




 **rkalla / imgscalr** Public

Simple Java image-scaling library implementing Chris Campbell's incremental scaling algorithm as well as Java2D's "best-practices" image-scaling techniques.

 www.thebuzzmedia.com/software/imgscalr-java-image-scaling-library/

 Apache-2.0 License

 1.1k stars  240 forks

 Star

 Watch ▾

Code

Issues **39**

Pull requests **3**


Actions

Projects

Security

Insights

45.25 MB

 master ▾

...



rkalla ...

on 1 Mar 2018



[View code](#)

README

imgscalr - Java Image-Scaling Library

<http://www.thebuzzmedia.com/software/imgscalr-java-image-scaling-library/>

Changelog

4.2

- * Added support for a new Method.ULTRA_QUALITY scaling method.

This new method uses 3.5x more incremental steps when scaling an image down than the QUALITY method, providing a much more accurate result. This is especially noticeable in thumbnails that have diagonal lines that get jagged during down-sizing with QUALITY or lower methods.

The quality of the ULTRA_QUALITY scaling method is almost on par with the image resize functionality built into Mac OS X; that is to say it is better than GIMP's Lancsoz3 and Windows 7 built-in resize.

#61

- * Fixed subtle bug with incremental scaling and Mode.FIT_EXACT causing the incremental scaling to stop too soon resulting in the wrong-sized result image.

The stop-condition for incremental scaling assumed that in every case the width AND height would be shrinking each iteration; when using Mode.FIT_EXACT this is not necessarily true as one dimension may not change

at all or stop changing before another.

#65

4.1

* Fixed NullPointerException that occurred when debugging was enabled

#60

Required a patch-release due to the show-stopping nature of the bug.

4.0

* [BREAKING] Package has changed from com.thebuzzmedia.imgscalr to org.imgscalr - I am sorry for the inconvenience of this, but this is necessary. There will be a family of imgscalr-based Java utilities coming out in the future (ExifTool is next) that will all be under this umbrella.

* [BREAKING] Java 6 is now required for using imgscalr.

The reason for this is because imgscalr includes specific types of ResizeOp and ColorConvertOps that actually segfault the latest Java 5 VM when applied, but run fine in Java 6 and 7.

imgscalr cannot knowingly ship VM-segfaulting code that could would introduce a potentially devastating situation into client applications.

This decision was not made lightly, but with Java 5 end-of-lifed and Java 6 being out for 5 years, it seemed like a reasonable requirement.

* [BREAKING] Rotation enum was totally redefined. All rotations were redefined in terms of 90,180,270 quadrant rotations as well as h/v FLIP.

* [BREAKING] All resize(...) methods that accepted Rotation enums are removed. All graphic operations are now separate and discrete, but can be easily combined when multiple effects are wanted.

* Added apply() support for applying an arbitrary list of BufferedImageOps SAFELY and efficiently working around all the bugs in the JDK pertaining to BufferedImageOps (also used internally when applying any optionally specified ops).

* Added crop() support.

* Added pad() support.

* Added rotate() support.

* All graphic operations (even new ones) were modified to allow the application of 1 or more BufferedImageOps to a final image result before returning it for convenience.

* Support for all the new operations (apply, crop, pad, rotate) were all added to AsyncScalr so these operations can all be asynchronously performed as well.

* Added support for horizontal and vertical flipping of the image via the Rotation enum and new rotate() method.

* Added pre-defined OP_DARKER and OP_BRIGHTER operations that can be applied

to any image to make them darker or brighter (respectively) by 10%.

* Added Mode.FIT_EXACT to support (for the first time) scaling images forced into a specific given dimension instead of honoring the image's orientation and proportions automatically.

* AsyncScalr's use of ExecutorService was rewritten; no more support for passing in custom ExecutorService implementations or modifying existing ones on the fly and having the class do something magic to them under the covers (that was bad) -- just extend the class and specify your own logic.

* AsyncScalr can be easily customized now through a single method:

- createService()
- OR
- createService(ThreadFactory)

* AsyncScalr provides two custom ThreadFactory implementations for subclasses to use if they want to customize the types of Threads generated and used internally for async scale operations.

- DefaultThreadFactory creates default threads with all default settings.
- ServerThreadFactory generates threads that are optimized to execute in a server environment (daemon threads w/ LOW_PRIORITY).

* AsyncScalr.DEFAULT_THREAD_COUNT was removed and replaced with THREAD_COUNT that can be customized and set via system properties.

* AsyncScalr.THREAD_COUNT's property name was separated into a String constant to make it easier to work with.

* Simplified the resize() calls as a result of making all operations discrete; 8 duplicate methods accepting "rotation" arguments were removed.

* Optimized the application of BufferedImageOps.

* Fixed a bug in the application of BufferedImageOps which could have led to an ImagingOpException bubbling up from native Java2D or a corrupt (black) image for poorly supported image types.

* Memory optimized the application of 2 or more BufferedImageOps (interim images are explicitly cleaned up just like in incremental scaling).

* Optimized log() implementation to avoid StringBuilder creation and string concatenation. Should be significant run-time savings over time if you are running in an environment with debugging turned on.

* Removed the identity-return functionality in each method to throw an exception instead of silently returning "src" unchanged.

This was done intentionally to avoid users getting caught in the situation where they have code that automatically calls flush() on "src" after an imgscalr method has returned (assuming they NOW have a modified copy to work with).

In the case of sending in invalid or null arguments, previously imgscalr would return "src" unchanged, which means the caller would be calling flush() on a perfectly good image they still needed and not a copy as was

assumed by using `imgscalr` (And there would be no way to tell if `imgscalr` had created a copy or not without using an `==` check with EVERY returned image result).

Instead, invalid or missing arguments passed to any `imgscalr` method are now considered an exception so the caller knows IMMEDIATELY when something is wrong and won't get magically different/unexpected behavior.

- * Exposed the potential for every method to fire an `ImagingOpException` if one of the `BufferedImageOps` fails to apply using the hardware-accelerated underlying Java2D code path. These exceptions were previously hidden in the guts of Java2D and could bubble up unexpectedly, now they are clearly defined directly on the `imgscalr` API so they can be cause and handled IF the caller wants or needs to do that when using custom `BufferedImageOps`.

- * Detailed notations about performance optimizations the caller can make to ensure their handling of images are as performant as possible were added to all the methods as a convenience.

- * Defined `DEBUG` system property name as a public constant that can be used to help avoid misspellings when trying to set debugging on.

- * Modified `LOG_PREFIX` so it can now be set via the `"imgscalr.logPrefix"` system property value now.

- * Rewrote `imgscalr` test suite to specifically test all discrete operations and all variations of the operations as well.

- * Added `AllTests` test suite so all tests can be easily run at one time to verify the release.

- * Rewrote Javadoc covering a lot of the return and exception conditions for all the methods to more clearly communicate what is happening inside the method and to the original images.

3.2

- * Added support for asynchronous & rate-limited scaling operations via the `AsyncScalr` class.

The `AsyncScalr` class wraps the parent `Scalr` class and submits scale jobs to an internal `ExecutorService`. The executor service can be used to serialize and queue up scaling operations to avoid blowing the heap and overloading the underlying host on a busy, multi-user system (e.g. a web app running `imgscalr`).

`AsyncScalr` by default uses a fixed-size `ThreadPoolExecutor` that can be modified at run time to any tuned level of threads the caller desires (default 2). The default settings are intended to be safe/efficient to use out of the box on most all systems.

Additionally, `AsyncScalr` can be configured to use *any* `ExecutorService` implementation passed to it so callers have ultimate control over how the `AsyncScalr` processes jobs if they need/want it.

Typically it is a good idea to roughly map # of Scaling Threads to the # of Cores on the server, especially on a server with plenty of memory and a large heap for the VM.

If you are running inside of a smaller VM heap or lower-memory server (regardless of core count) you will want to limit the number of simultaneous scale operations so as not to saturate the heap during scaling when the images are read into internal `BufferedImage` instances in VM memory.

* Added support for Rotation to the library. You can now specify the following rotations to be applied to your image:

`Rotation.NONE` - No rotation.

`Rotation.CLOCKWISE` - Clockwise (90 degrees to the right) rotation.

`Rotation.COUNTER_CLOCKWISE` - Counter-clockwise (90 degrees to the left) rotation.

`Rotation.FLIP` - Flip the image (180 degrees rotation).

The rotation is performed as tightly and efficiently as possible, explicitly cleaning up temporary resources created during the operation.

* API was simplified as duplicate methods without the vararg parameter were removed (these were effectively duplicates of the vararg methods make the API longer than it needed to be).

* Corrected a multitude of incorrect Javadoc comments pertaining to `@throws` conditions.

* Rewrote the method Javadoc. Manually reviewing uncovered too many copy-paste discrepancies that left out important information that would be helpful in a Javadoc popup in an IDE while using `imgscalr`.

* All new code heavily commented.

3.1

* You can now specify `Mode.FIT_TO_WIDTH` or `Mode.FIT_TO_HEIGHT` behaviors when resizing an image to get `imgscalr` to treat one dimension as the primary and recalculate the other dimension to best fit it, regardless of the image's orientation. Previously this was decided automatically for you by the orientation of the image.

* `resize` methods now accept 0 or more `BufferedImageOps` as var-arg arguments.

* Workaround for a 10-year-old JDK bug that causes `RasterExceptions` to get thrown from inside of Java2D when using `BufferedImageOps` was built directly into `imgscalr` so you don't have to worry about `RasterExceptions`. More info here: <https://github.com/thebuzzmedia/imgscalr/issues/closed#issue/23>

* API was made more strict and an `IAE` is thrown if `'src'` is null to any of the `resize` operations; a user reported that he spent a while debugging why "`imgscalr` wasn't working" only to find out it was silently returning due to a null source image. Would have been helpful if `imgscalr` had notified him of the issue immediately.

3.0

* Big thanks to Magnus Kvalheim from <http://www.movellas.com/> for help with this release!

* Support for hardware-accelerated `BufferedImageOp`'s was added to the library. You can now provide an optional `BufferedImageOp` to many of the methods in the `imgscalr` library and it will be applied to the resultant image before returning it.

- * Most common request was for imgscalr to apply an "anti-aliasing" filter to results before returning them; this was achieved by adding support for BufferedImageOps and providing a hand-tuned ConvolveOp to provide a good default that can be applied easily by folks that want the effect but don't want to learn all about BufferedImageOps and what "convolve" even means.
- * Speed/Balance/Quality THRESHOLD values were adjusted for more optimal results when relying on Method.AUTOMATIC to give good-looking results.
- * Javadoc was updated to clarify hardware acceleration behaviors.

2.1

- * Scaling of certain image types (and byte layouts) could result in very poor looking scaled images ("pixelated" look, discolored dithering, etc.). This was corrected by imgscalr forcibly scaling all source images into the most well-supported image types by Java2D, resulting in excellent scale result quality regardless of the Method specified.
- * The issue of scaling of poorly supported (by Java2D) image-types can lead to unexpectedly poor performance was also corrected as a side-effect of this because all source images are converted to the most commonly supported image type for Java2D.

2.0

- * API-break: `resize(BufferedImage, Method, int, int, boolean, boolean)` was removed and replaced by `resize(BufferedImage, Method, int, int)`.
- * DEBUG system variable added; set 'imgscalr.debug' to true to trigger debugging output in the console. The boolean debug and elapsedTime arguments to the resize method have been removed.
- * New BALANCED method added. Provides a better result than SPEED faster than QUALITY.
- * Added 2 optimized thresholds (in pixels) that the API uses to select the best Method for scaling when the user specifies AUTOMATIC (or doesn't specify a method). This helps provide much better results out of the box by default and tightens up the performance of the API a bit more.
- * Image comparison generator utility (ComparisonGenerator test class) added.
- * Functional portions of API broken into static protected methods that can be easily overridden by implementors to customize the API without needing to rewrite the resize methods.
- * Consolidated 5 locations of duplicated rendering code into a single method (`scaleImage`).
- * Tightened up image scaling operation to do everything possible to avoid memory leaks (every native resource is disposed or released explicitly)
- * Detailed logging information integrated. If the 'imgscalr.debug' system property is true, the API outputs exactly what it's doing, what argument values it is processing and how long it is taking to do each scale operation.
- * When AUTOMATIC method is specified, the API is more intelligent about selecting SPEED, BALANCED or QUALITY based on the images primary dimension only (more accurate).
- * Copious amounts of Javadoc added to new methods, new code and existing code.

Issues Resolved in 2.0:

<https://github.com/thebuzzmedia/imgscalr/issues/closed>

1.2

* Default proportional-scaling logic is more straight forward. If an image is landscape then width is the preferred dimension and the given height is ignored (and recalculated) and visa-versa if the image is portrait oriented. This gives much better "default behavior" results.

* Added new convenience method `resize(BufferedImage,int,int)`

* Modified build.xml to output Maven-friendly artifact names.

* Library now available from the <http://maven.thebuzzmedia.com> Maven repo, see "Maven" section on this page for more information:

<http://www.thebuzzmedia.com/software/imgscalr-java-image-scaling-library/>

* Javadoc is now available on the web at:

<http://www.thebuzzmedia.com/downloads/software/imgscalr/javadoc/index.html>

Issues Resolved in 1.2:

<https://github.com/thebuzzmedia/imgscalr/issues/closed>

1.1

* Initial public release.

License

This library is released under the Apache 2 License. See LICENSE.

Description

A class implementing performant (hardware accelerated), good-looking and intelligent image-scaling algorithms in pure Java 2D. This class implements the Java2D "best practices" when it comes to scaling images as well as Chris Campbell's incremental scaling algorithm proposed as the best method for down-sizes images for use as thumbnails (along with some additional minor optimizations).

imgscalr also provides support for applying arbitrary `BufferedImageOps` against resultant images directly in the library.

TIP: imgscalr provides a default "anti-aliasing" Op that will very lightly soften an image; this was a common request. Check `Scalr.OP_ANTIALIAS`

TIP: All resizing operations maintain the original images proportions.

TIP: You can ask imgscalr to fit an image to a specific width or height regardless of its orientation using a Mode argument.

This class attempts to make scaling images in Java as simple as possible by providing a handful of approaches tuned for scaling as fast as possible or as best-looking as possible and the ability to let the algorithm choose for you to optionally create the best-looking scaled image as fast as possible without boring you with the details if you don't want them.

Example

In the simplest use-case where an image needs to be scaled to proportionally fit a specific width (say 150px for a thumbnail) and the class is left to decide which method will look the best, the code would look like this:

```
BufferedImage srcImage = ImageIO.read(...); // Load image
BufferedImage scaledImage = Scalr.resize(srcImage, 150); // Scale image
```

You could even flatten that out further if you simply wanted to scale the image and write out the scaled result immediately to a single line:

```
ImageIO.write(Scalr.resize(ImageIO.read(...), 150));
```

Working with GIFs

Java's support for writing GIF is... terrible. In Java 5 it was patent-encumbered which made it mostly totally broken. In Java 6 the quantizer used to downsample colors to the most accurate 256 colors was fast but inaccurate, yielding poor-looking results. The handling of an alpha channel (transparency) while writing out GIF files (e.g. `ImageIO.write(...)`) was non-existent in Java 5 and in Java 6 would remove the alpha channel completely and replace it with solid BLACK.

In Java 7, support for writing out the alpha channel was added but unfortunately many of the remaining image operations (like `ConvolveOp`) still corrupt the resulting image when written out as a GIF.

NOTE: Support for scaling animated GIFs don't work at all in any version.

My recommendation for working with GIFs is as follows in order of preference:

1. Save the resulting `BufferedImage` from `imgscalr` as a PNG; it looks better as no quantizer needs to be used to cull down the color space and transparency is maintained.
2. If you mostly need GIF, check the resulting `BufferedImage.getType()` to see if it is `TYPE_INT_RGB` (no transparency) or `TYPE_INT_ARGB` (transparency); if the type is ARGB, then save the image as a PNG to maintain the alpha channel, if not, you can safely save it as a GIF.
3. If you MUST have GIF, upgrade your runtime to Java 7 and save your images as GIF. If you run Java 6, any GIF using transparency will have the transparent channel replaced with BLACK and in Java 5 I think the images will most all be corrupt/invalid.

REMINDER: Even in Java 7, applying some `BufferedImageOps` (like `ConvolveOp`) to the scaled GIF before saving it totally corrupts it; so you would need to avoid that if you didn't want to save it as a PNG. If you decide to save as a PNG, you can apply any Ops you want.

Troubleshooting

Image-manipulation in Java can take more memory than the size of the source image because the image has to be "decoded" into raw ARGB bytes when loaded into the

BufferedImage instance; fortunately on most platforms this is a hardware-accelerated operation by the video card.

If you are running into OutOfMemoryExceptions when using this library (e.g. if you dealing with 10+ MB source images from an ultra-high-MP DSLR) try and up the heap size using the "-Xmx" command line argument to your Java process.

An example of how to do this looks like:

```
java -Xmx128m com.site.MyApp
```

Reference

Chris Campbell Incremental Scaling - <http://today.java.net/pub/a/today/2007/04/03/perils-of-image-getscaledinstance.html>

Related Projects

ExifTool for Java - <http://www.thebuzzmedia.com/software/exiftool-enhanced-java-integration-for-exiftool/>

Contact

If you have questions, comments or bug reports for this software please contact us at: software@thebuzzmedia.com

Releases

 11 tags

Packages

No packages published

Used by 10.2k



Contributors 8



Languages

● Java 100.0%