http://cranix.net/365[**android CustomView 생성하기**](http://cranix.net/365)

[알짜정보/Android](http://cranix.net/category/%EC%95%8C%EC%A7%9C%EC%A0%95%EB%B3%B4/Android)

안드로이드에서 기본적으로 지원하지 않는 UI 를 만들때 CustomView 를 사용합니다.

이러한 CustomView 의 기본적인 작성방법을 알아보도록 하겠습니다.

CustomView 는 “android.view.View” 클래스를 상속해서 만들어 집니다.

기본적으로 onDraw() 메소드만 재정의해서 xml 에 view 태그만 추가하면 오류없이 출력되는것을 볼 수 있습니다.

이번 포스트에서는 간단히 클릭하면 반응하는 CustomView 를 만들어 보도록 하겠습니다.

먼저 CustomView 소스를 확인해 보도록 하겠습니다.

**- CustomView.java**

package net.cranix.android.customviewtest;  
  
import android.content.Context;  
import android.graphics.Canvas;  
import android.graphics.Color;  
import android.graphics.Paint;  
import android.util.AttributeSet;  
import android.util.Log;  
import android.view.KeyEvent;  
import android.view.MotionEvent;  
import android.view.View;  
  
public class CustomView extends View {  
   
 private String text = null;  
 private int backgroundColor = Color.RED;  
  
 private String tempText;  
   
   
 // 속성이 없는 생성자는 소스상에서 직접 생성할때만 쓰인다.   
 public CustomView(Context context) {  
 super(context);  
 Log.w(Constants.TAG,"CustomView("+context+")");  
 }  
 /\*  
 \* 리소스 xml 파일에서 정의하면 이 생성자가 사용된다.  
 \*   
 \* 대부분 this 를 이용해 3번째 생성자로 넘기고 모든 처리를 3번째 생성자에서 한다.  
 \*/  
 public CustomView(Context context,AttributeSet attrs) {  
 this(context,attrs,0);  
 Log.w(Constants.TAG,"CustomView("+context+","+attrs+")");  
 }  
   
 /\*  
 \* xml 에서 넘어온 속성을 멤버변수로 셋팅하는 역할을 한다.  
 \*/  
 public CustomView(Context context,AttributeSet attrs,int defStyle) {  
 super(context,attrs,defStyle);  
   
 this.text = attrs.getAttributeValue(null,"text");  
   
 Log.w(Constants.TAG,"CustomView("+context+","+attrs+","+defStyle+"),text:"+text);  
 }  
   
 /\*  
 \* xml 로 부터 모든 뷰를 inflate 를 끝내고 실행된다.  
 \*   
 \* 대부분 이 함수에서는 각종 변수 초기화가 이루어 진다.  
 \*   
 \* super 메소드에서는 아무것도 하지않기때문에 쓰지 않는다.  
 \*/  
 @Override  
 protected void onFinishInflate() {  
 setClickable(true);  
 Log.w(Constants.TAG,"onFinishInflate()");  
 }  
   
 /\*  
 \* 넘어오는 파라메터는 부모뷰로부터 결정된 치수제한을 의미한다.  
 \* 또한 파라메터에는 bit 연산자를 사용해서 모드와 크기를 같이 담고있다.  
 \* 모드는 MeasureSpec.getMode(spec) 형태로 얻어오며 다음과 같은 3종류가 있다.  
 \* MeasureSpec.AT\_MOST : wrap\_content (뷰 내부의 크기에 따라 크기가 달라짐)  
 \* MeasureSpec.EXACTLY : fill\_parent, match\_parent (외부에서 이미 크기가 지정되었음)  
 \* MeasureSpec.UNSPECIFIED : MODE 가 셋팅되지 않은 크기가 넘어올때 (대부분 이 경우는 없다)  
 \*   
 \* fill\_parent, match\_parent 를 사용하면 윗단에서 이미 크기가 계산되어 EXACTLY 로 넘어온다.  
 \* 이러한 크기는 MeasureSpec.getSize(spec) 으로 얻어낼 수 있다.  
 \*   
 \* 이 메소드에서는 setMeasuredDimension(measuredWidth,measuredHeight) 를 호출해 주어야 하는데  
 \* super.onMeasure() 에서는 기본으로 이를 기본으로 계산하는 함수를 포함하고 있다.  
 \*   
 \* 만약 xml 에서 크기를 wrap\_content 로 설정했다면 이 함수에서 크기를 계산해서 셋팅해 줘야한다.  
 \* 그렇지 않으면 무조껀 fill\_parent 로 나오게 된다.  
 \*/  
 @Override  
 protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  
   
 // height 진짜 크기 구하기  
 int heightMode = MeasureSpec.getMode(heightMeasureSpec);  
 int heightSize = 0;  
 switch(heightMode) {  
 case MeasureSpec.UNSPECIFIED: // mode 가 셋팅되지 않은 크기가 넘어올때  
 heightSize = heightMeasureSpec;  
 break;  
 case MeasureSpec.AT\_MOST: // wrap\_content (뷰 내부의 크기에 따라 크기가 달라짐)  
 heightSize = 20;  
 break;  
 case MeasureSpec.EXACTLY: // fill\_parent, match\_parent (외부에서 이미 크기가 지정되었음)  
 heightSize = MeasureSpec.getSize(heightMeasureSpec);  
 break;  
 }  
   
 // width 진짜 크기 구하기  
 int widthMode = MeasureSpec.getMode(widthMeasureSpec);  
 int widthSize = 0;  
 switch(widthMode) {  
 case MeasureSpec.UNSPECIFIED: // mode 가 셋팅되지 않은 크기가 넘어올때  
 widthSize = widthMeasureSpec;  
 break;  
 case MeasureSpec.AT\_MOST: // wrap\_content (뷰 내부의 크기에 따라 크기가 달라짐)  
 widthSize = 100;  
 break;  
 case MeasureSpec.EXACTLY: // fill\_parent, match\_parent (외부에서 이미 크기가 지정되었음)  
 widthSize = MeasureSpec.getSize(widthMeasureSpec);  
 break;  
 }  
  
   
 Log.w(Constants.TAG,"onMeasure("+widthMeasureSpec+","+heightMeasureSpec+")");  
   
 setMeasuredDimension(widthSize, heightSize);  
 }  
   
   
 /\*  
 \* onMeasure() 메소드에서 결정된 width 와 height 을 가지고 어플리케이션 전체 화면에서 현재 뷰가 그려지는 bound 를 돌려준다.  
 \*   
 \* 이 메소드에서는 일반적으로 이 뷰에 딸린 children 들을 위치시키고 크기를 조정하는 작업을 한다.  
 \* 유의할점은 넘어오는 파라메터가 어플리케이션 전체를 기준으로 위치를 돌려준다.  
 \*   
 \* super 메소드에서는 아무것도 하지않기때문에 쓰지 않는다.  
 \*/  
 @Override  
 protected void onLayout(boolean changed, int left, int top, int right,  
 int bottom) {  
 Log.w(Constants.TAG,"onLayout("+changed+","+left+","+top+","+right+","+bottom+")");  
 }  
   
   
 /\*  
 \* 이 뷰의 크기가 변경되었을때 호출된다.  
 \*   
 \* super 메소드에서는 아무것도 하지않기때문에 쓰지 않는다.  
 \*/  
 @Override  
 protected void onSizeChanged(int w, int h, int oldw, int oldh) {  
   
 Log.w(Constants.TAG,"onSizeChanged("+w+","+h+","+oldw+","+oldh+")");  
 }  
   
   
 /\*  
 \* 실제로 화면에 그리는 영역으로 View 를 상속하고 이 메소드만 구현해도 제대로 보여지게 된다.  
 \*   
 \* 그릴 위치는 0,0 으로 시작해서 getMeasuredWidth(), getMeasuredHeight() 까지 그리면 된다.  
 \*   
 \* super 메소드에서는 아무것도 하지않기때문에 쓰지 않는다.  
 \*/  
 @Override  
 protected void onDraw(Canvas canvas) {  
 final Paint p = new Paint();  
 p.setColor(backgroundColor);  
 canvas.drawRect(0,0,getMeasuredWidth(),getMeasuredHeight(), p);  
 if (text != null) {  
 p.setColor(Color.BLACK);  
 canvas.drawText(text, 10, 15, p); // 왼쪽 아래를 0,0 으로 보고있음  
 }  
 Log.w(Constants.TAG,"onDraw("+canvas+")");  
 }  
   
   
 /\*  
 \* 현재 view 가 focus 상태일때 key 를 누르면 이 메소드가 호출됨.  
 \* 즉 이 메소드를 사용하려면 setFocusable(true) 여야함.   
 \*   
 \* 그리고 super 메소드에서는 기본적인 키 작업(예를들면 BACK 키 누르면 종료)을 처리하기 때문에 일반적으로 return 시에 호출하는게 좋다.  
 \* 만약 기본적인 작업을 하지않게 하려면 super 함수를 호출하지 않아도 된다.  
 \*   
 \* 다른 event 메소드들도 유사하게 동작한다.  
 \*/  
 @Override  
 public boolean onKeyDown(int keyCode, KeyEvent event) {  
 Log.w(Constants.TAG,"onKeyDown("+keyCode+","+event+")");  
 return super.onKeyDown(keyCode, event);   
 }  
   
 /\*  
 \* 이 view 에 touch 가 일어날때 실행됨.  
 \*   
 \* 기본적으로 touch up 이벤트가 일어날때만 잡아내며   
 \* setClickable(true) 로 셋팅하면 up,move,down 모두 잡아냄  
 \*/  
 @Override  
 public boolean onTouchEvent(MotionEvent event) {  
 Log.w(Constants.TAG,"onTouchEvent("+event+")");  
 switch(event.getAction()) {  
 case MotionEvent.ACTION\_UP:  
 backgroundColor = Color.RED;  
 text = tempText;  
 break;  
 case MotionEvent.ACTION\_DOWN:  
 backgroundColor = Color.YELLOW;  
 tempText = text;  
 text = "Clicked!";  
 break;  
 case MotionEvent.ACTION\_MOVE:  
 backgroundColor = Color.BLUE;  
 text = "Moved!";  
 break;  
 }  
 invalidate();  
 return super.onTouchEvent(event);  
 }  
   
   
 public String getText() {  
 return text;  
 }  
 public void setText(String text) {  
 this.text = text;  
 }  
}

**- 크기 계산하기**

여기서 중요한 메소드는 **onMeasure()** 메소드 입니다.

이 메소드는 뷰의 전체 크기를 정하는 메소드 인데 안드로이드의 크기 정하는 방법에 따라 구현법이 달라져야 합니다.

안드로이드 레이아웃 xml 파일에서 크기를 지정하는 방법은 4가지가 있습니다.

   - fill\_parent (상위 View 의 크기에 따름)

   - match\_parent (상위 View 의 크기에 따름)

   - fixed (100px 와 같이 픽셀로 박아놨을때)

   - wrap\_content (현재 뷰의 내용에 따름)

이렇게 4가지 방법의 특성에 따라서 넘어오는 크기의 종류는 3가지로 구분됩니다.

   - MeasureSpec.EXACTLY : fill\_parent, match\_parent, fixed 와 같이 상위에서 이미 결정되어버린 크기가 넘어올때 선택됩니다.

   - MeasureSpec.AT\_MOST : wrap\_content 를 선택했을때 선택됩니다.

   - MeasureSpec.UNSPECIFIED : xml 에 의하지 않고 소스상에서 직접 넣었을 때 나옵니다.

여기서 EXACTLY 과 UNSPECIFIED 는 외부에서 크기가 구해져서 내려오는 것이기 때문에 따로 계산할 것이 없으나 AT\_MOST 는 내부적으로 크기계산을 해 주어야 합니다.

위의 소스에서는 간단하게 100,20 으로 박아놨지만 실제로 CustomView 를 구현하게 된다면 뷰의 특성에 따라 구현이 달라져야 할 것입니다.

**- xml 에서 파라메터 받아내기**

안드로이드 리소스 xml 에서 파라메터를 받아내려면 위 소스의 3번째 생성자에 있는것 처럼 아래와 같은 구문을 써야 합니다.

this.text = attrs.getAttributeValue(null,"text");

**- xml 파일 구성하기**

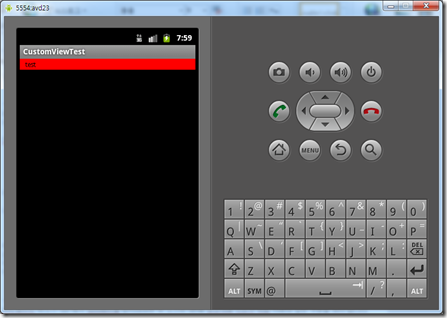
이렇게 만든 CustomView 를 xml 파일에서 사용하려면 아래와같은 xml 구성이 필요합니다.

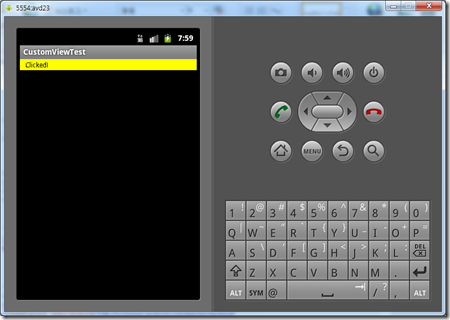
<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:orientation="vertical"  
 android:layout\_width="fill\_parent"  
 android:layout\_height="fill\_parent"  
 >  
 **<view class="net.cranix.android.customviewtest.CustomView"  
 android:layout\_width="fill\_parent"  
 android:layout\_height="wrap\_content"  
 text="test"  
 />**</LinearLayout>

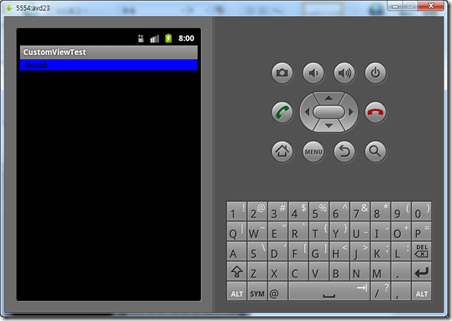
**- 실행해 보기**

이렇게 구성된 뷰를 Activity 에 넣고 실행해 보면 아래와 같은 화면이 나옵니다.

마우스를 클릭,이동 할때마다 색깔이 변경되는것을 볼 수 있습니다.

[](http://cfile10.uf.tistory.com/image/165A164B4D26C8D421307E)

[](http://cfile25.uf.tistory.com/image/162F2A474D26C8D50946EF)

[](http://cfile9.uf.tistory.com/image/166C3C494D26C8D62042E2)

아래의 순서로 작성하면 됨.

1. View나 View subclass를 확장

2. 필요한 메소드들을 오버라이드

끝

오버라이드할 메소드 리스트

1. 생성관련

  생성자 : code상으로 생성되는 생성자와 layout파일로부터 inflate되면서 생성되는 경우에 호출되는 생성자 두가지가 있으며 두번째 생성자는 layout파일에 정의되어 있는 속성들을 파싱하여 적용하는 부분이 추가로 코딩되어야 한다.

  onFinishInflate : layout XML파일로부터 뷰와 그의 모든 차일드들이 inflate된 후에 호출됨. 이 때 추가로 어떤 작업을 하고자 하는 경우 여기서 하면 된다.

2. Layout관련

  onMeasure(int, int) : 부모에서 계산된 width, height 스펙값내에서 뷰와 차일드들의 크기를 계산하여 적용한다.

  onLayout(boolean, int, int, int, int) : 뷰가 size와 postion을 모든 차일드에 지정해야 할경우 호출된다.

  onSizeChanged(int, int, int, int) : 뷰의 size가 변경된 후 호출된다.

3. 그리기관련

  onDraw(Canvas) : View가 내용을 다시 그려야 할 때 호출된다.

4. Event 처리

  onKeyDown(int, KeyEvent) : 새로운 key event가 발생하였을 때 호출된다.

  onKeyUp(int, KeyEvent) : key up event가 발생하였을 때 호출된다.

  onTrackballEvent(MotionEvent) : 트랙볼 모션이벤트가 발생하였을 때 호출된다.

  onTouchEvent(MotionEvent) : 터치스크린의 모션이벤트가 발생하였을 때 호출된다.

  onFocusChanged(boolean, int, Rect) :  View가 포커스를 얻거나 잃었을 때 호출된다.

  onWindowFocusChanged(boolean) : View를 포함하고 있는 윈도우가 포커스를 얻거나 잃었을 때 호출된다.

5. Attach관련

  onAttachedToWindow() : View가 Window에 attach되었을 때 호출된다.

  onDetachedFromWindow() : View가 Window로부터 detach되었을 때 호출된다.

  onWindowVisibilityChanged(int) : View를 포함하는 window의 visibility가 변경되었을 때 호출된다.