1. Butterknife

Butterknife is a light weight library to inject views into Android components. It uses annotation processing.

The @BindView annotation allow to inject views and performs the cast to the correct type for you. The @@OnClick(R.id.yourid) annotation allows to add OnClickListener to a view. You can optional define the method parameter of the view in case you want it injected.

Butterknife includes also 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.

You can also bind to fragments. Butterknife also allows to unbind again, via the Unbinder object.

```
public class YourFragment extends Fragment {
  @BindView(R.id.button1) Button button1;
  @BindView(R.id.button2) Button button2;
  private Unbinder unbinder;

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

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

Annotated attributes and methods cannot be private, as ButterKnife needs to be able to access them from a separate class.

2. How does Butterknife work

Butterknife uses annotation processing to generated modified Java classes based on your annotations. Annotation processing is a tool build in javac for scanning and processing annotations at compile time.

You can define custom annotations and a custom processor to handle them. These annotations are scanned and processed at compile time. The annotation processor does not change the exiting input class but it generates a new Java class. This generated Java code is compiled again as a regular Java class.

The Butterknife annotation processor scans all Java classes looking for the Butterknife annotations. If a class contains these annotations, it generates a new class based on the *<original_class>__ViewBinding* schema.

3. Exercise: Using Butterknife in your Android project

3.1. Create project

Create a new Android project with the package com.vogella.android.usinglibs. Add a text view with the @+id/textView to it and a button to the existing layout with the @+id/button ID.

3.2. Add Butterknife to your Gradle build configuration

Add the com.jakewharton:butterknife in its latest version as compile dependency build.gradle file.

```
apply plugin: 'com.android.application'

android {
    ...
}

dependencies {
    ...
    implementation 'com.jakewharton:butterknife:8.5.1'
    annotationProcessor 'com.jakewharton:butterknife-compiler:8.5.1'
}
```

3.3. Use view injection in your Android activity

Use @BindView, @OnClick and ButterKnife.bind to get the views injected.

The following shows a possible solution.

```
package com.vogella.android.usinglibs;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
import android.widget.Toast;

import butterknife.BindView;
import butterknife.ButterKnife;
import butterknife.OnClick;
```

```
public class MainActivity extends Activity {
    @BindView(R.id.textView)
    TextView title;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ButterKnife.bind(this);
        title.setText("Hello from Butterknife");
    }

    @OnClick(R.id.button)
    public void submit() {
        Toast.makeText(MainActivity.this,
                "Hello from Butterknife OnClick annotation", Toast.LENGTH_SHORT).show();
    }

}
```

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 value) field
  // ...
}
```

NON-ACTIVITY BINDING

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

```
public class FancyFragment extends Fragment {
  @BindView(R.id.button1) Button button1;
  @BindView(R.id.button2) Button button2;
  @Override public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {
    View view = inflater.inflate(R.layout.fancy_fragment, container, false);
    ButterKnife.bind(this, view);
    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 {
    @BindView(R.id.title) TextView name;
    @BindView(R.id.job_title) TextView jobTitle;

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

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 ...
}
```