Visit http://turtlebits.net/ to run your programs.

The free website is made possible by your purchase of this book.

"The love of beauty is taste.
The creation of beauty is Art."
- Ralph Waldo Emerson

A Programming Primer runs on TurtleBits
which unites the CoffeeScript language by Jeremy Ashkenas in 2009,
and Iced await/defer extensions created by Maxwell Krohn in 2012,
with the jQuery-turtle plugin devleoped by the author in 2011,
built on the jQuery library invented by John Resig in 2006.
This work is inspired by the beloved LOGO language
created by Seymour Papert and Wally Feurzeig in 1967.

Special thanks to the students in Lincoln Massachusetts,
Beaver Country Day School, and Dorchester McCormack School
who vetted this material.

Post questions, ideas, and bug reports to http://turtlebits.net/group

This book is typeset in Łukasz Dziedzic's 2010 open font Lato
and Paul D. Hunt's 2012 Adobe Source Code Pro.

No Thresholds and No Limits

The aim of this book is to teach you to write programs as you would use a pencil: as an outlet for creativity and as a tool for understanding.

These pages follow a fifty-year tradition of using programming as a liberating educational tool, with no thresholds for beginners, and no limits for experts. Seymour Papert's LOGO is the inspiration. Start with a few lines of code, and progress to writing programs to explore art, mathematics, language, algorithms, simulation, and thought.

The language is CoffeeScript. Although CoffeeScript is a production programming language used by pros, it was chosen here because it has an elegance and simplicity well-suited for beginners. While the first examples make the language look trivial, CoffeeScript has a good notation for all the important ideas: algebraic expressions, lists, loops, functions, objects, and concurrency. As you learn the language, remember that the goal should be not mastery of the syntax, but mastery of the underlying concepts.

Edit and run your programs on turtlebits.net. The site is a live experiment in community learning: everything posted is public, so write programs that would be interesting to others. Accounts are free.

As you experiment by building your own ideas, you will find that at first your programs will behave in ways that you do not intend. Details matter, and persistence pays off. If you are patient in adjusting and perfecting your work, you will be rewarded with insight.

Read, think, play, and strive to create something beautiful.

David Bau, 2013

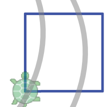
1. Lines

First

```
pen red
fd 50
rt 90
```

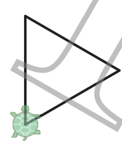
Square

```
pen blue
fd 50; rt 90
fd 50; rt 90
fd 50; rt 90
fd 50; rt 90
```



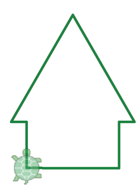
Triangle

```
pen black
fd 70; rt 120
fd 70; rt 120
fd 70; rt 120
```



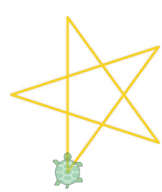
House

```
speed 5
pen green
fd 30; lt 90
fd 10; rt 120
fd 80; rt 120
fd 80; rt 120
fd 10; lt 90
fd 30; rt 90
fd 60; rt 90
```



Star

```
pen gold
fd 100; rt 144
fd 100; rt 144
fd 100; rt 144
fd 100; rt 144
fd 100; rt 144
```



A simple computer program is called a ***script***, because the computer performs it like reading lines like a play. Each command is followed, one at a time, from beginning to end.

Basic Movement

The scripts on this page use four basic functions to move a turtle:

`fd 100` moves the turtle forward 100 pixels.
`bk 100` goes backward.
`rt 90` turns right 90 degrees.
`lt 90` turns left.

In CoffeeScript, `fd` is different from FD (and FD is not defined), so all these function names should be typed in lowercase. It is important to put a space between the function name and the number. Do not indent the code for now, because indenting has special meaning in the language.

Notice that a small turn traces out an obtuse angle. An acute angle requires a turn more than 90. Turtles measure turns in ***exterior*** angles, so a complete circuit always adds to a multiple of 360.

On turtlebits.net, you can try single commands and ask for help in the console on the bottom of the right panel. It is a good way to experiment.

Drawing in Color

The turtle draws a line by selecting a pen.

`pen red` traces out a line in red.

Common color names such as red, black, white, blue, green, yellow, orange, and purple all work. There are 140 standard color names that are listed at the end of this book.

Unselect the pen by using `pen null`. Use `pen erase` for an eraser.

Speed

The turtle takes about a second to trace out any movement, but its speed can be changed.

`speed 10` sets the speed to 10 moves per second.
`speed Infinity` moves instantly.

Semicolons

The semicolon (;) that appears in the examples is just used for combining two commands on the same line. These programs would behave the same if all the commands separated by semicolons were written on separate lines.

2. Points

Dot Row

```
rt 90; dot lightgray
fd 30; dot gray
fd 30; dot()
fd 30
```

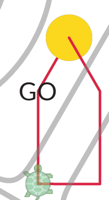
Message

```
message = 'Hello You.'
see 'message'
see message
```

```
message
Hello You.
```

Lighthouse

```
pen crimson
fd 60; label 'GO'
rt 30
fd 40; rt 120; dot gold, 30
fd 40; rt 30
fd 60; rt 90
fd 40; rt 90
```



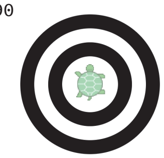
Smiley

```
speed 10
dot yellow, 160
fd 20
rt 90
fd 25
dot black, 20
bk 50
dot black, 20
bk 5
rt 90
fd 40
pen black, 7
lt 30
lt 120, 35
ht()
```

Bullseye

```
x = 18
see x * 5
dot black, x * 5
dot white, x * 4
dot black, x * 3
dot white, x * 2
```

```
90
```



Some new functions on this page:

`dot black, 20` draws a black dot of diameter 20 under the turtle.
`label 'GO'` draws the text GO under the turtle.
`see x * 5` shows the value of x * 5 in the test console.
`ht()` hides the turtle. Show it again with `st()`.
`lt 120, 35` traces an arc of radius 35 while turning left 120 degrees.

Debugging

If you are lost in a long program, add `dot red` or `label 'A'` or `see x` to understand a specific point in the code.

These three functions are useful for ***debugging*** because they make a visible record of the current state of the program without otherwise changing things.

Variables and Strings

Most of the words in our programs (including `fd`, `rt`, `speed` and `red`) are predefined in TurtleBits, but you can define your own words using the `=` equals assignment symbol.

The assignment `message = 'Hello You.'` defines the word `message` to stand for the text "Hello You." inside the program **Message**.

Bullseye defines `x = 18`. After the definition, x means 18. For example, if we were to write `see x` or `label x`, it would <u>not</u> draw the letter x on the screen. It would write out the number 18. Words like `x` without quotes are are called ***variables***. Variables can stand for numbers, functions, text, or other objects.

To literally write the letter "x" on the screen, put it in quotes: `label 'x'` will show the letter x, and `see 'message'` will write out the word "message". Quoted text in a program is called a *string*.

Arithmetic Operations

Mathematical operations are written as you would expect:

`x + y` addition.
`x - y` subtraction.
`x * y` multiplication.
`x / y` division.

Parentheses and order-of-operations work as taught in math class. When x is 18, `see x + x * x / (7 + x)` will do the computation and show 30.96.

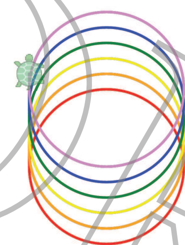
3. Loops

Rectangle

```
pen green
for d in [50, 100, 50, 100]
  fd d
  rt 90
```

Rainbow

```
for c in [
    red
    orange
    yellow
    green
    blue
    violet
    ]
  pen c
  rt 360, 50
  fd 10
```



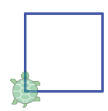
Range

```
see [1..5]
see [1...5]
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4]
```

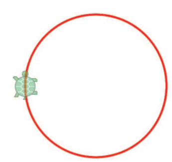
Square Loop

```
pen blue
for x in [1..4]
  fd 50
  rt 90
```



360 Loop

```
speed 100
pen red
for x in [1..360]
  fd 1
  rt 1
```



Descending Loop

```
pen purple
for x in [50..1] by -1
  rt 30, x
```

To draw a rectangle, we could write the following.

`fd 50; rt 90; fd 100; rt 90; fd 50; rt 90; fd 100; rt 90`

But that is wordy and repetitive. The program **Rectangle** is clearer because it uses a `for` loop to repeat the `fd` and `rt 90` commands.

The Parts of a Loop

Look closely at **Rectangle**. The `for` loop has three parts:

The *loop variable* `d`.
The *loop list* `[50, 100, 50, 100]`.
The *loop body* `fd d; rt 90`.

The prepositions `for` and `in` are special words in the language: they introduce a loop variable and its loop list.

Since the list countains four numbers, the loop repeats the body four times: once with `d` set to 50; then once with `d` as 100; then again as 50; then finally as 100 again.

Loop Lists and Ranges

A list is written by surrounding items with square brakets `[ ]`.

If you write list items on a single line, separate them with commas. Longer lists like the list of colors in **Rainbow** can be written on multiple lines for clarity; commas are not needed at linebreaks.

A range of numbers can be listed by putting two dots `..` between the lowest and highest numbers. If you use three dots `...`, the effect is similar, but the last number will not be included in the list.

Indenting is Important

The commands in the loop are indented underneath the `for` line to show that they are inside the loop. It is important to indent lines inside the loop body evenly with each other.

List items should also be indented evenly with each other when written on separate lines.

Simple Loops and Stride

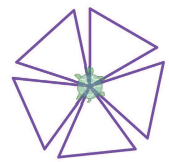
Notice that the loop variable does not need to be used inside the body of the loop. In **Square Loop** and **360 Loop**, the variable `x` is not used except to count the number of repetitions.

In **Descending Loop**, the word `by` after the list denotes a ***stride***, which is how much to skip foward when looping through the list. Looping `by 2` would skip every other number. Looping `by -1` counts down.

4. Nesting

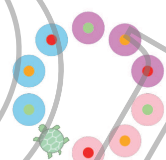
Violet

```
pen blueviolet
for x in [1..5]
  rt 72
  for y in [1..3]
    fd 50
    rt 120
```



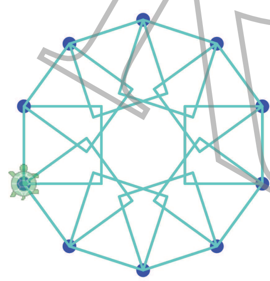
Combinations

```
for outside in [skyblue, violet, pink]
  for inside in [palegreen, orange, red]
    dot outside, 21
    dot inside, 7
    fd 25
    rt 36
```



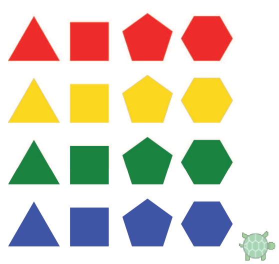
Decorated Nest

```
pen turquoise
for y in [1..10]
  dot blue
  for x in [1..4]
    fd 50
    rt 90
  lt 36
  bk 50
```



Catalog

```
speed 100
rt 90
for color in [red, gold, green, blue]
  jump 40, -160
  for sides in [3..6]
    pen path
    for x in [1..sides]
      fd 100 / sides
      lt 360 / sides
    fill color
    fd 40
```



Any code can be put in a loop, including another loop.

Nesting loops within loops can create beautiful effects. **Violet** arranges five triangles around a point by nesting a loop of 3 within a loop of 5. The single line `fd 50` is repeated 15 times with perfect symmetry.

Inner and Outer Loops

When loops are nested, the ***inner*** loop is the one that repeats most quickly. Consider **Combinations**.

A single pass through a loop is called an ***iteration***. On each iteration, the program draws a small dot within a big dot, then moves the turtle a bit. The color of the small dot comes from the variable `inside`, which is the loop variable of the inner loop. The large dot color comes from the outer loop variable `outside`.

Because the inner loop repeats most quickly, the small dot colors palegreen, orange, and red change on every iteration.

The outer loop repeats only after the inner loop has made a full set of iterations, so the `outside` dot colors change only after 3 inner iterations have been made.

Nesting Carefully

The level of indent indicates whether code is within an inner loop or an outer loop, or not within a loop at all.

In **Decorated Nest**, `fd 50` is indented twice to be in the innermost loop. It runs 40 times in total. However, `dot blue` is only indented once, so it is in the outer loop and done only 10 times. Lines that are not indented, such as `pen turquoise`, are not looped, and they are done only once.

Loops can be nested as deeply as you like. **Catalog** shows a triply-nested loop. Its innermost loop repeats by a number that varies (`sides`) because the loop range comes from the second level loop variable.

Jumping and Path Filling

Some new functions:

`jump 40, -160` jumps right 40 and back 160.
`pen path` traces with a special invisible path pen.
`fill color` fills the invisible path with color.

Note that `jump` jumps right and up relative to the current direction and position of the turtle, and it does not draw with the pen or turn the turtle. To jump to an absolute Cartesian coordinate, use `jumpto`.

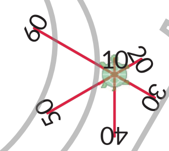
5. Functions

Scoot Function

```
pen purple
scoot = (x) -> fd 10 * x
rt 90
scoot 7
```

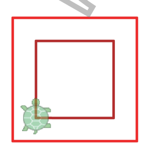
Spike Function

```
spike = (x) ->
  fd x
  label x
  bk x

pen crimson
for n in [1..6]
  spike n * 10
  rt 60
```



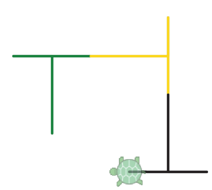
Square Function

```
square = (size) ->
  for y in [1..4]
    fd size
    rt 90

pen red
square 80

jump 15, 15
pen firebrick
square 50
```



Tee Function

```
tee = ->
  fd 50
  rt 90
  bk 25
  fd 50

pen green
tee()
pen gold
tee()
pen black
tee()
```



The most important idea in this book:

Programs define their own functions.

A function is a miniature program. In CoffeeScript, a function is written with an arrow `->` typed as two symbols next to each other (the minus and the greater-than) like this:

`(input) -> something to do`

Writing and Naming Functions

A function that advances the turtle by ten times a distance is

`(x) -> fd 10 * x`

Name a function like any variable, using `=`.

`scoot = (x) -> fd 10 * x`

After the definition, we can write `scoot 7` or `scoot 5 + 2`. In other words, `scoot` can be used just like predefined functions like `fd` or `rt`.

Parameters

The variable `x` in parentheses in the function definition is called a ***parameter***. Parameters may use any name. When the function is run, the parameter takes on the value passed to the function.

When `spike n * 10` is called, the code within the function binds parameter name `x` to the current value of `n * 10`, which is 10 during the first iteration of the loop.

Each time a function is called, its parameters can have different values. The last time `spike` is called, `n * 10` has advanced to 60, so the value of `x` during the last function call is 60.

Indenting Functions

The level of indenting is important for determining the scope of a line. If a line is indented under an arrow, that line is inside the function.

If the function itself contains loops, those should be indented further. There is no limit to the depth of nested intenting, but indenting must be done neatly. Each level of indenting indicates a particular function, loop, or nested scope.

Functions with No Parameters

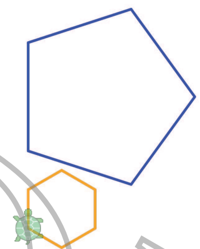
Functions like `tee` that have no parameters are written specially:

The definition `tee = -> ...` omits the parentheses.
Calling `tee()` requires empty parentheses.

6. Parameters

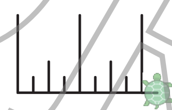
Polygon

```
polygon = (c, s, n) ->
  pen c
  for x in [1..n]
    fd s
    rt 360 / n
  pen null

polygon blue, 70, 5
bk 50
polygon(orange, 25, 6)
```



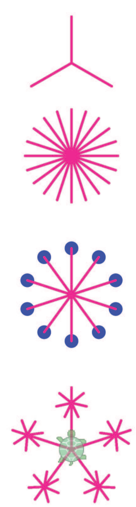
Rule

```
rule = (sizes) ->
  for x in sizes
    fd x
    bk x
    rt 90; fd 10; lt 90

pen black
rule [50, 10, 20, 10, 50, 10, 20, 10, 50]
```



Starburst

```
starburst = (x, shape) ->
  for z in [1..x]
    shape()
    rt 360 / x
stick = -> fd 30; bk 30

pen deeppink
starburst 3, stick

jump 0, -60
starburst 20, stick

jump 0, -90
starburst 10, -> fd 30; dot blue; bk 30

jump 0, -100
starburst 5, ->
  fd 30
  starburst 7, ->
    fd 10
    bk 10
  bk 30
```



Multiple parameters can be listed in a function definition with commas. The declaration `polygon = (c, s, n) ->` sets up three parameters: a color `c`, a side length `s`, and a number `n`.

Passing Arguments

The value passed to a parameter when using a function is called an ***argument***. When calling a function with several parameters, the arguments are listed with commas. For clarity, you can put parentheses around the argument list, like `polygon(orange, 25, 6)`.

When using parentheses around function arguments, do not put any space between the function name and the first parentheses, or else the parentheses will be interpreted as enclosing only the first argument.

Objects as Arguments

An argument may be a complex object such as a list. That is the approach taken in **Rule**.

The parameter named `sizes` is used as the loop list in a `for` loop. When `rule` is called, the whole list is passed as one argument.

Functions as Arguments

An argument may itself be another function. That is done in **Starburst**. The technique allows one mini-program to be attached to another.

The call to `starburst 3, stick` passes the function `stick` as the last argument. Inside `starburst`, `n` now stands for 3, and `shape` stands for the `stick` function. When `shape()` is written, `stick()` is called. In the end `stick` is called three times, drawing three symmetric sticks.

Calling `starburst 30, stick` calls `stick` 30 times, making a circular starburst of 30 sticks.

Unnamed Inline Functions

Calling `starburst n, something` means "Do something n times in a star." We can provide any code as something, even if unnamed.

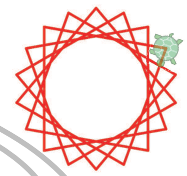
The call `starburst 10, -> fd 30; dot blue; bk 30` passes a lollipop-like function to `starburst`. The function has no name &endash; it is defined inline to draw line with a blue dot at the far end. The starburst function binds this unnaped function to its local parameter name `shape` and calls it 10 times. The result is a starburst with blue dots.

The last `starburst` call passes unnamed code that does another `starburst`. The result is a starburst made out of starbursts!

7. Time

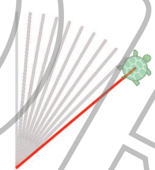
Pause

```
speed 100
pen red
for x in [1..20]
  fd 80
  rt 100
  if x is 10
    pause 2
```



Second Hand

```
speed Infinity
advance = ->
  pen lightgray
  bk 100
  rt 5
  pen red
  fd 100
tick advance
```



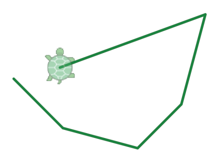
Countdown

```
seconds = 5
tick ->
  if seconds is 0
    write "Time's up!"
    tick null
  else
    write seconds
    seconds = seconds - 1
```

5
4
3
2
1
Time's up!

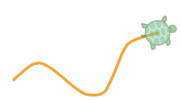
Click Draw

```
speed Infinity
pen green

tick ->
  moveto lastclick
```



Move Draw

```
speed Infinity
pen orange

tick 100, ->
  turnto lastmousemove
  fd 1
```



There are two techinques for organizing time in a program:
Queues: process lists of events over time in sequence.
Frames: process snapshots of the world at regular time intervals.

How `speed` Works with Queues

In TurtleBits, each turtle has its own ***animation queue*** that is used if you set `speed` to any number less than Infinity. (The default speed is one.)

Each movement command like `fd 100` adds the motion to the turtle's animation queue. When the program is finished running, the turtle has the whole plan, and it runs through its animation queue after your program is done.

The animation queue works well for timed motions that your program can plan ahead of time. But if you are writing a game or simulation that needs to respond to events in real time, then you may find it more sensible to to draw frames.

How `tick` Works with Frames

The `tick` command is used for frames: it calls the passed function at a regular rate. The optional first argument is the frame rate (the default rate is one frame per second).

The **Countdown** example writes a number on each tick callback. It also shows how to clear the callback once you are done: call `tick null`.

The **Move Draw** example is a very simple interactive program that uses `tick`. 100 times per second, it runs a function that turns the turtle toward the position on the screen where the mouse last moved, then advaces the turtle by one pixel. Because each frame should be drawn instantanously, it sets `speed Infinity`.

New Functions and Variables

Several new built-in names are used in these examples.

`pause 2` adds a 2-second pause to the animation queue.
`tick 100, fn` calls fn 100 times per second.
`write "Time's up"` writes a message on the screen.
`moveto lastclick` moves the turtle to the position of the last click.
`turnto lastmousemove` turns the turtle toward the last mouse position.

The `moveto` can be used with any cartesian coordinate or any object that has a position - it happens to be used here with the special variable `lastclick`. Similarly, `turnto` can be used with any absolute direction or coordinate. The special variable `lastmousemove` happens to keep the most recent mouse position.

8. Output

Poetry and Song

```
cry = (who, query) ->
  write "Oh #{who}, #{who}!"
  write "#{query} #{who}?"
cry "Romeo", "Wherefore art thou"
cry "kitty", "What did you eat"
play "fc/c/dcz"
```

Oh Romeo, Romeo!
Wherefore art thou
Romeo?
Oh kitty, kitty!
What did you eat kitty?

Imagery

```
url = "http://upload.wikimedia.org/wikipedia" +
      "/commons/6/61/Baby_Gopher_Tortoise.jpg"
write """<center><img src="#{url}" width=100>
      </center>"""
```

Bold Statement

```
n = write "<h1>Notice</h1>"
write """
<p>This long paragraph has
<b>bold</b>, <i>italic</i>,
and <u>underlined</u> text.
Horizontal rule below.</p>
"""
write "<hr>"
write """
<p><a href="//turtlebits.net/">
Link</a> with an &lt;a&gt;.
</p>
"""
n.css
  background: pink
```

Notice

This long paragraph has **bold**, *italic*, and underlined text. Horizontal rule below.

Link with an <a>.

Graffiti

```
n = write "<h1>Notice</h1>"
write """
<p>This long paragraph has
<b>bold</b>, <i>italic</i>,
and <u>underlined</u> text.
</p>"""
n.css
  background: pink
  display: 'inline-block'
n.pen purple, 10
n.bk 80
n.rt 45
n.fd 50
```

A string written with double quotes `"..."` can ***interpolate*** values written as `#{something}`, which means the value of something is inserted into the string.

A multiline string can be written by tripling the quotes (either double or single) around the string, as is done in the last string of **Imagery**.

HTML Elements

Codes like are called ***HTML*** tags. They set off text for special formatting: and mark bold text; <h1> and </h1> mark a first-level heading; the <hr> tag is a "horizontal rule".

A matching tag pair and its contents (or singleton tag, for tags like <hr> or <img> that are not paired) make up an HTML ***element***.

Attributes

HTML Elements can have attributes with special meanings such as the `href` attribute on the <a> element, which sets the URL for a hyperlink.

The other attributes seen on this page are the `src` and `width` attributes on the <img> element, which specify the location from which to load the image data, and the scaling width to use.

jQuery Objects

Programs can use ***jQuery objects*** to alter HTML elements on the screen.

The code n = write "<h1>Notice</h1>" returns a jQuery object for the <h1> element, and stores it in the variable `n`. Then the jQuery function `n.css` is used to alter its CSS.

In TurtleBits, all the turtle methods such as `fd` and `pen` are available as jQuery methods. Any element can be moved like a turtle.

CSS Properties

CSS properties can alter many of the details of HTML formatting such as such as an element's `background` (set here to `pink`). The `css` function can set more than one property at once, and the property list under `n.css` should be indented.

Exploring More

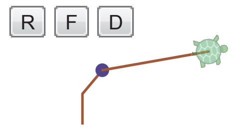
HTML is a rich subject. There are more than 100 types of HTML elements, more than 100 HTML attributes, more than 100 jQuery methods, and more than 100 CSS properties. The best way to explore all these options is to search for and consult the many resources on the Internet about these technologies.

And experiment.

9. Input

Button Control

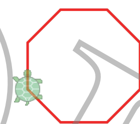
```
pen sienna
button 'R', -> rt 10
button 'F', -> fd 10
button 'D', -> dot 'darkslateblue'
```



Polygon to Order

```
read "Color?", (color) ->
  read "Sides?", (sides) ->
    pen color
    for x in [1..sides]
      fd 30
      rt 360 / sides
```

Color? red
Sides? 8



Guess My Number

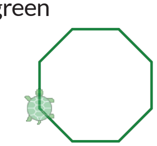
```
secret = random [1..100]
turns = 5
write "Guess my number."
dopick = (pick) ->
  if pick is secret
    write "You got it!"
    return
  if 1 <= pick < secret
    write "Too small!"
    turns = turns - 1
  else if secret < pick <= 100
    write "Too big!"
    turns = turns - 1
  if turns > 1
    write "#{turns} left."
    readnum dopick
  else if turns is 1
    write "Last guess!"
    readnum dopick
  else
    write "Game over."
    write "It was #{secret}."
readnum dopick
```

Guess my number.
⇒ 50
Too small!
4 left.
⇒ 75
Too big!
3 left.
⇒ 64
Too big!
2 left.
⇒ 55
Too small!
Last guess!
⇒ 59
You got it!

Polygon Revisited

```
await read "Color?", defer color
await read "Sides?", defer sides
pen color
for x in [1..sides]
  fd 30
  rt 360 / sides
```

Color? green
Sides? 8



The examples on this page gather input using ***callbacks***:

`button` sets up a function to be called whenever a button is pressed.
`read` calls a function once after the user answers a prompt.
`readnum` is like read, but for numbers only.

Chaining Callbacks

If more than one `read` input is needed in a program, callbacks can be chained: after the first callback receives one value, it can request another input value by setting up a nested callback function.

Randomness and Reassigning Variables

The **Guess My Number** example uses the `random` function to pick an unpredictable number from 1 to 100. (The argument to `random` is a list of numbers to choose from.)

The game allows five turns to guess the number, tracked in the variable `turns`. The ***assignment*** `turns = turns - 1` means "set the value of `turns` to be one less than the old value of `turns`".

Booleans and Conditionals

A true or false value is called a ***boolean***. The expression `turns > 1` is a boolean that is true when `turns` exceeds 1. When used with conditional words `if` and `else`, booleans control program flow. Other examples:

`pick is secret`, true when the two variables have the same value.
`pick isnt secret`, true when the two variables are unequal.
`secret < pick <= 100`, true when `pick` exceeds `secret` but not 100.
`secret < pick and pick <= 100`, the same thing written using `and`.
`not (secret >= pick or pick > 100)`, again with `not` and `or`.

Statements to be be run conditionally should be indented underneath the `if` or `else` line that controls the condition.

Repeating a Question

It is worth thinking about how the game asks repeated questions. Each guess is collected by calling `readnum dopick`. The callback function, named `dopick`, is a set of instructions of "what to do after a guess is made." When `readnum dopick` is run inside `dopick` itself, it repeats the whole process! Calling a function inside itself is called ***recursion***.

Chained callbacks and recursion can be simplified by generating ***continuations*** with `await` and `defer`. **Polygon Revisited** does exactly the same thing as **Polygon To Order**: the `await` statement automatically sets up a callback function that is passed using `defer`. More examples can be found in the sections on Arrays and Concurrency.

10. Numbers

Parsing

```
write '5' + '3'
write Number('5') + Number('3')
```

```
53
8
```

Ways to Count

```
counter = 0
write ++counter + 'a'
write (counter += 1) + 'b'
write (counter = counter + 1) + 'c'
```

```
1a
2b
3c
```

Circle Measurements

```
area = (radius) ->
  Math.PI * radius * radius

circumference = (radius) ->
  2 * Math.PI * radius

for r in [1, 5, 10]
  write 'radius ' + r
  write 'a ' + area r
  write 'c ' + circumference r
```

```
radius 1
a 3.141592653589793
c 6.283185307179586
radius 5
a 78.53981633974483
c 31.41592653589793
radius 10
a 314.1592653589793
c 62.83185307179586
```

Hypotenuse

```
hypotenuse = (a, b) ->
  Math.sqrt(a * a + b * b)

write hypotenuse 3, 4
write hypotenuse 5, 12
write hypotenuse 10, 10
write Math.floor(hypotenuse(10, 10))
```

```
5
13
14.142135623730951
14
```

Euclid's Method

```
gcf = (a, b) ->
  if a > b
    return gcf b, a
  remainder = b % a
  if remainder is 0
    return a
  gcf remainder, a

for x in [80..88]
  write "gcf(120,#{x})=" +
      gcf(120, x)
```

```
gcf(120,80)=40
gcf(120,81)=3
gcf(120,82)=2
gcf(120,83)=1
gcf(120,84)=12
gcf(120,85)=5
gcf(120,86)=2
gcf(120,87)=3
gcf(120,88)=8
```

In CoffeeScript, numbers are unquoted. The language treats numbers and strings differently: `5 + 3` is 8, while `'5' + '3'` is "53".

Numerical Conversion

Strings can be parsed to numbers using the `Number` function; the `String` function does the opposite.

CoffeeScript allows numbers and strings to be mixed, but you should be careful when doing it. Adding a number to a string will convert the number to a string and attach it. Multiplying a number by a string will convert the string to a number and do the numerical product.

Three Ways to Change a Variable

There are three types of statements that change the value of a variable.

`++counter` the increment operator. Putting `++` before the variable name increments the value before it is used, and putting `++` after the variable increments it after it is used. The `--` decrement is similar.

`counter += 1` the sum assignment operator, which changes a variable by adding a value. There are also `-=`, `*=`, and `/=` operators.

`counter = counter + 1` the ordinary assignment operator. Notice that the right hand side is computed *before* the left hand side is changed.

Floating Point Limits

Coffeescript uses *IEEE 754* "double-precision" floating-point numbers, which means numbers are stored using 64 bits. Scientific notation is written with an `e+` or `e-` followed by a power of 10: `1e+6` is one million and `1e-9` is one billionth.

There are 15 digits of precision, and every integer up to 9,007,199,254,740,992 can be written exactly. There are also special `Infinity` and `NaN` ("Not a Number") values. However, not every real number can be represented exactly: the next number after zero is `5e-324` and the largest number is `1.79e+308`.

The limits are expansive, so for most practical purposes, you can treat CoffeeScript numbers as if they were real numbers.

The Modulo Operator

The *modulo* operator `x % y` computes the remainder of `x` when divided by `y`. In other words, it removes the largest integer multiple of `y` from `x` and returns the remainder.

The modulo operator is useful for divisibility tests: `x % y` is zero if `x` is divisible by `y`. Euclid's famous algorithm uses the modulo operator to efficiently compute greatest common factors.

11. Computation

Power

```
power = (x, p) ->
  answer = 1
  answer *= x for i in [0...p]
  return answer
for n in [1..5]
  write power(2, n)
```

```
2
4
8
16
32
```

Built-in Power

```
write Math.pow(2, 5)
write Math.pow(2, 0.5)
```

```
32
1.4142135623730951
```

Factorial

```
factorial = (x) ->
  if x < 1 then 1
  else x * factorial(x - 1)
for x in [1..4]
  write factorial x
```

```
1
2
6
24
```

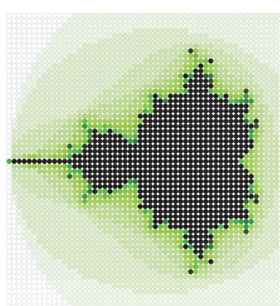
Fibonacci

```
fib = (n) ->
  if n <= 2
    1
  else
    fib(n - 1) + fib(n - 2)
for x in [3..8]
  write fib x
```

```
2
3
5
8
13
21
```

Complex

```
mandelbrot = (n, c, z) ->
  if n is 0 or z.r*z.r + z.i*z.i > 4
    return n
  else return mandelbrot n - 1, c,
    r: c.r + z.r*z.r - z.i*z.i
    i: c.i + 2*z.r*z.i
speed 100
ht()
scale 150
s = 0.05
for x in [-2..1] by s
  for y in [-1.5..1.5] by s
    n = mandelbrot 20, {r:x,i:y}, {r:x,i:y}
    moveto x, y
    dot hsl(100, 1, n/20), s
```



The `Math` object provides constants and functions you would find on a scientific calculator. A partial list:

`Math.E` the natural logarithm base, 2.71828...
`Math.PI` the circular ratio, 3.14159...
`Math.abs(x)` absolute value of x.
`Math.round(x)` round x to the ***nearest*** integer.
`Math.floor(x)` round x ***down***.
`Math.ceil(x)` round x ***up***.
`Math.max(x, y)` the greater of x and y.
`Math.min(x, y)` the lesser of x and y.
`Math.sqrt(x)` the square root of x.
`Math.pow(x, y)` x raised to the power y.
`Math.log(x)` the natural logarithm of x.
`Math.sin(x)` the sine of x (in radians).
`Math.cos(x)` the cosine of x (in radians).
`Math.atan(x)` the arctangent of x (in radians).

Returning Values, Recursion, and Base Cases

Other mathematical functions can be built yourself. The output, or ***return value***, of a CoffeeScript function is the last value computed in the function. The statement `return n` ends a function with the return value `n`.

The functions `fib` and `factorial` are are ***recursive***: they refer to themselves in their own definition. When writing a recursive function it is important that the recursion ends at a ***base case*** (such as where `fib` defines the value as 1 when `n <= 2`).

Recursion without a base case will loop forever and freeze up. There must be initial values for which the function does not depend on itself.

Generalizing

Although the built-in numbers represent reals, complex numbers can be represented as pairs of numbers. In **Mandelbrot**, the parameters `c` and `z` are complex numbers represented by objects that each contain an `r` and `i` property.

That example uses `scale 150` to grow the turtle by 150-fold. The `hsl` function generates colors based on hue, saturation, and lightness.

Mathematical algorithms have a long and fascinating history. It is worth researching how Mandelbrot's remarkable fractal works; how Gauss's Gamma function generalizes factorials to all numbers; and how the Fibonacci sequence relates to sunflower seeds and the golden mean.

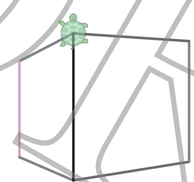
12. Objects

Page Coordinates

```
startpos =
  pageX: 80
  pageY: 10
moveto startpos
pen coral
moveto
  pageX: 30
  pageY: 50
moveto {pageX: 160, pageY: 50}
```

Figure

```
figure = [
  {c: dimgray, x: 75,  y: 12}
  {c: gray,    x: 0,   y: 78}
  {c: dimgray, x: -75, y: 5}
  {c: gray,    x: -35, y: -18}
  {c: plum,    x: 0,   y: -62}
  {c: gray,    x: 35,  y: -15}
  {c: black,   x: 0,   y: 95}
  ]
for line in figure
  pen line.c
  slide line.x, line.y
```



Scoring

```
points =
  a: 1, e: 1, i: 1, l: 1, n: 1, o: 1, r: 1, s: 1, t: 1, u: 1
  d: 2, g: 2, b: 3, c: 3, m: 3, p: 3, f: 4, h: 4, v: 4, w: 4, y: 4
  k: 5, j: 8, x: 8, q: 10, z: 10
score = (word) ->
  total = 0
  for letter in word
    total += points[letter]
  write "#{word}: #{total}"
score x for x in ['bison', 'armadillo', 'giraffe', 'zebra']
```

bison: 7
armadillo: 12
giraffe: 14
zebra: 16

Methods

```
memo =
  sum: 0
  count: 0
  add: (x) -> @sum += x; @count += 1
  stats: ->
    write "Total #{this.sum} / #{this.count}"
    write "Average #{this.sum / this.count}"
memo.add(n) for n in [40..50]
memo.stats()
```

Total 495 / 11
Average 45

An ***object*** is a value that has its own ***properties***. Each property of an object associates a name with a value. The object `startpos` has two properties: `pageX`, which has value 80, and `pageY`, which is 10.

The `moveto` function understands objects with a `pageX` and `pageY` property as a "page coordinate." (Page coordinates measure distances from the top-left corner of the page instead of from the center.)

Object Literals

In the **Page Coordinate** example, we can see that there are two styles for writing object literals in CoffeeScript. Each property can be put on separate lines, indented (*YAML style*); or the properties can be enclosed in curly braces and separated by commas (*JSON style*). The two styles are equivalent, and the program uses both.

Dot Notation

The properties of an object are referenced using a dot: `line.x` refers to the value of the property named "x" in the object named "line".

The most common use of objects is as a way of encapsulating a packet of related data together: in **Figure**, each object bundles the data needed for one line: a color and an x, y displacement.

Associative Array Notation

A property name can be any string, so an object can be used as an ***associative array*** that defines a map from strings to values.

In **Scoring**, `points` maps letters to point values. The square bracket notation `points[letter]` means "look up the value of the property whose name is the value of `letter`."

Mutation and Methods

Properties of an object may be changed by assigning a value using the normal `=` or `+=` or `++` variable-setting operators. (Changing a property of an object is sometimes called ***mutation***.)

Properties of an object that happen to be functions are called ***methods***. Methods are particularly useful, because they can use the word `this` or the symbol `@` to refer to the object on which the method was called.

(Note that the line `memo.add for n in [40..50]` puts the `for` at the end of the statement in order to repeat it.)

It is common to write methods like `memo.add` that mutate several properties of the object at once, or methods like `memo.stats` that do computation summarizing the properties of the object.

13. Arrays

Story

```
story = [
  'Exclamation?'
  '!  he said '
  'adverb?'
  ' as he jumped into his convertible '
  'noun?'
  ' and drove off with his '
  'adjective?'
  ' wife.'
  ]
for i in [0...story.length] by 2
  prompt = story[i]
  await read prompt, defer answer
  story[i] = answer
write story.join ''
```

Exclamation? Yowzer
adverb? slowly
noun?

Primes

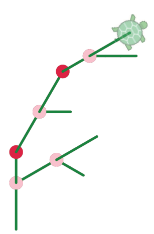
```
primes = []
candidate = 2
while primes.length < 10
  composite = false
  for p in primes
    if candidate % p is 0
      composite = true
      break
  if not composite
    primes.push candidate
    write candidate
  candidate = candidate + 1
```

2
3
5
7
11
13
17
19
23
29

Push and Pop

```
stack = []
pen green
speed Infinity
button 'R', -> rt 30
button 'F', -> fd 10
button 'Push', ->
  dot crimson
  stack.push [getxy(), direction()]
button 'Pop', ->
  if not stack.length then home(); return
  [xy, b] = stack.pop()
  jumpto xy
  turnto b
  dot pink
```

R F Push Pop



Arrays are objects that contain a sequence of values. Throughout this book we have used arrays for iteration in `for` loops; arrays are used wherever a program needs to organize sequential data.

Referencing and Joining Array Elements

The `i` th element of an array `story` is `story[i]`, and the number of elements is `story.length`. Indexing is zero-based, so the first element is `story[0]` and the last is `story[story.length - 1]`.

All the elements of an array can be joined together in one big string by `story.join ''`. The argument is the "glue" put between the elements.

Await and Defer

The statement `await read prompt, defer answer` pauses the program until the `read` is done. `defer answer` is a *continuation* funtion that resumes the program after putting the result in `answer`.

Building Arrays with Push

A program can use `push` to add elements to the end of an array.

Primes starts with `primes = []` as an empty array, and then it calls `primes.push candidate` to add each discovered prime to the array of divisors to check. This ancient algorithm is the ***Sieve of Eratosthenes***.

Stacks of Objects

Arrays have a `pop` method that reverses of `push` by removing and returning the last value. An array used by pushing and popping is called a *stack*.

It is common to use a stack of objects to undo a sequence. In **Push and Pop**, `stack` is an array where every element is a turtle position. Each element is itself a two-element array containing an [x, y] (itself another array) and a numerical direction.

Destructuring

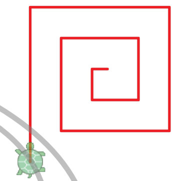
`getxy()` returns the turtle's current [x, y] as an array of two numbers.
`direction()` returns the current direction of the turtle in degrees.
`stack.push [getxy(), direction()]` reads the turtle's current xy coordinates and its current direction, forms an array with the results, and pushes it on the `stack`.
`[xy, b] = stack.pop` removes the last element from the `stack` (the element is itself an array), and assigns the first item within of the element to the variable `xy` and the second item in element to `b`.

The form `[xy, b] = value` is called a ***destructuring assignment***. It is a concise way to give local variable names to the elements of a short array.

14. Recursion

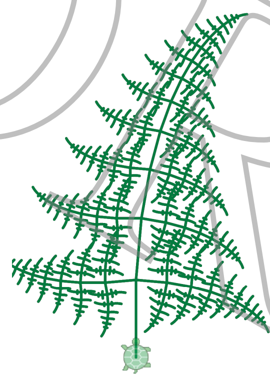
Recursive Spiral

```
spiral = (x) ->
  if x > 0
    fd x * 10
    rt 90
    spiral x - 1
    lt 90
    bk x * 10
pen red
spiral 10
```



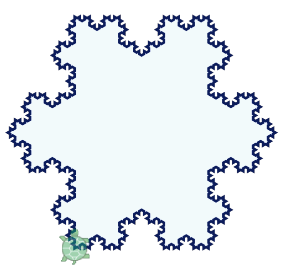
Fractal Fern

```
speed 1000
fern = (x) ->
  if x > 1
    fd x
    rt 95
    fern x * .4
    lt 190
    fern x * .4
    rt 100
    fern x * .8
    lt 5
    bk x
pen green
fern 50
```



Koch Snowflake

```
speed Infinity
flake = (x) ->
  if x < 3 then fd x
  else
    flake x / 3
    lt 60
    flake x / 3
    rt 120
    flake x / 3
    lt 60
    flake x / 3
pen 'path'
for s in [1..3]
  flake 150
  rt 120
fill 'azure strokeStyle navy'
```



Recursive functions refer to themselves, and they can achieve powerful effects. Recursion is at the core of fractals, language, and reasoning.

Recursion as a Stack

Operationally, recursion works by stepping through a stack of work. Consider the sequence as **Spiral** draws a shape and retraces it back.

```
spiral 10  sets x to 10
  rt 90; fd x * 10; spiral x - 1 ⇓                lt 90; bk x * 10 ⇑
```

```
  spiral 9  sets x to 9
    rt 90; fd x * 10; spiral x - 1 ⇓            lt 90; bk x * 10 ⇑
```

```
    spiral 8  sets x to 8
      rt 90; fd x * 10; spiral x - 1 ⇓        lt 90; bk x * 10 ⇑
```

... etc, until the base case `spiral 0` ⇑

Each time `spiral` is called, it puts the previous call on hold and does the smaller spiral. After the smaller spiral is done, it returns to finish work on the bigger one. `spiral 0` does nothing: that is called the ***base case***.

The `x` at different levels are ***local*** variables that do not interfere with each other. Each red box is a ***stack frame*** with its own "copy" of `x`.

Recursion as a Reduction

Conceptually, recursion reduces a problem to smaller cases. Consider how **Fern** draws a large fern by assuming it can draw smaller ferns:

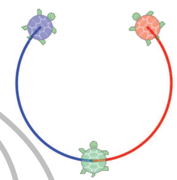
All `fern` does is draw a stem with three smaller ferns at the end. The main caveat is that the reduction has a limit: it ends when x ≤ 1.

Both **Spiral** and **Fern** return the turtle to exactly the same position and direction at the end of a function call. Maintaining an ***invariant*** like this can make recursion much easier to understand.

15. Concurrency

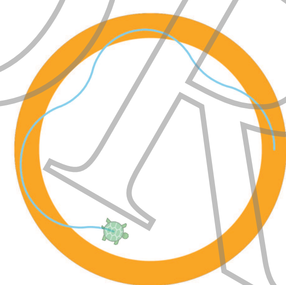
Race Condition

```
b = hatch blue
r = hatch red
b.lt 90; b.pen blue
b.play 'g'
b.rt 170, 50
b.dot 50, blue
r.rt 90; r.pen red
r.play 'd'
r.lt 170, 50
r.dot 50, red
```



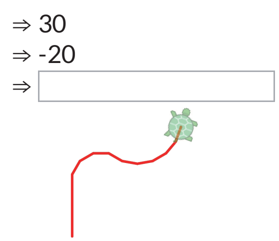
Line Follower

```
dot orange, 220
dot white, 180
jump 100, 0
pen skyblue
while true
  fd 3 + random 3
  await done defer()
  if touches orange
    lt 5
  else
    rt 5
```



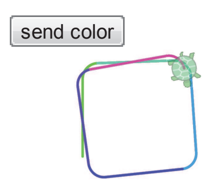
Shared Memory

```
shared = { d: 0 }
do ->
  while true
    await read defer shared.d
do ->
  pen red
  while true
    fd 10
    await done defer()
    rt shared.d
```



Message Passing

```
button 'send color', ->
  send 'go', random color
do ->
  for x in [1..25]
    await recv 'go', defer c
    pen c
    fd 50
    rt 88, 10
```



A *thread* is a sequence in a program that runs in parallel to other code. Iced CoffeeScript has ***cooperative threads***, which means that:

Only one thread runs at once.
Concurrency is done by switching between threads.
Switching is only done at `await` statements.

Some new idioms that appear in these examples:

`b = hatch blue` hatches a new turtle, wearing blue.
`b.lt 90` tells the turtle `b` to turn.
`await done defer()` waits until turtles stop moving.
`send 'go'` sends a message 'go'.
`await recv 'go', defer()` waits until 'go' is received.
`while true` repeats the enclosed code forever.
`touches orange` tests if the turtle touches any orange.
`do ->` runs the enclosed code as a function.

Multiple Turtles

In **Race Condition**, the second turtle to arrive will draw a dot that covers the first dot. The turtles run concurrently, and it is is not possible to predict which parallel turtle will arrive first.

If order is important, insert `await b.done defer()` before calling `r`. The program will wait for `b` to finish before moving the red turtle.

An `await` only pauses the current function, not its caller. That is why the last two examples run threads in parallel.

Synchronization

It is important to let a turtle finish moving before reading its state. If **Line Follower** did not `await done defer()` to let the turtle finish moving forward before checking the touched color, the turtle would still be in its start position when check is done.

Even if not reading turtle state, an infinite `while true` loop should contain an `await done defer()` so that other threads get a turn.

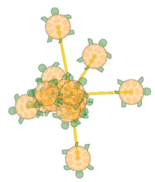
Communicating Between Threads

Threads can communicate using ***shared memory*** (sharing a common variable) or ***message passing*** (sending a value from one to the other). If a shared variable changes very quickly or slowly, the thread that reads the variable can skip a value or read the same value twice. On the other hand, a thread that uses `await recv` will wait to receive each message sent by `send` exactly once without duplication or omission.

16. Sets

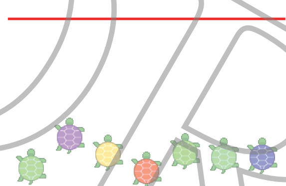
Scatter

```
turtle.remove()
s = hatch 15, orange
s.pen gold
s.plan ->
  this.rt random 360
  this.fd Math.abs(20 * random normal)
```



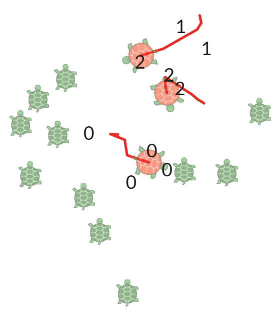
Turtle Race

```
fd 200; pen red; slide 200, 0
finished = 0
racers = hatch 7
racers.plan (j) ->
  @wear random color
  @speed 5 + random normal
  @slide j * 25 + 25, 0
  while not @touches red
    @fd random 5
    await @done defer()
  @label ++finished
```



Rescue Class

```
turtle.remove()
speed 100
randpos = ->
  [50 * random(normal), 50 * random(normal)]
hatch(20, green).scale(0.75).plan ->
  this.moveto randpos()
  this.addClass 'kid'
hatch(3, red).plan (num) ->
  hero = this
  count = 0
  hero.moveto randpos()
  hero.pen red
  while true
    await hero.done defer()
    kid = $('.kid').nearest(hero).eq(0)
    if kid.length is 0
      write "hero ##{num} got #{count}"
      return
    else if hero.touches(kid)
      count += 1
      kid.label num
      kid.remove()
    else
      hero.turnto(kid).fd(5)
```



Turtles are ***jQuery sets***. Although most sets we have worked with contain a single turtle, a set can contain any number of elements. `hatch 15` makes a set of 15 new turtles, and `$('.turtle')` is the set of all turtles.

JQuery Set Methods

Methods operating on a jQuery set `s` can:
Generate a related set: `s.nearest [0, 0]` is the subset nearest 0, 0.
Manipulate the elements: `s.fd 100` advances the elements by 100.
Test the elements: `s.touches red` tests pixels under the first element.

Generally a manipulation method like `s.fd 100` will do the same thing to every element of the set. However, the method `s.plan` applies a function that can run a distinct operation on each element.

Giving Turtles Individualized Plans

When `s.plan (j) -> action` runs, The action is done for each element with the following parameters:

`this` (aka `@`) is a jQuery set with the single element.
`j` is the element index, ranging from `0` to `crowd.length - 1`.

For example, **Scatter** uses `plan` to direct each turtle to turn and move a different random amount. The function call `random normal` returns a *normally distributed* random number with mean 0 and variance 1.

The program **Turtle Race** is similar, but it also uses an `await` loop to run the seven turtles in a parallel race. On each iteration, the turtles individually check if they have crossed the red line. The shared variable `finished` tracks the order in which the turtles finish.

Using and Selecting Classes

The loop in **Rescue Class** finds the nearest kid to each hero and removes that kid if the hero touches it. Otherwise the hero turns and moves towards the nearest kid and repeats the process.

At the beginning of that program, all the kids are marked with a class using `this.addClass('kid')`. On the hero thread, the jQuery *selector* `$('.kid')` obtains the set of all current elements in the `kid` class that have not yet been removed.

jQuery methods that return sets can be chained. For example, `$('.kid').nearest(hero).eq(0)` filters the set of kids to the subset nearest `hero`, and then filters that subset to its first element, if any.

There are a wide range of jQuery methods for finding and manipulating sets: much about jQuery has been written on the web.

17. Text

```
text = """If you can look into the seeds of time
          And say which grain will grow and which will not,
          Speak, then, to me."""
```

Substr

```
see text.indexOf 'which'
see text.substr 47, 7
```

```
47
which g
```

Unicode

```
see 'charCode', text.charCodeAt(0)
see 'string', String.fromCharCode(73)
for x in [88, 188, 9988]
  see x, String.fromCharCode(x)
```

```
charCode 73
string I
88 X
188 ¼
9988 ✄
```

Match

```
see text.match /w....g.../
see text.match /[a-z][a-z]/
see text.match /\s[a-z][a-z]\s/
see text.match /\b[a-z][a-z]\b/
see text.match /\b[a-z][a-z]\b/gi
see text.match /\b[gn][a-z]*\b/g
see text.match /z/
```

```
["will grow"]
["yo"]
[" of "]
["of"]
["If", "of", "to", "me"]
["grain", "grow", "not"]
null
```

Split

```
lines = text.split /\n/
see lines[2]
words = text.split /\s+/
see words[0..2]
```

```
Speak, then, to me.
["If", "you", "can"]
```

Groups

```
pattern = /\b([a-z]+) of ([a-z]+)\b/
matched = pattern.exec text
for g in [0..2]
  see "group #{g}: #{matched[g]}"
```

```
group 0: seeds of time
group 1: seeds
group 2: time
```

Replace

```
r = text.replace /[A-Z][a-z]*/g,
    "<mark>$&</mark>"
r = r.replace /\n/g,
    "<br>"
r = r.replace /\bw[a-z]*\b/g,
    (x) -> x.toUpperCase()
write r
```

If you can look into the seeds of time
And say WHICH grain WILL grow and WHICH WILL not,
Speak, then, to me.

String algorithms to locate patterns in text are a fundamental tool for understanding written language.

Characters

Strings are arrays of ***characters***. The ***Unicode character set*** supports textual communication around the world, so it includes characters for every international alphabet, every Asian pictographic word, and every common mathematical symbol.

Unicode assigns a number to every character. `String.fromCharCode` gets the character for the number, and `text.charCodeAt` gets the number for a character. Numbers up to 127 are ***ASCII*** codes that cover American English: 65 is uppercase A, 122 is lowercase z, 48 is the 0 digit, 32 is the space, and 36 is the $ dollar symbol.

Locating Substrings in Text

The simplest way to match text is to find an exact substring: `text.indexOf` returns the location of the first ocurrance of the given substring in `text`, or `-1` if none was found. Conversely, if you have an index of interest, `text.substr x, len` returns the `len` characters starting at index `x`.

Matching Patterns

Regular expressions, are flexible and precise text patterns written between pairs of slash `/.../` delimiters. Regular expression syntax is a whole language that is the topic of several good books and websites. Here are a few basics:

`(abc)*` matches zero or more repetitions of abc.
`[abc]` matches either an a or b or c.
`a..c` matches a followed by two characters then c.
`ab+c` matches a, then one or more b's, then c.
`[a-z]{3}` matches three lowercase letters.
`\d*` matches zero or more digits.
`x\s+` matches x followed by one or more spaces.
`z\b` matches a z followed by a word boundary.

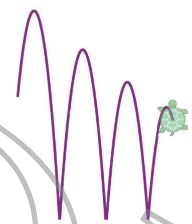
The `text.match` method returns matching substrings. Normally, the first match is found, but if the letter `g` (for "global") follows the pattern then all matching substrings are returned. There are other useful suffixes: `i` makes the match case-insensitive.

The `pattern.exec` method extracts of submatches within parentheses in the pattern, and the `text.replace` method replaces matches with a new string.

18. Motion

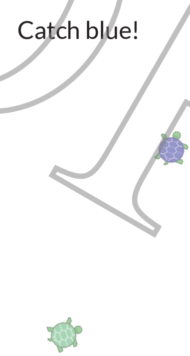
Bounce

```
speed Infinity
pen purple
vy = 10
tick 20, ->
  slide 1, vy
  if inside(window)
    vy -= 1
  else
    vy = Math.abs(vy) * 0.9
```



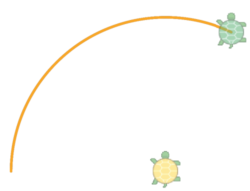
Tag

```
speed Infinity
write "Catch blue!"
b = hatch blue
bk 100
tick 10, ->
  turnto lastmousemove
  fd 5
  b.turnto 45 + direction b
  b.fd 6
  if b.touches(turtle)
    write "You win!"
    tick off
  else if not b.touches(window)
    write "Blue got away!"
    tick off
```



Orbit

```
speed Infinity; pen orange
G = 100
v = [0, 1]
sun = hatch(gold)
sun.slide G, 0
tick 100, ->
  sun.moveto lastclick
  s = sun.getxy()
  p = getxy()
  d = distance(sun)
  d3 = d * d * d
  if d3 > 0 then for i in [0..1]
    v[i] += G * (s[i] - p[i]) / d3
  slide v[0], v[1]
```



The thee examples on this page demonstrate how to simulate motion: a bouncing turtle, a game of tag, and an orbiting planet.

Newtonian Simulations

When Newton worked out his famous laws of motion, he discovered that the speed and direction of an object - its ***velocity*** - remains unchanged as long as no forces act on the object. And he discovered that forces do not directly change the position of an object: forces alter an object's velocity.

When simulating motion, the velocity of an object can be represented by a small change in position for each tick in time. An undisturbed object moves the same distance and direction on each tick, and a forced object will alter its velocity on each tick.

In **Bounce**, the two variables `vx` and `vy` are the x and y components of velocity. The gentle accelleration due to gravity is simulated by a slight change in velocity on each tick: `vy -= 1`. The sudden accelleration of a bounce off the floor (with some loss in energy) is represented by a sign change in velocity: `vy = Math.abs(vy) * 0.9`.

In **Tag**, velocity is simulated by moving each turtle forward 5 or 6 on each tick. The physics of this game are designed for fun: the main turtle picks its direction by pointing at the last position of the mouse. The blue turtle runs away by adding 45 degrees to the `direction` from the main turtle to itself.

Orbit is a representation of Newton's most profound discovery: that the gravity makes objects fall to the ground is the same force that governs the motions of the planets. In the orbital simulator, the x and y components of velocity are in the array `v`, and the velocity is accellerated on each tick using the formula `v[i] += G * (s[i] - p[i]) / d3`, where s is the position of the sun, p is the position of the planet, and d3 is the cube of the distance between them.

Click to move the sun. Experiment with elliptical and hyperbolic orbits. Notice the planet moves more quickly when it is near the sun.

Motion and Hit Testing Functions

`slide x, y` slides right by x and forward by y.
`getxy()` returns the absolute [x, y] position of the turtle.
`b.touches(turtle)` true if `b` touches the main turtle.
`inside(window)` true if the main turtle is fully inside the window.
`direction(b)` the direction from the turtle to `b`.
`distance(sun)` the distance from the turtle to `sun`.

19. Randomness

The `random` function can be used in several ways:

`random [1..6]` chooses a random member of a list.
`random 70` chooses a random integer from 0 to 69.
`random position` picks a random screen position.
`random color` picks a random color.
`random normal` picks a normally distributed number.

All of these are built using the lower-level function:

- `Math.random()` returns a random number between 0.0 and 1.0.

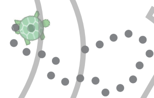
Two Dice

```
onedice = ->
  random [1..6]
twodice = ->
  onedice() + onedice()
for n in [1..5]
  write twodice()
```

6
9
11
8
10

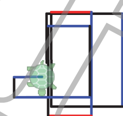
Random Walk

```
for n in [1..20]
  fd 10
  rt random(181) - 90
  dot gray, 5
```



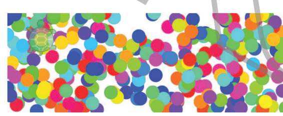
Cubism

```
for n in [1..14]
  pen random [red,black,blue]
  fd random 70
  rt 90
```



Confetti

```
for n in [1..300]
  moveto random position
  dot random color
```



Decimal Random

```
for n in [1..2]
  write Math.random()
```

0.3955826204144705
0.46279336348825273

Five Flips

```
c = [0, 0, 0, 0, 0, 0]
for n in [1..500]
  heads = 0
  for flips in [1..5]
    heads += random 2
  c[heads] += 1
for h of c
  b = write h + ":" + c[h]
  b.css
    background: skyblue
    width: c[h]
```

0:21
1:73
2:160
3:145
4:85
5:16

20. Styles

Thick Lines

```
pen blue, 10
fd 100; rt 90
pen pink, 3
fd 50; rt 90
pen 'orange ' +
    'lineWidth 10 ' +
    'lineCap square'
fd 100; rt 90
pen black
fd 50
```

Border

```
text = write 'Outlined.'
text.css { border: '2px solid red' }
turtle.css { border: '3px dotted blue' }
```

Outlined.

Font

```
h = write 'Fancy!'
h.css
  font: '55px Helvetica'
  fontStyle: 'italic'
```

Fancy!

Text Decoration

```
write 'Before'
d = write 'Decorated'
write 'After'
d.css
  display: 'inline-block'
  cursor: 'pointer'
  padding: '10px'
  margin: '-5px'
  opacity: '0.7'
  color: 'white'
  fontSize: '110%'
  letterSpacing: '5px'
  textDecoration: 'underline'
  boxShadow: '1px 1px black'
  background: 'mediumaquamarine'
  transform: 'rotate(10deg)translateX(20px)'
```

Before

Decorated

After

About styles. TBD.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

21. Selectors

About selectors. TBD.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Tags

```
write """
<style>
h2 { color: red; }
h3 { background: bisque; }
</style>
"""
write "<h2>Stylesheet</h2>"
write "<h3>Tag Styles</h3>"
write "<p>style specific tags</p>"
```

Stylesheet

Tag Styles

style specific tags

Classes

```
write """
<style>
.a { text-decoration: underline; }
.b { font-style: italic; }
</style>
"""
write "<p class='a'>Class a</p>"
write "<h3 class='b'>Class b</h3>"
write "<p class='b'>Classes apply to any tag.</p>"
```

Class a

Class b

Classes apply to any tag.

Composites

```
write """
<style>
i { border: 1px solid black; margin: 2px }
i:nth-of-type(1) { background: gold }
i:nth-of-type(2n+4) { background: skyblue }
i:nth-of-type(3n+9) { background: thistle }
</style>
"""
for x in [1..24]
  write "<i>#{x}</i>"
  write "<wbr>"
```

1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19
20 21 22 23 24

jQuery

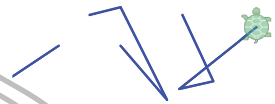
```
write "<p><mark>a</mark>v<mark>o</mark>" +
      "c<mark>a</mark>d<mark>o</mark></p>"
$('p').css { fontSize: '200%' }
$('mark').css { background: palegreen }
$('mark').animate {
  padding: '5px' }
$('mark:nth-of-type(2n)').animate {
  opacity: 0.3 }
```

a v o c a d o

22. Events

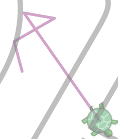
Shift Click

```
$(document).click (event) ->
  see event
  if event.shiftKey
    pen blue
  else
    pen null
  moveto event
```



Arrow Keys

```
pen plum
[L, R, U, D] = [37, 39, 38, 40]
keydown (event) ->
  if event.which is L then lt 5
  if event.which is R then rt 5
  if event.which is U then fd 5
  if event.which is D then bk 5
```



Can't Touch This

```
t = write "<button>Touch This</button>"
t.speed Infinity
t.moveto document
t.mousemove (event) ->
  t.rt random(91) - 45
  while t.touches(event)
    t.bk 1
```

Magic Hat

```
speed Infinity
turtle.remove()
t = write '<img>'
t.home()
start = ->
  t.wear 'openicon:magic-tophat'
  tick off
  t.click (event) -> play()
play = ->
  t.wear 'openicon:animals-rabbit'
  tick ->
    t.moveto random 'position'
  t.click (event) -> start()
start()
```

About events. TBD.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

23. Slicing

Choices

```
choices = (menu, sofar = []) ->
  if menu.length is 0
    write sofar.join ' '
  else for item in menu[0]
    choices menu[1...],
      sofar.concat item

choices [
  ['small', 'medium', 'large']
  ['vanilla', 'chocolate']
  ['cone', 'cup']
]
```

small vanilla cone
small vanilla cup
small chocolate cone
small chocolate cup
medium vanilla cone
medium vanilla cup
medium chocolate cone
medium chocolate cup
large vanilla cone
large vanilla cup
large chocolate cone
large chocolate cup

Shuffle

```
suits = ['\u2663', '\u2666', '\u2665', '\u2660']
deck = []
for v in [2..10].concat ['J', 'Q', 'K', 'A']
  deck.push (v + s for s in suits)...
shuffle = (d) ->
  for i in [1...d.length]
    choice = random(i + 1)
    [d[i], d[choice]] = [d[choice], d[i]]
deal = (d, n) -> d.splice(-n)

shuffle deck
for x in [1..3]
  write deal(deck, 5).join('/')
```

J♦/3♠/7♣/9♥/6♠
3♥/10♦/7♥/7♦/8♥
A♦/Q♥/2♣/8♠/K♦

Caesar Cipher

```
key = 13
a2z = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
rot = a2z[key...].concat a2z[...key]
box = write '<input>'
out = write ''

box.keyup ->
  result = for c in box.val()
    char = c.toUpperCase()
    if char in a2z
      rot[a2z.indexOf char]
    else
      char
  out.text result.join ''
```

attack at dawn

NGGNPX NG QNJA

About slicing. TBD.

The `concat` method creates an array that joins the elements of two arrays together. `[2...10].concat ['J', 'Q', 'K', 'A']` forms an array starting with the numbers 2 through 10 followed by the strings "J", "Q", "K", and "A".

The parenthesized loop `(v + s for s in suits)` is a ***list comprehension*** that creates an array using a for loop. The array consists of each value computed by the loop, in sequence.

Shuffle applies the *Fisher-Yates* algorithm for shuffling a deck of cards. The algorithm is called a "perfect shuffle" because every permuatation of the deck is equally likely.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

24. Sorting

Quick Sort

```
list = (random 10 for x in [1..8])
list.sort()
write list
```

3,4,4,5,6,7,7,8

Slow Selection Sort

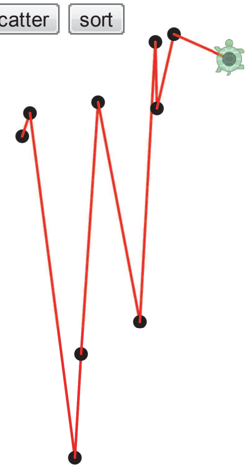
```
show = (points, highlight) ->
  render = for k, v of points
    if Number(k) in highlight
      "<mark>#{v}</mark>"
    else
      "#{v}"
  write "<div>#{render.join ','}</div>"

list = 'SORTME'.split ''
show list, []

for i in [0 ... list.length - 1]
  for j in [i + 1 ... list.length]
    if list[i] > list[j]
      [list[i], list[j]] =
        [list[j], list[i]]
    show list, [i, j]
```

S,O,R,T,M,E
O,S,R,T,M,E
O,S,R,T,M,E
O,S,R,T,M,E
M,S,R,T,O,E
E,S,R,T,O,M
E,R,S,T,O,M
E,R,S,T,O,M
E,O,S,T,R,M
E,M,S,T,R,O
E,M,S,T,R,O
E,M,R,T,S,O
E,M,O,T,S,R
E,M,O,S,T,R
E,M,O,R,T,S
E,M,O,R,S,T

Custom Quick Sort

```
sketch = (points) ->
  cg()
  pen null
  for p in points
    moveto p
    pen red
    dot black

array = []

button 'scatter', ->
  array = for x in [1..10]
    random 'position'
  sketch array

button 'sort',  ->
  array.sort (a, b) ->
    a.pageX - b.pageX
  sketch array
```

scatter sort



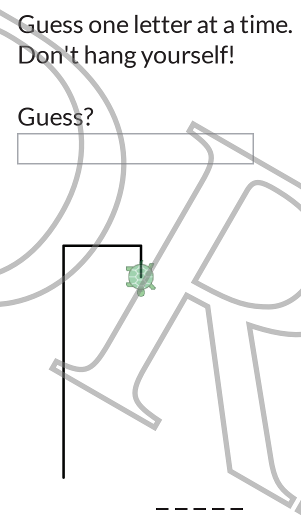
About sorting. TBD.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

25. Networks

Hangman

```
write "Guess one letter at a time.  Don't hang yourself!"
await $.get 'http://turtlebits.net/data/animals', defer file
secret = random file.split '\n'
hung = false
blanks = write ''
blanks.home()
blanks.css { fontSize: '20px' }

do ->
  guessed = []
  wrong = 0
  while wrong < 6
    missing = 0
    hint = ''
    for letter in secret
      if letter in guessed
        hint += " #{letter} "
      else
        hint += " _ "
        missing += 1
    blanks.html "#{hint}"
    if missing is 0
      write 'You win!'
      return
    await read 'Guess?', defer letter
    guessed.push letter
    if letter not in secret
      write 'Sorry'
      send 'hang'
      wrong += 1
  write "Game over.  It was #{secret}."

do ->
  pen black; fd 150; rt 90
  fd 50; rt 90; fd 20
  await recv 'hang', defer()
  lt 90; rt 540, 10; lt 90
  await recv 'hang', defer()
  fd 20; lt 45; bk 30; fd 30
  await recv 'hang', defer()
  rt 90; bk 30; fd 30; lt 45
  await recv 'hang', defer()
  fd 30
  await recv 'hang', defer()
  rt 45; fd 30
  await recv 'hang', defer()
  bk 30; lt 90; fd 30
```



Web pages make network requests by sending ***AJAX requests*** using functions like the jQuery methods `$.get` and `$.ajax`.

URLs and HTTP

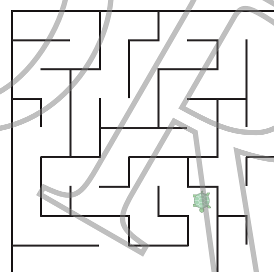
Every page on the web has a URL, which has five main parts:
The protocol.
The server address.
The path.
The query parameters.
The hash parameter.

Message Passing

26. Search

Maze

```
[width, height] = [9, 9]
grid = table(width, height).home()

sides = [
  {dx: 0, dy: -1, ob: 'borderTop', ib: 'borderBottom'}
  {dx: 1, dy: 0, ob: 'borderRight', ib: 'borderLeft'}
  {dx: 0, dy: 1, ob: 'borderBottom', ib: 'borderTop'}
  {dx: -1, dy: 0, ob: 'borderLeft', ib: 'borderRight'}
]

isopen = (x, y, side) ->
  return /none/.test(
    grid.cell(y, x).css side.ob)

isbox = (x, y) ->
  return false unless (
    0 <= x < width and
    0 <= y < height)
  for s in sides
    if isopen x, y, s
      return false
  return true

makemaze = (x, y) ->
  loop
    adj = (s for s in sides when isbox x + s.dx, y + s.dy)
    if adj.length is 0 then return
    choice = random adj
    [nx, ny] = [x + choice.dx, y + choice.dy]
    grid.cell(y, x).css choice.ob, 'none'
    grid.cell(ny, nx).css choice.ib, 'none'
    makemaze nx, ny

wander = (x, y, lastdir) ->
  moveto grid.cell y, x
  for d in [lastdir + 3 .. lastdir + 7]
    dir = d % 4
    s = sides[dir]
    if isopen x, y, s then break
  turnto grid.cell y + s.dy, x + s.dx unless dir is lastdir
  plan -> wander x + s.dx, y + s.dy, dir

makemaze 0, 0
speed 5
wander 4, 4, 0
```



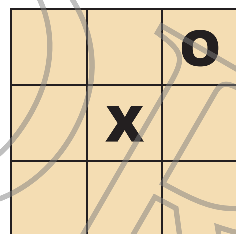
About search. TBD.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

27. Intelligence

Tic Tac Toe

```
grid = table 3, 3,
  {width: 48, height: 48, font: "32px Arial Black", background: "wheat"}
grid.home()
board = [0, 0, 0,   0, 0, 0,   0, 0, 0]

grid.cell().click ->
  move = grid.cell().index this
  return unless winner() is 0 and board[move] is 0
  board[move] = 1
  $(this).text 'X'
  setTimeout respond, 500

respond = ->
  response = bestmove(-1).move
  if response?
    board[response] = -1;
    grid.cell().eq(response).text 'O'
  colorwinner()

bestmove = (player) ->
  win = winner()
  if win isnt 0 then return {move: null, advantage: win}
  choices = {'-1': [], '0': [], '1': []}
  for think in [0..8] when board[think] is 0
    board[think] = player
    outcome = bestmove(-player).advantage
    choices[outcome].push {move: think, advantage: outcome}
    board[think] = 0
  for favorite in [player, 0, -player] when choices[favorite].length
    return random choices[favorite]
  return {move: null, advantage: 0}

rules = [[0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],[2,5,8],[0,4,8],[2,4,6]]

winner = ->
  for row in rules
    if board[row[0]] and board[row[0]] is board[row[1]] is board[row[2]]
      return board[row[0]]
  return 0

colorwinner = ->
  for row in rules
    if board[row[0]] and board[row[0]] is board[row[1]] is board[row[2]]
      for n in row
        grid.cell().eq(n).css {color: red}
```



About intelligence. TBD.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Extra page

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Reference

Movement

`fd 50`	forward 50 pixels
`bk 10`	backward 10 pixels
`rt 90`	turn right 90 degrees
`lt 120`	turn left 120 degrees
`home()`	go to the page center
`slide x, y`	slide right x and forward y
`moveto x, y`	go to x, y relative to home
`turnto 45`	set direction to 45 (NE)
`turnto obj`	point toward obj
`speed 30`	do 30 moves per second

Drawing

`pen blue`	draw in blue
`pen red, 9`	9 pixel wide red pen
`pen null`	use no color
`pen off`	pause use of the pen
`pen on`	use the pen again
`mark 'X'`	mark with an X
`dot green`	draw a green dot
`dot gold, 30`	30 pixel gold circle
`pen 'path'`	trace an invisible path
`fill cyan`	fill traced path in cyan

Appearance

`ht()`	hide the turtle
`st()`	show the turtle
`scale 8`	do everything 8x bigger
`wear yellow`	wear a yellow shell
`fadeOut()`	fade and hide the turtle
`remove()`	totally remove the turtle

Properties

`turtle`	name of the main turtle
`getxy()`	[x, y] position relative to home
`direction()`	direction of turtle
`hidden()`	if the turtle is hidden
`touches(obj)`	if the turtle touches obj
`inside(window)`	if enclosed in the window

Output

`p = write 'hi'`	adds HTML to the page
`p.html 'bye'`	changes old text
`button 'go',` `-> fd 10`	adds a button with an action
`read (n) ->` `write n*n`	adds a text input with an action
`t = table 3,5`	adds a 3x5 <table>
`t.cell(0, 0).` `text 'aloha'`	selects the first cell of the table and sets its text

Sets

`g = hatch 20`	hatch 20 new turtles
`g = $('img')`	select all <img> as a set
`g.plan (j) ->` `@fd j * 10`	direct the jth turtle to go forward by 10j pixels

Other Objects

`$(window)`	the visible window
`$('p').eq(0)`	the first <p> element
`$('#zed')`	the element with id="zed"

Other Functions

`see obj`	inspect the value of obj
`speed 8`	set default speed
`tick 5, -> fd 10`	go 5 times per second
`click -> fd 10`	go when clicked
`random [3,5,7]`	return 3, 5, or 7
`random 100`	random [0..99]
`play 'ceg'`	play musical notes

Colors

white	gainsboro	silver	darkgray	gray	dimgray	black
whitesmoke	lightgray	lightcoral	rosybrown	indianred	red	maroon
snow	mistyrose	salmon	orangered	chocolate	brown	darkred
seashell	peachpuff	tomato	darkorange	peru	firebrick	olive
linen	bisque	darksalmon	orange	goldenrod	sienna	darkolivegreen
oldlace	antiquewhite	coral	gold	limegreen	saddlebrown	darkgreen
floralwhite	navajowhite	lightsalmon	darkkhaki	lime	darkgoldenrod	green
cornsilk	blanchedalmond	sandybrown	yellow	mediumseagreen	olivedrab	forestgreen
ivory	papayawhip	burlywood	yellowgreen	springgreen	seagreen	darkslategray
beige	moccasin	tan	chartreuse	mediumspringgreen	lightseagreen	teal
lightyellow	wheat	khaki	lawngreen	aqua	darkturquoise	darkcyan
lightgoldenrodyellow	lemonchiffon	greenyellow	darkseagreen	cyan	deepskyblue	midnightblue
honeydew	palegoldenrod	lightgreen	mediumaquamarine	cadetblue	steelblue	navy
mintcream	palegreen	skyblue	turquoise	dodgerblue	blue	darkblue
azure	aquamarine	lightskyblue	mediumturquoise	lightslategray	blueviolet	mediumblue
lightcyan	paleturquoise	lightsteelblue	cornflowerblue	slategray	darkorchid	darkslateblue
aliceblue	powderblue	thistle	mediumslateblue	royalblue	fuchsia	indigo
ghostwhite	lightblue	plum	mediumpurple	slateblue	magenta	darkviolet
lavender	pink	violet	orchid	mediumorchid	mediumvioletred	purple
lavenderblush	lightpink	hotpink	palevioletred	deeppink	crimson	darkmagenta

2013.10.17