# Drawing gophers with Go

A more fun Go introduction for beginners
Quasilyte @ GolangKazan, 2019

# Questions that are asked

- Who uses Go?

# Questions that are asked

- Who uses Go?
- Are there really big examples of Go software?

# Questions that are asked

- Who uses Go?
- Are there really big examples of Go software?
- What tasks are usually solved with Go?

# Questions that are asked

- Who uses Go?
- Are there really big examples of Go software?
- What tasks are usually solved with Go?
- What are the main advantages of using Go?

# Questions that are asked

- Who uses Go?
- Are there really big examples of Go software?
- What tasks are usually solved with Go?
- What are the main advantages of using Go?
- Why Go instead of XYZ language?

# Questions that are asked

- Who uses Go?
- Are there really big examples of Go software?
- **What tasks are usually solved with Go?**
- What are the main advantages of using Go?
- Why Go instead of XYZ language?

# Infrastructure

- Dev tools
- Automation tools
- Build tools, generators
- Monitoring systems
- Specialized databases
- Integration layers
- System utilities

# Backend

- Web (app) servers
- Proxies
- Background workers
- ETL programs
- Microservices

# Other

- ML infrastructure
- Chat bots
- AD tech
- Blockchain
- Embedded*

Of course you can use Go for everything, but some areas lack good libraries.

We’ll do some simple image processing using Go standard library.

# Standard library

- image - basic 2D image library
- image/draw - image composition functions
- image/color - basic color library
- image/color/palette - standard color palettes
- image/{png,jpeg,gif} - encoders/decoders

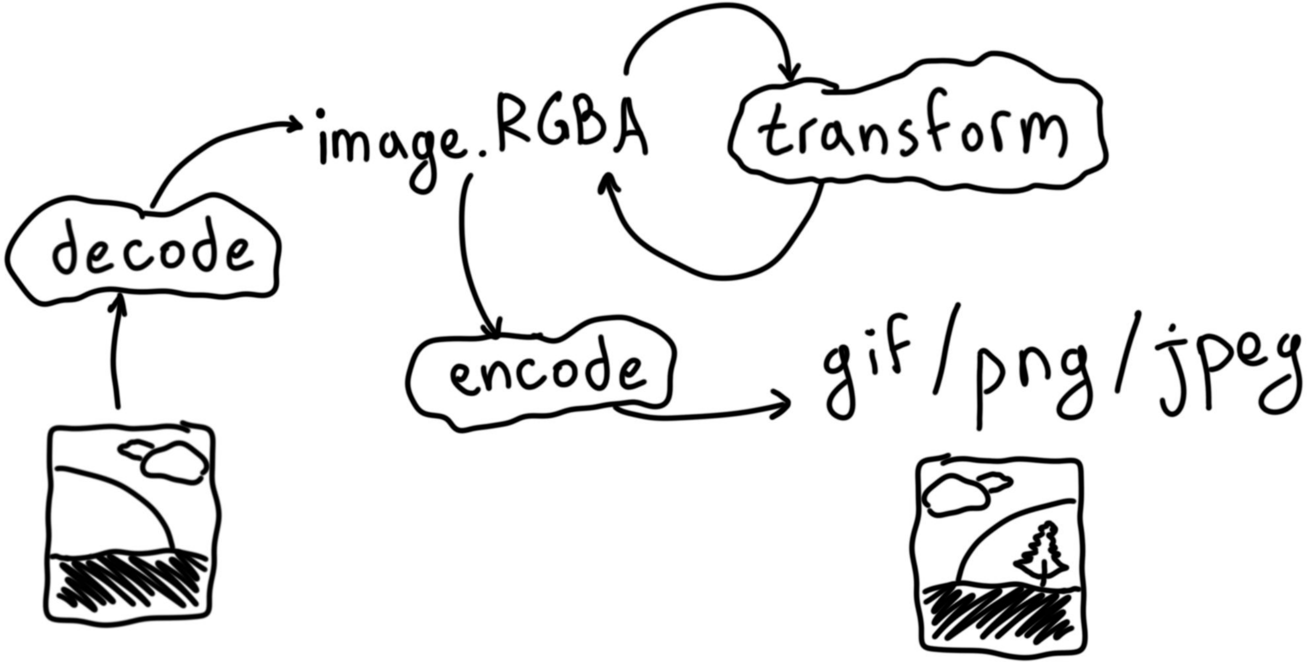


image.RGBA
transform
decode
encode
gif / png / jpeg

# Workflow

1. Get an “image object”
    - Read (decode) an image file
    - Create programmatically

# Workflow

1. Get an “image object”
    - Read (decode) an image file
    - Create programmatically
2. Apply transformations to the object

# Workflow

1. Get an “image object”
   - Read (decode) an image file
   - Create programmatically
2. Apply transformations to the object
3. Write (encode) an object to a file

We’ll with from something simple!

# example.go: snippet 1 (imports)

```go
package main

import (
 "image"       // 2D types and funcs
 "image/color" // To work with colors
 "image/png"   // We’ll save it as PNG
 "os"          // To create a new file
)

func main() { /* see next slide */ }
```

# example.go: snippet 1 (imports)

```
package main

import (
 "image"       // 2D types and funcs
 "image/color" // To work with colors
 "image/png"   // We’ll save it as PNG
 "os"          // To create a new file
)

func main() { /* see next slide */ }
```

# example.go: snippet 1 (imports)

```go
package main

import (
 "image"        // 2D types and funcs
 "image/color"  // To work with colors
 "image/png"    // We’ll save it as PNG
 "os"           // To create a new file
)

func main() { /* see next slide */ }
```

# example.go: snippet 2 (main without error handling)

```go
package main

import ( /* see previous slide */ )

func main() {
 img := image.NewGray(image.Rect(0, 0, 3, 3))
 img.Set(1, 1, color.Gray{Y: 255})
 f, _ := os.Create("art.png")
 png.Encode(f, img)
}
```

# example.go: snippet 2 (main without error handling)

```go
package main

import ( /* see previous slide */ )

func main() {
 img := image.NewGray(image.Rect(0, 0, 3, 3))
 img.Set(1, 1, color.Gray{Y: 255})
 f, _ := os.Create("art.png")
 png.Encode(f, img)
}
```

# example.go: snippet 2 (main without error handling)

```go
package main

import ( /* see previous slide */ )

func main() {
  img := image.NewGray(image.Rect(0, 0, 3, 3))
  img.Set(1, 1, color.Gray{Y: 255})
  f, _ := os.Create("art.png")
  png.Encode(f, img)
}
```

# example.go: snippet 2 (main without error handling)

```go
package main

import ( /* see previous slide */ )

func main() {
 img := image.NewGray(image.Rect(0, 0, 3, 3))
 img.Set(1, 1, color.Gray{Y: 255})
 f, _ := os.Create("art.png")
 png.Encode(f, img)
}
```

# example.go: snippet 2 (main without error handling)

```go
package main

import ( /* see previous slide */ )

func main() {
 img := image.NewGray(image.Rect(0, 0, 3, 3))
 img.Set(1, 1, color.Gray{Y: 255})
 f, _ := os.Create("art.png")
 png.Encode(f, img)
}
```

# example.go: snippet 2 (main without error handling)

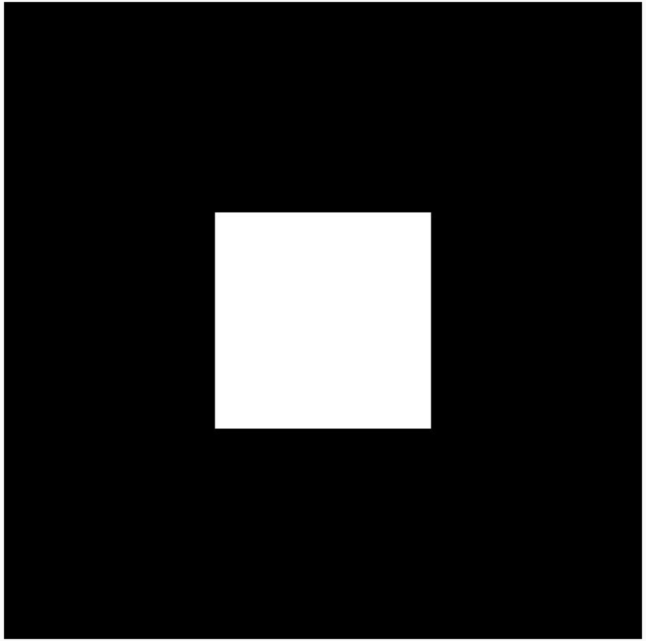
```go
package main

import ( /* see previous slide */ )

func main() {
 img := image.NewGray(image.Rect(0, 0, 3, 3))
 img.Set(1, 1, color.Gray{Y: 255})
 f, _ := os.Create("art.png")
 png.Encode(f, img)
}
```

# Let’s run it!

```
$ go run example.go
```

# It’s so majestic!

art.png →



Not quite a gopher yet, though.

Just a 3x3 PNG image with white pixel in the middle.

Before we continue,
we need to have a serious talk...

# example.go: snippet 3 (main with error handling)

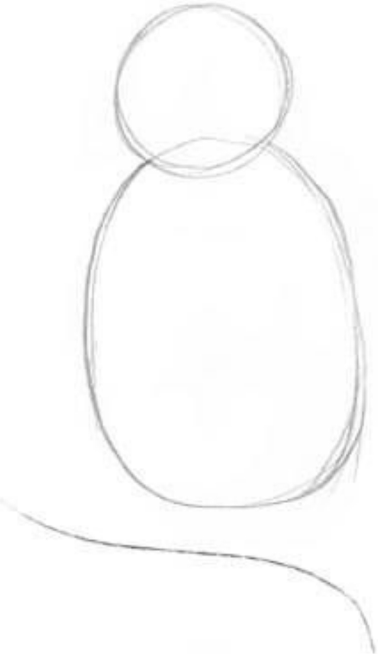
```go
img := image.NewGray(image.Rect(0, 0, 3, 3))
img.Set(1, 1, color.Gray{Y: 255})
f, err := os.Create("art.png")
if err != nil {
  // handle file creation error.
}
err = png.Encode(f, img)
if err != nil {
  // handle image encoding error.
}
```

# example.go: snippet 3 (main with error handling)

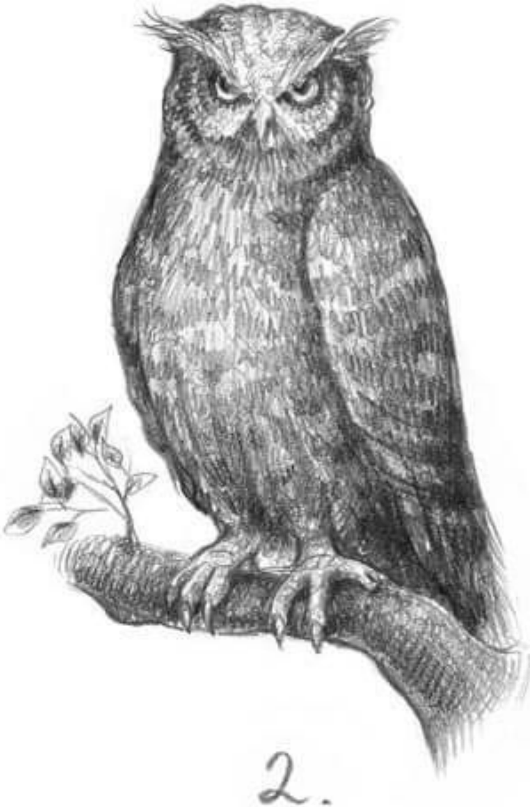
```go
img := image.NewGray(image.Rect(0, 0, 3, 3))
img.Set(1, 1, color.Gray{Y: 255})
f, err := os.Create("art.png")
if err != nil {
  // handle file creation error.
}
err = png.Encode(f, img)
if err != nil {
  // handle image encoding error.
}
```

# example.go: snippet 3 (main with error handling)

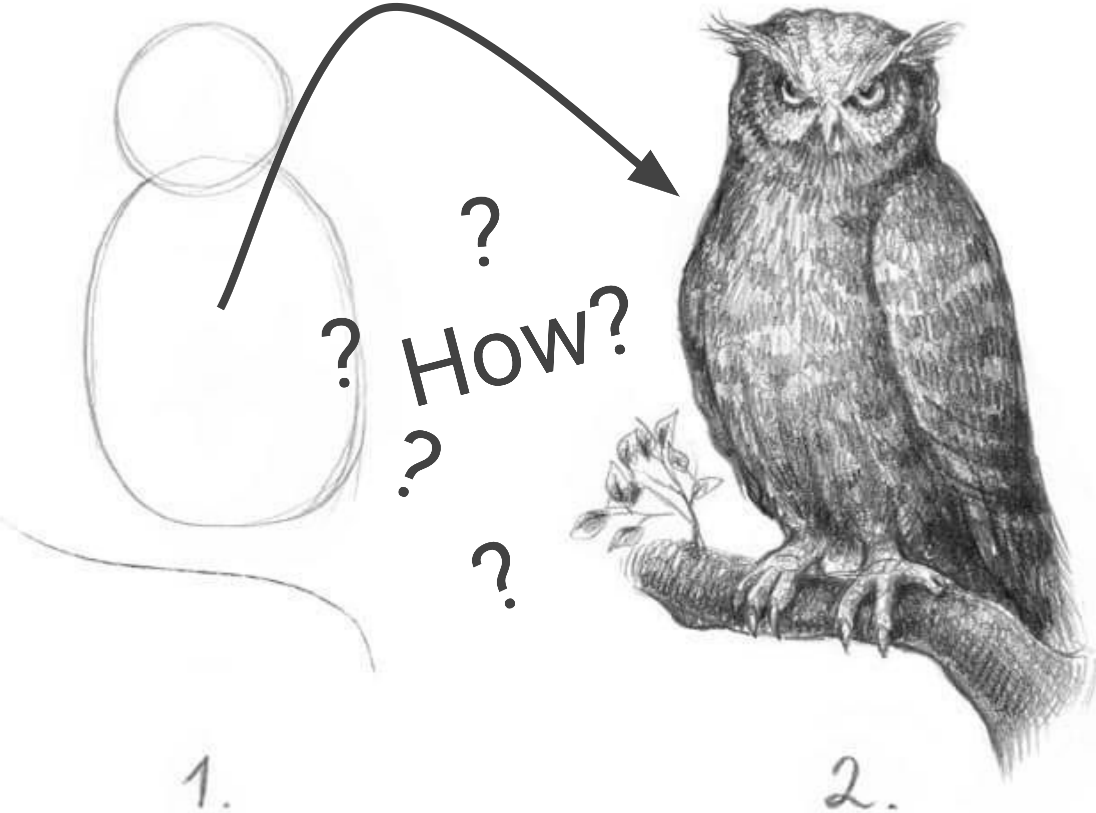
```go
img := image.NewGray(image.Rect(0, 0, 3, 3))
img.Set(1, 1, color.Gray{Y: 255})
f, err := os.Create("art.png")
if err != nil {
  // handle file creation error.
}
err = png.Encode(f, img)
if err != nil {
  // handle image encoding error.
}
```



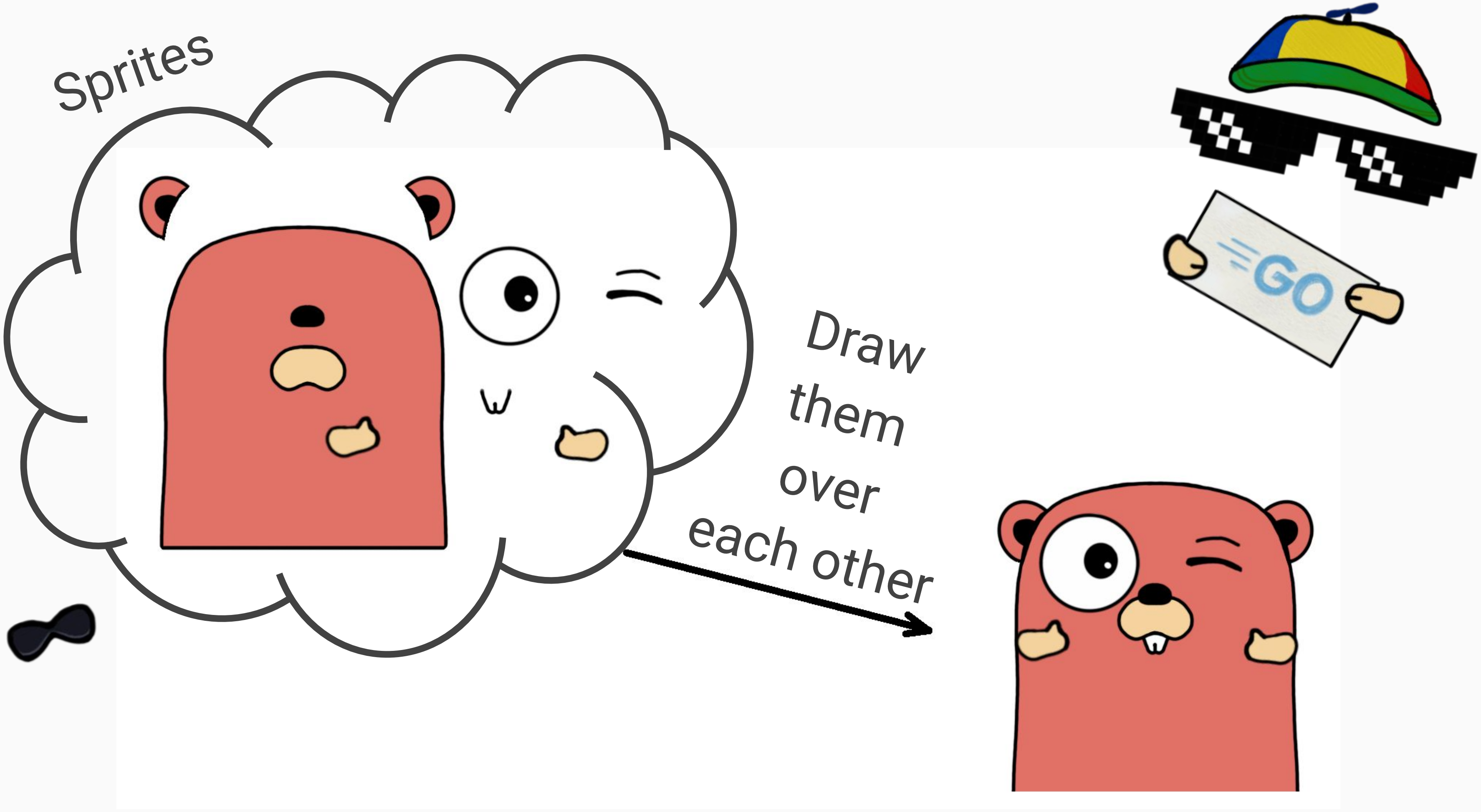
1.



2.


?
?
How?
?
?
1.
2.

We’ll draw a gopher by composing several images together.



Sprites

Draw them over each other

# Where can we get sprites?

Assets can be borrowed from the GopherKon sprites set.

A program that renders sprites together will be called “compose”.

# “Compose” program algorithm

1. Convert filename arguments to image objects.
2. Draw input images over output image.
3. Write output image object to a file.

Our “compose” program will accept filename arguments and write them to a new file, one over another.

# compose.go: snippet 1 (convert filenames)

```go
var layers []image.Image

for _, filename := range filenames {
  f, _ := os.Open(filename)
  defer f.Close()
  img, _ := png.Decode(f)

  layers = append(layers, img)
}
```

# compose.go: snippet 1 (convert filenames)

```go
var layers []image.Image

for _, filename := range filenames {
    f, _ := os.Open(filename)
    defer f.Close()
    img, _ := png.Decode(f)

    layers = append(layers, img)
}
```

# compose.go: snippet 1 (convert filenames)

```
var layers []image.Image

for _, filename := range filenames {
  f, _ := os.Open(filename)
  defer f.Close()
  img, _ := png.Decode(f)

  layers = append(layers, img)
}
```

# compose.go: snippet 1 (convert filenames)

```go
var layers []image.Image

for _, filename := range filenames {
  f, _ := os.Open(filename)
  defer f.Close()
  img, _ := png.Decode(f)

  layers = append(layers, img)
}
```

# Compose: snippet 1 (convert filenames)

```
var layers []image.Image

for _, filename := range filenames {
  f, _ := os.Open(filename)
  defer f.Close()
  img, _ := png.Decode(f)

  layers = append(layers, img)
}
```

# compose.go: snippet 1 (convert filenames)

```go
var layers []image.Image

for _, filename := range filenames {
  f, _ := os.Open(filename)
  defer f.Close()
  img, _ := png.Decode(f)

  layers = append(layers, img)
}
```

# compose.go: snippet 2 (fill output image)

```
bounds := image.Rect(0, 0, *width, *height)
outImage := image.NewRGBA(bounds)

draw.Draw(outImage, bounds,
          layers[0], image.ZP, draw.Src)

for _, layer := range layers[1:] {
  draw.Draw(outImage, bounds,
            layer, image.ZP, draw.Over)
}
```

# compose.go: snippet 2 (fill output image)

```
bounds := image.Rect(0, 0, *width, *height)
outImage := image.NewRGBA(bounds)

draw.Draw(outImage, bounds,
          layers[0], image.ZP, draw.Src)

for _, layer := range layers[1:] {
  draw.Draw(outImage, bounds,
            layer, image.ZP, draw.Over)
}
```

# compose.go: snippet 2 (fill output image)

```go
bounds := image.Rect(0, 0, *width, *height)
outImage := image.NewRGBA(bounds)

draw.Draw(outImage, bounds,
          layers[0], image.ZP, draw.Src)

for _, layer := range layers[1:] {
  draw.Draw(outImage, bounds,
            layer, image.ZP, draw.Over)
}
```

# compose.go: snippet 2 (fill output image)

```
bounds := image.Rect(0, 0, *width, *height)
outImage := image.NewRGBA(bounds)

draw.Draw(outImage, bounds,
          layers[0], image.ZP, draw.Src)

for _, layer := range layers[1:] {
  draw.Draw(outImage, bounds,
            layer, image.ZP, draw.Over)
}
```

# compose.go: snippet 2 (fill output image)

```
bounds := image.Rect(0, 0, *width, *height)
outImage := image.NewRGBA(bounds)

draw.Draw(outImage, bounds,
          layers[0], image.ZP, draw.Src)

for _, layer := range layers[1:] {
  draw.Draw(outImage, bounds,
            layer, image.ZP, draw.Over)
}
```

# compose.go: snippet 2 (fill output image)

```
bounds := image.Rect(0, 0, *width, *height)
outImage := image.NewRGBA(bounds)

draw.Draw(outImage, bounds,
          layers[0], image.ZP, draw.Src)

for _, layer := range layers[1:] {
  draw.Draw(outImage, bounds,
            layer, image.ZP, draw.Over)
}
```

# compose.go: snippet 3 (write to a file)

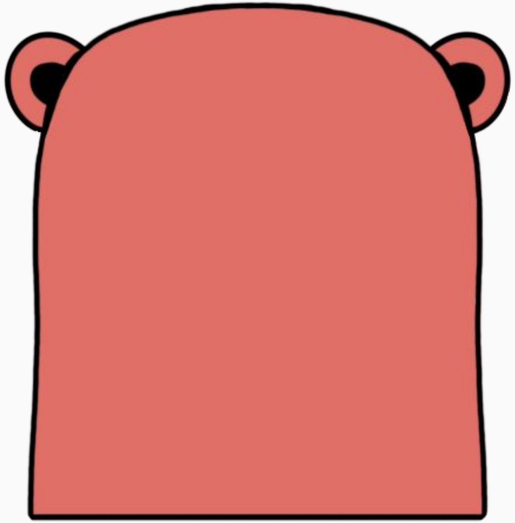
```
outFile, err := os.Create(outFilename)
if err != nil {
  log.Fatalf("create file: %v", err)
}

err = png.Encode(outFile, outImage)
if err != nil {
  log.Fatalf("encode: %v", err)
}
```

# Let’s run it!

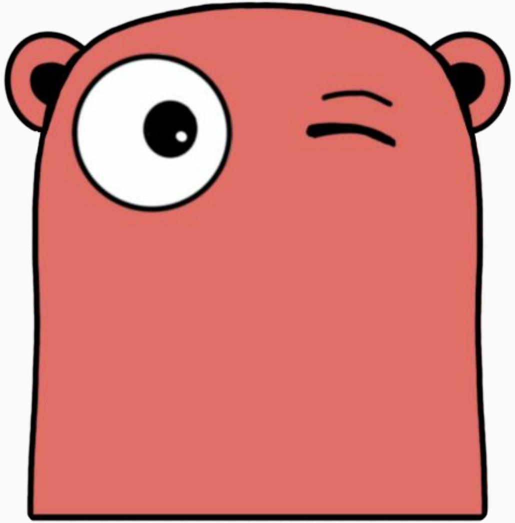
```
$ go run compose.go \
      ears.png body.png eyes.png \
      teeth.png undernose.png \
      nose.png hands.png
```

# Write ears.png

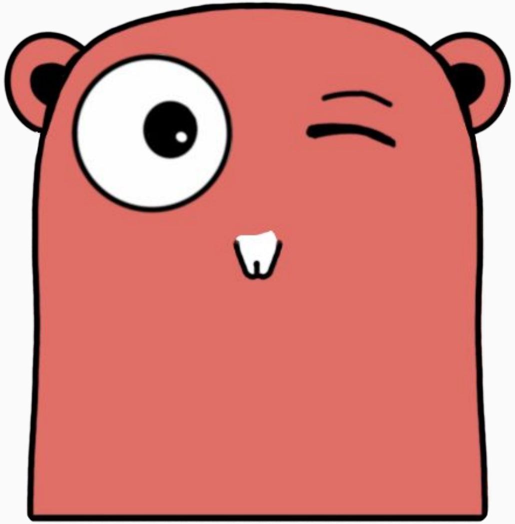
# Write body.png



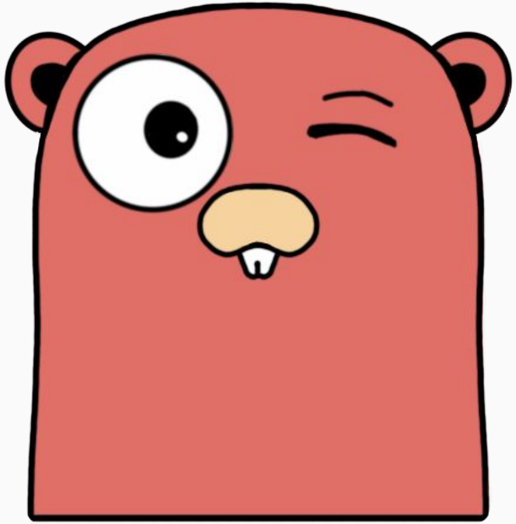
# Write eyes.png



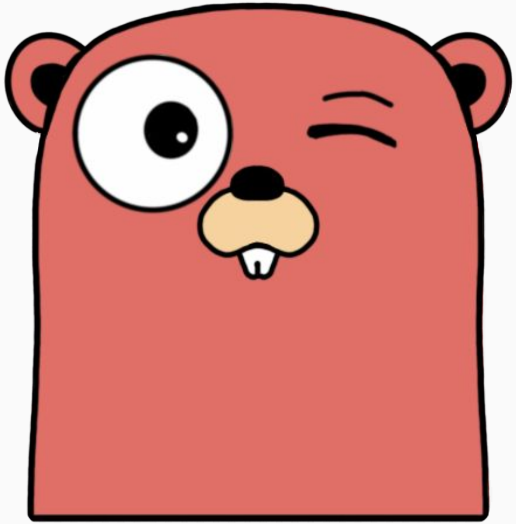
# Write teeth.png



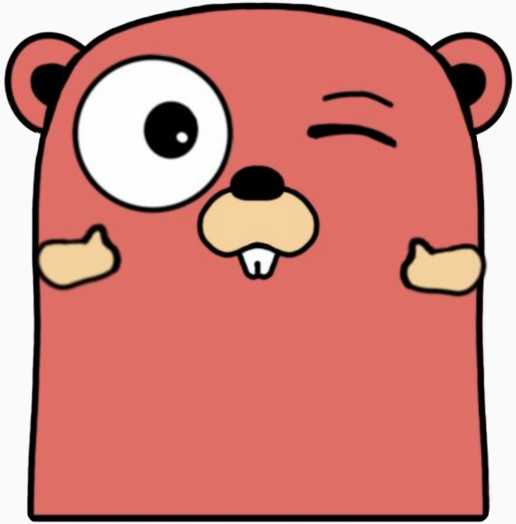
# Write undernose.png



# Write nose.png



# Write hands.png



# gopher.png is ready!

Any part can be changed to get an unique gopher, but the order of drawing is very important.

Complete example with assets:

**https://bit.ly/2lQuSHK**

# Can we convert images from one format to another?

Can we convert images from one format to another?

We can!

# Converting images

1. Decode an image in X format (png/jpeg/etc).
2. Encode that image in Y format (png/jpeg/etc).

We’ll create png2jpg program that converts PNG images to JPEG images with specified quality.

# png2jpg.go: encoding into JPEG

```
img, err := png.Decode()
if err != nil {
  log.Panicf("can’t  decode input PNG")
}

opts := &jpeg.Options{Quality: *quality}
jpeg.Encode(outFile, img, opts)
```

# png2jpg.go: encoding into JPEG

```
img, err := png.Decode()
if err != nil {
  log.Panicf("can’t  decode input PNG")
}

opts := &jpeg.Options{Quality: *quality}
jpeg.Encode(outFile, img, opts)
```

# Let’s run it!

```
$ go run png2jpg.go gopher.png
```

# png2jpg.go: encoding into JPEG

```
img, err := png.Decode()
if err != nil {
  log.Panicf("can’t  decode input PNG")
}

opts := &jpeg.Options{Quality: *quality}
jpeg.Encode(outFile, img, opts)
```

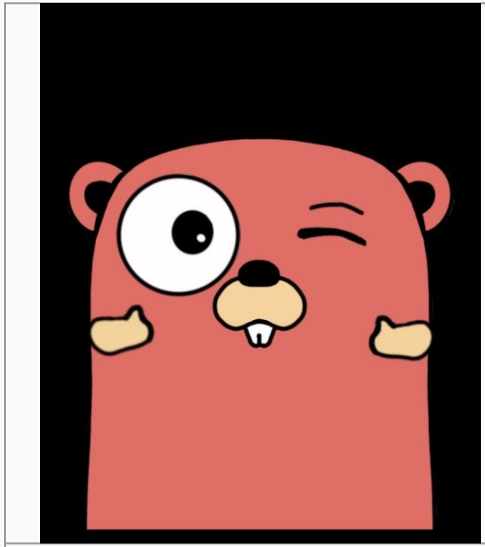
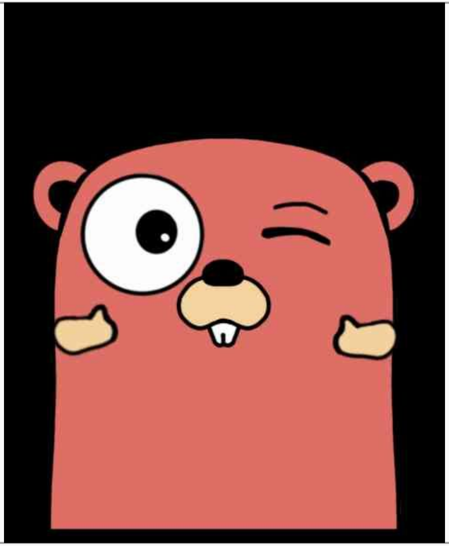
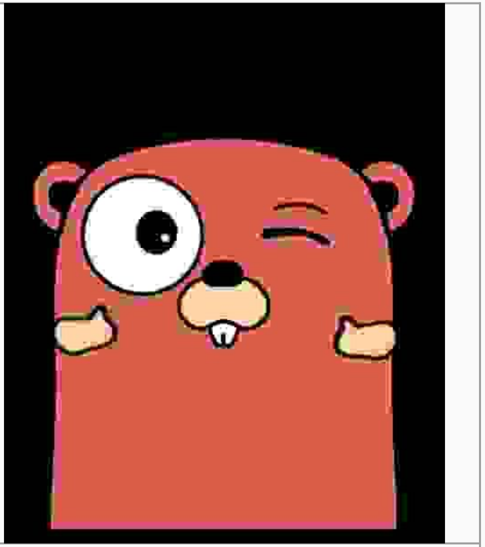
# png2jpg.go: encoding into JPEG

```go
quality := flag.Int(
  "q", 80, "output JPEG quality")
flag.Parse()

// ^ several lines above

opts := &jpeg.Options{Quality: *quality}
jpeg.Encode(outFile, img, opts)
```

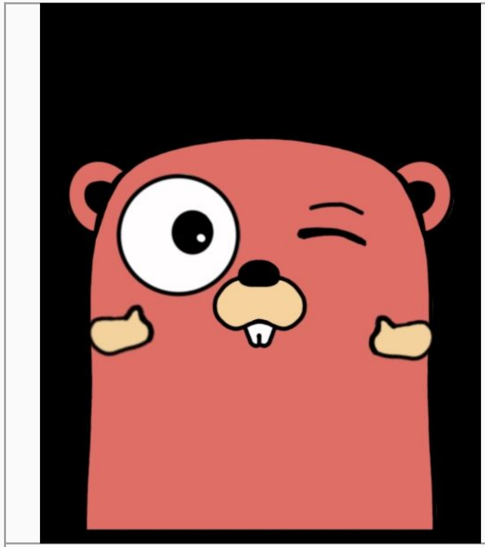
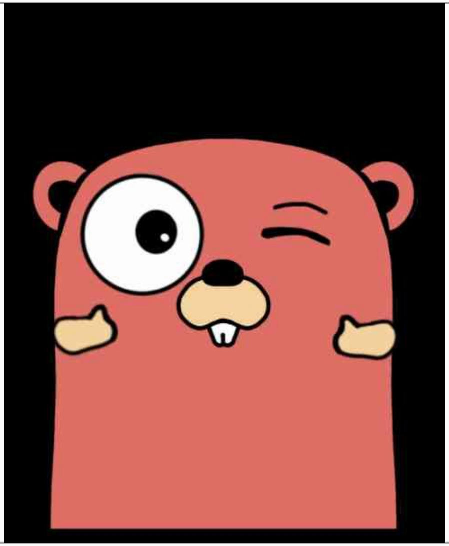
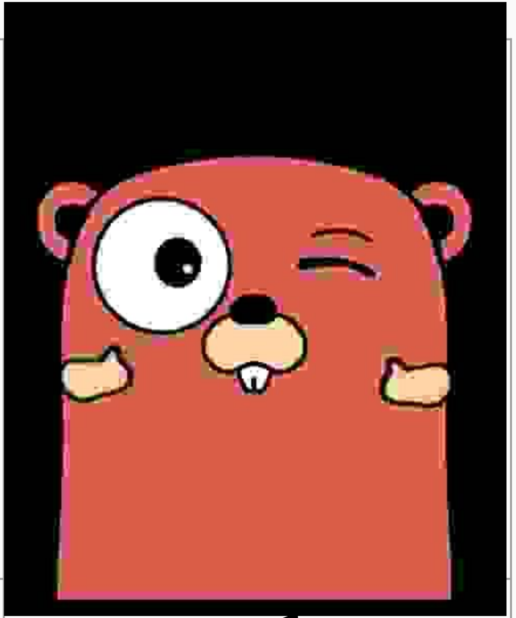
# Let’s run it!

```
$ go run png2jpg.go -q 1 gopher.png
```

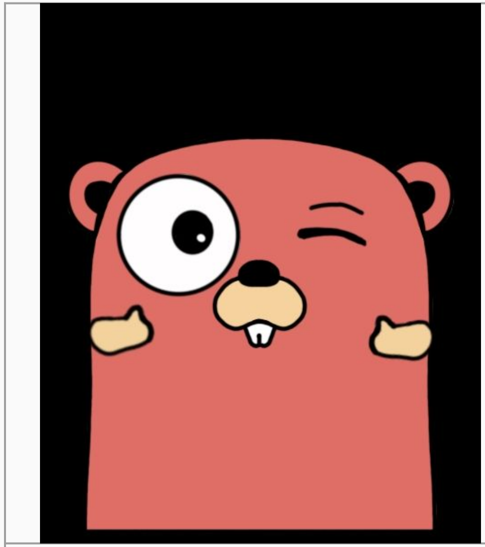
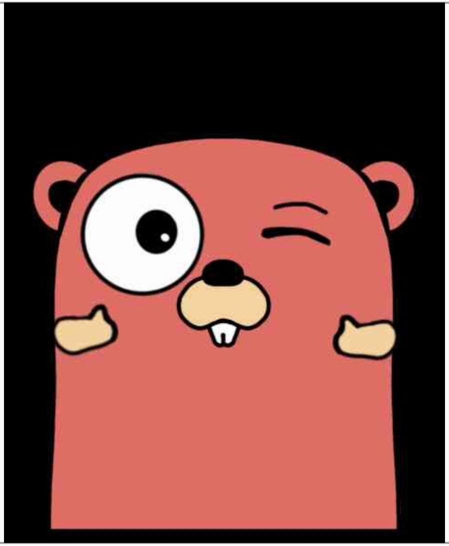
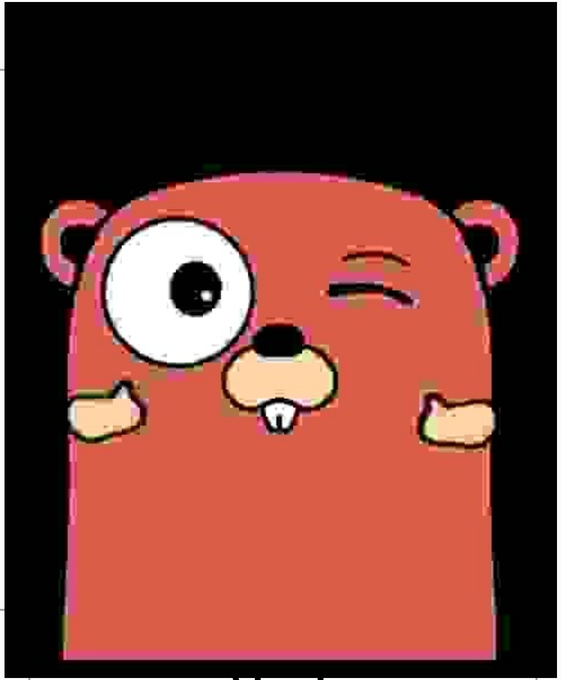
# png2jpg.go: quality impact

| | | |
|---|---|---|
|  |  |  |
| q=100 | q=30 | q=1 |

# png2jpg.go: quality impact

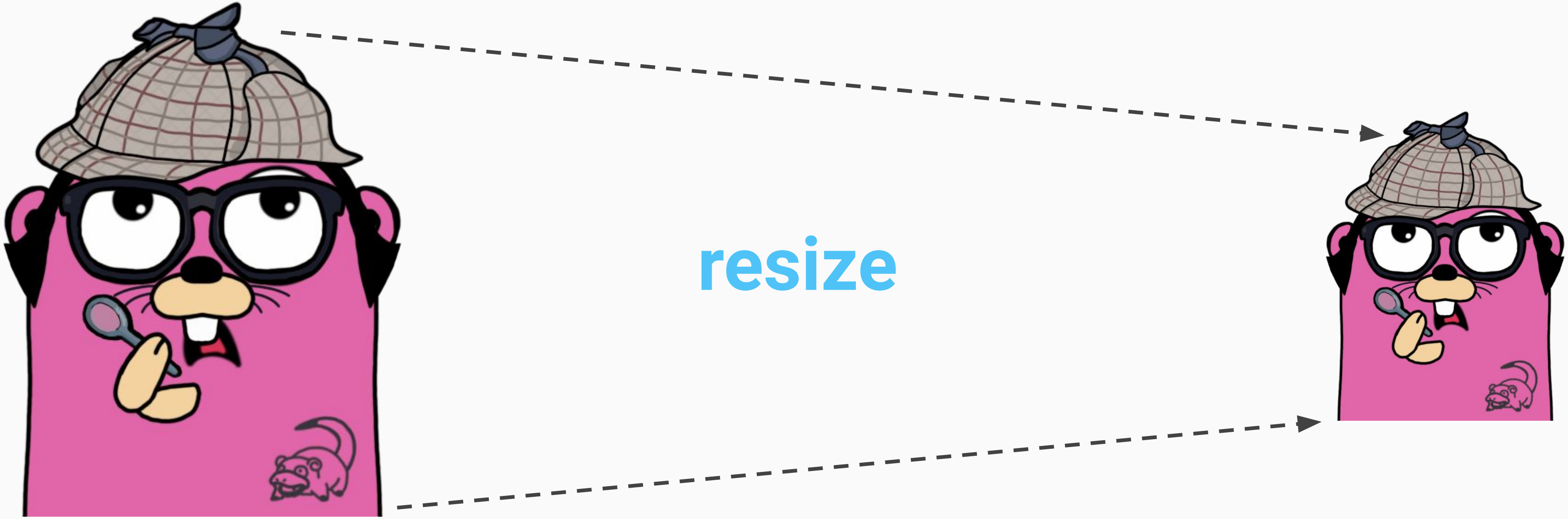
| | | |
|---|---|---|
|  |  |  |
| q=100 | q=30 | q=1 |

# png2jpg.go: quality impact

| | | |
|---|---|---|
|  |  |  |
| q=100 | q=30 | q=1 |

What if you need a smaller gopher?
There is no “resize” in stdlib.

# github.com/nfnt/resize package



resize

# Resize an image

```
// import "github.com/nfnt/resize"

algorithm := resize.Bicubic
result:= resize.Resize(w, h, img, algorithm)

// result contains resized image data.
```

# Let’s run it!

```
$ go run resize.go -w 100 gopher.png
```

Time to try manipulating individual pixels in existing image.

We’re about to invert gopher colors!

# Inverting PNG colors

1. Decode PNG image, cast it to NRGBA.
2. Invert every individual pixel inside NRGBA.
3. Encode NRGBA to file.

png.Decode returned type depends on the image contents.
For our gopher it’s NRGBA.

# invert.go: cast to NRGBA

```go
img, _ := png.Decode(f)

dst, ok := img.(*image.NRGBA)
if !ok {
  log.Panicf("not NRGBA")
}
```

# invert.go: cast to NRGBA

```go
img, _ := png.Decode(f)

dst, ok := img.(*image.NRGBA)
if !ok {
  log.Panicf("not NRGBA")
}
```

# invert.go: cast to NRGBA

```go
img, _ := png.Decode(f)

dst, ok := img.(*image.NRGBA)
if !ok {
  log.Panicf("not NRGBA")
}
```

# invert.go: loop over pixels

```go
bounds := dst.Bounds()
height := bounds.Size().Y
width := bounds.Size().X

for y := 0; y < height; y++ {
  i := y * dst.Stride
  for x := 0; x < width; x++ {
    /* Loop body. See next slides */
  }
}
```

# invert.go: loop over pixels

```
bounds := dst.Bounds()
height := bounds.Size().Y
width := bounds.Size().X

for y := 0; y < height; y++ {
  i := y * dst.Stride
  for x := 0; x < width; x++ {
    /* Loop body. See next slides */
  }
}
```

# Invert.go: inverting colors

```
d := dst.Pix[i : i+4 : i+4]

// Invert colors.
d[0] = 255 - d[0] // R
d[1] = 255 - d[1] // G
d[2] = 255 - d[2] // B
d[3] = d[3]       // Alpha, unchanged

i += 4 // Go to the next RGBA component.
```

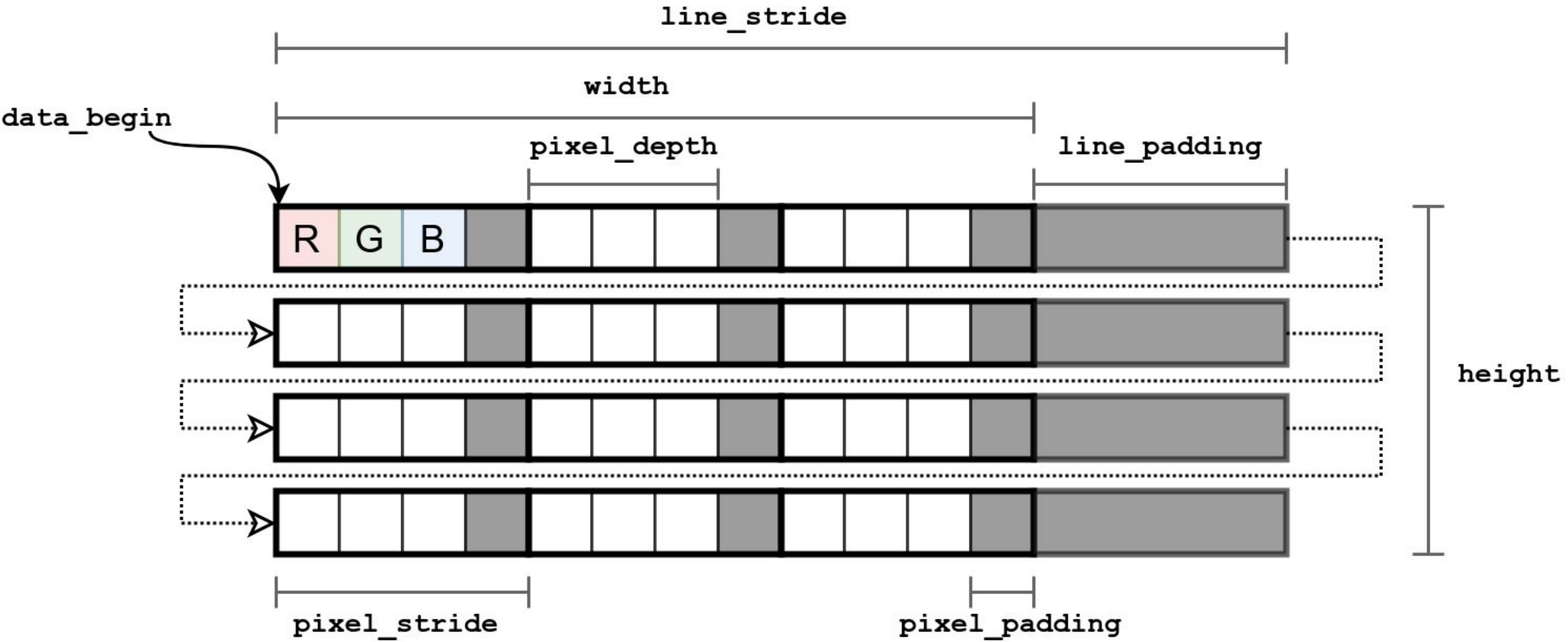
# Invert.go: inverting colors

```
d := dst.Pix[i : i+4 : i+4]

// Invert colors.
d[0] = 255 - d[0] // R
d[1] = 255 - d[1] // G
d[2] = 255 - d[2] // B
d[3] = d[3]       // Alpha, unchanged

i += 4 // Go to the next RGBA component.
```

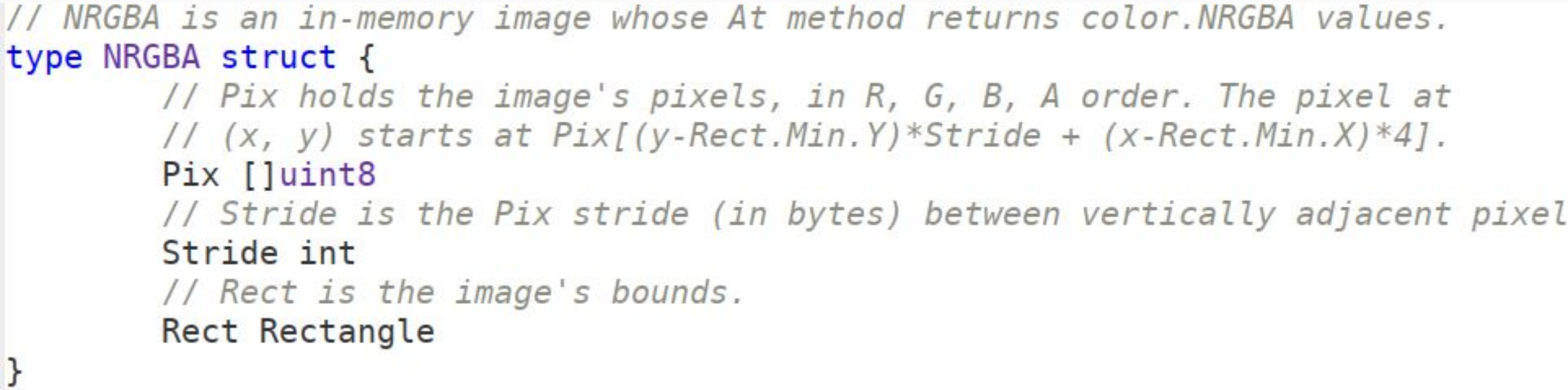
# Invert.go: inverting colors

```go
d := dst.Pix[i : i+4 : i+4]

// Invert colors.
d[0] = 255 - d[0] // R
d[1] = 255 - d[1] // G
d[2] = 255 - d[2] // B
d[3] = d[3]       // Alpha, unchanged

i += 4 // Go to the next RGBA component.
```

If you’re confused about pixels layout, here goes the explanation.

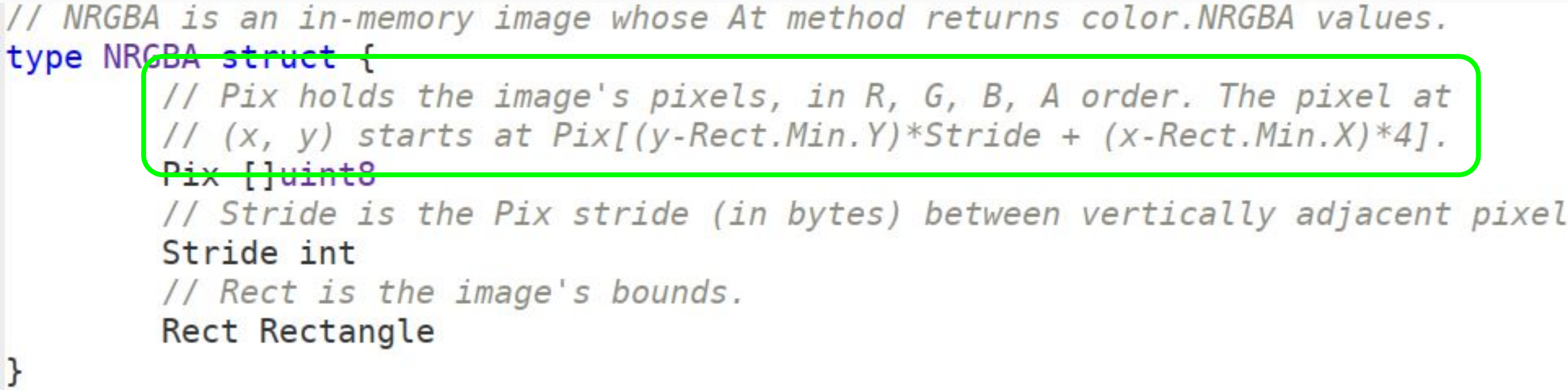

line_stride
width
data_begin
pixel_depth
line_padding
R
G
B
height
pixel_stride
pixel_padding

# NRGBA structure



```
// NRGBA is an in-memory image whose At method returns color.NRGBA values.
type NRGBA struct {
        // Pix holds the image's pixels, in R, G, B, A order. The pixel at
        // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)*4].
        Pix []uint8
        // Stride is the Pix stride (in bytes) between vertically adjacent pixel
        Stride int
        // Rect is the image's bounds.
        Rect Rectangle
}
```

# NRGBA structure



```
// NRGBA is an in-memory image whose At method returns color.NRGBA values.
type NRGBA struct {
        // Pix holds the image's pixels, in R, G, B, A order. The pixel at
        // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)*4].
        Pix []uint8
        // Stride is the Pix stride (in bytes) between vertically adjacent pixel
        Stride int
        // Rect is the image's bounds.
        Rect Rectangle
}
```

We store 2-D information inside 1-D array for efficiency.

This is why we need a “stride” and “i” index calculation.

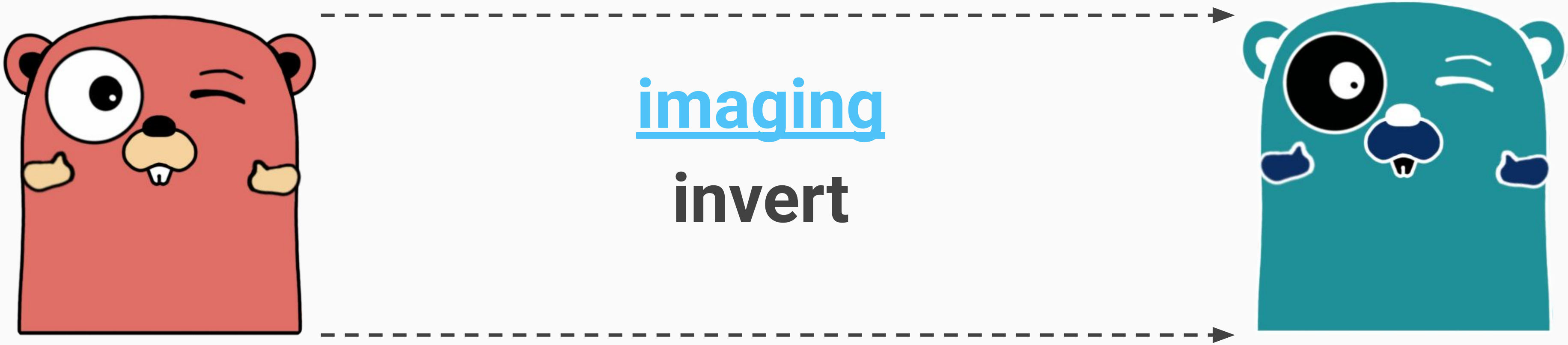
# invert.go: loop over pixels

```
for y := 0; y < height; y++ {
  i := y * dst.Stride
  for x := 0; x < width; x++ {
    d := dst.Pix[i : i+4 : i+4]

    /* ...rest of the loop. */
  }
}
```

# Let’s run it!

```
$ go run invert.go gopher.png
```

# github.com/disintegration/imaging package



imaging

**invert**

# github.com/disintegration/imaging package

- Crop, fit, resize
- Grayscale, invert, blur
- Convolutions
- Transpose, transverse, flip, rotate
- And more!

# Programs we created today (links):

compose

png2jpg

resize

invert

# Please, ask questions!

^ Slides ^

https://bit.ly/2ksX833