4-1-2012

Extending Java for Android Programming

Yoonsik Cheon
University of Texas at El Paso, ycheon@utep.edu

Comments:
Technical Report: UTEP-CS-12-13

Extending Java for Android Programming

Yoonsik Cheon

TR #12-13
April 2012

Keywords: application framework, domain specific language, Android, Java.

1998 CR Categories: D.2.3 [*Software Engineering*] Coding Tools and Techniques — Object-oriented programming; D.3.2 [*Programming Languages*] Language Classifications — Specialized application languages; D.3.3 [*Programming Languages*] Language Constructs and Features — Control structures, frameworks, classes and objects.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

Extending Java for Android Programming

(An Extended Abstract)

Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, U.S.A.
ycheon@utep.edu

***Abstract*—Android is one of the most popular platforms for developing mobile applications. However, its framework relies on programming conventions and styles to implement framework-specific concepts like activities and intents, causing problems such as reliability, readability, understandability, and maintainability. We propose to extend Java to support Android framework concepts explicitly as built-in language features. Our extension called *Android Java* will allow Android programmers to express these concepts in a more reliable, natural, and succinct way.**

I. Introduction

Android is an open-source and rapidly growing platform for developing applications running on mobile devices such as smartphones and tablet computers [1] [2]. It consists of a Linux-based kernel, libraries, and a Java-compatible application framework. The application framework provides a semi-complete application that can be specialized to produce a custom application quickly [3]. It defines conventions for extending classes provided by the framework so that newly added, application-specific classes can interact correctly with the framework classes as well as themselves.

However, the Android framework has a steep learning curve; it takes a long time to learn and be able to use the framework effectively. Reliability is a more serious issue. The framework relies on conventions to implement framework-specific concepts like activities and intents (see Section II). If these conventions are violated, an application may not work correctly. But, there is no automatic way of detecting such violations or enforcing the framework conventions.

In this position paper we propose a solution to the above problem. The key idea of our solution is to extend the Java programming language to support Android framework concepts like activities and intents as built-in language features by introducing a few new language constructs. The extension will allow one to not only express these concepts in a succinct and natural way but also check them automatically.

II. The Android Framework

Two fundamental concepts of the Android application framework are activities and intents. *Activities* are building blocks of Android programming in that an Android application consists of one or more activities. An activity is a single screen in an application, with supporting code. At runtime, there is a stack of activities, each created for one unique screen of the user interface. An activity may invoke another activity, and the invoked activity is pushed onto the top of the activity stack and becomes visible. The invoked activity is popped from the stack when its execution is finished, making the previous activity to resume its execution. Android runs each activity in a separate process each of which hosts a separate virtual machine. This is to provide a sandbox model of application execution to protect the system and other applications from badly-behaved code. One consequence of this decision is that an activity cannot directly invoke another activity.

```
public class MainActivity extends Activity {
  public void onCreate(Bundle savedState) {
    ...
    Intent i = new Intent("edu.utep.cs.GRADE");
    Bundle extras = new Bundle();
    extras.putString("name", "Joe");
    i.putExtras(extras);
    startActivityForResult(i, 0);
  }

  public void onActivityResult(int id, int o, Intent r) {
    if (id == 0 && o == RESULT_OK) {
      Bundle extras = r.getExtras();
      ... extras.getString("grade") ...
    }
  }
}

public class GradeActivity extends Activity {
  public void onCreate(Bundle savedState) {
    ...
    Bundle extras = getIntent().getExtras();
    if (extras != null) {
      String name = extras.getString("name");
      String grade = findGrade(name);
      Intent r = new Intent();
      Bundle extras = new Bundle();
      extras.putString("grade", grade);
      r.putExtras(extras);
      setResult(RESULT_OK, r);
      finish();
    }
  }
}
```

Fig. 1. Sample Android code

Android introduces another concept called an intent to combine and glue activities. An *intent* is a message to the Android system asking for performing a certain action on certain data. Upon receiving an intent, the system locates and starts an activity that can perform the requested action on the requested data. If an action requires additional data or returns results, they are piggy-backed on intents. In short, activities are invoked indirectly using intents, and intents are the core of the Android message system.

Figure 1 shows sample code illustrating the use of activities

```
public activity MainActivity {
  public void onCreate(Bundle savedState) {
    ...
    calls("edu.utep.cs.GRADE", "Joe")
      receiving(String grade) {
        ... grade ....
      };
  }
}

public activity GradeActivity {
  receives String name;
  provides String grade;

  public void onCreate(Bundle savedState) {
    ...
    String grade = findGrade(name);
    returns(grade);
  }
}
```

Fig. 2. The sample code rewritten in Android Java

and intents. It defines two activities, subclasses of the Activity class, and the first activity invokes the second. The framework method, `startActivityForResult`, invokes an activity by taking two arguments, an intent and a request code. The intent specifies the activity to be invoked along with optional arguments bundled as key-value pairs. The request code is an integer identifying a particular invocation. The main activity also defines a callback method, `onActivityResult`, to be invoked upon completion of the execution of an invoked activity. Because a single callback method handles all activity invocations, the request code—identifying the invocation—is provided as the first argument. As shown in the definition of the second activity class, results are returned by calling two framework methods, `setResult`, and `finish`. The `finish` method makes the control to return to the invoking activity and thus its callback method to get executed.

III. The Problem

The Android framework relies on conventions and programming styles to support the concepts of activities and intents, and this causes several problems.

- Reliability. Several factors contribute to this problem, including no parameter validation and no checking for framework conventions, e.g., overriding callback methods like `onActivityResult` and calling framework methods like `setResult` and `finish`. Manually bundling activity arguments is error prone, and missing definitions or statements may cause subtle errors that are often hard to detect and diagnose.
- Readability. The source code is not only verbose but also less readable, understandable, and maintainable. For example, the location where an activity is invoked and that of the results become available and used (`onActivityResult` method) are different. Note also that a single callback method handles the results of all invocations, resulting in error-prone case analysis code.
- Learning curve. Learning framework classes and their protocols—expected ways of using them, e.g., method overriding and calling—takes a time.

IV. Our Approach—Android Java

The key to our approach is to extend Java to support Android framework concepts as built-in language features. For this, we introduce a few new language constructs for activity declarations and invocations. Figure 2 shows the sample code from Section II rewritten in our extended Java, called *Android Java*. Activities are now built-in language concepts like classes as indicated with the use of the keyword **activity**. As shown in the GradeActivity activity, an activity declaration may include optional parameter declarations, **receives** and **provides** statements declaring input and output parameters. The activity parameter declarations specify the signature of an activity—the numbers of arguments and return values, their orders, and their types. The **returns** statement is used to return from an activity with optional results. An activity is invoked using the **calls** statement that specifies the name of the activity to be invoked, along with activity arguments; an optional **receiving** clause specifies the code to handle return values.

By mapping framework concepts to programming language constructs, Android Java addresses all the problems described previously. An explicit declaration of activity parameters will enable us to perform parameter validation, either statically or dynamically. An activity invocation and return is expressed in a more concise, natural, and readable way. One only needs to learn a few new language constructs that explicitly support the concepts of activities and intents.

V. Discussion

Android Java may be implemented in several ways including preprocessing, compiling, and annotations. Preprocessing is the easiest and quickest way to implement Android Java. Android Java code can be translated to plain Java code by essentially converting (a) the **calls** statement to a `startActivityForResult` method call wrapped with an appropriate check for parameter validation and (b) the **receiving** clause to the `onActivityResult` method with dispatching code. An Android Java compiler may be built to produce virtual machine code directly, by extending an open-source Java compiler like OpenJDK and Eclipse. Yet another possibility is to translate or express Android Java constructs in Java annotations and write an annotation preprocessor; however, this requires support for statement-level annotations.

Although we considered only activities and intents, there are many other concepts and features of the Android framework that could also be explicitly supported in Android Java, and the general problem is to map these features—currently supported in framework conventions and styles—to built-in language constructs in Java-based, domain specific programming languages.

References

[1] Google, "Android website," —http://www.android.com/—.

[2] D. Gavalas and D. Economou, "Development platforms for mobile applications: Status and trends," *IEEE Software*, vol. 28, no. 1, pp. 77–86, Jan.-Feb. 2011.

[3] R. E. Johnson, "Frameworks = (components + patterns)," *Communications of the ACM*, vol. 40, no. 10, pp. 39–42, Oct. 1997.