I express thanks and gratitude to Mr. …………………………………………H.O.D computer science department, ……………………………………………………College for his encouraging support and guidance in carrying out the project.

I would like to express gratitude and indebtedness to Mr……………………………………, for his valuable advice and guidance without which this project would not have seen the light of the day.

I thank Mr………………………………… , project guide , for his insistence on good programming technique which helped us to design and develop a successful model of an Image Processor.

Name
CONTENTS

• INTRODUCTION 3
  • PURPOSE OF THE PROJECT 3
  • PROBLEM IN EXISTING SYSTEM 3
  • SOLUTION OF THESE PROBLEMS 3
  • SCOPE OF THE PROJECT 7
  • ORGANIZATION PROFILE 8
  • HARDWARE & SOFTWARE SPECIFICATIONS 10

• PROJECT ANALYSIS 11
  • STUDY OF THE SYSTEM 11
  • INPUT & OUTPUT 11
  • PROCESS MODELS USED WITH JUSTIFICATION 11

• PROJECT DESIGN 12
  • DATA FLOW DIAGRAMS 13

• OUTPUT SCREENS 15

• PROJECT CODING 28
  • CODE EXPLANATION 28

• OOPS AND JAVA 57

• PROJECT TESTING 66
  • COMPILING TEST 66
  • EXECUTION TEST 66
  • OUTPUT TEST 66

• CONCLUSION 67

• BIBLIOGRAPHY 68
INTRODUCTION

PURPOSE OF THE PROJECT

Image Processing comes with a GUI (graphical user interface) program that lets us 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.

PROBLEM IN EXISTING SYSTEM

Available systems are for any one Operating System

Available systems supports only some Formats of images and uses color settings from system.

Available systems are only for stand alone systems

SOLUTION OF THESE PROBLEMS

To solve the inconveniences as mentioned above java is being proposed. The goal is to have readable, easily extendable code.

Even though 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 that a particular language is to be used, the library in C, C++, Haskell, Ada, Delphi, whatever could be implemented.
The main reasons for picking up the java are:

- **Cross-platform.**
  Java and its byte code concept lead to true cross-platform development—no more if defs 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.
SCOPE OF THE PROJECT

As there is lot of crazy towards graphics, photos or images editing, processing is very important in that process. This application is accessible on any system and any platform. This application can be used to change the setting of old photos, creating a new type of icons to add them into applications and web sites. This application can be used to change the format of images from one file type to another in simple way so that users or programmers can use one image in any format according to compatibility and accessibility in their applications.
ORGANIZATION PROFILE

GALAXY SOFTWARE SOLUTIONS

Galaxy Software Solutions (GSS) is an IT Solution Provider for a dynamic environment where business and technology strategies converge. Our approach focuses on new ways of business combining IT innovation and adoption while also leveraging an organization's current IT assets. We work with large global corporations and new generation technology companies - to build new products or services and to implement prudent business and technology strategies in today's environment.

Galaxy's range of expertise includes:

- Software Development Services
- Engineering Services
- Systems Integration
- Customer Relationship Management
- Supply Chain Management
- Product Development
- Electronic Commerce
- Consulting
- IT Outsourcing

We apply technology with innovation and responsibility to achieve two broad objectives:

- Effectively address the business issues our customers face today
- Generate new opportunities that will help them stay ahead in the future

This approach rests on:

- A strategy where we Architect, Integrate and Manage technology services and solutions — we call it AIM for success.
- A robust offshore development methodology and reduced demand on customer resources
- A focus on the use of reusable frameworks to provide cost and time benefits
We combine the best people, processes and technology to achieve excellent results consistently. We offer customers the advantages of:

**Speed:** We understand the importance of timing, of getting there before the competition. A rich portfolio of reusable, modular frameworks helps jump start projects. Tried and tested methodology ensures that we follow a predictable, low-risk path to achieve results. Our track record is testimony to complex projects delivered within and even before schedule.

**Expertise:** Our teams combine cutting edge technology skills with rich domain expertise. What's equally important — we share a strong customer orientation that means we actually start by listening to the customer. We're focused on coming up with solutions that serve customer requirements today and anticipate future needs.

**A Full Service Portfolio:** We offer customers the advantage of being able to Architect, Integrate and Manage technology services. This means that they can rely on one, fully accountable source instead of trying to integrate disparate multi-vendor solutions.

**Services:** GSS is providing its services to Sain medicaments Pvt. Ltd, Grace drugs and pharmaceuticals Private Limited Alka Drugs and Pharmaceuticals Pvt Ltd to name just a few with out rich experience and expertise in Information Technology we are in the best position to provide software solutions to distinct business requirements.
5) HARDWARE & SOFTWARE SPECIFICATIONS

Environment : Java Runtime Environment 1.1 and above
Operating System : Any with compatible JVM
Hard disk : 4 GB
Processor : PII or higher
PROJECT ANALYSIS

STUDY OF THE SYSTEM

This application can be mainly divided into two modules:

1. User Interface
2. Image Processing

INPUT AND OUTPUT:
User has to load an Image as an input to the application and implement the different options of processing on loaded image then application processes and displays the result of the process on given GUI application.

PROCESS MODEL USED WITH JUSTIFICATION

The model used here is a SPIRAL MODEL. This Model demands a direct consideration of technical risk at all stages of the project and if properly applied it reduces risk before they become problematic, hence it becomes easier to handle a project when using this kind of model where in the end user can evaluate the program at the end of each stage and suggest modification if required.
PROJECT DESIGN

1) DATA FLOW DIAGRAM

0\textsuperscript{th} Level

\begin{center}
\begin{tikzpicture}
  \node [draw, rounded rectangle] (user) {User};
  \node [draw, rounded rectangle, right of=user] (processor) {Image Processor};
  \node [draw, rounded rectangle, right of=processor] (render) {Render};
  \draw [->] (user) -- (processor);
  \draw [->] (processor) -- (render);
\end{tikzpicture}
\end{center}

1\textsuperscript{st} Level

\begin{center}
\begin{tikzpicture}
  \node [draw, rounded rectangle] (user) {User};
  \node [draw, rounded rectangle, right of=user] (load) {Image Load};
  \node [draw, rounded rectangle, right of=load] (process) {Image Process};
  \node [draw, rounded rectangle, right of=process] (render) {Render};
  \draw [->] (user) -- (load);
  \draw [->] (load) -- (process);
  \draw [->] (process) -- (render);
\end{tikzpicture}
\end{center}

2\textsuperscript{nd} Level
OUTPUT SCREENS

Starting of Application
Help:
Project Report

IMAGE PROCESSOR
Project Report

IMAGE PROCESSOR
PROJECT CODING

1. CODE EXPLANATION
Image data types:

Interfaces:
The package “jog.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.

Classes:
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:

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

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

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 jog.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.

**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 jog.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 jog.jiu.codecs.
Project Report

It will try all codecs that are part of JIU:

```java
import jog.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 Image To Image Operation, 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.

Creation and usage

Running operations always follows the same pattern:

- Create an object of the operation’s class.

SAVING IMAGES TO FILES

- 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 jog.jiu.color.adjustment.

```
import jog.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 jog.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.

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.

Accessing image data with the AWT classes
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 Java1.0 – 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.

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 jog.jiu.data—it’s relatively little and, hopefully, readable.

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.
**IntegerImage**

Derived from PixelImage is IntegerImage. It only adds the restriction that each sample must be an integer value between 0 and 231 - 1 (also known as Integer.MAXVALUE) 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.

**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 class.
- 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.

**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.

**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.

IMPLEMENTATIONS OF THE IMAGE INTERFACES

- 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.

Implementations of the image interfaces

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.

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).

IMAGE DATA TYPES
Demo program jiuawt, Image Processing 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.

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 jog.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.

DEMO PROGRAM JIUAWT

An overview of built-in classes
This 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

Image data
jog.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.

Operations
jog.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.

Codecs
Codecs are classes to read images from and write them to files (or in some cases, more generally, streams). The package jog.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.

Color
jog.jiu.color o_ers operations that modify or analyze the color of an image.

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.
AN OVERVIEW OF BUILT-IN CLASSES

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.

ErrorDiusionDithering (?) 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).

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.

Filters
The jog.jiu.filters package has support for convolution kernel filters and a few non-linear filters.

Transformations
The jog.jiu.transform package provides common transformation operations, including scaling, rotating, shearing, cropping, flipping and mirroring.

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.

**Color quantization**

jog.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.

**Applications**

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

- GUI - AWT 19
- GUI - AWT

The jog.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.

- **GUI - AWT dialogs**

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

- **Utility class**

jog.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.
AN OVERVIEW OF BUILT-IN CLASSES

Writing operations
Basics
The base class for all classes performing analysis, modification and serialization of images or image-related data is jog.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 first. Use some package from the jog.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.
• ImageToImageOperation – Any operation that takes one or more input images and produces one or more output images.
• 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.
Using Image To Image Operation

As mentioned before, Image To Image Operation takes one or more input images and produces one or more output images.

Exceptions

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

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 `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`
- `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.
Writing image codecs

Introduction
The package jog.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, jog.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.

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 must be used to maintain overall readability of the source code.

Basics
If the codec will be part of JIU, it must be put into the package jog.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:

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

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:

```java
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.

**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:

```java
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:

```java
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.

**Supported actions**

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

**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).

THE PROCESS METHOD
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.

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.

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.

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.
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 more complex primitive values 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 BufferedWriter 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.
**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 jog.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:

```java
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.

**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:

```java
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).
```

**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.

**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 preferred to speed, but higher speed can sometimes be reached by very simple means.

- The caller should use Buffered Input Stream and Buffered Output Stream 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.

**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.

Coding conventions

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.

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.

Indentation

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

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

**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 8.5). 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 su_xes like Interface or Impl to express that a class is an interface or an implementation of some interface.

**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 8.4. That is why the use of the underscore is allowed and encouraged in the names of final variables to separate words: MAX VALUE.

**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.

**Thrown exceptions in method signatures**

Do not explicitly specify exceptions that extend RuntimeException, e.g. IllegalArgumentException.
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.

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

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

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--;
}

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++;  

8.13 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);  
...  
}  
}
The application is developed using AWT (Abstract Window Toolkit).
The `java.awt` package is much useful for creating user interfaces and for painting graphics and images. A user interface object such as a button or a scrollbar is called, in AWT terminology, a component. The Component class is the root of all AWT components. Some components fire events when a user interacts with the components. A container is a component that can obtain components and other containers. A container can also have a layout manager that controls the visual placement of components in the container.

The `java.awt` package implements different interfaces like `LayoutManager`, which defines the interface for classes that know how to layout Containers.

`Paint` interface defines how color patterns can be generated for Graphics2D operations. A class implementing the Paint interface is added to the Graphics2D context in order to define the color pattern used by the draw and fill methods.

**OBJECT ORIENTED PROGRAMMING AND JAVA**

Object-oriented Programming was developed because of limitations found in earlier approaches of programming. To appreciate what OOP does, we need to understand what these limitations are and how they arose from traditional programming.

**PROCEDURAL LANGUAGES**

Pascal, C, Basic, FORTRAN, and similar languages are procedural languages. That is, each statement in the language tells the computer to do something: Get some input, add these numbers, divide by 6, display the output. A program in a procedural language is a list of instructions.

For very small programs no other organizing principle (often called a paradigm) is needed. The programmer creates the list of instructions, and the computer carries them out.
Division into Functions

When programs become larger, a single list of instructions becomes unwieldy. Few programmers can comprehend a program of more than a few hundred statements unless it is broken down into smaller units. For this reason the function was adopted as a way to make programs more comprehensible to their human creators. (The term functions is used in C++ and C. In other languages the same concept may be referred to as a subroutine, a subprogram, or a procedure.) A program is divided into functions, and (ideally, at least) each function has a clearly defined purpose and a clearly defined interface to the other functions in the program.

The idea of breaking a program into functions can be further extended by grouping a number of functions together into a larger entity called a module, but the principle is similar: grouping a number of components that carry out specific tasks.

Dividing a program into functions and modules is one of the cornerstones of structured programming, the somewhat loosely defined discipline that has influenced programming organization for more than a decade.

Problems with Structured Programming

As programs grow ever larger and more complex, even the structured programming approach begins to show signs of strain. You may have heard about, or been involved in, horror stories of program development. The project is too complex, the schedule slips, more programmers are added, complexity increases, costs skyrocket, the schedule slips further, and disaster ensues. Analyzing the reasons for these failures reveals that there are weaknesses in the procedural paradigm itself. No matter how well the structured programming approach is implemented, large programs become excessively complex.

What are the reasons for this failure of procedural languages? One of the most crucial is the role played by data.

Data Undervalued

52
In a procedural language, the emphasis is on doing things--read the keyboard, invert the vector, check for errors, and so on. The subdivision of a program into functions continues this emphasis. Functions do things just as single program statements do. What they do may be more complex or abstract, but the emphasis is still on the action.

What happens to the data in this paradigm? Data is, after all, the reason for a program's existence. The important part of an inventory program isn't a function that displays the data, or a function that checks for correct input; it's the inventory data itself. Yet data is given second-class status in the organization of procedural languages.

For example, in an inventory program, the data that makes up the inventory is probably read from a disk file into memory, where it is treated as a global variable. By global we mean that the variables that constitute the data are declared outside of any function, so they are accessible to all functions. These functions perform various operations on the data. They read it, analyze it, update it, rearrange it, display it, write it back to the disk, and so on.

We should note that most languages, such as Pascal and C, also support local variables, which are hidden within a single function. But local variables are not useful for important data that must be accessed by many different functions.

Now suppose a new programmer is hired to write a function to analyze this inventory data in a certain way. Unfamiliar with the subtleties of the program, the programmer creates a function that accidentally corrupts the data. This is easy to do, because every function has complete access to the data. It's like leaving your personal papers in the lobby of your apartment building: Anyone can change or destroy them. In the same way, global data can be corrupted by functions that have no business changing it.

Another problem is that, since many functions access the same data, the way the data is stored becomes critical. The arrangement of the data can't be changed without modifying all the functions that access it. If you add new data items, for example, you'll need to modify all the functions that access the data so that they can also access these new items. It will be hard to find all such functions, and even harder to modify all of them correctly. It's similar to what happens when your local supermarket moves the bread from aisle 4 to
aisle 12. Everyone who patronizes the supermarket must figure out where the bread has gone, and adjust their shopping habits accordingly.

What is needed is a way to restrict access to the data, to hide it from all but a few critical functions. This will protect the data, simplify maintenance, and offer other benefits as well.

**Relationship to the Real World**

Procedural programs are often difficult to design. The problem is that their chief components--functions and data structures--don't model the real world very well. For example, suppose you are writing a program to create the elements of a graphics user interface: menus, windows, and so on. Quick now, what functions will you need? What data structures? The answers are not obvious, to say the least. It would be better if windows and menus corresponded more closely to actual program elements.

**New Data Types**

There are other problems with traditional languages. One is the difficulty of creating new data types. Computer languages typically have several built-in data types: integers, floating-point numbers, characters, and so on. What if you want to invent your own data type? Perhaps you want to work with complex numbers, or two dimensional coordinates, or dates—quantities the built-in data types don’t handle easily. Being able to create your own types is called extensibility; you can extend the capabilities of the language. Traditional languages are not usually extensible. Without unnatural convolutions, you can’t bundle together both X and Y coordinates into a single variable called Point, and then add and subtract values of this type. The result is that traditional programs are more complex to write and maintain.

**The object oriented approach**

The fundamental idea behind object-oriented languages is to combine into a single unit both data and the functions that operate on that data. Such a unit is called an object.
An object’s functions, called member methods in Java, typically provide the only way to access its data. If you want to read the item and return the value to you, you call a member function in the object. It will read the item and return the value to you. You can’t access the data directly. The data is hidden, so it is safe from accidental modification. Data and its functions are said to be encapsulated into a single entity. Data encapsulation and data hiding are key terms in the description of object-oriented languages.

If you want to modify the data in an object, you know exactly what functions interact with it: the member functions in the object. No other functions can access the data. This simplifies writing, debugging, and maintaining the program.

A Java program typically consists of a number of objects, which communicate with each other by calling one another’s members functions. We should mention that what are called member functions in C++ are called methods in Java. Also, data items are referred to as instance variables. Calling an object’s member function is referred to as sending a message to the object.

An analogy

You might want to think of objects as departments—such as sales, accounting, personnel, and so on—in a company. Departments provide an important approach to corporate organization. In most companies (except very small ones), people don’t work on personnel problems one day, the payroll the next, and then go out in the field as sales people the week after. Each department has its own personnel, with clearly assigned duties. It also has its own data: payroll, sales figures, personnel records, inventory, or whatever, depending on the department.

The people in each department control and operate on those departments data. Dividing the company into departments makes it easier to comprehend and control the company’s activities, and helps them maintain the integrity of the information used by the company. The payroll department, for instance, is responsible for the payroll data. If you are from the sales department, and you need to know the total of all the salaries paid in the
southern region in July, you don’t just walk into the payroll department and start rummaging through file cabinets. You send a memo to the appropriate person in the department, and then you wait for that person to access the appropriate person in the department, and then you wait for that person to access the data and send you a reply with the information you want. This ensures that the data is accessed accurately and that it is not corrupted by inept outsiders. (This view of corporate organization is shown in figure). In the same way, objects provide an approach to program organization, while helping to maintain the integrity of the program's data.

**OOP: An approach to organization**

Keep in mind that object-oriented programming is not primarily concerned with the details of program operation. Instead, it deals with the overall organization of the program.

**Characteristics of object-oriented languages:**

Let’s briefly examine a few of the major elements of object-oriented languages in general and Java in particular.

**Objects**

When you approach a programming problem in an object-oriented language, you no longer ask how the problem will be divided into functions, but how it will be divided into objects. Thinking in terms of objects, rather than functions, has a surprisingly helpful effect on how easily programs can be designed and objects in the real world.

What kinds of things become object-oriented programs? The answer to this is limited only by your imagination, but there are some typical categories to start you thinking:

**Physical objects**

Automobile in a traffic-flow simulation
Electrical components in a circuit design to a program
Countries in an economics model
Aircraft in an air-traffic control system

- Elements of the computer-user environment
  - Windows
  - Menus
  - Graphics objects (lines, rectangles, circles)
  - The mouse and the keyboard

- Programming constructs
  - Customized arrays
  - Stacks
  - Linked lists

- Collection of data
  - An inventory
  - A personnel file
  - A dictionary

A table of the latitudes and longitudes of world cities

- User defined data types
  - Time
  - Angles
  - Complex numbers
  - Points on the plane

- Components in a computer games

Ghosts in maze game
Positions in a board game (chess, checkers)
Animals in an ecological simulation
Opponents and friends in adventure games
The match between programming objects and real-world objects us the happy result of combining data and functions: the resulting objects offer a revolution in program designing, no such close match between programming constructs and the items being modelled exists in a procedural language.

**Classes**

In OOP we say that objects are members of classes. What does this mean? Let’s look at an analogy. Almost all computer languages have built-in data types. For instance, a data type int, meaning integer is pre-defined in Java. You can declare as many variables of type int as you need in your program:

```
Int day;
Int count;
Int divisor;
Int answer;
```

A class serves as a plan, or template. It specifies what data, and what functions will be included in objects of that class. Defining the class doesn’t create any objects, just as the mere existence of a type int doesn’t create any variables.

A class is thus a collection of similar objects. This fits our non technical understanding of the word class, Prince, sting etc., are members of the class of rock musicians. There is no person called rock musician but specific people with specific names are members of this class if they possess certain characteristics.

**Abstraction**

An essential element of object-oriented programming is abstraction. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead they are free to utilize the object as a whole.
A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units. For instance, the sound system consists of a radio, a CD player, and/or a tape player. The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

Hierarchical abstractions of complex systems can also be applied to computer programs. The data from a traditional process-oriented program can be transformed by abstraction into its component objects. A sequence of process steps can become a collection of messages between these objects. Thus, each of each object describes its own unique behavior. You can treat these objects as concrete entities that respond to messages telling them to do something. This is the essence of object-oriented programming.

Object-oriented concepts form the heart of Java just as they form the basis for human understanding. It is important that you understand how these concepts translate into programs. As you will see, object-oriented programming is a powerful and natural paradigm for creating programs that survive the inevitable changes accompanying the life cycle of any major software project, including conception, growth, and aging. For example, once you have a well-defined objects and clean, reliable interfaces to those objects, you can gracefully decommission or replace parts of an older system without fear.

**Encapsulation**

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you
are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever. You can’t affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver’s point of view, they all work the same. This same idea can be applied to programming. The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

In Java the basis of encapsulation is the class. Although the class will be examined in great detail later in this book, the following brief discussion will be helpful now. A class defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as instances of a class. Thus, a class is a logical construct; an object has physical reality.

When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called members of the class. Specifically, the data defined by the class are referred to as member variables or instance variables. The code that operates on that data is referred to as member methods or just methods.

Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The public interface of a class represents everything that external users of the class need to know, or may know. The private methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the
class’ public methods, you can ensure that no improper actions take place. Of course, this means that the public interface should be carefully designed not to expose too much of the inner workings of a class.

Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification dog, which in turn is part of the mammal class, which is under the larger class animal. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

Most people naturally view the world as made up of objects that are related to each other in a hierarchical way, such as animals, mammals, and dogs. If you wanted to describe animals in an abstract way, you would say they have some attributes, such as size, intelligence, and type of skeletal system. Animals also have certain behavioral aspects; they eat, breathe, and sleep. This description of attributes and behavior is the class definition for animals.

If you wanted to describe a more specific class of animals, such as mammals, they would have more specific attributes, such as type of teeth, and mammary glands. This is known as a subclass of animals, where animals are referred to as mammals’ super class.

Since mammals are simply more precisely specified animals, they inherit all of the attributes from animals. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the class hierarchy.

Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes plus any that it adds as part of its specialization (see Figure 2-2). This is a key concept which lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits
all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.

Polymorphism
Polymorphism (from the Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stack. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non-object-oriented language, you would be required to create three difference sets of stack routines, with each set using different names. However, because of polymorphism, in Java you can specify a general set of stack routines that all share the same names.

More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a general class of action. It is the compiler’s job to select the specific action (that is, method) as it applies to each situation. You, the programmer, do not need to make this selection manually. You need only remember and utilize the general interface.

Extending the dog analogy, a dog’s sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog’s nose! This same general concept can be implemented in Java as it applies to methods within a Java program.

Polymorphism, Encapsulation, and Inheritance Work Together
When properly applied, polymorphism, encapsulation, and inheritance combine to produce a programming environment that supports the development of far more robust and scaleable programs than does the process-oriented model. A well-designed hierarchy
of classes is the basis for reusing the code in which you have invested time and effort
developing and testing. Encapsulation allows you to migrate your implementations over
time without breaking the code that depends on the public interface of your classes.
Polymorphism allows you to create clean, sensible, readable, and resilient code.

Of the two real-world examples, the automobile more completely illustrates the power of
object-oriented design. Dogs are fun to think about from an inheritance standpoint, but
cars are more like programs. All drivers rely on inheritance to drive different types
(subclasses) of vehicles. Whether the vehicle is a school bus, a Mercedes
sedan, a Porsche, or the family minivan, drivers can all more or less find and operate the
steering wheel, the brakes, and the accelerator. After a bit of gear grinding, most people
can even manage the difference between a stick shift and an automatic, because they
fundamentally understand their common super class, the transmission.

People interface with encapsulated features on cars all the time. The brake and gas pedals
hide an incredible array of complexity with an interface so simple you can operate them
with your feet! The implementation of the engine, the style of brakes, and the size of the
tires have no effect on how you interface with the class definition of the pedals.
The final attribute, polymorphism, is clearly reflected in the ability of car manufacturers
to offer a wide array of options on basically the same vehicle. For example, you can get
an antilock braking system or traditional brakes, power or rack-and-pinion steering, 4-, or
6-, or 8-cylinder engines. Either way, you will still press the break pedal to stop, turn the
steering wheel to change direction, and press the accelerator when you want to move.
PROJECT TESTING

COMPILING TEST

It was a good idea to do our stress testing early on, because it gave us time to fix some of the unexpected deadlocks and stability problems that only occurred when components were exposed to very high transaction volumes.

EXECUTION TEST

This program was successfully loaded and executed. Because of good programming there were no execution error.

OUTPUT TEST

The successful output screens are placed in the output screens section above.
CONCLUSION

Even though this application has been developed with the user,....

- Useful things can be done with Image Processing already, but it is not to be considered stable Major changes may be done on the API structure, resulting in possibly tedious modifications necessary for all those who rely on it.
- If the functionality you need is already in the runtime library. So if all you want to do is create thumbnails from images, you can get along fine without JIU.
- This project comes with its own set of image types which are incompatible with AWT and Swing. While there are conversion methods between JIU and AWT image types, it is somewhat inconvenient. If your application depends heavily on AWT image types, JIU may not be the right choice.
BIBLIOGRAPHY

Java in a Nutshell

An excellent quick reference. I use this book while coding all the time. It also has a nice intro which constrasts Java and C/C++.

Graphic Java

A detailed explanation of many things in the AWT window toolkit including images and animation.

Teach Yourself Java in 21 days

A good book for a structured learning plan for a quick push to get yourself up-to-speed. I learned the basics of Java using this book, and would do it again with no regrets. I wish they had an upgrade plan, because there's a new edition out that is even cooler than the first.

Java by Example

A good book for learning Java programming style and the Java object library.

○ Java 2: The Complete Reference, Fifth Edition by Herbert Schildt, Herb Schildt