Butter Knife

Field and method binding for Android views

Javadoc (javadoc/) · StackOverflow (http://stackoverflow.com/questions/ask?tags=butterknife)

Introduction

Annotate fields with `@Bind` and a view ID for Butter Knife to find and automatically cast the corresponding view in your layout.

```
class ExampleActivity extends Activity {
  @Bind(R.id.title) TextView title;
  @Bind(R.id.subtitle) TextView subtitle;
  @Bind(R.id.footer) TextView footer;

  @Override public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.simple_activity);
    ButterKnife.bind(this);
    // TODO Use fields...
  }
}
```

Instead of slow reflection, code is generated to perform the view look-ups. Calling `bind` delegates to this generated code that you can see and debug.

The generated code for the above example is roughly equivalent to the following:

```
public void bind(ExampleActivity activity) {
  activity.subtitle = (android.widget.TextView) activity.findViewById(2130968578);
  activity.footer = (android.widget.TextView) activity.findViewById(2130968579);
  activity.title = (android.widget.TextView) activity.findViewById(2130968577);
}
```

RESOURCE BINDING

Bind pre-defined resources with `@BindBool` , `@BindColor` , `@BindDimen` , `@BindDrawable` , `@BindInt` , `@BindString` , which binds an `R.bool` ID (or your specified type) to its corresponding field.

```
class ExampleActivity extends Activity {
  @BindString(R.string.title) String title;
  @BindDrawable(R.drawable.graphic) Drawable graphic;
  @BindColor(R.color.red) int red; // int or ColorStateList field
  @BindDimen(R.dimen.spacer) Float spacer; // int (for pixel size) or float (for exact v
  // ...
}
```

NON-ACTIVITY BINDING

You can also perform binding on arbitrary objects by supplying your own view root.

```
public class FancyFragment extends Fragment {
  @Bind(R.id.button1) Button button1;
  @Bind(R.id.button2) Button button2;

  @Override public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundl
    View view = inflater.inflate(R.layout.fancy_fragment, container, false);
    ButterKnife.bind(this, view);
    // TODO Use fields...
    return view;
  }
}
```

Another use is simplifying the view holder pattern inside of a list adapter.

```
public class MyAdapter extends BaseAdapter {
  @Override public View getView(int position, View view, ViewGroup parent) {
    ViewHolder holder;
    if (view != null) {
      holder = (ViewHolder) view.getTag();
    } else {
      view = inflater.inflate(R.layout.whatever, parent, false);
      holder = new ViewHolder(view);
      view.setTag(holder);
    }

    holder.name.setText("John Doe");
    // etc...

    return view;
  }

  static class ViewHolder {
    @Bind(R.id.title) TextView name;
    @Bind(R.id.job_title) TextView jobTitle;

    public ViewHolder(View view) {
      ButterKnife.bind(this, view);
    }
  }
}
```

You can see this implementation in action in the provided sample.

Calls to `ButterKnife.bind` can be made anywhere you would otherwise put `findViewById` calls.

Other provided binding APIs:

- Bind arbitrary objects using an activity as the view root. If you use a pattern like MVC you can bind the controller using its activity with `ButterKnife.bind(this, activity)` .

- Bind a view's children into fields using `ButterKnife.bind(this)` . If you use `<merge>` tags in a layout and inflate in a custom view constructor you can call this immediately after. Alternatively, custom view types inflated from XML can use it in the `onFinishInflate()` callback.

VIEW LISTS

You can group multiple views into a `List` or array.

```
@Bind({ R.id.first_name, R.id.middle_name, R.id.last_name })
List<EditText> nameViews;
```

The `apply` method allows you to act on all the views in a list at once.

```
ButterKnife.apply(nameViews, DISABLE);
ButterKnife.apply(nameViews, ENABLED, false);
```

`Action` and `Setter` interfaces allow specifying simple behavior.

```
static final ButterKnife.Action<View> DISABLE = new ButterKnife.Action<View>() {
  @Override public void apply(View view, int index) {
    view.setEnabled(false);
  }
};
static final ButterKnife.Setter<View, Boolean> ENABLED = new ButterKnife.Setter<View, Bo
  @Override public void set(View view, Boolean value, int index) {
    view.setEnabled(value);
  }
};
```

An Android `Property` (https://developer.android.com/reference/android/util/Property.html) can also be used with the `apply` method.

```
ButterKnife.apply(nameViews, View.ALPHA, 0.0f);
```

LISTENER BINDING

Listeners can also automatically be configured onto methods.

```
@OnClick(R.id.submit)
public void submit(View view) {
  // TODO submit data to server...
}
```

All arguments to the listener method are optional.

```
@OnClick(R.id.submit)
public void submit() {
  // TODO submit data to server...
}
```

Define a specific type and it will automatically be cast.

```
@OnClick(R.id.submit)
public void sayHi(Button button) {
  button.setText("Hello!");
}
```

Specify multiple IDs in a single binding for common event handling.

```
@OnClick({ R.id.door1, R.id.door2, R.id.door3 })
public void pickDoor(DoorView door) {
  if (door.hasPrizeBehind()) {
    Toast.makeText(this, "You win!", LENGTH_SHORT).show();
  } else {
    Toast.makeText(this, "Try again", LENGTH_SHORT).show();
  }
}
```

Custom views can bind to their own listeners by not specifying an ID.

```
public class FancyButton extends Button {
  @OnClick
  public void onClick() {
    // TODO do something!
  }
}
```

BINDING RESET

Fragments have a different view lifecycle than activities. When binding a fragment in `onCreateView`, set the views to `null` in `onDestroyView`. Butter Knife has an `unbind` method to do this automatically.

```
public class FancyFragment extends Fragment {
  @Bind(R.id.button1) Button button1;
  @Bind(R.id.button2) Button button2;

  @Override public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundl
    View view = inflater.inflate(R.layout.fancy_fragment, container, false);
    ButterKnife.bind(this, view);
    // TODO Use fields...
    return view;
  }

  @Override public void onDestroyView() {
    super.onDestroyView();
    ButterKnife.unbind(this);
  }
}
```

OPTIONAL BINDINGS

By default, both `@Bind` and listener bindings are required. An exception will be thrown if the target view cannot be found.

To suppress this behavior and create an optional binding, add a `@Nullable` annotation to the field or method.

Note: Any annotation named `@Nullable` can be used for this purpose. It is encouraged to use the `@Nullable` annotation from Android's "support-annotations" library, see Android Tools Project (http://tools.android.com/tech-docs/support-annotations).

```
@Nullable @Bind(R.id.might_not_be_there) TextView mightNotBeThere;

@Nullable @OnClick(R.id.maybe_missing) void onMaybeMissingClicked() {
  // TODO ...
}
```

MULTI-METHOD LISTENERS

Method annotations whose corresponding listener has multiple callbacks can be used to bind to any one of them. Each annotation has a default callback that it binds to. Specify an alternate using the `callback` parameter.

```
@OnItemSelected(R.id.list_view)
void onItemSelected(int position) {
  // TODO ...
}

@OnItemSelected(value = R.id.maybe_missing, callback = NOTHING_SELECTED)
void onNothingSelected() {
  // TODO ...
}
```

BONUS

Also included are `findById` methods which simplify code that still has to find views on a `View`, `Activity`, or `Dialog`. It uses generics to infer the return type and automatically performs the cast.

```
View view = LayoutInflater.from(context).inflate(R.layout.thing, null);
TextView firstName = ButterKnife.findById(view, R.id.first_name);
TextView lastName = ButterKnife.findById(view, R.id.last_name);
ImageView photo = ButterKnife.findById(view, R.id.photo);
```

Add a static import for `ButterKnife.findById` and enjoy even more fun.

Download

Butter Knife JAR (http://repository.sonatype.org/service/local/artifact/maven/redirect?r=central-proxy&g=com.jakewharton&a=butterknife&v=LATEST)

The source code to the library and sample application as well as this website is available on GitHub (http://github.com/JakeWharton/butterknife). The Javadoc is also available to browse (javadoc/index.html).

MAVEN

If you are using Maven for compilation you can declare the library as a dependency.

```
<dependency>
  <groupId>com.jakewharton</groupId>
  <artifactId>butterknife</artifactId>
  <version>(insert latest version)</version>
</dependency>
```

GRADLE

```
compile 'com.jakewharton:butterknife:(insert latest version)'
```

Be sure to suppress this lint warning in your `build.gradle`.

```
lintOptions {
  disable 'InvalidPackage'
}
```

Some configurations may also require additional exclusions.

```
packagingOptions {
  exclude 'META-INF/services/javax.annotation.processing.Processor'
}
```

IDE CONFIGURATION

Some IDEs require additional configuration in order to enable annotation processing.

- *IntelliJ IDEA* — If your project uses an external configuration (like a Maven `pom.xml`) then annotation processing should just work. If not, try manual configuration (ide-idea.html).
- *Eclipse* — Set up manual configuration (ide-eclipse.html).

PROGUARD

Butter Knife generates and uses classes dynamically which means that static analysis tools like ProGuard may think they are unused. In order to prevent them from being removed, explicitly mark them to be kept. To prevent ProGuard renaming classes that use @Bind on a member field the `keepclasseswithmembernames` option is used.

```
-keep class butterknife.** { *; }
-dontwarn butterknife.internal.**
-keep class **$$ViewBinder { *; }

-keepclasseswithmembernames class * {
    @butterknife.* <fields>;
}

-keepclasseswithmembernames class * {
    @butterknife.* <methods>;
}
```

License

(https://github.com/JakeWharton/butterknife)