The Shoes Manual

# Hello!

## Shoes is a tiny graphics toolkit. It's simple and straightforward. Shoes was born to be easy! Really, it was made for absolute beginners. There's really nothing to it.

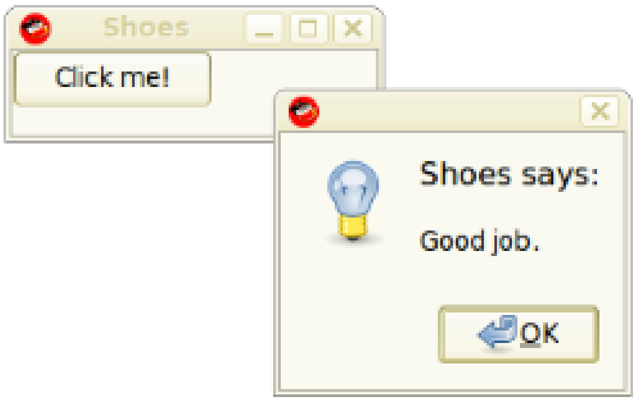
---

You see, the trivial Shoes program can be just one line:

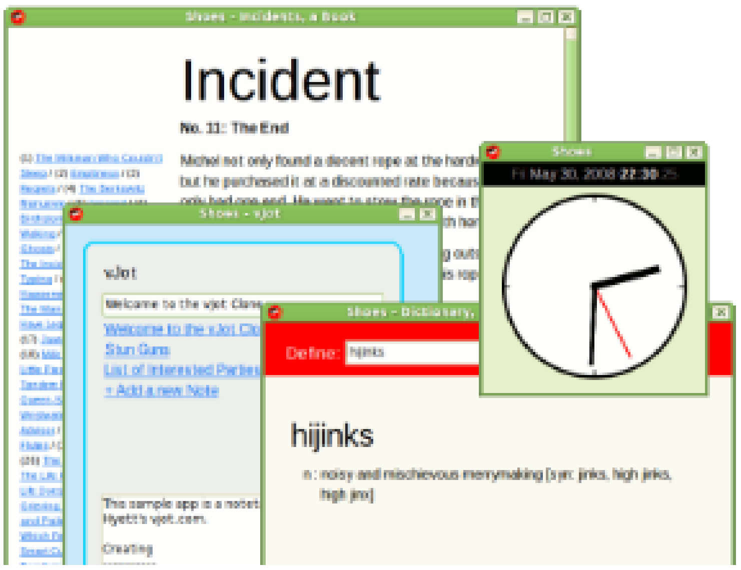
```ruby
Shoes.app { button("Click me!") { alert("Good job.") } }
```

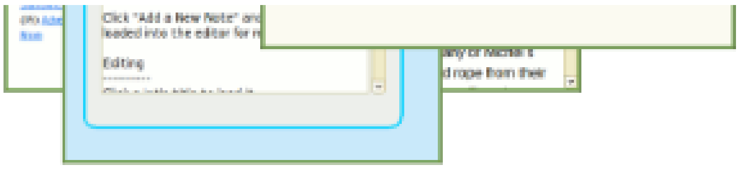
Shoes programs are written in a language called Ruby. When Shoes is handed this simple line of Ruby code, a window appears with a button inside reading "Click me!" When the button is clicked, a message pops up.

On Linux, here's how this might look:



While lots of Shoes apps are graphical games and art programs, you can also layout text and edit controls easily.


Shoes - Incidents, a book
Incident
No. 11: The End
Shoes - vJot
vJot
Welcome to the vJot Clone
Stun Guns
List of Interested Parties
+ Add a new Note
Shoes - Dictionary
Define:
hijinks
n.: noisy and mischievous merrymaking [syn: jinks, high jinks, high jinx]
Shoes
Fri May 30, 2008 22:30:25



And, ideally, Shoes programs will run on any of the major platforms out there. Microsoft Windows, Apple's Mac OS X, Linux and many others.

So, welcome to Shoes' built-in manual. This manual is a Shoes program itself!

Next: Introducing Shoes

The Shoes Manual

# Introducing Shoes

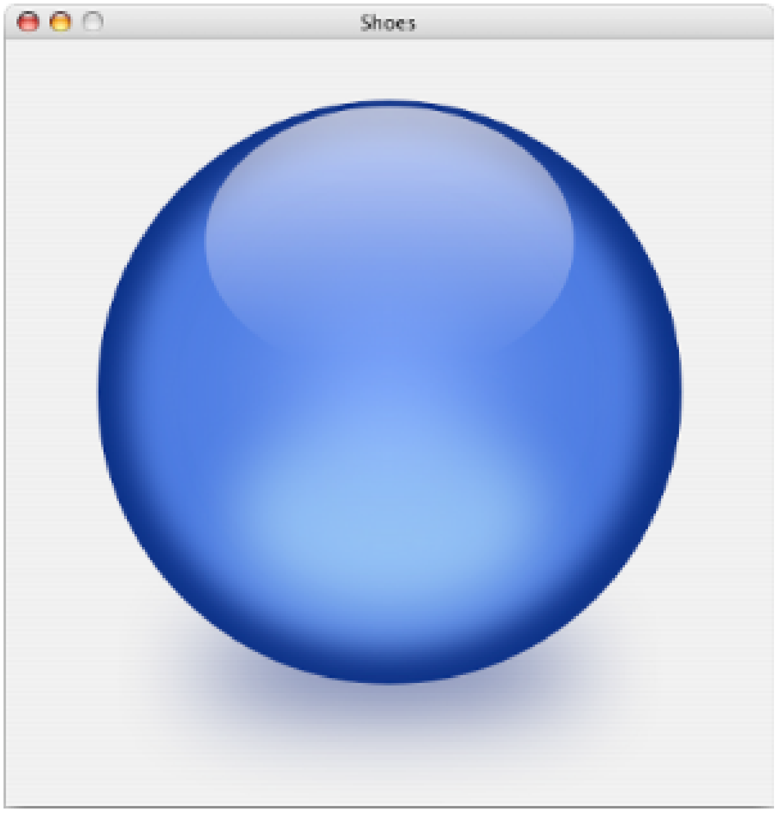
How does Shoes look on OS X and Windows? Does it really look okay? Is it all ugly and awkward? People must immediately convulse! It must be so watered down trying to do everything.

---

Well, before getting into the stuff about installing and running Shoes, time to just check out some screenshots, to give you an idea of what you can do.

## Mac OS X

Shoes runs on Apple Mac OS X Leopard, as well as Tiger. Shoes supports PowerPC machines as well, however, there is no video support on that platform.

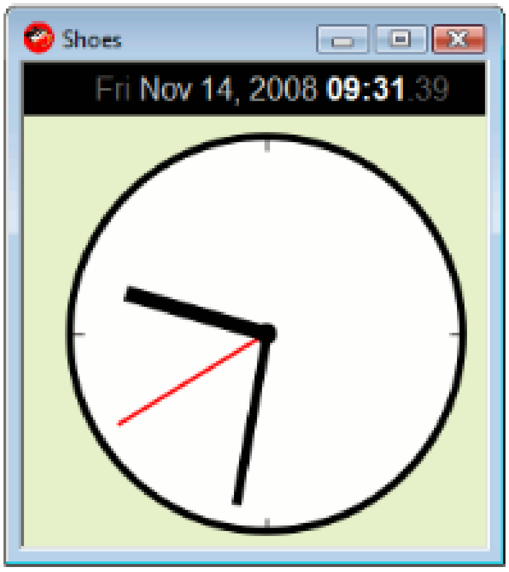


This is the `simple-sphere.rb` sample running on Tiger. Notice that the app runs inside a normal OS X window border.

The whole sphere is drawn with blurred ovals and shadows. You can draw and animate shapes and apply effects to those shapes in Shoes.

## Windows

Shoes runs on all versions of **Microsoft Windows XP**, **Microsoft Windows Vista**, and anything else **Windows 2000** compatible.

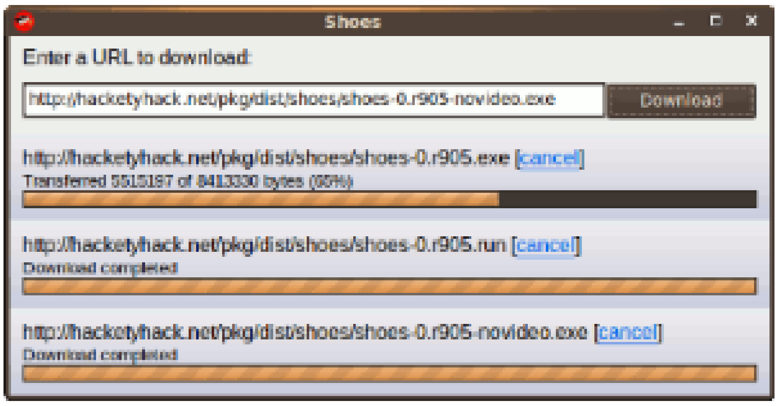


Above is pictured the `simple-clock.rb` sample running on Windows Vista. This example is also draws ovals and lines to build the clock, which is animated to repaint itself several times each second.

Notice the text on the top of the app, showing the current time. Shoes has the skills to layout words using any color, bold, italics, underlines, and supports loading fonts from a file.

## Linux

Here's a screenshot of the `simple-downloader.rb` sample running on **Ubuntu Linux**.



Shoes

Enter a URL to download:

http://hacketyhack.net/pkg/dist/shoes/shoes-0.r905-novideo.exe

Download

http://hacketyhack.net/pkg/dist/shoes/shoes-0.r905.exe [cancel]
Transferred 5515197 of 8413330 bytes (65%)

http://hacketyhack.net/pkg/dist/shoes/shoes-0.r905.run [cancel]
Download completed

http://hacketyhack.net/pkg/dist/shoes/shoes-0.r905-novideo.exe [cancel]
Download completed

Notice the buttons and progress bars. These types of controls look different on OS X and Windows. The text and links would look the same, though.

Shapes, text, images and videos all look the same on every platforms. However, native controls (like edit lines and edit boxes) will match the look of the window theme. Shoes will try to keep native controls all within the size you give them, only the look will vary.

---

Next: Installing Shoes

The Shoes Manual

# Installing Shoes

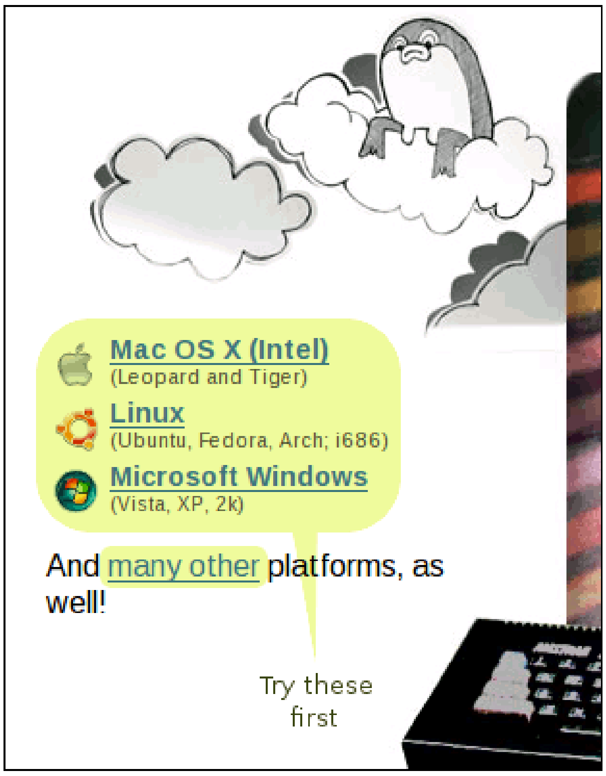
Okay, on to installing Shoes. I'm sure you're wondering: do I need to install Ruby? Do I need to unzip anything? What commands to I need to type?

Nope. You don't need Ruby. You don't need WinZip. Nothing to type.

On most systems, starting Shoes is just a matter of running the installer and clicking the Shoes icon. Shoes comes with everything built in. We'll talk through all the steps, though, just to be clear about it.

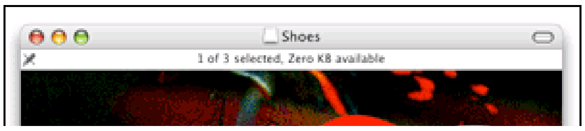
## Step 1: Installing Shoes

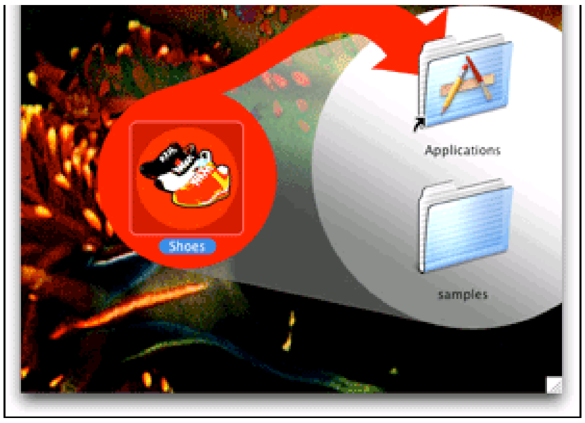
You'll want to visit shoooes.net to download the Shoes installer. Usually, you'll just want one of the installers on the top corner of the home page.



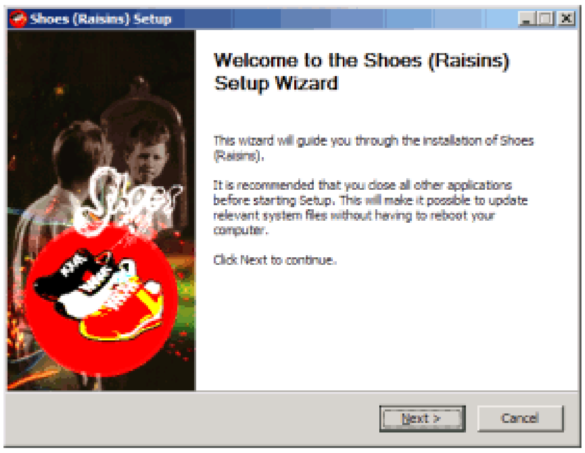
Here's how to run the installer:

- On **Mac OS X**, you'll have a file ending with **.dmg**. Double-click this file and a window should appear with a **Shoes** icon and an **Applications** folder. Following the arrow, drag the Shoes icon into the **Applications** folder.





- On **Windows**, you'll download a **.exe** file. Double-click this file and follow the instructions.



Welcome to the Shoes (Raisins) Setup Wizard

This wizard will guide you through the installation of Shoes (Raisins).

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

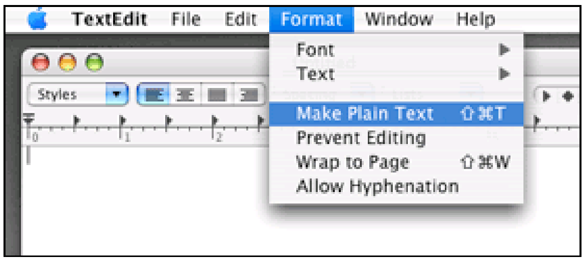
- On **Linux**, you'll download a file ending with **.run**. Double-click this file and Shoes will start up. (You can also run this file from a prompt as if it was a shell script. In fact, it is a shell script!)

## Step 2: Start a New Text File

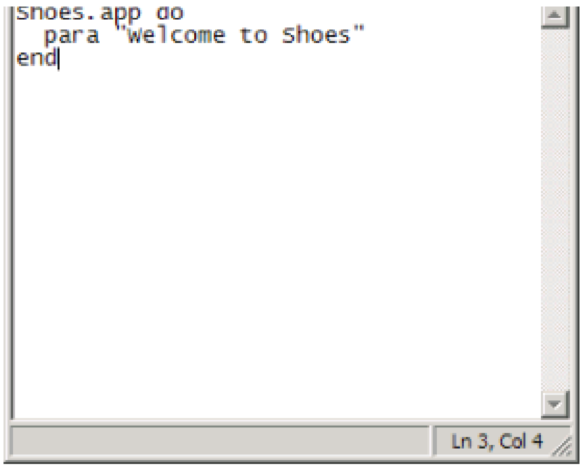
Shoes programs are just plain text files ending with a **.rb** extension.

Here are a few ways to create a blank text file:

- On **Mac OS X**, visit your **Applications** folder and double-click on the **TextEdit** app. A blank editor window should come up. Now, go to the **Format** menu and select the **Make Plain Text** option. Okay, you're all set!



- On **Windows**, go to the Start menu. Select **All Programs**, then **Accessories**, then **Notepad**.



```
Shoes.app do
  para "Welcome to Shoes"
end
```

- On **Linux**, most distros come with **gedit**. You might try running that. Or, if your distro is KDE-based, run **kate**.

Now, in your blank window, type in the following:

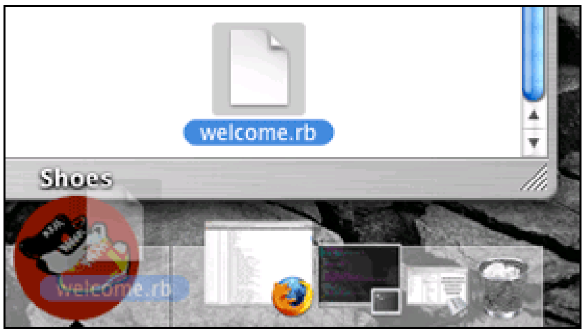
```
Shoes.app do
  background "#DFA"
  para "Welcome to Shoes"
end
```

Save to your desktop as `welcome.rb`.

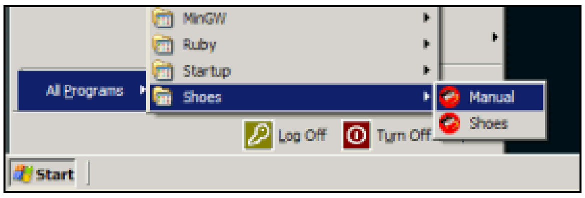
## Step 3: Run It! Go Shoes!

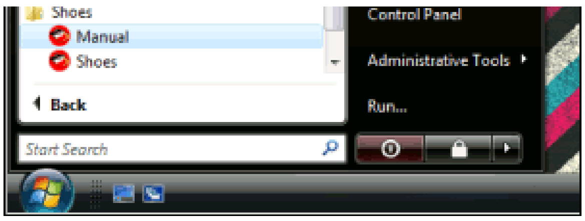
To run your program:

- On **Mac OS X**, visit your **Applications** folder again. This time, double-click the **Shoes** icon in that folder. You should see the red shoes icon appear in the dock. Drag your `welcome.rb` from the desktop on to that dock icon.



- On **Windows**, get to the Start menu. Go into **All Programs**, then **Shoes**, then **Shoes**. A file selector box should come up. Browse to your desktop and select `welcome.rb`. Click **OK** and you're on your way.





- On **Linux**, run Shoes just like you did in step one. You should see a file selector box. Browse to your desktop, select `welcome.rb` and hit **OK**.

So, not much of a program yet. But it's something! You've got the knack of it, at least!

## What Can You Make With Shoes?

Well, you can make windowing applications. But Shoes is inspired by the web, so applications tend to use images and text layout rather than a lot of widgets. For example, Shoes doesn't come with tabbed controls or toolbars. Shoes is a *tiny* toolkit, remember?

Still, Shoes does have a few widgets like buttons and edit boxes. And many missing elements (like tabbed controls or toolbars) can be simulated with images.

Shoes is written in part thanks to a very good art engine called Cairo, which is used for drawing with shapes and colors. In this way, Shoes is inspired by NodeBox and Processing, two very good languages for drawing animated graphics.

Next: The Rules of Shoes

The Shoes Manual

# The Rules of Shoes

Time to stop guessing how Shoes works. Some of the tricky things will come back to haunt you. I've boiled down the central rules to Shoes. These are the things you MUST know to really make it all work.

---

These are general rules found throughout Shoes. While Shoes has an overall philosophy of simplicity and clarity, there are a few points that need to be studied and remembered.

## Shoes Tricky Blocks

Okay, this is absolutely crucial. Shoes does a trick with blocks. This trick makes everything easier to read. But it also can make blocks harder to use once you're in deep.

**Let's take a normal Ruby block:**

```
ary = ['potion', 'swords', 'shields']
ary.each do |item|
  puts item
end
```

In Shoes, these sorts of blocks work the same. This block above loops through the array and stores each object in the `item` variable. The `item` variable disappears (goes out of scope) when the block ends.

One other thing to keep in mind is that `self` stays the same inside normal Ruby blocks. Whatever `self` was before the call to `each`, it is the same inside the `each` block.

**Both of these things are also true for most Shoes blocks.**

```
Shoes.app do
  stack do
    para "First"
    para "Second"
    para "Third"
  end
end
```

Here we have two blocks. The first block is sent to `Shoes.app`. This `app` block changes `self`.

The other block is the `stack` block. That block does NOT change self.

**For what reason does the `app` block change self?** Let's start by spelling out that last example completely.

```
Shoes.app do
  self.stack do
    self.para "First"
    self.para "Second"
    self.para "Third"
  end
end
```

All of the `self`s in the above example are the App object. Shoes uses Ruby's `instance_eval` to change self inside the `app` block. So the method calls to `stack` and `para` get sent to the app.

**This also is why you can use instance variables throughout a Shoes app:**

```
Shoes.app do
  @s = stack do
    @p1 = para "First"
    @p2 = para "Second"
    @p3 = para "Third"
  end
end
```

These instance variables will all end up inside the App object.

**Whenever you create a new window, `self` is also changed.** So, this means the window and dialog methods, in addition to Shoes.app.

```
Shoes.app :title => "MAIN" do
  para self
  button "Spawn" do
    window :title => "CHILD" do
      para self
    end
  end
end
```

## Block Redirection

The `stack` block is a different story, though. It doesn't change `self` and it's basically a regular block.

**But there's a trick:** when you attach a `stack` and give it a block, the App object places that stack in its memory. The stack gets popped off when the block ends. So all drawing inside the block gets **redirected** from the App's top slot to the new stack.

So those three `para`s will get drawn on the `stack`, even though they actually get sent to the App object first.

```
Shoes.app do
  stack do
    para "First"
    para "Second"
    para "Third"
  end
end
```

A bit tricky, you see? This can bite you even if you know about it.

One way it'll get you is if you try to edit a stack somewhere else in your program, outside the `app` block.

Like let's say you pass around a stack object. And you have a class that edits that object.

```
class Messenger
  def initialize(stack)
    @stack = stack
  end
  def add(msg)
    @stack.append do
      para msg
    end
  end
end
```

So, let's assume you pass the stack object into your Messenger class when the app starts. And, later, when a message comes in, the `add` method gets used to append a paragraph to that stack. Should work, right?

Nope, it won't work. The `para` method won't be found. The App object isn't around any more. And it's the one with the `para` method.

Fortunately, each Shoes object has an `app` method that will let you reopen the App object so you can do somefurther editing.

```
class Messenger
  def initialize(stack)
    @stack = stack
  end
  def add(msg)
    @stack.app do
      @stack.append do
        para msg
      end
    end
  end
end
```

As you can imagine, the `app` object changes `self` to the App object.

So the rules here are:

1. **Methods named "app" or which create new windows alter `self` to the App object.**
(This is true for both Shoes.app and Slot.app, as well as window and dialog.)
2. **Blocks attached to stacks, flows or any manipulation method (such as append) do not change self. Instead, they pop the slot on to the app's editing stack.**

## Careful With Fixed Heights

Fixed widths on slots are great so you can split the window into columns.

```
Shoes.app do
  flow do
    stack :width => 200 do
      caption "Column one"
      para "is 200 pixels wide"
    end
    stack :width => -200 do
      caption "Column two"
      para "is 100% minus 200 pixels wide"
    end
  end
end
```

Fixed heights on slots should be less common. Usually you want your text and images to just flow down the window as far as they can. Height usually happens naturally.

The important thing here is that fixed heights actually force slots to behave differently. To be sure that the end of the slot is chopped off perfectly, the slot becomes a **nested window**. A new layer is created by the operating system to keep the slot in a fixed square.

On difference between normal slots and nested window slots is that the latter can have scrollbars.

```
Shoes.app do
  stack :width => 200, :height => 200, :scroll => true do
    background "#DFA"
    100.times do |i|
      para "Paragraph No. #{i}"
    end
  end
end
```

These nested windows require more memory. They tax the application a bit more. So if you're experiencing some slowness with hundreds of fixed-height slots, try a different approach.

## Image and Shape Blocks

Most beginners start littering the window with shapes. It's just easier to throw all your rectangles and ovals in a slot.

**However, bear in mind that Shoes will create objects for all those shapes!**

```
Shoes.app do
  fill black(0.1)
  100.times do |i|
    oval i, i, i * 2
  end
end
```

In this example, one-hundred Oval objects are created. This isn't too bad. But things would be slimmer if we made these into a single shape.

```
Shoes.app do
  fill black(0.1)
  shape do
    100.times do |i|
      oval i, i, i * 2
    end
  end
end
```

Oh, wait. The ovals aren't filled in this time! That's because the ovals have been combined into a single hhuge shape. And Shoes isn't sure where to fill in this case.

So you usually only want to combine into a single shape when you're dealing strictly with outlines.

Another option is to combine all those ovals into a single image.

```
Shoes.app do
  fill black(0.1)
  image 300, 300 do
    100.times do |i|
      oval i, i, i * 2
    end
  end
end
```

There we go! The ovals are all combined into a single 300 x 300 pixel image. In this case, storing that image in memory might be much bigger than having one-hundred ovals around. But when you're dealing with thousands of shapes, the image block can be cheaper.

The point is: it's easy to group shapes together into image or shape blocks, so give it a try if you're looking to gain some speed. Shape blocks particularly will save you some memory and speed.

## UTF-8 Everywhere

Ruby itself isn't Unicode aware. And UTF-8 is a type of Unicode. (See Wikipedia for a full explanation of UTF-8.)

However, UTF-8 is common on the web. And lots of different platforms support it. So to cut down on the amount of conversion that Shoes has to do, Shoes expects all strings to be in UTF-8 format.

This is great because you can show a myriad of languages (Russian, Japanese, Spanish, English) using UTF-8 in Shoes. Just be sure that your text editor uses UTF-8!

To illustrate:

```
Shoes.app do
  stack :margin => 10 do
    @edit = edit_box :width => 1.0 do
      @para.text = @edit.text
    end
    @para = para ""
  end
end
```

This app will copy anything you paste into the edit box and display it in a Shoes paragraph. You can try copying some foreign text (such as Greek or Japanese) into this box to see how it displays.

This is a good test because it proves that the edit box gives back UTF-8 characters. And the paragraph can be set to any UTF-8 characters.

**Important note:** if some UTF-8 characters don't display for you, you will need to change the paragraph's font. This is especially common on OS X.

So, a good Japanese font on OS X is **AppleGothic**.

```
Shoes.app do
  para "てすと (te-su-to)", :font => "AppleGothic, Arial"
end
```

Again, anything which takes a string in Shoes will need a UTF-8 string. Edit boxes, edit lines, list boxes, window titles and text blocks all take UTF-8. If you give a string with bad characters in it, an error will show up in the console.

## The Main App and Its Requires

Each Shoes app is given a little room where it can create itself. You can create classes and set variables and they won't be seen by other Shoes programs. Each program runs inside its own anonymous class.

```
main = self
Shoes.app do
  para main.to_s
end
```

This anonymous class is called `(shoes)` and it's just an empty, unnamed class. The `Shoes` module is mixed into this class (using `include Shoes`) so that you can use either `Para` or `Shoes::Para` when referring to the paragraph class.

The advantages of this approach are:

- Shoes apps cannot share local variables.
- Classes created in the main app code are temporary.
- The Shoes module can be mixed in to the anonymous class, but not the top-level environment of Ruby itself.
- Garbage collection can clean up apps entirely once they complete.

The second part is especially important to remember.

```
class Storage; end
Shoes.app do
  para Storage.new
end
```

The `Storage` class will disappear once the app completes. Other apps aren't able to use the Storage class. And it can't be gotten to from files that are loaded using `require`.

When you `require` code, though, that code will stick around. It will be kept in the Ruby top-level environment.

So, the rule is: **keep your temporary classes in the code with the app and keep your permanent classes in requires.**

Next: Shoes

The Shoes Manual

# Shoes

Shoes is all about drawing windows and the stuff inside those windows. Let's focus on the window itself, for now. The other sections Slots and Elements cover everything that goes inside the window.

---

For here on, the manual reads more like a dictionary. Each page is mostly a list of methods you can use for each topic covered. The idea is to be very thorough and clear about everything.

So, if you've hit this far in the manual and you're still hazy about getting started, you should probably either go back to the beginning of the manual. Or you could try Nobody Knows Shoes, the beginner's leaflet on the web.

## Finding Your Way

This section covers:

- Built-in methods - general methods available anywhere in a Shoes program.
- The App window - methods found attached to every main Shoes window.
- The Styles Master List - a complete list of every style in Shoes.
- The Classes list - a chart showing what Shoes classes subclass what.
- The Colors list - a chart of all built-in colors and the rgb numbers for each.

If you find yourself paging around a lot and not finding something, give the Search page a try. It's the quickest way to get around.

After this general reference, there are two other more specific sections:

- Slots - covering stack and flow, the two types of slots.
- Elements - documentation for all the buttons, shapes, images, and so on.

Two really important pages in there are the Element Creation page (which lists all the elements you can add) and the Common Methods page (which lists methods you'll find on any slot or element.)

---

Next: Built-in Methods

The Shoes Manual

# Built-in Methods

## These methods can be used anywhere throughout Shoes programs.

All of these commands are unusual because you don't attach them with a dot. **Every other method in this manual must be attached to an object with a dot.** But these are built-in methods (also called: Kernel methods.) Which means no dot!

A common one is `alert`:

```
alert "No dots in sight"
```

Compare that to the method `reverse`, which isn't a Kernel method and is only available for Arrays and Strings:

```
"Plaster of Paris".reverse
 #=> "siraP fo retsalP"
[:dogs, :cows, :snakes].reverse
 #=> [:snakes, :cows, :dogs]
```

Most Shoes methods for drawing and making buttons and so on are attached to slots. See the section on Slots for more.

# Built-in Constants

Shoes also has a handful of built-in constants which may prove useful if you are trying to sniff out what release of Shoes is running.

**Shoes::RELEASE_NAME** contains a string with the name of the Shoes release. All Shoes releases are named, starting with Curious.

**Shoes::RELEASE_ID** contains a number representing the Shoes release. So, for example, Curious is number 1, as it was the first official release.

**Shoes::REVISION** is the Subversion revision number for this build.

**Shoes::FONTS** is a complete list of fonts available to the app. This list includes any fonts loaded by the font method.

### **alert(message: a string)** » nil

Pops up a window containing a short message.

```
alert("I'm afraid I must interject!")
```

Please use alerts sparingly, as they are incredibly annoying! If you are using alerts to show messages to help you debug your program, try checking out the debug or info methods.

### **ask(message: a string)** » a string

Pops up a window and asks a question. For example, you may want to ask someone their name.

```
name = ask("Please, enter your name:")
```

When the above script is run, the person at the computer will see a window with a blank box for entering their name. The name will then be saved in the `name` variable.

### **ask_color(title: a string)** » Shoes::Color

Pops up a color picker window. The program will wait for a color to be picked, then gives you back a Color object. See the `Color` help for some ways you can use this color.

```
backcolor = ask_color("Pick a background")
Shoes.app do
 background backcolor
end
```

### **ask_open_file()** » a string

Pops up an "Open file..." window. It's the standard window which shows all of your folders and lets you select a file to open. Hands you back the name of the file.

```
filename = ask_open_file
puts File.read(filename)
```

### **ask_save_file()** » a string

Pops up a "Save file..." window, similiar to `ask_open_file`, described previously.

```
save_as = ask_save_file
```

### **ask_open_folder()** » a string

Pops up an "Open folder..." window. It's the standard window which shows all of your folders and lets you select a folder to open. Hands you back the name of the folder.

```
folder = ask_open_folder
puts Dir[folder].entries
```

### **ask_save_folder()** » a string

»

Pops up a "Save folder..." window, similiar to `ask_open_folder`, described previously. On OS X, this method currently behaves like an alias of `ask_open_folder`.

```
save_to = ask_save_folder
```

**confirm(question: a string)** » true or false

Pops up a yes-or-no question. If the person at the computer, clicks **yes**, you'll get back a `true`. If not, you'll get back `false`.

```
if confirm("Draw a circle?")
 oval :top => 0, :left => 0, :radius => 50
end
```

**debug(message: a string)** » nil

Sends a debug message to the Shoes console. You can bring up the Shoes console by pressing `Alt-/` on any Shoes window (or `⌘-/` on OS X.)

```
debug("Running Shoes on " + RUBY_PLATFORM)
```

Also check out the error, warn and info methods.

**error(message: a string)** » nil

Sends an error message to the Shoes console. This method should only be used to log errors. Try the debug method for logging messages to yourself.

Oh, and, rather than a string, you may also hand exceptions directly to this method and they'll be formatted appropriately.

**exit()**

Stops your program. Call this anytime you want to suddenly call it quits.

**font(message: a string)** » an array of font family names

Loads a TrueType (or other type of font) from a file. While TrueType is supported by all platforms, your platform may support other types of fonts. Shoes uses each operating system's built-in font system to make this work.

Here's a rough idea of what fonts work on which platforms:

- Bitmap fonts (.bdf, .pcf, .snf) - Linux
- Font resource (.fon) - Windows
- Windows bitmap font file (.fnt) - Linux, Windows
- PostScript OpenType font (.otf) - Mac OS X, Linux, Windows
- Type1 multiple master (.mmm) - Windows
- Type1 font bits (.pfb) - Linux, Windows

- Type1 font metrics (.pfm) - Linux, Windows
- TrueType font (.ttf) - Mac OS X, Linux, Windows
- TrueType collection (.ttc) - Mac OS X, Linux, Windows

If the font is properly loaded, you'll get back an array of font names found in the file. Otherwise, `nil` is returned if no fonts were found in the file.

Also of interest: the `Shoes::FONTS` constant is a complete list of fonts available to you on this platform. You can check for a certain font by using `include?`.

```
if Shoes::FONTS.include? "Helvetica"
  alert "Helvetica is available on this system."
else
  alert "You do not have the Helvetica font."
end
```

If you have trouble with fonts showing up, make sure your app loads the font before it is used. Especially on OS X, if fonts are used before they are loaded, the font cache will tend to ignore loaded fonts.

**gradient(color1, color2)** » Shoes::Pattern

Builds a linear gradient from two colors. For each color, you may pass in a Shoes::Color object or a string describing the color.

**gray(the numbers: darkness, alpha)** » Shoes::Color

Create a grayscale color from a level of darkness and, optionally, an alpha level.

```
black = gray(0.0)
white = gray(1.0)
```

**info(message: a string)** » nil

Logs an informational message to the user in the Shoes console. So, where debug messages are designed to help the program figure out what's happening, `info` messages tell the user extra information about the program.

```
info("You just ran the info example on Shoes #{Shoes::RELEASE_NAME}.")
```

For example, whenever a Shy file loads, Shoes prints an informational message in the console describing the author of the Shy and its version.

**rgb(a series of numbers: red, green, blue, alpha)** » Shoes::Color

Create a color from red, green and blue components. An alpha level (indicating transparency) can also be added, optionally.

When passing in a whole number, use values from 0 to 255.

```
blueviolet = rgb(138, 43, 226)
darkgreen = rgb(0, 100, 0)
```

Or, use a decimal number from 0.0 to 1.0.

```
blueviolet = rgb(0.54, 0.17, 0.89)
darkgreen = rgb(0, 0.4, 0)
```

This method may also be called as `Shoes.rgb`.

### **warn(message: a string)** » nil

Logs a warning for the user. A warning is not a catastrophic error (see error for that.) It is just a notice that the program will be changing in the future or that certain parts of the program aren't reliable yet.

To view warnings and errors, open the Shoes console with `Alt-/` (or `⌘-/` on OS X.)

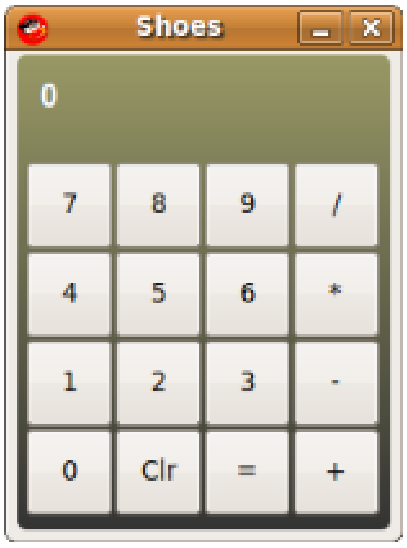
Next: The App Object

The Shoes Manual

# The App Object

An App is a single window running code at a URL. When you switch URLs, a new App object is created and filled up with stacks, flows and other Shoes elements.

The App is the window itself. Which may be closed or cleared and filled with new elements.



The App itself, in slot/box terminology, is a flow. See the *Slots* section for more, but this just means that any elements placed directly at the top-level will flow.

**Shoes.app(styles) { ... }** » Shoes::App

Starts up a Shoes app window. This is the starting place for making a Shoes program. Inside the block, you fill the window with various Shoes elements (buttons, artwork, etc.) and, outside the block, you use the `styles` to describe how big the window is. Perhaps also the name of the app or if it's resizable.

```
Shoes.app(:title => "White Circle",
  :width => 200, :height => 200, :resizable => false) {
    background black
    fill white
    oval :top => 20, :left => 20, :radius => 160
}
```

In the case above, a small window is built. 200 pixels by 200 pixels. It's not resizable. And, inside the window, two elements: a black background and a white circle.

Once an app is created, it is added to the Shoes.APPS list. If you want an app to spawn more windows, see the window method and the dialog method.

**Shoes.APPS()** » An array of Shoes::App objects

Builds a complete list of all the Shoes apps that are open right now. Once an app is closed, it is removed from the list. Yes, you can run many apps at once in Shoes. It's completely encouraged.

**clipboard()** » a string

Returns a string containing all of the text that's on the system clipboard. This is the global clipboard that every program on the computer cuts and pastes into.

**clipboard = a string**

Stores `a string` of text in the system clipboard.

**close()**

Closes the app window. If multiple windows are open and you want to close the entire application, use the built-in method `exit`.

**download(url: a string, styles)**

Starts a download thread (much like XMLHttpRequest, if you're familiar with JavaScript.) This method returns immediately and runs the download in the background. Each download thread also fires `start`, `progress` and `finish` events. You can send the download to a file or just get back a string (in the `finish` event.)

If you attach a block to a download, it'll get called as the `finish` event.

```
Shoes.app do
  stack do
    title "Searching Google", :size => 16
    @status = para "One moment..."
    # Search Google for 'shoes' and print the HTTP headers
    download "http://www.google.com/search?q=shoes" do |goog|
      @status.text = "Headers: " + goog.response.headers.inspect
    end
  end
end
```

And, if we wanted to use the downloaded data, we'd get it using `goog.response.body`. This example is truly the simplest form of `download`: pulling some web data down into memory and handling it once it's done.

Another simple use of `download` is to save some web data to a file, using the `:save` style.

```
Shoes.app do
  stack do
    title "Downloading Google image", :size => 16
    @status = para "One moment..."
    download "http://www.google.com/logos/nasa50th.gif",
      :save => "nasa50th.gif" do
        @status.text = "Okay, is downloaded."
    end
  end
end
```

In this case, you can still get the headers for the downloaded file, but `response.body` will be `nil`, since the data wasn't saved to memory. You will need to open the file to get the downloaded goods.

If you need to send certain headers or actions to the web server, you can use the `:method`, `:headers` and `:body` styles to customize the HTTP request. (And, if you need to go beyond these, you can always break out Ruby's OpenURI class.)

```
Shoes.app do
  stack do
    title "POSTing to Google", :size => 16
    @status = para "One moment..."
    download "http://www.stevex.net/dump.php",
             :method => "POST", :body => "v=1.0&q=shoes" do |dump|
      require 'hpricot'
      @status.text = Hpricot(dump.response.body).inner_text
    end
  end
end
```

As you can see from the above example, Shoes includes the Hpricot library for parsing HTML.

**location()** » a string

Gets a string containing the URL of the current app.

**mouse()** » an array of numbers: button, left, top

Identifies the mouse cursor's location, along with which button is being pressed.

```
Shoes.app do
  @p = para
  animate do
    button, left, top = self.mouse
    @p.replace "mouse: #{button}, #{left}, #{top}"
  end
end
```

**owner()** » Shoes::App

Gets the app which launched this app. In most cases, this will be `nil`. But if this app was launched using the window method, the owner will be the app which called `window`.

**started?()** » true or false

Has the window been fully constructed and displayed? This is useful for threaded code which may try to use the window before it is completely built. (Also see the `start` event which fires once the window is open.)

### visit(url: a string)

Changes the location, in order to view a different Shoes URL.

Absolute URLs (such as http://google.com) are okay, but Shoes will be expecting a Shoes application to be at that address. (So, google.com won't work, as it's an HTML app.)

---

Next: The Styles Master List

The Shoes Manual

# The Styles Master List

You want to mess with the look of things? Well, throughout Shoes, styles are used to change the way elements appear. In some cases, you can even style an entire class of elements. (Like giving all paragraphs a certain font.)

---

Styles are easy to spot. They usually show up when the element is created.

```
Shoes.app :title => "A Styling Sample" do
  para "Red with an underline", :stroke => red, :underline => "single"
end
```

Here we've got a `:title` style on the app. And on the paragraph inside the app, a red `:stroke` style and an `:underline` style.

The style hash can also be changed by using the style method, available on every element and slot.

```
Shoes.app :title => "A Styling Sample" do
  @text = para "Red with an underline"
  @text.style(:stroke => red, :underline => "single")
end
```

Most styles can also be set by calling them as methods. (I'd use the manual search to find the method.)

```
Shoes.app :title => "A Styling Sample" do
  @text = para "Red with an underline"
  @text.stroke = red
  @text.underline = "single"
end
```

Rather than making you plow through the whole manual to figure out what styles go where, this helpful page speeds through every style in Shoes and suggests where that style is used.

**:align** » a string

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*

The alignment of the text. It is either:

- 'left': Align the text to the left.
- 'center': Align the text in the center.

- 'right': Align the text to the right.

**:angle** » a number

For: *background, border, gradient*.

The angle at which to apply a gradient. Normally, gradient colors range from top to bottom. If the `:angle` is set to 90, the gradient will rotate 90 degrees counter-clockwise and the gradient will go from left to right.

**:attach** » a slot or element

For: *flow, stack*.

Pins a slot relative to another slot or element. Also, one may write `:attach => Window` to position the slot at the window's top, left corner. Taking this a bit further, the style `:top => 10, :left => 10, :attach => Window` would place the slot at (10, 10) in the window's coordinates.

If a slot is attached to an element that moves, the slot will move with it. If the attachment is reset to `nil`, the slot will flow in with the other objects that surround, as normal.

**:autoplay** » true or false

For: *video*.

Should this video begin playing after it appears? If set to `true`, the video will start without asking the user.

**:bottom** » a number

For: *all slots and elements*.

Sets the pixel coordinate of an element's lower edge. The edge is placed relative to its container's lower edge. So, `:bottom => 0` will align the element so that its bottom edge and the bottom edge of its slot touch.

**:cap** » :curve or :rect or :project

For: *arc, arrow, border, flow, image, mask, rect, star, shape, stack*.

Sets the shape of the line endpoint, whether curved or square. See the cap method for more explanation.

**:center** » true or false

For: *arc, image, oval, rect, shape*.

Indicates whether the `:top` and `:left` coordinates refer to the center of the shape or not. If set to `true`, this is similar to setting the transform method to `:center`.

**:change** » a proc

For: *edit_box, edit_line, list_box*.

The `change` event handler is stored in this style. See the change method for the edit_box, as an example.

**:checked** » true or false

For: *check, radio*.

Is this checkbox or radio button checked? If set to `true`, the box is checked. Also see the checked= method.

**:choose** » a string

For: *list_box*.

Sets the currently chosen item in the list. More information at choose.

**:click** » a proc

For: *arc, arrow, banner, button, caption, check, flow, image, inscription, line, link, mask, oval, para, radio, rect, shape, stack, star, subtitle, tagline, title*.

The `click` event handler is stored in this style. See the click method for a description.

**:curve** » a number

For: *background, border, rect*.

The radius of curved corners on each of these rectangular elements. As an example, if this is set to 6, the corners of the rectangle are given a curve with a 6-pixel radius.

**:displace_left** » a number

For: *all slots and elements*.

Moves a shape, text block or any other kind of object to the left or right. A positive number displaces to the right by the given number of pixels; a negative number displaces to the left. Displacing an object doesn't effect the actual layout of the page. Before using this style, be sure to read the displace docs, since its behavior can be a bit surprising.

**:displace_top** » a number

For: *all slots and elements*.

Moves a shape, text block or any other kind of object up or down. A positive number moves the object down by this number of pixels; a negative number moves it up. Displacing doesn't effect the actual layout of the page or the object's true coordinates. Read the displace docs, since its behavior can be a bit surprising.

**:emphasis** » a string

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*.

Styles the text with an emphasis (commonly italicized.)

This style recognizes three possible settings:

- "normal" - the font is upright.
- "oblique" - the font is slanted, but in a roman style.
- "italic" - the font is slanted in an italic style.

**:family** » a string

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*.

Styles the text with a given font family. The string should contain the family name or a comma-separated list of families.

**:fill** » a hex code, a Shoes::Color or a range of either

For: *arc, arrow, background, banner, caption, code, del, em, flow, image, ins, inscription, line, link, mask, oval, para, rect, shape, span, stack, star, strong, sub, sup, subtitle, tagline, title*.

The color of the background pen. For shapes, this is the fill color, the paint inside the shape. For text stuffs, this color is painted in the background (as if marked with a highlighter pen.)

**:font** » a string

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*.

Styles the text with a font description. The string is pretty flexible, but can take the form "[FAMILY-LIST] [STYLE-OPTIONS] [SIZE]", where FAMILY-LIST is a comma separated list of families optionally terminated by a comma, STYLE_OPTIONS is a whitespace separated list of words where each WORD describes one of style, variant, weight, stretch, or gravity, and SIZE is a decimal number (size in points) or optionally followed by the unit modifier "px" for absolute size. Any one of the options may be absent. If FAMILY-LIST is absent, then the default font family (Arial) will be used.

**:group** » a string

For: *radio*.

Indicates what group a radio button belongs to. Without this setting, radio buttons are grouped together with other radio buttons in their immediate slot. "Grouping" radio buttons doesn't mean they'll be grouped next to each other on the screen. It means that only one radio button from the group can be selected at a time.

By giving this style a string, the radio button will be grouped with other radio buttons that have the same group name.

**:height** » a number

For: *all slots and elements*.

Sets the pixel height of this object. If the number is a decimal number, the height becomes a percentage of its parent's height (with 0.0 being 0% and 1.0 being 100%.)

**:hidden** » true or false

For: *all slots and elements*.

Hides or shows this object. Any object with `:hidden => true` are not displayed on the screen. Neither are its children.

**:inner** » a number

For: *star*.

The size of the inner radius (in pixels.) The inner radius describes the solid circle within the star where the points begin to separate.

**:items** » an array

For: *list_box*.

The list of selections in the list box. See the list_box method for an example.

**:justify** » true or false

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*

Evenly spaces the text horizontally.

**:kerning** » a number

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*.

Adds to the natural spacing between letters, in pixels.

**:leading** » a number

For: *banner, caption, inscription, para, subtitle, tagline, title*.

Sets the spacing between lines in a text block. Defaults to 4 pixels.

**:left** » a number

For: *all slots and elements*.

Sets the left coordinate of this object to a specific pixel. Setting `:left => 10` places the object's left edge ten pixels away from the left edge of the slot containing it. If this style is left unset (or set to `nil`,) the object will flow in with the other objects surrounding it.

**:margin** » a number or an array of four numbers

For: *all slots and elements*.

Margins space an element out from its surroundings. Each element has a left, top, right, and bottom margin. If the `:margin` style is set to a single number, the spacing around the element uniformly matches that number. In other words, if `:margin => 8` is set, all the margins around the element are set to eight pixels in length.

This style can also be given an array of four numbers in the form `[left, top, right, bottom]`.

**:margin_bottom** » a number

For: *all slots and elements*.

Sets the bottom margin of the element to a specific pixel size.

**:margin_left** » a number

For: *all slots and elements*.

Sets the left margin of the element to a specific pixel size.

**:margin_right** » a number

For: *all slots and elements*.

Sets the right margin of the element to a specific pixel size.

**:margin_top** » a number

For: *all slots and elements*.

Sets the top margin of the element to a specific pixel size.

**:outer** » a number

For: *star*.

Sets the outer radius (half of the *total* width) of the star, in pixels.

**:points** » a number

For: *star*.

How many points does this star have? A style of `:points => 5` creates a five-pointed star.

**:radius** » a number

For: *arc, arrow, background, border, gradient, oval, rect, shape*.

Sets the radius (half of the diameter or total width) for each of these elements. Setting this is equivalent to setting both `:width` and `:height` to double this number.

**:right** » a number

For: *all slots and elements*.

Sets the pixel coordinate of an element's right edge. The edge is placed relative to its container's rightmost edge. So, `:right => 0` will align the element so that its own right edge and the right edge of its slot touch. Whereas `:right => 20` will position the right edge of the element off to the left of its slot's right edge by twenty pixels.

**:rise** » a number

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*.

Lifts or plunges the font baseline for some text. For example, a sup has a `:rise` of 10 pixels. Conversely, the sub element has a `:rise` of -10 pixels.

**:scroll** » true or false

For: *flow, stack*.

Establishes this slot as a scrolling slot. If `:scroll => true` is set, the slot will show a scrollbar if any of its contents go past its height. The scrollbar will appear and disappear as needed. It will also appear inside the width of the slot, meaning the slot's width will never change, regardless of whether there is a scrollbar or not.

**:secret** » true or false

For: *ask, edit_line*.

Used for password fields, this setting keeps any characters typed in from becoming visible on the screen. Instead, a replacement character (such as an asterisk) is show for each letter typed.

**:size** » a number

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*.

Sets the pixel size for the font used inside this text block or text fragment.

Font size may also be augmented, through use of the following strings:

- "xx-small" - 57% of present size.
- "x-small" - 64% of present size.
- "small" - 83% of present size.
- "medium" - no change in size.
- "large" - 120% of present size.
- "x-large" - 143% of present size.
- "xx-large" - 173% of present size.

**:state** » a string

For: *button, check, edit_box, edit_line, list_box, radio*.

The `:state` style is for disabling or locking certain controls, if you don't want them to be edited.

Here are the possible style settings:

- nil - the control is active and editable.
- "readonly" - the control is active but cannot be edited.
- "disabled" - the control is not active (grayed out) and cannot be edited.

**:stretch** » a string

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*.

Sets the font stretching used for a text object.

Possible settings are:

- "condensed" - a smaller width of letters.
- "normal" - the standard width of letters.
- "expanded" - a larger width of letters.

**:strikecolor** » a Shoes::Color

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*.

The color used to paint any lines stricken through this text.

**:strikethrough** » a string

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*.

Is this text stricken through? Two options here:

- "none" - no strikethrough
- "single" - a single-line strikethrough.

**:stroke** » a hex code, a Shoes::Color or a range of either

For: *arc, arrow, banner, border, caption, code, del, em, flow, image, ins, inscription, line, link, mask, oval, para, rect, shape, span, stack, star, strong, sub, sup, subtitle, tagline, title*.

The color of the foreground pen. In the case of shapes, this is the color the lines are drawn with. For paragraphs and other text, the letters are printed in this color.

**:strokewidth** » a number

For: *arc, arrow, border, flow, image, line, mask, oval, rect, shape, star, stack*.

The thickness of the stroke, in pixels, of the line defining each of these shapes. For example, the number two would set the strokewidth to 2 pixels.

**:text** » a string

For: *button, edit_box, edit_line*.

Sets the message displayed on a button control, or the contents of an edit_box or edit_line.

**:top** » a number

For: *all slots and elements*.

Sets the top coordinate for an object, relative to its parent slot. If an object is set with `:top => 40`, this means the object's top edge will be placed 40 pixels beneath the top edge of the slot that contains it. If no `:top` style is given, the object is automatically placed in the natural flow of its slot.

**:undercolor** » a Shoes::Color

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*.

The color used to underline text.

**:underline** » a string

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*.

Dictates the style of underline used in the text.

The choices for this setting are:

- "none" - no underline at all.
- "single" - a continuous underline.
- "double" - two continuous parallel underlines.
- "low" - a lower underline, beneath the font baseline. (This is generally recommended only for single characters, particularly when showing keyboard accelerators.)
- "error" - a wavy underline, usually found indicating a misspelling.

**:variant** » a string

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*.

Vary the font for a group of text. Two choices:

- "normal" - standard font.
- "smallcaps" - font with the lower case characters replaced by smaller variants of the capital characters.

**:weight** » a string

For: *banner, caption, code, del, em, ins, inscription, link, para, span, strong, sub, sup, subtitle, tagline, title*.

Set the boldness of the text. Commonly, this style is set to one of the following strings:

- "ultralight" - the ultralight weight (= 200)
- "light" - the light weight (=300)
- "normal" - the default weight (= 400)
- "semibold" - a weight intermediate between normal and bold (=600)
- "bold" - the bold weight (= 700)
- "ultrabold" - the ultrabold weight (= 800)
- "heavy" - the heavy weight (= 900)

However, you may also pass in the numerical weight directly.

**:width** » a number

For: *all slots and elements*.

Sets the pixel width for the element. If the number is a decimal, the width is converted to a percentage (with 0.0 being 0% and 1.0 being 100%.) A width of 100% means the object fills its parent slot.

Next: Classes List

The Shoes Manual

# Classes List

Here is a complete list of all the classes introduced by Shoes. This chart is laid out according to how classes inherits from each other. Subclasses are indented one level to the right, beneath their parent class.

---

- Canvas
  - Shoes
    - Shoes::Flow
    - Shoes::Mask
    - Shoes::Stack
    - Shoes::Widget
- Exception
  - StandardError
    - Shoes::ImageError
    - Shoes::InvalidModeError
    - Shoes::NotImplementedError
    - Shoes::SettingUp
    - Shoes::VideoError
- Shoes::App
  - Shoes::Dialog
- Shoes::Basic
  - Shoes::Background
  - Shoes::Border
  - Shoes::Check
  - Shoes::EditBox
  - Shoes::EditLine
  - Shoes::ListBox
  - Shoes::Progress

    - Shoes::Radio
    - Shoes::Text
        - Shoes::Code
        - Shoes::Del
        - Shoes::Em
        - Shoes::Ins
        - Shoes::Link
        - Shoes::LinkHover
        - Shoes::Span
        - Shoes::Strong
        - Shoes::Sub
        - Shoes::Sup
    - Shoes::TextBlock
        - Shoes::Banner
        - Shoes::Caption
        - Shoes::Inscription
        - Shoes::Para
        - Shoes::Subtitle
        - Shoes::Tagline
        - Shoes::Title
- Shoes::Color
- Shoes::Download
- Shoes::Effect
- Shoes::Image
- Shoes::LinkUrl
- Shoes::Native
    - Shoes::Button
- Shoes::Pattern
- Shoes::Search
- Shoes::Shape

- Shoes::TimerBase
  - Shoes::Animation
  - Shoes::Every
  - Shoes::Timer
- Shoes::Video

Next: Colors List

The Shoes Manual

# Colors List

The following list of colors can be used throughout Shoes. As background colors or border colors. As stroke and fill colors. Most of these colors come from the X11 and HTML palettes.

All of these colors can be used by name. (So calling the `tomato` method from inside any slot will get you a nice reddish color.) Below each color, also find the exact numbers which can be used with the rgb method.

| | |
|---|---|
| **aliceblue** | rgb(240, 248, 255) |
| **antiquewhite** | rgb(250, 235, 215) |
| **aqua** | rgb(0, 255, 255) |
| **aquamarine** | rgb(127, 255, 212) |
| **azure** | rgb(240, 255, 255) |
| **beige** | rgb(245, 245, 220) |
| **bisque** | rgb(255, 228, 196) |
| **black** | rgb(0, 0, 0) |
| **blanchedalmond** | rgb(255, 235, 205) |
| **blue** | rgb(0, 0, 255) |
| **blueviolet** | rgb(138, 43, 226) |
| **brown** | rgb(165, 42, 42) |
| **burlywood** | rgb(222, 184, 135) |
| **cadetblue** | rgb(95, 158, 160) |
| **chartreuse** | rgb(127, 255, 0) |
| **chocolate** | rgb(210, 105, 30) |
| **coral** | rgb(255, 127, 80) |
| **cornflowerblue** | rgb(100, 149, 237) |
| **cornsilk** | rgb(255, 248, 220) |
| **crimson** | rgb(220, 20, 60) |
| **cyan** | rgb(0, 255, 255) |
| **darkblue** | rgb(0, 0, 139) |
| **darkcyan** | rgb(0, 139, 139) |
| **darkgoldenrod** | rgb(184, 134, 11) |
| **darkgray** | rgb(169, 169, 169) |
| **darkgreen** | rgb(0, 100, 0) |
| **darkkhaki** | rgb(189, 183, 107) |
| **darkmagenta** | rgb(139, 0, 139) |
| **darkolivegreen** | rgb(85, 107, 47) |
| **darkorange** | rgb(255, 140, 0) |
| **darkorchid** | rgb(153, 50, 204) |
| **darkred** | rgb(139, 0, 0) |
| **darksalmon** | rgb(233, 150, 122) |
| **darkseagreen** | rgb(143, 188, 143) |
| **darkslateblue** | rgb(72, 61, 139) |
| **darkslategray** | rgb(47, 79, 79) |
| **darkturquoise** | rgb(0, 206, 209) |
| **darkviolet** | rgb(148, 0, 211) |
| **deeppink** | rgb(255, 20, 147) |
| **deepskyblue** | rgb(0, 191, 255) |
| **dimgray** | rgb(105, 105, 105) |
| **dodgerblue** | rgb(30, 144, 255) |
| **firebrick** | rgb(178, 34, 34) |
| **floralwhite** | rgb(255, 250, 240) |
| **forestgreen** | rgb(34, 139, 34) |
| **fuchsia** | rgb(255, 0, 255) |
| **gainsboro** | rgb(220, 220, 220) |
| **ghostwhite** | rgb(248, 248, 255) |
| **gold** | rgb(255, 215, 0) |
| **goldenrod** | rgb(218, 165, 32) |
| **gray** | rgb(128, 128, 128) |
| **green** | rgb(0, 128, 0) |
| **greenyellow** | rgb(173, 255, 47) |
| **honeydew** | rgb(240, 255, 240) |

| | | |
|---|---|---|
| **hotpink**<br>rgb(255, 105, 180) | **indianred**<br>rgb(205, 92, 92) | **indigo**<br>rgb(75, 0, 130) |
| **ivory**<br>rgb(255, 255, 240) | **khaki**<br>rgb(240, 230, 140) | **lavender**<br>rgb(230, 230, 250) |
| **lavenderblush**<br>rgb(255, 240, 245) | **lawngreen**<br>rgb(124, 252, 0) | **lemonchiffon**<br>rgb(255, 250, 205) |
| **lightblue**<br>rgb(173, 216, 230) | **lightcoral**<br>rgb(240, 128, 128) | **lightcyan**<br>rgb(224, 255, 255) |
| **lightgoldenrodyellow**<br>rgb(250, 250, 210) | **lightgreen**<br>rgb(144, 238, 144) | **lightgrey**<br>rgb(211, 211, 211) |
| **lightpink**<br>rgb(255, 182, 193) | **lightsalmon**<br>rgb(255, 160, 122) | **lightseagreen**<br>rgb(32, 178, 170) |
| **lightskyblue**<br>rgb(135, 206, 250) | **lightslategray**<br>rgb(119, 136, 153) | **lightsteelblue**<br>rgb(176, 196, 222) |
| **lightyellow**<br>rgb(255, 255, 224) | **lime**<br>rgb(0, 255, 0) | **limegreen**<br>rgb(50, 205, 50) |
| **linen**<br>rgb(250, 240, 230) | **magenta**<br>rgb(255, 0, 255) | **maroon**<br>rgb(128, 0, 0) |
| **mediumaquamarine**<br>rgb(102, 205, 170) | **mediumblue**<br>rgb(0, 0, 205) | **mediumorchid**<br>rgb(186, 85, 211) |
| **mediumpurple**<br>rgb(147, 112, 219) | **mediumseagreen**<br>rgb(60, 179, 113) | **mediumslateblue**<br>rgb(123, 104, 238) |
| **mediumspringgreen**<br>rgb(0, 250, 154) | **mediumturquoise**<br>rgb(72, 209, 204) | **mediumvioletred**<br>rgb(199, 21, 133) |
| **midnightblue**<br>rgb(25, 25, 112) | **mintcream**<br>rgb(245, 255, 250) | **mistyrose**<br>rgb(255, 228, 225) |
| **moccasin**<br>rgb(255, 228, 181) | **navajowhite**<br>rgb(255, 222, 173) | **navy**<br>rgb(0, 0, 128) |
| **oldlace**<br>rgb(253, 245, 230) | **olive**<br>rgb(128, 128, 0) | **olivedrab**<br>rgb(107, 142, 35) |
| **orange**<br>rgb(255, 165, 0) | **orangered**<br>rgb(255, 69, 0) | **orchid**<br>rgb(218, 112, 214) |
| **palegoldenrod**<br>rgb(238, 232, 170) | **palegreen**<br>rgb(152, 251, 152) | **paleturquoise**<br>rgb(175, 238, 238) |
| **palevioletred**<br>rgb(219, 112, 147) | **papayawhip**<br>rgb(255, 239, 213) | **peachpuff**<br>rgb(255, 218, 185) |

| | | |
|---|---|---|
| **peru**<br>rgb(205, 133, 63) | **pink**<br>rgb(255, 192, 203) | **plum**<br>rgb(221, 160, 221) |
| **powderblue**<br>rgb(176, 224, 230) | **purple**<br>rgb(128, 0, 128) | **red**<br>rgb(255, 0, 0) |
| **rosybrown**<br>rgb(188, 143, 143) | **royalblue**<br>rgb(65, 105, 225) | **saddlebrown**<br>rgb(139, 69, 19) |
| **salmon**<br>rgb(250, 128, 114) | **sandybrown**<br>rgb(244, 164, 96) | **seagreen**<br>rgb(46, 139, 87) |
| **seashell**<br>rgb(255, 245, 238) | **sienna**<br>rgb(160, 82, 45) | **silver**<br>rgb(192, 192, 192) |
| **skyblue**<br>rgb(135, 206, 235) | **slateblue**<br>rgb(106, 90, 205) | **slategray**<br>rgb(112, 128, 144) |
| **snow**<br>rgb(255, 250, 250) | **springgreen**<br>rgb(0, 255, 127) | **steelblue**<br>rgb(70, 130, 180) |
| **tan**<br>rgb(210, 180, 140) | **teal**<br>rgb(0, 128, 128) | **thistle**<br>rgb(216, 191, 216) |
| **tomato**<br>rgb(255, 99, 71) | **turquoise**<br>rgb(64, 224, 208) | **violet**<br>rgb(238, 130, 238) |
| **wheat**<br>rgb(245, 222, 179) | **white**<br>rgb(255, 255, 255) | **whitesmoke**<br>rgb(245, 245, 245) |
| **yellow**<br>rgb(255, 255, 0) | **yellowgreen**<br>rgb(154, 205, 50) | |

Next: Slots

The Shoes Manual

# Slots

Slots are boxes used to lay out images, text and so on. The two most common slots are `stacks` and `flows`. Slots can also be referred to as "boxes" or "canvases" in Shoes terminology.

---

Since the mouse wheel and PageUp and PageDown are so pervasive on every platform, vertical scrolling has really become the only overflow that matters. So, in Shoes, just as on the web, width is generally fixed. While height goes on and on.

Now, you can also just use specific widths and heights for everything, if you want. That'll take some math, but everything could be perfect.

Generally, I'd suggest using stacks and flows. The idea here is that you want to fill up a certain width with things, then advance down the page, filling up further widths. You can think of these as being analogous to HTML's "block" and "inline" styles.

## Stacks

A stack is simply a vertical stack of elements. Each element in a stack is placed directly under the element preceding it.

A stack is also shaped like a box. So if a stack is given a width of 250, that stack is itself an element which is 250 pixels wide.

To create a new stack, use the stack method, which is available inside any slot. So stacks can contain other stacks and flows.

## Flows

A flow will pack elements in as tightly as it can. A width will be filled, then will wrap beneath those elements. Text elements placed next to each other will appear as a single paragraph. Images and widgets will run together as a series.

Like the stack, a flow is a box. So stacks and flows can safely be embedded and, without respect to their contents, are identical. They just treat their contents differently.

Making a flow means calling the flow method. Flows may contain other flows and stacks.

Last thing: The Shoes window itself is a flow.

Next: Art for Slots

The Shoes Manual

# Art for Slots

Each slot is like a canvas, a blank surface which can be covered with an assortment of colored shapes or gradients.

Many common shapes can be drawn with methods like `oval` and `rect`. You'll need to set up the paintbrush colors first, though.

The `stroke` command sets the line color. And the `fill` command sets the color used to paint inside the lines.

```
Shoes.app do
  stroke red
  fill blue
  oval :top => 10, :left => 10,
    :radius => 100
end
```

That code gives you a blue pie with a red line around it. One-hundred pixels wide, placed just a few pixels southeast of the window's upper left corner.

The `blue` and `red` methods above are Color objects. See the section on Colors for more on how to mix colors.

## Inspiration from Processing and NodeBox

The artful methods generally come verbatim from NodeBox, a drawing kit for Python. In turn, NodeBox gets much of its ideas from Processing, a Java-like language for graphics and animation. I owe a great debt to the creators of these wonderful programs!

Shoes does a few things differently from NodeBox and Processing. For example, Shoes has different color methods, including having its own Color objects, though these are very similar to Processing's color methods. And Shoes also allows images and gradients to be used for drawing lines and filling in shapes.

Shoes also borrows some animation ideas from Processing and will continue to closely consult Processing's methods as it expands.

### **arc(left, top, width, height, angle1, angle2)** » Shoes::Shape

Draws an arc shape (a section of an oval) at coordinates (left, top). This method just give you a bit more control than oval, by offering the `:angle1` and `:angle2` styles. (In fact, you can mimick the `oval` method by setting `:angle1` to 0 and `:angle2` to `Shoes::TWO_PI`.)

### **arrow(left, top, width)** » Shoes::Shape

Draws an arrow at coordinates (left, top) with a pixel `width`.

### **cap(:curve or :rect or :project)** » self

Sets the line cap, which is the shape at the end of every line you draw. If set to `:curve`, the end is rounded. The default is `:rect`, a line which ends abruptly flat. The `:project` cap is also fat, but sticks out a bit longer.

### **fill(pattern)** » pattern

Sets the fill bucket to a specific color (or pattern.) Patterns can be colors, gradients or images. So, once the fill bucket is set, you can draw shapes and they will be colored in with the pattern you've chosen.

To draw a star with an image pattern:

```
Shoes.app do
  fill "images/shiny.png"
  star 200, 200, 5
end
```

To clear the fill bucket, use `nofill`. And to set the line color (the border of the star,) use the `stroke` method.

### **nofill()** » self

Blanks the fill color, so that any shapes drawn will not be filled in. Instead, shapes will have only a lining, leaving the middle transparent.

### **nostroke()** » self

Empties the line color. Shapes drawn will have no outer line. If `nofill` is also set, shapes drawn will not be visible.

### **line(left, top, x2, y2)** » Shoes::Shape

Draws a line using the current line color (aka "stroke") starting at coordinates (left, top) and ending at coordinates (x2, y2).

### **oval(left, top, radius)** » Shoes::Shape

Draws a circular form at pixel coordinates (left, top) with a width and height of `radius` pixels. The line and fill colors are used to draw the shape. By default, the coordinates are for the oval's leftmost, top corner, but this can be changed by calling the transform method or by using the `:center` style on the next method below.

```
Shoes.app do
  stroke blue
  strokewidth 4
  fill black
  oval 10, 10, 50
```

```
end
```

To draw an oval of varied proportions, you may also use the syntax: `oval(left, top, width, height)`.

**oval(styles)** » Shoes::Shape

Draw circular form using a style hash. The following styles are supported:

- `top`: the y-coordinate for the oval pen.
- `left`: the x-coordinate for the oval pen.
- `radius`: the width and height of the circle.
- `width`: a specific pixel width for the oval.
- `height`: a specific pixel height for the oval.
- `center`: do the coordinates specific the oval's center? (true or false)

These styles may also be altered using the `style` method on the Shape object.

**rect(top, left, width, height, corners = 0)** » Shoes::Shape

Draws a rectangle starting from coordinates (top, left) with dimensions of width x height. Optionally, you may give the rectangle rounded corners with a fifth argument: the radius of the corners in pixels.

As with all other shapes, the rectangle is drawn using the stroke and fill colors.

```
Shoes.app do
  stroke rgb(0.5, 0.5, 0.7)
  fill rgb(1.0, 1.0, 0.9)
  rect 10, 10, self.width - 10, self.height - 10
end
```

The above sample draws a rectangle which fills the area of its parent box, leaving a margin of 10 pixels around the edge. Also see the `background` method for a rectangle which defaults to filling its parent box.

**rect(styles)** » Shoes::Shape

Draw a rectangle using a style hash. The following styles are supported:

- `top`: the y-coordinate for the rectangle.
- `left`: the x-coordinate for the rectangle.
- `curve`: the pixel radius of the rectangle's corners.
- `width`: a specific pixel width for the rectangle.
- `height`: a specific pixel height for the rectangle.
- `center`: do the coordinates specific the rectangle's center? (true or false)

These styles may also be altered using the `style` method on the Shape object.

### **rotate(degrees: a number)** » self

Rotates the pen used for drawing by a certain number of `degrees`, so that any shapes will be drawn at that angle.

In this example below, the rectangle drawn at (30, 30) will be rotated 45 degrees.

```
Shoes.app do
  fill "#333"
  rotate 45
  rect 30, 30, 40, 40
end
```

### **shape(left, top) { ... }** » Shoes::Shape

Describes an arbitrary shape to draw, beginning at coordinates (left, top) and continued by calls to `line_to`, `move_to`, `curve_to` and `arc_to` inside the block. You can look at it as sketching a shape with a long line that curves and arcs and bends.

```
Shoes.app do
  fill red(0.2)
  shape do
    move_to(90, 55)
    arc_to(50, 55, 50, 50, 0, PI/2)
    arc_to(50, 55, 60, 60, PI/2, PI)
    arc_to(50, 55, 70, 70, PI, TWO_PI-PI/2)
    arc_to(50, 55, 80, 80, TWO_PI-PI/2, TWO_PI)
  end
end
```

A shape can also contain other shapes. So, you can place an oval, a rect, a line, a star or an arrow (and all of the other methods in this Art section) inside a shape, but they will not be part of the line. They will be more like a group of shapes are all drawn as one.

### **star(left, top, points = 10, outer = 100.0, inner = 50.0)** » Shoes::Shape

Draws a star using the stroke and fill colors. The star is positioned with its center point at coordinates (left, top) with a certain number of `points`. The `outer` width defines the full radius of the star; the `inner` width specifies the radius of the star's middle, where points stem from.

### **stroke(pattern)** » pattern

Set the active line color for this slot. The `pattern` may be a color, a gradient or an image, all of which are categorized as "patterns." The line color is then used to draw the borders of any subsequent shape.

So, to draw an arrow with a red line around it:

```
Shoes.app do
  stroke red
  arrow 0, 100, 10
```

```
end
```

To clear the line color, use the `nostroke` method.

**strokewidth(a number)** » self

Sets the line size for all drawing within this slot. Whereas the `stroke` method alters the line color, the `strokewidth` method alters the line size in pixels. Calling `strokewidth(4)` will cause lines to be drawn 4 pixels wide.

**transform(:center or :corner)** » self

Should transformations (such as `skew` and `rotate`) be performed around the center of the shape? Or the corner of the shape? Shoes defaults to `:corner`.

**translate(left, top)** » self

Moves the starting point of the drawing pen for this slot. Normally, the pen starts at (0, 0) in the top-left corner, so that all shapes are drawn from that point. With `translate`, if the starting point is moved to (10, 20) and a shape is drawn at (50, 60), then the shape is actually drawn at (60, 80) on the slot.

Next: Element Creation

The Shoes Manual

# Element Creation

Shoes has a wide variety of elements, many cherry-picked from HTML. This page describes how to create these elements in a slot. See the Elements section of the manual for more on how to modify and use these elements after they have been placed.

**animate(fps) { |frame| ... }** » Shoes::Animation

Starts an animation timer, which runs parallel to the rest of the app. The `fps` is a number, the frames per seconds. This number dictates how many times per second the attached block will be called.

The block is given a `frame` number. Starting with zero, the `frame` number tells the block how many frames of the animation have been shown.

```
Shoes.app do
  @counter = para "STARTING"
  animate(24) do |frame|
    @counter.replace "FRAME #{frame}"
  end
end
```

The above animation is shown 24 times per second. If no number is give, the `fps` defaults to 10.

**background(pattern)** » Shoes::Background

Draws a Background element with a specific color (or pattern.) Patterns can be colors, gradients or images. Colors and images will tile across the background. Gradients stretch to fill the background.

**PLEASE NOTE:** Backgrounds are actual elements, not styles. HTML treats backgrounds like styles. Which means every box can only have one background. Shoes layers background elements.

```
Shoes.app do
  background black
  background white, :width => 50
end
```

The above example paints two backgrounds. First, a black background is painted over the entire app's surface area. Then a 50 pixel white stripe is painted along the left side.

**banner(text)** » Shoes::Banner

Creates a Banner text block. Shoes automatically styles this text to 48 pixels high.

**border(text, :strokewidth => a number)** » Shoes::Border

Draws a Border element using a specific color (or pattern.) Patterns can be colors, gradients or images. Colors and images will tile across the border. Gradients stretch to fill the border.

**PLEASE NOTE:** Like Backgrounds, Borders are actual elements, not styles. HTML treats backgrounds and borders like styles. Which means every box can only have one borders. Shoes layers border and background elements, along with text blocks, images, and everything else.

**button(text) { ... }** » Shoes::Button

Adds a push button with the message `text` written across its surface. An optional block can be attached, which is called if the button is pressed.

**caption(text)** » Shoes::Caption

Creates a Caption text block. Shoes styles this text to 14 pixels high.

**check()** » Shoes::Check

Adds a check box.

**code(text)** » Shoes::Code

Create a Code text fragment. This text defaults to a monospaced font.

**del(text)** » Shoes::Del

Creates a Del text fragment (short for "deleted") which defaults to text with a single strikethrough in its middle.

**dialog(styles) { ... }** » Shoes::App

Opens a new app window (just like the window method does,) but the window is given a dialog box look.

**edit_box(text)** » Shoes::EditBox

Adds a large, multi-line textarea to this slot. The `text` is optional and should be a string that will start out the box. An optional block can be attached here which is called any type the user changes the text in the box.

```
Shoes.app do
  edit_box
  edit_box "HORRAY EDIT ME"
  edit_box "small one", :width => 100, :height => 160
end
```

**edit_line(text)** » Shoes::EditLine

Adds a single-line text box to this slot. The `text` is optional and should be a string that will start out the box. An optional block can be attached here which is called any type the user changes the text in the box.

**em(text)** » Shoes::Em

Creates an Em text fragment (short for "emphasized") which, by default, is styled with italics.

**every(seconds) { |count| ... }** » Shoes::Every

A timer similar to the `animation` method, but much slower. This timer fires a given number of seconds, running the block attached. So, for example, if you need to check a web site every five minutes, you'd call `every(300)` with a block containing the code to actually ping the web site.

**flow(styles) { ... }** » Shoes::Flow

A flow is an invisible box (or "slot") in which you place Shoes elements. Both flows and stacks are explained in great detail on the main Slots page.

Flows organize elements horizontally. Where one would use a stack to keep things stacked vertically, a flow places its contents end-to-end across the page. Once the end of the page is reached, the flow starts a new line of elements.

**image(path)** » Shoes::Image

Creates an Image element for displaying a picture. PNG, JPEG and GIF formats are allowed.

The `path` can be a file path or a URL. All images loaded are temporarily cached in memory, but remote images are also cached locally in the user's personal Shoes directory. Remote images are loaded in the background; as with browsers, the images will not appear right away, but will we shown when they are loaded.

**imagesize(path)** » [width, height]

Quickly grab the width and height of an image. The image won't be loaded into the cache or displayed.

URGENT NOTE: This method cannot be used with remote images (loaded from HTTP, rather than the hard drive.)

**ins(text)** » Shoes::Ins

Creates an Ins text fragment (short for "inserted") which Shoes styles with a single underline.

**inscription(text)** » Shoes::Inscription

Creates an Inscription text block. Shoes styles this text at 10 pixels high.

**link(text, :click => proc or string)** » Shoes::Link

Creates a Link text block, which Shoes styles with a single underline and colors with a #06E (blue) colored stroke.

The default LinkHover style is also single-underlined with a #039 (dark blue) stroke.

**list_box(:items => [strings, ...])** » Shoes::ListBox

Adds a drop-down list box containing entries for everything in the `items` array. An optional block may be attached, which is called if anything in the box becomes selected by the user.

```
Shoes.app do
  stack :margin => 10 do
    para "Pick a card:"
    list_box :items => ["Jack", "Ace", "Joker"]
  end
end
```

Call `ListBox#text` to get the selected string. See the `ListBox` section under `Native` controls for more help.

**progress()** » Shoes::Progress

Adds a progress bar.

**para(text)** » Shoes::Para

Create a Para text block (short for "paragraph") which Shoes styles at 12 pixels high.

**radio(group name: a string or symbol)** » Shoes::Radio

Adds a radio button. If a `group name` is given, the radio button is considered part of a group. Among radio buttons in the same group, only one may be checked. (If no group name is given, the radio button is grouped with any other radio buttons in the same slot.)

**stack(styles) { ... }** » Shoes::Stack

Creates a new stack. A stack is a type of slot. (See the main Slots page for a full explanation of both stacks and flows.)

In short, stacks are an invisible box (a "slot") for placing stuff. As you add things to the stack, such as buttons or images, those things pile up vertically. Yes, they stack up!

**strong(text)** » Shoes::Strong

Creates a Strong text fragment, styled in bold by default.

**sub(text)** » Shoes::Sub

Creates a Sub text fragment (short for "subscript") which defaults to lowering the text by 10 pixels and styling it in an x-small font.

**subtitle(text)** » Shoes::Subtitle

Creates a Subtitle text block. Shoes styles this text to 26 pixels high.

**sup(text)** » Shoes::Sup

Creates a Sup text fragment (short for "superscript") which defaults to raising the text by 10 pixels and styling it in an x-small font.

**tagline(text)** » Shoes::Tagline

Creates a Tagline text block. Shoes styles this text to 18 pixels high.

**timer(seconds) { ... }** » Shoes::Timer

A one-shot timer. If you want to schedule to run some code in a few seconds (or minutes, hours) you can attach the code as a block here.

To display an alert box five seconds from now:

```
Shoes.app do
  timer(5) do
    alert("Your five seconds are up.")
  end
end
```

**title(text)** » Shoes::Title

Creates a Title text block. Shoes styles these elements to 34 pixels high.

**video(path or url)** » Shoes::Video

Embeds a movie in this slot.

**window(styles) { ... }** » Shoes::App

Opens a new app window. This method is almost identical to the Shoes.app method used to start an app in the first place. The difference is that the `window` method sets the new window's owner property. (A normal Shoes.app has its `owner` set to `nil`.)

So, the new window's `owner` will be set to the Shoes::App which launched the window. This way the child window can call the parent.

```
Shoes.app :title => "The Owner" do
  button "Pop up?" do
    window do
      para "Okay, popped up from #{owner}"
    end
  end
end
```

Next: Events

The Shoes Manual

# Events

Wondering how to catch stray mouse clicks or keyboard typing? Events are sent to a slot whenever a mouse moves inside the slot. Or whenever a key is pressed. Even when the slot is created or destroyed. You can attach a block to each of these events.

Mouse events include `motion`, `click`, `hover` and `leave`. Keyboard typing is represented by the `keypress` event. And the `start` and `finish` events indicate when a canvas comes into play or is discarded.

So, let's say you want to change the background of a slot whenever the mouse floats over it. We can use the `hover` event to change the background when the mouse comes inside the slot. And `leave` to change back when the mouse floats away.

```
Shoes.app do
  stack :width => 200, :height => 200 do
    background red
    hover do
      clear { background blue }
    end
    leave do
      clear { background red }
    end
  end
end
```

### **click { |button, left, top| ... }** » self

The click block is called when a mouse button is clicked. The `button` is the number of the mouse button which has been pressed. The `left` and `top` are the mouse coordinates at which the click happened.

To catch the moment when the mouse is unclicked, see the release event.

### **finish { |self| ... }** » self

When a slot is removed, it's finish event occurs. The finish block is immediately handed `self`, the slot object which has been removed.

### **hover { |self| ... }** » self

The hover event happens when the mouse enters the slot. The block gets `self`, meaning the object which was hovered over.

To catch the mouse exiting the slot, check out the leave event.

### **keypress { |key| ... }** » self

Whenever a key (or combination of keys) is pressed, the block gets called. The block is sent a `key` which is a string representing the character (such as the letter or number) on the key. For special keys and key combos, a Ruby symbol is sent, rather than a string.

So, for example, if `Shift-a` is pressed, the block will get the string `"A"`.

However, if the F1 key is pressed, the `:f1` symbol is received. For `Shift-F1`, the symbol would be `:shift_f1`.

The modifier keys are `control`, `shift` and `alt`. They appear in that order. If `Shift-Control-Alt-PgUp` is pressed, the symbol will be `:control_shift_alt_page_up`.

One thing about the shift key. You won't see the shift key on most keys. On US keyboards, `Shift-7` is an ampersand. So you'll get the string `"&"` rather than `:shift_5`. And, if you press `Shift-Alt-7` on such a keyboard, you'll get the symbol: `:alt_&`. You'll only see the shift modifier on the special keys listed a few paragraphs down.

```
Shoes.app do
  @info = para "NO KEY is PRESSED."
  keypress do |k|
    @info.replace "#{k.inspect} was PRESSED."
  end
end
```

Keep in mind that Shoes itself uses a few hotkeys. Alt-Period (`:alt_.`), Alt-Question (`:alt_?`) and Alt-Slash (`:alt_/`) are reserved for Shoes.

The list of special keys is as follows: `:escape`, `:delete`, `:backspace`, `:tab`, `:page_up`, `:page_down`, `:home`, `:end`, `:left`, `:up`, `:right`, `:down`, `:f1`, `:f2`, `:f3`, `:f4`, `:f5`, `:f6`, `:f7`, `:f8`, `:f9`, `:f10`, `:f11` and `:f12`.

One caveat to all of those rules: normally the Return key gives you a string `"\n"`. When pressed with modifier keys, however, you end up with `:control_enter`, `:control_alt_enter`, `:shift_alt_enter` and the like.

### **leave { |self| ... }** » self

The leave event takes place when the mouse cursor exits a slot. The moment it no longer is inside the slot's edges. When that takes place, the block is called with `self`, the slot object which is being left.

Also see hover if you'd like to detect the mouse entering a slot.

### **motion { |left, top| ... }** » self

The motion block gets called every time the mouse moves around inside the slot. The block is handed the cursor's `left` and `top` coordinates.

```
Shoes.app :width => 200, :height => 200 do
```

```
  background black
  fill white
  @circ = oval 0, 0, 100, 100
  motion do |top, left|
    @circ.move top - 50, left - 50
  end
end
```

### **release { |button, left, top| ... }** » self

The release block runs whenever the mouse is unclicked (on mouse up). When the finger is lifted. The `button` is the number of the button that was depressed. The `left` and `top` are the coordinates of the mouse at the time the button was released.

To catch the actual mouse click, use the click event.

### **start { |self| ... }** » self

The first time the slot is drawn, the start event fires. The block is handed `self`, the slot object which has just been drawn.

Next: Manipulation Blocks

The Shoes Manual

# Manipulation Blocks

## The manipulation methods below make quick work of shifting around slots and inserting new elements.

### **append() { ... }** » self

Adds elements to the end of a slot.

```
@slot.append do
  title "Breaking News"
  tagline "Astronauts arrested for space shuttle DUI."
end
```

The `title` and `tagline` elements will be added to the end of the `@slot`.

### **after(element) { ... }** » self

Adds elements to a specific place in a slot, just after the `element` which is a child of the slot.

### **before(element) { ... }** » self

Adds elements to a specific place in a slot, just before the `element` which is a child of the slot.

### **clear()** » self

Empties the slot of any elements, timers and nested slots. This is effectively identical to looping through the contents of the slot and calling each element's `remove` method.

### **clear() { ... }** » self

The clear method also takes an optional block. The block will be used to replace the contents of the slot.

```
@slot = stack { para "Old text" }
@slot.clear { para "Brand new text" }
```

In this example, the "Old text" paragraph will be cleared out, replaced by the "Brand new text" paragraph.

### **prepend() { ... }** » self

Adds elements to the beginning of a slot.

```
@slot.prepend do
```

```
  para "Your car is ready."
end
```

The `para` element is added to the beginning of the `@slot`.

Next: Position of a Slot

The Shoes Manual

# Position of a Slot

Like any other element, slots can be styled and customized when they are created.

---

To set the width of a stack to 150 pixels:

```
stack(:width => 150) { para "Now that's precision." }
```

Each style setting also has a method, which can be used to grab that particular setting. (So, like, the `width` method returns the width of the slot in pixels.)

**displace(left: a number, top: a number)** » self

A shortcut method for setting the :displace_left and :displace_top styles. Displacing is a handy way of moving a slot without altering the layout. In fact, the `top` and `left` methods will not report displacement at all. So, generally, displacement is only for temporary animations. For example, jiggling a button in place.

The `left` and `top` numbers sent to `displace` are added to the slot's own top-left coordinates. To subtract from the top-left coordinate, use negative numbers.

**gutter()** » a number

The size of the scrollbar area. When Shoes needs to show a scrollbar, the scrollbar may end up covering up some elements that touch the edge of the window. The `gutter` tells you how many pixels to expect the scrollbar to cover.

This is commonly used to pad elements on the right, like so:

```
stack :margin_right => 20 + gutter do
  para "Insert fat and ratified declaration of
    independence here..."
end
```

**height()** » a number

The vertical size of the viewable slot in pixels. So, if this is a scrolling slot, you'll need to use `scroll_height()` to get the full size of the slot.

**hide()** » self

Hides the slot, so that it can't be seen. See also show and toggle.

**left()** » a number

The left pixel location of the slot. Also known as the x-axis coordinate.

**move(left, top)** » self

Moves the slot to specific coordinates, the (left, top) being the upper left hand corner of the slot.

**remove()** » self

Removes the slot. It will no longer be displayed and will not be listed in its parent's contents. It's gone.

**scroll()** » true or false

Is this slot allowed to show a scrollbar? True or false. The scrollbar will only appear if the height of the slot is also fixed.

**scroll_height()** » a number

The vertical size of the full slot, including any of it which is hidden by scrolling.

**scroll_max()** » a number

The top coordinate which this slot can be scrolled down to. The top coordinate of a scroll bar is always zero. The bottom coordinate is the full height of the slot minus one page of scrolling. This bottom coordinate is what `scroll_max` returns.

This is basically a shortcut for writing `slot.scroll_height - slot.height`.

To scroll to the bottom of a slot, use `slot.scroll_top = slot.scroll_max`.

**scroll_top()** » a number

The top coordinate which this slot is scrolled down to. So, if the slot is scrolled down twenty pixels, this method will return `20`.

**scroll_top = a number**

Scrolls the slot to a certain coordinate. This must be between zero and `scroll_max`.

**show()** » self

Reveals the slot, if it is hidden. See also hide and toggle.

**style()** » styles

Calling the `style` method with no arguments returns a hash of the styles presently applied to this slot.

While methods such as `height` and `width` return the true pixel dimensions of the slot, you can use `style[:height]` or `style[:width]` to get the dimensions originally requested.

```
Shoes.app do
  @s = stack :width => "100%"
  para @s.style[:width]
end
```

In this example, the paragraph under the stack will display the string "100%".

**style(styles)** » styles

Alter the slot using a hash of style settings. Any of the methods on this page (aside from this method, of course) can be used as a style setting. So, for example, there is a `width` method, thus there is also a `width` style.

```
Shoes.app do
  @s = stack
  @s.style(:width => 400)
end
```

**toggle()** » self

Hides the slot, if it is shown. Or shows the slot, if it is hidden.

**top()** » a number

The top pixel location of the slot. Also known as the y-axis coordinate.

**width()** » a number

The horizontal size of the slot in pixels.

Next: Traversing the Page

The Shoes Manual

# Traversing the Page

You may find yourself needing to loop through the elements inside a slot. Or maybe you need to climb the page, looking for a stack that is the parent of an element.

On any element, you may call the `parent` method to get the slot directly above it. And on slots, you can call the `contents` method to get all of the children. (Some elements, such as text blocks, also have a `contents` method for getting their children.)

**contents()** » an array of elements

Lists all elements in a slot.

**parent()** » a Shoes::Stack or Shoes::Flow

Gets the object for this element's container.

Next: Elements

The Shoes Manual

# Elements

Ah, here's the stuff of Shoes. An element can be as simple as an oval shape. Or as complex as a video stream. You've encountered all of these elements before in the Slots section of the manual.

---

Shoes has seven native controls: the Button, the EditLine, the EditBox, the ListBox, the Progress meter, the Check box and the Radio. By "native" controls, we mean that each of these seven elements is drawn by the operating system. So, a Progress bar will look one way on Windows and another way on OS X.

Shoes also has seven basic other types of elements: Background, Border, Image, Shape, TextBlock, Timer and Video. These all should look and act the same on every operating system.

Once an element is created, you will often still want to change it. To move it or hide it or get rid of it. You'll use the commands in this section to do that sort of stuff. (Especially check out the Common Methods section for commands you can use on any element.)

So, for example, use the `image` method of a Slot to place a PNG on the screen. The `image` method gives you back an Image object. Use the methods of the Image object to change things up.

---

Next: Common Methods

The Shoes Manual

# Common Methods

A few methods are shared by every little element in Shoes. Moving, showing, hiding. Removing an element. Basic and very general things. This list encompasses those common commands.

---

One of the most general methods of all is the `style` method (which is also covered as the style method for slots.)

```
Shoes.app do
  stack do
    # Background, text and a button: both are elements!
    @back  = background green
    @text  = banner "A Message for You, Rudy"
    @press = button "Stop your messin about!"
    # And so, both can be styled.
    @text.style :size => 12, :stroke => red, :margin => 10
    @press.style :width => 400
    @back.style :height => 10
  end
end
```

For specific commands, see the other links to the left in the Elements section. Like if you want to pause or play a video file, check the Video section, since pausing and playing is peculiar to videos. No sense pausing a button.

**displace(left: a number, top: a number)** » self

Displacing an element moves it. But without changing the layout around it. This is great for subtle animations, especially if you want to reserve a place for an element while it is still animating. Like maybe a quick button shake or a slot sliding into view.

When you displace an element, it moves relative to the upper-left corner where it was placed. So, if an element is at the coordinates (20, 40) and you displace it 2 pixels left and 6 pixels on top, you end up with the coordinates (22, 46).

```
Shoes.app do
  flow :margin => 12 do
    # Set up three buttons
    button "One"
    @two = button "Two"
    button "Three"
    # Bounce the second button
    animate do |i|
      @two.displace(0, (Math.sin(i) * 6).to_i)
    end
  end
```

```
end
```

Notice that while the second button bounces, the other two buttons stay put. If we used a normal `move` in this situation, the second button would be moved out of the layout and the buttons would act as if the second button wasn't there at all. (See the move example.)

**Of particular note:** if you use the `left` and `top` methods to get the coordinates of a displaced element, you'll just get back the normal coordinates. As if there was no displacement. Displacing is just intended for quick animations!

### **height()** » a number

The vertical screen size of the element in pixels. In the case of images, this is not the full size of the image. This is the height of the element as it is shown right now.

If you have a 150x150 pixel image and you set the width to 50 pixels, this method will return 50.

Also see the width method for an example and some other comments.

### **hide()** » self

Hides the element, so that it can't be seen. See also show and toggle.

### **left()** » a number

Gets you the pixel position of the left edge of the element.

### **move(left: a number, top: a number)** » self

Moves the element to a specific pixel position within its slot. The element is still inside the slot. But it will no longer be stacked or flowed in with the other stuff in the slot. The element will float freely, now absolutely positioned instead.

```
Shoes.app do
  flow :margin => 12 do
    # Set up three buttons
    button "One"
    @two = button "Two"
    button "Three"
    # Bounce the second button
    animate do |i|
      @two.move(40, 40 + (Math.sin(i) * 6).to_i)
    end
  end
end
```

The second button is moved to a specific place, allowing the third button to slide over into its place. If you want to move an element without shifting other pieces, see the displace method.

### **parent()** » a Shoes::Stack or Shoes::Flow

Gets the object for this element's container. Also see the slot's contents to do the opposite: get a container's elements.

**remove()** » self

Removes the element from its slot. (In other words: throws it in the garbage.) The element will no longer be displayed.

**show()** » self

Reveals the element, if it is hidden. See also hide and toggle.

**style()** » styles

Gives you the full set of styles applied to this element, in the form of a Hash. While methods like `width` and `height` and `top` give you back specific pixel dimensions, using `style[:width]` or `style[:top]`, you can get the original setting (things like "100%" for width or "10px" for top.)

```
Shoes.app do
  # A button which take up the whole page
  @b = button "All of it", :width => 1.0, :height => 1.0
  # When clicked, show the styles
  @b.click { alert(@b.style.inspect) }
end
```

**style(styles)** » styles

Changes the style of an element. This could include the `:width` and `:height` of an element, the font `:size` of some text, the `:stroke` and `:fill` of a shape. Or any other number of style settings.

**toggle()** » self

Hides an element if it is shown. Or shows the element, if it is hidden.

**top()** » a number

Gets the pixel position of the top edge of the element.

**width()** » a number

Gets the pixel width for the full size of the element. This method always returns an exact pixel size. In the case of images, this is not the full width of the image, just the size it is shown at. See the height method for more.

Also, if you create an element with a width of 100% and that element is inside a stack which is 120 pixels wide, you'll get back `120`. However, if you call `style[:width]`, you'll get `"100%"`.

```
Shoes.app do
  stack :width => 120 do
    @b = button "Click me", :width => "100%" do
      alert "button.width = #{@b.width}\n" +
        "button.style[:width] = #{@b.style[:width]}"
    end
  end
end
```

In order to set the width, you'll have to go through the style method again. So, to set the button to 150 pixels wide: `@b.style(:width => 150)`.

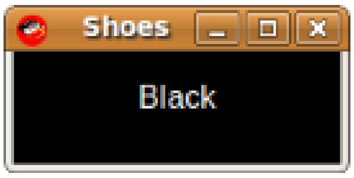
To let Shoes pick the element's width, go with `@b.style(:width => nil)` to empty out the setting.

Next: Background

The Shoes Manual

# Background

A background is a color, a gradient or an image that is painted across an entire slot. Both backgrounds and borders are a type of Shoes::Pattern.



Even though it's called a *background*, you may still place this element in front of other elements. If a background comes after something else painted on the slot (like a `rect` or an `oval`,) the background will be painted over that element.

The simplest background is just a plain color background, created with the background method, such as this black background:

```
Shoes.app do
  background black
end
```

A simple background like that paints the entire slot that contains it. (In this case, the whole window is painted black.)

You can use styles to cut down the size or move around the background to your liking.

To paint a black background across the top fifty pixels of the window:

```
Shoes.app do
  background black, :height => 50
end
```

Or, to paint a fifty pixel column on the right-side of the window:

```
Shoes.app do
  background black, :width => 50, :right => 50
end
```

Since Backgrounds are normal elements as well, see also the start of the Elements section for all of its other methods.

**to_pattern()** » a Shoes::Pattern

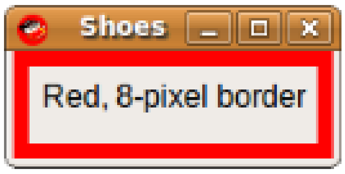
Yanks out the color, gradient or image used to paint this background and places it in a normal Shoes::Pattern object. You can then pass that object to other backgrounds and borders. Reuse it as you like.

Next: Border

The Shoes Manual

# Border

A border is a color, gradient or image painted in a line around the edge of any slot. Like the Background element in the last section, a Border is a kind of Shoes::Pattern.



The first, crucial thing to know about border is that all borders paint a line around the **inside** of a slot, not the outside. So, if you have a slot which is fifty pixels wide and you paint a five pixel border on it, that means there is a fourty pixel wide area inside the slot which is surrounded by the border.

This also means that if you paint a Border on top of a Background, the edges of the background will be painted over by the border.

Here is just such a slot:

```
Shoes.app do
  stack :width => 50 do
    border black, :strokewidth => 5
    para "=^.^=", :stroke => green
  end
end
```

If you want to paint a border around the outside of a slot, you'll need to wrap that slot in another slot. Then, place the border in the outside slot.

```
Shoes.app do
  stack :width => 60 do
    border black, :strokewidth => 5
    stack :width => 50 do
      para "=^.^=", :stroke => green
    end
  end
end
```

In HTML and many other languages, the border is painted on the outside of the box, thus increasing the overall width of the box. Shoes was designed with consistency in mind, so that if you say that a box is fifty pixels wide, it stays fifty pixels wide regardless of its borders or margins or anything else.

Please also check out the Elements section for other methods used on borders.

**to_pattern()** » a Shoes::Pattern

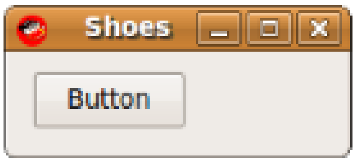
Creates a basic pattern object based on the color, gradient or image used to paint this border. The pattern may then be re-used in new borders and backgrounds.

Next: Button

The Shoes Manual

# Button

Buttons are, you know, push buttons. You click them and they do something. Buttons are known to say "OK" or "Are you sure?" And, then, if you're sure, you click the button.



```
Shoes.app do
  button "OK!"
  button "Are you sure?"
end
```

The buttons in the example above don't do anything when you click them. In order to get them to work, you've got to hook up a block to each button.

```
Shoes.app do
  button "OK!" do
    append { para "Well okay then." }
  end
  button "Are you sure?" do
    append { para "Your confidence is inspiring." }
  end
end
```

So now we've got blocks for the buttons. Each block appends a new paragraph to the page. The more you click, the more paragraphs get added.

It doesn't go much deeper than that. A button is just a clickable phrase.

Just to be pedantic, though, here's another way to write that last example.

```
Shoes.app do
  @b1 = button "OK!"
  @b1.click { para "Well okay then." }
  @b2 = button "Are you sure?"
  @b2.click { para "Your confidence is inspiring." }
end
```

This looks dramatically different, but it does the same thing. The first difference: rather than attaching the block directly to the button, the block is attached later, through the `click` method.

The second change isn't related to buttons at all. The `append` block was dropped since Shoes allows you to add new elements directly to the slot. So we can just call `para` directly. (This isn't the case with the `prepend`, `before` or `after` methods.)

Beside the methods below, buttons also inherit all of the methods that are Common.

**click() { |self| ... }** » self

When a button is clicked, its `click` block is called. The block is handed `self`. Meaning: the button which was clicked.

**focus()** » self

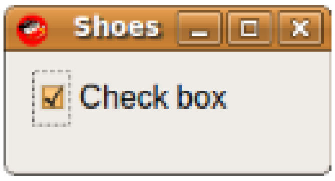
Moves focus to the button. The button will be highlighted and, if the user hits Enter, the button will be clicked.

Next: Check

The Shoes Manual

# Check

Check boxes are clickable square boxes than can be either checked or unchecked. A single checkbox usually asks a "yes" or "no" question. Sets of checkboxes are also seen in to-do lists.



---

Here's a sample checklist.

```
Shoes.app do
  stack do
    flow { check; para "Frances Johnson" }
    flow { check; para "Ignatius J. Reilly" }
    flow { check; para "Winston Niles Rumfoord" }
  end
end
```

You basically have two ways to use a check. You can attach a block to the check and it'll get called when the check gets clicked. And/or you can just use the `checked?` method to go back and see if a box has been checked or not.

Okay, let's add to the above example.

```
Shoes.app do
  @list = ['Frances Johnson', 'Ignatius J. Reilly',
    'Winston Niles Rumfoord']
  stack do
    @list.map! do |name|
      flow { @c = check; para name }
      [@c, name]
    end
    button "What's been checked?" do
      selected = @list.map { |c, name| name if c.checked? }.compact
      alert("You selected: " + selected.join(', '))
    end
  end
end
```

So, when the button gets pressed, each of the checks gets asked for its status, using the `checked?` method.

Button methods are listed below, but also see the list of Common methods, which all elements respond to.

**checked?()** » true or false

Returns whether the box is checked or not. So, `true` means "yes, the box is checked!"

**checked = true or false**

Marks or unmarks the check box. Using `checked = false`, for instance, unchecks the box.

**click() { |self| ... }** » self

When the check is clicked, its `click` block is called. The block is handed `self`, which is the check object which was clicked.

Clicks are sent for both checking and unchecking the box.

**focus()** » self

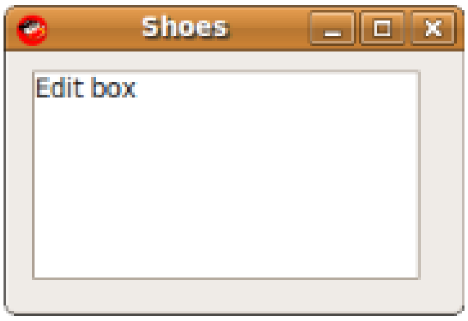
Moves focus to the check. The check will be highlighted and, if the user hits Enter, the check will be toggled between its checked and unchecked states.

Next: EditBox

The Shoes Manual

# EditBox

Edit boxes are wide, rectangular boxes for entering text. On the web, they call these textareas. These are multi-line edit boxes for entering longer descriptions. Essays, even!



Without any other styling, edit boxes are sized 200 pixels by 108 pixels. You can also use `:width` and `:height` styles to set specific sizes.

```
Shoes.app do
  edit_box
  edit_box :width => 100, :height => 100
end
```

Other controls (like Button and Check) have only click events, but both EditLine and EditBox have a `change` event. The `change` block is called every time someone types into or deletes from the box.

```
Shoes.app do
  edit_box do |e|
    @counter.text = e.text.size
  end
  @counter = strong("0")
  para @counter, " characters"
end
```

Notice that the example also uses the text method inside the block. That method gives you a string of all the characters typed into the box.

More edit box methods are listed below, but also see the list of Common methods, which all elements respond to.

**change() { |self| ... }** » self

Each time a character is added to or removed from the edit box, its `change` block is called. The block is given `self`, which is the edit box object which has changed.

**focus()** » self

Moves focus to the edit box. The edit box will be highlighted and the user will be able to type into the edit box.

**text()** » self

Return a string of characters which have been typed into the box.

**text = a string**

Fills the edit box with the characters of `a string`.

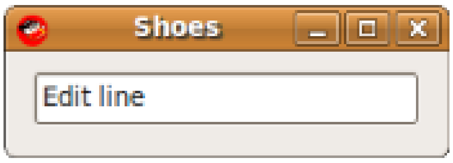
---

Next: EditLine

The Shoes Manual

# EditLine

Edit lines are a slender, little box for entering text. While the EditBox is multi-line, an edit line is just one. Line, that is. Horizontal, in fact.



The unstyled edit line is 200 pixels wide and 28 pixels wide. Roughly. The height may vary on some platforms.

```
Shoes.app do
  stack do
    edit_line
    edit_line :width => 400
  end
end
```

You can change the size by styling both the `:width` and the `:height`. However, you generally only want to style the `:width`, as the height will be sized to fit the font. (And, in current versions of Shoes, the font for edit lines and edit boxes cannot be altered anyway.)

If a block is given to an edit line, it receives `change` events. Check out the EditBox page for an example of using a change block. In fact, the edit box has all the same methods as an edit line. Also see the list of Common methods, which all elements respond to.

**change() { |self| ... }** » self

Each time a character is added to or removed from the edit line, its `change` block is called. The block is given `self`, which is the edit line object which has changed.

**focus()** » self

Moves focus to the edit line. The edit line will be highlighted and the user will be able to type into the edit line.

**text()** » self

Return a string of characters which have been typed into the box.

**text = a string**

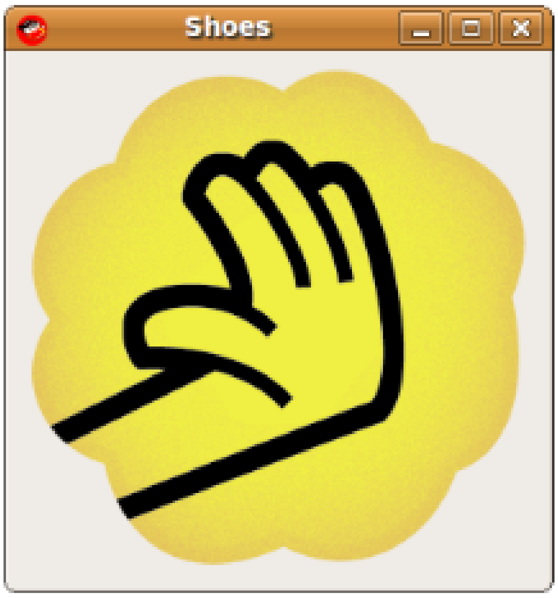
Fills the edit line with the characters of `a string`.

Next: Image

The Shoes Manual

# Image

An image is a picture in PNG, JPEG or GIF format. Shoes can resize images or flow them in with text. Images can be loaded from a file or directly off the web.



To create an image, use the `image` method in a slot:

```
Shoes.app do
  para "Nice, nice, very nice.  Busy, busy, busy."
  image "static/disheveled.gif"
 end
```

When you load any image into Shoes, it is cached in memory. This means that if you load up many image elements from the same file, it'll only really load the file once.

You can use web URLs directly as well.

```
Shoes.app do
  image "http://hacketyhack.net/images/design/Hacky-Mouse-Hand.png"
end
```

When an image is loaded from the web, it's cached on the hard drive as well as in memory. This prevents a repeat download unless the image has changed. (In case you're wondering: Shoes keeps track of modification times and etags just like a browser would.)

Shoes also loads remote images in the background using system threads. So, using remote images will not block Ruby or any intense graphical displays you may have going on.

**full_height()** » a number

The full pixel height of the image. Normally, you can just use the height method to figure out how many pixels high the image is. But if you've resized the image or styled it to be larger or something, then `height` will return the scaled size.

The `full_height` method gives you the height of image (in pixels) as it was stored in the original file.

**full_width()** » a number

The full pixel width of the image. See the full_height method for an explanation of why you might use this method rather than width.

**path()** » a string

The URL or file name of the image.

**path = a string**

Swaps the image with a different one, loaded from a file or URL.

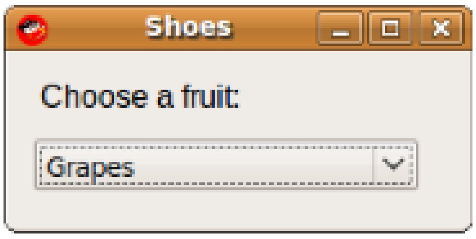
---

Next: ListBox

The Shoes Manual

# ListBox

List boxes (also called "combo boxes" or "drop-down boxes" or "select boxes" in some places) are a list of options that drop down when you click on the box.



---

A list box gets its options from an array. An array (a list) of strings, passed into the `:items` style.

```
Shoes.app do
  para "Choose a fruit:"
  list_box :items => ["Grapes", "Pears", "Apricots"]
end
```

So, the basic size of a list box is about 200 pixels wide and 28 pixels high. You can adjust this length using the `:width` style.

```
Shoes.app do
  para "Choose a fruit:"
  list_box :items => ["Grapes", "Pears", "Apricots"],
    :width => 120, :choose => "Apricots" do |list|
      @fruit.text = list.text
  end
  @fruit = para "No fruit selected"
end
```

Next to the `:width` style, the example uses another useful option. The `:choose` option tells the list box which of the items should be highlighted from the beginning. (There's also a choose method for highlighting an item after the box is created.)

List boxes also have a change event. In the last example, we've got a block hooked up to the list box. Well, okay, see, that's a `change` block. The block is called each time someone changes the selected item.

Those are the basics. Might you also be persuaded to look at the Common methods page, a complete list of the methods that all elements have?

**change() { |self| ... }** » self

Whenever someone highlights a new option in the list box (by clicking on an item, for instance,) its `change` block is called. The block is given `self`, which is the edit line object which has changed.

**choose(item: a string)** » self

Selects the option in the list box that matches the string given by `item`.

**focus()** » self

Moves focus to the list box. The list box will be highlighted and, if the user hits the up and down arrow keys, other options in the list will be selected.

**items()** » an array of strings

Returns the complete list of strings that the list box presently shows as its options.

**items = an array of strings**

Replaces the list box's options with a new list of strings.

**text()** » a string

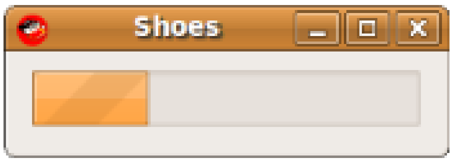
A string containing whatever text is shown highlighted in the list box right now. If nothing is selected, `nil` will be the reply.

Next: Progress

The Shoes Manual

# Progress

Progress bars show you how far along you are in an activity. Usually, a progress bar represents a percentage (from 0% to 100%.) Shoes thinks of progress in terms of the decimal numbers 0.0 to 1.0.



A simple progress bar is 200 pixels wide, but you can use the `:width` style (as with all Shoes elements) to lengthen it.

```
Shoes.app do
  stack :margin => 0.1 do
    title "Progress example"
    @p = progress :width => 1.0
    animate do |i|
      @p.fraction = (i % 100) / 100.0
    end
  end
end
```

Take a look at the Common methods page for a list of methods found an all elements, including progress bars.

**fraction()** » a decimal number

Returns a decimal number from 0.0 to 1.0, indicating how far along the progress bar is.

**fraction = a decimal number**

Sets the progress to a decimal number between 0.0 and 1.0.

Next: Radio

The Shoes Manual

# Radio

Radio buttons are a group of clickable circles. Click a circle and it'll be marked. Only one radio button can be marked at a time. (This is similar to the ListBox, where only one option can be selected at a time.)

---

So, how do you decide when to use radio buttons and when to use list boxes? Well, list boxes only show one highlighted item unless you click on the box and the drop-down appears. But radio buttons are all shown, regardless of which is marked.

```
Shoes.app do
  para "Among these films, which do you prefer?\n"
  radio; para strong("The Taste of Tea"), " by Katsuhito Ishii\n"
  radio; para strong("Kin-Dza-Dza"), " by Georgi Danelia\n"
  radio; para strong("Children of Heaven"), " by Majid Majidi\n"
end
```

Only one of these three radios can be checked at a time, since they are grouped together in the same slot (along with a bunch of `para`.)

If we move them each into their own slot, the example breaks.

```
Shoes.app do
  stack do
    para "Among these films, which do you prefer?"
    flow { radio; para "The Taste of Tea by Katsuhito Ishii" }
    flow { radio; para "Kin-Dza-Dza by Georgi Danelia" }
    flow { radio; para "Children of Heaven by Majid Majidi" }
  end
end
```

This can be fixed, though. You can group together radios from different slots, you just have to give them all the same group name.

Here, let's group all these radios in the `:films` group.

```
Shoes.app do
  stack do
    para "Among these films, which do you prefer?"
    flow do
      radio :films
      para "The Taste of Tea by Katsuhito Ishii"
    end
    flow do
      radio :films
      para "Kin-Dza-Dza by Georgi Danelia"
    end
    flow do
      radio :films
      para "Children of Heaven by Majid Majidi"
    end
  end
end
```

For more methods beyond those listed below, also look into the Common methods page. Because you get those methods on every radio as well.

### **checked?()** » true or false

Returns whether the radio button is checked or not. So, `true` means "yes, it is checked!"

### **checked = true or false**

Marks or unmarks the radio button. Using `checked = false`, for instance, clears the radio.

### **click() { |self| ... }** » self

When the radio button is clicked, its `click` block is called. The block is handed `self`, which is an object representing the radio which was clicked.

Clicks are sent for both marking and unmarking the radio.

### **focus()** » self

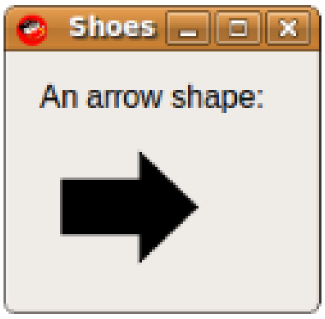
Moves focus to the radio. The radio will be highlighted and, if the user hits Enter, the radio will be toggled between its marked and unmarked states.

Next: Shape

The Shoes Manual

## Shape

A shape is a path outline usually created by drawing methods like `oval` and `rect`.



An arrow shape:

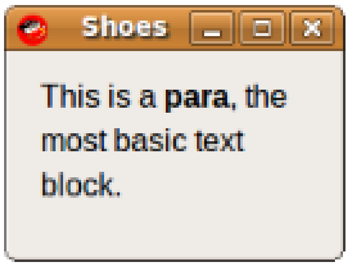
See the Common methods page. Shapes respond to all of those methods.

Next: TextBlock

The Shoes Manual

# TextBlock

The TextBlock object represents a group of text organized as a single element. A paragraph containing bolded text, for example. A caption containing links and bolded text. (So, a `caption` is a TextBlock type. However, `link` and `strong` are TextClass types.)



This is a **para**, the most basic text block.

All of the various types of TextBlock are found on the Element Creation page.

- banner, a 48 pixel font.
- title, a 34 pixel font.
- subtitle, a 26 pixel font.
- tagline, an 18 pixel font.
- caption, a 14 pixel font.
- para, a 12 pixel font.
- inscription, a 10 pixel font.

**contents()** » an array of elements

Lists all of the strings and styled text objects inside this block.

**replace(a string)**

Replaces the text of the entire block with the characters of `a string`.

**text()** » a string

Return a string of all of the characters in this text box. This will strip off any style or text classes and just return the actual characters, as if seen on the screen.

**text = a string**

Replaces the text of the entire block with the characters of `a string`.

**to_s()** » a string

An alias for text. Returns a flattened string of all of this TextBlock's contents.

Next: Timers

The Shoes Manual

# Timers

Shoes contains three timer classes: the Animation class, the Every class and the Timer class. Both Animations and Everies loop over and over after they start. Timers happen once. A one-shot timer.

Animations and Everies are basically the same thing. The difference is that Animations usually happen many, many times per second. And Everies happen only once every few seconds or rarely.

**start()** » self

Both types of timers automatically start themselves, so there's no need to use this normally. But if you stop a timer and would like to start it up again, then by all means: use this!

**stop()** » self

Pauses the animation or timer. In the case of a one-shot timer that's already happened, it's already stopped and this method will have no effect.

**toggle()** » self

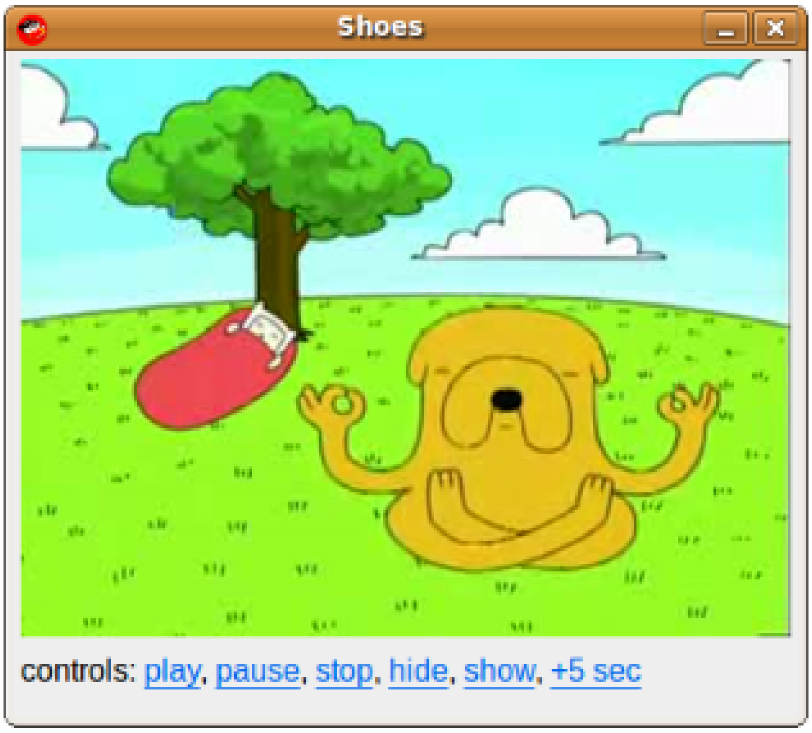
If the animation or timer is stopped, it is started. Otherwise, if it is already running, it is stopped.

Next: Video

The Shoes Manual

# Video

Shoes supports embedding of QuickTime, Flash video (FLV), DivX, Xvid and various other popular video formats. This is all thanks to VideoLAN and ffmpeg, two sensational open source libraries. Use the `video` method on a slot to setup a Shoes::Video object.



In addition to video formats, some audio formats are also supported, such as MP3, WAV and Ogg Vorbis.

Video support is optional in Shoes and some builds do not support video. For example, video support is unavailable for PowerPC. When you download Shoes, the build for your platform will be marked `novideo` in the filename if no video support is available.

**hide()** » self

Hides the video. If already playing, the video will continue to play. This just turns off display of the video. One possible use of this method is to collapse the video area when it is playing an audio file, such as an MP3.

**length()** » a number

The full length of the video in milliseconds. Returns nil if the video is not yet loaded.

**move(left, top)** » self

Moves the video to specific coordinates, the (left, top) being the upper left hand corner of the video.

**pause()** » self

Pauses the video, if it is playing.

**playing?()** » true of false

Returns true if the video is currently playing. Or, false if the video is paused or stopped.

**play()** » self

Starts playing the video, if it isn't already playing. If already playing, the video is restarted from the beginning.

**position()** » a decimal

The position of the video as a decimanl number (a Float) between the beginning (0.0) and the end (1.0). For instance, a Float value of 0.5 indicates the halfway point of the video.

**position = a decimal**

Sets the position of the video using a Float value. To move the video to its 25% position: `@video.position = 0.25`.

**remove()** » self

Removes the video from its slot. This will stop the video as well.

**show()** » self

Reveals the video, if it has been hidden by the `hide()` method.

**stop()** » self

Stops the video, if it is playing.

**time()** » a number

The time position of the video in milliseconds. So, if the video is 10 seconds into play, this method would return the number 10000.

**time = a number**

Set the position of the video to a time in milliseconds.

**toggle()** » self

Toggles the visibility of the video. If the video can be seen, then `hide` is called. Otherwise, `show` is called.