

안드로이드 SDK 개발 플랫폼 활용하기

디버거 사용법 익히기

- 컴파일 시점 오류
 - 문법적 오류이고 컴파일러가 이를 알려주기 때문에 수정이 쉬운 편임
- 실행 시점 오류
 - 원인을 알 수 없기 때문에 해결이 어려움
 - 디버거를 이용하면 문제를 쉽게 해결할 수 있음
 - 디버거는 메모리의 내부를 눈으로 들여다 볼 수 있게 해주기 때문에 가장 강력한 디버그 도구이며 사용법을 익히는 것이 필수적임

- Break Point의 역할
 - 실행 중 잠깐 멈추게 할 곳을 나타내며 프로그램을 실행 중에 멈추게 한 후 메모리 내부를 들여다 볼수 있음
 - 단 디버깅 모드로 실행 시켜야 함
 - 마우스 커서를 변수 위에 올리면 값이 보임

F5: 한단계하위로이동

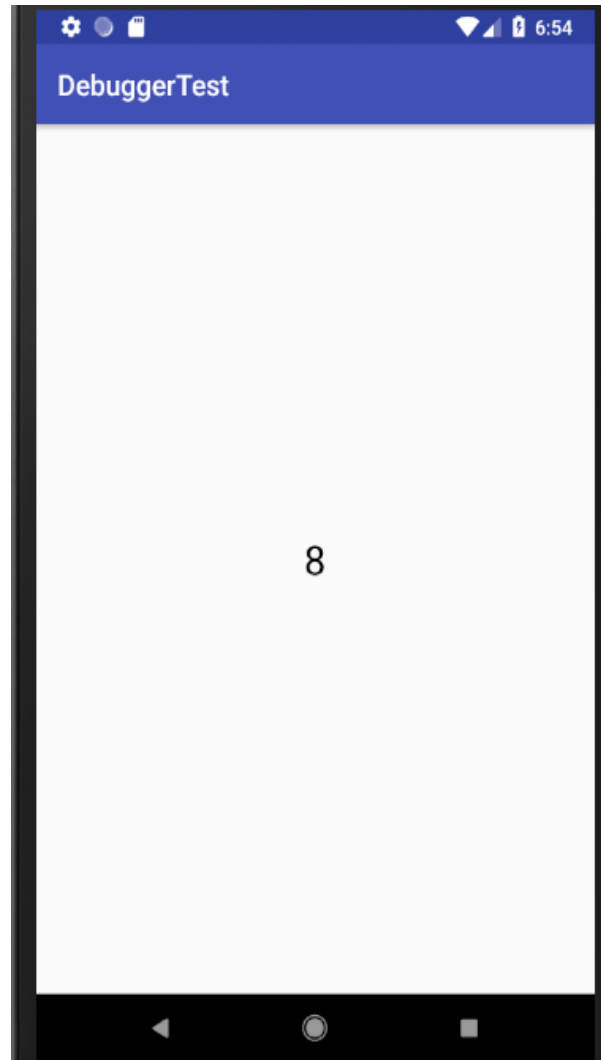
F6: 한줄앞으로이동

F7: 한단계상위로이동

F8: 작업재개

디버그 용 기본 단축키

- 디버깅을 위한 샘플 프로그램 구현



- activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>  
<android.support.constraint.ConstraintLayout xmlns:android="안드로이드 APK 파일의  
스키마 경로는 생략합니다."
```

```
xmlns:app="안드로이드 APK 파일의 스키마 경로는 생략"  
xmlns:tools="http://schemas.android.com/tools"  
android:layout_width="match_parent" // 부모를 안벗어나게 최대화  
android:layout_height="match_parent" // 부모를 안벗어나게 최대화  
tools:context=".MainActivity">
```

```
<TextView  
    android:id="@+id/textView"  
    android:layout_width="wrap_content" // 컨텐츠에 맞게 최소화  
    android:layout_height="wrap_content" // 컨텐츠에 맞게 최소화  
    android:text="Hello World!"  
    android:textColor="#000000"
```

- 디버깅을 위한 샘플 프로그램 구현

```
android:textSize="28dp"  
app:layout_constraintBottom_toBottomOf="parent" // 부모를 기준으로 맞춤  
app:layout_constraintLeft_toLeftOf="parent" // 부모를 기준으로 맞춤  
app:layout_constraintRight_toRightOf="parent" // 부모를 기준으로 맞춤  
app:layout_constraintTop_toTopOf="parent" // 부모를 기준으로 맞춤 />
```

```
</android.support.constraint.ConstraintLayout>
```

- MainActivity.java

```
package com.iot.debuggertest;

import android.support.v7.app.AppCompatActivity; // 화면 호환 패키지 포함
import android.os.Bundle; // 안드로이드 액티비티용 번들
import android.widget.TextView; // 화면에 문자열 표시를 위한 텍스트 뷰

public class MainActivity extends AppCompatActivity { // 메인 화면

    private TextView textView;

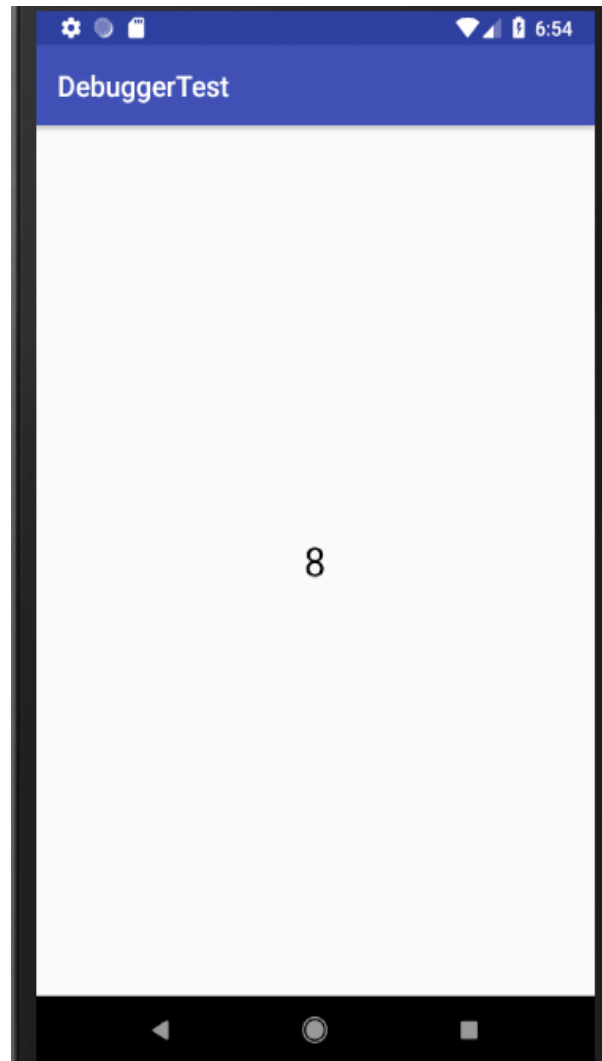
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        textView = findViewById(R.id.textView);

        int result = 0;
        int a = 2;
        int b = 3;
        int c = 5;
        result = a << 2;
        result += b;
```

```
        result = (result+c) >> 1;  
        result = add(result, 3);  
        textView.setText(String.valueOf(result));  
    }
```

```
int add(int a, int b) {  
    int sum = 0;  
    sum = a;  
    sum += b;  
    return sum;  
}  
}
```


- 왜 이렇게 나오는지 생각해보자.



- 프로그램을 디버그 모드로 실행
 - 프로그램이 브레이크포인트가 있는 곳에서 멈추는 지 확인
 - 디버깅 각 버튼의 사용법을 익히고 한 단계씩 진행
 - 내부 메모리를 들여다 본다.

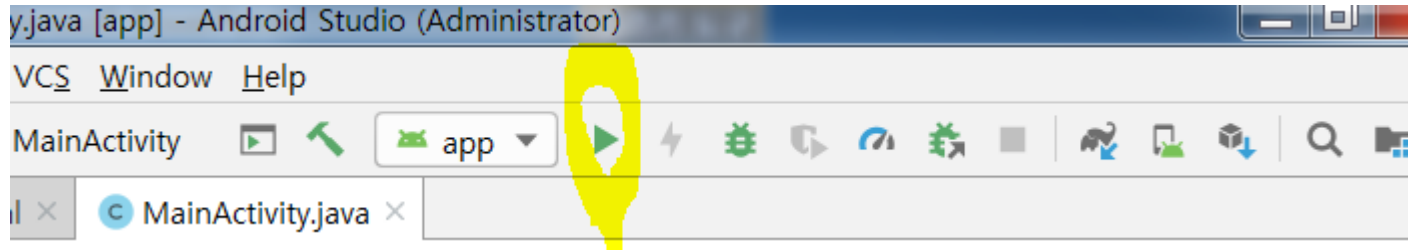
- 브레이크 포인트 거는 방법
 - 텍스트 입력창의 좌측 스크롤바가 있는 영역에 마우스 우클릭.
 - 세부 메뉴 중에 toggle break point 라는 기능
 - 브레이크 포인트를 추가하거나 삭제할 수 있는 기능
 - disable/ enable break point와 다른 점
 - 브레이크 포인트를 제거하거나 추가하지 않고 잠시 끄거나 켜기 기능

- Ctrl + F8
 - Menu > Run > Toggle Line BreakPoint


```
16      textView = findViewById(R.id.textView);
17
18  ●    int result = 0;
19      int a = 2;
20      int b = 3;
21  int c = 5;
22      result = a << 2;
23      result += b;
24      result = (result+c) >> 1;
25      result = add(result, b: 3);
26      textView.setText(String.valueOf(result));
27  }
```

- 이제 프로그램을 디버그 모드로 실행 시킨다.
 - 프로그램 시작 시 경고창이 표시
 - 디버그 모드로 실행 되기 때문에 준비와 실행 과정에 조금더 시간이 걸릴 수 있다는 내용
- 프로그램의 흐름이 break point에 도달하게 되면 프로그램은 즉시 멈춘다.
- 실제 프로그램이 멈추면 AVD의 화면에는 아무것도 표시되지 않는다.

- Shift + F9




- 브레이크 포인트가 걸린 상태

```
17  
18  int result = 0;  
19 int a = 2;  
20 int b = 3;  
21 int c = 5;  
22 result = a << 2;  
23 result += b;  
24 result = (result+c) >> 1;  
25 result = add(result, b: 3);  
26 textView.setText(String.valueOf(result));
```

- 앞서 배웠던 단축키를 이용해서 소스코드를 한줄 씩 실행
 - 최근에 변경된 값은 Debugger View에서 노랑색으로 표시
- 먼저 한줄 씩 아래로 진행해 보는데 step over 기능을 이용
 - 한 줄 씩 아래로 진행
 - 예상된 변수의 값과 실제 메모리 상에 저장된 값이 동일한지 확인하며 진행

- F8
 - Step Over 명령

```
17  
18  int result = 0; result: 8  
19 int a = 2; a: 2  
20 int b = 3; b: 3  
21 int c = 5; c: 5  
22 result = a << 2; a: 2  
23 result += b; b: 3  
24 result = (result+c) >> 1; c: 5  
25 result = add(result, b: 3); result: 8  
26 textView.setText(String.valueOf(result));
```


- 어떤 메소드를 호출한다면 그것을 1줄처럼 실행하려면 마찬가지로 step over 명령을 이용
 - 만약 그 메소드 안으로 이동하여 하위 코드를 한줄 씩 실행하려면, step into 명령을 이용
 - 반대로 현재 디버깅 중인 메소드의 한단계 상위 메소드로 벗어나려면 step out 명령을 이용

- Step Into 명령
 - F7
 - 메소드 안으로 들어가기

```
29  int add(int a, int b) {  a: 8  b: 3
30  int sum = 0;
31  sum = a;
32  sum += b;
33  return sum;
34
35 }
```


- Step Out

- Shift + F8
- 현재 메소드를 호출한 곳으로 한 단계 빠져나가기

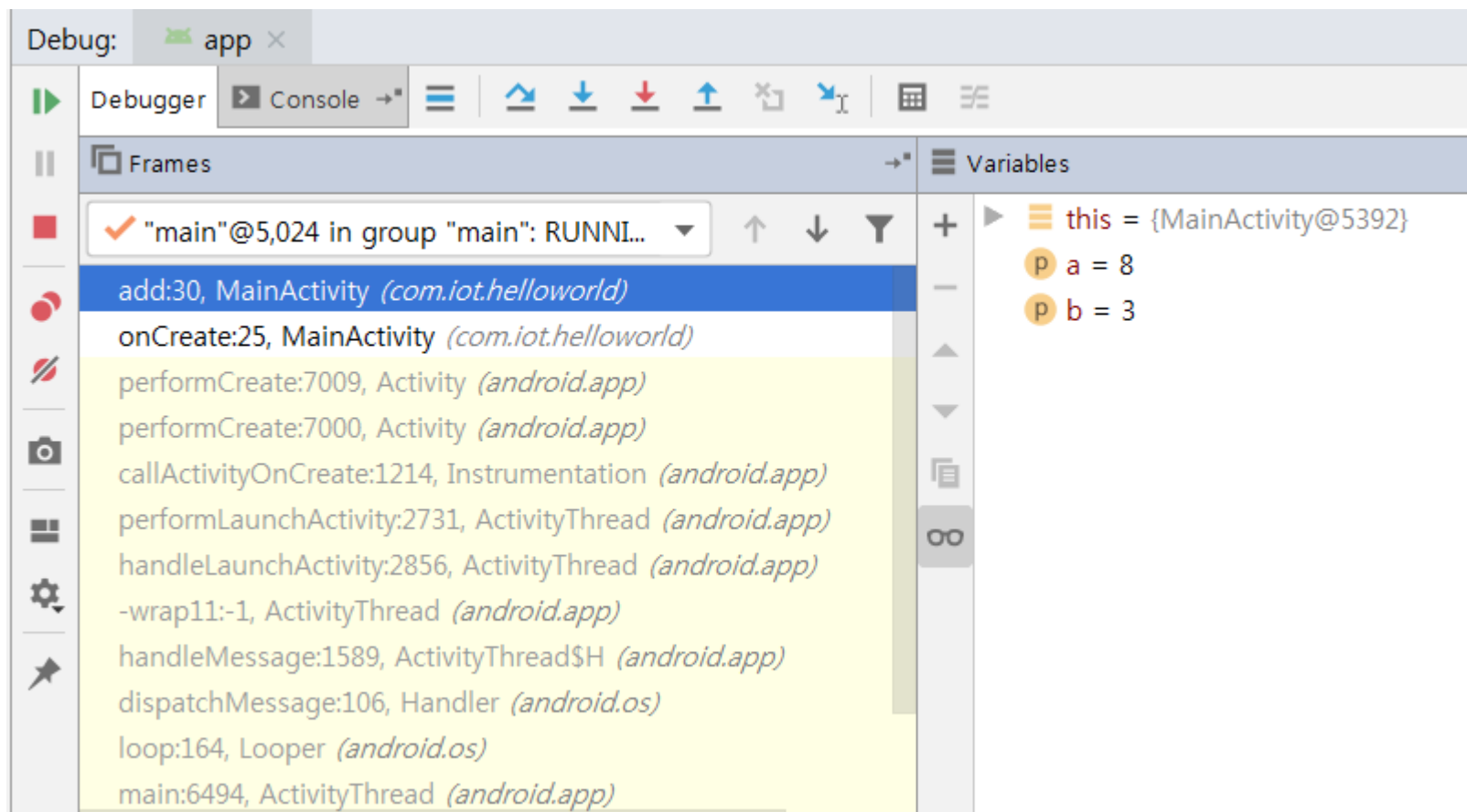
```
17  
18  int result = 0; result: 8  
19 int a = 2; a: 2  
20 int b = 3; b: 3  
21 int c = 5; c: 5  
22 result = a << 2; a: 2  
23 result += b; b: 3  
24 result = (result+c) >> 1; c: 5  
25 result = add(result, b: 3); result: 8  
26 textView.setText(String.valueOf(result));  
27 }
```

- Resume

- F9
- 현재 걸린 브레이크 포인트를 풀어버림
- onCreate() 메소드는 시작 시점에 한번만 불리므로 다시 테스트를 위해서는 앱을 재실행 시켜야 함

```
17  
18  int result = 0;  
19 int a = 2;  
20 int b = 3;  
21 int c = 5;  
22 result = a << 2;  
23 result += b;  
24 result = (result+c) >> 1;  
25 result = add(result, b: 3);  
26 textView.setText(String.valueOf(result));  
27 }
```

- 메소드의 호출 순서 관찰
 - 변수의 값도 함께 확인 가능



- 디버거를 이용해 스택에 쌓여있는 함수의 호출 과정 관찰
 - backtrace의 개념에 대한 이해
 - 모든 메소드는 호출이 될때 지역 변수 할당을 위한 스택 영역을 할당
 - 지역 변수가 스택 영역에 할당 되는 이유는 메소드의 호출 구조가 스택 구조와 일치하기 때문

- onCreate() 메소드가 a() 메소드를 호출 → a() 메소드가 b() 메소드 호출 시
 - onCreate() 메소드에서 할당되었던 지역변수 n1이 소멸되는 시점
 - onCreate()가 리턴되는 시점
 - 그 안에서 호출된 a() 메소드의 지역변수 n2가 소멸되는 시점
 - a() 메소드가 리턴되는 시점
 - a() 메소드에 의해 호출된 n3가 소멸되는 시점
 - b() 메소드가 소멸 되는 시점
 - 메소드의 호출 구조가 onCreate()에서 a()가 호출되었고, a()에 의해 b() 메소드가 호출 되었기 때문에 b()가 종료되기 전까지 a()를 종료 불가
 - 메소드의 종료 순서는 호출 순서와 반대
 - 메소드가 종료될 때 각 메소드를 위해 할당되었던 스택 영역도 pop 되면서 소멸
- 이러한 메소드의 호출 순서를 역으로 표시해 주는 것이 바로 back trace 기능
 - trace가 추적한다는 뜻
 - 메모리의 상태를 메소드 호출 역순으로 표시해 주는 기능