핑크레드의 실전 모바일 자바(SK-VM) 프로그래밍

안녕하세요 핑크레드 입니다.

정말 오랜만에 다시 강좌를 쓰게 되었습니다. 언제나 강좌를 쓰는 목적은 특별히 없습니다. 많은 부분을 공유하고 새로운 것을 배우고 나면 누구나다 이런 것을 할수 있다라고 생각이 들고 결국은 누구나다 할수 있게되죠. 하지만 그 길을 알려주거나 하는 사람들은 없습니다. 단지 겪어오다 보면서 알게 되는 거죠. 누군가가 노하우를 글을 적어준다면 누구나 다 쉽게 접근을 할 수 있다는 것과 뻘짓을 안해도 된다는 것이 참 좋조. 언제나 두서 없이 말을 하다 보면 이렇게 끝내도 되는 것인가 하는 생각이 들지만 그깟이것 대충 의미만 전달하면 되는 것 아니겠습니까 …ㅋㅋㅋ

제가 강좌를 쓴지 몇 년이 흘렀습니다. 이는 곳 모바일의 발전이 많이 되었다는 것이죠. 새로운 플랫폼도 나왔고, 하지만 결국 기본은 다 똑같습니다. 위피를 하던 SKVM을 하던 혹은 다른 플랫폼을 하던 언어 혹은 함수만 다를 뿐이지 이에 들어가는 기술을 다 같기 때문이죠. 단지 예전과 지금이 다른 것은 하드웨어의 발전으로 이제는 더 좋은 코드를 코딩할수 있다는 것이죠. 저 또한 많은 노력을 하지는 않았지만 예전 보다는 체계적으로 할려고 노력하고 있습니다.

솔직히 요즘에는 프로그래밍을 너무나 잘하시는 분들이 많아서 이런 것을 쓴다는 것 자체가 우습기도 합니다. 미천한 실력에 조금 한다고 강좌나 쓰고 있으니 말이죠.
여튼 서두가 길었습니다. 한가지씩 집어보면서 느껴봅시다.
여러가지 주제가 있지만 지금은 머리속에 정리가 잘 안되네요. 일단은 상태값의 변화에 대한 이슈를 가지고 생각해 봅시다. 위피나 혹은 SKVM을 보면

WIPI C	WIPI Java	SKVM
paintClet	paint	paint
handleCletEvent	keyNotify	keyPressed
		keyReleased

위와 같습니다. 거의 같은 형태로 이벤트를 받고 있습니다. 그렇다면 우리는 해당 컨텐츠 마다 각각의 상태값이 존재하여 여러가지 경우를 처리해야 하는 경우가 있습니다. 이럴경우에는 어떻게 하는 것이 좋을까요?

1. 해당 경우마다 Canvas or Card를 생성시킨다.
2. 상태값을 주어서 그때마다 값을 변경시킨다.
3. 등등등….

1번과 같은 방법은 예전에는 별로 좋지 않았습니다. 메모리를 많이 먹기도 한다등등의 속도보다는 메모리 위주의 프로그래밍이여서 많이 쓰이지는 않았지만 요즘에는 크게 분리되어야할 경

우에는 사용하기도 합니다. 2번과 같은 방법은 한동안 많이 쓰인 방법입니다.

예전 강의에도 이러한 문제에 대해서 생각해 보곤 했습니다. 그때는 아래와 같은 방법을 이용했습니다. 예제로 WIPI Java의 경우를 들어보겠습니다.

```
final static int INTRO = 0;
final static int MAIN = 1;
final static int HELP = 2;
final static int END = 3;

int status;
...

protected void paint(Graphics g)
{
  switch(status)
   {
      case INTRO:
            break;
      case MAIN:
         break;
      case HELP:
         break;
      case END:
         break;
   }
}

protected boolean keyNotify(int type, int key)
{
  switch(status)
   {
      case INTRO:
            break;
      case MAIN:
```

```
            break;
        case HELP:
            break;
        case END:
            break;
    }
}
```

대략 이런식으로 프로그래밍을 많이 하였습니다. 그래서 이러면 너무나 해당 상태값을 찾기에 힘든점과 paint, key이벤트를 나누어서 살펴봐야 하는 번거로움으로 다음과 같이 프로그래밍 하였습니다.

```
final static int INTRO = 0;
final static int MAIN = 1;
final static int HELP = 2;
final static int END = 3;

final static int EVT_PAINT = 1;
final static int EVT_KEY_PRESSED = 2;
final static int EVT_KEY_RELEASED 3;

int status;
...

protected void paint(Graphics g)
{
    switch(status)
    {
        case INTRO:
            intro_EventProc(EVT_PAINT, g, 0);
                break;
        case MAIN:
            main_EventProc(EVT_PAINT, g, 0);
            break;
```

```
        case HELP:
              help_EventProc(EVT_PAINT, g, 0);
            break;
        case END:
            break;
    }
}

protected boolean keyNotify(int type, int key)
{
     int keyCode = Display.getGameAction(key);
     switch(type)
     {
         case EventQueue.KEY_PRESSED:
           main_EventProc(EVT_KEY_PRESSED, null, keyCode);
           break;
        case EventQueue.KEY_RELEASED:
           main_EventProc(EVT_KEY_RELEASED, null, keyCode);
           break;
     }
}

public void main_EventProc(int event, Graphics g, int keyCode)
{

   switch(status)
    {
        case INTRO:
            switch(event)
            {
                case EVT_PAINT:
                   break;
                case EVT_KEY_PRESSED:
                   break;
```

```
            case EVT_KEY_RELEASED:
              break;
          }
            break;
       case MAIN:
          break;
       case HELP:
          break;
       case END:
          break;
   }
}
```

기존의 분활된 이벤트를 한 화면에 볼수 있게 되는 방법입니다. 그런데 잘 보시면 paint, keyNotify의 내용이 약간 다릅니다. 이것은 두가지 방법을 나누어서 보여드린것입니다. keyNotify의 방식은 main_EventProc라는 것을 두어서 이곳에서 상태값을 전부 표현한 방법이죠. 이렇게 하면 main_EventProc는 상당히 길어지겠죠. 그래서 paint에서 처럼 해당 상태값을 구분하되 이벤트를 받는곳을 상태마다 주게 되어서 _EventProc의 내용도 적고 가독성을 높이게 하는 것이죠. 이해가 잘 안가시면 keyNotify는 상태값을 전부 main_EventProc에 넣은 것이고 paint는 상태값을 구분하고 각각의 이벤트를 _EventProc에 넣었다고 생각하시면 됩니다. 저도 처음에는 keyNotify와 같은 방식으로 했지만 결국 가독성의 문제로 paint방식으로 바꾸게 되었습니다. 아마 여기까지가 예전의 강좌 내용일 것입니다. 기억이 새롭내요..ㅜㅜ ㅋㅋㅋ 문제는 가독성뿐만 아니라 속도가 느려진다는 것을 알게 되었습니다. 우리도 모르고 있지만 switch, if문은 상당히 속도를 많이 잡아먹습니다. 아무리 정렬이 되도 구문을 비교한다는 것은 속도에 많은 영향을 끼친다고 할수 있습니다. 그래서 생각하게 된것인 interface입니다.

```
interface Viewer
{
   abstract void paint(Graphics g);
   abstract boolean keyNotify(int type, int key);
}
```

이렇게 Viewer클래스를 선언하고 각각의 상태값들이 해당 Viewer 클래스를 implements해서 구현되는 것이죠.

```
class INTRO implements Viewer
```

```
{
   public void paint(Graphics g){}
   public boolean keyNotify(int type, int key){}
}
```

위와 같이 모든 상태값이 클래스로 존재한다면..

```
Viewer viewer;

public void setViewer(Viewer viewer)
{
    this.viewer = viewer;
}

protected void paint(Graphics g)
{
    viewer.paint(g);
}

protected boolean keyNotify(int type, int key)
{
    viewer.keyNotify(typem key);
}
```

이렇게 간단하게 구현되면서 가독성도 좋아지고 속도도 향상됩니다. 어떤 분은 말씀하시겠죠. "이렇게 간단한 상태을 나누어서 클래스로 분활할 필요까지 있겠느냐?" 맞습니다. 분명 클래스가 많이 생기는 것은 단점이 되기도 하죠. 하지만 분할하는 크기는 프로그래밍 하시는 분들이 조절하셔야 할것입니다. 일일이 전부 상태값을 저렇게 분할한다면 클래스가 너무 많아지겠죠. 이러한 코드를 소개하는 이유는 이러한 방법도 있다라는 것을 알려드리는 것입니다. 응용으로써 게임에서는 케릭터의 각각의 상태가 있을것입니다. 그러한 상태값들을 저렇게 분할하게 된다면 속도도 빨라지고 가독성도 훨씬 좋아집니다. 간단한 게임을 만들어서 테스트해본 결과 만족하고 있습니다.

프로그래밍시에는 메모리 절약도 중요하고 속도도 중요합니다. 하지만 가독성 또한 매우 중요한 부분이 되었습니다. 이미 다른 분야에서는 워낙 하드웨어가 뒷받침되기 때문에 좋은 코드를

짤려고 하고 있지만 모바일쪽은 그렇지 못하기 때문에 상당히 코드가 안좋았지만 이제는 할만하다고 생각됩니다. 아니 예전부터 할만했지만 실천을 하지 않고 있었던 것이죠. 이렇게 한다고 해서 더 많은 메모리와 속도가 떨어지는 것도 아니기 때문에 좋은 방법이라고 생각됩니다. 이러한 부분들은 제가 리펙토링이라는 책을 한번 보면서 좋은 코드는 속도와 메모리에 이점이 많다라는 부분을 실감한 부분입니다.

이제는 프로그래밍 시에는 나를 위한 프로그래밍이 아니라 남을 위한 프로그래밍을 해야합니다. 페어프로그래밍, 리팩토링등등이 전부 이러한 이슈를 잘 해결하기 위한 방법론이라고 할 수 있습니다. 왜 남을 위한 프로그래밍이냐 한번 잘 생각해 보십시오.

다음 강좌는 PinkComponent를 예전 버전 보다 업그레이드된 모습으로 정리해서보여드리겠습니다. 모든 컴포넌트는 모바일 프로그래밍의 기본으로 어디서든지 사용이 가능하며 어떤 플랫폼에서도 함수만 변경하면 되기 때문에 매우 효과적입니다. 단 몇몇 컴포넌트들은 너무 해당 플랫폼에 맞추어져 있어서 다시 만들어야 하는 것도 있습니다.

요즘에는 플랫폼이 너무나 많이 있습니다. 위피, skvm, Gnex, brew등등 너무나 많아서 무엇을 공부해야하는지도 참 막막한 시대입니다. 이럴때는 특히 기본기를 탄탄히 다진다면 어는 플랫폼이든 적응하는데 수월하다고 생각됩니다.

그럼 이만..즐건 하루 되십시오..핑크레드.

닉네임 : 핑크레드
성격 : 왕소심, 왕삐짐, 이중인격, 수다쟁이
이메일 : pinkred@hanafos.com

추신 : 강좌를 다른 사이트에 올리시는 것은 자유입니다..단지 저한테 메일 한 통 이라도 보내주시고 올리시면 감사하겠습니다. 언제나 좋은 하루 되십시오.