2017 Android of WMNTAA

Debug and Test

Debug

- Logcat
- Debugger
- Track object allocation

Logcat

- Overview
- API
- Example

Overview

The system log shows system messages while you debug your app. These messages include information from apps running on the device. If you want to use the system log to debug your app, make sure your code writes log messages and prints the stack trace for exceptions while your app is in the development phase.

API

- Log.v() : Verbose
- Log.d() : Debug
- Log.i() : Info
- Log.w() : Warning
- Log.e() : Error
- Log.wtf() : What a Terrible Failure

Example

```
import android.util.Log;
...
public class MyActivity extends Activity {
    private static final String TAG = MyActivity.class.getSimpleName();
    ...
    @Override
    public void onCreate(Bundle savedInstanceState) {
        if (savedInstanceState != null) {
            Log.d(TAG, "onCreate() Restoring previous state");
            /* restore state */
        } else {
            Log.d(TAG, "onCreate() No saved state available");
            /* initialize app */
        }
    }
}
```

Debugger

- Overview
- Debug types
- Work with breakpoints
- View and configure breakpoints
- Inspect variables
- View and change resource value display format

Overview

Android Studio includes a debugger that enables you to debug apps running on the Android Emulator or a connected Android device.

Debug Type

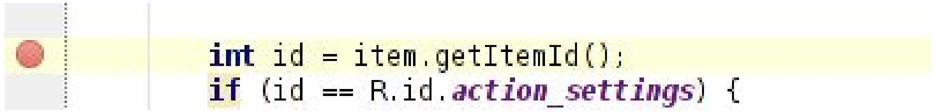
- Auto
  - Select if you want Android Studio to automatically choose the best option for the code you are debugging. For example, if you have any C or C++ code in your project, Android Studio automatically uses the Dual debug type. Otherwise, Android Studio uses the Java debug type.
- Java
  - Select if you want to debug only code written in Java—the Java debugger ignores any breakpoints or watches you set in your native code.

Debug Type

- Native
  - Select if you want to use only LLDB to debug your code. When using this debug type, the Java debugger session view is not available. By default, LLDB inspects only your native code and ignores breakpoints in your Java code. If you want to also debug your Java code, you should switch to either the Auto or Dual debug type.
- Dual
  - Select if you want to switch between debugging both Java and native code. Android Studio attaches both the Java debugger and LLDB to your app process, one for the Java debugger and one for LLDB, so you can inspect breakpoints in both your Java and native code without restarting your app or changing your debug configuration.

Work with breakpoints

- Locate the line of code where you want to pause execution, then either click the left gutter along that line of code or place the caret on the line and press Control+F8 (on Mac, Command+F8).
- If your app is already running, you don't need to update it to add the breakpoint—just click **Attach debugger to Android proccess** . Otherwise, start debugging by clicking **Debug** .



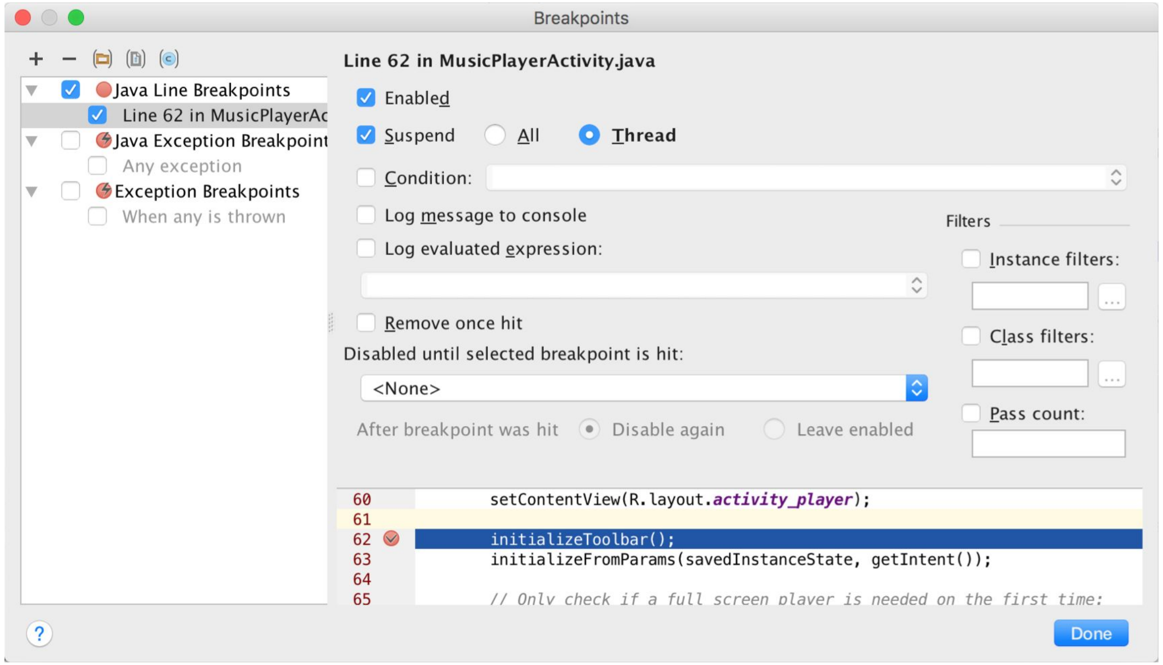
```
int id = item.getItemId();
if (id == R.id.action_settings) {
```

Work with breakpoints

- When your code execution reaches the breakpoint, Android Studio pauses execution of your app. You can then use the tools in the **Debugger** tab to identify the state of the app:
  - To examine the object tree for a variable, expand it in the **Variables** view. If the **Variables** view is not visible, click Restore Variables w .
  - To evaluate an expression at the current execution point, click **Evaluate Expression** .
  - To advance to the next line in the code (without entering a method), click **Step Over** .
  - To advance to the first line inside a method call, click **Step Into** .
  - To advance to the next line outside the current method, click **Step Out** .
  - To continue running the app normally, click **Resume Program** .

View and configure breakpoints

To view all the breakpoints and configure breakpoint settings, click **View Breakpoints** on the left side of the **Debug** window.



Line 62 in MusicPlayerActivity.java

[✓] Enabled

[✓] Suspend ( ) All [● Thread]

[ ] Condition:

[ ] Log message to console

[ ] Log evaluated expression:

[ ] Remove once hit

Disabled until selected breakpoint is hit:

<None>

After breakpoint was hit [● Disable again] ( ) Leave enabled

Filters

[ ] Instance filters:

[ ] Class filters:

[ ] Pass count:

```
        setContentView(R.layout.activity_player);

        initializeToolbar();
        initializeFromParams(savedInstanceState, getIntent());

        // Only check if a full screen player is needed on the first time:
```

Inspect variables

- To add a variable or expression to the Watches list, follow these steps:
    - Begin debugging.
    - In the **Watches** pane, click **Add** + .
    - In the text box that appears, type the name of the variable or expression you want to watch and then press Enter.
    - To remove an item from the **Watches** list, select the item and then click **Remove** − .
    - You can reorder the elements in the **Watches** list by selecting an item and then clicking **Up** ▲ or **Down** ▼ .

View and change resource value display format

- In debug mode, you can view resource values and select a different display format. With the **Variables** tab displayed and a frame selected, do the following:
  - In the **Variables** list, right-click anywhere on a resource line to display the drop-down list.
  - In the drop-down list, select **View as** and select the format you want to use.

View and change resource value display format

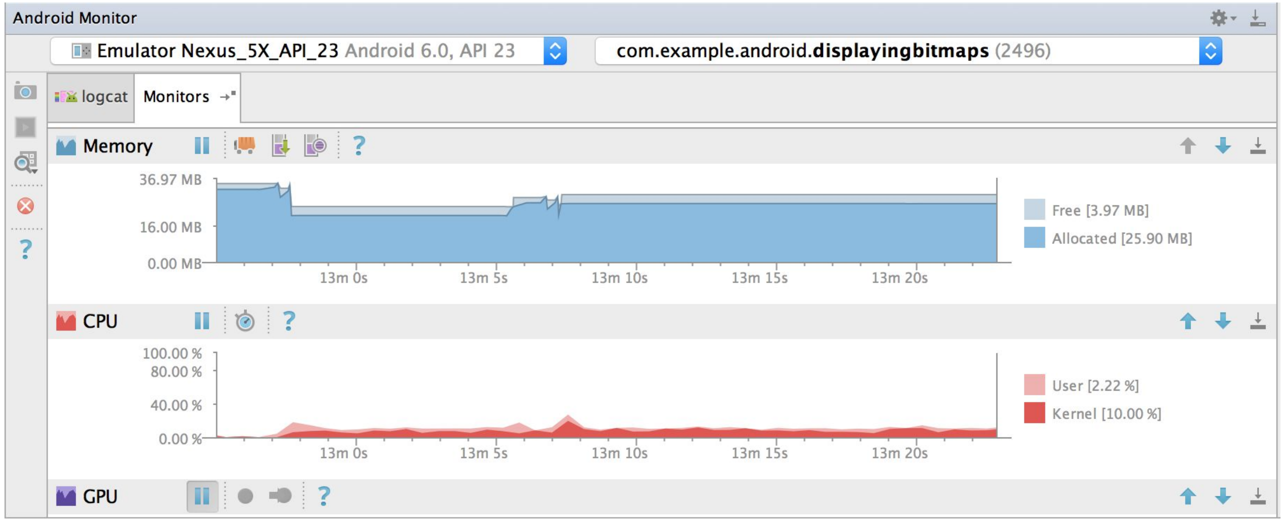
- The available formats depend on the data type of the resource you selected. You might see any one or more of the following options:
  - **Class**: Display the class definition.
  - **toString**: Display string format.
  - **Object**: Display the object (an instance of a class) definition.
  - **Array**: Display in an array format.
  - **Timestamp**: Display date and time as follows: yyyy-mm-dd hh:mm:ss.
  - **Auto**: Android Studio chooses the best format based on the data type.
  - **Binary**: Display a binary value using zeroes and ones.
  - **MeasureSpec**: The value passed from the parent to the selected child. See MeasureSpec.
  - **Hex**: Display as a hexadecimal value.
  - **Primitive**: Display as a numeric value using a primitive data type.
  - **Integer**: Display a numeric value of type Integer.

Track object allocation

- Overview
- Android Monitor

Overview

Android Studio lets you track objects that are being allocated on the Java heap and see which classes and threads are allocating these objects. This enables you to see the list of objects allocated during a period of interest. This information is valuable for assessing memory usage that can affect application performance.



Android Monitor

- Start your app as described in Run Your App in Debug Mode, and then select **View > Tool Windows > Android Monitor** (or click **Android Monitor** in the window bar).
- In the **Android Monitor** window, click the **Monitors** tab.
- At the top of the window, select your device and app process from the drop-down lists.
- In the **Memory** panel, click **Start Allocation Tracking** .
- Interact with your app on the device.
- Click the same button again to **Stop Allocation Tracking**.

Challenge

1. Use **Log.wtf()** to print some logging messages.
2. Create an arrayList and add an item into the list once 1 sec in a infinit loop. Add this list to **Watches** list to see how it changes and track the memory allocation in **Android Monitor**

Test

- Local unit tests
- Instrumented tests
- Add a new test
- Run a test

Local unit tests

- These are tests that run on your machine's local Java Virtual Machine (JVM). Use these tests to minimize execution time when your tests have no Android framework dependencies or when you can mock the Android framework dependencies.

Instrumented tests

- These are tests that run on a hardware device or emulator. These tests have access to Instrumentation APIs, give you access to information such as the Context of the app you are testing, and let you control the app under test from your test code. Use these tests when writing integration and functional UI tests to automate user interaction, or when your tests have Android dependencies that mock objects cannot satisfy.

Add a new test

- To create either a local unit test or an instrumented test, you can create a new test for a specific class or method by following these steps:
  - Open the Java file containing the code you want to test.
  - Click the class or method you want to test, then press Ctrl+Shift+T (⇧⌘T).
  - In the menu that appears, click **Create New Test**.
  - In the **Create Test** dialog, edit any fields and select any methods to generate, and then click OK.
  - In the **Choose Destination Directory** dialog, click the source set corresponding to the type of test you want to create: **androidTest** for an instrumented **test** or test for a local unit test. Then click **OK**.

Add a new test

- Be sure you specify the test library dependencies in your app module's build.gradle file:

```
dependencies {
    // Required for local unit tests (JUnit 4 framework)
    testCompile 'junit:junit:4.12'

    // Required for instrumented tests
    androidTestCompile 'com.android.support:support-annotations:24.0.0'
    androidTestCompile 'com.android.support.test:runner:0.5'
}
```

Run a test

- To run a test, proceed as follows:
  - Be sure your project is synchronized with Gradle by clicking **Sync Project** in the toolbar.
  - Run your test in one of the following ways:
    - In the **Project** window, right-click a test and click **Run** .
    - In the Code Editor, right-click a class or method in the test file and click **Run** to test all methods in the class.
    - To run all tests, right-click on the test directory and click **Run tests** .

Challenge

- Run a local test and an instrumented test.
- Use **Context** in the instrucmented test.