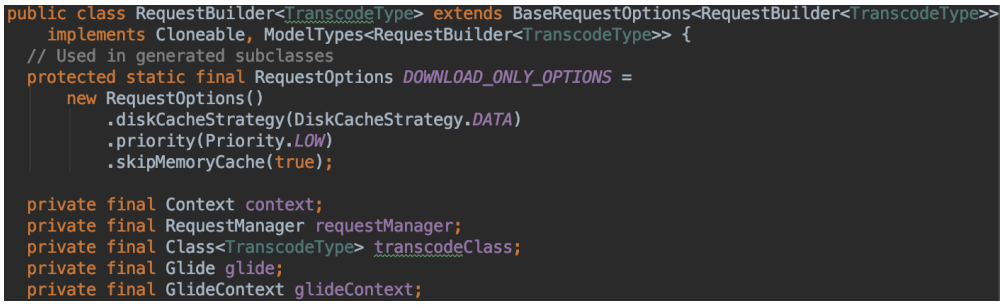
Feature 2: Media File Load/Display

As Glide is an open source media management and image loading framework, image display is definitely an essential feature. In assignment 1, we examined how this feature operates within a sample module that displays and processes svg images. However, since it is more meaningful to look at how this feature works across all types of media resources, we will reexamine this feature at a higher level.

Glide aims to make media fetching, resizing, and displaying a smooth experience for the user. Therefore, for each resource, the system builds a request that will handle various components of the overall image load/display process.

There are two components that are integral to our feature:

1) *RequestBuilder.java*, which handles various components of a resource such as resource loading, display, monitoring, etc.
   The class is located in the path: */glide/library/src/main/java/com/bumptech/glide/RequestBuilder.java*



```
public class RequestBuilder<TranscodeType> extends BaseRequestOptions<RequestBuilder<TranscodeType>>
    implements Cloneable, ModelTypes<RequestBuilder<TranscodeType>> {
  // Used in generated subclasses
  protected static final RequestOptions DOWNLOAD_ONLY_OPTIONS =
      new RequestOptions()
          .diskCacheStrategy(DiskCacheStrategy.DATA)
          .priority(Priority.LOW)
          .skipMemoryCache(true);

  private final Context context;
  private final RequestManager requestManager;
  private final Class<TranscodeType> transcodeClass;
  private final Glide glide;
  private final GlideContext glideContext;
```

2) *GlideApp.java*, a generated file, is the bridge between the system and other application.
   The class is located in the path: */glide/instrumentation/build/generated/source/apt/debug/com/bumptech/glide/test/GlideApp.java*

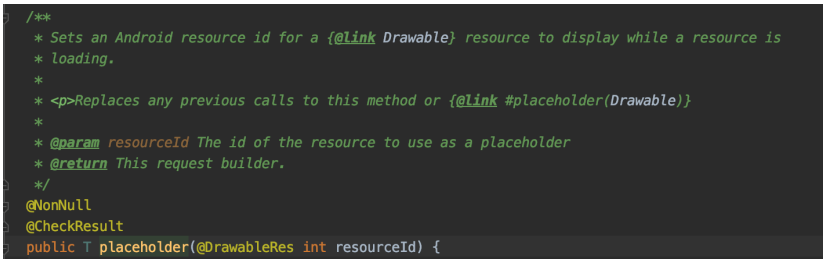
```
/**
 * The entry point for interacting with Glide for Applications
 *
 * <p>Includes all generated APIs from all
 * {@link com.bumptech.glide.annotation.GlideExtension}s in source and dependent libraries.
 *
 * <p>This class is generated and should not be modified
 * @see Glide
 */
public final class GlideApp {
```

GlideApp.with().as() can convert a context or an activity to a *GlideRequest*.

To load a media file, we need to call a request of type GlideRequest<TranscodeType>, which extends RequestBuilder<TranscodeType> class and then call its load() method.

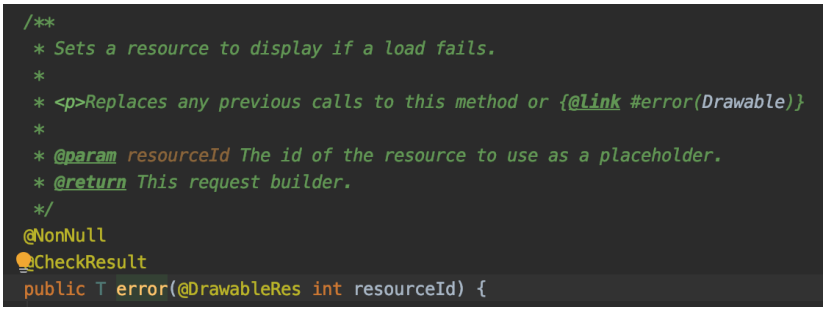
Within the GlideRequest class, it defines some basic methods, such as placeholder(), error(), transition(), listener(), addlistener() etc. altering the way a media file is loaded, displayed, and monitored.

1) When we call the GildeRequest.placeholder() method, it will return a new GlideRequest with an attribute holding the place for the loading media.



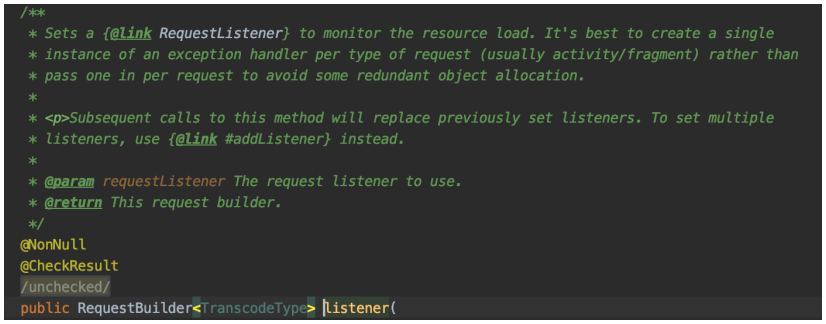
```
/**
 * Sets an Android resource id for a {@link Drawable} resource to display while a resource is
 * loading.
 *
 * <p>Replaces any previous calls to this method or {@link #placeholder(Drawable)}
 *
 * @param resourceId The id of the resource to use as a placeholder
 * @return This request builder.
 */
@NonNull
@CheckResult
public T placeholder(@DrawableRes int resourceId) {
```

2) When we call the GlideRequest.error() method, it will return a new GlideRequest with an attribute that will display a resource if the load fails.



```
/**
 * Sets a resource to display if a load fails.
 *
 * <p>Replaces any previous calls to this method or {@link #error(Drawable)}
 *
 * @param resourceId The id of the resource to use as a placeholder.
 * @return This request builder.
 */
@NonNull
@CheckResult
public T error(@DrawableRes int resourceId) {
```

3) When we call the GlideRequest.listener() method, it will return a new GlideRequest with an attribute that will monitor the resource load.

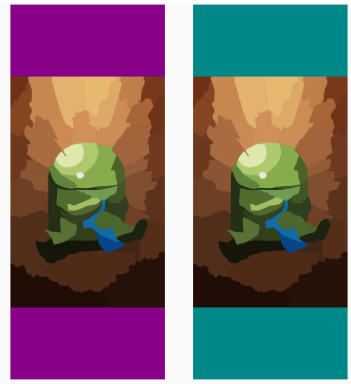


```
/**
 * Sets a {@link RequestListener} to monitor the resource load. It's best to create a single
 * instance of an exception handler per type of request (usually activity/fragment) rather than
 * pass one in per request to avoid some redundant object allocation.
 *
 * <p>Subsequent calls to this method will replace previously set listeners. To set multiple
 * listeners, use {@link #addListener} instead.
 *
 * @param requestListener The request listener to use.
 * @return This request builder.
 */
@NonNull
@CheckResult
/unchecked/
public RequestBuilder<TranscodeType> listener(
```

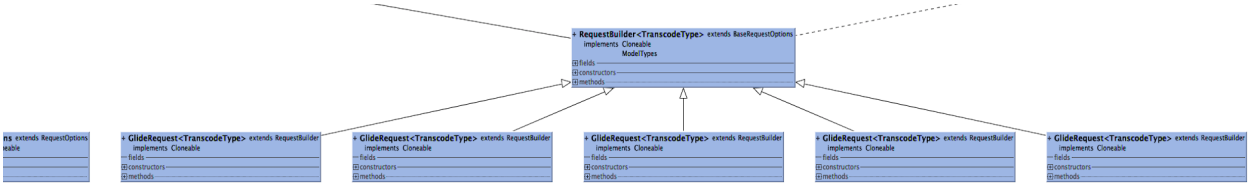
With this feature, Glide is able to use a simple fluent API that allows users to make most requests in a single line.

```
requestBuilder =
    GlideApp.with( activity: this) GlideRequests
        .as(PictureDrawable.class) GlideRequest<PictureDrawable>
        .placeholder(R.drawable.image_loading)
        .error(R.drawable.image_error)
        .transition(withCrossFade())
        .listener(new SvgSoftwareLayerSetter());
```

After adding all the attributes above, we call load() method to load the media, with the attributes we mentioned.



There are different classes of different types extending RequestBuilder class, which enables the system to start the request of loading/displaying different types of media.



Other relevant parts of the system and how they're relevant:

Without proper image file processing, we cannot expect to load an image correctly using this system. From our examination of the encoder, decoder, and transcoder components of this system (reported in feature 1 of assignment 2), we have determined that they are essential to achieving this goal. These interfaces allow for modularity of these file processing methods. This allows the user to make customized implementations with varying image file types.