는 상속의 목적을 살필 수밖에 없다. 아는 것처럼 상속의 목적은 재사용이다. 그리고 재사용은 대치성으로부터 온다. 형 검사와 대치성을 붙여놓으면 이 객체가 저 객체를 대치할 자격이 있는 지 미리 검사할 수 있는 장점이 있다. 이때 이 객체와 저 객체의 인터페이스(형) 사이의 관계, 이것이 서브타입 관계다.

상속에서 서브타이핑을 떼놓고 보면 서브클래싱만 남는다. 서브클래싱이 하는 일은 너무 단순하다. 그저 새로운 클래스를 만들 때 구조가 비슷한 클래스가 어디에 있다면, 깡그리 새로 만드는 것이 너무 번거롭기 때문에 그 클래스로부터 코드를 빌려와서 간편하게 만들어보자는 것이다. 우선 눈에 띄는 효과는 키보드를 덜 치게 해주기 때문에 프로그래머의 건강을 유지할 수 있다는 정도인데(그러고 보니 아주 중요하다), 정말 그것뿐이라면 마우스를 쓰는 게 낫다. 다시 말해, 언어를 복잡하게 만들면서까지 지켜할 유산은 아니라고 할 수 있다. 그러나 객체지향 언어라면 대부분 이 기능을 지원하고 있고, 중요한 기능이라고 열심히 선전까지 한다. 정말 그렇게 한심하기만 한 기능일까. 물론 서브클래싱만으로는 그렇게 쓸모 없다. 서브타이핑이 따라가 줘야 빛이 난다. 특히 서브클래스가 동시에 서브타입이라면(그런 경우가 더 많기 때문에) 아주 매력있는 기능이다. 그 때문에 여전히 자바와 C#은 물론이고 많은 프로그래밍 언어가 서브클래싱을 물려받은 대로 쓰고 있는 것이다. 예를 들어 자바의 extends와 C#/C++의 public derivation은 서브클래싱이자 동시에 서브타이핑이다. 일부에서는 이 두 기능을 완전히 떼어놓자고 주장하지만 필자는 그 반대편에 손을 들어주고 싶다. 그러나 반드시 서브클래싱만을 표현할 수 있는 기능이 있다는 가정에서다(물론 자바나 C#에는 그런 기능이 없다).

사실, 이론으로만 따지면 상속에서 서브클래싱은 빠져도 된다. 합성(Composition)으로 완전히 대체할 수 있기 때문이다. 합성(Composition, 더 정확하게 말하자면 delegation)이 상속이란 표현을 완전히 대치할 수 있다는 건 오래 전에 입증된 바 있다. 그러나 합성도 좋지만 남용하면 상속을 잘못 쓴 코드보다도 나을 게 없다. 정말 구현을 빌어 쓰고 싶을 때마다 합성만 쓰라고 강요한다면 자바에는 implements만 있으면 된다. 그러나 실제로는 쓰임새에 따라 섞어서 사용하는 게 옳다. 특히 서브타입 계보에 따라 서브클래스를 써도 되는 경우에, 추상 클래스(partial implementation)는 무척 쓸모 있는 기능이다.

이제 다시 두 기능을 상속 하나로 묶어서 크게 생각해보자. 실제 서브타이핑은 상속이라기보다 '약속을 지키겠다는 선언'에 더 가깝다. 이것을 인터페이스 '상속'이라고 표현하는 것이 더 어색하다. 서브타이핑이 필요한 이유는 대치 가능성을 검사하기 위해, B가 A형을 완전히 만족하는지 보는 것일 뿐이다. 이런 개념은 우리가 지금까지 직관으로 생각해온 '상속'과는 거리가 멀다. 그리고 이런 기능을 꼭 지금의 자바나 C#처럼 처리해야 할 이유는 더더욱 없다.

많은 학자들이 주장하듯이 객체지향 언어의 재사용성은 서브타입 관계로 표현하는 대치성에 있다. 이를 바탕으로, 이른바 종교와도 같은 '점진 비파괴 확장(incremental nondestructive extention)'이 가능하다. 이 문제를 이해할 만한 경험이나 지식이 없다고 하더라도 객체지향 언어에서 형 하나를 만든다는 게 쉬운 일이 아니라는 것쯤은 짐작할 수 있을 것이다. 그냥 한 클래스를 만드는 거야 쉬울 수도 있겠지만, 객체지향 언어에는 마치 족보와도 같은 커다란 '형 계보(type hierarchy)'가 있다. 새로운 클래스를 만들 때마다, 그게 장난삼아 한번 만들어 쓰고 버릴 게 아니라면 전체 서브타입 계보(subtype hierarchy)에서 가장 알맞은 자리를 찾아내야 하고, 새로 만든 클래스의 형은 이미 있는 것들과 다른 정확한 역할(role)이 있어야 하는데, 그걸 정의해 내기란 그리 쉬운 것이 아니다. 그러나 이렇게 하지 않으면 재사용성이란 그 '성스러운' 목적을 이룰 수 없다. 물론 이런 목적을 나 몰라라 할 수 있다.

그러나 그렇지 않다면 무엇 때문에 이런 어렵고 느린 객체지향 언어를 써야만 하는가. 이유가 있다면 자바말고는 다른 해결책이 없다고 믿는 엔지니어로 가득찬 회사에서 '일자리'를 잃지 않기 위해서거나, 자바나 닷넷으로 만들지 않으면 소프트웨어라고 봐주지 않는 이상한 고객을 속이기(?) 위해서 필요하다.

이쯤 됐으면 우리는 객체지향 프로그래밍을 한다는 것이 그냥 코드를 마구 써대는 게 아니라, 서브타입 관계를 정의하고 그 관계로부터 오는 유연함을 '꼭' 이용하는 복잡한 일이라는 것을 의심하지 말아야 하겠다. 실제로 한 클래스의 형을 잡는 일은, 쓰다 버릴 클래스를 만드는 게 아니라면 따로 잘라내서 하는 일이 아니기 때문에 전체 프로그램을 설계해 틀을 잡는 일이나 다름없다. 그리고 그 일을 아직 제대로 하지 못한다면, 객체지향을 안다고 해서는 안된다(물론 필자도 안다고 할 수 없다). 자바나 C#으로 코딩한다고 그게 객체지향 프로그래밍을 한다고 볼 수 없을 뿐더러, '객체지향'을 머리 아프다고 쓰지 않겠다면 자바나 C#을 쓸 일은 더더욱 없다.

객체지향을 착각하는 경우가 많다

안타깝게도 일부 프로그래머 중에는 오랫동안 자바를 썼다는 이유만으로 자기가 객체지향을 잘 쓰고 있다고 착각하는 경우가 많다. 그런 예를 하나 들어보겠다. 한 2년 전에 나는 후배들과 자바 프로그래밍 서적을 쓴 적이 있다. 필자가 맡은 부분은 자바의 쓰레드 프로그래밍 부분이었는데, 글을 쓰는 데 바빠서 다른 사람의 글을 훑어볼 틈이 없었다(이 책의 저자로 올라가지 않는 것이 얼마나 다행스러운 일인가). 나중에 별 생각없이 이 책으로 강의를 해나가다가, 정말 어처구니없는 코드를 발견하게 됐다. 그 사람은 꽤 자바 프로그래밍을 잘 하는 사람이라고 들었고, 자바에 대한 애정이 대단한 사람이었다. 그 코드를 여기 전부 옮겨 쓸 수는 없지만, 그 틀만 짐작할 수 있도록 보여주겠다. 무슨 스프레드시트를 자바로 만드는 코드였는데, 스프레드시트를 구성하는 각 셀에 집어넣을 데이터를 다음과 같이 표현했다

```
class CellData {
   private int intVal;
   private float floatVal;
   private String text;
   private int bType;
   ...
   public int getType() { return bType; }
   public void setValue(int ival) {
   bType = CellData.IntVal;
   intVal = ival;
   }

   public void setValue(float fval) {
   bType = CellData.FloatVal;
   intVal = fval;
   }

   public void setValue(String someData) {
   if (isformula(someData)) // 아미 문자열 내에 무슨 특수 표시가 있으면
                            // 식(formula)으로 치나 보다.
      {
      bType = CellData.FloatVal;
      float = calc(someData);
      }
      bType = CellData.Text;
      text = someData;
       }
      ...
    // 새로운 타입이 추가되면, 이 클래스 전체를 뜯어 고쳐야 한다.
} // 실제 코드는 옮긴 것보다 훨씬 화려(?)하다.
```

뭐가 잘못됐는지 보자. 스프레드시트는 구조로 볼 때, 이형 컬렉션(heterogenious collection)의 전형이라 할 수 있다. 각 셀에는 갖가지 데이터인 식, 변수, 그림 등을 집어넣을 수 있어야 한다. 그리고 식 하나만 보더라도, 처리 기능을 점차 키우려면 함수를 더할 수도 있고 새로 연산자를 더할 수도 있기 때문에 길게 보면 하나의 계산용 프로그래밍 언어를 만드는 거나 마찬가지다. 이렇게 계속 바뀌는 자료를 표현할 때 이미 있던 코드를 바꾸지 않고도 서브타이핑 다형성을 이용해 멋있게 처리할 수 있는 것이 객체지향 언어다. 그런데 앞의 코드는 자바로 썼다 뿐이지 영락없는 C 코드다. 특히 C에서 악명높은 Union을 그대로 자바로 답습했다. 만약 저 CellData를 각 데이터의 형에 따라 다르게 처리하려면 모든 코드는 다음과 같은 구조가 될 수밖에 없다.

```
public static void processData(CellData cellData) throws CellData {
  if (cellData.getType() == CellData.IntVal) {
  ...
  }
  if (cellData.getType() == CellData.floatVal) {
  ...
  }
  if (cellData.getType() == CellData.Text) {
  ...
  }
  else // 새로운 데이터가 추가된다면 이 코드를 고치지 않을 수 없다.
     throw new UnknownCellData(cellData);
  ...
} // 물론 switch 문을 쓰면 더 간단하다. 그러나 그렇다고 구조가 좋아지지는 것
  // 은 아니다.
```

간단히 말하자면 앞의 코드는 왜 객체지향 프로그래밍이 좋은지 설명할 때, 딱 그에 반하는 예제로 들면 좋은 예다. 즉 적절하지 못한 객체지향 코드의 전형을 보여준다(이게 저자의 의도였다면 할 말이 없다). 예를 들어 CellData에서 식이 더 복잡해졌다든가, 그림이 들어갈 수 있다든가, 하여간 데이터가 확장되면 앞의 두 코드를 뜯어고치지 않고 처리할 방법이 없다. 사실 객체지향 언어가 나온 이유가 저런 코드를 없애자는 것이고, 서브타입 관계와 오버라이딩(overriding)이 앞에서 보는 군더더기 코드를 (제어) 추상화하는 게 목적인데, 무슨 생각으로 저런 코드를 버젓이 책에다 올렸는지 이해할 수가 없다. 그런데 나중에 혹시나 하고 다른 책을 뒤져보니, 국내외 서적에서도 저런 예제가 꽤 많이 있었다.

물론 누구나 실수할 수 있다. 그래서 필자는 저런 코드가 나온 이유를 좀 달리 본다. 수년씩 자바를 써서 교재까지 쓰는 사람이 저런 코드를 무심코 쓰는 걸 보면, 객체지향이 정말 쉽지 않아서 그런 것이 아닐까 싶다. 배운 것이 아깝겠지만 어렵다면 꼭 쓰지 않아도 된다. 객체지향이 아닌 언어도 많고, 꼭 객체지향을 써야 재사용성이 뛰어난 프로그램을 쓸 수 있는 것도 아니다. 그리고 객체지향이 우리를 이렇게나 고생시키는 것이라면, 무슨 지켜내야 할 유산이 아니다. 분명히 말하지만 스몰토크 시절부터 이어왔던 환상은 이미 깨졌다. 그리고 지금 우리가 얘기하는 객체지향은 예전의 모습이 아니다.

'꼭 객체지향 언어를 쓸 필요는 없다'

앞에서도 말했지만, 객체지향의 상속이 주는 두 가지 기능, 즉 서브타이핑과 서브클래싱 중에 꼭 상속으로 표현해야 하는 것은 없다. 서브클래싱은 그저 구현 기법의 하나일 뿐이고, 불편하기는 하겠지만 객체지향에서 없애도 되는 기능이다. 서브타이핑은 객체지향에서 꼭 필요한 기능이지만, 대치성을 표현하려고 꼭 상속이란 기능을 써야 하는 것도 아니며 그렇게 표현할 이유도 없다. 그렇게 따지면 객체지향을 부르짖는 것이 얼마나 허망한 일인가. 그리고 서브타이핑이 그렇게 쓰기 힘들고 많은 경험이 필요하다면 왜 더 쉬운 언어를 두고 꼭 객체지향을 고집해야만 하는가.

물론 객체지향 프로그래밍을 내다 버리라는 소리가 아니다. 다만 그 가치가 어디에 있는지 제대로 알아야 하고, 그 가치를 얻어내는 대가가 그리 만만하지 않다는 말이다. 앞의 예에서 봤듯이 수년 경험의 객체지향 프로그래머도 초보자나 하는 실수를 한다. 그러므로 지레 겁을 먹고 물러설 필요도 없지만, 한번에 다 알겠다고 덤벼서도 안될 일이다. 좋은 프로그래밍 기법은 그렇게 몇 년만에 쉽게 익힐 수 있는 게 아니다. 다시 말해 '20일' 만에 완성되고 '30일' 만에 완전 정복할 수 있는 것이 아니다. 그런데도 객체지향 연구가 계속되고, 설계 패턴(Design Pattern)이니 하는 것들이 꾸준히 나오는 이유가 뭘까. 그냥 어디서 연구비를 받아내려고 일부러 그러는 것만은 아닐 것이다. 객체지향은 몇 줄 코드에 그 가치가 숨어 있는 게 아니다. 객체지향의 가치는 앞에서 보여준 예와 같이 프로그램의 구조를 잡아내는 데 있다. 즉 각 클래스(클래스는 구현을 포함하는 말이니 거기서 형만을 끄집어내야겠다) 형의 문법과 의미를 정의하는 과정에서 각 모듈이 해야 할 역할과 그 역할이 어떤 낱낱의 기능으로 구성되는지 잡아내는 데 있다. 그리고 비슷해 보이는 형(type)을 모아두고, 서브타입 관계를 뽑아내는 과정에서 전체 모듈의 재사용성을 끌어올리는 데 필요한 판단 자료를 얻을 수 있다. 그런 분석과 설계가 끝난 다음에는 아무 언어나 잘 써서 코드를 열심히 쓰면 그 뿐이며, 꼭 객체지향 프로그래밍 언어를 쓸 필요도 없다(물론 객체지향 언어가 더 편할 때가 가끔 있다). 꼭 자바나 C#을 쓴다고 특별히 더 나아지는 것은 없다.

더 잘 풀 수 있다면 뭐든지 배워서 써라

앞에서부터 지금까지 계속 두 언어를 비판하다보니, 두 언어를 배우고 싶은 마음이 갑자기 사라져버린 사람도 있을 지 모르겠다. 그러나 두 개발 환경을 미워하고 쓰지 말라는 데 있지 않음을 생각하기 바란다. 단지 잘 알고 제대로 쓰자고 동의를 구하는 것 뿐이다. 그런 의지를 실천에 옮기려면, 프로그래밍 '언어'에 집착하지 말고 '프로그래밍'에 더 마음을 쏟아야 한다. 그리고 천천히, 오랫동안, 하나씩, 필요에 따라 배워나가는 게 좋다.

- ◆ 자바는 수많은 언어(Lisp, Simula67, CLU, 스몰토크 등)의 계보를 이어가는 프로그래밍 언어다.
- ◆ 자바는 C나 C++에서 문법을 빌어왔기 때문에 겉으로는 닮았지만 깊이 들여다보면 그 언어들과 매우 다르다.

앞 구절을 인용한 까닭은 프로그래밍 언어를 겉으로만 봐서는 잘 알 수 없다는 말을 하고 싶어서다. 특히 자바나 C# 같이 더 좋은 프로그래밍 언어를 만들겠다는 것 이외에, 여러 다른 목적(사업성, 플랫폼 의존성(?) 등)을 한꺼번에 이루려는 야심을 갖고 있다면, 더더욱 그럴 수밖에 없다. 하물며 고상한 목적을 위해서 만들기 시작한 언어도 얼마든지 잘못된 길로 갈 수 있다.

자바에는 수십 개의 언어가 하나로 뭉쳐있다. 프로그래밍 언어 표현력이 그렇다기보다 전체 환경이 그렇다. 자바를 쓰는 사람은 JDK를 써야 뭐라도 만들 수 있다. 세상 어디에도 프로그래밍 언어 그 자체만으로 프로그래밍을 하지도 않고 할 수도 없다(너무 당연한 얘기다). 그래서 보통 표준 라이브러리는 그 언어의 일부며, 대부분 언어의 프로그래밍 도구로서의 결함을 메우거나, 운영체제의 기본 입출력 자원을 가리기 위한 목적으로 언어와 함께 붙어 다닌다. 특히 자바와 같이 갇힌 환경에서는 잘 쓰던 라이브러리가 있더라도, 그걸 버리고 JDK에 있는 걸 쓰는 게 속 편하다. 정말 JDK에는 없는 것이 없고 점점 덩치가 커진다.

자바 그 자체를 배우는 것은 JDK를 배우는 것에 비하면 그래도 작은 문제다. 공룡같은 JDK를 어떤 식으로든 배워야 한다는 건 골칫거리다. 새로 익혀야 할 사람이라면 덜 억울하겠지만, 지금까지 잘 써오던 걸 내다 버리고 같은 기능을 하는 JDK의 일부를 다시 익혀야 한다면 참 갑갑할 것이다. 어떤 툴킷(또는 라이브러리)도 마찬가지지만, 어떤 응용 분야에 해당하는 라이브러리의 일부는 새로운 언어를 하나 더 배우는 것과 같다.

단지 같은 언어를 쓴다고, 즉 문법이 같다고 쉽게 생각하면 곤란하다. 예를 들어 네트워크 프로그래밍을 하려면 네트워크에 대한 기본 지식을 알아야 하는 것은 물론이거니와 네트워크 라이브러리를 설계한 사람이 배려한 구성요소와 그 구성요소 간의 결합 방법, 효과를 잘 이해해야만 한다. 그 다음에 라이브러리를 잘 써야만 위험을 피할 수 있고, 문제가 발생할 때마다 어떤 식으로 해결해야 하는 지 그 관례와 패턴을 익혀야 한다. 그렇게 보면, 라이브러리의 한 응용 분야를 배운다는 것은 정말 '특수 기능 언어'를 하나 더 배우는 것과 다를 바가 없다. JDK 전체로 보자면 자바로도 모자라서 수많은 언어를 배워야 하는 것과 같은 부담을 안게 된다.

뭘 만들고 싶은지를 생각하라

먼저 충고부터 하자면 그런 특수 목적 언어를 필요도 없이 미리 익히려고 애쓰거나, 여러 개를 한꺼번에 다 알아야 한다고 욕심을 부리는 것은 정말 어리석은 짓이라는 걸 말해주고 싶다. 그저 알아야 한다는 생각으로 무작정 연습해봐야 아무런 효과도 없다. 차라리 그럴 시간에 '뭘 만들고 싶은 지' 정확한 목표를 정하는 게 낫다. 그리고 그 목적을 재밌게, 그리고 제대로 이루어 냈다면, 그 언어에서 뭘 더 배우려고 할 필요가 없다. 어차피 시간이 지나면 조금 씩 나아지게 마련이고, 목적이 더 이루기 어려운 것일 수록 더 많은 기능을 알게 되어있다.

언어를 배우는 사람들이 한 번씩 그 목적을 잃어버리고 귀중한 시간을 낭비하고 있는 걸 보면 안따까운 생각이 든다. 우리가 프

로그래밍을 배우는 이유가 무엇인가. 당연히 프로그램을 '더' 잘 짜기 위해서다. 그런데 당장 프로그램을 짜는 데 쓰지도 않을 기능으로 미리 머리를 싸맬 이유가 뭔가. 언어의 기능을 화려하게 쓰고 라이브러리 구석에 틀어박혀 있는 기능을 쓰지 않는다고 문제가 될 것이 없다. 그런 것 말고도 배우고 익혀야 할 것은 널려있다. 더구나 그런 것들은 얼마 안가서 바뀔 수도 있고, 아예 사라질 수도 있으며, 그 기술만으로는 좋은 직업을 가질 수 없을지도 모른다. 말하자면 JDK나 닷넷 프레임워크와 같은 방대한 라이브러리는 필요할 때 쓰라고 만든 편리를 위한 도구일 뿐이다. 다시 말하지만 한 회사의 상품이며, 모모 전자의 모모 압력밥솥과 같은 것이다. 더구나 그래픽스, 네트워크, 데이터베이스, 분산처리, 문서처리 등 수 많은 분야의 응용 기술을 하나로 붙여놓은 것을 어떻게 다 감당하려고 하는가. 평생 그 모든 걸 다 알고 쓸 수는 없다. 외운 걸 다 적어서 끌어모아 봐야 CD-ROM 하나라도 가득 채울 수 있을까.

이런 사실을 모르는 게 아닐텐데 많은 교육기관에서 같은 실수를 반복하고 있다. 그 아까운 시간에 그저 읽어 내려가도 될 책을 대신 읽어주느라 바쁘고, 배우는 사람이 스스로 풀어보면 될 예제를 대신 풀어준다고 공연한 애를 쓴다. 요소 기술을 열거하고, 그 사용법을 밤새도록 연습한다고 프로그램을 잘 짜는 게 아니다. 심하게 표현하면 그건 그저 단순 노동을 아까운 돈 들여 연습하는 것이다. 왜 십여 년이 넘도록 영어를 공부하고도 말 한마디 못해 문제라고는 하면서 프로그래밍 교육도 같은 방법을 쓰는지 이해할 수 없다.

프로그래밍은 어렵다

정작 가르치고 배워야 할 것은 따로 있다. 처음 프로그래밍을 배우는 사람이라면 가능한 많은 분야 문제를 꾸준히 풀어봐야 한다. 그리고 꼭 지켜줘야 할 것을 말하면, 절대 '프로그래밍이 쉽다'고 거짓말을 하면 안된다는 것이다. 알다시피 프로그래밍은 어렵다. 어려운 이유는 이미 충분히 설명했다. 그러나 그 어렵다는 것 때문에 재미있고, 바로 그 '진지한 재미'를 놓치지 않으면 될 일이다.

그리고 C/C++, 자바, C# 등 문법이 복잡하고 기능이 까다로운 언어를 처음 배우는 사람에게 권할 때는 한 번쯤 깊이 생각해줬으면 좋겠다. 그런 언어를 쓰면 언어를 배우다가 지쳐서 많은 문제를 풀어야 할 기회를 영영 놓쳐버릴 수도 있다. 그리고 그런 언어로 프로그래밍을 잘 가르치기란 그리 쉽지 않다. 한 가지 잘못된 프로그래밍 교육의 사례를 들어보겠다. 몇 년 전에 부탁을 받아 1개월간 자바 강의를 하러 간 적이 있다. 미리 학생들의 수준을 알아야 해서 뭘 배웠는지를 물어보니, C++ · MFC를 4개월간 배웠다고 한다. 하루에 10시간 가까이 교육을 받는 프로그램이라 4개월이면 무시 못할 시간이다. 그 정도면 아주 잘하지는 못해도 기본은 되어있을 거라 생각해 빠르게 앞부분을 훑어갔다. 그런데 점점 시간이 지날수록 이해하지 못할 일이 일어났다.

일주 정도 수업한 후 간단한 클래스를 스스로 만들어 보라고 했는데, 메소드 어디에 코드를 두고 멤버는 어디에 두는 지조차 모르며, 생성자니 선택자(selector, query)니 하는 기본 구성요소도 알지 못하는 것이다(자신들은 잘 알고 있다고 생각해서 더 문제였다. 이러다가 책까지 쓰게 되는 것이다). 이상한 생각이 들어 다음과 같은 예제를 내고 그 결과를 물었다.

```
struct Rabbit {
   ...
   virtual void sayCry() { cout << "..." << endl; }
}

struct CrazyRabbit : Rabbit {
   ...
   void sayCry() { cout << "Moo" << endl; }
}

void foo(Rabbit r, Rabbit* pr, Rabbit& rr) {
   r.sayCry();
   pr->sayCry();
   rr->sayCry();
}
...
Rabbit *pr = new CrazyRabbit();
foo(*pr, pr, *pr);
```

아주 간단한 C++ 코드지만 학생 중에 제대로 답을 하는 사람은 딱 한 명 있었다. 더 놀라운 사실은 4개월 교육의 결과로 각 팀마다 그럴싸한 메신저를 만들어냈다는 것이다. 그 메신저는 간단한 결재 기능도 들어있고 보기도 좋아서 꼭 여느 회사에서 만든 상품 같았다. 그런데 그런 작품을 만들어낸 사람들이 어떻게 저런 단순한 코드를 모를 수 있을까. 이유는 각자의 상상에 맡긴다.

필자는 깔끔하고 표현력이 뛰어난 언어를 권하는 편이다. 그러다 보니 반발도 심하다. 보통 그 이유는 의외로 간단하다. 유행을 따라가지 않아서거나, 객체지향 프로그래밍 언어가 아니라서거나, 자바나 C#, C++가 아니라서 그렇다. 이런 이유는 그래도 참을 만하다. 더 말이 안되는 이유는 필자가 권하는 언어가 더 어렵다고 할 때다. 정말일까? 긴 예제를 들 필요도 없이 간단히 코드를 보고 얘기를 계속하자.

```
odds = [1,3..]
```

필자가 권하는 언어에서 모든 홀수를 끝없이 찍어내려면 이와 같이 표현하면 된다. 다음에 그 중에서 10개만 가져다 쓰자.

```
take 10 odds
```

어려운 것은 언어가 아니라 '문제'

앞의 예는 자주 권하는 언어 중에 하나를 간단히 써본 것이다. 아무리 봐도 어려워 보이지 않는다. 그보다는 같은 문제를 '꼭 가르쳐야 하는 언어'를 써서 풀었을 때, 처음 배우는 사람이 얼마나 공부해야 할 지 스스로 생각해 보는 게 좋겠다.

이제 필자는 이런 언어를 왜 어렵다고 하는지 정확하게 이유를 안다. 사실 언어가 어려워서가 아니었다. 가령 저런 언어를 가르칠 때 욕심을 부리지 않고, 자바나 C# 수준만큼만 잘라내서 가르치면 언어만 배우는 데 이틀이면 충분하다. 그렇기 때문에, 저런 언어는 사흘쯤 됐을 때부터 언어 그 자체가 아니라 문제를 푸는 연습을 하게 된다(언어 기능을 가르치려고 억지로 만들어낸 문제는 아니다). 일주일이 지난 후에 푸는 문제는 자바나 C#을 몇 개월 배우고도 어려운 수준이다. 그러므로 어려운 것은 언어가 아니라 문제다. 그런데 어떤 수준의 문제를 푸는지 알지 못하고, 언어가 어려워서라고 한다. 그런 문제는 어떤 언어로 풀어도 어렵다. 그리고 당연한 얘기지만 어려운 문제를 쉽게 만드는 방법은 없다. 쉽게 풀 방법이 있다면 어려운 문제가 아니다. 다만 어려운 문제를 잘 풀 수 있는 원칙은 있다. 그리고 그 원칙을 바탕으로 끊임없이 문제를 즐기는 게, 프로그래밍을 공부하는 최고의 방법이라 할 수 있다.

'실천'이 중요하다

지금까지 두서없이 프로그래밍에 대한 이런 저런 얘기를 쏟아냈다. 자바나 C#을 평가절하하고, 객체지향이 쓸모 없다고 하는 얘기로 들릴까봐 걱정도 된다. 자바나 C#은 단지 예로 들기 좋아서 끌어들인 것일 뿐이다(그리고 필자가 가장 많이 쓰고 있는 언어이기도 하다). 이 글에서 프로그래밍에 대한 기술을 더 깊이 다루지 못한 것이 아쉽다. 실은 자바나 C#의 기능 중에 놓쳐버린 C++의 좋은 기능, 예를 들면 서브클래싱만 따로 표현할 수 있는 private 상속 같은 기능을 얘기하고 싶었고, C++ 템플릿이나 GJ에 대한 얘기를 깊이 다루고 싶었다. 또 Anonymous Object로 여러 기법을 섞어 쓰는 장점과 그런 기술의 흐름에 대해서도 얘기하고 싶었다. 그러나 각 주제마다 엄청난 분량이라 어찌 줄여 써야 할 지 몰라 마음을 접었으니 이해해주기 바란다.

우리나라에선 10년 넘게 프로그래밍을 하는 사람이 드물다. 소프트웨어 기술이 차곡차곡 쌓일 수 없는 환경이라는 말이다. 그리고 왜 이렇게 됐는지는 우리 스스로가 이미 잘 알고 있다. 다만 실천이 문제다. MS

정리 : 강경수 elegy@sbmedia.co.kr

참고 자료

❶ "An Interview with A. Stepanov," by Graziano Lo Russo, Edizioni Infomedia srl, http://www.stlport.org/resources/StepanovUSA.html
❷ "Deep Inside C#: An Interview with Microsoft Chief Architect Anders Hejlsberg," by John Osborn, O' Reilly & Associates, Inc., http://windows.oreilly.com/news/hejlsberg_0800.html
❸ "Design Patterns : Elements of Reusable Object-Oriented Software," Erich Gamma, et. al, Addison Wesley, 1995
❹ "Program Developement in Java : Abstraction, Specification, and Object-oriented Design," Barbara Liskov, with John Guttag, Addison Wesley, 2001
❺ "A COMPARISON OF MICROSOFT' S C# PROGRAMMING LANGUAGE TO SUN MICROSYSTEMS' JAVA PROGRAMMING LANGUAGE," By Dare Obasanjo, 2001.
❻ Bjarne Stroustrup' s FAQ, http://www.research.att.com/~bs/bs_faq.html