Developer's Guide

Table of contents

1. Introduction

This guide is an introduction to the im4java-library. You should be familiar with java-programming and should know how to read the API documentation. Also, this is no guide for the usage of the underlying tools (ImageMagick, GraphicsMagick and so on). You should be familiar with them and know how to read the respective documentation.

The basic architecture of im4java is quite simple. It boils down to calling all underlying tools simply by using a `ProcessBuilder`-object. All the magic of im4java ist to hide the complexities. If you have just one simple call of an external tool in your program, you might be better of by hardcoding the `ProcessBuilder`-call yourself. If you don't need the simplicity or the advanced features of the im4java-library, your code will certainly be faster and more efficient.

2. Before you begin: Setting up the Environment

To use the im4java-library, you should add the im4java-jarfile to you classpath. This is the first part of the setup. The second part is optional and only necessary, if the tools you want to use (e.g. `convert` or `exiftool`) are not on your *PATH*. This is typically a problem on Windows-systems.

To setup your searchpath for the tools you have three options:

- Set the environment-variable *IM4JAVA_TOOLPATH*. This variable should contain a list of directories to search for your tools separated by your platform-pathdelemiter (on *NIX typically ":", on Windows ";").
- Globally set the searchpath from within your java-progam:

  ```
  String myPath="C:\\Programs\\ImageMagick;C:\\Programs\\exiftool";
  ProcessStarter.setGlobalSearchPath(myPath);
  ```

  This will override any values set with *IM4JAVA_TOOLPATH*.
- Set the search path for an individual command:

  ```
  String imPath="C:\\Programs\\ImageMagick";
  ConvertCmd cmd = new ConvertCmd();
  cmd.setSearchPath(imPath);
  ```

  This will override any values set with *IM4JAVA_TOOLPATH* or with `ProcessStarter.setGlobalSearchPath()`.

Warning:
Note that I also encountered a problem using OpenJDK with a language-setting of *LANG=de_DE.UTF-8*. With *LANG=C* everything worked fine. With SUN's JDK, there were no problems regardless of the language-setting.

3. Simple Use

Basically, to use im4java, you need objects of two classes: an `ImageCommand` like `ConvertCmd`, and an `Operation` like `IMOperation`. The `ImageCommand` is more or less static, you would create an instance once and reuse it for the lifetime of your program. Exceptions to this rule are more advanced use cases, see the section below about parallel processing. In contrast, the *Operation* is the object wrapping all the commandline options you intend to pass to the given command. So you would typically create one `Operation` for every action (resizing, conversion) you intend to do.

As an example, consider resizing an image:

```
// create command
ConvertCmd cmd = new ConvertCmd();

// create the operation, add images and operators/options
IMOperation op = new IMOperation();
op.addImage("myimage.jpg");
op.resize(800,600);
op.addImage("myimage_small.jpg");

// execute the operation
cmd.run(op);
```

4. About ImageCommand

All command-classes subclass `ImageCommand`, which itself subclasses `org.im4java.process.ProcessStarter`. The latter class wraps `java.lang.ProcessBuilder`, handles input and output streams and supports asynchronous execution.

The `ImageCommand` class adds methods useful for all command-classes, things like support for reusing operations or for dynamic operations.

Note that `ImageCommand` is not stateless. In the default setting, it captures everything written to stderr. It also holds an internal *process ID* (unrelated to any operating system PID) via `ProcessStarter`. Nevertheless, if you only use the `ImageCommand` in synchronous mode, you can reuse the instance.

5. Using GraphicsMagick

GraphicsMagick is a fork of ImageMagick. GraphicsMagick has a number of advantages compared to ImageMagick, the most prominent is it's superior performance. Since the fork ImageMagick has improved the expressive power of it's command-line syntax, therefore, an ImageMagick commandline is not necessarely compatible with GraphicsMagick. But for most single-operation conversions it still is.

With im4java, you have three options if you want to use GraphicsMagick:

- use GraphicsMagick explicitely, passing the command at object-creation:

`GraphicsMagickCmd cmd = new GraphicsMagickCmd("convert");`.

- use GraphicsMagick explicitely, using wrapper classes: `ConvertCmd cmd = new ConvertCmd(true);`.
- decide at runtime: setting the system-property *im4java.useGM* to true will select GraphicsMagick at runtime. You can use this feature to compare the results and timings of both toolsets, provided that the commandline is compatible.

6. Reusing Operations

In the example above, image-names were hard-coded. The im4java-library supports an alternative use. Instead of hard-coding the image-names, you just add placeholders and resolve the image-names at execution time. This allows the reuse of operations for example within a loop.

The following example extends the example of the first section and loops over all images passed as method parameters:

```
public void resizeImages(String... pImageNames) {
  // create command
  ConvertCmd cmd = new ConvertCmd();

  // create the operation, add images and operators/options
  IMOperation op = new IMOperation();
  op.addImage();
  op.resize(800,600);
  op.addImage();

  for (String srcImage:pImageNames) {
    int lastDot = srcImage.lastIndexOf('.');
    String dstImage =
       srcImage.substring(1,lastDot-1)+"_small.jpg";
    cmd.run(op,srcImage,dstImage);
  }
}
```

You can pass an arbitrary number of image-names to `cmd.run()`, you can even pass an array of image-names. In the latter case you have to cast the array to `Object[]`, e.g. `cmd.run(op,(Object[]) imgNames)`.

Note that `op.addImage()` is actually a short form for `op.addImage(Operation.IMG_PLACEHOLDER)`. You can also add more than one placeholder at the same time with `op.addImage(int count)`.

The `op.addImage(String... images)`-method also supports ImageMagick's *read-modifiers*. Adding a read-modifier for hard-coded images is of course straightforward (you just add it to the argument string). For placeholders, you add only the read-modifier. The following two lines of code therefore have the same effect:

```
op.addImage("[300x200]");
op.addImage(Operation.IMG_PLACEHOLDER+"[300x200]");
```

The test-case class `org.im4java.test.TestCase7` uses read-modifiers to crop the source-images prior to composing them:

```
IMOperation op = new IMOperation();
op.blend(50);
op.addImage("[300x200+0+0]");  // read and crop first image
op.addImage("[300x200+0+0]");  // read and crop second image
op.addImage();                 // output image

CompositeCmd composite = new CompositeCmd();
composite.run(op,"rose1.jpg","rose2.jpg",outfile);
```

7. Adding Operations to Operations

Im4java supports a second variant of operation-reuse. You can define one `Operation` and just add it to another one. The following snippet defines a *rotate-resize-frame*-operation and adds it to another operation:

```
IMOperation frame = new IMOperation();
frame.rotate("90");
frame.resize(640);
frame.border(10,10);

IMOperation row = new IMOperation();
row.addImages(3);
row.add(frame);
row.p_append();
```

Adding operations as just described is valid for all supported im4java-tools. ImageMagick additionally supports options and operations within parenthesis thus limiting the effect of settings and operators on everything within the parenthesis. You add parenthesis with the methods `op.openOperation()` and `op.closeOperation()`:

```
IMOperation frame = new IMOperation();
frame.openOperation();
frame.rotate("90");
frame.resize(640);
frame.border(10,10);
frame.closeOperation();
```

An alternatative way of coding this is:

```
IMOperation frame = new IMOperation();
frame.rotate("90");
frame.resize(640);
frame.border(10,10);

IMOperation row = new IMOperation();
row.addImages(3);
row.addSubOperation(frame);
row.p_append();
```

The `op.addSubOperation()`-method just adds the surrounding parenthesis.

8. Dynamic Operations

Dynamic Operations are an advanced technique. Sometimes you only want to apply some operations to images fulfilling some requirements. ImageMagick itself has some special option-flags for this purpose, e.g. an image is only scaled (down) if it has a larger size than the target-size. For special cases not directly supported by ImageMagick, you can make use of im4java's *Dynamic Operations*. Basically, you implement the interface `org.im4java.core.DynamicOperation`, which has exactly one method *resolveOperation()*. At execution time, this method gets all argument images passed as parameters, and it must return an `Operation`. The returned object could also be *null*, in this case no `Operation` is added.

The test-case class `org.im4java.test.TestCase11` shows an example of dynamic operations. In this case, the `despeckle()` method is only added for images with a high iso-noise level.

9. Capturing Output

The default behaviour of all `ImageCommand`s is to pass all output of the wrapped commands to stdout, and to capture everything from stderr in an `CommandException`-object. You can change this behaviour with the methods `ImageCommand.setOutputConsumer(OutputConsumer oc)` and `ImageCommand.setErrorConsumer(ErrorConsumer ec)`. Both `OutputConsumer` and `ErrorConsumer` are interfaces in the `org.im4java.process`-package with single methods (`consumeOutput()` and `consumeError()`). These methods just read everything from the argument `InputStream`.

In the process-package there is an utility-class called `ArrayListOutputConsumer` which collects all lines of output in a String-array.

10. Piping

Most commandline tools allow piping of input or output. With the im4java-library you can create instances of `org.im4java.process.Pipe` to mimic this behaviour. This class implements the `OutputConsumer` and `ErrorConsumer`-interfaces mentioned above and are useful for piping the output of a commandline tool to an `OutputStream` (e.g. a network-socket). To use the pipe, instantiate it with an `OutputStream` and use the method `ImageCommand.setOutputConsumer(pipe)`.

If you want to provide input to stdin of a commandline tool, you have to create a pipe-object initialized with an `InputStream` and use the method `ImageCommand.setInputProvider(pipe)`. The pipe will read from the `InputStream` and write to the stdin of the respective `ImageCommand`.

The test-case `org.im4java.test.TestCase10` features pipes, reading from one image and writing to another. In real-life, you would of course process the files directly, but the example just wants to demonstrate what to do:

```
 IMOperation op = new IMOperation();
 op.addImage("-");                  // read from stdin
 op.addImage("tif:-");              // write to stdout in tif-format

// set up pipe(s): you can use one or two pipe objects
FileInputStream fis = new FileInputStream(iImageDir+"ipomoea.jpg");
FileOutputStream fos = new FileOutputStream(iImageDir+"ipomoea.tif");
// Pipe pipe = new Pipe(fis,fos);
Pipe pipeIn  = new Pipe(fis,null);
Pipe pipeOut = new Pipe(null,fos);

// set up command
ConvertCmd convert = new ConvertCmd();
convert.setInputProvider(pipeIn);
convert.setOutputConsumer(pipeOut);
convert.run(op);
fis.close();
fos.close();
```

11. Using BufferedImages

A `BufferedImage` is in a way the *java native* representation of an image-object. No commandline tool can deal directly with a `BufferedImage`. The good news is that im4java uses objects of type `BufferedImage` transparently, if you use pass these objects at invocation time:

```
IMOperation op = new IMOperation();
op.addImage();                      // input
op.blur(2.0).paint(10.0);
op.addImage();                      // output

ConvertCmd convert = new ConvertCmd();
BufferedImage img = ...;
String outfile = ...;
...
convert.run(op,img,outfile);
```

Note that the above use of `BufferedImage`s works fine for input-images. If you need to write to a `BufferedImage`, you must pipe the output of the commandline-tool to stdout, create an instance of the class `org.im4java.core.Stream2BufferedImage` and set it as the `OutputConsumer` of the command:

```
IMOperation op = new IMOperation();
op.addImage();                          // input
....
op.addImage("png:-");                   // output: stdout
...
images = ...;

// set up command
ConvertCmd convert = new ConvertCmd();
Stream2BufferedImage s2b = new Stream2BufferedImage();
convert.setOutputConsumer(s2b);

// run command and extract BufferedImage from OutputConsumer
convert.run(op,(Object[]) images);
BufferedImage img = s2b.getImage();
```

12. Asynchronous Execution

Long running operations belong into a seperate thread, especially in graphical applications. The im4java-library supports asynchronous execution with and without callbacks.

The latter case is simple (fire-and-forget). Befor you start the command, you just set the aynchronous-mode to true:

```
ConvertCmd cmd = new ConvertCmd();
cmd.setAsyncMode(true);
...
cmd.run(op);
```

In this case, you will know nothing about success or failure. If you need feedback (e.g. because you want to asynchronously convert a file and load the result into a window), you must write a class implementing the interface `org.im4java.process.ProcessEventListener`. This interface defines three methods: `processInitiated()`, `processStarted()` and `processTerminated()`. The first method is called synchronously from the original thread calling the run-method, the latter two methods are callbacks from the asynchronous thread. See `org.im4java.test.TestCase16` for a complete example.

With `cmd.setAsyncMode(true)` you only need minimal code-changes for asynchronous execution. If you prefer to control the flow of execution yourself, you could use some standard methods from `java.util.concurrent` to control execution:

```
ProcessTask pt = cmd.getProcessTask(op);
ExecutorService exec = Executors.newSingleThreadExecutor();
exec.execute(pt);
exec.shutdown();
```

The test-case 16a will give you a complete example. The third variant, test-case 16b replaces the standard executor returned by `Executors.newSingleThreadExecutor()` with an instance of class `org.im4java.process.ProcessExecutor`. For a discussion of this class, proceed to the next section.

13. Parallel Processing

The use case described above is fine for typical graphical applications with one asynchronous thread. In contrast, if you want to convert a number of files asynchronously, additional problems arise. Consider the following piece of code:

```
// load images into an array, e.g. from a directoy
ArrayList<String> images = load(myDir);

// convert all images
ConvertCmd cmd = new ConvertCmd();
cmd.setAsyncMode(true);
Operation op = ...;
for (String img:images) {
  String outfile = ...;
  cmd.run(op,img,outfile);
}
```

Although this will run perfectly fine, this code will flood your system with parallel convert-processes, making your system unusable for a while. So one of the issues is *ressource management*. Another issue is that you don't know when you are finished. In addition, you don't know which of your conversions succeeded and which failed.

The following sections deal with these three issues. This is advanced stuff, and you might not even need it. If you have to convert multiple images, you could first try to use the class `org.im4java.utils.BatchConverter`, which uses the building blocks described below. The class `BatchConverter` is covered here.

13.1. The ProcessExecutor

The classes in `java.util.concurrent` address these issues. All classes returned by the factory class `java.util.concurrent.Executors` operate on threads. They provide methods to queue and start requests up to a given limit, and also allow you to stop the queue and destroy running threads.

There is one big drawback with these thread-based executors. Once an `ImageCommand` is running within a java-thread, the thread will not be killable due to the active process. Therefore you should not use any of the standard executors, but use an instance of the class `org.im4java.process.ProcessExecutor`. A basic usage is very simple, the example above then looks like this:

```
// load images into an array, e.g. from a directoy
```

```
ArrayList<String> images = load(myDir);

// convert all images
ProcessExecutor exec = new ProcessExecutor();
Operation op = ...;
for (String img:images) {
  String outfile = ...;
  ConvertCmd cmd = new ConvertCmd();
  ProcessTask pt = cmd.getProcessTask(op,img,outfile);
  exec.execute(pt);
}
exec.shutdown();
```

The default constructor of `ProcessExecutor` will query the number of processors on the system and limit the number of parallel running processes to that number. You can also pass an integer to the constructor if you want to set the limit yourself.

The class `ProcessTask` extends `java.util.concurrent.FutureTask`. You can use all the standard methods of this class, e.g. to query results or to wait for termination.

13.2. Waiting for process termination

It is usually important to know when your processes have finished, maybe to give feedback to a user by updating a progress bar or to start some follow-up activity. If the processes take too long, you might also consider killing them.

Since `ProcessExecutor` extends `java.util.concurrent.ThreadPoolExecutor`, you can use the standard methods provided by this class. If you want to block until your processes terminate, you would use the following code snippet (this one extends the example above):

```
ProcessExecutor exec = new ProcessExecutor();
for (String img:images) {
...
}
exec.shutdown();
if (exec.awaitTermination(10,TimeUnit.SECONDS)) {
  System.err.println("processes terminated on their own");
} else {
  System.err.println("trying to cancel all running processes ...");
  exec.shutdownNow();
}
```

As an alternative to the blocking `awaitTermination()`-call you could also subclass `ProcessExecutor` and implement it's `terminated()`-method. Then you will receive a callback once all processes have terminated.

One final warning: the code implementing the parallel processing of commands is new and therefore untested in the wild. During development, a number of race-conditions came up (and were solved), but feedback on stability, functionality and implementation is highly welcome.

13.3. Exit status of finished asynchronous processes

The last issue with asynchronous processes is the exit status. For a single asynchronous process this is quite simple, you would implement a `ProcessEventListener` and use it's `processTerminated()`-method (see the section Asynchronous Execution above).

For multiple parallel process the situation is a bit more complicated. You have to link the processTerminated-event with the correct process. The class `ProcessEvent` implements a number of methods which help to identify the process. One is `ProcessEvent.getPID()`. The *PID* is an internal field of each `ImageCommand`. You can set this field explicitly overriding the PID set during object-creation. You can also query the `ImageCommand` object itself with `ProcessEvent.getProcessStarter()` (remenber that `ProcessStarter` is the base-class of `ImageCommand`).

For a complete example using these methods, see the class `org.im4java.test.TestCase21`.

14. Utilities

This section describes a number of utility-classes which facilitate the coding.

14.1. Image Information

If you only want to query image-information (e.g. width and height), you could typically use the class `IdentifyCmd`, wrapping ImageMagick's *identify*-command. Instead of using this class directly, you could instead use the `Info` class. The following code-snippet demonstrates its use:

```
Info imageInfo = new Info(filename,true);
System.out.println("Format: " + imageInfo.getImageFormat());
System.out.println("Width: " + imageInfo.getImageWidth());
System.out.println("Height: " + imageInfo.getImageHeight());
System.out.println("Geometry: " + imageInfo.getImageGeometry());
System.out.println("Depth: " + imageInfo.getImageDepth());
System.out.println("Class: " + imageInfo.getImageClass());
```

The second parameter (*true*) in the example requests *basic*-information. This is a bit faster than requesting and parsing the complete (verbose) output of the class `IndentifyCmd`. See the test-case class `org.im4java.test.TestCase8` for a complete example.

Prior to version 1.3.0 the implementation of the `Info`-class was severely flawed. It did not take into account that there are image-formats like TIF or GIF that support multiple images (ImageMagick calls them *scenes*) within a single file. As a consequence, the method

```
imageInfo.getImageWidth()
```

returns the image-width of the *first* scene (from basic-information), whereas the method

```
imageInfo.getProperty("Width")
```

will return the image-width of the *last* scene (from complete information). Starting with version 1.3.0, there are new methods with an additional parameter, the scene-number, e.g.

```
imageInfo.getImageWidth(3)
imageInfo.getProperty("Width",3)
```

To query the number of scenes use the method `getSceneCount()`. Note that information about multiple scenes is only available with complete-information.

Note that parsing the output of `identify -verbose` is inherently flawed, since this output is meant to be human-readable and not an an interface for computer programs. The parser makes a number of assumptions about the output, some of them are known to be incorrect in special situations (e.g. multi-line attribute-values with embedded colons). Also note that basic-information should always be correct, since it uses a different method to aquire the information. As an alternative to the `Info`-class you might consider using the wrapper class `ExiftoolCmd` for `exiftool`.

14.2. FilenameLoader

The class `org.im4java.utils.FilenameLoader` is useful for batch-processing a number of files from a directory. The core method is `public List<String> loadFilenames(String pDir)`. It loads all files from the given directory into a list of strings.

The constructor accepts an `ExtensionFilter`. You can instantiate your own filter as in the example below or use one of the predefined filters of the ExtensionFilter-class.

```
ExtensionFilter filter = new ExtensionFilter("jpg");
filter.setRecursion(true);
filter.ignoreDotDirs(true);
FilenameLoader  loader = new FilenameLoader(filter);
List<String> files = loader.loadFilenames(mydir);
```

As always, you should check the API-documentation for all the features of this class.

14.3. FilenamePatternResolver

When converting multiple files, the target filename usually depends on the source filename. For example a standard conversion from jpg to tif would keep the filename and just change the extension. Or all converted files should additionally go to a separate directory.

This is where the class `org.im4java.utils.FilenamePatternResolver` is useful. The following snippet will convert all argument-files to tif.

```
// define operation and command
IMOperation op = new IMOperation();
op.addImage();                         // input-file
op.addImage();                         // output-file
ConvertCmd cmd = new ConvertCmd();

// load files
ExtensionFilter filter = new ExtensionFilter("jpg");
FilenameLoader  loader = new FilenameLoader(filter);
List<String> files = loader.loadFilenames(mydir);

// create the resolver
FilenamePatternResolver resolver =
    new FilenamePatternResolver("%P/%f.tif");

// now iterate over all files
for (String img:files) {
  cmd.run(op,img,resolver.createName(img));
}
```

The FilenamePatternResolver recognizes the following escape-sequences within it's pattern:

- %P: full pathname of source-image (i.e. the directory)
- %p: last component of %P
- %F: full filename without directory part
- %f: filename without directory part and extension
- %e: only the extension
- %D: drive-letter (on windows systems). Not available for source-files with an UNC-name.

14.4. BatchConverter

The class `org.im4java.utils.BatchConverter` is a utility-class for client-applications. It will convert a given list of files in parallel making use of all available processors to speed up execution. It is not well suited for web-applications, since you don't want a single request to use up all of your ressources.

Usage of this utility-class is straightforward. First you load your files into a `List`. This could be from a GUI-application where a user selects multiple files. Or the list could contain all files from a given directory (see the section FilenameLoader above).

```
 ExtensionFilter filter = new ExtensionFilter("jpg");
 filter.setRecursion(false);
 FilenameLoader loader =  new FilenameLoader(filter);
 List<String> images=loader.loadFilenames(dir);
```

After you have the list, you create your `BatchConverter` and use it's `run()`-method to process the images:

```
// create a simple thumbnail operation
op = new IMOperation();
op.size(80);
op.addImage();     // placeholder input filename
op.thumbnail(80);
op.addImage();     // placeholder output filename

// create a template for the output-files:
// we put them in targetDir with the same filename as the original
// images
String template=targetDir+"%F";

// create instance of BatchConverter and convert images
BatchConverter bc = new BatchConverter(BatchConverter.Mode.PARALLEL);
bc.run(op,images,targetDir+"%F");
```

Since `BatchConverter` extends `ProcessExecutor`, you can use the methods described in the section about process termination to wait for the termination of the command (note that the `shutdown()`-method is called automatically).

The class `BatchConverter` knows three modes of operation: `BatchConverter.SEQUENTIAL`, `BatchConverter.PARALLEL` and `BatchConverter.BATCH`. The first mode is more or less for benchmarking the other two, it converts the images one after another sequentially. The second mode uses parallel processing, it runs in it's default setting on all available processors. The last mode uses convert's ability to process more than one image at the same time:

```
convert image1.jpg image2.jpg target_%d.tif
mv target_1.tif image1.tif
mv target_2.tif image2.tif
```

On modern computers with more than one processor `BatchConverter.PARALLEL` should be the fastest. If only one (real) processor is available, `BatchConverter.BATCH` should make the game.

For a complete example see `TestCase22`. This test-case subclasses `BatchConverter` and uses the `terminated()`-method to receive a callback after termination. After termination, the callback-methods uses the `getFailedConversions()`-method of `BatchConverter` to query a list of `BatchConverter.ConvertException`-objects. These objects wrap the cause and the index of the image responsible of the failure.

15. Debugging

Since version 1.0 im4java has a new method `ImageCommand.createScript()` to aid in debugging:

```
IMOperation op = new IMOperation();
...
ConvertCmd cmd = new ConvertCmd();
cmd.createScript("myscript.sh",op);
```

This will dump your command and operation to a script-file. You should change the execution-permission of the file and try the script to make sure that you in fact generate the commandline you intend to use.

Note that on windows-platforms, `createScript()`-method will automatically add the extension `.cmd` to the filename passed to the method.