안드로이드 앱 개발 강의 계획서

**강의 목표**

* 안드로이드 앱 제작의 기본 요소들을 소개하고 이해한다.
* 객채 지향 프로그래밍의 기본을 배운다.

**서론**

* 이 강의는 세 파트로 나뉘어져있다. 이 보고서에선 각 부분을 파트1, 파트2, 파트3이라고 칭한다.
* 수업은 파트1, 2, 3 순으로 진행된다.
* 강의 내용의 난이도는 파트 3 > 파트 2 > 파트 1 순으로 어렵다.
* 파트 3은 파트 1과 파트 2의 내용을 합쳐서 심화한다.

**파트 1 – 구체**(球體)

1. 학습 목표: a. 안드로이드 UI의 기본인 Activity에 대해 배운다.

b. 안드로이드 SDK에서 간단한 그래픽을 구현한다.

c. 안드로이드 센서를 이용하는 방법을 배운다.

2. 수업 내용:

기본

모바일 앱을 제작한다는건 기본적으로 유저가 앱의 입력 체계를 통해 도달할 수 있는 여러가지 페이지(張)들을 배치 해놓는 것이다. 이 각 페이지들을 안드로이드는 Activity라고 칭한다. 한 Activity는 보통 한 자바 class안에 담긴다. 다음은 안드로이드 스튜디오를 이용해 기본적인 앱 제작 준비를 하는 순서이다.

현재 1.2.1.1버전인 안드로이드 스튜디오를 처음 실행해 메뉴에서 File > New > New Project를 선택한다. 새로운 프로젝트를 만들기 위해 다이얼로그가 나타난다. 아무것도 바꾸지 않고 계속 Next를 누르며 Finish버튼까지 도달한다. 이 단계에서 Activity의 이름을 정해야 한다. 이 또한 바꾸지 않고 default 그대로 MainActivity로 놔둔 후 Finish를 누른다. 프로젝트가 처음 생성되었을 때, 프로젝트에 포함된 파일은 세가지이다:

* + app/src/main/java/com/사용자 이름 /myapplication/MainActivity.java
  + app/src/main/res/layout/activity\_main.xml
  + app/src/main/AndroidManifest.xml

이 세 파일 외에는 중요하지 않다. 다음은 각 파일에 대한 설명이다.

* MainActivity.java: 하나의 Activity이다. 앞서 언급했든 한 Activity는 앱의 한 페이지를 표시한다. 현재 이 앱에는 Activity가 MainActivity하나 뿐이며, 앱이 실행됐을 때 MainActivity가 나타난다.
* activity\_main.xml: 안드로이드에서 한 Activity와 그 안의 요소들이 사용자에게 어떤식으로 보여지는지는 태그 언어인 xml파일을 이용한다. 현재 Hello World라는 string을 표시하는 “TextView”와 그 textview를 담고있는 RelativeView 두 개가 정의돼있는 것을 볼 수 있다. 이들은 Activity를 구성하는 요소들이며, “View”라고 부른다. 개발자는 여러 종류의 View들을 Activity에 마음대로 배치할 수 있다. 우리는 파트1에서 ImageView를 사용해 이미지를 그릴것이다.
* AndroidManifext.xml: 개발하는 앱의 전반적인 요약을 제공하는 파일이다. 개발자의 앱이 요구하는 user permission(wifi access, Bluetooth access, camera access등)과 앱에 포함될 Activity들의 이름이 명시되어있다.

다음은 개발자가 코딩한 한 Activity를 안드로이드가 어떤 방식으로 화면에 나타내는지 설명하겠다.

앞서 말했듯 Activity를 구선하는 것은 두 가지다. 화면의 layout을 정의한 xml 문서와 그 layout이 어떻게 사용자와 상호작용하는지 명시하는 java 파일이다. 우리의 프로젝트에서 전자는 activity\_main.xml이며 후자는 MainActivity.java이다. 사용자가 앱을 처음 실행할 때, 안드로이드 운영체제는 개발자가 표시한 첫번째 activity를 로드한다 (현재 activity는 MainActivity.java 한 개 뿐이므로 시작 activity는 자동으로 그 액티비티가 된다).

모든 프레임워크가 그렇겠지만, 특히 모바일 앱 개발은 순수한 창작보다는 여권 문서 작성과 더 비슷하다. 운영체제가 요구하는 변수와 함수들을 빈 칸 체우기 식으로 작성해나가는것이 바람직하다. 한 Activity를 정의할 때, 개발자는 그 Activity의 java 파일 속에 다음 7가지 함수를 선언해야 한다.

onCreate, onStart, onReStart, onResume, onPause, onStop, onDestroy.

이 7개의 함수는 시스템이 이 화면을 처음 실행하거나, 멈추거나, 프로세스를 종료시키거나, 다시 시작했을 때 일어나야 하는 일들을 정의한다. 이 7개의 method들을 각각 명백하게 정의하지 않아도 오류는 뜨지 않는다. 이 java 파일이 Activity라는 걸 안 이상, 운영체제는 내장되어있는 위의 함수들로 Activity를 다룰 수 있다(MainActivity class의 superclass인 Activity 속에 이미 정의되어있다).

현재 MainActivity의 onCreate 함수가 override되어있다. onCreate 속에는 super.onCreate와 setContentView함수가 불린다. super.onCreate는 MainActivity가 상속받은 Activity의 onCreate을 부른다. 여기서 onCreate의 인자인 Bundle은 현재 신경쓰지 않는다. 안드로이드 스튜디오 버젼에 따라 MainActivity가 Activity가 아닌 ActionBarActivity를 extend하고 있다면 ActionBarActivity를 Activity로 고친다. 다음으로 setContentView 함수는 Activity를 구성하는 앞서 언급한 두 요소 (xml 레이아웃과 java 코드)를 연결해준다. R.layout.activity\_main이란 파일이 MainActivity란 Activity의 레이아웃을 담당한다는 뜻의 함수 호출이다. 여기서 “R”이란 이 프로젝트 패키지를 뜻한다. R이 선언되지 않았다는 에러가 뜬다면 파일 맨 꼭데기에 패키지 선언을 해주지 않았다는 뜻이다.

본문

우리의 앱은 화면에 구체를 만들어 단말기의 가속 센서에 의해 구체를 움직이게 만들 것이다. 그러기 위해 앱을 실행하자마자 준비해야할 사안은 다음과 같다:

1. 기기 화면의 가로, 세로 크기를 변수에 저장한다.
2. 구체 이미지가 표시될 수 있는 공간을 정의한다.
3. 기기에 탑재된 센서의 변화를 감지하는 코드를 정의한다.
4. 감지한 변화에 따라 공을 움직일 수 있도록 공과 데이터를 연결한다.

이러한 셋업은 앱이 실행되자 마자 준비해야 하기 때문에, MainActivity의 onCreate method를 override한다. Override한 onCreate 속에서 DisplayMetrics 클래스를 이용해 화면의 크기를 알아낸다.

*// Screen specs.*DisplayMetrics displaymetrics = **new** DisplayMetrics();  
getWindowManager().getDefaultDisplay().getMetrics(displaymetrics);  
*screenWidth* = displaymetrics.**widthPixels**;  
*screenHeight* = displaymetrics.**heightPixels**;

다음 코드는 구체의 이미지를 표시할 수 있는 ImageView를 Activity안에 정의한다. ImageView란 앞서 언급한 TextView, RelativeView와 같은 사용자 인터페이스의 일원이다.

**public class** BallView **extends** ImageView {  
  
 **public** BallView(Context context) {  
 **super**(context);  
 }  
  
 @Override  
 **protected void** onDraw(Canvas canvas) {

**super**.onDraw(canvas);  
 **for** (Ball ball : **balls**) {  
 ball.drawBall(canvas);  
 }  
 }  
}

MainActivity속에 inner class로 ImageView의 변형본인 BallView를 선언한다. 여기서 중요한건, Activity의 layout은 반드시 xml 파일로 정의되어야 하는게 아니라는 점이다.

BallView의 onDraw 함수를 살펴보자. Canvas 종류의 인자를 받고, balls 라는 리스트 속의 Ball 종류의 구성원을 불러 drawBall이란 함수를 호출한다. 다음은 이 코드에 대한 설명이다.

* onDraw: 어떤 View의onDraw함수는 사용자 UI의 일부인 그 View가 화면에 그려질 때 불리는 함수이다. BallView는 View 종류 중 하나인 ImageView 클래스를 상속한다. BallView가 override한 ImgeView의 onDraw는 BallView가 화면에 어떤식으로 나타나야하는지 시스템에세 알려준다. super.onDraw란 코드는 기본적인 이미지뷰의 onDraw를 불러주기 위해 쓴다. (이 부분을 지워도 코드는 작동한다)
* balls: 우리가 그린 공들을 담을 ArrayList이다. MainActivity에 static variable로 선언해준다.
* Ball: 추후에 다른 자바 클래스로 작성할 공을 정의하는 클래스이다.

이 BallView는, onCreate속 다음 코드에서 UI의 레이아웃을 점령한다.

*// Set up views.***ballView** = **new** BallView(**this**);  
**ballView**.setBackgroundColor(randomColor());setContentView(**ballView**);

**이처럼 미리 정의된 Layout요소를 사용자가 볼 수 있는 화면에 불러들이는 것을 Layout을 inflate한다고 말한다. setContentView 메소드로 inflation이 이루어진다.**

위 코드에서 두번째 줄의 randomColor() 함수는 파일 맨 밑에 정의한 랜덤 색 결정 함수이다.

**private int** randomColor() {  
 Random rand = **new** Random();  
 **int** r = rand.nextInt(255);  
 **int** g = rand.nextInt(255);  
 **int** b = rand.nextInt(255);  
 **return** Color.*rgb*(r, g, b);  
}

자바의 Random과 안드로이드의 Color 라이브러리를 이용해, 무작위로 얻은 0부터 255의 값을 rgb형태(int)로 변환해 리턴한다.

다음은 공들을 가속 센서에 변화에 따라 움직일 수 있도록 기기의 센서를 불러온다. 이 과정 또한 onCreate속에서 이루어진다.

*// Set up sensors.***sensorManager** = (SensorManager) getSystemService(***SENSOR\_SERVICE***);  
**accelerometer** = **sensorManager** .getDefaultSensor(Sensor.***TYPE\_ACCELEROMETER***);

sensorMagager은 MainActivity의 class variable로 선언된 변수이다. 시스템에 대한 getSystemService 함수로 기기의 센서 제어 도구인 센서 메니저를 불러온다. 여러가지 매니저가 존재하기 때문에, (SensorManager)로 캐스팅 해준다. 기기의 sensor manager을 얻은 후, 가속 센서를 access하는 함수는 getDefaultSensor이다. accelerometer역시 이전에 class variable로 선언되었다.

센서의 변화를 감지해 UI의 변화를 주기 위해

1. 감지
2. UI 업데이트

이 두 단계가 필요하다. 변화 감지를 위해, sensor의 변화에 반응하는 listner을 설정한다. 안드로이드에 Lisnter은 여러가지가 있다. 시스템의 통보를 받는 broasdcastReciever, 사용자의 터치를 감지하는 onTouchListner, 센서의 변화를 감지하는 sensorEventListner등이 있는데, 여기서는 SensorEventListner을 이용한다. SensorEventListner은 상속받아야 하는 인터페이스이기 때문에, Mainactivity가 SensorEvnetListner 인터페이스를 implement하도록 바꿔준다.

**public class** MainActivity **extends** Activity **implements** SensorEventListener

위의 코드로 인하여 MainActvity는 센서의 변화에 민감해진다. SensorEventListner 인터페이스를 이용하려면 onAccuracyChanged, onSensorChanged두 메소드를 정의해야한다. 전자는 센서의 정확도, 후자는 센서의 데이터가 변했을 때 자동으로 불러지는 메소드이다. onAccuracyChanged는 최소한의 구현만 하고, onSensorChanged를 변형해 센서값의 변화에 따라 UI업데이트를 실행한다.

@Override  
**public void** onSensorChanged(SensorEvent event) {  
 **if** (event.**sensor**.getType() == Sensor.***TYPE\_ACCELEROMETER***) {  
 **int** xAcc = (**int**) event.**values**[0];  
 **int** yAcc = (**int**) event.**values**[1];  
 **for** (Ball ball : **balls**) {  
 ball.moveBall(*screenWidth*, *screenHeight*, xAcc, yAcc);  
 }  
 **ballView**.invalidate();  
 }  
}

위 코드에서, onSensorChanged는 SensorEvent 타입의 인자를 받는다. SensorEvent는 갖가지 종류의 센서의 변화로 일어난 모든 정보를 말한다. 따라서 이 변화를 실질적인 데이터값으로 변환해주는 과정이 필요하다. 이에 앞서 SensorEvent를 가속 센서가 trigger했는지 확인해준다. 만약 event.sensor.getType()의 리턴값이 가속도 센서가 맞다면, event.values라는 Array는 세 값을 갖고 있을 것이다. 0번 인덱스에서는 x축 방향의 변화, 1번 인덱스에는 y축 방향의 변화, 그리고 2번 인덱스에는 z 축 방향의 변화가 저장되어있다. 우리는 기기의 수평 기준 기울기를 알아내고 싶기 때문에 x축과 y축 기울기를 int로 캐스팅해 각각 xAcc, yAcc변수에 저장한다. 그 후, 화면에 존재하는 모든 ball을 (balls라는 arraylist안에 담겨있다) 센서의 변화에 따라 이동시킨다. 공은 이동하는건 moveBall이라는 Ball 클래스에 정의된 함수를 이용한다.

마지막으로 invalidate함수를 부른다. 이 함수는 ballView의 instance method라는점이 중요하다. Instance method는 Static method와 반대되는 개념이며, 클래스에 종속된 메쏘드가 아닌 각 개체마다 소유하는 메쏘드이다. 이 함수의 역할은 이 함수를 부르는 instance를 다시 그리는 것이다. 따라서, View클래스의 instance들에게 주어지는 메쏘드이다. 공들의 위치가 바뀌었기 때문에, invalidate을 불러 UI를 업데이트 해준다.

화면의 공을 센서에 맞춰 움직일 모든 준비가 끝났다. 다음은 공을 생성하는 과정을 설명한다. 이 예시에는 사용자가 화면을 터치하면 새로운 공이 생성되도록 설정한다.

onCreate에서 터치에 반응하는listner을 설정해준다. 앞서 언급한 onTouchEventListner을 BallView의 instance인 ballView에 부착해주는 작업이다.

**ballView**.setOnTouchListener(**new** OnTouchListener() {  
 @Override  
 **public boolean** onTouch(View v, MotionEvent event) {  
 **int** action = event.getAction();  
 **int** x = (**int**) event.getX();  
 **int** y = (**int**) event.getY();  
 **switch**(action){  
 **case** MotionEvent.***ACTION\_DOWN***:  
 *// Make new ball inside screen boundary.* x = (x + *ballSize* >= *screenWidth*) ? *screenWidth* - *ballSize* : x;  
 y = (y + *ballSize* >= *screenHeight*) ? *screenHeight* - *ballSize* : y;  
 **ball** = **new** Ball(x,y,*ballSize*);  
 **balls**.add(**ball**);  
 **ballView**.invalidate();  
 **break**;  
 **default**:  
 **break**;  
 }  
 **return true**;  
 }});

위의 코드에서 onTouchListner의 instance를 만드는데 필요한 최소한의 메쏘드인 onTouch를 정의한다. MotionEvent도 SensorEvent와 같이, 일어난 사건에 대한 여러가지 정보를 담고있는 번들이다. getAction 함수를 통해 이 onTouch가 발동되는 시점이 손가락이 화면을 터치한 순간임을 확인하고, getX, getY함수로 손가락의 좌표를 구한다.

여기서 Ball의 instanace를 만들기 전, x와 y점을 기준으로 공을 생성하면 공의 사이즈에 따라 공의 일부가 화면 밖에 만들어질 수 있다. 따라서 x좌표와 y좌표를 보정해 공이 화면에 생성되도록 보정 작업을 한다.

Static Variable로 정의한 ballSize를 인자로 받는 Ball의 instance를 생성한 후, 구체 instance를 모아놓을 balls에 추가한다. Invalidate()을 호출해 UI를 다시 그려준다.

여기서, ballSize는 픽셀 갯수를 나타내기 때문에, 사용하는 기기의 해상도에 따라 실제 공의 크기가 달라질 수 있다.

마지막으로, 앞서 추상적으로만 소개된 Ball 클래스에 대한 설명이다. Ball 클래스를 마지막에 설명하는 이유는, 의도에 따라 천차만별로 구현될 수 있는 클래스이기 때문이다. 이 예시에서 사용하는건 구체를 나타내는데 필요한 최소한이고, 필요에 따라서 collision detection, bounce등을 추가할 수 있다.

안드로이드 스튜디오에서 File > New > Java Class 를 선택해 Ball.java라는 새로운 자바 파일을 생성한다. 이 파일은 자동으로 프로젝트의 src 디렉토리에 추가된다. 우선 Ball 클래스에 들어가야할 필수 요소를 나열해본다.

1. Ball의 현재 위치. 처음 생성되는 위치는 constructor의 인자로 받으야 하겠고, 업데이트되는 위치는 instance variable로 저장되어야 하겠다.
2. Ball의 크기. 이 변수 또한 constructor의 인자로 받으면 편리하다.
3. Ball의 색깔. 색깔은 바탕처럼 랜덤으로 한다.
4. Ball의 위치를 업데이트 시키는 함수. 1번에서 Ball의 좌표를 나타내는 변수는 생성해 놓았다. 하지만 자연스럽게 구체를 움직이려면 간단하게나마 수학적 계산이 필요하고, 속도, 가속도 등을 계산하는 함수가 필요하다.
5. Ball을 그리는 함수.

1번부터 4번은 단순한 계산과 변수 지정이다. 5번은 새로운 개념이다. 안드로이드에서 화면에 그리는 방법은 앞서 언급한 onDraw를 정의해주고, invalidate()을 통해 특정 View의 onDraw를 반복해서 불러주는 것이다. 여기서 주의해야할 점은, invalidate()과 onDraw는 현재 캔버스에 존재하는 UI를 지우고 업데이트 하라는 명령을 줄 뿐이지, 각 UI의 데이터적인 변화를 적용하거나 실제로 캔버스에 그림을 그리는 함수가 아니다. 개체의 데이터를 수정하는 명령은 Drawable 리소스의 setBounds함수를 호출함으로써 이루어지며, 실제 캔버스에 그림을 그리는 명령은 Drawable 리소스의 draw 메쏘드이다.

안드로이드에서 Drawable이란 “그려질 수 있는 것”이라고 정의된다. Bitmap이 될 수도 있고 기본적으로 제공되는 Drawable Object종류중 하나일 수도 있다. 우리가 이용할 Drawable은 안드로이드 라이브러리에서 제공하는 ShapeDrawble 종류이며, 그 중에서 OvalShape 클래스이다.

Ball 클래스의 constructor에서 구체의 위치와 크기를 받은 후, ShapeDrawable의 instance를 만든다. 각 Ball 클래스의 instance마다 ShapeDrawable이 생기기 때문에, Ball 마다 Shapdrawable을 불러 그리라는 명령을 해줄 수 있다. (이는 BallView의 onDraw 속에서 이루어진다.)

**public** Ball(**int** inputXPosition, **int** inputYPosition, **int** inputBallSize) {  
 **xPosition** = inputXPosition;  
 **yPosition** = inputYPosition;  
 **ballSize** = inputBallSize;  
  
 **ballDrawable** = **new** ShapeDrawable(**new** OvalShape());  
 **ballDrawable**.getPaint().setColor(randomColor());  
 **ballDrawable**.setBounds(**xPosition**, **yPosition**, **xPosition** + **ballSize**, **yPosition** + **ballSize**);  
}

한 가지 중요한 것은, Drawable의 setBounds의 첫 두 인자가 나타내는 지점을 기준으로 그림을 그리기 때문에, Ball의 좌표를 x,y라고 설정해도 공의 중심은 x + ballSize/2, y + ballSize/2에 위치하게 된다. 이를 방지하기 위해선 setBounds의 인자를 xPosition - ballSize/2, yPosition – ballSize/2, xPosition + ballSize/2, yPosition + ballSize/2로 바꾸어 주는 등 여러 해결책이 있을 것이다.

가속 센서에 따라 구체가 움직인 후, BallView는 invalidate을 불러 구체를 새로 그리려 한다. Invalidate은 onDraw를 부르고, 그 onDraw는 각 Ball instance의 drawBall 메쏘드를 호출한다. 다음은 Ball 클래스의 drawBall이다.

**public void** drawBall(Canvas canvas) {  
 **ballDrawable**.setBounds(**xPosition**, **yPosition**, **xPosition** + **ballSize**, **yPosition** + **ballSize**);  
 **ballDrawable**.draw(canvas);  
}

setBounds를 통해 각 Ball의 좌표를 업데이트하고, draw메쏘드로 ShapeDrawable을 화면에 그린다.

여기서 중요한 점은, Ball과 ShapeDrawable은 명백히 다른 두 instance라는 사실이다. 데이터만을 포함하는 Ball의 instance마다 Drawable resource인 ShapeDrable이 저장되어 있는 것이다. 이 둘을 분리해야 Ball을 더 복잡한 환경에서 다룰 때 제약을 줄일 수 있다.

파트1의 마지막으로 공을 움직이는 공식에 대해 설명한다.

**public void** moveBall(**int** screenWidth, **int** screenHeight, **int** xAcc, **int** yAcc) {  
 *// velocity = initial velocity + acceleration \* time* **xVelocity** = **xVelocity** + xAcc;  
 **yVelocity** = **yVelocity** + yAcc;  
 **nextXPosition** = **xPosition** - **xVelocity**;  
 **nextYPosition** = **yPosition** + **yVelocity**;  
 **if** (**nextXPosition** > 0 && **nextXPosition** + **ballSize** < screenWidth) **xPosition** = **nextXPosition**;  
 **else xVelocity** = 0; *// Met screen edge!* **if** (**nextYPosition** > 0 && **nextYPosition** + **ballSize** < screenHeight) **yPosition** = **nextYPosition**;  
 **else yVelocity** = 0; *// Met screen edge!*}

(속도 = 기존 속도 + 가속도\*시간) 공식을 이용해 공의 모션을 정의한다. 인자인 xAcc과 yAcc은 가속도센서의 변화값으로, BallView의 onSensorChanged에서 제공한다. 이 함수가 불릴 때마다 속력에 가속도를 더해 준다면 , 결국 속력에 가속도\*시간을 더하는 효과와 같다. moveBall은 공이 움직일 때 호출되고, 공의 움직임은 시간 유닛마다 끊임없이 이루어지기 때문이다. 현재 position에 속도를 더하면 다음 프레임의 고으이 위치를 알 수 이있다. 이 때, 공이 화면의 edge밖이라면 나가는 방향의 속력을 0으로 바꾸어 움직임을 제한한다.

**파트 2 – 블루투스**

1. 학습 목표: a. 안드로이드 시스템의 블루투스 기능을 다룬다.

b. Activity간 통신의 기본인 Intent에 대해 배운다.

c. 기기간 커뮤니케이션의 기본인 소켓 프로그래밍을 이해한다.

d. 안드로이드와 자바에서 멀티쓰레딩의 기초를 배운다.

2. 수업 내용:

소개

블루투스 연결 강의는 두 단계로 나뉜다. 첫째는 기기의 블루투스 어댑터를 제어하는 단계이고, 두 번째는 블루투스 소켓을 만들고 두 기기를 연결해 데이터를 주고 받는 작업이다. 파트 1 에서 다룬 SensorManager와 같이, 기기의 블루투스 칩셋 제어를 위한 BluetoothAdapter가 존재한다.

첫 단계에선 이 블루투스 어탭터를 활성화하고 주변 기기를 검색할것이다. 먼저 버튼을 화면에 표시하고 버튼 press에 callback function을 추가한다. 새로운 Activity에서 통신이 이루어질 것이기 때문에, 새로운 Activity로 넘어가기 위해 필요한 과정을 배운다. Activity간 정보를 pass한다. 또한, 전 파트의 Listner들 처럼 시스템의 Broadcast(블루투스 상태 변경, 스캔 시작 등의 메시지)에 반응하는 BroadcastListner을 구현할 것이다.

두 번째 단계에선, 첫 Activity에서 넘겨받은 블루투스 기기 정보로 블루투스 소켓을 만든다. 서버, 클라이언트, 연결 후 통신을 위해 총 세개의 쓰레드를 새로 시작해야하며, Activity가 multithreading에 주는 제약을 배운다.

소켓 프로그래밍

블루투스 연결은 블루투스 소켓을 통해 이루어진다. 정확히 말하면, 두 기기가 존재할 때, 일단 두 기기는 자신의 블루투스 어댑터를 활성화 시켜 블루투스 기능 사용을 가능케 한다. 추후의 단계를 나열하면,

1. 한 기기의 블루투스 어댑터는 블루투스 스캔을 시작해 다른 기기의 존재를 인식한다.
2. 발견된 기기는 커넥션을 accept하는 서버가 되고 스캔을 한 기기는 그 서버에 접속할 client가 된다. (스캔을 하지 않은 서버 역할의 기기는 다른 기기가 존재하는지 모르기 때문)
3. 서버 기기는 BluetoothServerSocket을 만들어 클라이언트 기기를 기다린다.
4. 클라이언트 기기는 BluetoothSocket을 만들어 BluetoothServerSocket에 연결한다. 이 때 서버 기기에겐 클라이언트의 BluetoothSocket과 연결된 또 다른 BluetoothSocket이 생긴다.
5. 연결된 후 각 기기의 BluetoothSocket 마다 주어진 inputStream과 outputStream으로 데이터를 주고받는다.

본문

전 파트와 같이 새 프로젝트를 만든다. 시작 Activity의 이름은 MainActivity로 하며, Mainactivity는 안드로이드의 기본 Activity 클래스를 상속받는다. 맨 처음 준비할 것은 화면에 버튼을 셋업하는 것이다. res/layout폴더에 자동으로 생성된 activity\_main.xml 파일을 수정한다.

프로젝트를 새롭게 생성하면 MainActivity의 레이아웃 파일인 activity\_main속에는 RelativeView와 그 속 TextView 두 개가 존재한다. Xml은 태그 언어이다. 한 태그를 열고 그 태그를 닫기 전에 다른 태그를 열면, 다른 태그는 처음 연 태그 안에 생성된다. 안드로이드 레이아웃 xml파일에서 태그 하나는 Layout element 하나(버튼, 글씨 상자, 프레임, 다른 요소를 넣는 틀)를 나타낸다. RelativeLayout은 실제로 UI에 표시되진 않지만 다를 Layout element들을 담는 그릇 역할을 한다. RelativeLayout 속에 정의된 요소들은 서로와 부모에 대한 상대적인 위치로 좌표가 정의된다. TextView는 텍스트를 표시하는 상자이다. 이번 예시에서는 RelativeLayout 대신 LinearLayout을 이용한다. LinearLayout도 RelativeLayout처럼 다른 View 요소들을 담는 그릇이지만, element끼리의 상대적 위치에 의해 element를 배열 하지 않고 element들을 세로나 가로로 줄 세워 표시한다.

activity\_main.xml 을 전부 지우고, 다음 처럼 새로 구성한다.

*<?***xml version="1.0" encoding="utf-8"***?>*<**LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:orientation="vertical" android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"**>  
  
 <**Button  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Turn Bluetooth On"  
 android:id="@+id/onButton"  
 android:layout\_gravity="center\_horizontal"** />  
  
 <**Button  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Scan Nearby Devices"  
 android:id="@+id/scanButton"  
 android:layout\_gravity="center\_horizontal"** />  
  
 <**Button  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="I am the server"  
 android:id="@+id/serverButton"  
 android:layout\_gravity="center\_horizontal"** />  
  
 <**ListView  
 android:layout\_width="match\_parent"  
 android:layout\_height="415dp"  
 android:id="@+id/listView"** />  
</**LinearLayout**>

각 Layout element마다 필수로 정의 해주어야 하는 항목들이 있다. 이 항목을 attribute라고 한다. 모든 요소에 필수인 항목은 layout\_width와 layout\_height이다. 위 코드에는 이 두 변수의 값이 전부 “match\_parent” 또는 “wrap\_content”로 설정 되어있다. “match\_parent”는 그 Layout element의 부모 element를 꽉 체우란 명령이고, “wrap\_content”는 그 element에 표시되는 내용(text, 숫자 등)에 크기에 맞추라는 명력이다. id attribute은 각 레이아웃 요소에 id를 부여해, 코딩이 이루어지는 java 파일에서 reference가 가능하게 한다. text attribute은 그 요소 위에 나타나는 제목이다. layout\_gravity란 요소의 어떤 부위가 요소의 위치를 결정하는데 기준이 되는가를 뜻한다.

위 activity\_main.xml을 분석하면, vertical로 레이아웃 요소를 정렬하는 LinearLayout 속에 버튼 네 개와 ListView가 들어있다. 버튼의 아이디는 각 onButton, scanButton, serverButton이다. 이 버튼들의 각 용도를 설명하며 블루투스 셋업 과정을 설명한다.

MainActivity.java로 돌아와서, MainActivity 클래스의 onCreate 메쏘드 속에서 버튼을 변수에 저장한다. activity\_main.xml을 MainActivity의 레이아웃으로 설정하기 위해, setContentView(activity\_main)을 추가한다. 이는 파트1에서 BallView를 해당 MainActivity의 레이아웃으로 inflate한 것과 같은 과정이다. 레이아웃을 inflate 했기 때문에, 그 레이아웃에 딸린 id pool도 access할 수 있게 된다.

*// Set up Buttons***onButton** = (Button) findViewById(R.id.***onButton***);  
**scanButton** = (Button) findViewById(R.id.***scanButton***);  
**serverButton** = (Button) findViewById(R.id.***serverButton***);

findViewByID 메쏘드는 스트링 형식의 아이디를 인자로 받아 id pool에서 layout element를 리턴한다. 레이아우스이 요소는 버튼, 텍스트뷰 등 여러 종류가 있기 때문에, 변수에 저장할 떄엔 원하는 type으로 캐스팅 해주어야 한다. 위 코드에선 Class Variable로 저장된 세 버튼을 레이아웃으로부터 찾아와 변수에 연결시킨다. 이제 Button 클래스의 메쏘드를 override하며 각 버튼의 속성을 바꿀 수 있다.

세 버튼의 용도는 다음과 같이 코딩 될 것이다.

1. onButton: 기기의 블루투스가 꺼져 있으면 눌러서 블루투스를 켠다.
2. scanButton: 블루투스 어댑터에게 스캔 명령을 보내, 주변 기기를 검색한다.
3. serverButton: 자신이 서버 기기임을 선언한 후 클라이언트를 받을 준비를 시작한다.

따라서, 정상적인 경우, scanButton을 누르는 기기는 serverButton을 누를 일이 없고 serverButton을 누르는 기기는 scanButton을 누를 일이 없다. (앞의 소켓 프로그래밍 설명 참고)

레이아웃만 inflate해도 앱을 실행했을 때 버튼 세개가 보인다. 하지만 버튼을 눌러도 아무 일이 일어나지 않는다. 이제 버튼마다 callback function을 지정한다.

Java 파일에서 직접 버튼에 기능을 더하는 방법은 그 버튼에 onClickListner을 부탁하고, 그 리스너의 onClick 메쏘드를 override하는 것이다.

onButton.setOnClickListener(**new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View v) {  
 askAdapterOn();  
 }  
});  
**scanButton**.setOnClickListener(**new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View v) {  
 scanForDevices();  
 }  
});  
**serverButton**.setOnClickListener(**new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View v) {  
 *// Collect current devices into a set.* callButtons(1, **null**);  
 }  
});

1. onButton: askAdapterOn()이라는 함수를 부른다. 기기의 블루투스를 켜는 함수이다. 기기의 블루투스를 킬지 사용자의 의사를 묻는 다이얼로그를 띄운다. 기기의 블루투스가 꺼져 있다는 것은 블루투스 어댑터를 활성화 시키지 않았다는 뜻이다. askAdapterOn 함수를 분석하기 전, 기기의 BluetoothAdapter을 access하는 코드를 onCreate에 추가한다.

bluetoothAdapter = BluetoothAdapter.*getDefaultAdapter*();

askAdapterOn함수는 화면의 포커스를 “블루투스를 켤까요?”라고 묻는 다이얼로그에 맞춘다. 즉, 새로운 Activity가 불러진다는 뜻이다. 팝업되는 다이얼로그도 하나의 액티비티이다.

※ 새로운 Activity를 실행하는법. + Intention에 대한 이해

-> Activity는 한번에 한 개씩 화면에 표시된다 (Activity 대신 fragment를 사용하면 여러개의 화면을 한번에 표시할 수 있다). Activity A가 Activity B를 부르면, Activity A는 백그라운드에서 실행되고, Activity B가 사용자에게 보여지게 된다. 다음은 Activity 내에서 또 다른 Activity를 부르기 위한 절차이다.

한 Activity A에서 다른 Activity B로 넘어갈 때, Activity A는 화면 이동의 이유를 밝혀야 한다. 이 이유를 안드로이드는 “Intent”라 칭한다. Intent는 안드로이드에서 아주 중요한 개념이다. 파트1 구체에서 SensorEvent 클래스의 instnace는 정체모를 정보를 여러가지 담고 있는 데이터라고 하였다. Intent도 마찬가지이다. Activity A가 Activity B를 왜 부르는지에 대한 이유와(시스템이 호출의 이유를 알기 때문에 어떤 방식으로 불러야 할지 안다) 동시에 Activity A에서 Activity B로 넘겨줄 데이터 모두 Intent 안에 넣어서 전달한다.

**protected void** askAdapterOn() {  
 Intent enableBtIntent = **new** Intent(BluetoothAdapter.***ACTION\_REQUEST\_ENABLE***);  
 startActivityForResult(enableBtIntent, ***REQUEST\_ENABLE\_BT***);  
}

위는 askAdapterOn함수이다. 함수 속에서 새로운 Intent를 생성한다. Intent를 생성하면서 타깃 Activity를 안드로이드에 내장된 시스템 다이얼로그로 설정한다. BluetoothAdapter 클래스의 ACTION\_REQUEST\_ENABLE은 안드로이드 시스템이 사용자의 정의가 필요 없이 알아 듣는 변수이다.

Intent를 준비한 후, 다음 Activity를 “실행”하는 함수는 startActivityForResult가 쓰인다. startActivity와 startActivityForResult 모두 다른 Activity를 호출하는 함수이다. 이 둘의 차이점은, startActivityForResult 함수는 새로운 Activity에서 작업을 마치고 현재 Activity로 복귀했을 때, 함수가 리턴값을 반환하듯 새 Activity내에서 쓰였던 데이터를 리턴할 수 있다.

새로운 Activity가 끝나고 데이터가 리턴됐을 때, 기존의 Activity로의 귀환은 onActivityResult로 이루어진다. 이 때, Activity를 호출했던 startActivityForResult 함수의 두번째 인자(위의 경우에는 REQUEST\_ENABLE\_BT) 로 어떤 Activity로 부터의 리턴인지 식별할 수 있다. 이 식별자는 아무 정수값을 쓸 수 있기 때문에, 좋아하는 축구선수의 등번호가 적당하다. 보통 constants.java라는 파일에 모든 상수를 정의하는데, 이 변수도 그 파일에 포함된다. 다음은 onActivityResult 함수 코드이다.

**protected void** onActivityResult(**int** requestCode, **int** resultCode, Intent data) {  
 **if** (requestCode == ***REQUEST\_ENABLE\_BT***) {  
 **if** (resultCode == ***RESULT\_OK***){  
 **if** (*bluetoothAdapter*.isEnabled()){  
 **onButton**.setEnabled(**false**);  
 **scanButton**.setEnabled(**true**);  
 **serverButton**.setEnabled(**true**);  
 }  
 **else** {  
 **onButton**.setEnabled(**true**);  
 **scanButton**.setEnabled(**false**);  
 **serverButton**.setEnabled(**false**);  
 }  
 }  
 }  
}

블루투스 어댑터 전원을 키는 다이얼로그에서 동의를 하면, 다이얼로그가 종료되면서 MainActivity의 onActivityResult로 돌아오게 된다. 이 때, RESULT\_OK의 값은 동의했다는 뜻이 아닌, 에러 없이 Activity가 마무리 되었다는 뜻이다. 참고로 안드로이드 Activity에 정의된 RESULT\_OK의 값은 -1 이다.

RESULT\_OK를 확인한 후, 세번째 if문에서 블루투스 어댑터가 활성화 되었는지 확인한다. 활성화 돼있다면 블루투스 스캔과 기기의 서버로서의 작동이 가능하기 때문에 scanButton과 serverButton을 활선화 시킨다. setEnabled(true)가 아닌 버튼은 회색이며 누르는 것이 불가능하다. 반면, onButton은 비활성화 시킨다. 만약 bluetoothAdapter가 켜지지 않았다면(사용자가 다이얼로그에서 아니오를 눌렀다면), onButton 이외의 버튼들은 비활성화시킨다.

1. scanButton: 주변의 블루투스 기기를 스캔 하는 scanForDevices 함수를 호출한다. 이 때 주의할 점은 bluetoothAdapter가 null이 아닌 상태, 즉 블루투스가 켜져 있는 상태여야 한다. 이는 앞서 setEnabled함수로 해결한 사안이다.

블루투스 어댑터에 내장되어있는 스캔 함수를 부르는건 어려운 일이 아니다. 하지만 주변의 기기가 발견될 때마다 발견했다는 사실과 발견한 기기를 MainActivity가 알아차리고 리스트 형식으로 표시하는 일은 추가 작업이 필요하다. 우선 블루투스 기기가 검색되었을 때 시스템에서 이를 알아차리는 방법을 설명한다.

scanButton의 onClickListner은 scanForDevices 메쏘드를 호출한다. 다음은 scanForDevices를 선언한 코드이다.

**protected void** scanForDevices() {  
 IntentFilter filter = **new** IntentFilter();  
  
 filter.addAction(BluetoothDevice.***ACTION\_FOUND***);  
 filter.addAction(BluetoothAdapter.***ACTION\_DISCOVERY\_STARTED***);  
 filter.addAction(BluetoothAdapter.***ACTION\_DISCOVERY\_FINISHED***);  
  
 **reciever** = **new** ScanReciever();  
 registerReceiver(**reciever**, filter);  
 *bluetoothAdapter*.startDiscovery();  
}

주변에 활성화된 블루투스 기기를 스캔하는 함수이다. 블루투스 스캔을 위해 필요한건 마지막 줄 뿐이다. bluetoothAdapter.startDiscovery()를 이용해 스캔을 시작하며, bluetoothAdapter.cancelDiscovery()로 스캔을 멈춘다. 하지만 그 과정을 기록하고 표시하기 위에 위의 IntentFilter와 ScanReciever을 setup 해주어야 한다.

IntentFilter에 대한 설명을 하기 전 BroadCastReciever에 대한 설명이 필요하다. BroadCastReciever란, 말 그대로 시스템의 broadcast가 있을 때 마다 실행되는 함수이다. 시스템의 broadcast는 intent형식으로 전달된다. BroadcastReciever는 그 broadcast에 반응한다. Intent의 일종인 Broadcast 가 자신이 반응해서 행동해야 할 intent인지 확인한다. 이 확인을 가능하게 하는 것이 IntentFilter이다. BroadcastReciever에 IntentFilter을 부착시켜, 그 BroadcastReciever의 역할을 정의하는 것이다. 위의 코드에선 BroadcastReciever을 상속한 ScanReciever 클래스를 이용하였다. ScanReciever의 코드를 살펴보자.

**private class** ScanReciever **extends** BroadcastReceiver {  
 **public void** onReceive(Context context, Intent intent) {  
 String action = intent.getAction();  
  
 **if** (BluetoothAdapter.***ACTION\_DISCOVERY\_STARTED***.equals(action)) {  
 System.***out***.println(**"Scan started..."**);} **else if** (BluetoothAdapter.***ACTION\_DISCOVERY\_FINISHED***.equals(action)) {  
 System.***out***.println(**"Scan finished"**);  
 *//discovery finished, dismiss progress dialog* } **else if** (BluetoothDevice.***ACTION\_FOUND***.equals(action)) {  
 *//bluetooth device found* BluetoothDevice device = (BluetoothDevice) intent.getParcelableExtra(BluetoothDevice.***EXTRA\_DEVICE***);  
 System.***out***.println(**"FOUND: "** + device.getName());  
 **deviceList**.add(device);  
 **arrayAdapter**.add(device.getName()+**" - Click to pair"**);  
 **arrayAdapter**.notifyDataSetChanged();  
 }  
 }  
};

시스템의 broadcast를 받을 때 실행되는 함수는 onRecieve이다. BroadcastReciever 클래스를 상속받았기 때문에 코드의 다른 곳에서 onRecieve가 어떻게 작동하는지 설절할 필요는 없다. onRecieve는 자신이 인자로 받은 broadcast로서의 intent에 getAction()을 통해 컨텐츠를 확인한다. 이 컨텐츠는 String으로 변환하여 리시버에 부착된 IntentFilter에 명시된 String 중 하나와 일치하는지 확인한다. 이 중 설명이 필요한 조건문은BluetoothDevice.ACTION\_FOUND이다. 이 else if문 안의 코드는 기기가 검색되었을 때 기기의 화면의 기기 이름 표시를 담당한다. 검색된 기기들은 ArrayList와 ArrayAdapter두 곳에 저장되어, activity\_main에 명시된 ListView에 표시되는데, 이 과정을 설명한다. (이 예시에서 ArrayList는 기기 자체를,ArrayAdapter은 기기 이름은 저장한다.)

※ ArrayList, ArrayAdapter, ListView

ArrayList는 자바에서 유용하게 쓰이는 다이내믹 행렬이다. ListView는 항목들을 List형식으로 화면에 나타내주는layout element의 한 종류이다. ArryaAdapter은 이 둘은 연결한다. ArrayList에 기기가 저장될 때 그 기기의 이름을 ArrayAdapter에 저장한다. ArrayLists는 이 기기 목록을 ListView에 전부 표시하며, ArrayAdapter의 내용이 바뀌면 메쏘드 하나로 ListView를 업데이트한다. 다음은ArrayList, ArrayAdapter, ListView를 연동하기 위한 준비과정이다.

onCreate에서, 우선 기기를 저장할 ArrayList인 deviceList와 ArrayAdapter을 초기화한다.

deviceList = **new** ArrayList<>();  
**arrayAdapter** = **new** ArrayAdapter<String>(**this**, R.layout.***table\_item***);

ArrayAdapter는 R.layout.table\_item이라는 항목을 인자로 받는 것을 볼 수 있다. 이 인자는 레이아웃 xml파일이다. ListView를 이용하려면, ListView의 한 줄을 차지할 TextView를 xml파일로 제공해야한다. table\_item.xml을 살펴보면,

*<?***xml version="1.0" encoding="utf-8"***?>*<**TextView xmlns:android="http://schemas.android.com/apk/res/android"  
 android:id="@android:id/text1"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:textAppearance="?android:attr/textAppearanceListItemSmall"  
 android:gravity="center\_vertical"  
 android:paddingStart="?android:attr/listPreferredItemPaddingStart"  
 android:paddingEnd="?android:attr/listPreferredItemPaddingEnd"  
 android:minHeight="?android:attr/listPreferredItemHeightSmall"** />

하나의 TextView를 정의하고 있다. 이 TextView를 어떻게 정의하느냐에 따라서 나타나는 글씨 레이아웃을 바꾸고 목록에 그림까지 표시 할 수 있다.

ArrayAdapter과 ArrayList를 초기화 한 후, ListView를 layout파일에서 아이디로 찾아 변수에 집어넣는다. findViewByID를 이용한다.

*// Set up listview***listView** = (ListView) findViewById(R.id.***listView***);  
**listView**.setAdapter(**arrayAdapter**);  
**listView**.setOnItemClickListener(**new** AdapterView.OnItemClickListener() {  
 **public void** onItemClick(AdapterView<?> arg0, View v, **int** position, **long** id) {  
 callButtons(0,**deviceList**.get(position));  
 }  
});

listView.setAdapter로 listView에 ArrayAdapter을 attach시킨다. onItemClickListnerg함수를 정의해 리스트뷰의 항목이 눌렸을 때 일어날 일을 선언한다. n번째 항목이 눌리면 deviceList의 n번째 기기와 연결을 감행 할 것이다. 연결을 하면서 다음 액티비티로 넘어가는데, 이건 serverButton에 대해 설명하면서 이야기한다.

다시 ScanReciever로 돌아와서, 시스템으로부터 받은 Broadcast가 BluetoothDevice.ACTION\_FOUND일 때, Intent에 저장되어있는 블루투스 디바이스를 device변수에 저장한다. Intent에 저장된 정보를 받는 함수 중 하나인 getParcelableExtra함수의 리턴값을 블루투스 디바이스로 캐스팅하여 받는다. 그 후 deviceList에는 device를 추가하고, ArrayAdapter에는 davice의 이름을 추가한다. 마지막으로 notifyDataSetChanged 메쏘드로 리스트뷰를 업데이트한다.

1. serverButton: 이 버튼을 누르면 buttons액티비티로 넘어간다. 액티비티가 넘어간다는 것은 화면이 바뀐다는 것이다. serverButton의 onCLickListner와 ListView의 onItemClick 모두 callButtons라는 함수를 부른다. 이 함수는 Buttons class를 부르는 함수이다.

**protected void** callButtons(**int** server,BluetoothDevice device) {  
 *bluetoothAdapter*.cancelDiscovery();  
 Intent toButton = **new** Intent(getApplicationContext(), buttons.**class**);  
 toButton.putExtra(**"selected"**,device);  
 **if** (server == 0) {  
 toButton.putExtra(**"isServer"**,0);  
 }  
 **else** {  
 toButton.putExtra(**"isServer"**,1);  
 }  
 startActivity(toButton);  
}

이 함수를 설명하기 전, src/main/res/AndroidManifest.xml파일에 buttons를 하나의 액티비티로 선언 해주어야 한다. Manifest파일에 액티비티 이름이 등록되지 않으면 그 액티비티를 부를 때 널 포인터 익셉션이 뜬다.

또, 블루투스 기능을 사용하기 위해 블루투스 permission항목도 추가한다. AndroidManifest.xml을 다음 처럼 고친다.

*<?***xml version="1.0" encoding="utf-8"***?>*<**manifest xmlns:android="http://schemas.android.com/apk/res/android"  
 package="com.seongminpark.bluetooth"** >  
  
 <**application  
 android:allowBackup="true"  
 android:icon="@mipmap/ic\_launcher"  
 android:label="@string/app\_name"  
 android:theme="@style/AppTheme"** >  
 <**activity  
 android:name=".MainActivity"  
 android:label="@string/app\_name"** >  
 <**intent-filter**>  
 <**action android:name="android.intent.action.MAIN"** />  
  
 <**category android:name="android.intent.category.LAUNCHER"** />  
 </**intent-filter**>  
 </**activity**>  
 <**activity android:name=".buttons"**> </**activity**>  
 </**application**>  
  
 <**uses-permission android:name="android.permission.BLUETOOTH"** />  
 <**uses-permission android:name="android.permission.BLUETOOTH\_ADMIN"** />  
</**manifest**>

callButtons에 대한 설명으로 돌아와서, 먼저 인자에 대한 설명이다. callButtons는 두 개의 인자를 받는다. 첫 번째 server인자는 이 함수가 불리는 기기가 블루투스 서버 역할을 할지, 클라이언트 역할을 할지 명시한다. server가 0 이라면 클라이언트, 1 이라면 서버이다. 서버냐 아니냐에 따라 연결을 위한 기기의 다음 과정이 달라진다. ListView의 onItemClick은 클라이언트가 서버로 커넥트 할 때 부르는 함수이기 때문에 그 속의 callButtons의 server가 0이고, serverButton의 onClickListner가 호출하는 callButtons는 이 기기를 서버로 지정하는 버튼이기 때문에 server 인자가 1이다. 이 server 식별자는 Buttons 액티비티로 넘어갈 때 Buttons 액티비티로 전달되어 사용된다.

다음 Activity로 정보를 넘기는 방법은 앞에서 언급했듯 intent를 이용하는 것이다. StartACtivity()함수의 인자로 들어가는 intent는, 우리가 도달하고 싶어하는 액티비티를 명시함과 동시에 다른 정보도 번들로 포함해서 넘어간다. 이 정보 저장은 putExtra 메쏘드로 가능하다. Extra 정보는 Key-value pair로 구성되며, 예시에선 isServer라는 스트링과 server인자를 묶어 buttons 클래스로 전달한다. 다시 MainActivity로 돌아오지 않기 때문에, 앞서 블루투스 활성화 다이얼로그를 호출할 때완 다르게 startActivityForResult가 아닌 startActivity를 사용한다.

Buttons 부터 블루투스 예제의 두 번째 단계이다. Buttons 액티비티에서는 두 기기가 블루투스로 연결되어, 한 기기의 1,2,3 버튼 중 하나를 누르면 그 기기와 다른 기기 동시에 textView에 그 버튼의 숫자를 표시한다. 우선, buttons의 레이아웃 파일인 buttons.xml을 다음페이지와 같이 바꾼다.

MainActivity와 다르게, buttons는 RelaiveLayout응 이용해 레이아웃 요소들을 담는다. 네개의 버튼과 한개의 TextView가 존재한다. 제일 먼저, buttons.java의 onCreate 메쏘드에 buttons.xml의 레이아웃을 inflate시킨다. (setContentView). 그 후, MainActivity에서 넘겨받은 intent속 서버 식별자를 미리 선언해 둔 isServer 변수에 저장한다. 그 후, connectionSetup이라는 함수를 호출해 연결을 시작한다.

**private void** connectionSetup() {  
 **if** (**isServer** == 0) {  
 BluetoothDevice device = getIntent().getExtras().getParcelable(**"selected"**);  
 **connectThread** = **new** ConnectThread(device);  
 **connectThread**.start();  
 }  
 **else** {  
 **acceptThread** = **new** AcceptThread();  
 **acceptThread**.start();  
 }  
}

setContentView(R.layout.***buttons***);  
**isServer** = getIntent().getExtras().getInt(**"isServer"**);  
connectionSetup();

connectionSetup 메쏘드 안에서, 현재 기기가 서버가 아니라면, 타깃 기기를 intent에서 뽑아 connectThread를 시작하고, 서버라면 acceptThread를 시작한다. acceptThread, connectThread, 그리고 추후에 나올connectedThread 전부 안드로이드 개발자 홈페이지에서 권장하는 방식의 예제를 조금 개조한 것이다. 쓰레드의 instance를 만들어 start를 부르면 쓰레드의 run method가 시작된다.

블루투스 연결에 멀티 쓰레딩을 이용하는이유는 나중에 설명한다. 다다음 페이지에서 AcceptThread를 설명한다.

<**RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools" android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent" android:paddingLeft="@dimen/activity\_horizontal\_margin"  
 android:paddingRight="@dimen/activity\_horizontal\_margin"  
 android:paddingTop="@dimen/activity\_vertical\_margin"  
 android:paddingBottom="@dimen/activity\_vertical\_margin" tools:context=".MainActivity"**>  
  
 <**Button  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="1"  
 android:id="@+id/button1"  
 android:layout\_alignParentBottom="true"  
 android:layout\_alignParentStart="true"** />  
  
 <**Button  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="2"  
 android:id="@+id/button2"  
 android:layout\_alignTop="@+id/button1"  
 android:layout\_centerHorizontal="true"  
 android:layout\_alignParentBottom="true"** />  
  
 <**Button  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="3"  
 android:id="@+id/button3"  
 android:layout\_alignTop="@+id/button2"  
 android:layout\_alignParentEnd="true"** />  
  
 <**TextView  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:textAppearance="?android:attr/textAppearanceLarge"  
 android:id="@+id/textView"  
 android:layout\_alignParentTop="true"  
 android:layout\_centerHorizontal="true"  
 android:layout\_marginTop="115dp"** />  
  
 <**Button  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Unpair"  
 android:id="@+id/unpairButton"  
 android:layout\_centerVertical="true"  
 android:layout\_centerHorizontal="true"** />  
  
</**RelativeLayout**>

**private class** AcceptThread **extends** Thread {  
 **private final** BluetoothServerSocket **serverSocket**;  
  
 **public** AcceptThread() {  
 *// Use a temporary object that is later assigned to mmServerSocket,  
 // because mmServerSocket is final* BluetoothServerSocket tmp = **null**;  
 **try** {  
 *// MY\_UUID is the app's UUID string, also used by the client code* tmp = MainActivity.*bluetoothAdapter*.listenUsingRfcommWithServiceRecord(***NAME***, ***MY\_UUID***);  
 } **catch** (IOException e) { }  
 **serverSocket** = tmp;  
 }  
  
 **public void** run() {  
 *//BluetoothSocket socket = null;  
 // Keep listening until exception occurs or a socket is returned* **while** (**true**) {  
 **try** {  
 **passiveSocket** = **serverSocket**.accept();  
 manageConnection();  
 } **catch** (IOException e) {  
 **break**;  
 }  
 *// If a connection was accepted* **if** (**passiveSocket** != **null**) {  
 *// Do work to manage the connection (in a separate thread)  
 //manageConnectedSocket(socket);* **try** {  
 **serverSocket**.close();  
 } **catch** (IOException e) { }  
 **break**;  
 }  
 }  
 }  
  
}

AcceptThread는 서버 역할의 기기에서 실행된다. AccpetThreadrk 초기화 될 때 constructor에선 BluetoothServerSocket을 만든다. listenUsingRfcommWithServiceRecord 메쏘드를 사용하여 BluetoothServerSocket이 만들어지는데, 이 메쏘드는 NAME과 UUID 두 인자를 받는다. 이 중 중요한건 UUID이다. UUID는 인터넷의 port number과 비슷한 개념이다. 블루투스 클라이언트의 BluetoothSocket이 서버의 BluetoothServerSocket과 연결하려 할 때, 이 UUID를 비교해 자신의 UUID와 매칭되어야 연결을 감행한다.

위 AcceptThread에서 맨 처음 class variable로 serverSocket을 선언 했음에도 불구하고 tmp변수를 이용해 BluetoothServerSocket을 만드는 이유는, 이 작업은 exception을 catch하지 않고는 코딩 할 수 없도록 안드로이드 sdk가 제한을 두었기 때문이다. serverSocket을 얻은 후, run 메쏘드에서 들어오는 클라이언트의 커넥션을 받는 코드를 완성한다.

run 메쏘드는 buttons의 클래스 변수로 선언해 둔 passiveSocket에 값을 부여하는 역할이다. BluetoothServerSocket은, accept라는 함수를 이용해 접속하는 BluetoothSocket을 받아들인다. 커넥션이 완료 되었을 때, accpet는 새로운 BluetoothSocket을 리턴한다(위 코드의 passiveSocket). 이 시점에서 서버 기기와 클라이언트 기기 둘 다 각각 하나씩의 BluetoothSocket의 instance를 가지게 된다. 두 개의 BluetoothSocket이지만, 같은 UUID를 가지고 서로 연결되어있는 상태이다.

BluetoothServerSocket(위 코드에선 그의 instance인 serverSocket)의 accept thread는 blocking call이다. 이는 함수가 리턴 때까지 그 함수의 쓰레드에서는 아무 행동도 취할 수 없다는 뜻이다. 따라서, 서버소킷의 준비과정을 새로운 쓰레드로 시작해야만 메인 쓰레드의 작업에 방해가 되지 않는 것이다. while문 속 if문에서, passiveSocket에 블루투스 소켓이 할당된 것을 확인하면, serverSocket은 더이상 필요하지 않기 때문에 닫는다.

try문 안에서 accept메쏘드가 성공적으로 이루어진 후, manageConnection함수가 불린다. 이 함수는 클라이언트 기기의 connectThread를 설명 한 후 살펴본다. manageConnection에서 두 기기간 연결 상태를 제어하는 새로운 쓰레드를 시작하는데, 이도 서로의 데이터 열람을 위한 함수가 blocking call이기 때문에 새로운 쓰레드를 필요로 한다.

AcceptThread와는 반대로 클라이언트에서만 실행되는 ConnectThread이다. 서버 소켓이 열린 블루투스 기기를 찾아 자신의 소켓을 끼워 넣는 쓰레드이다. 먼저 constructor에서 bluetoothSocket을 만든다. 이는 createRfcommSocketToServiceRecord를 이용하는데, 블루투스 디바이스의 메쏘드이기 때문에 앞서 intent에 넣어서 전달받은 bluetoothDevice를 constructor의 인자로 받는다. 여기서 제작된 socket은 서버에서 만들어지는 소켓과의 구별을 위해 activeSocket이라고 부른다. run메쏘드에선, 만든 socket의 connect함수를 부른다. 이 함수도 blocking call이다. connect는 같은 uuid의 서버소킷이 있기 때문에, 안드로이드 시스템의 블루투스 연결 다이얼로그를 불러 연결을 시작한다. ConnectThread 역시 manageConnection함수를 불러 연결을 관리한다.

**private class** ConnectThread **extends** Thread {  
 **private final** BluetoothSocket **socket**;  
  
 **public** ConnectThread(BluetoothDevice device) {  
 *// Use a temporary object that is later assigned to socket,  
 // because socket is final* BluetoothSocket tmp = **null**;  
  
 *// Get a BluetoothSocket to connect with the given BluetoothDevice* **try** {  
 *// MY\_UUID is the app's UUID string, also used by the server code* tmp = device.createRfcommSocketToServiceRecord(***MY\_UUID***);  
 } **catch** (IOException e) { }  
 **socket** = tmp;  
 **activeSocket** = **socket**;  
 }  
  
 **public void** run() {  
 **try** {  
 *// Connect the device through the socket. This will block  
 // until it succeeds or throws an exception* **socket**.connect();  
 manageConnection();  
 } **catch** (IOException connectException) {  
 *// Unable to connect; close the socket and get out* **try** {  
 **socket**.close();  
 } **catch** (IOException closeException) { }  
 **return**;  
 }  
 }  
}

manageConnection은 기기가 서버냐 클라이언트냐에 따라 작동을 달리한다. 기기의 역할에 따라 passiveSocket과 activeSocket 중 하나만 생성되어있기 때문이다. 해당되는 소켓을 인자로 넣고 connectedThread를 만든다. 위 코드에서 BluetoothSocket 변수를 생성해 if문 속에서 그 변수만 지정해주고 connectThread 초기화는 한번에 할 수 있지만, 그렇게 하지 않았다. 다음 페이지에서 ConnectedThread와 기기간의 대화에 대해 설명한다.

**private void** manageConnection() {  
 **if** (**isServer** == 0) {  
 **connectedThread** = **null**;  
 **try** {  
 **connectedThread** = **new** ConnectedThread(**activeSocket**);  
 **connectedThread**.start();  
 } **catch** (Exception e) {  
 Log.*e*(**"Connthread not started"**, e.getMessage());  
 }  
 }  
 **else** {  
 **connectedThread** = **null**;  
 **try** {  
 **connectedThread** = **new** ConnectedThread(**passiveSocket**);  
 **connectedThread**.start();  
 } **catch** (Exception e) {  
 Log.*e*(**"Connthread not started"**, e.getMessage());  
 }  
 }  
}

**private class** ConnectedThread **extends** Thread {  
 **private final** BluetoothSocket **bluetoothSocket**;  
 **private final** InputStream **inputStream**;  
 **private final** OutputStream **outputStream**;  
  
 **public** ConnectedThread(BluetoothSocket socket) {  
 **cn\_established** = **true**;  
 **bluetoothSocket** = socket;  
 InputStream tmpIn = **null**;  
 OutputStream tmpOut = **null**;  
 **try** {  
 tmpIn = **bluetoothSocket**.getInputStream();  
 tmpOut = **bluetoothSocket**.getOutputStream();  
 } **catch** (IOException e) {  
 }  
  
 **inputStream** = tmpIn;  
 **outputStream** = tmpOut;  
 }  
  
 **public void** run() {  
 **byte**[] buffer = **new byte**[4];  
 **int** bytes;  
 **while** (**true**) {  
 **try** {  
 **if** (**inputStream**.available() > 0) {  
 bytes = **inputStream**.read(buffer);  
 **handler**.obtainMessage(***MESSAGE\_READ***, bytes, -1,buffer).sendToTarget();  
 }  
 } **catch** (IOException e) {  
 **break**;  
 }  
 }  
 }  
  
 **public void** write(**int** i) {  
 **try** {  
 **byte**[] buffer = **new byte**[4];  
 buffer = intToByteArray(i);  
 System.***out***.println((**byte**) i);  
 **outputStream**.write(buffer);  
 **handler**.obtainMessage(***MESSAGE\_WRITE***, -1, -1,  
 buffer).sendToTarget();  
 } **catch** (IOException e) {  
 System.***out***.println(**"write error"**);  
 }  
 }  
}

ConnectedThread의 설명에는 Handler와 Message개념이 같이 등장한다. Handler도 BroadcastReciever과 비슷한 역할을 갖는다. 시스템의 어떤 행동에 trigger되어 적당한 함수를 부르는 역할이다. 차이점은, Handler가 반응하는 메시지는 우리의 코드에서 보내진다는 점이다. Handler의 구현을 살펴보기 전 ConnectedThread의 constructor 부터 설명한다.

블루투스 소켓이 곧 커넥션의 모든것인 만큼 connectedThread는 생성될 때 블루투스 소켓을 인자로 받는다. ConnectedThread가 초기화 됐을 때 가장 먼저 하는 일은 buttons 클래스의 cn\_established변수를 true로 바꾸어 주는 것이다. 이 변수는 연결이 시작됐음을 표시하며, cn\_establiched가 true라면 connectedThread에 대한 접근이 nullPointerException을 일으키지 않는 것을 확신할 수 있다.

그 후, socket의 inputStream와 outputStream을 변수에 지정한다. 모든 BLuetoothSocket은 하나의 inputStream과 하나의 outputStream을 가진다. 두 기기가 connect 되었을 때, 한 기기의 ouputStream에 정보를 보내면 다른 기기의 inputStream으로 전달된다. 따라서 ConnectedThread에서는 inputStream에서 읽는 함수와 outputStream에서 쓰는 함수를 선언한다.

쓰레드가 실행될 때 돌아가는 run 함수를 inputStream을 읽는데에 사용하고, 새로운 write 함수를 정의해 outputStream에 정보를 입력 가능하게 만든다. 위 코드의 run 메쏘드를 보면 reading 프로세스는 간단하다. 빈 byte array를 만들고, inputStream의 read메쏘드로 inputStream에 있는 정보를 읽어들인다. inputStream의 read 메쏘드 역시 blocking call이다. 읽을 거리가 있을 때 까지 프로세스는 마비된다. 읽어들인 후 handler의 obtainMessage를 이용해, handler에세 신호를 보낸다. obtainMessage는 핸들러가 메시지를 전달받았다는 함수이다. MESSAGE\_READ역시 상수이며, handler의 handleMessage 내에서 어떤 메시지를 받았는지 확인하는 용도이다. 다음은 handler을 정의한 코드이다.

Hanlder을 부른 obtainMessage의 식별자가 MESSAGE\_READ였을 때, obtainMEssage의 마지막 인자인 msg.obj를 알맞은 타입으로 캐스팅해서 꺼낸다. 그 후, byteArrayToInt로 정수로 변환하고 또 스트링으로 변환해 textView에 표시한다.

**private final** Handler **handler** = **new** Handler() {  
 @Override  
 **public void** handleMessage(Message msg) {  
 **switch** (msg.**what**) {  
 **case *MESSAGE\_WRITE***:  
 **byte**[] writeBuf = (**byte**[]) msg.**obj**;  
 String writeMessage = **new** String(Integer.*toString*(byteArrayToInt(writeBuf)));  
 **tv**.setText(writeMessage);  
 **break**;  
 **case *MESSAGE\_READ***:  
 **byte**[] readBuf = (**byte**[]) msg.**obj**;  
 String readMessage = **new** String(Integer.*toString*(byteArrayToInt(readBuf)));  
 System.***out***.println(readMessage);  
 **tv**.setText(readMessage);  
 **break**;  
 }  
 }  
};

write함수는 정반대이다. intToByteArray로 정수를 byteArray로 변환하고, outputStream.write 메쏘드로 상대의 inputStream으로 보낸다. 각 버튼의 onClickListner는 connectedThread의 write을 부른다. write과 read가 같은 쓰레드의 인스턴스의 메쏘드이고 read가 blocking call이라도 write이 가능한 이유이다.

마지막으로 unpairButton이 있다. BluetoothDevice 클래스의 removeBond를 이용한다.

**private void** unpair() {  
 **if** (!**cn\_established**) **return**;  
 BluetoothDevice device;  
 **if** (**isServer**==0) device = **activeSocket**.getRemoteDevice();  
 **else** device = **passiveSocket**.getRemoteDevice();  
 **try** {  
 Method m = device.getClass().getMethod(**"removeBond"**, (Class[]) **null**);  
 m.invoke(device, (Object[]) **null**);  
 } **catch** (Exception e) {  
 Log.*e*(**"Unpair failed."**, e.getMessage());  
 }  
}

**파트 3 – 구체 + 블루투스**

세 번째 파트에선 파트 1과 파트 2를 합친다. 블루투스로 공을 전달하는 과정이다. 폰을 거꾸로 기울여서 공을 쏟아내면 다른 기기에 공이 나타난다.

파트 2의 buttons클래스 자리에 파트 1의 BallView클래스가 들어간다. 이 과정에서 전 단계와의 차이점이 몇 가지 있는데, 차이점을 위주로 설명한다.

Ball객체를 다루는방법에서 차이점이 발생한다

.

1. 파트1에선 손가락으로 터치할 때 매다 공이 생겼지만, 멀티쓰레딩에선 array 관리가 어렵시 때문에 파트2에선 공을 하나만 다룬다.
2. 파트1에서는 Ball instance의 ShapeDrawable이 Ball instance 안에 포함되었다(ShapeDrawable이 Ball 클래스 안에 정의됨). 파트3에서는 BallView에서 ShapeDrawable을 관리한다.

이 이유는 outputStream과 inputStream을 통과하려면serializable 한 오브젝트 여야 하기 때문이다. ShapeDrawable은 serializable하지 않다.

1. 따라서, Ball 클래스는 Serializable클래스를 implement한다.
2. outputStream, inputStream으로 opbestOutputStream, objectInputStream을 만들어 이용한다.

*// Set up object input, output streams.*ObjectInputStream tmpObIn = **null**;  
ObjectOutputStream tmpObOut = **null**;  
**try** {  
 tmpObOut = **new** ObjectOutputStream(**outputStream**);  
 tmpObOut.flush();  
 tmpObIn = **new** ObjectInputStream(**inputStream**);  
} **catch** (StreamCorruptedException e) {  
} **catch** (IOException e) {  
}