www.kandroid.org

| 벚꽃 4월, 칸드로이드 저널

Beyond Android

Since 2008

www.kandroid.org

20100419. No.3

Contents

칸드로이드 저널 인사말 남들이 가지 않은 뒤안길에 꽃밭이 있다..

김 은 영

옛날 옛적, 스티브 잡스는 매킨토시로 세상을 놀라게 했었다.
하지만 '개방'을 내세운 MS의 도스와 윈도우에 뒷덜미를 잡힌 쓰디쓴 경험이 있다. 그래서 스티브 잡스는 '개방'을 싫어한다. 그런데 한때 파트너이자 동갑내기 친구인 에릭 슈미츠가 구글로 가더니 불길하게도 '개방형' OS인 안드로이드를 만든 것이다.
그래서 잡스는 더더욱 불길.불길.불길한 것이다.

그러던 지난 3월, 어느.. 날 좋은 봄날, 실리콘밸리의 한 모퉁이 길다방에서 애플의 스티브 잡스와 구글의 에릭 슈미츠가 '우연히' 만났다. 일각에서는 비서들의 계략으로 각각 서로 다른 사람(잡스는 보노 / 슈미츠는 페이스북의 거물)을 만나는 줄 알고 나왔다고 하지만, 가장 설득력 있는 설은 '이상한 클라우드 기반 세계에서 바람결에 구름결에 흘러 들어온 것'이 맞을 듯 하다. Anyway, 잡스와 슈미츠는 무슨 이야기를 나눴을까.
'우리 둘이 세상을 나눠 먹을까?!' 그랬을까.
그도 그럴 것이 넥서스원의 부진이 스티브 잡스의 마음을 조금은 누그러뜨렸다는 설도 있고, 애플 아이패드의 판로개척 어려움이 슈미츠를 위로했다는 설도 있다. 그리고 두 사람 사이에 모종의 합의가 있었다는 추측도 발빠르게 퍼지고 있다. 과연 어떤 합의? 누가 불을 지르고, 누가 불을 끌 지 순서를 정하는? 아무튼 세계가 지켜보는 스마트폰 전쟁이 지나치게 과열양상을 띄는 동시에 지나치게 배타적인 정책을 펼치고 있어서 소비자가 피해를 보지 않을까 하는 분석도 나오고 있다. 하지만 어쩌랴. 소비자는 이미, 언제나 굿이나 보고 떡이나 먹는 것에 익숙하지 않은가.
우리의 포커스는 언제나 이 모든 고래들 사이에 끼어있는 개발자들이다.

> [작은 차별화의 기반은 기술인 경우가 많다. 제품, 가격, 서비스 등등 기술이 필요하다. 하지만 작은 차별화는 곧 다들 따라 한다. 대신 큰 차별화는 개념과 기술 모두 필요하다.]

애플과 구글의 스마트폰 경쟁은 바로 이런 큰 차별화에 기준을 둔 개념 있는 전쟁이기에 더 매력적이다. '다름'은 '매력'을 만들고, 매력은 '시장'을 접수하기 때문이다. 개발자도 마찬가지다. 개념을 탑재한 개발자, 철학을 탑재한 개발자, 멋지지 않은가.
남들이 가지 않는 뒤안길에 꽃밭이 있다.
이마티 히로유키 [인간, 시장, 전략, 결단과 경영의 다이내믹스] 중에서 빌린 글이다.

자유 소프트웨어와 오픈소스 소프트웨어의 정의

윤 종 민

물질을 소유하고, 그대 자신을 위해 그 물질의 소유자로 여기되,
필요로 하는 자가 있거든 쓰게 하라. 이는 자비가 아니라 의무이니라.
-- 움베르토 에코, "장비의 이름" 중에서, 1980년

Open Source Software(이하 OSS)의 정의는 소스코드가 공개되어 있으며, 일반적으로 자유롭게 사용,복제,배포,수정할 수 있는 소프트웨어를 이야기 합니다. OSS의 대표적인 예로는 Linux 커널을 비롯한 GPLed Software, 아파치 웹 서버, Firefox등이 있으며, 지금도 전 세계의 수 많은 개발자에 의해 개발되고 있다.

일반적으로 OSS는 FSF의 자유 소프트웨어 (Free Software)를 포함한 넓은 의미로 사용되고 있지만, 자유 소프트웨어와 OSS는 역사 및 추구하는 방향 등에서 미묘한 차이가 있습니다.

이 글을 통해서, 자유 소프트웨어와 OSS의 차이를 설명하고, 관련된 소프트웨의 구분에 대한 이야기를 하고자 합니다.

자유 소프트웨어의 개념은 1980년대 중반 리차드 스톨만(Richard M. Stallman) 및 그가 세운 자유 소프트웨어 재단(Free Software Foundation)과 함께 생겨났습니다. 리차드 스톨만이 MIT에서 직업적인 연구활동을 시작했던 1971년에 그는 자유 소프트웨어만을 사용하는 연구 그룹에서 일하게 되었는데, 그 시절은 상업적인 컴퓨터 회사들 조차도 자유 소프트웨어를 배포하던 때였으므로 프로그래머들은 아무런 제약 없이 서로 협력할 수 있었습니다. 그러나 1980년대에 이르러 거의 모든 소프트웨어들은 소유와 독점에 관한 법률에 의해서 제한되었으며, 소유권자들은 소프트웨어의 자유로운 이용을 통한 사용자들의 상호 협력을 그들의 권리를 내세워서 금지시켰습니다. 그래서 스톨만은 소프트웨어가 컴퓨터 하드웨어로부터 떨어져 나오면서 상업적인 제품으로 팔리기 시작한 것은 지극히 잘못된 일이라고 주장했습니다. 그는 모든 소프트웨어는 무제한 공유되고 무료로 배포돼야 한다는 주장과 함께, MIT를 떠나 FSF를 설립하고, 상업용 소프트웨어와의 전쟁을 선포합니다. 바로 이것이 GNU 프로젝트가 시작된 이유였습니다.

GNU 프로젝트는 초기의 컴퓨터 공동체 안에 충만해 있던 호의적인 상호 협력의 정신을 재건하기 위한 구체적인 실현 방법으로 1983년에 시작되었으며, 이는 독점 소프트웨어의 소유자들이 만든 장벽들을 제거함으로써 상호 협력의 풍토를 다시 부활시키는 것을 그 목적으로 합니다.

그럼 다시 자유 소프트웨어의 이야기로 돌아가 보도록 하겠습니다. 우리가 소프트웨어를 사용하는데 있어서 돈을 내지 않아도 된다는 것은 전혀 자유로운 것이 아닙니다. 우리가 가지고 있는 소프트웨어를 다른 사람에게

배포하거나 새로운 기능을 추가하고자 할 때, 이를 금지당할 수도 있습니다. 무료로 배포되는 대부분의 소프트웨어들을 보면, 관련 제품을 판촉하거나, 사업상의 전략으로 경쟁자를 시장에서 몰아내거나, 영업을 방해하기 위한 한 수단으로 사용되는 경우가 많습니다. 앞서 스톨만의 경우와 같이, 이렇게 배포된 소프트웨어가 영구적으로 무료 소프트웨어로 남아 있을 것이라고는 그 누구도 장담하지 못합니다.

진정한 자유 소프트웨어라면, 항상 자유로워야 합니다. 일반적으로 배포된 자유 소프트웨어는 비(□) 자유 소프트웨어로 변경될 수도 있습니다. 그렇게 되면, 기존의 자유 소프트웨어상태일 때, 개선되었던 많은 부분과 많은 가능성 들이 사라질 수도 있습니다. 자유 소프트웨어가 진정한 자유 소프트웨어로 자유롭게 남아 있기 위해서는 소프트웨어는 저작권(copyright)과 라이센스가 반드시 명시되어야 합니다.

좀 더 간단하게 이야기 하면, 소프트웨어는 자유로울 수도 있고, 그렇지 않을 수도 있습니다. 하지만, 실제로는 이렇게 쉽게 구분지을 수가 없습니다. 많은 사람들이 "이 소프트웨어는 자유 소프트웨어이다"라고 이야기 할 때, 이것이 어떤 의미를 가지는 있는지는 그 소프트웨어가 가지고 있는 라이센스를 읽어보면 좀 더 자세히 알 수 있습니다.

GNU/FSF에서는 어떤 프로그램이 "자유 소프트웨어"인가 아닌가를 판단할 수 있는 기준으로 다음과 같은 4가지 조건을 규정하고 있는데, 이 조건들을 모두 충족시키는 소프트웨어를 자유 소프트웨어라고 할 수 있습니다.

자유 0 : 프로그램을 어떠한 목적을 위해서라도 실행할 수 있는 자유
자유 1 : 프로그램의 작동 원리를 연구하고, 이를 자신의 필요에 맞게 변경시킬 수있는 자유
자유 2 : 이웃을 돕기 위해서 프로그램을 복제하고 배포할 수 있는 자유
자유 3 : 프로그램을 향상시키고 이를 공동체 전체의 이익을 위해서 다시 환원시킬 수 있는 자유

위의 4가지 조건들은 자유 소프트웨어를 명확하게 규정하기 위한 것이며, 이러한 기준을 만족하는 소프트웨어를 보호하기 위한 법률적 장치가 GNU GPL이라고 불리는 사용권 허가 방식입니다. 흔히 말하는 카피레프트라는 단어는 특정한 소프트웨어를 자유 소프트웨어로 만드는 방식들을 통칭하는 일반적인 명칭이며 카피레프트가 성립하기 위해서는 "자유2"에 해당하는 프로그램에 대한 배포가 일어날 경우에 원시 코드의 개작 여부에 상관없이 배포 기준을 그대로 유지시켜야 합니다. 예를 들면, 자신이 얻은 원시 코드를 개작해서 새로운 기능을 추가하거나 보다 확장된 프로그램을 만들었다 하더라도 최초의 원시 코드에 설정된 사용권 허가 방식을 변경하거나 새로운 제한 사항을 추가하지 않고 원래의 내용을 그대로 유지시켜야 합니다. 따라서 카피레프트로 설정된 프로그램을 기반으로 만들어진 프로그램들은 모두 카피레프트가 될 수밖에 없는 순환적인 구조를 갖게 됩니다. 자유 소프트웨어의 범위와 체계가 현재와 같이 커질 수 있었던 가장 근본적인 외적 이유는 카피레프트가 이러한 순환적이고 팽창적인 구조를 갖고 있기 때문입니다.

자유 소프트웨어를 가름하는 4가지 기준에 따라 다양한 종류의 소프트웨어들을 다음과 같은 다이어그램으로

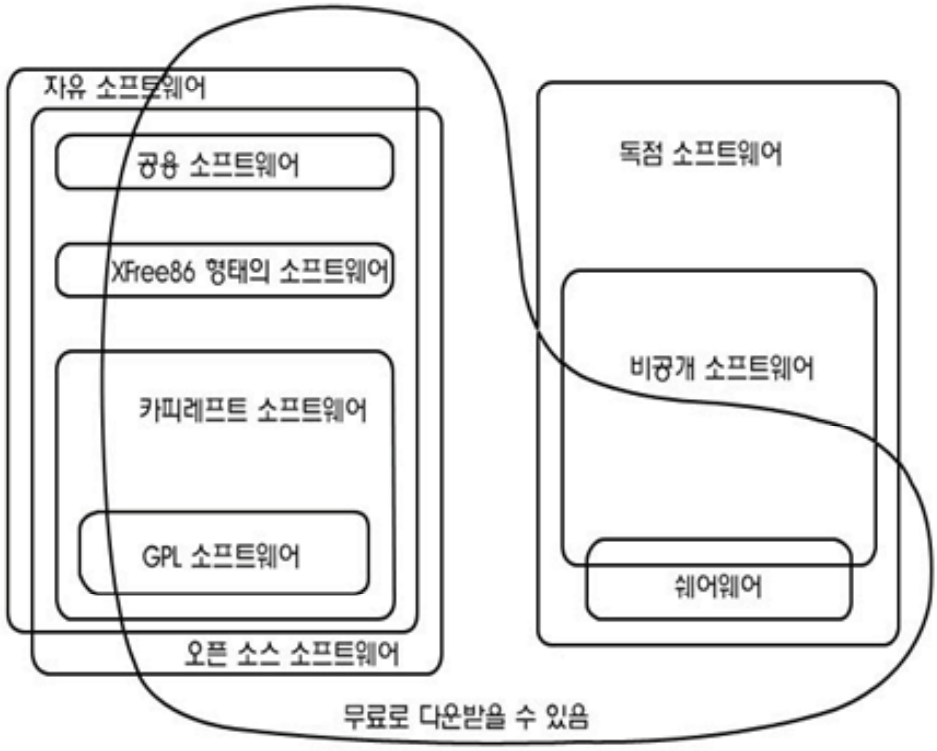


그림 1) 자유 소프트웨어가 되기 위한 4가지 조건에 의한 소프트웨어의 분류

GPL 소프트웨어 - GPL'ed software

GPL을 사용권 허가 방법으로 사용하고 있는 소프트웨어를 지칭합니다. GNU/FSF가 배포하고 있는 거의 대부분의 소프트웨어들은 GPL 소프트웨어입니다.

카피레프트 소프트웨어 - Copylefted software

GNU/FSF가 정의하고 있는 자유 소프트웨어 대한 4가지 조건을 충족시키는 소프트웨어 중에서 카피레프트 방식의 배포가 이루어지는 소프트웨어를 지칭합니다. 즉 원시 코드의 개작 여부에 관계없이 원래의 배포 기준을 그대로 유지시켜야하는 소프트웨어를 말합니다. GPL은 카피레프트를 구현하는 방식이기 때문에 GPL 소프트웨어는 카피레프트 소프트웨어의 부분 집합이 됩니다.

공용 소프트웨어 - Public domain software

저작권자가 저작권을 명시적으로 포기했거나 저작권자를 알 수 없는 공개된 소프트웨어를 지칭합니다. 카피레프트는 저작권을 인정하는 방식이기 때문에 공용 소프트웨어가 카피레프트에 포함되지는 않지만, 저작권이

없음으로 인해서 사용상의 어떠한 제한도 존재하지 않기 때문에 자유 소프트웨어처럼 사용될 수 있습니다. 이 말은 공용 소프트웨어가 독점 소프트웨어로 사용될 수도 있다는 것을 함축합니다. [그림 1]에서는 보다 명확한 분류를 위해 공용 소프트웨어를 자유 소프트웨어의 부분 집합으로만 표시했지만, 경우에 따라서 독점 소프트웨어의 부분 집합으로 표시될 수도 있습니다. GNU/FSF에서는 사용자에게 유리한 방향으로 정의하는 기준을 적용해서 공용 소프트웨어를 `카피레프트 이외의 자유 소프트웨어'로 분류하고 있습니다. [그림 1]에서 자유 소프트웨어를 구성하는 부분 집합 중 카피레프트 소프트웨어의 여집합이 `카피레프트 이외의 자유 소프트웨어'입니다.

XFree86 형태의 소프트웨어 - XFree86 style software

공용 소프트웨어와 마찬가지로 `카피레프트 이외의 자유 소프트웨어' 중 하나입니다. 대표적인 그래픽 유저 인터페이스인 X 윈도우 시스템에 적용되는 사용권 허가 방식이 여기에 해당하는데, 이 방식에서는 개작과 재배포가 허용되지만, 카피레프트에는 허용되지 않는 추가적인 제약의 설정이 가능합니다. 이 말은 이러한 형태의 소프트웨어를 이용해서 독점 소프트웨어를 만드는 것이 가능하다는 뜻입니다.

오픈 소스 소프트웨어 - Open Source Software

오픈 소스에 대한 정의를 충족시키는 소프트웨어를 지칭합니다. Open Source Initiative(OSI)는 다음의 배포 기준을 따르는 소프트웨어를 OSS라 정의 하고 있습니다.

1. 자유로운 재배포
   오픈 소스 사용 허가(license)는 몇 개의 다른 출처로부터 모아진 프로그램들로 구성된 집합 저작물 형태의 배포판의 일부로 소프트웨어를 판매하거나 무상 배포하는 것을 제한해서는 안됩니다. 또한 그러한 판매에 대해 사용료나 그밖의 다른 비용을 요구해서도 안됩니다.

2. 원시 코드
   오픈 소스 프로그램에는 원시 코드(source code)가 포함되어야 하며, 컴파일된 형태 뿐 아니라 원시 코드의 배포도 허용되어야 합니다. 만약 원시 코드가 함께 제공되지 않는 제품이 있다면 원시 코드를 복제하는데 필요한 합당한 비용만으로 원시 코드를 구할 수 있는 널리 알려진 방법이 제공되어야만 합니다. 이러한 경우에 있어 가장 권장할 만한 방법은 별도의 비용없이 인터넷을 통해 원시 코드를 다운받을 수 있도록 하는 것입니다. 원시 코드는 프로그래머가 이를 개작하기에 용이한 형태여야 하며, 고의로 복잡하고 혼란스럽게 만들어진 형태와 선행 처리기나 번역기에 의해 생성된 중간 형태의 코드는 인정되지 않습니다.

3. 파생 저작물
   오픈 소스 사용 허가에는 프로그램의 개작과 2차적 프로그램의 창작이 허용되어야 하며, 이러한 파생 저작물들이 원프로그램에 적용된 것과 동일한 사용 허가의 규정에 따라 배포되는 것을 허용해야만 합니다.

4. 저작자의 원시 코드 원형 유지

오픈 소스 사용 허가는 바이너리를 생성할 시점에서 프로그램을 수정할 목적으로, 원시 코드를 수반한 ``패치 파일''의 배포를 허용한 경우에 한해서 패치로 인해 변경된 원시 코드의 배포를 제한할 수 있습니다. 그러나 이 경우에도 변경된 원시 코드를 통해 만들어진 소프트웨어의 배포는 명시적으로 허용해야만 합니다. 오픈 소스 사용 허가는 파생 저작물에 최초의 소프트웨어와 다른 판 번호(version)와 이름이 사용되도록 규정할 수 있습니다.

5. 개인 및 단체에 대한 차별 금지

오픈 소스 사용 허가는 특정 개인이나 단체를 차별해서는 안됩니다.

6. 사용 분야에 대한 차별 금지

오픈 소스 사용 허가는 프로그램이 특정 분야에서 사용되는 것을 금지하는 제한을 설정해서는 안됩니다. 예를 들면, 기업이나 유전학 연구에 프로그램을 사용할 수 없다는 등과 같은 제한을 설정해서는 안됩니다.

7. 사용 허가의 배포

프로그램에 대한 권리는 배포에 따른 각 단계에서 배포자에 의한 별도의 사용 허가 없이도 프로그램을 재배포받은 모든 사람에게 동일하게 인정되어야만 합니다.

8. 특정 제품에만 유효한 사용 허가의 금지

프로그램에 대한 권리는 프로그램이 특정한 소프트웨어 배포판의 일부가 될 때에 한해서만 유효해서는 안됩니다. 만약 특정 배포판에 포함되어 있던 프로그램을 별도로 분리한 경우라 하더라도 프로그램에 적용된 사용 허가에 따라 프로그램이 사용되거나 배포된다면 프로그램을 재배포받은 모든 사람에게 최초의 소프트웨어 배포판을 통해 프로그램을 배포받은 사람과 동일한 권리가 보장되어야만 합니다.

9. 다른 소프트웨어를 제한하는 사용 허가의 금지

오픈 소스 사용 허가는 오픈 소스 사용 허가가 적용된 소프트웨어와 함께 배포되는 다른 소프트웨어에 대한 제한을 포함해서는 안됩니다. 예를 들면, 사용 허가 안에 동일한 매체를 통해 배포되는 다른 소프트웨어들이 모두 오픈 소스 소프트웨어여야 한다는 제한을 두어서는 안됩니다.

10. 라이센스는 기술 중립적이어야 한다.

라이센스의 어떤 조항도 어떤 개인적인 기술 또는 인터페이스 스타일에 근거하여 이루어져서는 안됩니다.
원문 : http://www.opensource.org/docs/osd

셰어웨어 - Shareware

일정한 기간 동안 무료로 사용할 수 있게 하는 등의 부분적인 제한을 설정해서 배포되지만, 계속해서 사용하기 위해서는 비용을 지불해야 하는 소프트웨어를 지칭합니다. 셰어웨어는 상업적인 목적을 위한 마케팅 방법의 하나로 대부분 원시 코드가 제공되지 않거나 배포상의 제약이 설정되므로 독점 소프트웨어에 속합니다.

프리웨어 - Freeware

셰어웨어와 유사한 형태의 소프트웨어로, 일반적으로 배포는 허용하지만 개작은 허용하지 않는 경향을 갖고 있습니다. 프리웨어라는 표현에 특히 주의해야 하는 것은 자유 소프트웨어가 아님에도 불구하고 유사한 어감의 단어를 사용함으로써 사용자들을 혼동시키고 있기 때문입니다. GNU/FSF에서는 프리웨어라는 표현을 사용하지 않고 있으며, 프리웨어는 결코 자유 소프트웨어가 아닙니다.

비공개 소프트웨어 - Closed software

(오픈 소스 소프트웨어에 대한 상대적인 표현으로) 원시 코드가 공개되지 않는 소프트웨어를 지칭합니다. 원시 코드가 공개되지 않는 것은 `자유1'과 `자유 3'을 성립시킬 수 없기 때문에 비공개 소프트웨어는 독점 소프트웨어에 속합니다.

독점 소프트웨어 - Proprietary software

원시 코드가 공개되지 않거나 프로그램에 대한 복제 및 배포가 금지되는 등의 자유 소프트웨어에 대한 4가지 조건이 충족되지 않는 소프트웨어를 지칭합니다. [그림 1]에 의하면 무료로 다운 받을 수 있는 소프트웨어의 영역에 독점 소프트웨어의 일부가 포함되어 있기도 한데, 무료로 다운 받을 수 있다고 하더라도 사용과 복제 및 배포 상의 제약이 있다면 이는 자유 소프트웨어가 될 수 없기 때문에 독점 소프트웨어에 포함됩니다. Acrobat Reader나 Real Player와 같은 프로그램이 이러한 형식을 갖는 대표적인 예라고 할 수 있습니다. 이와 반대로 무료로 다운 받을 수 있는 소프트웨어의 영역에 자유 소프트웨어의 일부가 제외되어 있는 이유는 자유 소프트웨어라고 해서 무조건 무료로 제공되어야 한다는 조건이 자유 소프트웨어에 대한 4가지 조건에는 포함되어 있지 않기 때문입니다. 따라서 A라는 기업이 GPL로 사용권 허가를 설정한 프로그램을 새롭게 창작한 뒤에 이를 유료로 판매하면서 제품을 구입하는 사람에게만 원시 코드를 제공하는 방식으로 사업을 영위한다고 해도 이는 GPL 위반이 되지 않습니다. GNU GPL은 카피레프트를 실제로 구현한 것이기 때문에 카피레프트 방식을 충족시키기만 하면 구체적인 배포 형태에 대한 제한은 두지 않습니다.

상용 소프트웨어 - Commercial software

[그림 1]에는 언급되어 있지 않지만, 소프트웨어의 종류를 구분할 때 빼놓을 수 없는 중요한 형태의 하나가 바로 상용 소프트웨어입니다. 상용 소프트웨어는 판매 수익을 통해 돈을 벌기 위한 목적으로 만들어진 소프트웨어를 말합니다. 자유 소프트웨어에서 사용된 "자유"는 무료를 의미하는 금전적인 측면의 자유가 아니라 구속되지 않는다는 관점에서의 자유라고 이미 언급한 바 있습니다. 따라서 자유 소프트웨어를 유료로 판매하는 것은 전혀 문제가 되지 않기 때문에 경우에 따라서 자유 소프트웨어가 상용 소프트웨어가 될 수도 있습니다. 또한 독점 소프트웨어라고 해서 모두 상용 소프트웨어가 되는 것도 아닙니다. A라는 기업에 의해서 만들어진 인터넷 유해 정보 차단 프로그램이 인터넷을 통해서 무료로 배포되고 있다고 가정해 봅시다. A가 프로그램의 원시 코드를 공개하지 않고 제3자에 의한 개작과 배포를 허용하지 않는다고 해도 프로그램을 개발한 목적 자체가 공익을 위한 것이었고 또한 프로그램을 전액 무료로 배포하고 있다면 이 프로그램을 상용 소프트웨어로 볼 수 없습니다. 그러나 이 프로그램이 자유 소프트웨어의 조건들을 충족시키지는 못하기 때문에 자유 소프트웨어로 간주될 수도 없습니다. 이 프로그램을 굳이 [그림 1]의 기준에 따라 분류해 본다면 `비상용 독점 소프트웨어'가 된다고 할 수 있습니다. 따라서 상용 소프트웨어는 자유 소프트웨어와 독점 소프트웨어를 가늠하는 구분이 아니라 단지 돈을 받고 판매하는가 아닌가에 근거한 구분 방식입니다. 자유 소프트웨어와 독점 소프트웨어는 경우에 따라서 모두 상용 소프트웨어와 비상용 소프트웨어가 될 수 있습니다.

위의 분류 형태를 가만히 살펴보면, 그 기준이 저작권이 누구에게 있는가에 초점을 맞추고 있는 것이 아니라 프로그램의 원시 코드가 제공되는 지와 개작과 배포를 허용하는지의 여부에 있다는 것을 알 수 있습니다. 이는 소프트웨어의 분류가 다른 사람들의 사용을 염두에 둔 기준을 바탕으로 한 것이기 때문입니다.

일반적으로 라이센스(License)라는 말로 불리는 "사용권 허가" 또는 "사용 허가"는 정해진 범위 안에서의 사용을 허가해 주는 저작권자로부터의 일방적인 승인입니다. 이러한 상황에서는 사용자가 저작권자의 승인 내용에 포함된 조건에 동의할 경우에만 사용 계약이 성립됩니다. GNU GPL(GNU General Public License)은 마지막에 포함된 라이센스라는 단어가 말해주듯이 저작권 설정 문서가 아닌 사용권 허가 문서입니다. 새로운 저작물을 창작하게 되면 명시적인 저작권 설정 절차를 거치지 않는다고 하더라도 창작과 함께 저작권이 자동적으로 인정됩니다. 컴퓨터 프로그램 또한 저작권법에 의해서 이러한 권리가 자동으로 인정되는데, 미국과는 달리 한국에서는 "컴퓨터 프로그램 보호법"이라는 별개의 특별법이 적용됩니다. 그러나 실제적으로는 컴퓨터 프로그램의 특수성을 감안해서 법률의 이름만을 따로 만들 뒤에 세부적인 조항을 저작권법의 범위 안에서 컴퓨터 프로그램에 맞게 적용하도록 입법한 것이기 때문에 저작권법의 틀에서 벗어나지 않습니다.

지금까지, 자유 소프트웨어와, 오픈소스 소프트웨어에 대한 이야기를 해 보았습니다. 앞에서 한 이야기를 다음과 같이 간단히 정리할 수 있습니다.

자유 소프트웨어 재단(FSF, Free Software Foundation)의 자유 소프트웨어 (Free Software) : 소프트웨어에 대한 사용, 복제, 배포의 자유와 소스 코드에 대한 접근을 통해 학습, 수정, 개선할 수 있는 자유를 부여하는 소프트웨어

오프소스 이니셔티브(OSI, Open Source Initiative)의 오픈소스 (Open Source) : 저작권자가 소스코드를 공개하여 누구나 특별한 제한 없이 자유롭게 사용, 복제, 배포, 수정할 수 있는 소프트웨어

두 가지 정의는 거의 비슷하지만 FSF는 소프트웨어의 완전한 "자유"를 중시하고 OSI는 "누구나", "특별한 제한 없이"라는 자유와 평등의 개념을 중시하는 것이 다른 것을 알 수 있습니다.

마지막으로, OSS는 어느 한계까지 발전할 수 있을까요? 특허 제도와 같은 법률적 강제 장치가 자유 소프트웨어를 전면적으로 금지시키지 않는 한 그 가능성은 무한하다고 생각하고 있습니다. GNU/FSF 와, OSI의 공통적인 목표는 컴퓨터 사용자들이 희망하는 어떠한 형태의 작업도 완벽하게 실현시킬 수 있는 OSS를 제공하는 것이며, 그것은 곧 소프트웨어에 있어서 독점이라는 해악을 영원히 사라지게 만들기 위한 것입니다.

안드로이드 애플리케이션 개발자가 알아야 할 Java 언어 기초

양 성 일

이 글은 안드로이드에 입문하면서 자바를 통하여 객체지향 설계에 대해 알고자 하는 개발자를 위해 작성하였다. 자바는 이제 10여년 이란 세월 동안 객체지향 언어의 대표 주자로서 많은 개발자 풀을 가지고 있는 언어이다.

모바일부터 엔터프라이즈 환경까지 다양한 기술 스펙을 가지고 있으며 현 엔터프라이즈 시장에서 WAS(Web Application Server) 기반의 시스템이 주름잡고 있다. 이러한 가운데 구글은 자바 개발자를 안드로이드 개발자로 흡수할 목적으로 안드로이드에 Dalvik VM과 자바기반의 프레임워크를 만들어 자바 개발자들이 자유로이 안드로이드 애플리케이션을 제작 배포할 수 있도록 지원하고 있다.

이 글은 먼저 객체지향을 자바언어를 통하여 설명하고 두 번째로 객체지향 설계의 기본 구조를 제공하는 디자인 패턴에 대해 설명한다.

1. 객체지향의 개요

객체 지향 기술은 소프트웨어 기술의 발전 속도가 하드웨어의 발전에 비하여 매우 느리고 소프트웨어 생산성이 그 수요를 따르지 못하는 근원적인 문제를 해결하고자 복잡한 시스템을 상태(State)와 행동(Behavior)으로 구성된 객체유형(Class)으로 모델링하고 변경에 대한 작업(Overriding, Overloading)이 수월하며 재사용이 용이(Inheritance, Package, Component)하도록 지원한다.

객체 지향의 상태와 행동은 소프트웨어에서는 상태와 행동이 각각 필드(Field)와 메소드(Method)로 표현된다. 메소드는 객체들의 상태를 변경하고 객체간 의사소통의 주요 메커니즘을 지원한다. 그리고 필드들을 직접 접근하는 것을 제한하고 메소드를 통한 상호작용에 의해서만 필드가 변경되게 하는 것 즉, 데이터의 블랙박스화 하는 것을 캡슐화(Encapsulation)라고 한다.

표 1-1 객체지향의 구성요소

구성요소	설명
객체(Object)	세상에 실제로 존재하는 모든 사물 또는 개념
클래스(Class)	객체(Object)가 가질 수 있는 상태와 행동들에 대해 추상화한 것
인스턴스(Instance)	클래스(Class)의 상태와 행동들을 할 수 있는 실세계에 존재하는 것으로서 애플리케이션에서는 변수에 선언되어 메모리에 존재하는 것을 말한다.
메시지(Message)	객체(Object)들 간의 상호작용을 위한 것으로 메소드를 통해 주고 받는 통신 정보들을 말한다.

표 1-2 객체지향의 기본 원리

원리	설명
추상화	불필요한 부분은 생략하고 객체의 속성과 중요한 메시지에 초점을 두어 간략화 하는 것
캡슐화	데이터를 메소드를 통해서만 접근하게 하여 데이터를 보호하는 개념
상속	상위클래스의 속성과 메소드를 하위 클래스가 재정의 없이 물려받아 사용하는 것
다형성	같은 이름의 메소드가 각각 다른 행동을 수행하는 것 Overriding: 부모에 선언된 메소드를 재정의 Overloading: 하나의 클래스에서 같은 행동을 처리하지만 상황에 따라 메시지 유형을 다르게 하는 메소드들을 정의

이제부터 객체지향의 구성요소들과 기본원리에 대해 자세히 살펴보도록 하자.

2. 클래스 (Class)의 정의

우리 주변의 실세계를 살펴보면 대부분의 사물에는 종류나 유형이 있다. 자동차는 화물차, 승용차, 버스 등으로 나눌 수 있으며 자전거도 산악자전거, 세발자전거, 사이클, 하이브리드 자전거등 종류가 다양하다. 그러나 종류가 많아도 같은 유형의 청사진들을 가지고 제작되어 진다. 이렇게 청사진과 같이 객체의 상태와 행동을 모델링한 것을 클래스(Class)라 하며 청사진을 통해 만들어져 실세계에 존재하는 것을 인스턴스(Instance)라고 한다.

클래스(Class)란 어떤 종류의 모든 객체에서 공통된 변수와 메소드들이 정의되어져 있는 Blueprint 혹은 Prototype 이다. 인스턴스(Instance)란 클래스를 통해 실세계(메모리 상)에 존재하는 변수를 말한다.

자 이제 자바에서의 클래스의 구조에 대해 살펴보자. 먼저 클래스 선언문에 대해 살펴보면 다음과 같다.

public	외부에서 접근 가능
abstract	추상 클래스 선언, 인스턴스화 할 수 없음
final	더 이상 자식 클래스 없음
class [클래스 이름]	클래스 명 선언
extents [부모 클래스 이름]	상속받을 클래스 선언

{
Class Body
}	

굵은 문자로 표시한 부분은 반드시 선언해야 하는 부분이고 그 외는 옵션이다.

처음에는 클래스의 접근자를 선언하게 되는데 클래스의 접근자는 public과 선언을 하지 않는 것이다. 클래스의 접근자를 선언하지 않으면 같은 패키지 내에서만 접근이 가능하고 만약 public을 선언하게 되면 외부에서 이

클래스에 대한 접근이 가능하게 된다.

그 다음 클래스의 메소드의 일부를 자식 클래스에서 정의하도록 위임하고 싶을 경우에는 abstract를 선언하여 추상클래스로 정의해야 한다. 추상 클래스는 공통된 메소드를 구현하고 일부의 메소드는 정의만 해놓은 클래스이다. 추상클래스는 인스턴스로 생성할 수 없다.
추상 클래스에 대해서는 뒤에서 자세히 설명하도록 하겠다.

final은 더 이상 자식 클래스를 만들지 않겠다고 선언하는 것이다. 즉 어떤 클래스에서도 부모 클래스로 선언될 수 없게 된다.

클래스 명을 선언해야 한다. 클래스 명은 자바의 키워드와 특수문자를 사용할 수 없으며 명사형과 첫 글자는 대문자로 하는 것이 권고 사항이다.

만약 상속을 하고 싶으면 상속할 부모 클래스를 선언한다. 단, 자바에서는 C++과 달리 단일 상속만 지원한다. 단일 상속이란 하나의 부모 클래스만 가질 수 있는 것이고 C++등에서 지원하는 다중 상속은 여러 부모 클래스의 필드와 메소드를 상속받는 것을 말한다.

마지막으로 구현할 인터페이스가 있으면 인터페이스를 선언한다. 인터페이스 선언을 해주면 인터페이스의 메소드를 다 구현해야 컴파일을 할 수 있다. 만약 구현을 할 수 없을 경우에는 클래스를 abstract로 선언하여야 한다.

```
public class Car {

}
```

Class Body의 요소를 살펴 보면 다음과 같다.

Variable	클래스에서 상태 정보를 저장하는 변수
Method	클래스의 행동을 정의
Property	오직 set/get 메소드를 통해서만 접근할 수 있는 variable
Constructor	객체를 생성할 시 호출되며 객체에 대한 초기화를 위한 코드가 존재
finalize	객체가 가비지 콜렉션에 의해 소멸될 때 호출되는 메소드

Class Body 요소 중 Variable, Method, Constructor에 대해 살펴보도록 하겠다.

Variable 선언은 다음과 같이 구성되어진다.

accessLevel	변수에 대한 외부 접근 제한
static	클래스 전역 변수 선언
final	상태 값이 변하지 않는 변수 선언
transient	객체의 직렬화 (Serializable)시 직렬화에 포함시키지 않을 변수로 선언

volatile	멀티쓰레드에서 상태값의 안정성 보장을 위한 선언
type nam	변수의 형과 이름을 정의

accessLevel은 변수에 대한 외부에서의 접근 제한자를 선언하는 것으로private, protected, public이 있으며 어떠한 것도 선언하지 않아도 된다. 접근 제한자에 대한 설명은 뒤의 캡슐화에서 자세히 설명하겠다.

static은 클래스의 전역변수를 선언하는 것으로서 [클래스 명].[변수 명]으로 접근할 수 있다. 전역변수는 애플리케이션에서 하나만 존재하며 모든 인스턴스에 공유하여 사용되어진다.

final 은 상태 값이 한번 초기화되면 더 이상 변경할 수 없는 변수로 선언할 때 사용한다.

transient 는 자바에서 객체의 영속성을 위한 객체 직렬화에서 사용되는 선언인데 객체 직렬화는 파일이나 네트워크를 통하여 저장 혹은 전송할 때 객체의 현재 상태를 표현하는 프로토콜이다. 만약 변수를 transient로 선언하면 직렬화시 제외된다. 역 직렬화 시 적절한 값으로 초기화해야 한다.

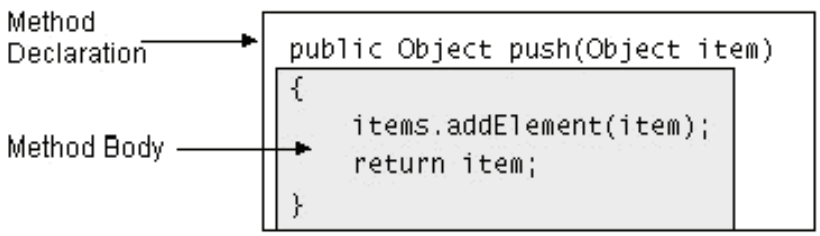
volatile 은 자바 프로그래머에게는 잘 사용되지 않는 변수지만 임베디드 개발자에게는 익숙할 것이다. 이 선언은 멀티 쓰레드 환경에서 변수의 값처리를 한번에 하게 하여 예상치 못하게 변경되는 것을 막기 위한 것이다.

마지막으로 유형과 이름을 정의하면 된다. 유형은 byte, short, int, long, float, double, Boolean, char 등 primitive type과 클래스 명으로 선언하면 되고 이름은 대소문자를 구분하며 알파벳문자와 '$', '_'를 사용할 수 있다.

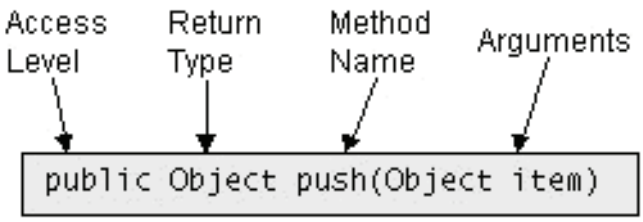
자바에서는 일반적으로 소문자로 시작하며 만약 여러 개의 단어로 할 경우에는 첫 단어는 소문자로 시작하고 다음단어 부터는 대문자로 시작하도록 권고 하고 있다.

```
public class Car {
        private int gear;
        private int speed;
}
```

메소드(Method)는 메소드 선언과 { }로 둘러 쌓인 메소드 몸체로 구분되어진다.



메소드 선언의 기본적으로는 접근레벨, 반환 유형, 메소드 이름, 파라메터들로 구성되어진다.



Method 선언에서 상황에 따라 사용할 수 있는 다양한 키워드를 다음과 같이 제공한다.

accessLevel	메소드의 접근 제한자
static	클래스의 전역 메소드 선언
abstract	구현하지 않은 메소드 선언 (추상화)
final	더 이상 오버라이딩 할 수 없는 메소드 선언
native	C등 자바 이외의 언어로 구현될 메소드 선언
synchronized	쓰레드의 동시 접근을 제어
returnType methodName	반환형과 메소드 명
(paramlist)	Argument 리스트
throws exceptions	메소드에서 발생할 수 있는 예외사항들

accessLevel은 외부에서 이 메소드에 접근할 수 있는 제한자를 선언하며 자세한 것은 캡슐화에서 설명하도록 하겠다.

static 은 클래스의 전역 메소드로 선언하는 것이며 [클래스명].[메소드]로 접근할 수 있다. (Camera.open() 과 같은 메소드)

abstract는 메소드를 구현하지 않고 정의만 할 때 사용하는 것으로 abstract 메소드가 포함된 클래스는 반드시 abstract 선언을 하여야 한다. 이러한 클래스를 추상 클래스라고 한다. 참고로 추상 클래스는 인스턴스로 생성될 수 없다. 추상 클래스에서 자세히 살펴보도록 하겠다.

final은 더 이상 해당 메소드를 자식 클래스에서 override 할 수 없게 할 때 선언한다.

native는 JNI (안드로이드에서는 NDK를 통한 개발)를 통해 구현하기 위해 선언하는 것으로 클래스에서 native로 선언된 메소드는 javah 툴을 통해 header 소스가 만들어져 C, C++로 구현할 수 있다.

synchronized는 멀티 쓰레드에서 해당 메소드에 하나의 쓰레드만 접근할 수 있도록 하기 위해 선언된다. 하나의 쓰레드가 메소드에 대해 실행하고 있는 중에는 다른 쓰레드는 대기 상태에 있게 된다.

returnType methodName은 메소드의 반환 유형과 메소드 명을 선언하는 것이며 반환 유형은 primitive 형과 클래스를 선언하며 메소드 명은 변수명 규칙을 따른다.

```
public class Car {
```


```
        private int gear;
        private int speed;

        public void changeGear(int newValue){
                gear = newValue;
        }

        public void speedUp(int increment){
                speed = speed + increment;
        }

        public void applyBrakes(int decrement){
                speed = speed - decrement;
        }

        public void printStates(){
                System.out.println("speed:"+speed+" gear:"+gear);
        }

        public Car newCar(){
                return new Car();
        }
}
```

생성자는 클래스의 인스턴스를 생성할 때 사용되는 것으로 구조적으로 메소드와 같지만 반환 유형을 선언하지 않는다. 그리고 이름은 클래스 이름을 사용하여야 한다. 일반적으로 인스턴스를 초기화 하는 코드가 존재한다.

자바에서는 클래스 정의 시 생성자가 없으며 자동으로 기본 생성자를 추가하여 컴파일된다.

```
public class DefaultConstructor {

        public void hello(){
                System.out.println("생성자가 정의되지 않은 클래스입니다.");
        }
}

public class ConstructorMain {

        public static void main(String[] args) {
                DefaultConstructor test = new DefaultConstructor();
                test.hello();
        }

}

<실행>
생성자가 정의되지 않은 클래스입니다.
```


```
public class DefaultConstructor {

	public void hello(){
		System.out.println("생성자가 정의되지 않은 클래스입니다.");
	}
}

public class ConstructorMain {

	public static void main(String[] args) {
		DefaultConstructor test = new DefaultConstructor();
		test.hello();
	}

}
```

<실행>
생성자가 정의되지 않은 클래스입니다.

생성자는 다른 형식으로 여러 개를 정의할 수 있다.

```
public class MultiConstructor {

	private String mName;

	public MultiConstructor(){
		mName = "생성자";
	}

	public MultiConstructor(String aName){
		mName = aName;
	}

	public void hello(){
		System.out.println("이름 : " + mName);
	}
}

public class ConstructorMain {

	public static void main(String[] args) {
		MultiConstructor test1 = new MultiConstructor();
		test1.hello();
		MultiConstructor test2 = new MultiConstructor("칸드로이드");
		test2.hello();
	}
```


```
}

<실행>
이름 : 생성자
이름 : 칸드로이드
```

생성자에서 다른 형식의 생성자를 호출할 수 있는데 이때 this(...)를 사용한다. 위의 MultiConstructor 클래스의 생성자를 다음과 같이 변경할 수 있다.

```
public class MultiConstructor {

	private String mName;

	public MultiConstructor(){
		this("생성자");
	}

	public MultiConstructor(String aName){
		mName = aName;
	}

	public void hello(){
		System.out.println("이름 : " + mName);
	}
}
```

기본 생성자는 MultiConstructor(String aName) 생성자를 호출하여 초기화한다.
여기서 주의해야 할 점이 있는데 this(...)는 생성자 코드의 첫 줄에서 호출해야 한다. 중간 혹은 마지막에 있으면 컴파일 에러가 발생한다.

부모 클래스에서 기본 생성자가 존재하지 않으면 자식 클래스에서는 기본 생성자를 정의할 수 없다. 이것에 대한 자세한 설명은 상속에서 살펴보도록 하겠다.

이제 클래스의 이해도를 높이기 위해 예제를 통하여 살펴보도록 하겠다.

```
public class Car {
	private int gear;
	private int speed;

	public void changeGear(int newValue){
		gear = newValue;
	}
```


```
        public void speedUp(int increment){
                speed = speed + increment;
        }

        public void applyBrakes(int decrement){
                speed = speed - decrement;
        }

        public void printStates(){
                System.out.println("speed:"+speed+" gear:"+gear);
        }

        public Car newCar(){
                return new Car();
        }
}
```

위의 코드는 클래스에 대해 설명하면서 정의한 자동차에 대한 클래스로서 gear, speed 필드가 객체의 상태이고 외부에서는 클래스의 인스턴스를 생성하여 changeGear, speedUp, applyBrakes 와 같은 메소드를 호출하여 실행시킨다.

마지막 printStates 메소드는 상태의 값이 최종적으로 어떻게 변했는지 확인하기 위한 것으로 System.out은 자바의 기본 출력스트림으로 PrintStream의 인스턴스이다. 그리고 println은 출력 스트림에 한 라인의 문자열을 출력시킨다.

Car 클래스는 main 메소드를 포함하고 있지 않으므로 완벽한 자바 애플리케이션이 아니다. 자바 애플리케이션의 시작은 main 메소드를 가지고 있는 클래스부터 시작한다. 다음은 Car 클래스를 사용하는 메인 클래스이다.

```
class CarDemo {

        public static void main(String[] args) {
            Car car1 = new Car();  // 인스턴스 생성
            Car car2 = new Car();

            car1.speedUp(10);
            car1.changeGear(2);
            car1.printStates();

            car2.speedUp(10);
            car2.changeGear(2);
            car2.speedUp(10);
            car2.changeGear(3);
            car2.printStates();
        }
}
```


위와 같인 자바는 main 메소드가 시작 점이다. main 메소드는 항상 public static void 로 선언되어져야 하며 문자열 배열을 매개변수로 받아야 한다. 옵션으로 내부에서 발생하는 예외사항을 처리하지 않고 가상머신에게 넘기고자 할 때는 throws Exception을 붙여서 선언한다.

public static void main(String[] args)throws Exception

Car 클래스 객체를 2개 생성하였다. 이렇게 메모리상에 존재하게 되었을 때 car1 인스턴스, car2 인스턴스라고도 한다.

인스턴스를 생성한 후 클래스에 정의된 메소드를 통해 메시지를 전송하며 인스턴스의 상태를 변경시킨다.

위의 소스를 실행하면 다음과 같이 표시된다.

```
speed:10 gear:2
speed:20 gear:3
```

3. 캡슐화(Encapsulation)

위의 Car 클래스에서 보는 것과 같이 외부에서는 변수를 직접 접근 못하게 제한하고 오직 메소드를 통해서만 값이 변경되게 하는 것을 정보은닉 즉 캡슐화(Encapsulation) 라고 한다.

자바에서 호출에 대한 접근을 제한하는 경우로는 다음과 같이 4가지 경우가 있으며 접근 제한자는 3가지가 있다.

키워드	class	subclass	package	world	설명
private	O				오직 클래스 내에서만 사용
protected	O	O	O		같은 패키지내에서 호출 가능
public	O	O	O	O	어디서든 호출 가능
선언 안 함	O		O		같은 패키지내에서 호출 가능 (안드로이드 소스 내에서는 가독성을 높이기 위해 /*package*/로 표시되어 있음)

접근 제한자는 클래스의 변수, 메소드, 생성자에서 사용할 수 있으며 클래스에는 public과 선언 안 함만 사용할 수 있다.

클래스의 캡슐화를 구현하려면 변수에는 외부에서 접근하지 못하도록 private으로 선언하고 메소드는 private 외의 접근 제한자로 선언하여 외부에서는 오직 메소드를 통해서만 상태를 변경할 수 있도록 하는 것이다.

위의 Car 클래스는 변수에 대해 private으로 선언되어 있으므로 CarDemo의 main 메소드에서 다음과 같이 직접 변수의 상태를 변경하려는 코드가 있으면 컴파일 에러가 발생한다.

```
car1.gear = 4;
car1.printStates();
```

접근 제한자는 생성자에서도 사용할 수 있는데 생성자를 private으로 선언하면 오직 해당 클래스의 코드내에서만 접근 할 수 있으므로 외부에서는 private 생성자로는 인스턴스를 생성할 수 없다. 이러한 방식은 디자인 패턴 중 singleton 패턴에서 사용한다.

```
public class Singleton {
	private static Singleton instance;

	public static Singleton getInstance(){
		if(instance == null){
			instance = new Singleton();
		}

		return instance;
	}

	private String name;

	private Singleton(){
		name = "Singleton Instance";
	}

	public void setName(String name){
		this.name = name;
	}

	public void printName(){
		System.out.println(name);
	}
}
```

```
public class SingletonMain {

	public static void main(String[] args) {
		//Singleton obj = new Singleton(); // 컴파일 에러
		Singleton obj1 = Singleton.getInstance();
		obj1.printName();
		Singleton obj2 = Singleton.getInstance();
		obj2.setName("change name");
		obj1.printName();
```



```
        }

}
```

<실행>
Singleton Instance
change name

4. 상속(Inheritance)

이제 클래스의 계층구조를 이루게 하는 상속에 대해 살펴보도록 하겠다.

자전거를 예를 들면 자전거의 공통 상태와 행동 외에 산악자전거, 일반자전거, 사이클에 대해 각각 다른 상태와 행동들이 있을 것이다. 이런 경우 자전거의 공통된 사항을 각각의 자전거 유형마다 설계하면 많은 자원낭비가 될 것이다.

이럴 경우 부모와 자식 클래스 관계를 해주는 상속을 이용하는 것이다. 자전거의 공통된 사항을 하나의 자전거 클래스로 정의하고 각각 자전거 유형에 따라 클래스를 정의하는데 앞서 정의한 자전거 클래스를 상속받고 추가적인 특성에 대해서만 정의하는 것이다. 이렇게 되면 각각 자전거 유형의 클래스를 정의할 때 따로 공통된 사항을 중복하여 기술할 필요가 없는 것이다.

```
public class Bicycle {

    // 3개의 변수를 정의
    private int cadence;
    private int gear;
    public int speed;

    // Bicycle 는 하나의 생성자 정의
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // 4개의 메소드를 정의
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }
```


```
    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

자전거의 기본 특징을 정의 한 후

```
public class MountainBike extends Bicycle {

    // MountainBike 클래스는 하나의 변수를 추가
    public int seatHeight;

    // 하나의 생성자 정의
    public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
        // 인스턴스 생성시 부모 클래스의 변수에 대해 초기화 하기 위해 super 생성자 호출
            super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // 1개의 메소드 추가
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }

}
```

산악 자전거의 특징을 추가적으로 정의한다.
클래스 상속을 위해서는 extends 키워드를 사용하여 부모 클래스를 선언하며 자식 클래스에서 부모클래스의 변수, 메소드를 사용할 수 있다. 생성자 정의 시 부모 클래스에 정의된 변수도 초기화가 필요한데 이럴 때 부모의 생성자를 호출한다. 부모의 생성자는 super(...) 를 사용하여 호출한다.

```
public class MountainBike extends Bicycle {

public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {

super(startCadence, startSpeed, startGear);
        ...
    }

    ...
```



```
    public void printSpeedAfterUp(int increment){
        speedUp(increment);
        System.out.println("speed : " + speed);
    }
}
```

부모 클래스에서 private으로 선언 된 변수와 메소드는 상속 대상에서 제외된다. 여기서 Bicycle 클래스의 speed 변수을 private으로 선언을 변경하면 MoutainBike 클래스의 printSpeedAfterUp 메소드에서 접근할 수 없으므로 컴파일 에러가 발생한다.

```
public class Bicycle {

    private int cadence;
    private int gear;
    /*public -> */ private int speed;
    ...
}
```

```
public void printSpeedAfterUp(int increment){
    speedUp(increment);
    System.out.println("speed : " + speed); // 컴파일 에러 발생
}
```

5. 다형성(Polymorphism)

다형성이란 의미는 여러 가지 형태를 가지고 있다는 것으로서 하나의 메소드로 여러 형태의 메시지를 받아 동착할 수 있는 것을 뜻한다.

객체지향에서 다형성은 Overriding, Overloading 을 말하는데 Overriding은 상속과 연관되어져 있는 개념으로서 부모 클래스에서 정의된 메소드의 기능을 확장하거나 재정의하는 것이고 Overloading은 같은 이름의 다양한 파라메터 리스트를 가진 메소드들을 정의하여 여러 형태의 메시지를 단일 행위가 처리할 수 있도록 하는 것이다.

구분	Overloading	Overriding
의미	다중 정의	재정의
정의 규칙	- 같은 메소드 명 - 다른 파라메터 형식	- 같은 메소드 명 - 같은 파라메터 형식
적용	여러 형태의 메시지 처리를 위한 단일 행위 구현	부모 클래스의 메소드를 재정의


```
public class Printer {
	public void print(String value){
		System.out.println("String->" + value);
	}

	public void print(boolean value){
		System.out.println("boolean->" + value);
	}

	public void print(int value){
		System.out.println("int->" + value);
	}

	public void print(float value){
		System.out.println("float->" + value);
	}
}
```

Printer 클래스는 print 라는 하나의 행위를 가지고 있지만 출력되는 자료의 유형에 따라 여러 형식으로 각각 다중정의(Overloading)되어져 있다.

```
public class PrinterMain {

	public static void main(String[] args) {
		Printer obj = new Printer();
		obj.print("칸드로이드");
		obj.print(true);
		obj.print(1234);
		obj.print(12.34f);
	}

}

<실행>
String->칸드로이드
boolean->true
int->1234
float->12.34
```

생성자도 또한 다중 정의(Overloading)가 가능하다. 생성자 다중 정의는 보통 상태들을 기본적으로 제공할 값으로 처리할 경우 사용하며 만약 다른 형식의 생성자로 초기화를 위임할 경우 this(...)를 호출한다.

```
public class Mail {
	private String subject;
	private String to;
	private String from;
```


```
        private String message;

        public Mail(String message, String to){
                this("No Title", message, to);
        }

        public Mail(String subject, String message, String to){
                this(subject, message, to, "webmaster");
        }

        public Mail(String subject, String message,
                                        String to, String from){
                this.subject = subject;
                this.message = message;
                this.to = to;
                this.from = from;
        }

        public void print(){
                System.out.println(
                        "제목[" + subject + "],내용[" + message + "]," +
                        to + "에게 " + from + "부터");
        }
}
```

Mail 클래스는 메일를 간단하게 정의하였다. 첫번째 생성자는 메시지와 받는사람만 파라메터로 받는다. 이때 제목은 "No title"로 초기화하도록 두번째 생성자를 호출한다. 두번째 생성자도 보내는 사람을 "webmaster"로 초기화 하여 세번째 생성자를 호출한다. 마지막 생성자는 모든 변수를 초기화한다.

```
public class MailMain {

        public static void main(String[] args) {
                Mail mail1 = new Mail("좋은 하루입니다.", "영희");
                mail1.print();

                Mail mail2 = new Mail("안녕하세요.", "좋은 날씨네요", "영희", "철수");
                mail2.print();
        }

}
<실행>
제목[No Title],내용[좋은 하루입니다.],영희에게 webmaster부터
제목[안녕하세요.],내용[좋은 날씨네요],영희에게 철수부터
```

첫번째 Mail 인스턴스는 메시지와 받는 사람만 설정하는 생성자를 사용하였지만 결과 값을 보면 기본 값인 "No Title"과 "webmaster"가 초기화 되어져 있는 것을 볼 수 있으며, 두번째 Mail 인스턴스는 제목, 메시지, 받는 사람, 보내는 사람을 설정하는 생성자를 사용하였고 결과값도 기본값이 아닌 설정한 값으로 표시되는 것을 볼 수 있다.

자 이제 재정의(Overriding)에 대해 살펴보도록 하자. 재정의는 상속과 연관되어져 있는 객체지향의 원리이며 부모 클래스에서 정의된 메소드를 자식 클래스에서 다르게 정의하는 것이다.

```
public class ChildPrinter extends Printer {
	public void print(String value){
		super.print(value);
		System.out.println("Child print String : " + value);
	}

	public void print(boolean value){
		System.out.println("Child print boolean : " + value);
	}
}
```

ChildPrinter는 Printer 클래스를 상속 받았으며 String을 파라메터를 받는 print 메소드와 Boolean을 파라메터로 받는 print 메소드를 재정의 하였다.
여기서 String 파라메터의 print 메소드에서는 부모 클래스에서 정의한 print(String) 메소드를 호출하기 위해 super를 사용하였다. 이와 같이 부모에 정의된 메소드나 변수를 참조할 때에는 super 키워드를 사용한다.

```
public class PrinterMain {

	public static void main(String[] args) {
		ChildPrinter obj = new ChildPrinter();
		obj.print("칸드로이드");
		obj.print(true);
		obj.print(1234);
		obj.print(12.34f);
	}

}
<실행>
String->칸드로이드
Child print String : 칸드로이드
Child print boolean : true
int->1234
float->12.34
```

여기서 하나 더 살펴보면 부모클래스 유형의 변수는 자식 클래스의 인스턴스를 참조할 수 있다는 것이다. 위의 main 메소드에서 ChildPrinter obj를 Printer obj로 변경한 후 실행해보면 결과가 같은 것을 볼 수 있다.

```
public class PrinterMain {

	public static void main(String[] args) {
```


```
                Printer obj = new ChildPrinter();
                obj.print("칸드로이드");
                obj.print(true);
                obj.print(1234);
                obj.print(12.34f);
        }

}
<실행>
String->칸드로이드
Child print String : 칸드로이드
Child print boolean : true
int->1234
float->12.34
```

그러나 부모 클래스 변수로 참조하면 부모 클래스에 정의된 메소드만을 호출할 수 있다.

부모 클래스 생성자는 상속이 되지 않으므로 재정의 대상이 아니다. 그러나 생성자를 정의할 때 상태들의 초기화를 위해서는 반드시 부모클래스의 생성자를 호출하여야 한다. 부모클래스의 생성자는 super(...) 를 호출한다.

```
public class ChildMail extends Mail {

        public ChildMail(String message, String to) {
                super("제목 없음", message, to, "관리자");
        }

}
```

ChildMail은 Mail 클래스를 상속받았다. Mail의 첫번째 생성자였던 Mail(String message, String to)를 통해 제목과 보낸 사람을 한글로 설정하도록 정의하였다.

```
public class MailMain {

        public static void main(String[] args) {
                Mail mail1 = new ChildMail("좋은 하루입니다.", "영희");
                mail1.print();

                // 부모 클래스의 생성자 유형으로는 생성할 수 없음.
                //Mail mail2 = new ChildMail("안녕하세요.", "좋은 날씨네요", "영희", "철수");
                //mail2.print();
        }

}
<실행>
제목[제목 없음],내용[좋은 하루입니다.],영희에게 관리자부터
```


6. 추상 클래스(Abstract Class)

추상 클래스는 메소드를 구현 하지 않고 정의만 한 것이 하나 이상 존재하는 클래스이다. abstract 키워드를 클래스와 메소드에 각각 선언해주어야 한다.

```
public abstract class GraphicObject {
	private String text;

	public void setText(String text){
		this.text = text;
	}

	public String getText(){
		return text;
	}

	// 메소드 정의
	public abstract void draw();
}
```

추상 클래스는 부모 클래스를 정의할 때 사용한다. 즉 공통으로 정의할 수있는 대부분의 메소드는 부모 클래스에 존재하고 일부 특성화 시킬 메소드만 자식 클래스에서 정의하도록 유도하기 위한 것이다.

추상 클래스는 인스턴스로 생성될 수 없으며 자식 클래스들이 반드시 정의되어야 하는 클래스이므로 클래스 선언 시 더 이상 자식 클래스를 정의 할 수 없다는 final 키워드를 사용할 수 없다.

추상 클래스를 상속받은 자식 클래스에서는 반드시 추상 메소드를 구현하여야 한다.

```
public class GradientGraphic extends GraphicObject {

	@Override
	public void draw() {
		System.out.println("drawing " + getText());
	}

}
```

```
public class GraphicMain {

	public static void main(String[] args) {
	   // 컴파일 에러
		//GraphicObject obj = new GraphicObject();
```


```
                GraphicObject obj = new GradientGraphic();
                obj.setText("Graphic");
                obj.draw();
        }

}
<실행>
drawing Graphic
```

7. 네스티드 클래스(Nested Class)

Nested Class는 어떤 클래스 내에 정의되어진 클래스를 말하는데 외부 접근 방식에 따라 static, non-static 으로 나눠진다. 이 때 Nested 클래스를 포함하고 잇는 클래스를 상대적으로 Outer 클래스라고 한다. static nested 클래스는 클래스 선언 시 static 선언을 해준 것이고 그렇지 않은 nested 클래스를 non-static 클래스인데 이러한 클래스를 inner 클래스라고 한다.

안드로이드 개발 시 만들어지는 R 클래스내에 id, drawable, layout, string등 nested 클래스에 대한 정의가 만들어지는데 이러한 클래스들을 static nested class라고 부른다.

```
package sample;

public final class R {

public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class id {
        public static final int sclView=0x7f050000;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

Nested 클래스를 사용하는 목적은 다음과 같다.
- 오직 하나의 클래스에서만 사용되어질 경우 사용하는 클래스 내에 사용되는 클래스의 정의를 포함시켜 논리적으로 그룹핑하는 것이다.

- 캡슐화를 증대시키는 목적으로도 사용한다. 즉, A라는 클래스에 private으로 선언된 멤버(변수와 메소드)를 B라는 클래스에서 사용해야 할 경우 B 클래스를 A클래스 내에 정의한다. 만약 B 클래스가 static nested 클래스일 경우에는 A 클래스의 전역 멤버(static 선언)만 접근할 수 있다.

```
public class OuterClass {
	private String name;

	public static class InnerClass{
		public void printName(){
			System.out.println(name); // 컴파일 에러
		}
	}
}
```

- 클래스 내에 Nested 클래스를 사용하는 코드 영역과 가까이 있게 하여 소스의 가독성과 유지보수성을 높이는데 목적이 있다.

Nested 클래스의 인스턴스를 생성할 때 static nested 클래스와 inner 클래스가 방식이 다르다. static nested 클래스는 new [Outer 클래스 명].[nested 클래스 명]()으로 바로 생성하지만 inner 클래스는 먼저 Outer 클래스에 대한 인스턴스를 생성 한 후 outerInstance.new [inner 클래스명]()으로 생성해야 한다.

```
public class OuterClass {
	private String name;

	public void setName(String name){
		this.name = name;
	}

	public class InnerClass{
		public void printName(){
			System.out.println(name); // 컴파일 에러
		}
	}

	public static class StaticNestedClass{
		public void printName(){
			System.out.println("static nested class");
		}
	}
}
```

```
public class NestedMain {
	public static void main(String[] args){
```



```
                OuterClass outer = new OuterClass();
                outer.setName("Outer class");
                // Inner 클래스는 outer 클래스의 인스턴스를 생성한 후 생성 가능
                OuterClass.InnerClass inner = outer.new InnerClass();
                inner.printName();

                // static nested 클래스는 바로 생성 가능
                OuterClass.StaticNestedClass staticClass =
                        new OuterClass.StaticNestedClass();
                staticClass.printName();
        }
}

<실행>
Outer class
static nested class
```

마지막으로 nested 클래스 중 코드상에서 오직 한번 만 인스턴스로 생성하기 위해 정의하는 클래스가 있는데 이러한 클래스를 Anonymous 클래스라고 한다.

Anonymous 클래스는 보통 이벤트의 리스너 인스턴스를 생성할 때 많이 사용한다.

```
Button btn = new Button(getApplicationContext());

btn.setOnClickListener(new OnClickListener(){

        public void onClick(View v) {

                Toast.makeText(getApplicationContext(),
                "Button Click", Toast.LENGTH_SHORT);

        }

});
```

8. 인터페이스(Interface)

객체 지향 시스템의 동작은 행위들에 메시지 송수신 함으로서 이루어 진다고 앞서 설명하였다. 이때 메시지 처리에 대한 전체적인 행위에 대해서 선언만 한 것이 인터페이스 이다. 실세계를 예를 들면 데이터베이스 관리자를 채용할 때 DBA 자격증을 가지고 있는 사람을 뽑는데 이 때 DBA 자격증이 인터페이스와 같은 개념으로 자격증을 가지고 있는 사람한테 데이터베이스 관리 업무를 처리하도록 하는 것과 같이 인터페이스를 구현한 인스턴스에게 메시지 처리를 호출하는 것이다.

인터페이스 선언 요소는 다음과 같다.

public	외부에서 접근 가능
interface [인터페이스 이름]	인터페이스 명 선언
extents [부모 인터페이스 이름],...	상속받을 인터페이스 선언, ','로 구분하여 여러 인터페이스 선언 가능

public {
Interface Body
}	

추상클래스는 공통된 메소드에 대해서는 구현을 하고 자식 클래스에서 특성화하는 메소드만 정의하는 것이지만 인터페이스는 메소드 전체에 대해 정의만 하는 차이가 있다.

인터페이스는 인터페이스들을 상속받아 추가적으로 정의할 수 있는데 이때 클래스와 달리 다중 상속이 가능하다. (클래스는 단일 상속만 가능)

```
public interface Interface1 {
        //public int size; // 컴파일 에러
public int size = 0;
            public void method1();
}

public interface Interface2 {
            public void method2();
}

public interface Interface3 extends Interface1, Interface2 {
            public void method3();
}
```

Interface1 에서 보는 것과 같이 인터페이스는 변수를 선언할 수 없으며 상수, 메소드 정의등이 포함된다.

인터페이스의 구현은 클래스에 implements 키워드로 인터페이스를 선언하여 인터페이스에 정의된 메소드들을 구현한다. 이 때 인터페이스에 정의된 메소드 전체를 구현해야 하며 만약 그렇지 않을 경우에는 abstract를 사용하여 추상 클래스로 선언해야 한다.

자 이제 예제를 통하여 인터페이스를 살펴보도록 하겠다. 예제는 명령어 인터페이스를 정의 한 후 여러 명령어를 구현하여 콘솔을 통하여 명령어 입력을 받아 처리 결과를 콘솔에 보여주는 것이다.

```
public interface Command {
            // 명령어 이름 반환
            public String getName();
```


```
        // 명령어 파라메터 설정
        public void setParameters(String params);
        // 명령어 실행
        public String execute();
}
```

명령어 인터페이스는 명령어를 구분하기 위한 이름을 반환해주는 getName, 명령어의 파라메터들을 설정하기 위한 setParameters, 그리고 명령어 처리를 위한 execute 메소드 3개를 정의하였다.

예제에서는 2개의 명령어를 구현하는데 하나는 숫자들에 대한 합산을 처리하는 AddCommand와 설정한 파라메터를 그대로 반환하는 EchoCommand를 정의하였다.

```
public class EchoCommand implements Command {
        private String params;
        public String execute() {
                return params;  // 파라메터를 그대로 반환
        }
        public void setParameters(String params) {
                this.params = params; // 파라메터 설정
        }
        public String getName() {
                return "echo";
        }
}
public class AddCommand implements Command {
        private int[] nums;
        public String execute() {
                int sum = 0;
                for(int i = 0; i < nums.length; i++){ // 파라메터로 입력된 숫자들에 대한 합산 계산
                        sum += nums[i];
                }
                return String.valueOf(sum); // 합산 결과 반환
        }
        public void setParameters(String params) {
                String[] numStr = params.split(","); // ','를 구분으로 파라메터를 파싱
                nums = new int[numStr.length];
                for(int i = 0; i < numStr.length; i++){ // 파라메터를 숫자로 변환
                        nums[i] = Integer.parseInt(numStr[i].trim());
                }
        }
        public String getName() {
                return "add";
        }
}
```

이제 main 메소드를 구현하는데 main에서는 명령어들을 미리 생성한 후 콘솔로 명령어와 파라메터를 입력받아

명령어 처리 후 반환된 문자열을 콘솔에 보여준다. 만약 명령어가 없을 경우에는 '알 수 없는 명령어입니다.' 메시지를 보여준 후 다음 명령어를 대기한다. 명령어가 'quit'이면 프로그램을 종료한다.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Hashtable;

public class CommandMain {

	public static void main(String[] args) throws IOException {
		Hashtable<String, Command> commands =
			new Hashtable<String, Command>();

		// 명령어를 생성하여 보존
		Command cmd;
		cmd = new AddCommand();
		commands.put(cmd.getName(), cmd);
		cmd = new EchoCommand();
		commands.put(cmd.getName(), cmd);

		// 명령어 처리를 위한 로컬 변수
		String name = "";
		String params = "";

		// 콘솔에서 한 문자열을 받기 위한 변수
		BufferedReader br = new BufferedReader(
				new InputStreamReader(System.in));

		while(true){
			System.out.print("명령어 : ");
			// 명령어 획득
			name = br.readLine();
			if(name.equals("quit")) // 명령어가 quit이면 종료
				break;

			// 명령어 조회
			cmd = commands.get(name);
			System.out.println();
			if(cmd == null){ // 존재하지 않는 명령어
				System.out.println(
						"알 수 없는 명령어입니다.");
				continue;
			}else{
				System.out.print("파라메터 : ");
				// 명령어의 마라메터 획득
				params = br.readLine();
				cmd.setParameters(params);
				// 결과 출력
```

```
                                        System.out.println(cmd.execute());
                                    }
                                }
                            }

}

<실행>
명령어 : add

파라메터 : 1,2,3
6
명령어 : echo

파라메터 : kandroid
kandroid
명령어 : quit
```

Main 메소드의 명령어 처리부분에서 보는 것과 같이 명령어 인터페이스가 어떻게 구현되어져 있는지 상관없이 인터페이스를 통하여 동작되는 것을 볼 수 있다.

이러한 인터페이스 원리는 현재 재사용에 중점을 둔 컴포넌트 기반 개발의 기본 토대가 된다.

9. Enum Type

enum은 WEST, SOUTH, EARCH, NORTH와 같이 정해진 상수의 집합으로 구성되어진 필드 유형을 말한다. enum 유형은 enum 키워드를 사용하여 선언해주며 필드들은 상수이므로 보통 대문자를 사용하여 명명한다.

```
public enum Day {
        SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
  THURSDAY, FRIDAY, SATURDAY
}
```

enum 은 java.lang.Enum 클래스를 자동으로 상속받기 때문에 enum으로정의된 모든 상수는 java.lang.Enum 클래스의 모든 멤버를 사용할 수 있다.

enum의 정의는 먼저 상수 정의가 먼저 있어야 하며 ';' 을 구분으로 ';'다음부터 클래스와 같이 생성자와 필드(프로퍼티), 메소드를 정의할 수 있다.

```
public enum Planet {
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
```


```
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7); // ';'으로 구분

    // enum 정의

    private  double mass;   // in kilograms
    private  double radius; // in meters

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    private double mass()   { return mass; }
    private double radius() { return radius; }

    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
    public static void main(String[] args) {

        double earthWeight = 175;
        double mass = earthWeight/EARTH.surfaceGravity();
        for (Planet p : Planet.values())
          System.out.printf("Your weight on %s is %f%n",
                            p, p.surfaceWeight(mass));
    }
}
<실행>
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
```

10. 패키지(Package)

패키지는 name space를 관리하고 접근 보호를 위해 연관된 클래스, 인터페이스, Enum 유형을 그룹핑 한 것이다.

패키지 선언은 package 키워드를 사용하며 모든 자바 소스에서 최상위에 표시되어져야 한다. 그리고 일반적으로 클래스나 인터페이스의 이름과 충돌되지 않도록 소문자와 '.' 로 구분하여 표기한다.

```
package android.app;
...
public class Activity extends ContextThemeWrapper
...
```

패키지로 그룹핑되면 다른 패키지의 클래스 중 public으로 선언되지 않은 클래스는 사용할 수 없으며 public으로 선언된 클래스 내에서 public으로 선언되지 않은 변수와 메소드도 또한 접근할 수 없다.

```
package package1;

class NonPublicClass {
	public void print(){
		System.out.println("Non-public Class");
	}
}
```

```
package package1;

public class PublicClass {
	public void print(){
		System.out.println("Public Class");
	}
}
```

package1 패키지 안에는 NonPublicClass와 PublicClass가 있다. 그러나 NonPublicClass는 public으로 선언되지 않았기 때문에 package1 내의 자바 소스외에서는 인스턴스를 생성할 수 없다.

```
package package2;

import package1.*;

public class PackageMain {
	public static void main(String[] args){
		// 접근 불가로 컴파일 에러
		//NonPublicClass clazz = new NonPublicClass();

		PublicClass pclazz = new PublicClass();
		pclazz.print();

	}
}
```


그러나 NonPublicClass가 외부에 공개되지 않는 클래스라 할지라도 상속한 자식클래스가 public 이라면 외부에서 접근하여 NonPublicClass내에 정의된 메소드를 실행할 수 있다.

```
package package1;

public class NonPublicClassChild extends NonPublicClass {

}
```

```
package package2;

import package1.*;

public class PackageMain {
	public static void main(String[] args){
		// 접근 불가로 컴파일 에러
		//NonPublicClass clazz = new NonPublicClass();
		PublicClass pclazz = new PublicClass();
		pclazz.print();

		// public 으로 선언된 자식클래스는 사용 가능
		NonPublicClassChild nclazz = new NonPublicClassChild();
		nclazz.print();
	}
}
```

마지막으로 권장사항으로 패키지에 대해 설명할 것은 클래스나 인터페이스를 unique 하게 존재하도록 패키지를 정의하는 것인데 그러기 위해서는 다음의 예와 같이 조직의 도메인 명을 활용하여 패키지 이름을 정의하며 만약 도메인 명에 자바의 키워드가 포함되어져 있다면 '_'를 마지막에 붙여준다.

도메인 명	패키지명
kandroid.org	org.kandroid
google.com	com.google
free.font.int	int_.font.free

이상으로 자바를 통한 객체지향의 원리에 대해 설명하였다. 다음은 객체지향 설계의 기본 골격을 제공하는 디자인 패턴을 설명 하여 독자들이 안드로이드 애플리케이션 설계에 도움을 줄 뿐만 아니라 안드로이드 소스를 보는데 용이하도록 도움을 주고자 한다.

디지털 국가, 모바일 국가

김 은 영

얼마전 미국의 공영방송 PBS가 제작한 다큐멘터리 [DIGITAL NATION]을 봤다.

디지털 디바이스로 인한 사람들의 집중력과 기억력의 소실. 단편적인 사고방식의 확산, 글쓰기 능력의 저하. 비언어적 소통능력의 저하, 정신질환..등등의 다각적인 문제점을 지적했다. 세계의 똑똑이들이 모인다는 MIT나 스탠포드 대학의 강의실도 인상적이었다.
강의하는 교수들도 자신의 노트북으로 화면을 띄워놓고 강의를 하고, 그 강의를 듣는 학생들도 저마다 자신의 노트북과 스마트폰을 켜놓고 수업을 듣고 있었다. 수업 중에 노트북으로 스마트폰으로 메일을 체크하고, 답장을 보내고, 트위터에 글을 올리고 페이스북을 한다. 학생들은 모두가 한 목소리로 이제는 아이폰이나 블랙베리 없이 못산다고 말했다. 그리고 어른들은 그런 멀티태스킹을 못마땅해하지만, 아무런 문제가 없다고 말했다. 접속이 끊어지면 오히려 더 불안해 문제가 생길 것이라는 주장이다.
그래서 멀티태스킹이 세상을 바보로 만들 수도 있다고 생각하는 한 교수가 실제로 실험을 했다. 실험대상은 멀티태스킹에 완전 자신있다는 남자 대학생이었다. 결과는 그 많은 작업들 중 제대로 하는 것은 하나도 없었다. 집중력이 분산되기 때문이었다.

하지만 또 한 주장은 이렇다. 초등학교에 개인 노트북을 도입한 교장선생님인데, 이제 다음 세대들은 기억을 할 필요가 없다는 것이다. 필요한 정보는 검색하는 능력, 그것들은 필요한 논리대로 배열하고 구조화시킬 줄 아는 능력이 필요한 세상에 살 아이들이라는 것이다. 그렇기 때문에 오히려 디지털 디바이스를 더 친근하게 다루게 할 필요가 있다는 것이다. 누가 맞다, 틀리다를 따질 문제는 아닐 것이다. 적응하느냐, 못하느냐의 차이쯤?! 어차피 불가항력적으로 우리는 디지털 월드, 디지털 국가, 디지털 가정에서 살고 있기 때문이다.

아직은 노트북이 디지털 시대의 중심에 있는 듯하다. 하지만 이제 곧 노트북을 능가하는 스마트폰이 디지털 라이프의 중심이 될 것이다. 가장 개인적인 디지털 디바이스, 다른 어떤 디바이스들처럼 다른 사람과 공유하지 않고 혼자만의 기록과 추억을 담아내는 아주 특별한 나의 아바타같은 디바이스로 말이다.

디지털 라이프의 가능성은 앞으로도 더 많이 발전할 것이다. 모바일 라이프의 가능성도 무섭게 발달할 것이다. 이것은 확장성에 대한 이야기다. 세상의 모든 경험과 진실과 현실을 향한 나로부터의 확장, 나에게로의 확장. 그 중심에 스마트폰이 있는 것이다.

스마트폰 덕분에 세상 곳곳에서 '스마트(smart)'의 홍수다.
급기야는 글로벌 위기 이후 신성장 전략'으로 5가지 스마트 전략도 등장했다. 스마트 매니지먼트(Management),

스마트 IT, 스마트 그리드(Grid·친환경 에너지), 스마트 소비(Consumption·가치 지향적 소비), 스마트 규제(Regulation).
갈 길이 많다. 가까운 지 먼 지는 가보면 알게 될 것이다.
그러나 분명한 것은 세상은 이미 멀티 스피트 & 멀티 디렉션의 시대다.
너무나 다양한 산업군들이 다양한 속도로 다양한 방향을 향해 달리고 있다. 그 다양한 산업군 아래에 있는 크고 작은 기술과 기업들까지 모두가 서로 다른 속도로, 다른 방향으로 각자의 철학과 능력을 펼치고 있는 것이다. 그래서 우리는 시장의 속도와 방향, 경쟁사 혹은 경쟁자의 속도와 방향을 읽을 수 있는 리딩 파워를 길러야 한다. 더불어 나의 속도와 방향을 시기 적절하게, 다양하게 조정하고 조율할 수 있는 컨트롤 파워를 길러야 한다.

Multy Speed, Multy Direction, Multy Control...그것이 smart한 것이다.
Multy에 깊이까지 더한 T자형 인간이라면 금상첨화겠지만 말이다.

