Manual
JIU – The Java Imaging Utilities

Marco Schmidt

http://schmidt.devlib.org/jiu/

October 16, 2005

Contents

Chapter 1

Introduction

1.1 About JIU

I started the *JIU* (*Java Imaging Utilities*) project [jiu00] in late 2000 in order to learn about algorithms for image processing, editing, analysis and compression and implement codecs for some file formats.

All code is written in the *Java programming language*. *JIU* is supposed to run with Java runtime environments version 1.1 and higher. This has the advantage of working under most operating systems in use today, 1.1-compatible virtual machines have been written for about any platform. A minor disadvantage is the lack of interesting classes from any higher Java versions, such as the `Iterator` class introduced in 1.2. Some (seemingly) basic functionality like sorting had to be rewritten because it was not yet available in Java 1.1.

JIU is distributed under the *GNU General Public License* version 2.

I still consider JIU as beta software. Not only is it not heavily tested, but I know that I will make changes to it, including its package structure.

So please, do not rely on JIU for any mission-critical parts of your project!

1.2 About this manual

This manual is a work in progress, just like *JIU* itself. Both are very dependent on my spare time. I started writing the manual on October 17th, 2001.

It is written for LaTeX, so that DVI, PostScript and PDF documents can be generated from it. The text file that is the basis for this manual (`manual.tex`) is also included in the source code distribution of *JIU*.

1.3 Why Java?

It seems that image processing algorithms demand more resources—CPU cycles and memory—than many other fields of computing. So why use Java, which seems to be so high-level and wasteful with these resources? It's not like I *had to* use that particular language, I could have implemented the library in C, C++, Haskell, Ada, Delphi, whatever.

When I started this section, I had separated parts for advantages and disadvantages. I gave that up and will just list points of interest in regard to picking Java. For some of these points, it

is simply not clear whether they are a true advantage or disadvantage.

- *Cross-platform.* Java and its bytecode concept lead to true cross-platform development–no more ifdefs to differentiate between platforms with different-sized integer types, etc.
- *Availability* Especially in its 1.1 version, which is used by *JIU*, Java is available on most platforms. C and C++ may still have an advantage there, but Java also covers almost all systems from PDAs to high-end servers.
- *Runtime library.* Java's runtime library is very rich. From lists and hashes to Unicode support and other features for i18n, the developer does not have to reinvent the wheel.
- *Built-in cross-platform GUI* Actually, this is more of a combination of points already mentioned. But writing a GUI application that will not look, but at least mostly work the same on very different platforms, is great when dealing with images.
- *Object-orientation.* It is true that OOP is not a panacea, but it helps enforcing good design. Encapsulation, polymorphism and inheritance and the well-known patterns often lead to more elegant solutions. Unfortunately, Java—at least in its current version(s)—lacks a few features of a true OOP language. As an example, there are primitive types that are not derived from `Object`.

TODO

1.4 Why another Java library for imaging?

Okay, so there are reasons to use Java. But why start a completely new thing? There is Sun's own Java extension for imaging, *Java Advanced Imaging* (JAI), as well as a couple of other projects, with full source code availability and a user base. Why not implement for one of those libraries instead of starting from scratch?

TODO

1.5 Credits

- Thanks to SourceForge [sou00] for hosting the *JIU* project!

Chapter 2

JIU basics for developers

This chapter is for people who don't know *JIU* and want to learn the basics.

2.1 Image data types

2.1.1 Interfaces

The package `net.sourceforge.jiu.data` contains interfaces for the most basic types of images that get typically used in image processing. In the same package you will find implementations of those interfaces that store image data completely in memory. This will work nicely with images that fit well into the system's memory.

- `BilevelImage` for images with the two colors black and white (e.g. faxes).
- `Gray8Image` for images with shades of gray (e.g. photographs).
- `Paletted8Image` for color images that have 256 or less different colors.
- `RGB24Image` for truecolor image in RGB color space.

More on the topic of the various classes and interfaces dealing with image data types can be found in chapter 4.

2.1.2 Classes

As was said before, for all these interfaces data types exist that implement them by storing the complete image in memory. The names of those classes start with `Memory` and then simply copy the name of the interface that they are implementing. Examples: `MemoryBilevelImage`, `MemoryRGB24Image`.

In order to create a new image object, simply call the constructor with width and height as arguments. If you want an RGB truecolor image, this would be it:

```
import net.sourceforge.jiu.data.*;
...
MemoryRGB24Image image = new MemoryRGB24Image(1200, 800);
```

See the API docs of the interfaces for code examples to get and set pixels.

2.1.3 AWT image data types

What about Java's existing image classes, `java.awt.Image` and its children? You can convert between them and *JIU*'s own image data classes, but you cannot use them directly. Conversion is done with *JIU*'s `ImageCreator` class. Example:

```
import java.awt.*;
import net.sourceforge.jiu.gui.awt.*;
...
PixelImage jiuImage = ...; // initialize
java.awt.Image awtImage = ImageCreator.convertToAwtImage(jiuImage);
// and back
RGB24Image anotherJiuImage = ImageCreator.convertImageToRGB24Image(awtImage);
```

As the conversion method from AWT to *JIU* uses AWT's `PixelGrabber` (which works on 24 bit pixels in RGB color space) you will always get back an `RGB24Image`.

2.2 Loading images from files

Most of the time, images will be loaded from a file, processed and then saved back to a file. *JIU* comes with a number of codecs that can do exactly that. In addition, *JIU* can use `java.awt.Toolkit` to reuse the image loading functionality built into the Java Runtime Environment. That makes it possible to load images from JPEG, GIF and PNG files.

```
import net.sourceforge.jiu.gui.awt.*;
...
PixelImage image = ToolkitLoader.loadViaToolkitOrCodecs("filename.jpg");
```

This basically just tries all *JIU* codecs plus `java.awt.Toolkit`. If you use codecs directly, you can retrieve additional information, make the codec load another image than the first one, make the codec load only part of an image, etc. But the easiest method of loading an image is the one-liner you see above.

If you must avoid AWT for some reason (the bug that keeps an AWT thread from terminating, missing X server, etc.), use the class `ImageLoader` from the package `net.sourceforge.jiu.codecs`. It will try all codecs that are part of *JIU*:

```
import net.sourceforge.jiu.codecs.ImageLoader;
...
PixelImage image = ImageLoader.load("filename.bmp");
```

2.3 Operations

Now that you've created or loaded the image you will probably want to do something with it. In *JIU*, you'll need an *operation* class to do that (all classes that extend `Operation` are operations). Many operations are not derived directly from `Operation` but from `ImageToImageOperation`, which requires that the operation takes an input image and produces an output image. *JIU*'s operations are put into different packages, depending on the type of operation.

2.3.1 Creation and usage

Running operations always follows the same pattern:

- Create an object of the operation's class.

- Give all necessary parameters to that object (via methods whose names start with `set`). Read the API documentation of the operation class to learn about what parameters exist for a given operation, if they are mandatory or optional and what the default values are. Most of the time you will also find a code snippet that demonstrates the usage.

- Call the `process()` method of that object, catching all exceptions that may be thrown (`OperationFailedException` or children of that exception class).

- Retrieve any output that the operation might have produced (via methods whose names start with `get`). Again, refer to the API documentation of the specific operation that you are trying to use.

As an example, let's say that you have just loaded `image` from a file, as seen in the previous section. Now you want to make the image brighter and decide for a 30% brightness increase. There is a class `Brightness` in the package `net.sourceforge.jiu.color.adjustment`.

```
import net.sourceforge.jiu.color.adjustment.*;
...
Brightness brightness = new Brightness();
brightness.setInputImage(image);
brightness.setBrightness(30);
brightness.process();
PixelImage adjustedImage = brightness.getOutputImage();
```

Just in case you wonder - `PixelImage` is the most basic interface for image data. Everything in *JIU* that stores an image must implement it. Because `ImageToImageOperation` does not make any assumptions about the types of image data classes that its extensions will deal with, both `getInputImage` and `setInputImage` deal with this interface.

2.3.2 Exceptions

Not all errors made when using operations can be determined at compile time. As an example, if you give an image to an operation which is not supported by that operation, this will be determined only after `process` has been called. Also, some operations may fail under specific circumstances only – not enough memory, a particular file does not exist, etc. For these errors the `process()` method of `Operation` can throw exceptions of type `OperationFailedException`. Catch these exceptions to find out about what went wrong, they contain textual descriptions in English:

```
Operation operation = ...; // initialize
try
{
  operation.process();
}
catch (OperationFailedException ofe)
{
  System.err.println("Operation failed: " + ofe.toString());
}
```

2.4 Saving images to files

There is no class like `ImageLoader` to do saving with a single line of code. But saving works pretty much the same with all of *JIU*'s codecs. Here is an example that uses `PNMCodec` (which supports PBM, PGM and PPM). It will save `image` to a file `output.pnm`:

```
import net.sourceforge.jiu.codecs.*;
...
PNMCodec codec = new PNMCodec();
codec.setFile("output.pnm", CodecMode.SAVE);
codec.setImage(image); // image to be saved
codec.process();
codec.close();
```

Except for the first line where the codec object is created, the rest of the code can be used with `BMPCodec`, `PalmCodec` or any of the other codecs that support saving.

Chapter 3

Terminology

This chapter defines some terminology to be used throughout this manual, other parts of the library's documentation and its source code. It clarifies language for those with background knowledge on the field of image processing and introduces concepts to beginners.

3.1 Images

Images are at the core of this library. They are data structures storing color dots arranged in a two-dimensional grid of a fixed size.

An *image* is defined as a sequence of *rows*. The number of rows in an image is called its *height*. Rows are numbered from top to bottom from 0 to $height - 1$.

Each row of an image is a sequence of *pixels*. The number of pixels in a row is called its *width*. All rows of an image must have the same width. Pixels within a row are numbered from left to right from 0 to $width - 1$.

Given these definitions, every pixel can be uniquely addressed using a pair of integer numbers: its zero-based horizontal and vertical position. When stating a pixel position, this library names the horizontal position followed by the vertical position: (x, y). Note: this is somewhat different from mathematical notation with regard to matrices, which is usually defined the other way around, vertical before horizontal.

A pixel is a single color point. It is made up of one or more *samples*. All pixels of an image have the same number of samples. The samples of a single pixel define its color.

Behind every image there is a color model which defines how to get from a pixel's samples to that pixel's color.

There has been no mentioning yet of what a sample really is.

Chapter 4

Image data types

There are quite a few interfaces and classes for image types in *JIU*. In fact, enough to be a bit confusing for someone trying to get a first impression of the library. This chapter is about both about the image classes from the Java runtime library and those introduced by *JIU*.

4.1 Accessing image data with the AWT classes

As I wrote in the first chapter, one of my reasons for writing *JIU* was that I cannot agree with some of the design decisions made in the runtime library with regard to imaging. The lack of different image class types in the AWT (Abstract Windowing Toolkit, the package `java.awt` and everything in it) is one of them. A single abstract `Image` class with almost no methods is nice if all you do with images is load them from somewhere and display them in your GUI application. Which is exactly what imaging was like in Java 1.0. Remember applets?

However, once you want to manipulate images, there really should be some methods to access data. With `ImageProducer`, `ImageConsumer` and the various filter classes in `java.awt.image` there was a way to manipulate data, but not a straight-forward one. Besides, the setPixels method takes a byte array and a color model as parameters. You cannot just put a green pixel at position (x, y). Or a grey value with 16 bits of precision.

Only with Java 1.2 an image class was introduced (`BufferedImage`) that comes with getter and setter methods for pixels. Unfortunately, these access methods are restricted to RGB pixels with 24 bits which must be encoded as `int` values for some reason. Each RGB pixel must be put together using shift and or operations, requiring to specify a transparency value as well. Not straight-forward. There isn't even a helper encoder and decoder class for those ARGB values. You can also access the data buffers of a `BufferedImage` object, but again, you better know what types were used and that data is stored top to bottom, and in each row from left to right. Also, it's not easy to find out how to manipulate a palette (color map) for an image with 256 distinct colors.

To summarize–a single (or two) classes aren't enough to represent the wide variety of image types in use. Faxes, medical imaging, satellite data, photos, image data suitable for printing all need different data types. Being forced to have knowledge on how things are done internally is bad design.

4.2 Image data interfaces in *JIU*

This gives an overview of the most important image data interfaces and their implementations. You might also want to look into the source code of `net.sourceforge.jiu.data`–it's relatively little and, hopefully, readable.

4.2.1 PixelImage

JIU's base pixel image data interface is `PixelImage`. It only knows about its resolution (width and height) and the number of channels that it consists of. The smallest common denominator. A *sample* is a single value in one channel at a certain position. A *pixel* is the combination of all samples from all channels at the same position. If an image has only one channel, the terms sample and pixel can be used interchangeably.

4.2.2 IntegerImage

Derived from `PixelImage` is `IntegerImage`. It only adds the restriction that each sample must be an integer value between 0 and $2^{31} - 1$ (also known as `Integer.MAX_VALUE`) so that it can be stored in Java's `int` type (which is a signed 32 bit integer type). Note that this still does not make any assumptions on how those integer values are interpreted as colors. A value of 0 at a certain position in a one-channel image has no meaning yet.

4.2.3 GrayImage, RGBImage, PalettedImage

The meaning of what the numbers represent comes with this set of interfaces. Note that they are not derived from any other interface.

- `GrayImage` is for images with one channel where values go from black over various shades of gray (the number depends on the available precision) to white.
- `RGBImage` is for truecolor images using the RGB color space. It requires three channels for the color components red, green and blue.
- `PalettedImage` is for images that store index values into a list of colors, the palette. This image type always has only one channel.

4.2.4 GrayIntegerImage, RGBIntegerImage, PalettedIntegerImage

This layer of interfaces combines `IntegerImage` and the three interfaces from the previous section which define the meaning of an image type's content.

- `GrayIntegerImage` is for grayscale images that use integer values up to `int` as samples.
- `PalettedIntegerImage` is for paletted images that use `int` samples.
- `RGBIntegerImage` - same here, each of the three components stores `int` samples.

Although these interfaces describe the meaning, it is still unclear what interval a sample must be from. Values must fit into 32 bits because of the super interface `IntegerImage`, so there is an upper limit. But samples can be smaller, and for efficiency reasons there are types that use less space than an int for each sample.

4.2.5 BilevelImage, Gray8Image, Gray16Image, RGB24Image, Paletted8Image

These four interfaces are derived from the aforementioned three interfaces and define the number of bits for each sample.

- `BilevelImage` extends `GrayIntegerImage` and its pixels have only two possible values – black and white.
- `Gray8Image` also extends `GrayIntegerImage` and uses eight bits per sample, allowing for 256 shades of gray.

- `Gray16Image` also extends `GrayIntegerImage` and uses sixteen bits per sample, allowing for 65536 shades of gray.
- `Paletted8Image` is a `PalettedIntegerImage` that uses eight bits for the index values. Thus, it can be used for palettes with up to 256 entries.
- `RGB24Image` uses eight bits for each of its three channels, for a total of 24 bits.

4.3 Implementations of the image interfaces

Keep in mind that the previous section introduced a lot of types, but they were all interfaces. You will need an implementation of them to actually work on real images. Right now, there is only an in-memory implementation of them.

It is planned to provide disk-based implementations as well. In combination with a caching system this will enable the processing of very large images.

4.3.1 In-memory: MemoryBilevelImage, MemoryGray8Image, MemoryRGB24Image, MemoryPaletted8Image

These are in-memory implementations of the four interfaces described in the previous section. A byte array that is large enough will be allocated for each channel. This will allow fast and random access to samples. However, resolution is limited by the system's main memory (or more precisely, the amount of memory given to a virtual machine).

Chapter 5

Demo program `jiuawt`

JIU comes with a GUI (graphical user interface) program that lets you play with its various features. The program is based on AWT (the Abstract Windowing Toolkit), not Swing, so that more people can run it (Swing has only been part of Java since version 1.2). All that is required is an installed Java 1.1 Runtime Environment and a system that can be put into some sort of graphics mode.

If you have downloaded the non-core version of *JIU*, you should have a JAR archive called `jiu.jar` in the ZIP archive that you downloaded. Let's say that you have extracted that JAR archive from the ZIP archive.

There are several ways to start the jiuawt application.

- With some Java Runtime Environments (JREs), it is enough to double-click on the JAR archive in some file manager, e.g. the file explorer under Windows.
- If double-clicking doesn't work for you, open a shell (sometimes it's called console, or command prompt, DOS prompt, etc.). Some window where you can enter commands. Change to the directory where the JAR archive is stored. Start the program by typing `java -jar jiu.jar`. Under Windows, `start jiu.jar` might work as well.
- If your JRE is a bit older, the `-jar` switch may be unknown. In that case, change to the directory in the shell and type `java -cp jiu.jar net.sourceforge.jiu.apps.jiuawt`.

The jiuawt program requires quite a bit of memory, depending on the size of the images that are processed in it. Java virtual machines often do not give all of the available memory to a running program. In order to give an application more than the default amount, use the `-mx` switch of the `java` program. Example: `java -mx128m -jar jiu.jar`. This will give 128 MB to the virtual machine. Make sure that the `-mx` switch is the first argument to the VM.

If you are planning to use jiuawt on a regular basis, you might want to create a link to the program. Under Windows, try calling it with `javaw.exe` instead of `java.exe`. That way, you will not have a DOS box popping up.

Chapter 6

An overview of built-in classes

This chapter is a reference of all classes that are part of JIU. These classes can be classified more or less to belong into the categories operations, data classes and helper classes.

UNFINISHED

6.1 Image data

`net.sourceforge.jiu.data` contains interfaces and classes for the storage of image data, maybe the most essential package of *JIU* – after all, operations work on image data.

6.2 Operations

`net.sourceforge.jiu.ops` has the base operation classes, plus exceptions for most typical failures that can occur in operations. An interface for progress notification is also provided here.

6.3 Codecs

Codecs are classes to read images from and write them to files (or in some cases, more generally, streams). The package `net.sourceforge.jiu.codecs` provides the base codec class `ImageCodec` and codecs for several image file formats.

ImageCodec Abstract base class for image I/O operations. Supports progress notification and bounds definitions to load or save only part of an image.

6.4 Color

`net.sourceforge.jiu.color` offers operations that modify or analyze the color of an image.

6.4.1 Analyzing color

AutoDetectColorType (**??**) Checks if an image can be converted to an image type that uses less memory without losing information. Can perform that conversion if wanted.

TextureAnalysis Takes the co-occurrence matrix of an image and computes several properties based on it.

6.4.2 Decreasing color depth

Several operations deal with the process of converting images in a way so that they will be of a different color type. The following operation deal with conversions that will lead to a loss of information, which usually also means that less memory will be required for the new version of the image.

ErrorDiffusionDithering (**??**) Adjust the brightness of an image, from -100 percent (resulting image is black) to 100 percent (resulting image is white).

OrderedDither (**??**) Adjust the brightness of an image, from -100 percent (resulting image is black) to 100 percent (resulting image is white).

RgbToGrayConversion (**??**) Adjust the brightness of an image, from -100 percent (resulting image is black) to 100 percent (resulting image is white).

6.5 Other color modifications

Brightness (**??**) Adjust the brightness of an image, from -100 percent (resulting image is black) to 100 percent (resulting image is white).

Invert Replace each pixel with its negative counterpart - light becomes dark, and vice versa. For color images, each channel is processed independent from the others. For paletted images, only the palette is inverted.

6.6 Filters

The `net.sourceforge.jiu.filters` package has support for convolution kernel filters and a few non-linear filters.

6.7 Transformations

The `net.sourceforge.jiu.transform` package provides common transformation operations, including scaling, rotating, shearing, cropping, flipping and mirroring.

6.8 Color data

A set of interfaces and classes for histograms, co-occurrence matrices and co-occurrence frequency matrices. Operations to create and initialize these data classes can be found in the color package.

6.9 Color quantization

`net.sourceforge.jiu.color.quantization` provides interfaces and classes for dealing with color quantization, the lossy process of reducing the number of unique colors in a color image. There are enough classes related to this field of color operations to justify a package of its own.

6.10 Applications

JIU comes with a couple of demo applications. The package `net.sourceforge.jiu.apps` contains these applications as well as classes with functionality used by all demo applications.

6.11 GUI - AWT

The `net.sourceforge.jiu.gui.awt` hierarchy contains all classes that rely on the Abstract Windowing Toolkit (AWT), the packages from the `java.awt` hierarchy of the Java core libraries.

If you don't use this part of JIU, your application will not be dependent on a target system having an X Window server installed, or any other GUI capabilities.

This package provides classes for interoperability of *JIU* and AWT classes like `java.awt.Image`.

6.12 GUI - AWT dialogs

`net.sourceforge.jiu.gui.awt.dialogs` contains a set of dialog classes that are used by the AWT demo application `jiuawt`.

6.13 Utility class

`net.sourceforge.jiu.util` holds everything that didn't fit elsewhere. Right now, this includes things as different as sorting, getting system information and operations on arrays.

Chapter 7

Writing operations

7.1 Basics

The base class for all classes performing analysis, modification and serialization of images or image-related data is `net.sourceforge.jiu.ops.Operation`. Any new operation will have to be directly or indirectly derived from that ancestor class.

If you are going to contribute your code to *JIU* itself, contact the maintainers, describe the operation and ask if it is of interest for *JIU*. Maybe somebody is already writing this sort of operation, or maybe it does not fit into *JIU*. If you contribute to *JIU*, read the coding conventions (chapter 9 on page 31ff) first. Use some package from the `net.sourceforge.jiu` hierarchy (also ask the maintainers for a suitable package; maybe a new one has to be created).

Instead of directly extending `Operation`, study some of its child classes, maybe it is more suitable to extend one of them.

- `ImageCodec` – An operation to load or save images from or to streams or files. Chapter 8 is dedicated completely to image codecs.

- `ImageToImageOperation` – Any operation that takes one or more input images and produces one or more output images. See 7.2 for more information.

- `LookupTableOperation` – An extension of `ImageToImageOperation` that takes an input image of type `IntegerImage` and some tables and produces an output image of the same type by looking up each sample of each channel of the input image in the appropriate table and writing the value found that way to the output image at the same position in the same channel.

  This is the right choice for operations that process each sample independent from all other samples of the same pixel and all other pixels of the image. As a side effect, it is—at least in theory—easy to parallelize these kinds of operations, in order to take advantage of a multiprocessor system. This kind of optimization is not implemented in JIU (yet).

  Note that looking up a value in an array is relatively expensive. If the operation in question is just a simple addition, you might want to compute the result instead of looking it up. It depends on the Java Virtual Machine, the hardware and the exact nature of the operation which approach is faster. You might want to do some tests.

7.2 Using `ImageToImageOperation`

As mentioned before, `ImageToImageOperation` takes one or more input images and produces one or more output images.

7.3 Exceptions

The `Operation.process()` method has one exception in its `throws` clause: `OperationFailedException`. That exception class is the base for all exceptions to be thrown during the execution of `process`.

7.4 Progress notification

In some cases, operations might get used in end user applications. Human beings tend to be impatient or fear that the computer has locked up if nothing happens on the screen for a longer time. That is why the `Operation` class supports the concept of *progress notification*.

All objects that want to be notified about the progress status (in terms of percentage of completion) of an operation must implement the `ProgressListener` interface. The objects must then be registered with the `Operation` by giving them as arguments to `Operation.addProgressListener`.

An operation supporting the progress notification concept must call one of the the `setProgress` methods in regular intervals. The `setProgress` methods of `Operation` are very simple—they go over all registered `ProgressListener` objects and call their respective `setProgress` methods with the same progress argument(s). This could lead to a progress bar being updated in a GUI environment, or a dot printed on the console.

Also see the API docs of

- `Operation` and
- `ProgressListener`

and check out some operation classes that use `setProgress`. Most of the time, it will be called after each row that has been processed, with the current row number of the total number of rows as parameters.

Chapter 8

Writing image codecs

8.1 Introduction

The package `net.sourceforge.jiu.codecs` is responsible for loading images from and saving them to files or arbitrary streams. `ImageCodec` is the ancestor class for all operations loading or saving images. It extends JIU's base class for operations, `net.sourceforge.jiu.ops.Operation`. This section of the manual describes how to write a new codec that fits into JIU. Looking at the source code of an existing codec should help, too, although this section will contain code examples.

It is recommended to read chapter 7 on writing operations first.

If the codec is supposed to be included into JIU itself (which is not necessary, everybody is free to use JIU as long as the licensing rules are obeyed when distributing JIU itself as part of a product), the JIU maintainer(s) should be contacted first and asked whether a new codec for a particular file format is already in the making and if that file format is of interest for JIU.

If the codec will become part of JIU, its coding conventions (see chapter 9) must be used to maintain overall readability of the source code.

8.2 Basics

If the codec will be part of JIU, it must be put into the package `net.sourceforge.jiu.codecs`. When writing a new codec, you will have to override the `ImageCodec` class. Let's say you want to implement the (fictional) ACME image file format. The class name should be assembled from the file format's name (its short form, for brevity reasons) and `ImageCodec` at the end of the name, so in this case:

```
public class ACMEImageCodec extends ImageCodec { ... }
```

8.2.1 Format name

Override the method `getFormatName` and make it return a short String containing the name of the file format with the most popular file extension in parentheses:

```
public void getFormatName()
{
  return "ACME Inc. (ACM)";
}
```

Do not include *file format* or *image file format* in that description so that the format name can be used in programs with other natural languages than English.

8.2.2 File extensions

Override the method `getFileExtensions` and make it return all file extensions that are typical for the file format in lowercase. In case of our fictional ACME file format, this could be:

```
public void getFileExtensions()
{
  return new String[] {".acm", ".acme"};
}
```

Override the method `suggestFileExtension(PixelImage image)` to return a file extension that is most appropriate for the argument image object. In most cases, a file format only has one typical file extension anyway. However, some formats (like Portable Anymap) have a different file extension for every image type (grayscale will use `.pgm`, color `.ppm` etc.). For the sake of simplicity, let's say that ACME uses `.acm` most of the time, so:

```
public String suggestFileExtension(PixelImage image)
{
  return ".acm";
}
```

This does not have to take into account that the argument image may not be supported at all. That will be checked elsewhere.

8.2.3 Supported actions

Override the methods `isLoadingSupported` and `isSavingSupported` to indicate whether loading and saving are supported.

8.3 Usage example

For a moment, let's get away from writing a codec and take a look at how it will be used. Minimum code example for loading an image:

```
ACMEImageCodec codec = new ACMEImageCodec();
codec.setFile("image.acm", CodecMode.LOAD);
codec.process();
PixelImage image = codec.getImage();
codec.close();
```

Minimum code example for saving an image:

```
PixelImage image = ...; // this image is to be saved, initialize it somehow
ACMEImageCodec codec = new ACMEImageCodec();
codec.setImage(image);
codec.setFile("image.acm", CodecMode.SAVE);
codec.process();
codec.close();
```

To sum it up, the following steps are relevant for anybody using the codec:

1. Create an object of the codec class, using the constructor with an empty argument list.
2. Call the `setFile` method with the file name and the appropriate `CodecMode` object (`CodecMode.LOAD` or `CodecMode.SAVE`).

3. Give input parameters to it if they are necessary. Most of the time all you have to do is provide an image if you want to save to a file. Other parameters are possible, but they either depend on the file format or are not essential for the codec to work (e.g. defining bounds or dealing with progress notification).

4. Call the `process()` method which will do the actual work.

5. Call the `close()` method, which will close any input or output streams or files that have been specified. Maybe process itself calls `close()`, but calling it a second time shouldn't do any harm. Not closing streams can become a problem when very many streams are used in a program, e.g. in a batch converter.

6. This step is optional: get results, output parameters. The most obvious example for this is a `PixelImage` object when loading. The codec must provide get methods for all possible results, e.g. `getImage` for an image that was loaded in `process`.

This will be reflected in the codec itself.

8.4 The process method

It is the core of any `Operation` implementation and does the actual work. `ImageCodec` extends `Operation`, so this remains true.

As the inner workings of an image codec can become quite complex, having additional methods is a good idea–one huge process method is probably unreadable, unless it is a very simple file format. All methods (except for process and set and get methods to be used to provide and query information) of the new codec *must be declared private*. They are implementation details and of no relevance to the user of the codec.

8.5 Checking parameters

The first thing to do in any `process` method is checking the parameters.

- Are the mandatory parameters available? If not, throw a `MissingParameterException`.

- Are all parameters that have been specified valid? If not, throw a WrongParameterException (if the parameters type is wrong etc.) or a an UnsupportedTypeException (if a compression type to be used for saving is not supported by your codec etc.).

For all optional parameters that are missing, initialize them to their default values.

Note that some errors can only be detected later in the process. Example: when loading an image, you will have to do some decoding before you find out that, as an example, the file uses a compression method that you do not support.

8.6 Load or save

Now, find out whether you will have to load or save an image. Call `initModeFromIOObjects()`, it will find out whether to load or save from the kinds of I/O objects that have been given to the codec. If no I/O objects have been specified, that method will throw an appropriate exception. Then get the `CodecMode` using `getMode()`. If the resulting mode—either `CodecMode.LOAD` or `CodecMode.SAVE`—is not supported by your implementation, throw an `UnsupportedTypeException` with a descriptive error message.

8.7 I/O

`ImageCodec` provides set methods to specify input and output objects so that the codec can read or write data. The codec knows the following classes, in ascending order of their abilities:

- `InputStream` and `OutputStream` which only let you read and write byte(s) in a linear way without random access.
- Everything implementing `DataInput` and `DataOutput`, which let you do the same as `InputStream` and `OutputStream` but can also read and write types like `int` or `short` in network byte order (big endian).
- `RandomAccessFile` which implements both `DataInput` and `DataOutput`, thus offering everything these two do plus random access–you will be able to seek to any offset in the file and continue to read or write there.

If you can choose which of these classes you will use, pick the most primitive ones that will work for you. This will make it possible to use your codec in more environments. If your codec is alright with having an `InputStream`, it will not only work on files (`FileInputStream`) but also on network and other streams. However, some file formats like TIFF need `RandomAccessFile`.

For some file formats it may be necessary to wrap `InputStream` and `OutputStream` into some other I/O classes. As an example, to read or write text (as used in some subformat of Portable Anymap), `BufferedReader` and `BufferedWriter` are very convenient. To improve speed, put your `InputStream` and `OutputStream` objects into a `BufferedInputStream` or `BufferedOutputStream` object.

If your codec works with the interfaces `DataInput` and `DataOutput`, you will be able to cover the most cases. Try to do this whenever possible. Call the method `getInputAsDataInput` and you will be given either a `DataInput` object (if you directly specified one), or a `DataInputStream` (created from an `InputStream` if you specified one), or a `RandomAccessFile` object. All of them implement `DataInput`. That works with `DataOutput` as well, if you are saving an image.

Anyway, make sure that you have I/O objects and that they are of the correct type. If not, throw a `MissingParameterException`.

8.8 Reading and writing primitive values

Normally, image file formats demand that you read a lot of `byte`, `short` and `int` values in a certain order that you will have to interpret as, e.g., image width, color depth or compression type.

Instead of calling `read()` or `readInt()` each time you have to read a primitive value, load the complete header to a byte array. Then get the interesting primitives from that array using the class `net.sourceforge.jiu.util.ArrayConverter`. The following example will read 32 bytes and get bytes 12, 13, 14 and 15 as an `int` value in little endian byte order:

```
byte[] header = new byte[32];
in.readFully(header);
int width = ArrayConverter.getIntLE(12);
```

This approach will restrict the number of places for I/O errors to one, the call to the method that reads all the bytes (in the example `readFully`). Also, you don't have to skip over values that you don't care about – you just read everything and interpret only those places of the array that are necessary.

The same approach can also be used when writing an array. Create an array (it will by default be filled with zeroes), call the appropriate put methods of `ArrayConverter` and write the complete array to output.

8.9 Bounds

After you have read image width and height from the image, deal with the bounds. Check if bounds have been defined by querying `hasBounds()`. If there are no bounds, set them to the complete image:

```
setBounds(0, 0, width - 1, height - 1);
```

If there are bounds and you don't want to support them for whatever reason, throw an `OperationFailedException`. If the bounds do not match image width and height, throw an `WrongParameterException` (the bounds parameters were false).

8.10 Loading

After you have parsed the input stream for information, you can do some more checks.

If the image file format that you support in your new codec can have multiple images in one stream, call `getImageIndex` to find out which one to load. If the index is invalid (if there are not enough images in the stream), throw an `InvalidImageIndexException`.

Next, check if the image type, compression method etc. are supported by your codec and throw an `UnsupportedTypeException` otherwise.

Also check if the bounds—if present—fit the actual image resolution.

Create an image object of the right type and resolution. If you are regarding the bounds, use `getBoundsWidth` and `getBoundsHeight` instead of width and height as found in the stream.

Load the image and try to use progress notification via the `setProgress` methods.

Do not use more memory than necessary. As an example, do not load an uncompressed image into a large byte array, that will require twice the memory of the uncompressed image. Instead, create a row buffer, read row by row and put those rows into the image.

8.11 Performance issues

Even nicer than correct codecs (which read and write image files according to the specifications) are correct *and* fast codecs. Whatever fast means... Correctness should be prefered to speed, but higher speed can sometimes be reached by very simple means.

- The caller should use `BufferedInputStream` and `BufferedOutputStream` when giving streams to the codec. The codecs should not create buffered versions, that is the job of the caller.
- Instead of single bytes, process several bytes at a time. As an example, read a complete row and put a complete row into the image using `putSamples` instead of `putSample`.

8.12 Documentation

As for any other operation, create javadoc-compatible documentation for your image codec. Specify

- whether both loading and saving are supported,
- which I/O objects the codec can work with,
- which image types the codec can work with,
- which flavors of the file format are supported (e.g. compression types),
- whether the bounds concept of ImageCodec is supported,
- whether the progress notification concept of ImageCodec is supported,

- a description of all parameters new to this codec - what can be done with them, are they optional etc. and
- some background information on the file format if available—who created it and why, where is it used, etc.

For external HTML links in your documentation, use the `target` attribute with the value `_top`:

```
<a target="_top" href="http://somesite.com">Some site</a>
```

This way, a website will be displayed in the complete browser, not only the frame on the right side that previously held the class documentation.

Chapter 9

Coding conventions

This chapter is of interest for everybody who wants to contribute code to *JIU*. It is described here how code is to be formatted. Having the code follow certain rules consistently throughout all of *JIU*'s packages is important for readability and maintainability.

9.1 Import statements

All classes that are used in a class must be explicitly imported, except for classes from the `java.lang` package. Each imported class gets its own line, no empty lines between import statements. Do not use the asterisk to import a complete package like in `java.io.*`. Sort imports in ascending order, first by package name, then by the class (or interface) name.

9.2 Indentation

Use tab characters to express indentation. The tabs should be interpreted as four space characters.

9.3 Avoid large indentation levels

In order to make code more readable, avoid large levels of indentation. If quite a few lines are at level 4, 5, 6 or higher, you probably want to put that code in another method and call that method instead.

Nested loops often lead to high indentation levels.

If statements with else cases at the end of methods should be rewritten if one of the cases includes little and the other one much code.

if (a ¡ 0) a = 0; else // a lot of code // end of method

should become

if (a ¡ 0) a = 0; return; // a lot of code // end of method

9.4 Identifier names

Use meaningful names for identifiers. An exception to this can be made for local variables, especially when they are names of loop variables. Use only English as natural language. Do not use any characters but `a` to `z`, `A` to `Z` and digits `0` to `9` (so, no underscore character _). Avoid the digits whenever possible. Variable and method names must start with a lowercase letter (exception: final variable names, see 9.5). Class and interface names must start with an uppercase letter. Method names should start with a verb (get, set, is, etc.). Use capitalization to make reading names easier, e.g. `maxValue` instead of `maxvalue`. Avoid suffixes like `Interface` or `Impl` to express that a class is an interface or an implementation of some interface.

9.5 Final variable names

The letters in names of variables that are declared final must all be uppercase. This makes it of course impossible to use capitalization as suggested in 9.4. That is why the use of the underscore is allowed and encouraged in the names of final variables to separate words: `MAX_VALUE`.

9.6 Methods sorted by name

All methods of a class must be sorted by their names in ascending order, no matter what the access modifier of a method is.

9.7 Thrown exceptions in method signatures

Do not explicitly specify exceptions that extend `RuntimeException`, e.g. `IllegalArgumentException`. However, do document them.

9.8 Declaration of fields in classes

All fields, no matter whether they are class or instance variables, public or private, must be declared at the beginning of a class, in one block.

9.9 Declaration of fields in interfaces

All fields in interfaces must be declared without any access modifiers like `public` or `static`.

9.10 No debug or error messages to System.out or System.err

Writing warning or error messages to `System.out` or `System.err` is forbidden. Whenever errors occur, the exception system must be used, exceptions can carry textual messages in them.

9.11 Opening braces

Always give opening braces a line of their own:

```
while (i > 12)
{
    System.out.println("i is now " + i);
    i--;
}
```

Do not write:

```
while (i > 12) {
    System.out.println("i is now " + i);
    i--;
}
```

9.12 One line statement blocks

In `if` statements and `while` and `for` loops, always include single line statements in braces:

```
if (i > 0)
{
    i++;
}
```

Do not write:

```
if (i > 0)
    i++;
```

Also do not write:

```
if (i > 0) i++;
```

9.13 Conditional operator

Avoid the ternary conditional operator `?:`, use an if statement instead:

```
if (i >= 0 && i < array.length)
{
    return array[i];
}
else
{
    return "Not a valid index: " + i;
}
```

Do not write:

```
return (i >= 0 && i < ARRAY.length) ? ARRAY[i] : "?";
```

An exception to this can be made when the argument to a constructor must be picked from two alternatives within a constructor (`this` or `super`). Example:

```
public class MyFrame extends Frame
{
    public MyFrame(String title)
    {
        super(title == null ? "Untitled" : title, true);
        ...
    }
}
```

Bibliography

[jiu00] Java Imaging Utilities homepage, http://schmidt.devlib.org/jiu/.

[sou00] SourceForge homepage, http://www.sourceforge.net/.