Data Binding Library

This document explains how to use the Data Binding Library to write declarative layouts and minimize the glue code necessary to bind your application logic and layouts.

The Data Binding Library offers both flexibility and broad compatibility — it's a support library, so you can use it with all Android platform versions back to Android 2.1 (API level 7+).

To use data binding, Android Plugin for Gradle 1.5.0-alpha1 or higher is required.

Build Environment

To get started with Data Binding, download the library from the Support repository in the Android SDK manager.

To configure your app to use data binding, add the `dataBinding` element to your `build.gradle` file in the app module.

Use the following code snippet to configure data binding:

```
android {
      ....
      dataBinding {
            enabled = true
      }
}
```

If you have an app module that depends on a library which uses data binding, your app module must configure data binding in its `build.gradle` file as well.

Also, make sure you are using a compatible version of Android Studio. Android Studio 1.3 and later provides support for data binding as described in Android Studio Support for Data Binding.

Data Binding Layout Files

Writing your first set of data binding expressions

Data-binding layout files are slightly different and start with a root tag of layout followed by a data element and a view root element. This view element is what your root would be in a non-binding layout file. A sample file looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <variable name="user" type="com.example.User"/>
    </data>
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.firstName}"/>
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.lastName}"/>
    </LinearLayout>
</layout>
```

The user variable within data describes a property that may be used within this layout.

```
<variable name="user" type="com.example.User"/>
```

Expressions within the layout are written in the attribute properties using the "@{}" syntax. Here, the TextView's text is set to the firstName property of user:

```
<TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.firstName}"/>
```

Data Object

Let's assume for now that you have a plain-old Java object (POJO) for User:

```
public class User {
    public final String firstName;
    public final String lastName;
    public User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

This type of object has data that never changes. It is common in applications to have data that is read once and never changes thereafter. It is also possible to use a JavaBeans objects:

```
public class User {
     private final String firstName;
     private final String lastName;
     public User(String firstName, String lastName) {
           this.firstName = firstName;
           this.lastName = lastName;
     }
     public String getFirstName() {
           return this.firstName;
     }
     public String getLastName() {
           return this.lastName;
     }
}
```

From the perspective of data binding, these two classes are equivalent. The expression `@{user.firstName}` used for the TextView's `android:text` attribute will access the `firstName` field in the former class and the `getFirstName()` method in the latter class. Alternatively, it will also be resolved to `firstName()` if that method exists.

Binding Data

By default, a Binding class will be generated based on the name of the layout file, converting it to Pascal case and suffixing "Binding" to it. The above layout file was `main_activity.xml` so the generate class was `MainActivityBinding`. This class holds all the bindings from the layout properties (e.g. the `user` variable) to the layout's Views and knows how to assign values for the binding expressions.The easiest means for creating the bindings is to do it while inflating:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     MainActivityBinding binding = DataBindingUtil.setContentView(this, R.layout.main_activity);
     User user = new User("Test", "User");
     binding.setUser(user);
}
```

You're done! Run the application and you'll see Test User in the UI. Alternatively, you can get the view via:

```
MainActivityBinding binding = MainActivityBinding.inflate(getLayoutInflater());
```

If you are using data binding items inside a ListView or RecyclerView adapter, you may prefer to use:

```
ListItemBinding binding = ListItemBinding.inflate(layoutInflater, viewGroup, false);
//or
ListItemBinding binding = DataBindingUtil.inflate(layoutInflater, R.layout.list_item, viewGroup,
false);
```

Event Handling

Data Binding allows you to write expressions handling events that are dispatched from the views (e.g. onClick). Event attribute names are governed by the name of the listener method with a few exceptions. For example, `View.OnLongClickListener` has a method `onLongClick()`, so the attribute for this event is `android:onLongClick`. There are two ways to handle an event.

- Method References: In your expressions, you can reference methods that conform to the signature of the listener method. When an expression evaluates to a method reference, Data Binding wraps the method reference and owner object in a listener, and sets that listener on the target view. If the expression evaluates to null, Data Binding does not create a listener and sets a null listener instead.
- Listener Bindings: These are lambda expressions that are evaluated when the event happens. Data Binding always creates a listener, which it sets on the view. When the event is dispatched, the listener evaluates the lambda expression.

Method References

Events can be bound to handler methods directly, similar to the way `android:onClick` can be assigned to a method in an Activity. One major advantage compared to the `View#onClick` attribute is that the expression is processed at compile time, so if the method does not exist or its signature is not correct, you receive a compile time error.

The major difference between Method References and Listener Bindings is that the actual listener implementation is created when the data is bound, not when the event is triggered. If you prefer to evaluate the expression when the event happens, you should use listener binding.

To assign an event to its handler, use a normal binding expression, with the value being the method name to call. For example, if your data object has two methods:

```
public class MyHandlers {
      public void onClickFriend(View view) { ... }
}
```

The binding expression can assign the click listener for a View:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
     <data>
```

```
        <variable name="handlers" type="com.example.Handlers"/>
        <variable name="user" type="com.example.User"/>
    </data>
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.firstName}"
            android:onClick="@{handlers::onClickFriend}"/>
    </LinearLayout>
</layout>
```

Note that the signature of the method in the expression must exactly match the signature of the method in the Listener object.

Listener Bindings

Listener Bindings are binding expressions that run when an event happens. They are similar to method references, but they let you run arbitrary data binding expressions. This feature is available with Android Gradle Plugin for Gradle version 2.0 and later.

In method references, the parameters of the method must match the parameters of the event listener. In Listener Bindings, only your return value must match the expected return value of the listener (unless it is expecting void). For example, you can have a presenter class that has the following method:

```
public class Presenter {
    public void onSaveClick(Task task){}
}
```

Then you can bind the click event to your class as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <variable name="task" type="com.android.example.Task" />
        <variable name="presenter" type="com.android.example.Presenter" />
    </data>
    <LinearLayout android:layout_width="match_parent" android:layout_height="match_parent">
        <Button android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:onClick="@{() -> presenter.onSaveClick(task)}" />
    </LinearLayout>
</layout>
```

Listeners are represented by lambda expressions that are allowed only as root elements of your expressions. When a callback is used in an expression, Data Binding automatically creates the necessary listener and registers for the event. When the view fires the event, Data Binding evaluates the given expression. As in regular binding expressions, you still get the null and thread safety of Data Binding while these listener expressions are being evaluated.

Note that in the example above, we haven't defined the `view` parameter that is passed into `onClick(android.view.View)`. Listener bindings provide two choices for listener parameters: you can either ignore all parameters to the method or name all of them. If you prefer to name the parameters, you can use them in your expression. For example, the expression above could be written as:

```
android:onClick="@{(view) -> presenter.onSaveClick(task)}"
```

Or if you wanted to use the parameter in the expression, it could work as follows:

```
public class Presenter {
      public void onSaveClick(View view, Task task){}
}
```

```
android:onClick="@{(theView) -> presenter.onSaveClick(theView, task)}"
```

You can use a lambda expression with more than one parameter:

```
public class Presenter {
      public void onCompletedChanged(Task task, boolean completed){}
}
```

```
<CheckBox android:layout_width="wrap_content" android:layout_height="wrap_content"
          android:onCheckedChanged="@{(cb, isChecked) -> presenter.completeChanged(task,
isChecked)}" />
```

If the event you are listening to returns a value whose type is not `void`, your expressions must return the same type of value as well. For example, if you want to listen for the long click event, your expression should return `boolean`.

```
public class Presenter {
      public boolean onLongClick(View view, Task task){}
}
```

```
android:onLongClick="@{(theView) -> presenter.onLongClick(theView, task)}"
```

If the expression cannot be evaluated due to `null` objects, Data Binding returns the default Java value for that type. For example, `null` for reference types, `0` for `int`, `false` for `boolean`, etc.

If you need to use an expression with a predicate (e.g. ternary), you can use `void` as a symbol.

```
android:onClick="@{(v) -> v.isVisible() ? doSomething() : void}"
```

Avoid Complex Listeners

Listener expressions are very powerful and can make your code very easy to read. On the other hand, listeners containing complex expressions make your layouts hard to read and unmaintainable. These expressions should be as simple as passing available data from your UI to your callback method. You should implement any business logic inside the callback method that you invoked from the listener expression.
Some specialized click event handlers exist and they need an attribute other than `android:onClick` to avoid a conflict. The following attributes have been created to avoid such conflicts:

Class	Listener Setter	Attribute
SearchView	setOnSearchClickListener(View.OnClickListener)	android:onSearchClick
ZoomControls	setOnZoomInClickListener(View.OnClickListener)	android:onZoomIn
ZoomControls	setOnZoomOutClickListener(View.OnClickListener)	android:onZoomOut

Layout Details

Imports

Zero or more `import` elements may be used inside the `data` element. These allow easy reference to classes inside your layout file, just like in Java.

```
<data>
      <import type="android.view.View"/>
</data>
```

Now, View may be used within your binding expression:

```
<TextView
     android:text="@{user.lastName}"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:visibility="@{user.isAdult ? View.VISIBLE : View.GONE}"/>
```

When there are class name conflicts, one of the classes may be renamed to an "alias:"

```
<import type="android.view.View"/>
```

```
<import type="com.example.real.estate.View"
            alias="Vista"/>
```

Now, `Vista` may be used to reference the `com.example.real.estate.View` and `View` may be used to reference `android.view.View` within the layout file. Imported types may be used as type references in variables and expressions:

```
<data>
      <import type="com.example.User"/>
      <import type="java.util.List"/>
      <variable name="user" type="User"/>
      <variable name="userList" type="List<User>"/>
</data>
```

Note: Android Studio does not yet handle imports so the autocomplete for imported variables may not work in your IDE. Your application will still compile fine and you can work around the IDE issue by using fully qualified names in your variable definitions.

```
<TextView
     android:text="@{((User)(user.connection)).lastName}"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"/>
```

Imported types may also be used when referencing static fields and methods in expressions:

```
<data>
      <import type="com.example.MyStringUtils"/>
      <variable name="user" type="com.example.User"/>
</data>
...
<TextView
     android:text="@{MyStringUtils.capitalize(user.lastName)}"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"/>
```

Just as in Java, `java.lang.*` is imported automatically.

Variables

Any number of `variable` elements may be used inside the `data` element. Each `variable` element describes a property that may be set on the layout to be used in binding expressions within the layout file.

```
<data>
      <import type="android.graphics.drawable.Drawable"/>
      <variable name="user"   type="com.example.User"/>
      <variable name="image" type="Drawable"/>
      <variable name="note"   type="String"/>
</data>
```

The variable types are inspected at compile time, so if a variable implements `Observable` or is an observable collection, that should be reflected in the type. If the variable is a base class or interface that does not implement the Observable* interface, the variables will not be observed!

When there are different layout files for various configurations (e.g. landscape or portrait), the variables will be combined. There must not be conflicting variable definitions between these layout files.

The generated binding class will have a setter and getter for each of the described variables. The variables will take the default Java values until the setter is called — `null` for reference types, `0` for `int`, `false` for `boolean`, etc.

A special variable named `context` is generated for use in binding expressions as needed. The value for `context` is the `Context` from the root View's `getContext()`. The `context` variable will be overridden by an explicit variable declaration with that name.

Custom Binding Class Names

By default, a Binding class is generated based on the name of the layout file, starting it with upper-case, removing underscores (_) and capitalizing the following letter and then suffixing "Binding". This class will be placed in a databinding package under the module package. For example, the layout file `contact_item.xml` will generate `ContactItemBinding`. If the module package is `com.example.my.app`, then it will be placed in `com.example.my.app.databinding`.

Binding classes may be renamed or placed in different packages by adjusting the `class` attribute of the `data` element. For example:

```
<data class="ContactItem">
      ...
</data>
```

This generates the binding class as `ContactItem` in the databinding package in the module package. If the class should be generated in a different package within the module package, it may be prefixed with ".":

```
<data class=".ContactItem">
      ...
```

```
</data>
```

In this case, `ContactItem` is generated in the module package directly. Any package may be used if the full package is provided:

```
<data class="com.example.ContactItem">
     ...
</data>
```

Includes

Variables may be passed into an included layout's binding from the containing layout by using the application namespace and the variable name in an attribute:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
           xmlns:bind="http://schemas.android.com/apk/res-auto">
    <data>
          <variable name="user" type="com.example.User"/>
    </data>
    <LinearLayout
          android:orientation="vertical"
          android:layout_width="match_parent"
          android:layout_height="match_parent">
          <include layout="@layout/name"
                bind:user="@{user}"/>
          <include layout="@layout/contact"
                bind:user="@{user}"/>
    </LinearLayout>
</layout>
```

Here, there must be a `user` variable in both the `name.xml` and `contact.xml` layout files.

Data binding does not support include as a direct child of a merge element. For example, the following layout is not supported:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
           xmlns:bind="http://schemas.android.com/apk/res-auto">
    <data>
          <variable name="user" type="com.example.User"/>
    </data>
    <merge>
          <include layout="@layout/name"
```

```
                bind:user="@{user}"/>
        <include layout="@layout/contact"
                bind:user="@{user}"/>
    </merge>
</layout>
```

Expression Language

Common Features

The expression language looks a lot like a Java expression. These are the same:

- Mathematical `+ - / * %`
- String concatenation `+`
- Logical `&& ||`
- Binary `& | ^`
- Unary `+ - ! ~`
- Shift `>> >>> <<`
- Comparison `== > < >= <=`
- `instanceof`
- Grouping `()`
- Literals - character, String, numeric, `null`
- Cast
- Method calls
- Field access
- Array access `[]`
- Ternary operator `?:`

Examples:

```
android:text="@{String.valueOf(index + 1)}"
android:visibility="@{age < 13 ? View.GONE : View.VISIBLE}"
android:transitionName='@{"image_" + id}'
```

Missing Operations

A few operations are missing from the expression syntax that you can use in Java.

- `this`
- `super`
- `new`
- Explicit generic invocation

Null Coalescing Operator

The null coalescing operator (`??`) chooses the left operand if it is not null or the right if it is null.

```
android:text="@{user.displayName ?? user.lastName}"
```

This is functionally equivalent to:

```
android:text="@{user.displayName != null ? user.displayName : user.lastName}"
```

Property Reference

The first was already discussed in the Writing your first data binding expressions above: short form JavaBean references. When an expression references a property on a class, it uses the same format for fields, getters, and ObservableFields.

```
android:text="@{user.lastName}"
```

Avoiding NullPointerException

Generated data binding code automatically checks for nulls and avoid null pointer exceptions. For example, in the expression `@{user.name}`, if `user` is null, `user.name` will be assigned its default value (null). If you were referencing `user.age`, where age is an `int`, then it would default to 0.

Collections

Common collections: arrays, lists, sparse lists, and maps, may be accessed using the `[]` operator for convenience.

```
<data>
      <import type="android.util.SparseArray"/>
      <import type="java.util.Map"/>
      <import type="java.util.List"/>
      <variable name="list" type="List<String>"/>
      <variable name="sparse" type="SparseArray<String>"/>
      <variable name="map" type="Map<String, String>"/>
      <variable name="index" type="int"/>
      <variable name="key" type="String"/>
</data>
...
android:text="@{list[index]}"
...
android:text="@{sparse[index]}"
...
android:text="@{map[key]}"
```

String Literals

When using single quotes around the attribute value, it is easy to use double quotes in the expression:

```
android:text='@{map["firstName"]}'
```

It is also possible to use double quotes to surround the attribute value. When doing so, String literals should either use the ' or back quote (`).

```
android:text="@{map[`firstName`]}"
android:text="@{map['firstName']}"
```

Resources

It is possible to access resources as part of expressions using the normal syntax:

```
android:padding="@{large? @dimen/largePadding : @dimen/smallPadding}"
```

Format strings and plurals may be evaluated by providing parameters:

```
android:text="@{@string/nameFormat(firstName, lastName)}"
android:text="@{@plurals/banana(bananaCount)}"
```

When a plural takes multiple parameters, all parameters should be passed:

```

   Have an orange
   Have %d oranges

android:text="@{@plurals/orange(orangeCount, orangeCount)}"
```

Some resources require explicit type evaluation.

Type	Normal Reference	Expression Reference
String[]	@array	@stringArray
int[]	@array	@intArray
TypedArray	@array	@typedArray
Animator	@animator	@animator
StateListAnimator	@animator	@stateListAnimator

color int	@color	@color
ColorStateList	@color	@colorStateList

Data Objects

Any plain old Java object (POJO) may be used for data binding, but modifying a POJO will not cause the UI to update. The real power of data binding can be used by giving your data objects the ability to notify when data changes. There are three different data change notification mechanisms, Observable objects, observable fields, and observable collections.

When one of these observable data object is bound to the UI and a property of the data object changes, the UI will be updated automatically.

Observable Objects

A class implementing the Observable interface will allow the binding to attach a single listener to a bound object to listen for changes of all properties on that object.

The Observable interface has a mechanism to add and remove listeners, but notifying is up to the developer. To make development easier, a base class, BaseObservable, was created to implement the listener registration mechanism. The data class implementer is still responsible for notifying when the properties change. This is done by assigning a Bindable annotation to the getter and notifying in the setter.

```
private static class User extends BaseObservable {
    private String firstName;
    private String lastName;
    @Bindable
    public String getFirstName() {
        return this.firstName;
    }
    @Bindable
    public String getLastName() {
        return this.lastName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
        notifyPropertyChanged(BR.firstName);
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
        notifyPropertyChanged(BR.lastName);
    }
```

```
}
```

The Bindable annotation generates an entry in the BR class file during compilation. The BR class file will be generated in the module package. If the base class for data classes cannot be changed, the Observable interface may be implemented using the convenient PropertyChangeRegistry to store and notify listeners efficiently.

ObservableFields

A little work is involved in creating Observable classes, so developers who want to save time or have few properties may use ObservableField and its siblings ObservableBoolean, ObservableByte, ObservableChar, ObservableShort, ObservableInt, ObservableLong, ObservableFloat, ObservableDouble, and ObservableParcelable. ObservableFields are self-contained observable objects that have a single field. The primitive versions avoid boxing and unboxing during access operations. To use, create a public final field in the data class:

```
private static class User {
     public final ObservableField<String> firstName =
           new ObservableField<>();
     public final ObservableField<String> lastName =
           new ObservableField<>();
     public final ObservableInt age = new ObservableInt();
}
```

That's it! To access the value, use the set and get accessor methods:

```
user.firstName.set("Google");
int age = user.age.get();
```

Observable Collections

Some applications use more dynamic structures to hold data. Observable collections allow keyed access to these data objects. ObservableArrayMap is useful when the key is a reference type, such as String.

```
ObservableArrayMap<String, Object> user = new ObservableArrayMap<>();
user.put("firstName", "Google");
user.put("lastName", "Inc.");
user.put("age", 17);
```

In the layout, the map may be accessed through the String keys:

```
<data>
```

```
      <import type="android.databinding.ObservableMap"/>
      <variable name="user" type="ObservableMap<String, Object>"/>
</data>
...
<TextView
     android:text='@{user["lastName"]}'
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"/>
<TextView
     android:text='@{String.valueOf(1 + (Integer)user["age"])}'
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"/>
```

ObservableArrayList is useful when the key is an integer:

```
ObservableArrayList<Object> user = new ObservableArrayList<>();
user.add("Google");
user.add("Inc.");
user.add(17);
```

In the layout, the list may be accessed through the indices:

```
<data>
      <import type="android.databinding.ObservableList"/>
      <import type="com.example.my.app.Fields"/>
      <variable name="user" type="ObservableList<Object>"/>
</data>
...
<TextView
     android:text='@{user[Fields.LAST_NAME]}'
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"/>
<TextView
     android:text='@{String.valueOf(1 + (Integer)user[Fields.AGE])}'
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"/>
```

Generated Binding

The generated binding class links the layout variables with the Views within the layout. As discussed earlier, the name and package of the Binding may be customized. The Generated binding classes all extend ViewDataBinding.

Creating

The binding should be created soon after inflation to ensure that the View hierarchy is not disturbed prior to binding to the Views with expressions within the layout. There are a few ways to bind to a layout. The most common is to use the static methods on the Binding class.The inflate method inflates the View hierarchy and binds to it all it one step. There is a simpler version that only takes a LayoutInflater and one that takes a ViewGroup as well:

```
MyLayoutBinding binding = MyLayoutBinding.inflate(layoutInflater);
MyLayoutBinding binding = MyLayoutBinding.inflate(layoutInflater, viewGroup, false);
```

If the layout was inflated using a different mechanism, it may be bound separately:

```
MyLayoutBinding binding = MyLayoutBinding.bind(viewRoot);
```

Sometimes the binding cannot be known in advance. In such cases, the binding can be created using the DataBindingUtil class:

```
ViewDataBinding binding = DataBindingUtil.inflate(LayoutInflater, layoutId,
      parent, attachToParent);
ViewDataBinding binding = DataBindingUtil.bindTo(viewRoot, layoutId);
```

Views With IDs

A public final field will be generated for each View with an ID in the layout. The binding does a single pass on the View hierarchy, extracting the Views with IDs. This mechanism can be faster than calling findViewById for several Views. For example:

```
<layout xmlns:android="http://schemas.android.com/apk/res/android">
     <data>
           <variable name="user" type="com.example.User"/>
     </data>
     <LinearLayout
           android:orientation="vertical"
           android:layout_width="match_parent"
           android:layout_height="match_parent">
           <TextView android:layout_width="wrap_content"
                 android:layout_height="wrap_content"
                 android:text="@{user.firstName}"
     android:id="@+id/firstName"/>
           <TextView android:layout_width="wrap_content"
                 android:layout_height="wrap_content"
                 android:text="@{user.lastName}"
```

```
   android:id="@+id/lastName"/>
     </LinearLayout>
</layout>
```

Will generate a binding class with:

```
public final TextView firstName;
public final TextView lastName;
```

IDs are not nearly as necessary as without data binding, but there are still some instances where access to Views are still necessary from code.

Variables

Each variable will be given accessor methods.

```
<data>
      <import type="android.graphics.drawable.Drawable"/>
      <variable name="user"   type="com.example.User"/>
      <variable name="image" type="Drawable"/>
      <variable name="note"   type="String"/>
</data>
```

will generate setters and getters in the binding:

```
public abstract com.example.User getUser();
public abstract void setUser(com.example.User user);
public abstract Drawable getImage();
public abstract void setImage(Drawable image);
public abstract String getNote();
public abstract void setNote(String note);
```

ViewStubs

ViewStubs are a little different from normal Views. They start off invisible and when they either are made visible or are explicitly told to inflate, they replace themselves in the layout by inflating another layout.

Because the ViewStub essentially disappears from the View hierarchy, the View in the binding object must also disappear to allow collection. Because the Views are final, a ViewStubProxy object takes the place of the ViewStub, giving the developer access to the ViewStub when it exists and also access to the inflated View hierarchy when the ViewStub has been inflated.

When inflating another layout, a binding must be established for the new layout. Therefore, the

`ViewStubProxy` must listen to the `ViewStub`'s `ViewStub.OnInflateListener` and establish the binding at that time. Since only one can exist, the `ViewStubProxy` allows the developer to set an `OnInflateListener` on it that it will call after establishing the binding.

Advanced Binding

Dynamic Variables

At times, the specific binding class won't be known. For example, a `RecyclerView.Adapter` operating against arbitrary layouts won't know the specific binding class. It still must assign the binding value during the `onBindViewHolder(VH, int)`.

In this example, all layouts that the RecyclerView binds to have an "item" variable. The `BindingHolder` has a `getBinding` method returning the `ViewDataBinding` base.

```
public void onBindViewHolder(BindingHolder holder, int position) {
    final T item = mItems.get(position);
    holder.getBinding().setVariable(BR.item, item);
    holder.getBinding().executePendingBindings();
}
```

Immediate Binding

When a variable or observable changes, the binding will be scheduled to change before the next frame. There are times, however, when binding must be executed immediately. To force execution, use the `executePendingBindings()` method.

Background Thread

You can change your data model in a background thread as long as it is not a collection. Data binding will localize each variable / field while evaluating to avoid any concurrency issues.

Attribute Setters

Whenever a bound value changes, the generated binding class must call a setter method on the View with the binding expression. The data binding framework has ways to customize which method to call to set the value.

Automatic Setters

For an attribute, data binding tries to find the method setAttribute. The namespace for the attribute does not matter, only the attribute name itself.
For example, an expression associated with TextView's attribute `android:text` will look for a setText(String). If the expression returns an int, data binding will search for a setText(int) method. Be careful to have the expression return the correct type, casting if necessary. Note that data binding will work even if no attribute exists with the given name. You can then easily "create"

attributes for any setter by using data binding. For example, support DrawerLayout doesn't have any attributes, but plenty of setters. You can use the automatic setters to use one of these.

```
<android.support.v4.widget.DrawerLayout
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      app:scrimColor="@{@color/scrim}"
      app:drawerListener="@{fragment.drawerListener}"/>
```

Renamed Setters

Some attributes have setters that don't match by name. For these methods, an attribute may be associated with the setter through `BindingMethods` annotation. This must be associated with a class and contains `BindingMethod` annotations, one for each renamed method. For example, the `android:tint` attribute is really associated with `setImageTintList(ColorStateList)`, not `setTint`.

```
@BindingMethods({
           @BindingMethod(type = "android.widget.ImageView",
                                  attribute = "android:tint",
                                  method = "setImageTintList"),
})
```

It is unlikely that developers will need to rename setters; the android framework attributes have already been implemented.

Custom Setters

Some attributes need custom binding logic. For example, there is no associated setter for the `android:paddingLeft` attribute. Instead, `setPadding(left, top, right, bottom)` exists. A static binding adapter method with the `BindingAdapter` annotation allows the developer to customize how a setter for an attribute is called.

The android attributes have already had `BindingAdapter`s created. For example, here is the one for `paddingLeft`:

```
@BindingAdapter("android:paddingLeft")
public static void setPaddingLeft(View view, int padding) {
     view.setPadding(padding,
                              view.getPaddingTop(),
                              view.getPaddingRight(),
                              view.getPaddingBottom());
}
```

Binding adapters are useful for other types of customization. For example, a custom loader can be

called off-thread to load an image.

Developer-created binding adapters will override the data binding default adapters when there is a conflict.

You can also have adapters that receive multiple parameters.

```
@BindingAdapter({"bind:imageUrl", "bind:error"})
public static void loadImage(ImageView view, String url, Drawable error) {
     Picasso.with(view.getContext()).load(url).error(error).into(view);
}
```

```
<ImageView app:imageUrl="@{venue.imageUrl}"
app:error="@{@drawable/venueError}"/>
```

This adapter will be called if both imageUrl and error are used for an ImageView and imageUrl is a string and error is a drawable.

- Custom namespaces are ignored during matching.
- You can also write adapters for android namespace.

Binding adapter methods may optionally take the old values in their handlers. A method taking old and new values should have all old values for the attributes come first, followed by the new values:

```
@BindingAdapter("android:paddingLeft")
public static void setPaddingLeft(View view, int oldPadding, int newPadding) {
     if (oldPadding != newPadding) {
           view.setPadding(newPadding,
                                 view.getPaddingTop(),
                                 view.getPaddingRight(),
                                 view.getPaddingBottom());
     }
}
```

Event handlers may only be used with interfaces or abstract classes with one abstract method. For example:

```
@BindingAdapter("android:onLayoutChange")
public static void setOnLayoutChangeListener(View view, View.OnLayoutChangeListener oldValue,
           View.OnLayoutChangeListener newValue) {
      if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
            if (oldValue != null) {
                  view.removeOnLayoutChangeListener(oldValue);
            }
```

```
            if (newValue != null) {
                  view.addOnLayoutChangeListener(newValue);
            }
      }
}
```

When a listener has multiple methods, it must be split into multiple listeners. For example, View.OnAttachStateChangeListener has two methods: onViewAttachedToWindow() and onViewDetachedFromWindow(). We must then create two interfaces to differentiate the attributes and handlers for them.

```
@TargetApi(VERSION_CODES.HONEYCOMB_MR1)
public interface OnViewDetachedFromWindow {
      void onViewDetachedFromWindow(View v);
}

@TargetApi(VERSION_CODES.HONEYCOMB_MR1)
public interface OnViewAttachedToWindow {
      void onViewAttachedToWindow(View v);
}
```

Because changing one listener will also affect the other, we must have three different binding adapters, one for each attribute and one for both, should they both be set.

```
@BindingAdapter("android:onViewAttachedToWindow")
public static void setListener(View view, OnViewAttachedToWindow attached) {
      setListener(view, null, attached);
}

@BindingAdapter("android:onViewDetachedFromWindow")
public static void setListener(View view, OnViewDetachedFromWindow detached) {
      setListener(view, detached, null);
}

@BindingAdapter({"android:onViewDetachedFromWindow", "android:onViewAttachedToWindow"})
public static void setListener(View view, final OnViewDetachedFromWindow detach,
            final OnViewAttachedToWindow attach) {
      if (VERSION.SDK_INT >= VERSION_CODES.HONEYCOMB_MR1) {
            final OnAttachStateChangeListener newListener;
            if (detach == null && attach == null) {
                  newListener = null;
            } else {
                  newListener = new OnAttachStateChangeListener() {
```

```
                        @Override
                        public void onViewAttachedToWindow(View v) {
                              if (attach != null) {
                                    attach.onViewAttachedToWindow(v);
                              }
                        }

                        @Override
                        public void onViewDetachedFromWindow(View v) {
                              if (detach != null) {
                                    detach.onViewDetachedFromWindow(v);
                              }
                        }
                  };
            }
            final OnAttachStateChangeListener oldListener = ListenerUtil.trackListener(view,
                        newListener, R.id.onAttachStateChangeListener);
            if (oldListener != null) {
                  view.removeOnAttachStateChangeListener(oldListener);
            }
            if (newListener != null) {
                  view.addOnAttachStateChangeListener(newListener);
            }
      }
}
```

The above example is slightly more complicated than normal because View uses add and remove for the listener instead of a set method for `View.OnAttachStateChangeListener`. The `android.databinding.adapters.ListenerUtil` class helps keep track of the previous listeners so that they may be removed in the Binding Adaper.

By annotating the interfaces `OnViewDetachedFromWindow` and `OnViewAttachedToWindow` with `@TargetApi(VERSION_CODES.HONEYCOMB_MR1)`, the data binding code generator knows that the listener should only be generated when running on Honeycomb MR1 and new devices, the same version supported by `addOnAttachStateChangeListener(View.OnAttachStateChangeListener)`.

Converters

Object Conversions

When an Object is returned from a binding expression, a setter will be chosen from the automatic, renamed, and custom setters. The Object will be cast to a parameter type of the chosen setter.

This is a convenience for those using ObservableMaps to hold data. for example:

```
<TextView
    android:text='@{userMap["lastName"]}'
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

The `userMap` returns an Object and that Object will be automatically cast to parameter type found in the setter `setText(CharSequence)`. When there may be confusion about the parameter type, the developer will need to cast in the expression.

Custom Conversions

Sometimes conversions should be automatic between specific types. For example, when setting the background:

```
<View
    android:background="@{isError ? @color/red : @color/white}"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

Here, the background takes a `Drawable`, but the color is an integer. Whenever a `Drawable` is expected and an integer is returned, the `int` should be converted to a `ColorDrawable`. This conversion is done using a static method with a BindingConversion annotation:

```
@BindingConversion
public static ColorDrawable convertColorToDrawable(int color) {
    return new ColorDrawable(color);
}
```

Note that conversions only happen at the setter level, so it is not allowed to mix types like this:

```
<View
    android:background="@{isError ? @drawable/error : @color/white}"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

Android Studio Support for Data Binding

Android Studio supports many of the code editing features for data binding code. For example, it supports the following features for data binding expressions:

- Syntax highlighting
- Flagging of expression language syntax errors
- XML code completion

- References, including navigation (such as navigate to a declaration) and quick documentation

Note: Arrays and a generic type, such as the Observable class, might display errors when there are no errors.

The Preview pane displays default values for data binding expressions if provided. In the following example excerpt of an element from a layout XML file, the Preview pane displays the PLACEHOLDER default text value in the TextView.

```
<TextView android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="@{user.firstName, default=PLACEHOLDER}"/>
```

If you need to display a default value during the design phase of your project, you can also use tools attributes instead of default expression values, as described in Designtime Layout Attributes.

Get news & tips
Sign up for the newsletter