Amd

Getting Started with AMD

The **Asynchronous Module Definition** API specifies a mechanism for defining JavaScript modules such that the module and its dependencies can be asynchronously loaded. This is particularly well suited for the browser environment where synchronous loading of modules incurs performance, usability, debugging, and cross-domain access problems.

Basic concept

```
// The basic AMD API consists of nothing but two methods: `define` and `require`
// and is all about module definition and consumption:
// `define(id?, dependencies?, factory)` defines a module
// `require(dependencies, callback)` imports a set of dependencies and
// consumes them in the passed callback

// Let's start by using define to define a new named module
// that has no dependencies. We'll do so by passing a name
// and a factory function to define:
define('awesomeAMD', function(){
  var isAMDAwesome = function(){
    return true;
  };
  // The return value of a module's factory function is
  // what other modules or require calls will receive when
  // requiring our `awesomeAMD` module.
  // The exported value can be anything, (constructor) functions,
  // objects, primitives, even undefined (although that won't help too much).
  return isAMDAwesome;
});

// Now, let's define another module that depends upon our `awesomeAMD` module.
// Notice that there's an additional argument defining our
// module's dependencies now:
define('loudmouth', ['awesomeAMD'], function(awesomeAMD){
  // dependencies will be passed to the factory's arguments
  // in the order they are specified
  var tellEveryone = function(){
    if (awesomeAMD()){
      alert('This is sOoOo rad!');
    } else {
      alert('Pretty dull, isn\'t it?');
    }
  };
  return tellEveryone;
});

// As we do know how to use define now, let's use `require` to
// kick off our program. `require`'s signature is `(arrayOfDependencies, callback)`.
require(['loudmouth'], function(loudmouth){
  loudmouth();
});
```

```
// To make this tutorial run code, let's implement a very basic
// (non-asynchronous) version of AMD right here on the spot:
function define(name, deps, factory){
  // notice how modules without dependencies are handled
  define[name] = require(factory ? deps : [], factory || deps);
}

function require(deps, callback){
  var args = [];
  // first let's retrieve all the dependencies needed
  // by the require call
  for (var i = 0; i < deps.length; i++){
    args[i] = define[deps[i]];
  }
  // satisfy all the callback's dependencies
  return callback.apply(null, args);
}
// you can see this code in action here: http://jsfiddle.net/qap949pd/
```

Real-world usage with require.js

In contrast to the introductory example, `require.js` (the most popular AMD library) actually implements the **A** in **AMD**, enabling you to load modules and their dependencies asynchronously via XHR:

```
/* file: app/main.js */
require(['modules/someClass'], function(SomeClass){
  // the callback is deferred until the dependency is loaded
  var thing = new SomeClass();
});
console.log('So here we are, waiting!'); // this will run first
```

By convention, you usually store one module in one file. `require.js` can resolve module names based on file paths, so you don't have to name your modules, but can simply reference them using their location. In the example `someClass` is assumed to be in the `modules` folder, relative to your configuration's `baseUrl`:

- app/
- main.js
- modules/
  - someClass.js
  - someHelpers.js
  - …
- daos/
  - things.js
  - …

This means we can define `someClass` without specifying a module id:

```
/* file: app/modules/someClass.js */
define(['daos/things', 'modules/someHelpers'], function(thingsDao, helpers){
  // module definition, of course, will also happen asynchronously
  function SomeClass(){
    this.method = function(){/**/};
    // ...
  }
  return SomeClass;
});
```

To alter the default path mapping behavior use `requirejs.config(configObj)` in your `main.js`:

```
/* file: main.js */
requirejs.config({
  baseUrl : 'app',
  paths : {
    // you can also load modules from other locations
    jquery : '//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min',
    coolLibFromBower : '../bower_components/cool-lib/coollib'
  }
});
require(['jquery', 'coolLibFromBower', 'modules/someHelpers'], function($, coolLib, helpers){
  // a `main` file needs to call require at least once,
  // otherwise no code will ever run
  coolLib.doFancyStuffWith(helpers.transform($('#foo')));
});
```

`require.js`-based apps will usually have a single entry point (`main.js`) that is passed to the `require.js` script tag as a data-attribute. It will be automatically loaded and executed on pageload:

```
<!DOCTYPE html>
<html>
<head>
  <title>A hundred script tags? Never again!</title>
</head>
<body>
  <script src="require.js" data-main="app/main"></script>
</body>
</html>
```

Optimizing a whole project using r.js

Many people prefer using AMD for sane code organization during development, but still want to ship a single script file in production instead of performing hundreds of XHRs on page load.

`require.js` comes with a script called `r.js` (that you will probably run in node.js, although Rhino is supported too) that can analyse your project's dependency graph, and build a single file containing all your modules (properly named), minified and ready for consumption.

Install it using `npm`:

```
$ npm install requirejs -g
```

Now you can feed it with a configuration file:

```
$ r.js -o app.build.js
```

For our above example the configuration might look like:

```
/* file : app.build.js */
({
  name : 'main', // name of the entry point
  out : 'main-built.js', // name of the file to write the output to
  baseUrl : 'app',
  paths : {
    // `empty:` tells r.js that this should still be loaded from the CDN, using
    // the location specified in `main.js`
    jquery : 'empty:',
    coolLibFromBower : '../bower_components/cool-lib/coollib'
  }
})
```

To use the built file in production, simply swap `data-main`:

```
<script src="require.js" data-main="app/main-built"></script>
```

An incredibly detailed overview of build options is available in the GitHub repo.

Topics not covered in this tutorial

- Loader plugins / transforms
- CommonJS style loading and exporting
- Advanced configuration
- Shim configuration (loading non-AMD modules)
- CSS loading and optimizing with require.js
- Using almond.js for builds

Further reading:

- Official Spec
- Why AMD?
- Universal Module Definition

Implementations:

- require.js
- dojo toolkit
- cujo.js
- curl.js
- lsjs
- mmd

Asymptotic Notations

What are they?

Asymptotic Notations are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases. This is also known as an algorithm's growth rate. Does the algorithm suddenly become incredibly slow when the input size grows? Does it mostly maintain its quick run time as the input size increases? Asymptotic Notation gives us the ability to answer these questions.

Are there alternatives to answering these questions?

One way would be to count the number of primitive operations at different input sizes. Though this is a valid solution, the amount of work this takes for even simple algorithms does not justify its use.

Another way is to physically measure the amount of time an algorithm takes to complete given different input sizes. However, the accuracy and relativity (times obtained would only be relative to the machine they were computed on) of this method is bound to environmental variables such as computer hardware specifications, processing power, etc.

Types of Asymptotic Notation

In the first section of this doc we described how an Asymptotic Notation identifies the behavior of an algorithm as the input size changes. Let us imagine an algorithm as a function f, n as the input size, and f(n) being the running time. So for a given algorithm f, with input size n you get some resultant run time f(n). This results in a graph where the Y axis is the runtime, X axis is the input size, and plot points are the resultants of the amount of time for a given input size.

You can label a function, or algorithm, with an Asymptotic Notation in many different ways. Some examples are, you can describe an algorithm by its best case, worse case, or equivalent case. The most common is to analyze an algorithm by its worst case. You typically don't evaluate by best case because those conditions aren't what you're planning for. A very good example of this is sorting algorithms; specifically, adding elements to a tree structure. Best case for most algorithms could be as low as a single operation. However, in most cases, the element you're adding will need to be sorted appropriately through the tree, which could mean examining an entire branch. This is the worst case, and this is what we plan for.

Types of functions, limits, and simplification

```
Logarithmic Function - log n
Linear Function - an + b
Quadratic Function - an^2 + bn + c
Polynomial Function - an^z + . . . + an^2 + a*n^1 + a*n^0, where z is some constant
Exponential Function - a^n, where a is some constant
```

These are some basic function growth classifications used in various notations. The list starts at the slowest growing function (logarithmic, fastest execution time) and goes on to the fastest growing (exponential, slowest execution time). Notice that as 'n', or the input, increases in each of those functions, the result clearly increases much quicker in quadratic, polynomial, and exponential, compared to logarithmic and linear.

One extremely important note is that for the notations about to be discussed you should do your best to use simplest terms. This means to disregard constants, and lower order terms, because as the input size (or n in our f(n) example) increases to infinity (mathematical limits), the lower order terms and constants are of little to no importance. That being said, if you have constants that are 2^9001, or some other ridiculous, unimaginable amount, realize that simplifying will skew your notation accuracy.

Since we want simplest form, lets modify our table a bit…

```
Logarithmic - log n
Linear - n
Quadratic - n^2
Polynomial - n^z, where z is some constant
Exponential - a^n, where a is some constant
```

Big-O

Big-O, commonly written as O, is an Asymptotic Notation for the worst case, or ceiling of growth for a given function. Say `f(n)` is your algorithm runtime, and `g(n)` is an arbitrary time complexity you are trying to relate to your algorithm. `f(n)` is O(g(n)), if for any real constant c (c > 0), `f(n) <= c g(n)` for every input size n (n > 0).

Example 1

```
f(n) = 3log n + 100
g(n) = log n
```

Is `f(n)` O(g(n))? Is `3 log n + 100` O(log n)? Let's look to the definition of Big-O.

```
3log n + 100 <= c * log n
```

Is there some constant c that satisfies this for all n?

```
3log n + 100 <= 150 * log n, n > 2 (undefined at n = 1)
```

Yes! The definition of Big-O has been met therefore `f(n)` is O(g(n)).

Example 2

```
f(n) = 3*n^2
g(n) = n
```

Is `f(n)` O(g(n))? Is `3 * n^2` O(n)? Let's look at the definition of Big-O.

```
3 * n^2 <= c * n
```

Is there some constant c that satisfies this for all n? No, there isn't. `f(n)` is NOT O(g(n)).

Big-Omega

Big-Omega, commonly written as Ω, is an Asymptotic Notation for the best case, or a floor growth rate for a given function.

`f(n)` is Ω(g(n)), if for any real constant c (c > 0), `f(n)` is >= `c g(n)` for every input size n (n > 0).

Feel free to head over to additional resources for examples on this. Big-O is the primary notation used for general algorithm time complexity.

Ending Notes

It's hard to keep this kind of topic short, and you should definitely go through the books and online resources listed. They go into much greater depth with definitions and examples. More where x='Algorithms & Data Structures' is on its way; we'll have a doc up on analyzing actual code examples soon.

Books

- Algorithms
- Algorithm Design

Online Resources

- MIT
- KhanAcademy

Bash

Bash is a name of the unix shell, which was also distributed as the shell for the GNU operating system and as default shell on Linux and Mac OS X. Nearly all examples below can be a part of a shell script or executed directly in the shell.

Read more here.

```
#!/bin/bash
# First line of the script is shebang which tells the system how to execute
# the script: http://en.wikipedia.org/wiki/Shebang_(Unix)
# As you already figured, comments start with #. Shebang is also a comment.

# Simple hello world example:
echo Hello world!

# Each command starts on a new line, or after semicolon:
echo 'This is the first line'; echo 'This is the second line'

# Declaring a variable looks like this:
Variable="Some string"

# But not like this:
Variable = "Some string"
# Bash will decide that Variable is a command it must execute and give an error
# because it can't be found.

# Or like this:
Variable= 'Some string'
# Bash will decide that 'Some string' is a command it must execute and give an
# error because it can't be found. (In this case the 'Variable=' part is seen
# as a variable assignment valid only for the scope of the 'Some string'
# command.)

# Using the variable:
echo $Variable
echo "$Variable"
echo '$Variable'
# When you use the variable itself - assign it, export it, or else - you write
# its name without $. If you want to use the variable's value, you should use $.
# Note that ' (single quote) won't expand the variables!

# String substitution in variables
echo ${Variable/Some/A}
# This will substitute the first occurrence of "Some" with "A"

# Substring from a variable
Length=7
echo ${Variable:0:Length}
# This will return only the first 7 characters of the value

# Default value for variable
echo ${Foo:-"DefaultValueIfFooIsMissingOrEmpty"}
# This works for null (Foo=) and empty string (Foo=""); zero (Foo=0) returns 0.
# Note that it only returns default value and doesn't change variable value.

# Builtin variables:
# There are some useful builtin variables, like
echo "Last program's return value: $?"
echo "Script's PID: $$"
echo "Number of arguments passed to script: $#"
echo "All arguments passed to script: $@"
echo "Script's arguments separated into different variables: $1 $2..."

# Reading a value from input:
echo "What's your name?"
read Name # Note that we didn't need to declare a new variable
echo Hello, $Name!

# We have the usual if structure:
# use 'man test' for more info about conditionals
if [ $Name != $USER ]
then
    echo "Your name isn't your username"
else
    echo "Your name is your username"
fi

# NOTE: if $Name is empty, bash sees the above condition as:
if [ != $USER ]
# which is invalid syntax
# so the "safe" way to use potentially empty variables in bash is:
if [ "$Name" != $USER ] ...
# which, when $Name is empty, is seen by bash as:
if [ "" != $USER ] ...
# which works as expected

# There is also conditional execution
echo "Always executed" || echo "Only executed if first command fails"
echo "Always executed" && echo "Only executed if first command does NOT fail"

# To use && and || with if statements, you need multiple pairs of square brackets:
if [ "$Name" == "Steve" ] && [ "$Age" -eq 15 ]
then
    echo "This will run if $Name is Steve AND $Age is 15."
fi

if [ "$Name" == "Daniya" ] || [ "$Name" == "Zach" ]
then
    echo "This will run if $Name is Daniya OR Zach."
fi

# Expressions are denoted with the following format:
echo $(( 10 + 5 ))

# Unlike other programming languages, bash is a shell so it works in the context
# of a current directory. You can list files and directories in the current
# directory with the ls command:
ls

# These commands have options that control their execution:
ls -l # Lists every file and directory on a separate line

# Results of the previous command can be passed to the next command as input.
# grep command filters the input with provided patterns. That's how we can list
# .txt files in the current directory:
```

```
ls -l | grep "\.txt"

# You can redirect command input and output (stdin, stdout, and stderr).
# Read from stdin until ^EOF$ and overwrite hello.py with the lines
# between "EOF":
cat > hello.py << EOF
#!/usr/bin/env python
from __future__ import print_function
import sys
print("#stdout", file=sys.stdout)
print("#stderr", file=sys.stderr)
for line in sys.stdin:
    print(line, file=sys.stdout)
EOF

# Run hello.py with various stdin, stdout, and stderr redirections:
python hello.py < "input.in"
python hello.py > "output.out"
python hello.py 2> "error.err"
python hello.py > "output-and-error.log" 2>&1
python hello.py > /dev/null 2>&1
# The output error will overwrite the file if it exists,
# if you want to append instead, use ">>":
python hello.py >> "output.out" 2>> "error.err"

# Overwrite output.out, append to error.err, and count lines:
info bash 'Basic Shell Features' 'Redirections' > output.out 2>> error.err
wc -l output.out error.err

# Run a command and print its file descriptor (e.g. /dev/fd/123)
# see: man fd
echo <(echo "#helloworld")

# Overwrite output.out with "#helloworld":
cat > output.out <(echo "#helloworld")
echo "#helloworld" > output.out
echo "#helloworld" | cat > output.out
echo "#helloworld" | tee output.out >/dev/null

# Cleanup temporary files verbosely (add '-i' for interactive)
rm -v output.out error.err output-and-error.log

# Commands can be substituted within other commands using $( ):
# The following command displays the number of files and directories in the
# current directory.
echo "There are $(ls | wc -l) items here."

# The same can be done using backticks `` but they can't be nested - the preferred way
# is to use $( ).
echo "There are `ls | wc -l` items here."

# Bash uses a case statement that works similarly to switch in Java and C++:
case "$Variable" in
    #List patterns for the conditions you want to meet
    0) echo "There is a zero.";;
    1) echo "There is a one.";;
    *) echo "It is not null.";;
esac

# for loops iterate for as many arguments given:
# The contents of $Variable is printed three times.
for Variable in {1..3}
do
    echo "$Variable"
done

# Or write it the "traditional for loop" way:
for ((a=1; a <= 3; a++))
do
    echo $a
done

# They can also be used to act on files..
# This will run the command 'cat' on file1 and file2
for Variable in file1 file2
do
    cat "$Variable"
done

# ..or the output from a command
# This will cat the output from ls.
for Output in $(ls)
do
    cat "$Output"
done

# while loop:
while [ true ]
do
    echo "loop body here..."
    break
done

# You can also define functions
# Definition:
function foo ()
{
    echo "Arguments work just like script arguments: $@"
    echo "And: $1 $2..."
    echo "This is a function"
    return 0
}

# or simply
bar ()
{
    echo "Another way to declare functions!"
    return 0
```

```
}

# Calling your function
foo "My name is" $Name

# There are a lot of useful commands you should learn:
# prints last 10 lines of file.txt
tail -n 10 file.txt
# prints first 10 lines of file.txt
head -n 10 file.txt
# sort file.txt's lines
sort file.txt
# report or omit repeated lines, with -d it reports them
uniq -d file.txt
# prints only the first column before the ',' character
cut -d ',' -f 1 file.txt
# replaces every occurrence of 'okay' with 'great' in file.txt, (regex compatible)
sed -i 's/okay/great/g' file.txt
# print to stdout all lines of file.txt which match some regex
# The example prints lines which begin with "foo" and end in "bar"
grep "^foo.*bar$" file.txt
# pass the option "-c" to instead print the number of lines matching the regex
grep -c "^foo.*bar$" file.txt
# if you literally want to search for the string,
# and not the regex, use fgrep (or grep -F)
fgrep "^foo.*bar$" file.txt


# Read Bash shell builtins documentation with the bash 'help' builtin:
help
help help
help for
help return
help source
help .

# Read Bash manpage documentation with man
apropos bash
man 1 bash
man bash

# Read info documentation with info (? for help)
apropos info | grep '^info.*('
man info
info info
info 5 info

# Read bash info documentation:
info bash
info bash 'Bash Features'
info bash 6
info --apropos bash
```

Brainfuck

Brainfuck (not capitalized except at the start of a sentence) is an extremely minimal Turing-complete programming language with just 8 commands.

You can try brainfuck on your browser with brainfuck-visualizer.

```
Any character not "><+-.,[]" (excluding quotation marks) is ignored.

Brainfuck is represented by an array with 30,000 cells initialized to zero
and a data pointer pointing at the current cell.

There are eight commands:
+ : Increments the value at the current cell by one.
- : Decrements the value at the current cell by one.
> : Moves the data pointer to the next cell (cell on the right).
< : Moves the data pointer to the previous cell (cell on the left).
. : Prints the ASCII value at the current cell (i.e. 65 = 'A').
, : Reads a single input character into the current cell.
[ : If the value at the current cell is zero, skips to the corresponding ] .
    Otherwise, move to the next instruction.
] : If the value at the current cell is zero, move to the next instruction.
    Otherwise, move backwards in the instructions to the corresponding [ .

[ and ] form a while loop. Obviously, they must be balanced.

Let's look at some basic brainfuck programs.

++++++ [ > ++++++++++ < - ] > +++++ .

This program prints out the letter 'A'. First, it increments cell #1 to 6.
Cell #1 will be used for looping. Then, it enters the loop ([) and moves
to cell #2. It increments cell #2 10 times, moves back to cell #1, and
decrements cell #1. This loop happens 6 times (it takes 6 decrements for
cell #1 to reach 0, at which point it skips to the corresponding ] and
continues on).

At this point, we're on cell #1, which has a value of 0, while cell #2 has a
value of 60. We move on cell #2, increment 5 times, for a value of 65, and then
print cell #2's value. 65 is 'A' in ASCII, so 'A' is printed to the terminal.


, [ > + < - ] > .

This program reads a character from the user input and copies the character into
cell #1. Then we start a loop. Move to cell #2, increment the value at cell #2,
move back to cell #1, and decrement the value at cell #1. This continues on
until cell #1 is 0, and cell #2 holds cell #1's old value. Because we're on
cell #1 at the end of the loop, move to cell #2, and then print out the value
in ASCII.

Also keep in mind that the spaces are purely for readability purposes. You
could just as easily write it as:

,[>+<-]>.
```

```
Try and figure out what this program does:

,>,< [ > [ >+ >+ << -] >> [- << + >>] <<< -] >>

This program takes two numbers for input, and multiplies them.

The gist is it first reads in two inputs. Then it starts the outer loop,
conditioned on cell #1. Then it moves to cell #2, and starts the inner
loop conditioned on cell #2, incrementing cell #3. However, there comes a
problem: At the end of the inner loop, cell #2 is zero. In that case,
inner loop won't work anymore since next time. To solve this problem,
we also increment cell #4, and then recopy cell #4 into cell #2.
Then cell #3 is the result.
```

And that's brainfuck. Not that hard, eh? For fun, you can write your own brainfuck programs, or you can write a brainfuck interpreter in another language. The interpreter is fairly simple to implement, but if you're a masochist, try writing a brainfuck interpreter… in brainfuck.

C++

C++ is a systems programming language that, according to its inventor Bjarne Stroustrup, was designed to

- be a "better C"
- support data abstraction
- support object-oriented programming
- support generic programming

Though its syntax can be more difficult or complex than newer languages, it is widely used because it compiles to native instructions that can be directly run by the processor and offers tight control over hardware (like C) while offering high-level features such as generics, exceptions, and classes. This combination of speed and functionality makes C++ one of the most widely-used programming languages.

```
//////////////////
// Comparison to C
//////////////////

// C++ is _almost_ a superset of C and shares its basic syntax for
// variable declarations, primitive types, and functions.

// Just like in C, your program's entry point is a function called
// main with an integer return type.
// This value serves as the program's exit status.
// See http://en.wikipedia.org/wiki/Exit_status for more information.
int main(int argc, char** argv)
{
    // Command line arguments are passed in by argc and argv in the same way
    // they are in C.
    // argc indicates the number of arguments,
    // and argv is an array of C-style strings (char*)
    // representing the arguments.
    // The first argument is the name by which the program was called.
    // argc and argv can be omitted if you do not care about arguments,
    // giving the function signature of int main()
    // An exit status of 0 indicates success.
    return 0;
}

// However, C++ varies in some of the following ways:

// In C++, character literals are chars
sizeof('c') == sizeof(char) == 1

// In C, character literals are ints
sizeof('c') == sizeof(int)


// C++ has strict prototyping
void func(); // function which accepts no arguments

// In C
void func(); // function which may accept any number of arguments

// Use nullptr instead of NULL in C++
int* ip = nullptr;

// C standard headers are available in C++,
// but are prefixed with "c" and have no .h suffix.
#include <cstdio>

int main()
{
    printf("Hello, world!\n");
    return 0;
}

///////////////////////
// Function overloading
///////////////////////

// C++ supports function overloading
// provided each function takes different parameters.

void print(char const* myString)
{
    printf("String %s\n", myString);
}

void print(int myInt)
{
    printf("My int is %d", myInt);
}

int main()
{
    print("Hello"); // Resolves to void print(const char*)
    print(15); // Resolves to void print(int)
}
```

```
/////////////////////////////
// Default function arguments
/////////////////////////////

// You can provide default arguments for a function
// if they are not provided by the caller.

void doSomethingWithInts(int a = 1, int b = 4)
{
    // Do something with the ints here
}

int main()
{
    doSomethingWithInts();      // a = 1,  b = 4
    doSomethingWithInts(20);    // a = 20, b = 4
    doSomethingWithInts(20, 5); // a = 20, b = 5
}

// Default arguments must be at the end of the arguments list.

void invalidDeclaration(int a = 1, int b) // Error!
{
}


/////////////
// Namespaces
/////////////

// Namespaces provide separate scopes for variable, function,
// and other declarations.
// Namespaces can be nested.

namespace First {
    namespace Nested {
        void foo()
        {
            printf("This is First::Nested::foo\n");
        }
    } // end namespace Nested
} // end namespace First

namespace Second {
    void foo()
    {
        printf("This is Second::foo\n");
    }
}

void foo()
{
    printf("This is global foo\n");
}

int main()
{
    // Includes all symbols from namespace Second into the current scope. Note
    // that simply foo() no longer works, since it is now ambiguous whether
    // we're calling the foo in namespace Second or the top level.
    using namespace Second;

    Second::foo(); // prints "This is Second::foo"
    First::Nested::foo(); // prints "This is First::Nested::foo"
    ::foo(); // prints "This is global foo"
}

///////////////
// Input/Output
///////////////

// C++ input and output uses streams
// cin, cout, and cerr represent stdin, stdout, and stderr.
// << is the insertion operator and >> is the extraction operator.

#include <iostream> // Include for I/O streams

using namespace std; // Streams are in the std namespace (standard library)

int main()
{
   int myInt;

   // Prints to stdout (or terminal/screen)
   cout << "Enter your favorite number:\n";
   // Takes in input
   cin >> myInt;

   // cout can also be formatted
   cout << "Your favorite number is " << myInt << "\n";
   // prints "Your favorite number is <myInt>"

    cerr << "Used for error messages";
}

//////////
// Strings
//////////

// Strings in C++ are objects and have many member functions
#include <string>

using namespace std; // Strings are also in the namespace std (standard library)

string myString = "Hello";
string myOtherString = " World";
```

```
// + is used for concatenation.
cout << myString + myOtherString; // "Hello World"

cout << myString + " You"; // "Hello You"

// C++ strings are mutable and have value semantics.
myString.append(" Dog");
cout << myString; // "Hello Dog"


/////////////
// References
/////////////

// In addition to pointers like the ones in C,
// C++ has _references_.
// These are pointer types that cannot be reassigned once set
// and cannot be null.
// They also have the same syntax as the variable itself:
// No * is needed for dereferencing and
// & (address of) is not used for assignment.

using namespace std;

string foo = "I am foo";
string bar = "I am bar";


string& fooRef = foo; // This creates a reference to foo.
fooRef += ". Hi!"; // Modifies foo through the reference
cout << fooRef; // Prints "I am foo. Hi!"

// Doesn't reassign "fooRef". This is the same as "foo = bar", and
//   foo == "I am bar"
// after this line.
cout << &fooRef << endl; //Prints the address of foo
fooRef = bar;
cout << &fooRef << endl; //Still prints the address of foo
cout << fooRef;  // Prints "I am bar"

//The address of fooRef remains the same, i.e. it is still referring to foo.


const string& barRef = bar; // Create a const reference to bar.
// Like C, const values (and pointers and references) cannot be modified.
barRef += ". Hi!"; // Error, const references cannot be modified.

// Sidetrack: Before we talk more about references, we must introduce a concept
// called a temporary object. Suppose we have the following code:
string tempObjectFun() { ... }
string retVal = tempObjectFun();

// What happens in the second line is actually:
//   - a string object is returned from tempObjectFun
//   - a new string is constructed with the returned object as argument to the
//     constructor
//   - the returned object is destroyed
// The returned object is called a temporary object. Temporary objects are
// created whenever a function returns an object, and they are destroyed at the
// end of the evaluation of the enclosing expression (Well, this is what the
// standard says, but compilers are allowed to change this behavior. Look up
// "return value optimization" if you're into this kind of details). So in this
// code:
foo(bar(tempObjectFun()))

// assuming foo and bar exist, the object returned from tempObjectFun is
// passed to bar, and it is destroyed before foo is called.

// Now back to references. The exception to the "at the end of the enclosing
// expression" rule is if a temporary object is bound to a const reference, in
// which case its life gets extended to the current scope:

void constReferenceTempObjectFun() {
  // constRef gets the temporary object, and it is valid until the end of this
  // function.
  const string& constRef = tempObjectFun();
  ...
}

// Another kind of reference introduced in C++11 is specifically for temporary
// objects. You cannot have a variable of its type, but it takes precedence in
// overload resolution:

void someFun(string& s) { ... }  // Regular reference
void someFun(string&& s) { ... }  // Reference to temporary object

string foo;
someFun(foo);  // Calls the version with regular reference
someFun(tempObjectFun());  // Calls the version with temporary reference

// For example, you will see these two versions of constructors for
// std::basic_string:
basic_string(const basic_string& other);
basic_string(basic_string&& other);

// Idea being if we are constructing a new string from a temporary object (which
// is going to be destroyed soon anyway), we can have a more efficient
// constructor that "salvages" parts of that temporary string. You will see this
// concept referred to as "move semantics".

/////////////////////
// Enums
/////////////////////

// Enums are a way to assign a value to a constant most commonly used for
// easier visualization and reading of code
enum ECarTypes
{
  Sedan,
  Hatchback,
  SUV,
  Wagon
};

ECarTypes GetPreferredCarType()
{
    return ECarTypes::Hatchback;
}

// As of C++11 there is an easy way to assign a type to the enum which can be
// useful in serialization of data and converting enums back-and-forth between
// the desired type and their respective constants
enum ECarTypes : uint8_t
{
  Sedan, // 0
  Hatchback, // 1
  SUV = 254, // 254
  Hybrid // 255
};

void WriteByteToFile(uint8_t InputValue)
{
    // Serialize the InputValue to a file
}

void WritePreferredCarTypeToFile(ECarTypes InputCarType)
{
    // The enum is implicitly converted to a uint8_t due to its declared enum type
    WriteByteToFile(InputCarType);
}

// On the other hand you may not want enums to be accidentally cast to an integer
// type or to other enums so it is instead possible to create an enum class which
// won't be implicitly converted
enum class ECarTypes : uint8_t
{
  Sedan, // 0
  Hatchback, // 1
  SUV = 254, // 254
  Hybrid // 255
};

void WriteByteToFile(uint8_t InputValue)
{
    // Serialize the InputValue to a file
}

void WritePreferredCarTypeToFile(ECarTypes InputCarType)
{
    // Won't compile even though ECarTypes is a uint8_t due to the enum
    // being declared as an "enum class"!
    WriteByteToFile(InputCarType);
```

```
}

//////////////////////////////////////////
// Classes and object-oriented programming
//////////////////////////////////////////

// First example of classes
#include <iostream>

// Declare a class.
// Classes are usually declared in header (.h or .hpp) files.
class Dog {
    // Member variables and functions are private by default.
    std::string name;
    int weight;

// All members following this are public
// until "private:" or "protected:" is found.
public:

    // Default constructor
    Dog();

    // Member function declarations (implementations to follow)
    // Note that we use std::string here instead of placing
    // using namespace std;
    // above.
    // Never put a "using namespace" statement in a header.
    void setName(const std::string& dogsName);

    void setWeight(int dogsWeight);

    // Functions that do not modify the state of the object
    // should be marked as const.
    // This allows you to call them if given a const reference to the object.
    // Also note the functions must be explicitly declared as _virtual_
    // in order to be overridden in derived classes.
    // Functions are not virtual by default for performance reasons.
    virtual void print() const;

    // Functions can also be defined inside the class body.
    // Functions defined as such are automatically inlined.
    void bark() const { std::cout << name << " barks!\n"; }

    // Along with constructors, C++ provides destructors.
    // These are called when an object is deleted or falls out of scope.
    // This enables powerful paradigms such as RAII
    // (see below)
    // The destructor should be virtual if a class is to be derived from;
    // if it is not virtual, then the derived class' destructor will
    // not be called if the object is destroyed through a base-class reference
    // or pointer.
    virtual ~Dog();
```

```
}; // A semicolon must follow the class definition.

// Class member functions are usually implemented in .cpp files.
Dog::Dog()
{
    std::cout << "A dog has been constructed\n";
}

// Objects (such as strings) should be passed by reference
// if you are modifying them or const reference if you are not.
void Dog::setName(const std::string& dogsName)
{
    name = dogsName;
}

void Dog::setWeight(int dogsWeight)
{
    weight = dogsWeight;
}

// Notice that "virtual" is only needed in the declaration, not the definition.
void Dog::print() const
{
    std::cout << "Dog is " << name << " and weighs " << weight << "kg\n";
}

Dog::~Dog()
{
    cout << "Goodbye " << name << "\n";
}

int main() {
    Dog myDog; // prints "A dog has been constructed"
    myDog.setName("Barkley");
    myDog.setWeight(10);
    myDog.print(); // prints "Dog is Barkley and weighs 10 kg"
    return 0;
} // prints "Goodbye Barkley"

// Inheritance:

// This class inherits everything public and protected from the Dog class
// as well as private but may not directly access private members/methods
// without a public or protected method for doing so
class OwnedDog : public Dog {

    void setOwner(const std::string& dogsOwner);

    // Override the behavior of the print function for all OwnedDogs. See
    // http://en.wikipedia.org/wiki/Polymorphism_(computer_science)#Subtyping
    // for a more general introduction if you are unfamiliar with
    // subtype polymorphism.
    // The override keyword is optional but makes sure you are actually
    // overriding the method in a base class.
    void print() const override;

private:
    std::string owner;
};

// Meanwhile, in the corresponding .cpp file:

void OwnedDog::setOwner(const std::string& dogsOwner)
{
    owner = dogsOwner;
}

void OwnedDog::print() const
{
    Dog::print(); // Call the print function in the base Dog class
    std::cout << "Dog is owned by " << owner << "\n";
    // Prints "Dog is <name> and weights <weight>"
    //        "Dog is owned by <owner>"
}

//////////////////////////////////////////
// Initialization and Operator Overloading
//////////////////////////////////////////

// In C++ you can overload the behavior of operators such as +, -, *, /, etc.
// This is done by defining a function which is called
// whenever the operator is used.

#include <iostream>
using namespace std;

class Point {
public:
    // Member variables can be given default values in this manner.
    double x = 0;
    double y = 0;

    // Define a default constructor which does nothing
    // but initialize the Point to the default value (0, 0)
    Point() { };

    // The following syntax is known as an initialization list
    // and is the proper way to initialize class member values
    Point (double a, double b) :
        x(a),
        y(b)
    { /* Do nothing except initialize the values */ }

    // Overload the + operator.
    Point operator+(const Point& rhs) const;

    // Overload the += operator
    Point& operator+=(const Point& rhs);
    // It would also make sense to add the - and -= operators,
    // but we will skip those for brevity.
};

Point Point::operator+(const Point& rhs) const
{
    // Create a new point that is the sum of this one and rhs.
    return Point(x + rhs.x, y + rhs.y);
}

Point& Point::operator+=(const Point& rhs)
{
    x += rhs.x;
    y += rhs.y;
    return *this;
}

int main () {
    Point up (0,1);
    Point right (1,0);
    // This calls the Point + operator
    // Point up calls the + (function) with right as its parameter
    Point result = up + right;
    // Prints "Result is upright (1,1)"
    cout << "Result is upright (" << result.x << ',' << result.y << ")\n";
    return 0;
}

/////////////////////
// Templates
/////////////////////

// Templates in C++ are mostly used for generic programming, though they are
// much more powerful than generic constructs in other languages. They also
// support explicit and partial specialization and functional-style type
// classes; in fact, they are a Turing-complete functional language embedded
// in C++!

// We start with the kind of generic programming you might be familiar with. To
// define a class or function that takes a type parameter:
template<class T>
class Box {
public:
    // In this class, T can be used as any other type.
    void insert(const T&) { ... }
};

// During compilation, the compiler actually generates copies of each template
// with parameters substituted, so the full definition of the class must be
// present at each invocation. This is why you will see template classes defined
// entirely in header files.

// To instantiate a template class on the stack:
```

```
Box<int> intBox;

// and you can use it as you would expect:
intBox.insert(123);

// You can, of course, nest templates:
Box<Box<int> > boxOfBox;
boxOfBox.insert(intBox);

// Until C++11, you had to place a space between the two '>'s, otherwise '>>'
// would be parsed as the right shift operator.

// You will sometimes see
//   template<typename T>
// instead. The 'class' keyword and 'typename' keywords are _mostly_
// interchangeable in this case. For the full explanation, see
//   http://en.wikipedia.org/wiki/Typename
// (yes, that keyword has its own Wikipedia page).

// Similarly, a template function:
template<class T>
void barkThreeTimes(const T& input)
{
    input.bark();
    input.bark();
    input.bark();
}

// Notice that nothing is specified about the type parameters here. The compiler
// will generate and then type-check every invocation of the template, so the
// above function works with any type 'T' that has a const 'bark' method!

Dog fluffy;
fluffy.setName("Fluffy")
barkThreeTimes(fluffy); // Prints "Fluffy barks" three times.

// Template parameters don't have to be classes:
template<int Y>
void printMessage() {
  cout << "Learn C++ in " << Y << " minutes!" << endl;
}

// And you can explicitly specialize templates for more efficient code. Of
// course, most real-world uses of specialization are not as trivial as this.
// Note that you still need to declare the function (or class) as a template
// even if you explicitly specified all parameters.
template<>
void printMessage<10>() {
  cout << "Learn C++ faster in only 10 minutes!" << endl;
}

printMessage<20>();  // Prints "Learn C++ in 20 minutes!"
printMessage<10>();  // Prints "Learn C++ faster in only 10 minutes!"
```

```
/////////////////////
// Exception Handling
/////////////////////

// The standard library provides a few exception types
// (see http://en.cppreference.com/w/cpp/error/exception)
// but any type can be thrown an as exception
#include <exception>
#include <stdexcept>

// All exceptions thrown inside the _try_ block can be caught by subsequent
// _catch_ handlers.
try {
    // Do not allocate exceptions on the heap using _new_.
    throw std::runtime_error("A problem occurred");
}

// Catch exceptions by const reference if they are objects
catch (const std::exception& ex)
{
    std::cout << ex.what();
}

// Catches any exception not caught by previous _catch_ blocks
catch (...)
{
    std::cout << "Unknown exception caught";
    throw; // Re-throws the exception
}

///////
// RAII
///////

// RAII stands for "Resource Acquisition Is Initialization".
// It is often considered the most powerful paradigm in C++
// and is the simple concept that a constructor for an object
// acquires that object's resources and the destructor releases them.

// To understand how this is useful,
// consider a function that uses a C file handle:
void doSomethingWithAFile(const char* filename)
{
    // To begin with, assume nothing can fail.

    FILE* fh = fopen(filename, "r"); // Open the file in read mode.

    doSomethingWithTheFile(fh);
    doSomethingElseWithIt(fh);

    fclose(fh); // Close the file handle.
}
```

```
// Unfortunately, things are quickly complicated by error handling.
// Suppose fopen can fail, and that doSomethingWithTheFile and
// doSomethingElseWithIt return error codes if they fail.
//  (Exceptions are the preferred way of handling failure,
//   but some programmers, especially those with a C background,
//   disagree on the utility of exceptions).
// We now have to check each call for failure and close the file handle
// if a problem occurred.
bool doSomethingWithAFile(const char* filename)
{
    FILE* fh = fopen(filename, "r"); // Open the file in read mode
    if (fh == nullptr) // The returned pointer is null on failure.
        return false; // Report that failure to the caller.

    // Assume each function returns false if it failed
    if (!doSomethingWithTheFile(fh)) {
        fclose(fh); // Close the file handle so it doesn't leak.
        return false; // Propagate the error.
    }
    if (!doSomethingElseWithIt(fh)) {
        fclose(fh); // Close the file handle so it doesn't leak.
        return false; // Propagate the error.
    }

    fclose(fh); // Close the file handle so it doesn't leak.
    return true; // Indicate success
}

// C programmers often clean this up a little bit using goto:
bool doSomethingWithAFile(const char* filename)
{
    FILE* fh = fopen(filename, "r");
    if (fh == nullptr)
        return false;

    if (!doSomethingWithTheFile(fh))
        goto failure;

    if (!doSomethingElseWithIt(fh))
        goto failure;

    fclose(fh); // Close the file
    return true; // Indicate success

failure:
    fclose(fh);
    return false; // Propagate the error
}

// If the functions indicate errors using exceptions,
// things are a little cleaner, but still sub-optimal.
void doSomethingWithAFile(const char* filename)
{
    FILE* fh = fopen(filename, "r"); // Open the file in read mode
    if (fh == nullptr)
        throw std::runtime_error("Could not open the file.");

    try {
        doSomethingWithTheFile(fh);
        doSomethingElseWithIt(fh);
    }
    catch (...) {
        fclose(fh); // Be sure to close the file if an error occurs.
        throw; // Then re-throw the exception.
    }

    fclose(fh); // Close the file
    // Everything succeeded
}

// Compare this to the use of C++'s file stream class (fstream)
// fstream uses its destructor to close the file.
// Recall from above that destructors are automatically called
// whenever an object falls out of scope.
void doSomethingWithAFile(const std::string& filename)
{
    // ifstream is short for input file stream
    std::ifstream fh(filename); // Open the file

    // Do things with the file
    doSomethingWithTheFile(fh);
    doSomethingElseWithIt(fh);

} // The file is automatically closed here by the destructor

// This has _massive_ advantages:
// 1. No matter what happens,
//    the resource (in this case the file handle) will be cleaned up.
//    Once you write the destructor correctly,
//    It is _impossible_ to forget to close the handle and leak the resource.
// 2. Note that the code is much cleaner.
//    The destructor handles closing the file behind the scenes
//    without you having to worry about it.
// 3. The code is exception safe.
//    An exception can be thrown anywhere in the function and cleanup
//    will still occur.

// All idiomatic C++ code uses RAII extensively for all resources.
// Additional examples include
// - Memory using unique_ptr and shared_ptr
// - Containers - the standard library linked list,
//   vector (i.e. self-resizing array), hash maps, and so on
//   all automatically destroy their contents when they fall out of scope.
// - Mutexes using lock_guard and unique_lock

// containers with object keys of non-primitive values (custom classes) require
// compare function in the object itself or as a function pointer. Primitives
// have default comparators, but you can override it.
```

```
class Foo {
public:
    int j;
    Foo(int a) : j(a) {}
};
struct compareFunction {
    bool operator()(const Foo& a, const Foo& b) const {
        return a.j < b.j;
    }
};
//this isn't allowed (although it can vary depending on compiler)
//std::map<Foo, int> fooMap;
std::map<Foo, int, compareFunction> fooMap;
fooMap[Foo(1)]  = 1;
fooMap.find(Foo(1)); //true

///////////////////////////////////////
// Lambda Expressions (C++11 and above)
///////////////////////////////////////

// lambdas are a convenient way of defining an anonymous function
// object right at the location where it is invoked or passed as
// an argument to a function.

// For example, consider sorting a vector of pairs using the second
// value of the pair

vector<pair<int, int> > tester;
tester.push_back(make_pair(3, 6));
tester.push_back(make_pair(1, 9));
tester.push_back(make_pair(5, 0));

// Pass a lambda expression as third argument to the sort function
// sort is from the <algorithm> header

sort(tester.begin(), tester.end(), [](const pair<int, int>& lhs, const pair<int, int>& rhs) {
        return lhs.second < rhs.second;
    });

// Notice the syntax of the lambda expression,
// [] in the lambda is used to "capture" variables
// The "Capture List" defines what from the outside of the lambda should be available inside the functi
// It can be either:
//     1. a value : [x]
//     2. a reference : [&x]
//     3. any variable currently in scope by reference [&]
//     4. same as 3, but by value [=]
// Example:

vector<int> dog_ids;
// number_of_dogs = 3;
for(int i = 0; i < 3; i++) {
    dog_ids.push_back(i);
}
```

```
int weight[3] = {30, 50, 10};

// Say you want to sort dog_ids according to the dogs' weights
// So dog_ids should in the end become: [2, 0, 1]

// Here's where lambda expressions come in handy

sort(dog_ids.begin(), dog_ids.end(), [&weight](const int &lhs, const int &rhs) {
        return weight[lhs] < weight[rhs];
    });
// Note we captured "weight" by reference in the above example.
// More on Lambdas in C++ : http://stackoverflow.com/questions/7627098/what-is-a-lambda-expression-in-c

///////////////////////////////
// Range For (C++11 and above)
///////////////////////////////

// You can use a range for loop to iterate over a container
int arr[] = {1, 10, 3};

for(int elem: arr){
    cout << elem << endl;
}

// You can use "auto" and not worry about the type of the elements of the container
// For example:

for(auto elem: arr) {
    // Do something with each element of arr
}

/////////////////////
// Fun stuff
/////////////////////

// Aspects of C++ that may be surprising to newcomers (and even some veterans).
// This section is, unfortunately, wildly incomplete; C++ is one of the easiest
// languages with which to shoot yourself in the foot.

// You can override private methods!
class Foo {
  virtual void bar();
};
class FooSub : public Foo {
  virtual void bar();  // Overrides Foo::bar!
};


// 0 == false == NULL (most of the time)!
bool* pt = new bool;
*pt = 0; // Sets the value points by 'pt' to false.
pt = 0;  // Sets 'pt' to the null pointer. Both lines compile without warnings.
```

```
// nullptr is supposed to fix some of that issue:
int* pt2 = new int;
*pt2 = nullptr; // Doesn't compile
pt2 = nullptr;  // Sets pt2 to null.

// There is an exception made for bools.
// This is to allow you to test for null pointers with if(!ptr),
// but as a consequence you can assign nullptr to a bool directly!
*pt = nullptr;  // This still compiles, even though '*pt' is a bool!


// '=' != '=' != '='!
// Calls Foo::Foo(const Foo&) or some variant (see move semantics) copy
// constructor.
Foo f2;
Foo f1 = f2;

// Calls Foo::Foo(const Foo&) or variant, but only copies the 'Foo' part of
// 'fooSub'. Any extra members of 'fooSub' are discarded. This sometimes
// horrifying behavior is called "object slicing."
FooSub fooSub;
Foo f1 = fooSub;

// Calls Foo::operator=(Foo&) or variant.
Foo f1;
f1 = f2;


// How to truly clear a container:
class Foo { ... };
vector<Foo> v;
for (int i = 0; i < 10; ++i)
  v.push_back(Foo());

// Following line sets size of v to 0, but destructors don't get called
// and resources aren't released!
v.empty();
v.push_back(Foo());  // New value is copied into the first Foo we inserted

// Truly destroys all values in v. See section about temporary objects for
// explanation of why this works.
v.swap(vector<Foo>());
```

Further Reading:

An up-to-date language reference can be found at http://cppreference.com/w/cpp

Additional resources may be found at http://cplusplus.com

C

Ah, C. Still **the** language of modern high-performance computing.

C is the lowest-level language most programmers will ever use, but it more than makes up for it with raw speed. Just be aware of its manual memory management and C will take you as far as you need to go.

```
// Single-line comments start with // - only available in C99 and later.

/*
Multi-line comments look like this. They work in C89 as well.
*/

/*
Multi-line comments don't nest /* Be careful */  // comment ends on this line...
*/ // ...not this one!

// Constants: #define <keyword>
// Constants are written in all-caps out of convention, not requirement
#define DAYS_IN_YEAR 365

// Enumeration constants are also ways to declare constants.
// All statements must end with a semicolon
enum days {SUN = 1, MON, TUE, WED, THU, FRI, SAT};
// MON gets 2 automatically, TUE gets 3, etc.


// Import headers with #include
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// (File names between <angle brackets> are headers from the C standard library.)
// For your own headers, use double quotes instead of angle brackets:
//#include "my_header.h"

// Declare function signatures in advance in a .h file, or at the top of
// your .c file.
void function_1();
int function_2(void);

// Must declare a 'function prototype' before main() when functions occur after
// your main() function.
int add_two_ints(int x1, int x2); // function prototype
// although `int add_two_ints(int, int);` is also valid (no need to name the args),
// it is recommended to name arguments in the prototype as well for easier inspection

// Your program's entry point is a function called
// main with an integer return type.
int main(void) {
  // your program
}

// The command line arguments used to run your program are also passed to main
// argc being the number of arguments - your program's name counts as 1
// argv is an array of character arrays - containing the arguments themselves
// argv[0] = name of your program, argv[1] = first argument, etc.
int main (int argc, char** argv)
{
  // print output using printf, for "print formatted"
  // %d is an integer, \n is a newline
```

```
printf("%d\n", 0); // => Prints 0

///////////////////////////////////////
// Types
///////////////////////////////////////

// All variables MUST be declared at the top of the current block scope
// we declare them dynamically along the code for the sake of the tutorial

// ints are usually 4 bytes
int x_int = 0;

// shorts are usually 2 bytes
short x_short = 0;

// chars are guaranteed to be 1 byte
char x_char = 0;
char y_char = 'y'; // Char literals are quoted with ''

// longs are often 4 to 8 bytes; long longs are guaranteed to be at least
// 64 bits
long x_long = 0;
long long x_long_long = 0;

// floats are usually 32-bit floating point numbers
float x_float = 0.0f; // 'f' suffix here denotes floating point literal

// doubles are usually 64-bit floating-point numbers
double x_double = 0.0; // real numbers without any suffix are doubles

// integer types may be unsigned (greater than or equal to zero)
unsigned short ux_short;
unsigned int ux_int;
unsigned long long ux_long_long;

// chars inside single quotes are integers in machine's character set.
'0'; // => 48 in the ASCII character set.
'A'; // => 65 in the ASCII character set.

// sizeof(T) gives you the size of a variable with type T in bytes
// sizeof(obj) yields the size of the expression (variable, literal, etc.).
printf("%zu\n", sizeof(int)); // => 4 (on most machines with 4-byte words)


// If the argument of the `sizeof` operator is an expression, then its argument
// is not evaluated (except VLAs (see below)).
// The value it yields in this case is a compile-time constant.
int a = 1;
// size_t is an unsigned integer type of at least 2 bytes used to represent
// the size of an object.
size_t size = sizeof(a++); // a++ is not evaluated
printf("sizeof(a++) = %zu where a = %d\n", size, a);
// prints "sizeof(a++) = 4 where a = 1" (on a 32-bit architecture)
// Arrays must be initialized with a concrete size.
char my_char_array[20]; // This array occupies 1 * 20 = 20 bytes
int my_int_array[20]; // This array occupies 4 * 20 = 80 bytes
// (assuming 4-byte words)


// You can initialize an array to 0 thusly:
char my_array[20] = {0};

// Indexing an array is like other languages -- or,
// rather, other languages are like C
my_array[0]; // => 0

// Arrays are mutable; it's just memory!
my_array[1] = 2;
printf("%d\n", my_array[1]); // => 2

// In C99 (and as an optional feature in C11), variable-length arrays (VLAs)
// can be declared as well. The size of such an array need not be a compile
// time constant:
printf("Enter the array size: "); // ask the user for an array size
int size;
fscanf(stdin, "%d", &size);
int var_length_array[size]; // declare the VLA
printf("sizeof array = %zu\n", sizeof var_length_array);

// Example:
// > Enter the array size: 10
// > sizeof array = 40

// Strings are just arrays of chars terminated by a NULL (0x00) byte,
// represented in strings as the special character '\0'.
// (We don't have to include the NULL byte in string literals; the compiler
//  inserts it at the end of the array for us.)
char a_string[20] = "This is a string";
printf("%s\n", a_string); // %s formats a string

printf("%d\n", a_string[16]); // => 0
// i.e., byte #17 is 0 (as are 18, 19, and 20)

// If we have characters between single quotes, that's a character literal.
// It's of type `int`, and *not* `char` (for historical reasons).
int cha = 'a'; // fine
char chb = 'a'; // fine too (implicit conversion from int to char)

// Multi-dimensional arrays:
int multi_array[2][5] = {
  {1, 2, 3, 4, 5},
  {6, 7, 8, 9, 0}
};
// access elements:
int array_int = multi_array[0][2]; // => 3

///////////////////////////////////////
```

```
// Operators
////////////////////////////////////////

// Shorthands for multiple declarations:
int i1 = 1, i2 = 2;
float f1 = 1.0, f2 = 2.0;

int b, c;
b = c = 0;

// Arithmetic is straightforward
i1 + i2; // => 3
i2 - i1; // => 1
i2 * i1; // => 2
i1 / i2; // => 0 (0.5, but truncated towards 0)

// You need to cast at least one integer to float to get a floating-point result
(float)i1 / i2; // => 0.5f
i1 / (double)i2; // => 0.5 // Same with double
f1 / f2; // => 0.5, plus or minus epsilon
// Floating-point numbers and calculations are not exact

// Modulo is there as well
11 % 3; // => 2

// Comparison operators are probably familiar, but
// there is no Boolean type in c. We use ints instead.
// (Or _Bool or bool in C99.)
// 0 is false, anything else is true. (The comparison
// operators always yield 0 or 1.)
3 == 2; // => 0 (false)
3 != 2; // => 1 (true)
3 > 2; // => 1
3 < 2; // => 0
2 <= 2; // => 1
2 >= 2; // => 1

// C is not Python - comparisons don't chain.
// Warning: The line below will compile, but it means `(0 < a) < 2`.
// This expression is always true, because (0 < a) could be either 1 or 0.
// In this case it's 1, because (0 < 1).
int between_0_and_2 = 0 < a < 2;
// Instead use:
int between_0_and_2 = 0 < a && a < 2;

// Logic works on ints
!3; // => 0 (Logical not)
!0; // => 1
1 && 1; // => 1 (Logical and)
0 && 1; // => 0
0 || 1; // => 1 (Logical or)
0 || 0; // => 0

// Conditional ternary expression ( ? : )
```

```c
int e = 5;
int f = 10;
int z;
z = (e > f) ? e : f; // => 10 "if e > f return e, else return f."

// Increment and decrement operators:
char *s = "ILoveC";
int j = 0;
s[j++]; // => "i". Returns the j-th item of s THEN increments value of j.
j = 0;
s[++j]; // => "L". Increments value of j THEN returns j-th value of s.
// same with j-- and --j

// Bitwise operators!
~0x0F; // => 0xFFFFFFF0 (bitwise negation, "1's complement", example result for 32-bit int)
0x0F & 0xF0; // => 0x00 (bitwise AND)
0x0F | 0xF0; // => 0xFF (bitwise OR)
0x04 ^ 0x0F; // => 0x0B (bitwise XOR)
0x01 << 1; // => 0x02 (bitwise left shift (by 1))
0x02 >> 1; // => 0x01 (bitwise right shift (by 1))

// Be careful when shifting signed integers - the following are undefined:
// - shifting into the sign bit of a signed integer (int a = 1 << 31)
// - left-shifting a negative number (int a = -1 << 2)
// - shifting by an offset which is >= the width of the type of the LHS:
//   int a = 1 << 32; // UB if int is 32 bits wide

///////////////////////////////////////
// Control Structures
///////////////////////////////////////

if (0) {
  printf("I am never run\n");
} else if (0) {
  printf("I am also never run\n");
} else {
  printf("I print\n");
}

// While loops exist
int ii = 0;
while (ii < 10) { //ANY value not zero is true.
  printf("%d, ", ii++); // ii++ increments ii AFTER using its current value.
} // => prints "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "

printf("\n");

int kk = 0;
do {
  printf("%d, ", kk);
} while (++kk < 10); // ++kk increments kk BEFORE using its current value.
// => prints "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "

printf("\n");
```

```
// For loops too
int jj;
for (jj=0; jj < 10; jj++) {
  printf("%d, ", jj);
} // => prints "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "

printf("\n");

// *****NOTES*****:
// Loops and Functions MUST have a body. If no body is needed:
int i;
for (i = 0; i <= 5; i++) {
  ; // use semicolon to act as the body (null statement)
}
// Or
for (i = 0; i <= 5; i++);

// branching with multiple choices: switch()
switch (a) {
case 0: // labels need to be integral *constant* expressions
  printf("Hey, 'a' equals 0!\n");
  break; // if you don't break, control flow falls over labels
case 1:
  printf("Huh, 'a' equals 1!\n");
  break;
  // Be careful - without a "break", execution continues until the
  // next "break" is reached.
case 3:
case 4:
  printf("Look at that.. 'a' is either 3, or 4\n");
  break;
default:
  // if `some_integral_expression` didn't match any of the labels
  fputs("Error!\n", stderr);
  exit(-1);
  break;
}
/*
using "goto" in C
*/
typedef enum { false, true } bool;
// for C don't have bool as data type :(
bool disaster = false;
int i, j;
for(i=0;i<100;++i)
for(j=0;j<100;++j)
{
  if((i + j) >= 150)
      disaster = true;
  if(disaster)
      goto error;
}
error :
```

```
printf("Error occured at i = %d & j = %d.\n", i, j);
/*
https://ideone.com/GuPhd6
this will print out "Error occured at i = 52 & j = 99."
*/


///////////////////////////////////////
// Typecasting
///////////////////////////////////////

// Every value in C has a type, but you can cast one value into another type
// if you want (with some constraints).

int x_hex = 0x01; // You can assign vars with hex literals

// Casting between types will attempt to preserve their numeric values
printf("%d\n", x_hex); // => Prints 1
printf("%d\n", (short) x_hex); // => Prints 1
printf("%d\n", (char) x_hex); // => Prints 1

// Types will overflow without warning
printf("%d\n", (unsigned char) 257); // => 1 (Max char = 255 if char is 8 bits long)

// For determining the max value of a `char`, a `signed char` and an `unsigned char`,
// respectively, use the CHAR_MAX, SCHAR_MAX and UCHAR_MAX macros from <limits.h>

// Integral types can be cast to floating-point types, and vice-versa.
printf("%f\n", (float)100); // %f formats a float
printf("%lf\n", (double)100); // %lf formats a double
printf("%d\n", (char)100.0);

///////////////////////////////////////
// Pointers
///////////////////////////////////////

// A pointer is a variable declared to store a memory address. Its declaration will
// also tell you the type of data it points to. You can retrieve the memory address
// of your variables, then mess with them.

int x = 0;
printf("%p\n", (void *)&x); // Use & to retrieve the address of a variable
// (%p formats an object pointer of type void *)
// => Prints some address in memory;


// Pointers start with * in their declaration
int *px, not_a_pointer; // px is a pointer to an int
px = &x; // Stores the address of x in px
printf("%p\n", (void *)px); // => Prints some address in memory
printf("%zu, %zu\n", sizeof(px), sizeof(not_a_pointer));
// => Prints "8, 4" on a typical 64-bit system

// To retrieve the value at the address a pointer is pointing to,
// put * in front to dereference it.
// Note: yes, it may be confusing that '*' is used for _both_ declaring a
// pointer and dereferencing it.
printf("%d\n", *px); // => Prints 0, the value of x

// You can also change the value the pointer is pointing to.
// We'll have to wrap the dereference in parenthesis because
// ++ has a higher precedence than *.
(*px)++; // Increment the value px is pointing to by 1
printf("%d\n", *px); // => Prints 1
printf("%d\n", x); // => Prints 1

// Arrays are a good way to allocate a contiguous block of memory
int x_array[20]; //declares array of size 20 (cannot change size)
int xx;
for (xx = 0; xx < 20; xx++) {
  x_array[xx] = 20 - xx;
} // Initialize x_array to 20, 19, 18,... 2, 1

// Declare a pointer of type int and initialize it to point to x_array
int* x_ptr = x_array;
// x_ptr now points to the first element in the array (the integer 20).
// This works because arrays often decay into pointers to their first element.
// For example, when an array is passed to a function or is assigned to a pointer,
// it decays into (implicitly converted to) a pointer.
// Exceptions: when the array is the argument of the `&` (address-of) operator:
int arr[10];
int (*ptr_to_arr)[10] = &arr; // &arr is NOT of type `int *`!
// It's of type "pointer to array" (of ten `int`s).
// or when the array is a string literal used for initializing a char array:
char otherarr[] = "foobarbazquirk";
// or when it's the argument of the `sizeof` or `alignof` operator:
int arraythethird[10];
int *ptr = arraythethird; // equivalent with int *ptr = &arr[0];
printf("%zu, %zu\n", sizeof arraythethird, sizeof ptr);
// probably prints "40, 4" or "40, 8"


// Pointers are incremented and decremented based on their type
// (this is called pointer arithmetic)
printf("%d\n", *(x_ptr + 1)); // => Prints 19
printf("%d\n", x_array[1]); // => Prints 19

// You can also dynamically allocate contiguous blocks of memory with the
// standard library function malloc, which takes one argument of type size_t
// representing the number of bytes to allocate (usually from the heap, although this
// may not be true on e.g. embedded systems - the C standard says nothing about it).
int *my_ptr = malloc(sizeof(*my_ptr) * 20);
for (xx = 0; xx < 20; xx++) {
  *(my_ptr + xx) = 20 - xx; // my_ptr[xx] = 20-xx
} // Initialize memory to 20, 19, 18, 17... 2, 1 (as ints)

// Note that there is no standard way to get the length of a
// dynamically allocated array in C. Because of this, if your arrays are
```

```
  // going to be passed around your program a lot, you need another variable
  // to keep track of the number of elements (size) of an array. See the
  // functions section for more info.
  int size = 10;
  int *my_arr = malloc(sizeof(int) * size);
  // Add an element to the array
  my_arr = realloc(my_arr, ++size);
  my_arr[10] = 5;

  // Dereferencing memory that you haven't allocated gives
  // "unpredictable results" - the program is said to invoke "undefined behavior"
  printf("%d\n", *(my_ptr + 21)); // => Prints who-knows-what? It may even crash.

  // When you're done with a malloc'd block of memory, you need to free it,
  // or else no one else can use it until your program terminates
  // (this is called a "memory leak"):
  free(my_ptr);

  // Strings are arrays of char, but they are usually represented as a
  // pointer-to-char (which is a pointer to the first element of the array).
  // It's good practice to use `const char *' when referring to a string literal,
  // since string literals shall not be modified (i.e. "foo"[0] = 'a' is ILLEGAL.)
  const char *my_str = "This is my very own string literal";
  printf("%c\n", *my_str); // => 'T'

  // This is not the case if the string is an array
  // (potentially initialized with a string literal)
  // that resides in writable memory, as in:
  char foo[] = "foo";
  foo[0] = 'a'; // this is legal, foo now contains "aoo"

  function_1();
} // end main function

///////////////////////////////////////
// Functions
///////////////////////////////////////

// Function declaration syntax:
// <return type> <function name>(<args>)

int add_two_ints(int x1, int x2)
{
  return x1 + x2; // Use return to return a value
}

/*
Functions are call by value. When a function is called, the arguments passed to
the function are copies of the original arguments (except arrays). Anything you
do to the arguments in the function do not change the value of the original
argument where the function was called.

Use pointers if you need to edit the original argument values.
```

```
Example: in-place string reversal
*/

// A void function returns no value
void str_reverse(char *str_in)
{
  char tmp;
  int ii = 0;
  size_t len = strlen(str_in); // `strlen()` is part of the c standard library
  for (ii = 0; ii < len / 2; ii++) {
    tmp = str_in[ii];
    str_in[ii] = str_in[len - ii - 1]; // ii-th char from end
    str_in[len - ii - 1] = tmp;
  }
}

/*
char c[] = "This is a test.";
str_reverse(c);
printf("%s\n", c); // => ".tset a si sihT"
*/
/*
as we can return only one variable
to change values of more than one variables we use call by reference
*/
void swapTwoNumbers(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
/*
int first = 10;
int second = 20;
printf("first: %d\nsecond: %d\n", first, second);
swapTwoNumbers(&first, &second);
printf("first: %d\nsecond: %d\n", first, second);
// values will be swapped
*/

/*
With regards to arrays, they will always be passed to functions
as pointers. Even if you statically allocate an array like `arr[10]`,
it still gets passed as a pointer to the first element in any function calls.
Again, there is no standard way to get the size of a dynamically allocated
array in C.
*/
// Size must be passed!
// Otherwise, this function has no way of knowing how big the array is.
void printIntArray(int *arr, int size) {
    int i;
    for (i = 0; i < size; i++) {
        printf("arr[%d] is: %d\n", i, arr[i]);
    }
```

```
}
/*
int my_arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int size = 10;
printIntArray(my_arr, size);
// will print "arr[0] is: 1" etc
*/

// if referring to external variables outside function, must use extern keyword.
int i = 0;
void testFunc() {
  extern int i; //i here is now using external variable i
}

// make external variables private to source file with static:
static int j = 0; //other files using testFunc2() cannot access variable j
void testFunc2() {
  extern int j;
}
//**You may also declare functions as static to make them private**



///////////////////////////////////////
// User-defined types and structs
///////////////////////////////////////

// Typedefs can be used to create type aliases
typedef int my_type;
my_type my_type_var = 0;

// Structs are just collections of data, the members are allocated sequentially,
// in the order they are written:
struct rectangle {
  int width;
  int height;
};

// It's not generally true that
// sizeof(struct rectangle) == sizeof(int) + sizeof(int)
// due to potential padding between the structure members (this is for alignment
// reasons). [1]

void function_1()
{
  struct rectangle my_rec;

  // Access struct members with .
  my_rec.width = 10;
  my_rec.height = 20;

  // You can declare pointers to structs
  struct rectangle *my_rec_ptr = &my_rec;
  // Use dereferencing to set struct pointer members...
  (*my_rec_ptr).width = 30;

  // ... or even better: prefer the -> shorthand for the sake of readability
  my_rec_ptr->height = 10; // Same as (*my_rec_ptr).height = 10;
}

// You can apply a typedef to a struct for convenience
typedef struct rectangle rect;

int area(rect r)
{
  return r.width * r.height;
}

// if you have large structs, you can pass them "by pointer" to avoid copying
// the whole struct:
int areaptr(const rect *r)
{
  return r->width * r->height;
}

///////////////////////////////////////
// Function pointers
///////////////////////////////////////
/*
At run time, functions are located at known memory addresses. Function pointers are
much like any other pointer (they just store a memory address), but can be used
to invoke functions directly, and to pass handlers (or callback functions) around.
However, definition syntax may be initially confusing.

Example: use str_reverse from a pointer
*/
void str_reverse_through_pointer(char *str_in) {
  // Define a function pointer variable, named f.
  void (*f)(char *); // Signature should exactly match the target function.
  f = &str_reverse; // Assign the address for the actual function (determined at run time)
  // f = str_reverse; would work as well - functions decay into pointers, similar to arrays
  (*f)(str_in); // Just calling the function through the pointer
  // f(str_in); // That's an alternative but equally valid syntax for calling it.
}

/*
As long as function signatures match, you can assign any function to the same pointer.
Function pointers are usually typedef'd for simplicity and readability, as follows:
*/

typedef void (*my_fnp_type)(char *);

// Then used when declaring the actual pointer variable:
// ...
// my_fnp_type f;

//Special characters:
```

```
/*
'\a'; // alert (bell) character
'\n'; // newline character
'\t'; // tab character (left justifies text)
'\v'; // vertical tab
'\f'; // new page (form feed)
'\r'; // carriage return
'\b'; // backspace character
'\0'; // NULL character. Usually put at end of strings in C.
//   hello\n\0. \0 used by convention to mark end of string.
'\\'; // backslash
'\?'; // question mark
'\''; // single quote
'\"'; // double quote
'\xhh'; // hexadecimal number. Example: '\xb' = vertical tab character
'\0oo'; // octal number. Example: '\013' = vertical tab character

//print formatting:
"%d";    // integer
"%3d";   // integer with minimum of length 3 digits (right justifies text)
"%s";    // string
"%f";    // float
"%ld";   // long
"%3.2f"; // minimum 3 digits left and 2 digits right decimal float
"%7.4s"; // (can do with strings too)
"%c";    // char
"%p";    // pointer
"%x";    // hexadecimal
"%o";    // octal
"%%";    // prints %
*/
///////////////////////////////////////
// Order of Evaluation
///////////////////////////////////////

//---------------------------------------------------//
//        Operators                  | Associativity //
//---------------------------------------------------//
// () [] -> .                        | left to right //
// ! ~ ++ -- + = *(type)sizeof       | right to left //
// * / %                             | left to right //
// + -                               | left to right //
// << >>                             | left to right //
// < <= > >=                         | left to right //
// == !=                             | left to right //
// &                                 | left to right //
// ^                                 | left to right //
// |                                 | left to right //
// &&                                | left to right //
// ||                                | left to right //
// ?:                                | right to left //
// = += -= *= /= %= &= ^= |= <<= >>= | right to left //
// ,                                 | left to right //
//---------------------------------------------------//
```

```c
/******************************* Header Files **********************************

Header files are an important part of c as they allow for the connection of c
source files and can simplify code and definitions by seperating them into
seperate files.

Header files are syntactically similar to c source files but reside in ".h"
files. They can be included in your c source file by using the precompiler
command #include "example.h", given that example.h exists in the same directory
as the c file.
*/

/* A safe guard to prevent the header from being defined too many times. This */
/* happens in the case of circle dependency, the contents of the header is    */
/* already defined.                                                           */
#ifndef EXAMPLE_H /* if EXAMPLE_H is not yet defined. */
#define EXAMPLE_H /* Define the macro EXAMPLE_H. */

/* Other headers can be included in headers and therefore transitively */
/* included into files that include this header.                       */
#include <string.h>

/* Like c source files macros can be defined in headers and used in files */
/* that include this header file.                                         */
#define EXAMPLE_NAME "Dennis Ritchie"
/* Function macros can also be defined. */
#define ADD(a, b) (a + b)

/* Structs and typedefs can be used for consistency between files. */
typedef struct node
{
    int val;
    struct node *next;
} Node;

/* So can enumerations. */
enum traffic_light_state {GREEN, YELLOW, RED};

/* Function prototypes can also be defined here for use in multiple files,  */
/* but it is bad practice to define the function in the header. Definitions */
/* should instead be put in a c file.                                       */
Node createLinkedList(int *vals, int len);

/* Beyond the above elements, other definitions should be left to a c source */
/* file. Excessive includeds or definitions should, also not be contained in */
/* a header file but instead put into separate headers or a c file.          */

#endif /* End of the if precompiler directive. */
```

Further Reading

Best to find yourself a copy of K&R, aka "The C Programming Language" It is *the* book about C, written by Dennis Ritchie, the creator of C, and Brian Kernighan. Be careful, though - it's ancient and it contains some inaccuracies (well, ideas that are not considered good anymore) or now-changed practices.

Another good resource is Learn C The Hard Way.

If you have a question, read the compl.lang.c Frequently Asked Questions.

It's very important to use proper spacing, indentation and to be consistent with your coding style in general. Readable code is better than clever code and fast code. For a good, sane coding style to adopt, see the Linux kernel coding style.

Other than that, Google is your friend.

1 http://stackoverflow.com/questions/119123/why-isnt-sizeof-for-a-struct-equal-to-the-sum-of-sizeof-of-each-member

Chapel

You can read all about Chapel at Cray's official Chapel website. In short, Chapel is an open-source, high-productivity, parallel-programming language in development at Cray Inc., and is designed to run on multi-core PCs as well as multi-kilocore supercomputers.

More information and support can be found at the bottom of this document.

```
// Comments are C-family style
// one line comment
/*
  multi-line comment
*/

// Basic printing
write( "Hello, " );
writeln( "World!" );
// write and writeln can take a list of things to print.
// each thing is printed right next to each other, so include your spacing!
writeln( "There are ", 3, " commas (\",\") in this line of code" );
// Different output channels
stdout.writeln( "This goes to standard output, just like plain writeln() does");
stderr.writeln( "This goes to standard error" );

// Variables don't have to be explicitly typed as long as
// the compiler can figure out the type that it will hold.
var myVar = 10; // 10 is an int, so myVar is implicitly an int
myVar = -10;
var mySecondVar = myVar;
// var anError; // this would be a compile-time error.

// We can (and should) explicitly type things
var myThirdVar: real;
var myFourthVar: real = -1.234;
myThirdVar = myFourthVar;

// There are a number of basic types.
var myInt: int = -1000; // Signed ints
var myUint: uint = 1234; // Unsigned ints
var myReal: real = 9.876; // Floating point numbers
var myImag: imag = 5.0i; // Imaginary numbers
var myCplx: complex = 10 + 9i; // Complex numbers
myCplx = myInt + myImag ; // Another way to form complex numbers
var myBool: bool = false; // Booleans
var myStr: string = "Some string..."; // Strings

// Some types can have sizes
var my8Int: int(8) = 10; // 8 bit (one byte) sized int;
var my64Real: real(64) = 1.516; // 64 bit (8 bytes) sized real

// Typecasting
var intFromReal = myReal : int;
var intFromReal2: int = myReal : int;

// consts are constants, they cannot be changed after set in runtime.
const almostPi: real = 22.0/7.0;

// params are constants whose value must be known statically at compile-time
// Their value cannot be changed.
param compileTimeConst: int = 16;

// The config modifier allows values to be set at the command line
// and is much easier than the usual getOpts debacle
// config vars and consts can be changed through the command line at run time
config var varCmdLineArg: int = -123;
config const constCmdLineArg: int = 777;
// Set with --VarName=Value or --VarName Value at run time

// config params can be set/changed at compile-time
config param paramCmdLineArg: bool = false;
// Set config with --set paramCmdLineArg=value at compile-time
writeln( varCmdLineArg, ", ", constCmdLineArg, ", ", paramCmdLineArg );

// refs operate much like a reference in C++
var actual = 10;
ref refToActual = actual; // refToActual refers to actual
writeln( actual, " == ", refToActual ); // prints the same value
actual = -123; // modify actual (which refToActual refers to)
writeln( actual, " == ", refToActual ); // prints the same value
refToActual = 99999999; // modify what refToActual refers to (which is actual)
writeln( actual, " == ", refToActual ); // prints the same value

// Math operators
var a: int, thisInt = 1234, thatInt = 5678;
a = thisInt + thatInt;  // Addition
a = thisInt * thatInt;  // Multiplication
a = thisInt - thatInt;  // Subtraction
a = thisInt / thatInt;  // Division
a = thisInt ** thatInt; // Exponentiation
a = thisInt % thatInt;  // Remainder (modulo)

// Logical Operators
```

```
var b: bool, thisBool = false, thatBool = true;
b = thisBool && thatBool; // Logical and
b = thisBool || thatBool; // Logical or
b = !thisBool;            // Logical negation

// Relational Operators
b = thisInt > thatInt;           // Greater-than
b = thisInt >= thatInt;          // Greater-than-or-equal-to
b = thisInt < a && a <= thatInt; // Less-than, and, less-than-or-equal-to
b = thisInt != thatInt;          // Not-equal-to
b = thisInt == thatInt;          // Equal-to

// Bitwise operations
a = thisInt << 10;     // Left-bit-shift by 10 bits;
a = thatInt >> 5;      // Right-bit-shift by 5 bits;
a = ~thisInt;          // Bitwise-negation
a = thisInt ^ thatInt; // Bitwise exclusive-or

// Compound assignment operations
a += thisInt;          // Addition-equals ( a = a + thisInt;)
a *= thatInt;          // Times-equals ( a = a * thatInt; )
b &&= thatBool;        // Logical-and-equals ( b = b && thatBool; )
a <<= 3;               // Left-bit-shift-equals ( a = a << 10; )
// and many, many more.
// Unlike other C family languages there are no
// pre/post-increment/decrement operators like
//  ++j, --j, j++, j--

// Swap operator
var old_this = thisInt;
var old_that = thatInt;
thisInt <=> thatInt; // Swap the values of thisInt and thatInt
writeln( (old_this == thatInt) && (old_that == thisInt) );

// Operator overloads can also be defined, as we'll see with procedures

// Tuples can be of the same type
var sameTup: 2*int = (10,-1);
var sameTup2 = (11, -6);
// or different types
var diffTup: (int,real,complex) = (5, 1.928, myCplx);
var diffTupe2 = ( 7, 5.64, 6.0+1.5i );

// Accessed using array bracket notation
// However, tuples are all 1-indexed
writeln( "(", sameTup[1], ",", sameTup[2], ")" );
writeln( diffTup );

// Tuples can also be written into.
diffTup[1] = -1;

// Can expand tuple values into their own variables
var (tupInt, tupReal, tupCplx) = diffTup;
writeln( diffTup == (tupInt, tupReal, tupCplx) );
```

```
// Useful for writing a list of variables ( as is common in debugging)
writeln( (a,b,thisInt,thatInt,thisBool,thatBool) );

// Type aliasing
type chroma = int;        // Type of a single hue
type RGBColor = 3*chroma; // Type representing a full color
var black: RGBColor = ( 0,0,0 );
var white: RGBColor = ( 255, 255, 255 );

// If-then-else works just like any other C-family language
if 10 < 100 then
  writeln( "All is well" );

if -1 < 1 then
  writeln( "Continuing to believe reality" );
else
  writeln( "Send mathematician, something's wrong" );

if ( 10 > 100 ) {
  writeln( "Universe broken. Please reboot universe." );
}

if ( a % 2 == 0 ) {
  writeln( a, " is even." );
} else {
  writeln( a, " is odd." );
}

if ( a % 3 == 0 ) {
  writeln( a, " is even divisible by 3." );
} else if ( a % 3 == 1 ){
  writeln( a, " is divided by 3 with a remainder of 1." );
} else {
  writeln( b, " is divided by 3 with a remainder of 2." );
}

// Ternary: if-then-else in a statement
var maximum = if ( thisInt < thatInt ) then thatInt else thisInt;

// Select statements are much like switch statements in other languages
// However, Select statements don't cascade like in C or Java
var inputOption = "anOption";
select( inputOption ){
  when "anOption" do writeln( "Chose 'anOption'" );
  when "otherOption" {
    writeln( "Chose 'otherOption'" );
    writeln( "Which has a body" );
  }
  otherwise {
    writeln( "Any other Input" );
    writeln( "the otherwise case doesn't need a do if the body is one line" );
  }
}
```

```
// While and Do-While loops are basically the same in every language.
var j: int = 1;
var jSum: int = 0;
while( j <= 1000 ){
  jSum += j;
  j += 1;
}
writeln( jSum );

// Do-While loop
do{
  jSum += j;
  j += 1;
}while( j <= 10000 );
writeln( jSum );

// For loops are much like those in python in that they iterate over a range.
// Ranges themselves are types, and can be stuffed into variables
// (more about that later)
for i in 1..10 do write( i , ", ") ;
writeln( );

var iSum: int = 0;
for i in 1..1000 {
  iSum += i;
}
writeln( iSum );

for x in 1..10 {
  for y in 1..10 {
    write( (x,y), "\t" );
  }
  writeln( );
}

// Ranges and Domains
// For-loops and arrays both use ranges and domains to
// define an index set that can be iterated over.
// Ranges are single dimensional
// Domains can be multi-dimensional and can
// represent indices of different types as well.
// They are first-class citizen types, and can be assigned into variables
var range1to10: range = 1..10;  // 1, 2, 3, ..., 10
var range2to11 = 2..11; // 2, 3, 4, ..., 11
var rangeThistoThat: range = thisInt..thatInt; // using variables
var rangeEmpty: range = 100..-100 ; // this is valid but contains no indices

// Ranges can be unbounded
var range1toInf: range(boundedType=BoundedRangeType.boundedLow) = 1.. ;
// 1, 2, 3, 4, 5, ...
// Note: the range(boundedType= ... ) is only
// necessary if we explicitly type the variable
```

```
var rangeNegInfto1 = ..1; // ..., -4, -3, -2, -1, 0, 1

// Ranges can be strided using the 'by' operator.
var range2to10by2: range(stridable=true) = 2..10 by 2; // 2, 4, 6, 8, 10
// Note: the range(stridable=true) is only
// necessary if we explicitly type the variable

// Use by to create a reverse range
var reverse2to10by2 = 10..2 by -2; // 10, 8, 6, 4, 2

// The end point of a range can be determined using the count (#) operator
var rangeCount: range = -5..#12; // range from -5 to 6

// Can mix operators
var rangeCountBy: range(stridable=true) = -5..#12 by 2; // -5, -3, -1, 1, 3, 5
writeln( rangeCountBy );

// Can query properties of the range
// Print the first index, last index, number of indices,
// stride, and ask if 2 is include in the range
writeln( ( rangeCountBy.first, rangeCountBy.last, rangeCountBy.length,
           rangeCountBy.stride, rangeCountBy.member( 2 ) ) );

for i in rangeCountBy{
  write( i, if i == rangeCountBy.last then "\n" else ", " );
}

// Rectangular domains are defined using the same range syntax
// However they are required to be bounded (unlike ranges)
var domain1to10: domain(1) = {1..10};        // 1D domain from 1..10;
var twoDimensions: domain(2) = {-2..2,0..2}; // 2D domain over product of ranges
var thirdDim: range = 1..16;
var threeDims: domain(3) = {thirdDim, 1..10, 5..10}; // using a range variable

// Can iterate over the indices as tuples
for idx in twoDimensions do
  write( idx , ", ");
writeln( );

// or can deconstruct the tuple
for (x,y) in twoDimensions {
  write( "(", x, ", ", y, ")", ", " );
}
writeln( );

// Associative domains act like sets
var stringSet: domain(string); // empty set of strings
stringSet += "a";
stringSet += "b";
stringSet += "c";
stringSet += "a"; // Redundant add "a"
stringSet -= "c"; // Remove "c"
writeln( stringSet );
```

```
// Both ranges and domains can be sliced to produce a range or domain with the
// intersection of indices
var rangeA = 1.. ; // range from 1 to infinity
var rangeB =  ..5; // range from negative infinity to 5
var rangeC = rangeA[rangeB]; // resulting range is 1..5
writeln( (rangeA, rangeB, rangeC ) );

var domainA = {1..10, 5..20};
var domainB = {-5..5, 1..10};
var domainC = domainA[domainB];
writeln( (domainA, domainB, domainC) );

// Array are similar to those of other languages.
// Their sizes are defined using domains that represent their indices
var intArray: [1..10] int;
var intArray2: [{1..10}] int; //equivalent

// Accessed using bracket notation
for i in 1..10 do
  intArray[i] = -i;
writeln( intArray );
// We cannot access intArray[0] because it exists outside
// of the index set, {1..10}, we defined it to have.
// intArray[11] is illegal for the same reason.

var realDomain: domain(2) = {1..5,1..7};
var realArray: [realDomain] real;
var realArray2: [1..5,1..7] real;   // Equivalent
var realArray3: [{1..5,1..7}] real; // Equivalent

for i in 1..5 {
  for j in realDomain.dim(2) {   // Only use the 2nd dimension of the domain
    realArray[i,j] = -1.61803 * i + 0.5 * j;  // Access using index list
    var idx: 2*int = (i,j);                   // Note: 'index' is a keyword
    realArray[idx] = - realArray[(i,j)];      // Index using tuples
  }
}

// Arrays have domains as members that we can iterate over
for idx in realArray.domain {  // Again, idx is a 2*int tuple
  realArray[idx] = 1 / realArray[idx[1],idx[2]]; // Access by tuple and list
}

writeln( realArray );

// Can also iterate over the values of an array
var rSum: real = 0;
for value in realArray {
  rSum += value; // Read a value
  value = rSum;  // Write a value
}
writeln( rSum, "\n", realArray );

// Using associative domains we can create associative arrays (dictionaries)
```

```
var dictDomain: domain(string) = { "one", "two" };
var dict: [dictDomain] int = [ "one" => 1, "two" => 2 ];
dict["three"] = 3;
for key in dictDomain do writeln( dict[key] );

// Arrays can be assigned to each other in different ways
var thisArray : [{0..5}] int = [0,1,2,3,4,5];
var thatArray : [{0..5}] int;

// Simply assign one to the other.
// This copies thisArray into thatArray, instead of just creating a reference.
// Modifying thisArray does not also modify thatArray.
thatArray = thisArray;
thatArray[1] = -1;
writeln( (thisArray, thatArray) );

// Assign a slice one array to a slice (of the same size) of the other.
thatArray[{4..5}] = thisArray[{1..2}];
writeln( (thisArray, thatArray) );

// Operation can also be promoted to work on arrays.
var thisPlusThat = thisArray + thatArray;
writeln( thisPlusThat );

// Arrays and loops can also be expressions, where loop
// body's expression is the result of each iteration.
var arrayFromLoop = for i in 1..10 do i;
writeln( arrayFromLoop );

// An expression can result in nothing,
// such as when filtering with an if-expression
var evensOrFives = for i in 1..10 do if (i % 2 == 0 || i % 5 == 0) then i;

writeln( arrayFromLoop );

// Or could be written with a bracket notation
// Note: this syntax uses the 'forall' parallel concept discussed later.
var evensOrFivesAgain = [ i in 1..10 ] if (i % 2 == 0 || i % 5 == 0) then i;

// Or over the values of the array
arrayFromLoop = [ value in arrayFromLoop ] value + 1;

// Note: this notation can get somewhat tricky. For example:
// evensOrFives = [ i in 1..10 ] if (i % 2 == 0 || i % 5 == 0) then i;
// would break.
// The reasons for this are explained in depth when discussing zipped iterators.

// Chapel procedures have similar syntax to other languages functions.
proc fibonacci( n : int ) : int {
  if ( n <= 1 ) then return n;
  return fibonacci( n-1 ) + fibonacci( n-2 );
}

// Input parameters can be untyped (a generic procedure)
```

```
proc doublePrint( thing ): void {
  write( thing, " ", thing, "\n");
}

// Return type can be inferred (as long as the compiler can figure it out)
proc addThree( n ) {
  return n + 3;
}

doublePrint( addThree( fibonacci( 20 ) ) );

// Can also take 'unlimited' number of parameters
proc maxOf( x ...?k ) {
  // x refers to a tuple of one type, with k elements
  var maximum = x[1];
  for i in 2..k do maximum = if (maximum < x[i]) then x[i] else maximum;
  return maximum;
}
writeln( maxOf( 1, -10, 189, -9071982, 5, 17, 20001, 42 ) );

// The ? operator is called the query operator, and is used to take
// undetermined values (like tuple or array sizes, and generic types).

// Taking arrays as parameters.
// The query operator is used to determine the domain of A.
// This is important to define the return type (if you wanted to)
proc invertArray( A: [?D] int ): [D] int{
  for a in A do a = -a;
  return A;
}

writeln( invertArray( intArray ) );

// Procedures can have default parameter values, and
// the parameters can be named in the call, even out of order
proc defaultsProc( x: int, y: real = 1.2634 ): (int,real){
  return (x,y);
}

writeln( defaultsProc( 10 ) );
writeln( defaultsProc( x=11 ) );
writeln( defaultsProc( x=12, y=5.432 ) );
writeln( defaultsProc( y=9.876, x=13 ) );

// Intent modifiers on the arguments convey how
// those arguments are passed to the procedure
// in: copy arg in, but not out
// out: copy arg out, but not in
// inout: copy arg in, copy arg out
// ref: pass arg by reference
proc intentsProc( in inarg, out outarg, inout inoutarg, ref refarg ){
  writeln( "Inside Before: ", (inarg, outarg, inoutarg, refarg) );
  inarg = inarg + 100;
  outarg = outarg + 100;
  inoutarg = inoutarg + 100;
  refarg = refarg + 100;
  writeln( "Inside After: ", (inarg, outarg, inoutarg, refarg) );
}

var inVar: int = 1;
var outVar: int = 2;
var inoutVar: int = 3;
var refVar: int = 4;
writeln( "Outside Before: ", (inVar, outVar, inoutVar, refVar) );
intentsProc( inVar, outVar, inoutVar, refVar );
writeln( "Outside After: ", (inVar, outVar, inoutVar, refVar) );

// Similarly we can define intents on the return type
// refElement returns a reference to an element of array
proc refElement( array : [?D] ?T, idx ) ref : T {
  return array[ idx ]; // returns a reference to
}

var myChangingArray : [1..5] int = [1,2,3,4,5];
writeln( myChangingArray );
// Store reference to element in ref variable
ref refToElem = refElement( myChangingArray, 5 );
writeln( refToElem );
refToElem = -2; // modify reference which modifies actual value in array
writeln( refToElem );
writeln( myChangingArray );
// This makes more practical sense for class methods where references to
// elements in a data-structure are returned via a method or iterator

// We can query the type of arguments to generic procedures
// Here we define a procedure that takes two arguments of
// the same type, yet we don't define what that type is.
proc genericProc( arg1 : ?valueType, arg2 : valueType ): void {
  select( valueType ){
    when int do writeln( arg1, " and ", arg2, " are ints" );
    when real do writeln( arg1, " and ", arg2, " are reals" );
    otherwise writeln( arg1, " and ", arg2, " are somethings!" );
  }
}

genericProc( 1, 2 );
genericProc( 1.2, 2.3 );
genericProc( 1.0+2.0i, 3.0+4.0i );

// We can also enforce a form of polymorphism with the 'where' clause
// This allows the compiler to decide which function to use.
// Note: that means that all information needs to be known at compile-time.
// The param modifier on the arg is used to enforce this constraint.
proc whereProc( param N : int ): void
 where ( N > 0 ) {
  writeln( "N is greater than 0" );
}
```

```
proc whereProc( param N : int ): void
 where ( N < 0 ) {
  writeln( "N is less than 0" );
}

whereProc( 10 );
whereProc( -1 );
// whereProc( 0 ) would result in a compiler error because there
// are no functions that satisfy the where clause's condition.
// We could have defined a whereProc without a where clause that would then have
// served as a catch all for all the other cases (of which there is only one).

// Operator definitions are through procedures as well.
// We can define the unary operators:
// + - ! ~
// and the binary operators:
// + - * / % ** == <= >= < > << >> & | ^ by
// += -= *= /= %= **= &= |= ^= <<= >>= <=>

// Boolean exclusive or operator
proc ^( left : bool, right : bool ): bool {
  return (left || right) && !( left && right );
}

writeln( true  ^ true  );
writeln( false ^ true  );
writeln( true  ^ false );
writeln( false ^ false );

// Define a * operator on any two types that returns a tuple of those types
proc *( left : ?ltype, right : ?rtype): ( ltype, rtype ){
  return (left, right );
}

writeln( 1 * "a" ); // Uses our * operator
writeln( 1 * 2 );   // Uses the default * operator

/*
Note: You could break everything if you get careless with your overloads.
This here will break everything. Don't do it.
proc +( left: int, right: int ): int{
  return left - right;
}
*/

// Iterators are a sisters to the procedure, and almost
// everything about procedures also applies to iterators
// However, instead of returning a single value,
// iterators yield many values to a loop.
// This is useful when a complicated set or order of iterations is needed but
// allows the code defining the iterations to be separate from the loop body.
iter oddsThenEvens( N: int ): int {
  for i in 1..N by 2 do
    yield i; // yield values instead of returning.
  for i in 2..N by 2 do
    yield i;
}

for i in oddsThenEvens( 10 ) do write( i, ", " );
writeln( );

// Iterators can also yield conditionally, the result of which can be nothing
iter absolutelyNothing( N ): int {
  for i in 1..N {
    if ( N < i ) { // Always false
      yield i;     // Yield statement never happens
    }
  }
}

for i in absolutelyNothing( 10 ){
  writeln( "Woa there! absolutelyNothing yielded ", i );
}

// We can zipper together two or more iterators (who have the same number
// of iterations)  using zip() to create a single zipped iterator, where each
// iteration of the zipped iterator yields a tuple of one value yielded
// from each iterator.
                                 // Ranges have implicit iterators
for (positive, negative) in zip( 1..5, -5..-1) do
  writeln( (positive, negative) );

// Zipper iteration is quite important in the assignment of arrays,
// slices of arrays, and array/loop expressions.
var fromThatArray : [1..#5] int = [1,2,3,4,5];
var toThisArray : [100..#5] int;

// The operation
toThisArray = fromThatArray;
// is produced through
for (i,j) in zip( toThisArray.domain, fromThatArray.domain) {
  toThisArray[ i ] = fromThatArray[ j ];
}

toThisArray = [ j in -100..#5 ] j;
writeln( toThisArray );
// is produced through
for (i, j) in zip( toThisArray.domain, -100..#5 ){
  toThisArray[i] = j;
}
writeln( toThisArray );

// This is all very important in understanding why the statement
// var iterArray : [1..10] int = [ i in 1..10 ] if ( i % 2 == 1 ) then j;
// exhibits a runtime error.
// Even though the domain of the array and the loop-expression are
// the same size, the body of the expression can be thought of as an iterator.
// Because iterators can yield nothing, that iterator yields a different number
// of things than the domain of the array or loop, which is not allowed.

// Classes are similar to those in C++ and Java.
// They currently lack privatization
class MyClass {
  // Member variables
  var memberInt : int;
  var memberBool : bool = true;

  // Classes have default constructors that don't need to be coded (see below)
  // Our explicitly defined constructor
  proc MyClass( val : real ){
    this.memberInt = ceil( val ): int;
  }

  // Our explicitly defined destructor
  proc ~MyClass( ){
    writeln( "MyClass Destructor called ", (this.memberInt, this.memberBool) );
  }

  // Class methods
  proc setMemberInt( val: int ){
    this.memberInt = val;
  }

  proc setMemberBool( val: bool ){
    this.memberBool = val;
  }

  proc getMemberInt( ): int{
    return this.memberInt;
  }

  proc getMemberBool( ): bool {
    return this.memberBool;
  }

}

// Construct using default constructor, using default values
var myObject = new MyClass( 10 );
    myObject = new MyClass( memberInt = 10 ); // Equivalent
writeln( myObject.getMemberInt( ) );
// ... using our values
var myDiffObject = new MyClass( -1, true );
    myDiffObject = new MyClass( memberInt = -1,
                                memberBool = true ); // Equivalent
writeln( myDiffObject );

// Construct using written constructor
var myOtherObject = new MyClass( 1.95 );
    myOtherObject = new MyClass( val = 1.95 ); // Equivalent
writeln( myOtherObject.getMemberInt( ) );
```

```
// We can define an operator on our class as well but
// the definition has to be outside the class definition
proc +( A : MyClass, B : MyClass) : MyClass {
  return new MyClass( memberInt = A.getMemberInt( ) + B.getMemberInt( ),
                      memberBool = A.getMemberBool( ) || B.getMemberBool( ) );
}

var plusObject = myObject + myDiffObject;
writeln( plusObject );

// Destruction
delete myObject;
delete myDiffObject;
delete myOtherObject;
delete plusObject;

// Classes can inherit from one or more parent classes
class MyChildClass : MyClass {
  var memberComplex: complex;
}

// Generic Classes
class GenericClass {
  type classType;
  var classDomain: domain(1);
  var classArray: [classDomain] classType;

  // Explicit constructor
  proc GenericClass( type classType, elements : int ){
    this.classDomain = {1..#elements};
  }

  // Copy constructor
  // Note: We still have to put the type as an argument, but we can
  // default to the type of the other object using the query (?) operator
  // Further, we can take advantage of this to allow our copy constructor
  // to copy classes of different types and cast on the fly
  proc GenericClass( other : GenericClass(?otherType),
                     type classType = otherType ) {
    this.classDomain = other.classDomain;
    // Copy and cast
    for idx in this.classDomain do this[ idx ] = other[ idx ] : classType;
  }

  // Define bracket notation on a GenericClass
  // object so it can behave like a normal array
  // i.e. objVar[ i ] or objVar( i )
  proc this( i : int ) ref : classType {
    return this.classArray[ i ];
  }

  // Define an implicit iterator for the class
  // to yield values from the array to a loop
  // i.e. for i in objVar do ....
  iter these( ) ref : classType {
    for i in this.classDomain do
      yield this[i];
  }

}

var realList = new GenericClass( real, 10 );
// We can assign to the member array of the object using the bracket
// notation that we defined ( proc this( i: int ){ ... }  )
for i in realList.classDomain do realList[i] = i + 1.0;
// We can iterate over the values in our list with the iterator
// we defined ( iter these( ){ ... } )
for value in realList do write( value, ", " );
writeln( );

// Make a copy of realList using the copy constructor
var copyList = new GenericClass( realList );
for value in copyList do write( value, ", " );
writeln( );

// Make a copy of realList and change the type, also using the copy constructor
var copyNewTypeList = new GenericClass( realList, int );
for value in copyNewTypeList do write( value, ", " );
writeln( );

// Modules are Chapel's way of managing name spaces.
// The files containing these modules do not need to be named after the modules
// (as in Java), but files implicitly name modules.
// In this case, this file implicitly names the 'learnchapel' module

module OurModule {
  // We can use modules inside of other modules.
  use Time; // Time is one of the standard modules.

  // We'll use this procedure in the parallelism section.
  proc countdown( seconds: int ){
    for i in 1..seconds by -1 {
      writeln( i );
      sleep( 1 );
    }
  }

  // Submodules of OurModule
  // It is possible to create arbitrarily deep module nests.
  module ChildModule {
    proc foo(){
      writeln( "ChildModule.foo()");
    }
  }

  module SiblingModule {
    proc foo(){
      writeln( "SiblingModule.foo()" );
    }
  }
} // end OurModule

// Using OurModule also uses all the modules it uses.
// Since OurModule uses Time, we also use time.
use OurModule;

// At this point we have not used ChildModule or SiblingModule so their symbols
// (i.e. foo ) are not available to us.
// However, the module names are, and we can explicitly call foo() through them.
SiblingModule.foo();         // Calls SiblingModule.foo()

// Super explicit naming.
OurModule.ChildModule.foo(); // Calls ChildModule.foo()

use ChildModule;
foo();   // Less explicit call on ChildModule.foo()

// We can declare a main procedure
// Note: all the code above main still gets executed.
proc main(){

  // Parallelism
  // In other languages, parallelism is typically done with
  // complicated libraries and strange class structure hierarchies.
  // Chapel has it baked right into the language.

  // A begin statement will spin the body of that statement off
  // into one new task.
  // A sync statement will ensure that the progress of the main
  // task will not progress until the children have synced back up.
  sync {
    begin { // Start of new task's body
      var a = 0;
      for i in 1..1000 do a += 1;
      writeln( "Done: ", a);
    } // End of new tasks body
    writeln( "spun off a task!");
  }
  writeln( "Back together" );

  proc printFibb( n: int ){
    writeln( "fibonacci(",n,") = ", fibonacci( n ) );
  }

  // A cobegin statement will spin each statement of the body into one new task
  cobegin {
    printFibb( 20 ); // new task
    printFibb( 10 ); // new task
    printFibb( 5 );  // new task
    {
      // This is a nested statement body and thus is a single statement
      // to the parent statement and is executed by a single task
      writeln( "this gets" );
      writeln( "executed as" );
      writeln( "a whole" );
    }
  }
  // Notice here that the prints from each statement may happen in any order.

  // Coforall loop will create a new task for EACH iteration
  var num_tasks = 10; // Number of tasks we want
  coforall taskID in 1..#num_tasks {
    writeln( "Hello from task# ", taskID );
  }
  // Again we see that prints happen in any order.
  // NOTE! coforall should be used only for creating tasks!
  // Using it to iterating over a structure is very a bad idea!

  // forall loops are another parallel loop, but only create a smaller number
  // of tasks, specifically --dataParTasksPerLocale=number of task
  forall i in 1..100 {
    write( i, ", ");
  }
  writeln( );
  // Here we see that there are sections that are in order, followed by
  // a section that would not follow ( e.g. 1, 2, 3, 7, 8, 9, 4, 5, 6, ).
  // This is because each task is taking on a chunk of the range 1..10
  // (1..3, 4..6, or 7..9) doing that chunk serially, but each task happens
  // in parallel.
  // Your results may depend on your machine and configuration

  // For both the forall and coforall loops, the execution of the
  // parent task will not continue until all the children sync up.

  // forall loops are particularly useful for parallel iteration over arrays.
  // Lets run an experiment to see how much faster a parallel loop is
  use Time; // Import the Time module to use Timer objects
  var timer: Timer;
  var myBigArray: [{1..4000,1..4000}] real; // Large array we will write into

  // Serial Experiment
  timer.start( ); // Start timer
  for (x,y) in myBigArray.domain { // Serial iteration
    myBigArray[x,y] = (x:real) / (y:real);
  }
  timer.stop( ); // Stop timer
  writeln( "Serial: ", timer.elapsed( ) ); // Print elapsed time
  timer.clear( ); // Clear timer for parallel loop

  // Parallel Experiment
  timer.start( ); // start timer
  forall (x,y) in myBigArray.domain { // Parallel iteration
    myBigArray[x,y] = (x:real) / (y:real);
  }
  timer.stop( ); // Stop timer
  writeln( "Parallel: ", timer.elapsed( ) ); // Print elapsed time
```

```
timer.clear( );
// You may have noticed that (depending on how many cores you have)
// that the parallel loop went faster than the serial loop

// The bracket style loop-expression described
// much earlier implicitly uses a forall loop.
[ val in myBigArray ] val = 1 / val; // Parallel operation

// Atomic variables, common to many languages, are ones whose operations
// occur uninterrupted. Multiple threads can both modify atomic variables
// and can know that their values are safe.
// Chapel atomic variables can be of type bool, int, uint, and real.
var uranium: atomic int;
uranium.write( 238 );      // atomically write a variable
writeln( uranium.read() ); // atomically read a variable

// operations are described as functions, you could define your own operators.
uranium.sub( 3 ); // atomically subtract a variable
writeln( uranium.read() );

var replaceWith = 239;
var was = uranium.exchange( replaceWith );
writeln( "uranium was ", was, " but is now ", replaceWith );

var isEqualTo = 235;
if ( uranium.compareExchange( isEqualTo, replaceWith ) ) {
  writeln( "uranium was equal to ", isEqualTo,
           " so replaced value with ", replaceWith );
} else {
  writeln( "uranium was not equal to ", isEqualTo,
           " so value stays the same...  whatever it was" );
}

sync {
  begin { // Reader task
    writeln( "Reader: waiting for uranium to be ", isEqualTo );
    uranium.waitFor( isEqualTo );
    writeln( "Reader: uranium was set (by someone) to ", isEqualTo );
  }

  begin { // Writer task
    writeln( "Writer: will set uranium to the value ", isEqualTo, " in..." );
    countdown( 3 );
    uranium.write( isEqualTo );
  }
}

// sync vars have two states: empty and full.
// If you read an empty variable or write a full variable, you are waited
// until the variable is full or empty again
var someSyncVar$: sync int; // varName$ is a convention not a law.
sync {
  begin { // Reader task
    writeln( "Reader: waiting to read." );
    var read_sync = someSyncVar$;
    writeln( "Reader: value is ", read_sync );
  }

  begin { // Writer task
    writeln( "Writer: will write in..." );
    countdown( 3 );
    someSyncVar$ = 123;
  }
}

// single vars can only be written once. A read on an unwritten single results
// in a wait, but when the variable has a value it can be read indefinitely
var someSingleVar$: single int; // varName$ is a convention not a law.
sync {
  begin { // Reader task
    writeln( "Reader: waiting to read." );
    for i in 1..5 {
      var read_single = someSingleVar$;
      writeln( "Reader: iteration ", i,", and the value is ", read_single );
    }
  }

  begin { // Writer task
    writeln( "Writer: will write in..." );
    countdown( 3 );
    someSingleVar$ = 5; // first and only write ever.
  }
}

// Heres an example of using atomics and a synch variable to create a
// count-down mutex (also known as a multiplexer)
var count: atomic int; // our counter
var lock$: sync bool;   // the mutex lock

count.write( 2 );       // Only let two tasks in at a time.
lock$.writeXF( true );  // Set lock$ to full (unlocked)
// Note: The value doesnt actually matter, just the state
// (full:unlocked / empty:locked)
// Also, writeXF() fills (F) the sync var regardless of its state (X)

coforall task in 1..#5 { // Generate tasks
  // Create a barrier
  do{
    lock$;                 // Read lock$ (wait)
  }while ( count.read() < 1 ); // Keep waiting until a spot opens up

  count.sub(1);          // decrement the counter
  lock$.writeXF( true ); // Set lock$ to full (signal)

  // Actual 'work'
  writeln( "Task #", task, " doing work." );
  sleep( 2 );
```

```
    count.add( 1 );        // Increment the counter
    lock$.writeXF( true ); // Set lock$ to full (signal)
  }

  // we can define the operations + * & | ^ && || min max minloc maxloc
  // over an entire array using scans and reductions
  // Reductions apply the operation over the entire array and
  // result in a single value
  var listOfValues: [1..10] int = [15,57,354,36,45,15,456,8,678,2];
  var sumOfValues = + reduce listOfValues;
  var maxValue = max reduce listOfValues; // 'max' give just max value

  // 'maxloc' gives max value and index of the max value
  // Note: We have to zip the array and domain together with the zip iterator
  var (theMaxValue, idxOfMax) = maxloc reduce zip(listOfValues,
                                                  listOfValues.domain);

  writeln( (sumOfValues, maxValue, idxOfMax, listOfValues[ idxOfMax ] ) );

  // Scans apply the operation incrementally and return an array of the
  // value of the operation at that index as it progressed through the
  // array from array.domain.low to array.domain.high
  var runningSumOfValues = + scan listOfValues;
  var maxScan = max scan listOfValues;
  writeln( runningSumOfValues );
  writeln( maxScan );
} // end main()
```

Who is this tutorial for?

This tutorial is for people who want to learn the ropes of chapel without having to hear about what fiber mixture the ropes are, or how they were braided, or how the braid configurations differ between one another. It won't teach you how to develop amazingly performant code, and it's not exhaustive. Refer to the language specification and the module documentation for more details.

Occasionally check back here and on the Chapel site to see if more topics have been added or more tutorials created.

What this tutorial is lacking:

- Exposition of the standard modules
- Multiple Locales (distributed memory system)
- Records
- Parallel iterators

Your input, questions, and discoveries are important to the developers!

The Chapel language is still in-development (version 1.12.0), so there are occasional hiccups with performance and language features. The more information you give the Chapel development team about issues you encounter or features you would like to see, the better the language becomes. Feel free to email the team and other developers through the sourceforge email lists.

If you're really interested in the development of the compiler or contributing to the project, check out the master Github repository. It is under the Apache 2.0 License.

Installing the Compiler

Chapel can be built and installed on your average 'nix machine (and cygwin). Download the latest release version and it's as easy as

1. `tar -xvf chapel-1.12.0.tar.gz`
2. `cd chapel-1.12.0`
3. `make`
4. `source util/setchplenv.bash # or .sh or .csh or .fish`

You will need to `source util/setchplenv.EXT` from within the Chapel directory (`$CHPL_HOME`) every time your terminal starts so it's suggested that you drop that command in a script that will get executed on startup (like .bashrc).

Chapel is easily installed with Brew for OS X

1. `brew update`
2. `brew install chapel`

Compiling Code

Builds like other compilers:

`chpl myFile.chpl -o myExe`

Notable arguments:

- `--fast`: enables a number of optimizations and disables array bounds checks. Should only enable when application is stable.
- `--set <Symbol Name>=<Value>`: set config param `<Symbol Name>` to `<Value>` at compile-time.
- `--main-module <Module Name>`: use the main() procedure found in the module `<Module Name>` as the executable's main.
- `--module-dir <Directory>`: includes `<Directory>` in the module search path.

Clojure-Macros

As with all Lisps, Clojure's inherent homoiconicity gives you access to the full extent of the language to write code-generation routines called "macros". Macros provide a powerful way to tailor the language to your needs.

Be careful though. It's considered bad form to write a macro when a function will do. Use a macro only when you need control over when or if the arguments to a form will be evaluated.

You'll want to be familiar with Clojure. Make sure you understand everything in Clojure in Y Minutes.

```
;; Define a macro using defmacro. Your macro should output a list that can
;; be evaluated as clojure code.
;;
;; This macro is the same as if you wrote (reverse "Hello World")
(defmacro my-first-macro []
  (list reverse "Hello World"))

;; Inspect the result of a macro using macroexpand or macroexpand-1.
;;
```

```
;; Note that the call must be quoted.
(macroexpand '(my-first-macro))
;; -> (#<core$reverse clojure.core$reverse@xxxxxxxx> "Hello World")

;; You can eval the result of macroexpand directly:
(eval (macroexpand '(my-first-macro)))
; -> (\d \l \o \r \W \space \o \l \l \e \H)

;; But you should use this more succinct, function-like syntax:
(my-first-macro)  ; -> (\d \l \o \r \W \space \o \l \l \e \H)

;; You can make things easier on yourself by using the more succinct quote syntax
;; to create lists in your macros:
(defmacro my-first-quoted-macro []
  '(reverse "Hello World"))

(macroexpand '(my-first-quoted-macro))
;; -> (reverse "Hello World")
;; Notice that reverse is no longer function object, but a symbol.

;; Macros can take arguments.
(defmacro inc2 [arg]
  (list + 2 arg))

(inc2 2) ; -> 4

;; But, if you try to do this with a quoted list, you'll get an error, because
;; the argument will be quoted too. To get around this, clojure provides a
;; way of quoting macros: `. Inside `, you can use ~ to get at the outer scope
(defmacro inc2-quoted [arg]
  `(+ 2 ~arg))

(inc2-quoted 2)

;; You can use the usual destructuring args. Expand list variables using ~@
(defmacro unless [arg & body]
  `(if (not ~arg)
     (do ~@body))) ; Remember the do!

(macroexpand '(unless true (reverse "Hello World")))
;; ->
;; (if (clojure.core/not true) (do (reverse "Hello World")))

;; (unless) evaluates and returns its body if the first argument is false.
;; Otherwise, it returns nil

(unless true "Hello") ; -> nil
(unless false "Hello") ; -> "Hello"

;; Used without care, macros can do great evil by clobbering your vars
(defmacro define-x []
  '(do
     (def x 2)
     (list x)))
```

```
(def x 4)
(define-x) ; -> (2)
(list x) ; -> (2)

;; To avoid this, use gensym to get a unique identifier
(gensym 'x) ; -> x1281 (or some such thing)

(defmacro define-x-safely []
  (let [sym (gensym 'x)]
    `(do
       (def ~sym 2)
       (list ~sym))))

(def x 4)
(define-x-safely) ; -> (2)
(list x) ; -> (4)

;; You can use # within ` to produce a gensym for each symbol automatically
(defmacro define-x-hygenically []
  `(do
     (def x# 2)
     (list x#)))

(def x 4)
(define-x-hygenically) ; -> (2)
(list x) ; -> (4)

;; It's typical to use helper functions with macros. Let's create a few to
;; help us support a (dumb) inline arithmetic syntax
(declare inline-2-helper)
(defn clean-arg [arg]
  (if (seq? arg)
    (inline-2-helper arg)
    arg))

(defn apply-arg
  "Given args [x (+ y)], return (+ x y)"
  [val [op arg]]
  (list op val (clean-arg arg)))

(defn inline-2-helper
  [[arg1 & ops-and-args]]
  (let [ops (partition 2 ops-and-args)]
    (reduce apply-arg (clean-arg arg1) ops)))

;; We can test it immediately, without creating a macro
(inline-2-helper '(a + (b - 2) - (c * 5))) ; -> (- (+ a (- b 2)) (* c 5))

; However, we'll need to make it a macro if we want it to be run at compile time
(defmacro inline-2 [form]
  (inline-2-helper form))

(macroexpand '(inline-2 (1 + (3 / 2) - (1 / 2) + 1)))
```

```
; -> (+ (- (+ 1 (/ 3 2)) (/ 1 2)) 1)

(inline-2 (1 + (3 / 2) - (1 / 2) + 1))
; -> 3 (actually, 3N, since the number got cast to a rational fraction with /)
```

Further Reading

Writing Macros from Clojure for the Brave and True http://www.braveclojure.com/writing-macros/

Official docs http://clojure.org/macros

When to use macros? http://dunsmor.com/lisp/onlisp/onlisp_12.html

Clojure

Clojure is a Lisp family language developed for the Java Virtual Machine. It has a much stronger emphasis on pure functional programming than Common Lisp, but includes several STM utilities to handle state as it comes up.

This combination allows it to handle concurrent processing very simply, and often automatically.

(You need a version of Clojure 1.2 or newer)

```
; Comments start with semicolons.

; Clojure is written in "forms", which are just
; lists of things inside parentheses, separated by whitespace.
;
; The clojure reader assumes that the first thing is a
; function or macro to call, and the rest are arguments.

; The first call in a file should be ns, to set the namespace
(ns learnclojure)

; More basic examples:

; str will create a string out of all its arguments
(str "Hello" " " "World") ; => "Hello World"

; Math is straightforward
(+ 1 1) ; => 2
(- 2 1) ; => 1
(* 1 2) ; => 2
(/ 2 1) ; => 2

; Equality is =
(= 1 1) ; => true
(= 2 1) ; => false

; You need not for logic, too
(not true) ; => false

; Nesting forms works as you expect
(+ 1 (- 3 2)) ; = 1 + (3 - 2) => 2
```

```
; Types
;;;;;;;;;;;;;

; Clojure uses Java's object types for booleans, strings and numbers.
; Use `class` to inspect them.
(class 1) ; Integer literals are java.lang.Long by default
(class 1.); Float literals are java.lang.Double
(class ""); Strings always double-quoted, and are java.lang.String
(class false) ; Booleans are java.lang.Boolean
(class nil); The "null" value is called nil

; If you want to create a literal list of data, use ' to stop it from
; being evaluated
'(+ 1 2) ; => (+ 1 2)
; (shorthand for (quote (+ 1 2)))

; You can eval a quoted list
(eval '(+ 1 2)) ; => 3

; Collections & Sequences
;;;;;;;;;;;;;;;;;;;

; Lists are linked-list data structures, while Vectors are array-backed.
; Vectors and Lists are java classes too!
(class [1 2 3]); => clojure.lang.PersistentVector
(class '(1 2 3)); => clojure.lang.PersistentList

; A list would be written as just (1 2 3), but we have to quote
; it to stop the reader thinking it's a function.
; Also, (list 1 2 3) is the same as '(1 2 3)

; "Collections" are just groups of data
; Both lists and vectors are collections:
(coll? '(1 2 3)) ; => true
(coll? [1 2 3]) ; => true

; "Sequences" (seqs) are abstract descriptions of lists of data.
; Only lists are seqs.
(seq? '(1 2 3)) ; => true
(seq? [1 2 3]) ; => false

; A seq need only provide an entry when it is accessed.
; So, seqs which can be lazy -- they can define infinite series:
(range 4) ; => (0 1 2 3)
(range) ; => (0 1 2 3 4 ...) (an infinite series)
(take 4 (range)) ;  (0 1 2 3)

; Use cons to add an item to the beginning of a list or vector
(cons 4 [1 2 3]) ; => (4 1 2 3)
(cons 4 '(1 2 3)) ; => (4 1 2 3)

; Conj will add an item to a collection in the most efficient way.
; For lists, they insert at the beginning. For vectors, they insert at the end.
(conj [1 2 3] 4) ; => [1 2 3 4]
```

```
(conj '(1 2 3) 4) ; => (4 1 2 3)

; Use concat to add lists or vectors together
(concat [1 2] '(3 4)) ; => (1 2 3 4)

; Use filter, map to interact with collections
(map inc [1 2 3]) ; => (2 3 4)
(filter even? [1 2 3]) ; => (2)

; Use reduce to reduce them
(reduce + [1 2 3 4])
; = (+ (+ (+ 1 2) 3) 4)
; => 10

; Reduce can take an initial-value argument too
(reduce conj [] '(3 2 1))
; = (conj (conj (conj [] 3) 2) 1)
; => [3 2 1]

; Functions
;;;;;;;;;;;;;;;;;;;;;

; Use fn to create new functions. A function always returns
; its last statement.
(fn [] "Hello World") ; => fn

; (You need extra parens to call it)
((fn [] "Hello World")) ; => "Hello World"

; You can create a var using def
(def x 1)
x ; => 1

; Assign a function to a var
(def hello-world (fn [] "Hello World"))
(hello-world) ; => "Hello World"

; You can shorten this process by using defn
(defn hello-world [] "Hello World")

; The [] is the list of arguments for the function.
(defn hello [name]
  (str "Hello " name))
(hello "Steve") ; => "Hello Steve"

; You can also use this shorthand to create functions:
(def hello2 #(str "Hello " %1))
(hello2 "Fanny") ; => "Hello Fanny"

; You can have multi-variadic functions, too
(defn hello3
  ([] "Hello World")
  ([name] (str "Hello " name)))
(hello3 "Jake") ; => "Hello Jake"
```

```
(hello3) ; => "Hello World"

; Functions can pack extra arguments up in a seq for you
(defn count-args [& args]
  (str "You passed " (count args) " args: " args))
(count-args 1 2 3) ; => "You passed 3 args: (1 2 3)"

; You can mix regular and packed arguments
(defn hello-count [name & args]
  (str "Hello " name ", you passed " (count args) " extra args"))
(hello-count "Finn" 1 2 3)
; => "Hello Finn, you passed 3 extra args"


; Maps
;;;;;;;;;;

; Hash maps and array maps share an interface. Hash maps have faster lookups
; but don't retain key order.
(class {:a 1 :b 2 :c 3}) ; => clojure.lang.PersistentArrayMap
(class (hash-map :a 1 :b 2 :c 3)) ; => clojure.lang.PersistentHashMap

; Arraymaps will automatically become hashmaps through most operations
; if they get big enough, so you don't need to worry.

; Maps can use any hashable type as a key, but usually keywords are best
; Keywords are like strings with some efficiency bonuses
(class :a) ; => clojure.lang.Keyword

(def stringmap {"a" 1, "b" 2, "c" 3})
stringmap  ; => {"a" 1, "b" 2, "c" 3}

(def keymap {:a 1, :b 2, :c 3})
keymap ; => {:a 1, :c 3, :b 2}

; By the way, commas are always treated as whitespace and do nothing.

; Retrieve a value from a map by calling it as a function
(stringmap "a") ; => 1
(keymap :a) ; => 1

; Keywords can be used to retrieve their value from a map, too!
(:b keymap) ; => 2

; Don't try this with strings.
;("a" stringmap)
; => Exception: java.lang.String cannot be cast to clojure.lang.IFn

; Retrieving a non-present key returns nil
(stringmap "d") ; => nil

; Use assoc to add new keys to hash-maps
(def newkeymap (assoc keymap :d 4))
newkeymap ; => {:a 1, :b 2, :c 3, :d 4}
```

```
; But remember, clojure types are immutable!
keymap ; => {:a 1, :b 2, :c 3}

; Use dissoc to remove keys
(dissoc keymap :a :b) ; => {:c 3}

; Sets
;;;;;;

(class #{1 2 3}) ; => clojure.lang.PersistentHashSet
(set [1 2 3 1 2 3 3 2 1 3 2 1]) ; => #{1 2 3}

; Add a member with conj
(conj #{1 2 3} 4) ; => #{1 2 3 4}

; Remove one with disj
(disj #{1 2 3} 1) ; => #{2 3}

; Test for existence by using the set as a function:
(#{1 2 3} 1) ; => 1
(#{1 2 3} 4) ; => nil

; There are more functions in the clojure.sets namespace.

; Useful forms
;;;;;;;;;;;;;;;;;

; Logic constructs in clojure are just macros, and look like
; everything else
(if false "a" "b") ; => "b"
(if false "a") ; => nil

; Use let to create temporary bindings
(let [a 1 b 2]
  (> a b)) ; => false

; Group statements together with do
(do
  (print "Hello")
  "World") ; => "World" (prints "Hello")

; Functions have an implicit do
(defn print-and-say-hello [name]
  (print "Saying hello to " name)
  (str "Hello " name))
(print-and-say-hello "Jeff") ;=> "Hello Jeff" (prints "Saying hello to Jeff")

; So does let
(let [name "Urkel"]
  (print "Saying hello to " name)
  (str "Hello " name)) ; => "Hello Urkel" (prints "Saying hello to Urkel")
```

```
; Use the threading macros (-> and ->>) to express transformations of
; data more clearly.

; The "Thread-first" macro (->) inserts into each form the result of
; the previous, as the first argument (second item)
(->
   {:a 1 :b 2}
   (assoc :c 3) ;=> (assoc {:a 1 :b 2} :c 3)
   (dissoc :b)) ;=> (dissoc (assoc {:a 1 :b 2} :c 3) :b)

; This expression could be written as:
; (dissoc (assoc {:a 1 :b 2} :c 3) :b)
; and evaluates to {:a 1 :c 3}

; The double arrow does the same thing, but inserts the result of
; each line at the *end* of the form. This is useful for collection
; operations in particular:
(->>
   (range 10)
   (map inc)     ;=> (map inc (range 10)
   (filter odd?) ;=> (filter odd? (map inc (range 10))
   (into []))    ;=> (into [] (filter odd? (map inc (range 10)))
                 ; Result: [1 3 5 7 9]

; Modules
;;;;;;;;;;;;;;;

; Use "use" to get all functions from the module
(use 'clojure.set)

; Now we can use set operations
(intersection #{1 2 3} #{2 3 4}) ; => #{2 3}
(difference #{1 2 3} #{2 3 4}) ; => #{1}

; You can choose a subset of functions to import, too
(use '[clojure.set :only [intersection]])

; Use require to import a module
(require 'clojure.string)

; Use / to call functions from a module
; Here, the module is clojure.string and the function is blank?
(clojure.string/blank? "") ; => true

; You can give a module a shorter name on import
(require '[clojure.string :as str])
(str/replace "This is a test." #"[a-o]" str/upper-case) ; => "THIs Is A tEst."
; (#"" denotes a regular expression literal)

; You can use require (and use, but don't) from a namespace using :require.
; You don't need to quote your modules if you do it this way.
(ns test
  (:require
    [clojure.string :as str]
    [clojure.set :as set]))

; Java
;;;;;;;;;;;;;;;;;

; Java has a huge and useful standard library, so
; you'll want to learn how to get at it.

; Use import to load a java module
(import java.util.Date)

; You can import from an ns too.
(ns test
  (:import java.util.Date
           java.util.Calendar))

; Use the class name with a "." at the end to make a new instance
(Date.) ; <a date object>

; Use . to call methods. Or, use the ".method" shortcut
(. (Date.) getTime) ; <a timestamp>
(.getTime (Date.)) ; exactly the same thing.

; Use / to call static methods
(System/currentTimeMillis) ; <a timestamp> (system is always present)

; Use doto to make dealing with (mutable) classes more tolerable
(import java.util.Calendar)
(doto (Calendar/getInstance)
  (.set 2000 1 1 0 0 0)
  .getTime) ; => A Date. set to 2000-01-01 00:00:00

; STM
;;;;;;;;;;;;;;;;;

; Software Transactional Memory is the mechanism clojure uses to handle
; persistent state. There are a few constructs in clojure that use this.

; An atom is the simplest. Pass it an initial value
(def my-atom (atom {}))

; Update an atom with swap!.
; swap! takes a function and calls it with the current value of the atom
; as the first argument, and any trailing arguments as the second
(swap! my-atom assoc :a 1) ; Sets my-atom to the result of (assoc {} :a 1)
(swap! my-atom assoc :b 2) ; Sets my-atom to the result of (assoc {:a 1} :b 2)

; Use '@' to dereference the atom and get the value
my-atom  ;=> Atom<#...> (Returns the Atom object)
@my-atom ; => {:a 1 :b 2}

; Here's a simple counter using an atom
(def counter (atom 0))
(defn inc-counter []
  (swap! counter inc))

(inc-counter)
(inc-counter)
(inc-counter)
(inc-counter)
(inc-counter)

@counter ; => 5

; Other STM constructs are refs and agents.
; Refs: http://clojure.org/refs
; Agents: http://clojure.org/agents
```

Further Reading

This is far from exhaustive, but hopefully it's enough to get you on your feet.

Clojure.org has lots of articles: http://clojure.org/

Clojuredocs.org has documentation with examples for most core functions: http://clojuredocs.org/quickref/Clojure%20Core

4Clojure is a great way to build your clojure/FP skills: http://www.4clojure.com/

Clojure-doc.org (yes, really) has a number of getting started articles: http://clojure-doc.org/

Coffeescript

CoffeeScript is a little language that compiles one-to-one into the equivalent JavaScript, and there is no interpretation at runtime. As one of the successors to JavaScript, CoffeeScript tries its best to output readable, pretty-printed and smooth-running JavaScript code, which works well in every JavaScript runtime.

See also the CoffeeScript website, which has a complete tutorial on CoffeeScript.

```
# CoffeeScript is a hipster language.
# It goes with the trends of many modern languages.
# So comments are like Ruby and Python, they use number symbols.

###
Block comments are like these, and they translate directly to '/ *'s and '* /'s
for the resulting JavaScript code.

You should understand most of JavaScript semantics
before continuing.
###

# Assignment:
number   = 42 #=> var number = 42;
opposite = true #=> var opposite = true;

# Conditions:
number = -42 if opposite #=> if(opposite) { number = -42; }

# Functions:
```

```
square = (x) -> x * x #=> var square = function(x) { return x * x; }

fill = (container, liquid = "coffee") ->
  "Filling the #{container} with #{liquid}..."
#=>var fill;
#
#fill = function(container, liquid) {
#  if (liquid == null) {
#    liquid = "coffee";
#  }
#  return "Filling the " + container + " with " + liquid + "...";
#};

# Ranges:
list = [1..5] #=> var list = [1, 2, 3, 4, 5];

# Objects:
math =
  root:   Math.sqrt
  square: square
  cube:   (x) -> x * square x
#=> var math = {
#    "root": Math.sqrt,
#    "square": square,
#    "cube": function(x) { return x * square(x); }
#   };

# Splats:
race = (winner, runners...) ->
  print winner, runners
#=>race = function() {
#    var runners, winner;
#    winner = arguments[0], runners = 2 <= arguments.length ? __slice.call(arguments, 1) : [];
#    return print(winner, runners);
#  };

# Existence:
alert "I knew it!" if elvis?
#=> if(typeof elvis !== "undefined" && elvis !== null) { alert("I knew it!"); }

# Array comprehensions:
cubes = (math.cube num for num in list)
#=>cubes = (function() {
#     var _i, _len, _results;
#     _results = [];
#   for (_i = 0, _len = list.length; _i < _len; _i++) {
#         num = list[_i];
#         _results.push(math.cube(num));
#     }
#     return _results;
# })();

foods = ['broccoli', 'spinach', 'chocolate']
eat food for food in foods when food isnt 'chocolate'
```

```
#=>foods = ['broccoli', 'spinach', 'chocolate'];
#
#for (_k = 0, _len2 = foods.length; _k < _len2; _k++) {
#  food = foods[_k];
#  if (food !== 'chocolate') {
#    eat(food);
#  }
#}
```

Additional resources

- Smooth CoffeeScript
- CoffeeScript Ristretto

Coldfusion

ColdFusion is a scripting language for web development. Read more here.

CFML

**C**old**F**usion **M**arkup **L**anguage
ColdFusion started as a tag-based language. Almost all functionality is available using tags.

```
<em>HTML tags have been provided for output readability</em>

<!--- Comments start with "<!---" and end with "--->" --->
<!---
    Comments can
    also
    span
    multiple lines
--->

<!--- CFML tags have a similar format to HTML tags. --->
<h1>Simple Variables</h1>
<!--- Variable Declaration: Variables are loosely typed, similar to javascript --->
<p>Set <b>myVariable</b> to "myValue"</p>
<cfset myVariable = "myValue" />
<p>Set <b>myNumber</b> to 3.14</p>
<cfset myNumber = 3.14 />

<!--- Displaying simple data --->
<!--- Use <cfoutput> for simple values such as strings, numbers, and expressions --->
<p>Display <b>myVariable</b>: <cfoutput>#myVariable#</cfoutput></p><!--- myValue --->
<p>Display <b>myNumber</b>: <cfoutput>#myNumber#</cfoutput></p><!--- 3.14 --->

<hr />

<h1>Complex Variables</h1>
<!--- Declaring complex variables --->
<!--- Declaring an array of 1 dimension: literal or bracket notation --->
<p>Set <b>myArray1</b> to an array of 1 dimension using literal or bracket notation</p>
```

```
<cfset myArray1 = [] />
<!--- Declaring an array of 1 dimension: function notation --->
<p>Set <b>myArray2</b> to an array of 1 dimension using function notation</p>
<cfset myArray2 = ArrayNew(1) />

<!--- Outputting complex variables --->
<p>Contents of <b>myArray1</b></p>
<cfdump var="#myArray1#" /> <!--- An empty array object --->
<p>Contents of <b>myArray2</b></p>
<cfdump var="#myArray2#" /> <!--- An empty array object --->

<!--- Operators --->
<!--- Arithmetic --->
<h1>Operators</h1>
<h2>Arithmetic</h2>
<p>1 + 1 = <cfoutput>#1 + 1#</cfoutput></p>
<p>10 - 7 = <cfoutput>#10 - 7#<br /></cfoutput></p>
<p>15 * 10 = <cfoutput>#15 * 10#<br /></cfoutput></p>
<p>100 / 5 = <cfoutput>#100 / 5#<br /></cfoutput></p>
<p>120 % 5 = <cfoutput>#120 % 5#<br /></cfoutput></p>
<p>120 mod 5 = <cfoutput>#120 mod 5#<br /></cfoutput></p>

<hr />

<!--- Comparison --->
<h2>Comparison</h2>
<h3>Standard Notation</h3>
<p>Is 1 eq 1? <cfoutput>#1 eq 1#</cfoutput></p>
<p>Is 15 neq 1? <cfoutput>#15 neq 1#</cfoutput></p>
<p>Is 10 gt 8? <cfoutput>#10 gt 8#</cfoutput></p>
<p>Is 1 lt 2? <cfoutput>#1 lt 2#</cfoutput></p>
<p>Is 10 gte 5? <cfoutput>#10 gte 5#</cfoutput></p>
<p>Is 1 lte 5? <cfoutput>#1 lte 5#</cfoutput></p>

<h3>Alternative Notation</h3>
<p>Is 1 == 1? <cfoutput>#1 eq 1#</cfoutput></p>
<p>Is 15 != 1? <cfoutput>#15 neq 1#</cfoutput></p>
<p>Is 10 > 8? <cfoutput>#10 gt 8#</cfoutput></p>
<p>Is 1 < 2? <cfoutput>#1 lt 2#</cfoutput></p>
<p>Is 10 >= 5? <cfoutput>#10 gte 5#</cfoutput></p>
<p>Is 1 <= 5? <cfoutput>#1 lte 5#</cfoutput></p>

<hr />

<!--- Control Structures --->
<h1>Control Structures</h1>

<cfset myCondition = "Test" />

<p>Condition to test for: "<cfoutput>#myCondition#</cfoutput>"</p>

<cfif myCondition eq "Test">
    <cfoutput>#myCondition#. We're testing.</cfoutput>
<cfelseif myCondition eq "Production">
```

```
    <cfoutput>#myCondition#. Proceed Carefully!!!</cfoutput>
<cfelse>
    myCondition is unknown
</cfif>

<hr />

<!--- Loops --->
<h1>Loops</h1>
<h2>For Loop</h2>
<cfloop from="0" to="10" index="i">
    <p>Index equals <cfoutput>#i#</cfoutput></p>
</cfloop>

<h2>For Each Loop (Complex Variables)</h2>

<p>Set <b>myArray3</b> to [5, 15, 99, 45, 100]</p>

<cfset myArray3 = [5, 15, 99, 45, 100] />

<cfloop array="#myArray3#" index="i">
    <p>Index equals <cfoutput>#i#</cfoutput></p>
</cfloop>

<p>Set <b>myArray4</b> to ["Alpha", "Bravo", "Charlie", "Delta", "Echo"]</p>

<cfset myArray4 = ["Alpha", "Bravo", "Charlie", "Delta", "Echo"] />

<cfloop array="#myArray4#" index="s">
    <p>Index equals <cfoutput>#s#</cfoutput></p>
</cfloop>

<h2>Switch Statement</h2>

<p>Set <b>myArray5</b> to [5, 15, 99, 45, 100]</p>

<cfset myArray5 = [5, 15, 99, 45, 100] />

<cfloop array="#myArray5#" index="i">
    <cfswitch expression="#i#">
        <cfcase value="5,15,45" delimiters=",">
            <p><cfoutput>#i#</cfoutput> is a multiple of 5.</p>
        </cfcase>
        <cfcase value="99">
            <p><cfoutput>#i#</cfoutput> is ninety-nine.</p>
        </cfcase>
        <cfdefaultcase>
            <p><cfoutput>#i#</cfoutput> is not 5, 15, 45, or 99.</p>
        </cfdefaultcase>
    </cfswitch>
</cfloop>

<hr />
```

```
<h1>Converting types</h1>

<style>
    table.table th, table.table td {
        border: 1px solid #000000;
        padding: 2px;
    }

    table.table th {
        background-color: #CCCCCC;
    }
</style>

<table class="table" cellspacing="0">
    <thead>
        <tr>
            <th>Value</th>
            <th>As Boolean</th>
            <th>As number</th>
            <th>As date-time</th>
            <th>As string</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <th>"Yes"</th>
            <td>TRUE</td>
            <td>1</td>
            <td>Error</td>
            <td>"Yes"</td>
        </tr>
        <tr>
            <th>"No"</th>
            <td>FALSE</td>
            <td>0</td>
            <td>Error</td>
            <td>"No"</td>
        </tr>
        <tr>
            <th>TRUE</th>
            <td>TRUE</td>
            <td>1</td>
            <td>Error</td>
            <td>"Yes"</td>
        </tr>
        <tr>
            <th>FALSE</th>
            <td>FALSE</td>
            <td>0</td>
            <td>Error</td>
            <td>"No"</td>
        </tr>
        <tr>
            <th>Number</th>
            <td>True if Number is not 0; False otherwise.</td>
            <td>Number</td>
            <td>See &#34;Date-time values&#34; earlier in this chapter.</td>
            <td>String representation of the number (for example, &#34;8&#34;).</td>
        </tr>
        <tr>
            <th>String</th>
            <td>If "Yes", True <br>If "No", False <br>If it can be converted to 0, False <br>If it can
            <td>If it represents a number (for example, &#34;1,000&#34; or &#34;12.36E-12&#34;), it is
            <td>If it represents a date-time (see next column), it is converted to the numeric value of
            <td>String</td>
        </tr>
        <tr>
            <th>Date</th>
            <td>Error</td>
            <td>The numeric value of the date-time object.</td>
            <td>Date</td>
            <td>An ODBC timestamp.</td>
        </tr>
    </tbody>
</table>

<hr />

<h1>Components</h1>

<em>Code for reference (Functions must return something to support IE)</em>

<cfcomponent>
    <cfset this.hello = "Hello" />
    <cfset this.world = "world" />

    <cffunction name="sayHello">
        <cfreturn this.hello & ", " & this.world & "!" />
    </cffunction>

    <cffunction name="setHello">
        <cfargument name="newHello" type="string" required="true" />

        <cfset this.hello = arguments.newHello />

        <cfreturn true />
    </cffunction>

    <cffunction name="setWorld">
        <cfargument name="newWorld" type="string" required="true" />

        <cfset this.world = arguments.newWorld />

        <cfreturn true />
    </cffunction>

    <cffunction name="getHello">
        <cfreturn this.hello />
    </cffunction>

    <cffunction name="getWorld">
        <cfreturn this.world />
    </cffunction>
</cfcomponent>

<cfset this.hello = "Hello" />
<cfset this.world = "world" />

<cffunction name="sayHello">
    <cfreturn this.hello & ", " & this.world & "!" />
</cffunction>

<cffunction name="setHello">
    <cfargument name="newHello" type="string" required="true" />

    <cfset this.hello = arguments.newHello />

    <cfreturn true />
</cffunction>

<cffunction name="setWorld">
    <cfargument name="newWorld" type="string" required="true" />

    <cfset this.world = arguments.newWorld />

    <cfreturn true />
</cffunction>

<cffunction name="getHello">
    <cfreturn this.hello />
</cffunction>

<cffunction name="getWorld">
    <cfreturn this.world />
</cffunction>


<b>sayHello()</b>
<cfoutput><p>#sayHello()#</p></cfoutput>
<b>getHello()</b>
<cfoutput><p>#getHello()#</p></cfoutput>
<b>getWorld()</b>
<cfoutput><p>#getWorld()#</p></cfoutput>
<b>setHello("Hola")</b>
<cfoutput><p>#setHello("Hola")#</p></cfoutput>
<b>setWorld("mundo")</b>
<cfoutput><p>#setWorld("mundo")#</p></cfoutput>
<b>sayHello()</b>
<cfoutput><p>#sayHello()#</p></cfoutput>
<b>getHello()</b>
<cfoutput><p>#getHello()#</p></cfoutput>
<b>getWorld()</b>
```

```
<cfoutput><p>#getWorld()#</p></cfoutput>
```

CFScript

C*old****F****usion* ***S****cript*
In recent years, the ColdFusion language has added script syntax to mirror tag functionality. When using an up-to-date CF server, almost all functionality is available using scrypt syntax.

Further Reading

The links provided here below are just to get an understanding of the topic, feel free to Google and find specific examples.

1. Coldfusion Reference From Adobe
2. Open Source Documentation

Common-Lisp

ANSI Common Lisp is a general purpose, multi-paradigm programming language suited for a wide variety of industry applications. It is frequently referred to as a programmable programming language.

The classic starting point is Practical Common Lisp and freely available.

Another popular and recent book is Land of Lisp.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 0. Syntax
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; General form.

;; Lisp has two fundamental pieces of syntax: the ATOM and the
;; S-expression. Typically, grouped S-expressions are called `forms`.

10  ; an atom; it evaluates to itself

:THING ;Another atom; evaluating to the symbol :thing.

t  ; another atom, denoting true.

(+ 1 2 3 4) ; an s-expression

'(4 :foo  t)  ;another one


;;; Comments

;; Single line comments start with a semicolon; use two for normal
;; comments, three for section comments, and four for file-level
;; comments.

#| Block comments
   can span multiple lines and...
```

```
    #|
       they can be nested!
    |#
|#

;;; Environment.

;; A variety of implementations exist; most are
;; standard-conformant. CLISP is a good starting one.

;; Libraries are managed through Quicklisp.org's Quicklisp system.

;; Common Lisp is usually developed with a text editor and a REPL
;; (Read Evaluate Print Loop) running at the same time. The REPL
;; allows for interactive exploration of the program as it is "live"
;; in the system.


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 1. Primitive Datatypes and Operators
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Symbols

'foo ; => FOO  Notice that the symbol is upper-cased automatically.

;; Intern manually creates a symbol from a string.

(intern "AAAA") ; => AAAA

(intern "aaa") ; => |aaa|

;;; Numbers
9999999999999999999999 ; integers
#b111                  ; binary => 7
#o111                  ; octal => 73
#x111                  ; hexadecimal => 273
3.14159s0              ; single
3.14159d0              ; double
1/2                    ; ratios
#C(1 2)                ; complex numbers


;; Function application is written (f x y z ...)
;; where f is a function and x, y, z, ... are operands
;; If you want to create a literal list of data, use ' to stop it from
;; being evaluated - literally, "quote" the data.
'(+ 1 2) ; => (+ 1 2)
;; You can also call a function manually:
(funcall #'+ 1 2 3) ; => 6
;; Some arithmetic operations
(+ 1 1)              ; => 2
(- 8 1)              ; => 7
(* 10 2)             ; => 20
```

```
(expt 2 3)           ; => 8
(mod 5 2)            ; => 1
(/ 35 5)             ; => 7
(/ 1 3)              ; => 1/3
(+ #C(1 2) #C(6 -4)) ; => #C(7 -2)

                     ;;; Booleans
t                    ; for true (any not-nil value is true)
nil                  ; for false - and the empty list
(not nil)            ; => t
(and 0 t)            ; => t
(or 0 nil)           ; => 0

                     ;;; Characters
#\A                  ; => #\A
#\                  ; => #\GREEK_SMALL_LETTER_LAMDA
#\u03BB              ; => #\GREEK_SMALL_LETTER_LAMDA

;;; Strings are fixed-length arrays of characters.
"Hello, world!"
"Benjamin \"Bugsy\" Siegel"   ; backslash is an escaping character

;; Strings can be concatenated too!
(concatenate 'string "Hello " "world!") ; => "Hello world!"

;; A string can be treated like a sequence of characters
(elt "Apple" 0) ; => #\A

;; format can be used to format strings:
(format nil "~a can be ~a" "strings" "formatted")

;; Printing is pretty easy; ~% is the format specifier for newline.
(format t "Common Lisp is groovy. Dude.~%")


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 2. Variables
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; You can create a global (dynamically scoped) using defparameter
;; a variable name can use any character except: ()",'`;#|\

;; Dynamically scoped variables should have earmuffs in their name!

(defparameter *some-var* 5)
*some-var* ; => 5

;; You can also use unicode characters.
(defparameter *AΛB* nil)


;; Accessing a previously unbound variable is an
;; undefined behavior (but possible). Don't do it.
;; Local binding: `me` is bound to "dance with you" only within the
;; (let ...). Let always returns the value of the last `form` in the
;; let form.

(let ((me "dance with you"))
  me)
;; => "dance with you"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 3. Structs and Collections
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Structs
(defstruct dog name breed age)
(defparameter *rover*
    (make-dog :name "rover"
              :breed "collie"
              :age 5))
*rover* ; => #S(DOG :NAME "rover" :BREED "collie" :AGE 5)

(dog-p *rover*) ; => true  #| -p signifies "predicate". It's used to
                              check if *rover* is an instance of dog. |#
(dog-name *rover*) ; => "rover"

;; Dog-p, make-dog, and dog-name are all created by defstruct!

;;; Pairs
;; `cons' constructs pairs, `car' and `cdr' extract the first
;; and second elements
(cons 'SUBJECT 'VERB) ; => '(SUBJECT . VERB)
(car (cons 'SUBJECT 'VERB)) ; => SUBJECT
(cdr (cons 'SUBJECT 'VERB)) ; => VERB

;;; Lists

;; Lists are linked-list data structures, made of `cons' pairs and end
;; with a `nil' (or '()) to mark the end of the list
(cons 1 (cons 2 (cons 3 nil))) ; => '(1 2 3)
;; `list' is a convenience variadic constructor for lists
(list 1 2 3) ; => '(1 2 3)
;; and a quote can also be used for a literal list value
'(1 2 3) ; => '(1 2 3)

;; Can still use `cons' to add an item to the beginning of a list
(cons 4 '(1 2 3)) ; => '(4 1 2 3)

;; Use `append' to - surprisingly - append lists together
(append '(1 2) '(3 4)) ; => '(1 2 3 4)

;; Or use concatenate -

(concatenate 'list '(1 2) '(3 4))

;; Lists are a very central type, so there is a wide variety of functionality for
;; them, a few examples:
(mapcar #'1+ '(1 2 3))             ; => '(2 3 4)
(mapcar #'+ '(1 2 3) '(10 20 30))  ; => '(11 22 33)
(remove-if-not #'evenp '(1 2 3 4)) ; => '(2 4)
(every #'evenp '(1 2 3 4))         ; => nil
(some #'oddp '(1 2 3 4))           ; => T
(butlast '(subject verb object))   ; => (SUBJECT VERB)


;;; Vectors

;; Vector's literals are fixed-length arrays
#(1 2 3) ; => #(1 2 3)

;; Use concatenate to add vectors together
(concatenate 'vector #(1 2 3) #(4 5 6)) ; => #(1 2 3 4 5 6)

;;; Arrays

;; Both vectors and strings are special-cases of arrays.

;; 2D arrays

(make-array (list 2 2))

;; (make-array '(2 2)) works as well.

; => #2A((0 0) (0 0))

(make-array (list 2 2 2))

; => #3A(((0 0) (0 0)) ((0 0) (0 0)))

;; Caution- the default initial values are
;; implementation-defined. Here's how to define them:

(make-array '(2) :initial-element 'unset)

; => #(UNSET UNSET)

;; And, to access the element at 1,1,1 -
(aref (make-array (list 2 2 2)) 1 1 1)

; => 0

;;; Adjustable vectors

;; Adjustable vectors have the same printed representation
;; as fixed-length vector's literals.

(defparameter *adjvec* (make-array '(3) :initial-contents '(1 2 3)
      :adjustable t :fill-pointer t))

*adjvec* ; => #(1 2 3)
```

```
;; Adding new element:
(vector-push-extend 4 *adjvec*) ; => 3

*adjvec* ; => #(1 2 3 4)



;;; Naively, sets are just lists:

(set-difference '(1 2 3 4) '(4 5 6 7)) ; => (3 2 1)
(intersection '(1 2 3 4) '(4 5 6 7)) ; => 4
(union '(1 2 3 4) '(4 5 6 7))        ; => (3 2 1 4 5 6 7)
(adjoin 4 '(1 2 3 4))     ; => (1 2 3 4)

;; But you'll want to use a better data structure than a linked list
;; for performant work!

;;; Dictionaries are implemented as hash tables.

;; Create a hash table
(defparameter *m* (make-hash-table))

;; set a value
(setf (gethash 'a *m*) 1)

;; Retrieve a value
(gethash 'a *m*) ; => 1, t

;; Detail - Common Lisp has multiple return values possible. gethash
;; returns t in the second value if anything was found, and nil if
;; not.

;; Retrieving a non-present value returns nil
 (gethash 'd *m*) ;=> nil, nil

;; You can provide a default value for missing keys
(gethash 'd *m* :not-found) ; => :NOT-FOUND

;; Let's handle the multiple return values here in code.

(multiple-value-bind
      (a b)
    (gethash 'd *m*)
  (list a b))
; => (NIL NIL)

(multiple-value-bind
      (a b)
    (gethash 'a *m*)
  (list a b))
; => (1 T)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; 3. Functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Use `lambda' to create anonymous functions.
;; A function always returns the value of its last expression.
;; The exact printable representation of a function will vary...

(lambda () "Hello World") ; => #<FUNCTION (LAMBDA ()) {1004E7818B}>

;; Use funcall to call lambda functions
(funcall (lambda () "Hello World")) ; => "Hello World"

;; Or Apply
(apply (lambda () "Hello World") nil) ; => "Hello World"

;; De-anonymize the function
(defun hello-world ()
   "Hello World")
(hello-world) ; => "Hello World"

;; The () in the above is the list of arguments for the function
(defun hello (name)
   (format nil "Hello, ~a" name))

(hello "Steve") ; => "Hello, Steve"

;; Functions can have optional arguments; they default to nil

(defun hello (name &optional from)
    (if from
        (format t "Hello, ~a, from ~a" name from)
        (format t "Hello, ~a" name)))

 (hello "Jim" "Alpacas") ;; => Hello, Jim, from Alpacas

;; And the defaults can be set...
(defun hello (name &optional (from "The world"))
   (format t "Hello, ~a, from ~a" name from))

(hello "Steve")
; => Hello, Steve, from The world

(hello "Steve" "the alpacas")
; => Hello, Steve, from the alpacas


;; And of course, keywords are allowed as well... usually more
;;   flexible than &optional.

(defun generalized-greeter (name &key (from "the world") (honorific "Mx"))
    (format t "Hello, ~a ~a, from ~a" honorific name from))

(generalized-greeter "Jim")   ; => Hello, Mx Jim, from the world
```

```
(generalized-greeter "Jim" :from "the alpacas you met last summer" :honorific "Mr")
; => Hello, Mr Jim, from the alpacas you met last summer

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 4. Equality
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Common Lisp has a sophisticated equality system. A couple are covered here.

;; for numbers use `='
(= 3 3.0) ; => t
(= 2 1) ; => nil

;; for object identity (approximately) use `eql`
(eql 3 3) ; => t
(eql 3 3.0) ; => nil
(eql (list 3) (list 3)) ; => nil

;; for lists, strings, and bit-vectors use `equal'
(equal (list 'a 'b) (list 'a 'b)) ; => t
(equal (list 'a 'b) (list 'b 'a)) ; => nil

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 5. Control Flow
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Conditionals

(if t                 ; test expression
    "this is true"    ; then expression
    "this is false") ; else expression
; => "this is true"

;; In conditionals, all non-nil values are treated as true
(member 'Groucho '(Harpo Groucho Zeppo)) ; => '(GROUCHO ZEPPO)
(if (member 'Groucho '(Harpo Groucho Zeppo))
    'yep
    'nope)
; => 'YEP

;; `cond' chains a series of tests to select a result
(cond ((> 2 2) (error "wrong!"))
      ((< 2 2) (error "wrong again!"))
      (t 'ok)) ; => 'OK

;; Typecase switches on the type of the value
(typecase 1
  (string :string)
  (integer :int))

; => :int

;;; Iteration
```

```
;; Of course recursion is supported:

(defun walker (n)
  (if (zerop n)
      :walked
      (walker (- n 1))))

(walker 5) ; => :walked

;; Most of the time, we use DOLIST or LOOP


(dolist (i '(1 2 3 4))
  (format t "~a" i))

; => 1234

(loop for i from 0 below 10
      collect i)

; => (0 1 2 3 4 5 6 7 8 9)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 6. Mutation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Use `setf' to assign a new value to an existing variable. This was
;; demonstrated earlier in the hash table example.

(let ((variable 10))
    (setf variable 2))
 ; => 2


;; Good Lisp style is to minimize destructive functions and to avoid
;; mutation when reasonable.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 7. Classes and Objects
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; No more Animal classes, let's have Human-Powered Mechanical
;; Conveyances.

(defclass human-powered-conveyance ()
  ((velocity
    :accessor velocity
    :initarg :velocity)
   (average-efficiency
    :accessor average-efficiency
   :initarg :average-efficiency))
  (:documentation "A human powered conveyance"))
```

```
;; defclass, followed by name, followed by the superclass list,
;; followed by slot list, followed by optional qualities such as
;; :documentation.

;; When no superclass list is set, the empty list defaults to the
;; standard-object class. This *can* be changed, but not until you
;; know what you're doing. Look up the Art of the Metaobject Protocol
;; for more information.

(defclass bicycle (human-powered-conveyance)
  ((wheel-size
    :accessor wheel-size
    :initarg :wheel-size
    :documentation "Diameter of the wheel.")
   (height
    :accessor height
    :initarg :height)))

(defclass recumbent (bicycle)
  ((chain-type
    :accessor chain-type
    :initarg  :chain-type)))

(defclass unicycle (human-powered-conveyance) nil)

(defclass canoe (human-powered-conveyance)
  ((number-of-rowers
    :accessor number-of-rowers
    :initarg :number-of-rowers)))


;; Calling DESCRIBE on the human-powered-conveyance class in the REPL gives:

(describe 'human-powered-conveyance)

; COMMON-LISP-USER::HUMAN-POWERED-CONVEYANCE
;  [symbol]
;
; HUMAN-POWERED-CONVEYANCE names the standard-class #<STANDARD-CLASS
;                                                     HUMAN-POWERED-CONVEYANCE>:
;  Documentation:
;    A human powered conveyance
;  Direct superclasses: STANDARD-OBJECT
;  Direct subclasses: UNICYCLE, BICYCLE, CANOE
;  Not yet finalized.
;  Direct slots:
;    VELOCITY
;      Readers: VELOCITY
;      Writers: (SETF VELOCITY)
;    AVERAGE-EFFICIENCY
;      Readers: AVERAGE-EFFICIENCY
;      Writers: (SETF AVERAGE-EFFICIENCY)

;; Note the reflective behavior available to you! Common Lisp is
;; designed to be an interactive system

;; To define a method, let's find out what our circumference of the
;; bike wheel turns out to be using the equation: C = d * pi

(defmethod circumference ((object bicycle))
  (* pi (wheel-size object)))

;; pi is defined in Lisp already for us!

;; Let's suppose we find out that the efficiency value of the number
;; of rowers in a canoe is roughly logarithmic. This should probably be set
;; in the constructor/initializer.

;; Here's how to initialize your instance after Common Lisp gets done
;; constructing it:

(defmethod initialize-instance :after ((object canoe) &rest args)
  (setf (average-efficiency object)  (log (1+ (number-of-rowers object)))))

;; Then to construct an instance and check the average efficiency...

(average-efficiency (make-instance 'canoe :number-of-rowers 15))
; => 2.7725887




;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 8. Macros
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Macros let you extend the syntax of the language

;; Common Lisp doesn't come with a WHILE loop- let's add one.
;; If we obey our assembler instincts, we wind up with:

(defmacro while (condition &body body)
    "While `condition` is true, `body` is executed.

`condition` is tested prior to each execution of `body`"
    (let ((block-name (gensym)) (done (gensym)))
        `(tagbody
           ,block-name
           (unless ,condition
               (go ,done))
           (progn
           ,@body)
           (go ,block-name)
           ,done)))

;; Let's look at the high-level version of this:
```

```
(defmacro while (condition &body body)
    "While `condition` is true, `body` is executed.

`condition` is tested prior to each execution of `body`"
  `(loop while ,condition
         do
         (progn
            ,@body)))

;; However, with a modern compiler, this is not required; the LOOP
;; form compiles equally well and is easier to read.

;; Note that ``` is used, as well as `,` and `@`. ``` is a quote-type operator
;; known as quasiquote; it allows the use of `,` . `,` allows "unquoting"
;; variables. @ interpolates lists.

;; Gensym creates a unique symbol guaranteed to not exist elsewhere in
;; the system. This is because macros are expanded at compile time and
;; variables declared in the macro can collide with variables used in
;; regular code.

;; See Practical Common Lisp for more information on macros.
```

Further Reading

- Keep moving on to the Practical Common Lisp book.
- A Gentle Introduction to…

Extra Info

- CLiki
- common-lisp.net
- Awesome Common Lisp

Credits.

Lots of thanks to the Scheme people for rolling up a great starting point which could be easily moved to Common Lisp.

- Paul Khuong for some great reviewing.

Compojure

Getting Started with Compojure

Compojure is a DSL for *quickly* creating *performant* web applications in Clojure with minimal effort:

```
(ns myapp.core
  (:require [compojure.core :refer :all]
            [org.httpkit.server :refer [run-server]])) ; httpkit is a server
```

```
(defroutes myapp
  (GET "/" [] "Hello World"))

(defn -main []
  (run-server myapp {:port 5000}))
```

Step 1: Create a project with Leiningen:

```
lein new myapp
```

Step 2: Put the above code in `src/myapp/core.clj`

Step 3: Add some dependencies to `project.clj`:

```
[compojure "1.1.8"]
[http-kit "2.1.16"]
```

Step 4: Run:

```
lein run -m myapp.core
```

View at: http://localhost:5000/

Compojure apps will run on any ring-compatible server, but we recommend http-kit for its performance and massive concurrency.

Routes

In compojure, each route is an HTTP method paired with a URL-matching pattern, an argument list, and a body.

```
(defroutes myapp
  (GET "/" [] "Show something")
  (POST "/" [] "Create something")
  (PUT "/" [] "Replace something")
  (PATCH "/" [] "Modify Something")
  (DELETE "/" [] "Annihilate something")
  (OPTIONS "/" [] "Appease something")
  (HEAD "/" [] "Preview something"))
```

Compojure route definitions are just functions which accept request maps and return response maps:

```
(myapp {:uri "/" :request-method :post})
; => {:status 200
;     :headers {"Content-Type" "text/html; charset=utf-8}
;     :body "Create Something"}
```

The body may be a function, which must accept the request as a parameter:

```
(defroutes myapp
  (GET "/" [] (fn [req] "Do something with req")))
```

Or, you can just use the request directly:

```
(defroutes myapp
  (GET "/" req "Do something with req"))
```

Route patterns may include named parameters:

```
(defroutes myapp
  (GET "/hello/:name" [name] (str "Hello " name)))
```

You can adjust what each parameter matches by supplying a regex:

```
(defroutes myapp
  (GET ["/file/:name.:ext" :name #".*", :ext #".*"] [name ext]
    (str "File: " name ext)))
```

Middleware

Clojure uses Ring for routing. Handlers are just functions that accept a request map and return a response map (Compojure will turn strings into 200 responses for you).

You can easily write middleware that wraps all or part of your application to modify requests or responses:

```
(defroutes myapp
  (GET "/" req (str "Hello World v" (:app-version req))))

(defn wrap-version [handler]
  (fn [request]
    (handler (assoc request :app-version "1.0.1"))))

(defn -main []
  (run-server (wrap-version myapp) {:port 5000}))
```

Ring-Defaults provides some handy middlewares for sites and apis, so add it to your dependencies:

```
[ring/ring-defaults "0.1.1"]
```

Then, you can import it in your ns:

```
(ns myapp.core
  (:require [compojure.core :refer :all]
            [ring.middleware.defaults :refer :all]
            [org.httpkit.server :refer [run-server]]))
```

And use `wrap-defaults` to add the `site-defaults` middleware to your app:

```
(defn -main []
  (run-server (wrap-defaults myapp site-defaults) {:port 5000}))
```

Now, your handlers may utilize query parameters:

```
(defroutes myapp
  (GET "/posts" req
    (let [title (get (:params req) :title)
          author (get (:params req) :author)]
      (str "Title: " title ", Author: " author))))
```

Or, for POST and PUT requests, form parameters as well

```
(defroutes myapp
  (POST "/posts" req
    (let [title (get (:params req) :title)
          author (get (:params req) :author)]
      (str "Title: " title ", Author: " author))))
```

Return values

The return value of a route block determines the response body passed on to the HTTP client, or at least the next middleware in the ring stack. Most commonly, this is a string, as in the above examples. But, you may also return a response map:

```
(defroutes myapp
  (GET "/" []
    {:status 200 :body "Hello World"})
  (GET "/is-403" []
    {:status 403 :body ""})
  (GET "/is-json" []
    {:status 200 :headers {"Content-Type" "application/json"} :body "{}"}))
```

Static Files

To serve up static files, use `compojure.route.resources`. Resources will be served from your project's `resources/` folder.

```
(require '[compojure.route :as route])

(defroutes myapp
  (GET "/")
  (route/resources "/")) ; Serve static resources at the root path

(myapp {:uri "/js/script.js" :request-method :get})
; => Contents of resources/public/js/script.js
```

Views / Templates

To use templating with Compojure, you'll need a template library. Here are a few:

Stencil

Stencil is a Mustache template library:

```
(require '[stencil.core :refer [render-string]])

(defroutes myapp
  (GET "/hello/:name" [name]
    (render-string "Hello {{name}}" {:name name})))
```

You can easily read in templates from your resources directory. Here's a helper function

```
(require 'clojure.java.io)

(defn read-template [filename]
  (slurp (clojure.java.io/resource filename)))

(defroutes myapp
  (GET "/hello/:name" [name]
    (render-string (read-template "templates/hello.html") {:name name})))
```

Selmer

Selmer is a Django and Jinja2-inspired templating language:

```
(require '[selmer.parser :refer [render-file]])

(defroutes myapp
  (GET "/hello/:name" [name]
    (render-file "templates/hello.html" {:name name})))
```

Hiccup

Hiccup is a library for representing HTML as Clojure code

```
(require '[hiccup.core :as hiccup])

(defroutes myapp
  (GET "/hello/:name" [name]
    (hiccup/html
      [:html
        [:body
          [:h1 {:class "title"}
            (str "Hello " name)]]])))
```

Markdown

Markdown-clj is a Markdown implementation.

```
(require '[markdown.core :refer [md-to-html-string]])

(defroutes myapp
  (GET "/hello/:name" [name]
    (md-to-html-string "## Hello, world")))
```

Further reading:

- Official Compojure Documentation
- Clojure for the Brave and True

Csharp

C# is an elegant and type-safe object-oriented language that enables developers to build a variety of secure and robust applications that run on the .NET Framework.

Read more here.

```
// Single-line comments start with //
/*
Multi-line comments look like this
*/
/// <summary>
/// This is an XML documentation comment which can be used to generate external
/// documentation or provide context help within an IDE
/// </summary>
//public void MethodOrClassOrOtherWithParsableHelp() {}

// Specify the namespaces this source code will be using
// The namespaces below are all part of the standard .NET Framework Class Libary
using System;
using System.Collections.Generic;
using System.Dynamic;
using System.Linq;
using System.Net;
using System.Threading.Tasks;
using System.IO;
```

```
// But this one is not:
using System.Data.Entity;
// In order to be able to use it, you need to add a dll reference
// This can be done with the NuGet package manager: `Install-Package EntityFramework`

// Namespaces define scope to organize code into "packages" or "modules"
// Using this code from another source file: using Learning.CSharp;
namespace Learning.CSharp
{
    // Each .cs file should at least contain a class with the same name as the file.
    // You're allowed to do otherwise, but shouldn't for sanity.
    public class LearnCSharp
    {
        // BASIC SYNTAX - skip to INTERESTING FEATURES if you have used Java or C++ before
        public static void Syntax()
        {
            // Use Console.WriteLine to print lines
            Console.WriteLine("Hello World");
            Console.WriteLine(
                "Integer: " + 10 +
                " Double: " + 3.14 +
                " Boolean: " + true);

            // To print without a new line, use Console.Write
            Console.Write("Hello ");
            Console.Write("World");

            ///////////////////////////////////////////////////
            // Types & Variables
            //
            // Declare a variable using <type> <name>
            ///////////////////////////////////////////////////

            // Sbyte - Signed 8-bit integer
            // (-128 <= sbyte <= 127)
            sbyte fooSbyte = 100;

            // Byte - Unsigned 8-bit integer
            // (0 <= byte <= 255)
            byte fooByte = 100;

            // Short - 16-bit integer
            // Signed - (-32,768 <= short <= 32,767)
            // Unsigned - (0 <= ushort <= 65,535)
            short fooShort = 10000;
            ushort fooUshort = 10000;

            // Integer - 32-bit integer
            int fooInt = 1; // (-2,147,483,648 <= int <= 2,147,483,647)
            uint fooUint = 1; // (0 <= uint <= 4,294,967,295)

            // Long - 64-bit integer
        long fooLong = 100000L; // (-9,223,372,036,854,775,808 <= long <= 9,223,372,036,854,775,807)
            ulong fooUlong = 100000L; // (0 <= ulong <= 18,446,744,073,709,551,615)
            // Numbers default to being int or uint depending on size.
            // L is used to denote that this variable value is of type long or ulong

            // Double - Double-precision 64-bit IEEE 754 Floating Point
            double fooDouble = 123.4; // Precision: 15-16 digits

            // Float - Single-precision 32-bit IEEE 754 Floating Point
            float fooFloat = 234.5f; // Precision: 7 digits
            // f is used to denote that this variable value is of type float

        // Decimal - a 128-bits data type, with more precision than other floating-point types,
            // suited for financial and monetary calculations
            decimal fooDecimal = 150.3m;

            // Boolean - true & false
            bool fooBoolean = true; // or false

            // Char - A single 16-bit Unicode character
            char fooChar = 'A';

            // Strings -- unlike the previous base types which are all value types,
            // a string is a reference type. That is, you can set it to null
            string fooString = "\"escape\" quotes and add \n (new lines) and \t (tabs)";
            Console.WriteLine(fooString);

            // You can access each character of the string with an indexer:
            char charFromString = fooString[1]; // => 'e'
            // Strings are immutable: you can't do fooString[1] = 'X';

            // Compare strings with current culture, ignoring case
            string.Compare(fooString, "x", StringComparison.CurrentCultureIgnoreCase);

            // Formatting, based on sprintf
            string fooFs = string.Format("Check Check, {0} {1}, {0} {1:0.0}", 1, 2);

            // Dates & Formatting
            DateTime fooDate = DateTime.Now;
            Console.WriteLine(fooDate.ToString("hh:mm, dd MMM yyyy"));

            // You can split a string over two lines with the @ symbol. To escape " use ""
            string bazString = @"Here's some stuff
on a new line! ""Wow!"", the masses cried";

            // Use const or read-only to make a variable immutable
            // const values are calculated at compile time
            const int HoursWorkPerWeek = 9001;

            ///////////////////////////////////////////////////
            // Data Structures
            ///////////////////////////////////////////////////

            // Arrays - zero indexed
            // The array size must be decided upon declaration
            // The format for declaring an array is follows:
```

```
// <datatype>[] <var name> = new <datatype>[<array size>];
int[] intArray = new int[10];

// Another way to declare & initialize an array
int[] y = { 9000, 1000, 1337 };

// Indexing an array - Accessing an element
Console.WriteLine("intArray @ 0: " + intArray[0]);
// Arrays are mutable.
intArray[1] = 1;

// Lists
// Lists are used more frequently than arrays as they are more flexible
// The format for declaring a list is follows:
// List<datatype> <var name> = new List<datatype>();
List<int> intList = new List<int>();
List<string> stringList = new List<string>();
List<int> z = new List<int> { 9000, 1000, 1337 }; // initialize
// The <> are for generics - Check out the cool stuff section

// Lists don't default to a value;
// A value must be added before accessing the index
intList.Add(1);
Console.WriteLine("intList @ 0: " + intList[0]);

// Others data structures to check out:
// Stack/Queue
// Dictionary (an implementation of a hash map)
// HashSet
// Read-only Collections
// Tuple (.Net 4+)

///////////////////////////////////////
// Operators
///////////////////////////////////////
Console.WriteLine("\n->Operators");

int i1 = 1, i2 = 2; // Shorthand for multiple declarations

// Arithmetic is straightforward
Console.WriteLine(i1 + i2 - i1 * 3 / 7); // => 3

// Modulo
Console.WriteLine("11%3 = " + (11 % 3)); // => 2

// Comparison operators
Console.WriteLine("3 == 2? " + (3 == 2)); // => false
Console.WriteLine("3 != 2? " + (3 != 2)); // => true
Console.WriteLine("3 > 2? " + (3 > 2)); // => true
Console.WriteLine("3 < 2? " + (3 < 2)); // => false
Console.WriteLine("2 <= 2? " + (2 <= 2)); // => true
Console.WriteLine("2 >= 2? " + (2 >= 2)); // => true

// Bitwise operators!
```

```
/*
~       Unary bitwise complement
<<      Signed left shift
>>      Signed right shift
&       Bitwise AND
^       Bitwise exclusive OR
|       Bitwise inclusive OR
*/

// Incrementations
int i = 0;
Console.WriteLine("\n->Inc/Dec-rementation");
Console.WriteLine(i++); //i = 1. Post-Incrementation
Console.WriteLine(++i); //i = 2. Pre-Incrementation
Console.WriteLine(i--); //i = 1. Post-Decrementation
Console.WriteLine(--i); //i = 0. Pre-Decrementation

///////////////////////////////////////
// Control Structures
///////////////////////////////////////
Console.WriteLine("\n->Control Structures");

// If statements are c-like
int j = 10;
if (j == 10)
{
    Console.WriteLine("I get printed");
}
else if (j > 10)
{
    Console.WriteLine("I don't");
}
else
{
    Console.WriteLine("I also don't");
}

// Ternary operators
// A simple if/else can be written as follows
// <condition> ? <true> : <false>
int toCompare = 17;
string isTrue = toCompare == 17 ? "True" : "False";

// While loop
int fooWhile = 0;
while (fooWhile < 100)
{
    //Iterated 100 times, fooWhile 0->99
    fooWhile++;
}

// Do While Loop
int fooDoWhile = 0;
do
```

```
        {
            // Start iteration 100 times, fooDoWhile 0->99
            if (false)
                continue; // skip the current iteration

            fooDoWhile++;

            if (fooDoWhile == 50)
                break; // breaks from the loop completely

        } while (fooDoWhile < 100);

        //for loop structure => for(<start_statement>; <conditional>; <step>)
        for (int fooFor = 0; fooFor < 10; fooFor++)
        {
            //Iterated 10 times, fooFor 0->9
        }

        // For Each Loop
    // foreach loop structure => foreach(<iteratorType> <iteratorName> in <enumerable>)
    // The foreach loop loops over any object implementing IEnumerable or IEnumerable<T>
        // All the collection types (Array, List, Dictionary...) in the .Net framework
        // implement one or both of these interfaces.
    // (The ToCharArray() could be removed, because a string also implements IEnumerable)
        foreach (char character in "Hello World".ToCharArray())
        {
            //Iterated over all the characters in the string
        }

        // Switch Case
        // A switch works with the byte, short, char, and int data types.
        // It also works with enumerated types (discussed in Enum Types),
        // the String class, and a few special classes that wrap
        // primitive types: Character, Byte, Short, and Integer.
        int month = 3;
        string monthString;
        switch (month)
        {
            case 1:
                monthString = "January";
                break;
            case 2:
                monthString = "February";
                break;
            case 3:
                monthString = "March";
                break;
            // You can assign more than one case to an action
            // But you can't add an action without a break before another case
            // (if you want to do this, you would have to explicitly add a goto case x
            case 6:
            case 7:
            case 8:
                monthString = "Summer time!!";
                break;
            default:
                monthString = "Some other month";
                break;
        }

        ///////////////////////////////////////
        // Converting Data Types And Typecasting
        ///////////////////////////////////////

        // Converting data

        // Convert String To Integer
        // this will throw a FormatException on failure
        int.Parse("123");//returns an integer version of "123"

        // try parse will default to type default on failure
        // in this case: 0
        int tryInt;
        if (int.TryParse("123", out tryInt)) // Function is boolean
            Console.WriteLine(tryInt);       // 123

        // Convert Integer To String
        // Convert class has a number of methods to facilitate conversions
        Convert.ToString(123);
        // or
        tryInt.ToString();

        // Casting
        // Cast decimal 15 to a int
        // and then implicitly cast to long
        long x = (int) 15M;
    }

    ///////////////////////////////////////
    // CLASSES - see definitions at end of file
    ///////////////////////////////////////
    public static void Classes()
    {
        // See Declaration of objects at end of file

        // Use new to instantiate a class
        Bicycle trek = new Bicycle();

        // Call object methods
        trek.SpeedUp(3); // You should always use setter and getter methods
        trek.Cadence = 100;

        // ToString is a convention to display the value of this Object.
        Console.WriteLine("trek info: " + trek.Info());

        // Instantiate a new Penny Farthing
        PennyFarthing funbike = new PennyFarthing(1, 10);
        Console.WriteLine("funbike info: " + funbike.Info());
        Console.Read();
    } // End main method

    // CONSOLE ENTRY A console application must have a main method as an entry point
    public static void Main(string[] args)
    {
        OtherInterestingFeatures();
    }

    //
    // INTERESTING FEATURES
    //

    // DEFAULT METHOD SIGNATURES

    public // Visibility
    static // Allows for direct call on class without object
    int // Return Type,
    MethodSignatures(
        int maxCount, // First variable, expects an int
        int count = 0, // will default the value to 0 if not passed in
        int another = 3,
        params string[] otherParams // captures all other parameters passed to method
    )
    {
        return -1;
    }

    // Methods can have the same name, as long as the signature is unique
    // A method that differs only in return type is not unique
    public static void MethodSignatures(
        ref int maxCount, // Pass by reference
        out int count)
    {
    //the argument passed in as 'count' will hold the value of 15 outside of this function
        count = 15; // out param must be assigned before control leaves the method
    }

    // GENERICS
    // The classes for TKey and TValue is specified by the user calling this function.
    // This method emulates the SetDefault of Python
    public static TValue SetDefault<TKey, TValue>(
        IDictionary<TKey, TValue> dictionary,
        TKey key,
        TValue defaultItem)
    {
        TValue result;
        if (!dictionary.TryGetValue(key, out result))
            return dictionary[key] = defaultItem;
        return result;
    }

    // You can narrow down the objects that are passed in
```

```
public static void IterateAndPrint<T>(T toPrint) where T: IEnumerable<int>
{
    // We can iterate, since T is a IEnumerable
    foreach (var item in toPrint)
        // Item is an int
        Console.WriteLine(item.ToString());
}

// YIELD
// Usage of the "yield" keyword indicates that the method it appears in is an Iterator
// (this means you can use it in a foreach loop)
public static IEnumerable<int> YieldCounter(int limit = 10)
{
    for (var i = 0; i < limit; i++)
        yield return i;
}

// which you would call like this :
public static void PrintYieldCounterToConsole()
{
    foreach (var counter in YieldCounter())
        Console.WriteLine(counter);
}

// you can use more than one "yield return" in a method
public static IEnumerable<int> ManyYieldCounter()
{
    yield return 0;
    yield return 1;
    yield return 2;
    yield return 3;
}

// you can also use "yield break" to stop the Iterator
// this method would only return half of the values from 0 to limit.
public static IEnumerable<int> YieldCounterWithBreak(int limit = 10)
{
    for (var i = 0; i < limit; i++)
    {
        if (i > limit/2) yield break;
        yield return i;
    }
}

public static void OtherInterestingFeatures()
{
    // OPTIONAL PARAMETERS
    MethodSignatures(3, 1, 3, "Some", "Extra", "Strings");
MethodSignatures(3, another: 3); // explicitly set a parameter, skipping optional ones

    // BY REF AND OUT PARAMETERS
    int maxCount = 0, count; // ref params must have value
    MethodSignatures(ref maxCount, out count);
```

```
        // EXTENSION METHODS
        int i = 3;
        i.Print(); // Defined below

        // NULLABLE TYPES - great for database interaction / return values
        // any value type (i.e. not a class) can be made nullable by suffixing a ?
        // <type>? <var name> = <value>
        int? nullable = null; // short hand for Nullable<int>
        Console.WriteLine("Nullable variable: " + nullable);
        bool hasValue = nullable.HasValue; // true if not null

        // ?? is syntactic sugar for specifying default value (coalesce)
        // in case variable is null
        int notNullable = nullable ?? 0; // 0

       // ?. is an operator for null-propagation - a shorthand way of checking for null
        nullable?.Print(); // Use the Print() extension method if nullable isn't null

      // IMPLICITLY TYPED VARIABLES - you can let the compiler work out what the type is:
       var magic = "magic is a string, at compile time, so you still get type safety";
        // magic = 9; will not work as magic is a string, not an int

        // GENERICS
        //
        var phonebook = new Dictionary<string, string>() {
            {"Sarah", "212 555 5555"} // Add some entries to the phone book
        };

        // Calling SETDEFAULT defined as a generic above
    Console.WriteLine(SetDefault<string,string>(phonebook, "Shaun", "No Phone")); // No Phone
        // nb, you don't need to specify the TKey and TValue since they can be
        // derived implicitly
        Console.WriteLine(SetDefault(phonebook, "Sarah", "No Phone")); // 212 555 5555

        // LAMBDA EXPRESSIONS - allow you to write code in line
        Func<int, int> square = (x) => x * x; // Last T item is the return value
        Console.WriteLine(square(3)); // 9

        // ERROR HANDLING - coping with an uncertain world
        try
        {
            var funBike = PennyFarthing.CreateWithGears(6);

            // will no longer execute because CreateWithGears throws an exception
            string some = "";
            if (true) some = null;
            some.ToLower(); // throws a NullReferenceException
        }
        catch (NotSupportedException)
        {
            Console.WriteLine("Not so much fun now!");
        }
        catch (Exception ex) // catch all other exceptions
        {
```

```
        throw new ApplicationException("It hit the fan", ex);
        // throw; // A rethrow that preserves the callstack
    }
    // catch { } // catch-all without capturing the Exception
    finally
    {
        // executes after try or catch
    }

   // DISPOSABLE RESOURCES MANAGEMENT - let you handle unmanaged resources easily.
// Most of objects that access unmanaged resources (file handle, device contexts, etc.)
    // implement the IDisposable interface. The using statement takes care of
    // cleaning those IDisposable objects for you.
    using (StreamWriter writer = new StreamWriter("log.txt"))
    {
        writer.WriteLine("Nothing suspicious here");
        // At the end of scope, resources will be released.
        // Even if an exception is thrown.
    }

    // PARALLEL FRAMEWORK
// http://blogs.msdn.com/b/csharpfaq/archive/2010/06/01/parallel-programming-in-net-framework-4-
    var websites = new string[] {
        "http://www.google.com", "http://www.reddit.com",
        "http://www.shaunmccarthy.com"
    };
    var responses = new Dictionary<string, string>();

    // Will spin up separate threads for each request, and join on them
    // before going to the next step!
    Parallel.ForEach(websites,
        new ParallelOptions() {MaxDegreeOfParallelism = 3}, // max of 3 threads
        website =>
    {
        // Do something that takes a long time on the file
        using (var r = WebRequest.Create(new Uri(website)).GetResponse())
        {
            responses[website] = r.ContentType;
        }
    });

    // This won't happen till after all requests have been completed
    foreach (var key in responses.Keys)
        Console.WriteLine("{0}:{1}", key, responses[key]);

    // DYNAMIC OBJECTS (great for working with other languages)
    dynamic student = new ExpandoObject();
    student.FirstName = "First Name"; // No need to define class first!

    // You can even add methods (returns a string, and takes in a string)
    student.Introduce = new Func<string, string>(
   (introduceTo) => string.Format("Hey {0}, this is {1}", student.FirstName, introduceTo));
    Console.WriteLine(student.Introduce("Beth"));
```

```
        // IQUERYABLE<T> - almost all collections implement this, which gives you a lot of
        // very useful Map / Filter / Reduce style methods
        var bikes = new List<Bicycle>();
        bikes.Sort(); // Sorts the array
        bikes.Sort((b1, b2) => b1.Wheels.CompareTo(b2.Wheels)); // Sorts based on wheels
        var result = bikes
            .Where(b => b.Wheels > 3) // Filters - chainable (returns IQueryable of previous type)
            .Where(b => b.IsBroken && b.HasTassles)
            .Select(b => b.ToString()); // Map - we only this selects, so result is a IQueryable<string>

        var sum = bikes.Sum(b => b.Wheels); // Reduce - sums all the wheels in the collection

        // Create a list of IMPLICIT objects based on some parameters of the bike
        var bikeSummaries = bikes.Select(b=>new { Name = b.Name, IsAwesome = !b.IsBroken && b.HasTassles });
        // Hard to show here, but you get type ahead completion since the compiler can implicitly work
        // out the types above!
        foreach (var bikeSummary in bikeSummaries.Where(b => b.IsAwesome))
            Console.WriteLine(bikeSummary.Name);

        // ASPARALLEL
        // And this is where things get wicked - combines linq and parallel operations
        var threeWheelers = bikes.AsParallel().Where(b => b.Wheels == 3).Select(b => b.Name);
        // this will happen in parallel! Threads will automagically be spun up and the
        // results divvied amongst them! Amazing for large datasets when you have lots of
        // cores

        // LINQ - maps a store to IQueryable<T> objects, with delayed execution
        // e.g. LinqToSql - maps to a database, LinqToXml maps to an xml document
        var db = new BikeRepository();

        // execution is delayed, which is great when querying a database
        var filter = db.Bikes.Where(b => b.HasTassles); // no query run
        if (42 > 6) // You can keep adding filters, even conditionally - great for "advanced search" function
            filter = filter.Where(b => b.IsBroken); // no query run

        var query = filter
            .OrderBy(b => b.Wheels)
            .ThenBy(b => b.Name)
            .Select(b => b.Name); // still no query run

        // Now the query runs, but opens a reader, so only populates are you iterate through
        foreach (string bike in query)
            Console.WriteLine(result);



    }

} // End LearnCSharp class

// You can include other classes in a .cs file

public static class Extensions
{
    // EXTENSION METHODS
    public static void Print(this object obj)
    {
        Console.WriteLine(obj.ToString());
    }
}

// Class Declaration Syntax:
// <public/private/protected/internal> class <class name>{
//    //data fields, constructors, functions all inside.
//    //functions are called as methods in Java.
// }

public class Bicycle
{
    // Bicycle's Fields/Variables
    public int Cadence // Public: Can be accessed from anywhere
    {
        get // get - define a method to retrieve the property
        {
            return _cadence;
        }
        set // set - define a method to set a property
        {
            _cadence = value; // Value is the value passed in to the setter
        }
    }
    private int _cadence;

    protected virtual int Gear // Protected: Accessible from the class and subclasses
    {
        get; // creates an auto property so you don't need a member field
        set;
    }

    internal int Wheels // Internal: Accessible from within the assembly
    {
        get;
        private set; // You can set modifiers on the get/set methods
    }

  int _speed; // Everything is private by default: Only accessible from within this class.
                // can also use keyword private
    public string Name { get; set; }

    // Enum is a value type that consists of a set of named constants
   // It is really just mapping a name to a value (an int, unless specified otherwise).
 // The approved types for an enum are byte, sbyte, short, ushort, int, uint, long, or ulong.
    // An enum can't contain the same value twice.
    public enum BikeBrand
    {
        AIST,
        BMC,
        Electra = 42, //you can explicitly set a value to a name
```

```
            Gitane // 43
        }
        // We defined this type inside a Bicycle class, so it is a nested type
        // Code outside of this class should reference this type as Bicycle.Brand

     public BikeBrand Brand; // After declaring an enum type, we can declare the field of this type

     // Decorate an enum with the FlagsAttribute to indicate that multiple values can be switched on
     [Flags] // Any class derived from Attribute can be used to decorate types, methods, parameters etc
        public enum BikeAccessories
        {
            None = 0,
            Bell = 1,
            MudGuards = 2, // need to set the values manually!
            Racks = 4,
            Lights = 8,
            FullPackage = Bell | MudGuards | Racks | Lights
        }

        // Usage: aBike.Accessories.HasFlag(Bicycle.BikeAccessories.Bell)
     // Before .NET 4: (aBike.Accessories & Bicycle.BikeAccessories.Bell) == Bicycle.BikeAccessories.Bell
        public BikeAccessories Accessories { get; set; }

        // Static members belong to the type itself rather then specific object.
        // You can access them without a reference to any object:
        // Console.WriteLine("Bicycles created: " + Bicycle.bicyclesCreated);
        public static int BicyclesCreated { get; set; }

        // readonly values are set at run time
        // they can only be assigned upon declaration or in a constructor
        readonly bool _hasCardsInSpokes = false; // read-only private

        // Constructors are a way of creating classes
        // This is a default constructor
        public Bicycle()
        {
            this.Gear = 1; // you can access members of the object with the keyword this
            Cadence = 50;  // but you don't always need it
            _speed = 5;
            Name = "Bontrager";
            Brand = BikeBrand.AIST;
            BicyclesCreated++;
        }

        // This is a specified constructor (it contains arguments)
        public Bicycle(int startCadence, int startSpeed, int startGear,
                       string name, bool hasCardsInSpokes, BikeBrand brand)
            : base() // calls base first
        {
            Gear = startGear;
            Cadence = startCadence;
            _speed = startSpeed;
            Name = name;
            _hasCardsInSpokes = hasCardsInSpokes;
```

```
    Brand = brand;
}

// Constructors can be chained
public Bicycle(int startCadence, int startSpeed, BikeBrand brand) :
    this(startCadence, startSpeed, 0, "big wheels", true, brand)
{
}

// Function Syntax:
// <public/private/protected> <return type> <function name>(<args>)

// classes can implement getters and setters for their fields
// or they can implement properties (this is the preferred way in C#)

// Method parameters can have default values.
// In this case, methods can be called with these parameters omitted
public void SpeedUp(int increment = 1)
{
    _speed += increment;
}

public void SlowDown(int decrement = 1)
{
    _speed -= decrement;
}

// properties get/set values
// when only data needs to be accessed, consider using properties.
// properties may have either get or set, or both
private bool _hasTassles; // private variable
public bool HasTassles // public accessor
{
    get { return _hasTassles; }
    set { _hasTassles = value; }
}

// You can also define an automatic property in one line
// this syntax will create a backing field automatically.
// You can set an access modifier on either the getter or the setter (or both)
// to restrict its access:
public bool IsBroken { get; private set; }

// Properties can be auto-implemented
public int FrameSize
{
    get;
    // you are able to specify access modifiers for either get or set
    // this means only Bicycle class can call set on Framesize
    private set;
}

// It's also possible to define custom Indexers on objects.
// All though this is not entirely useful in this example, you
```

```
    // could do bicycle[0] which returns "chris" to get the first passenger or
    // bicycle[1] = "lisa" to set the passenger. (of this apparent quattrocycle)
    private string[] passengers = { "chris", "phil", "darren", "regina" };

    public string this[int i]
    {
        get {
            return passengers[i];
        }

        set {
            passengers[i] = value;
        }
    }

    //Method to display the attribute values of this Object.
    public virtual string Info()
    {
        return "Gear: " + Gear +
                " Cadence: " + Cadence +
                " Speed: " + _speed +
                " Name: " + Name +
                " Cards in Spokes: " + (_hasCardsInSpokes ? "yes" : "no") +
                "\n------------------------------\n"
                ;
    }

    // Methods can also be static. It can be useful for helper methods
    public static bool DidWeCreateEnoughBycles()
    {
        // Within a static method, we only can reference static class members
        return BicyclesCreated > 9000;
 } // If your class only needs static members, consider marking the class itself as static.


} // end class Bicycle

// PennyFarthing is a subclass of Bicycle
class PennyFarthing : Bicycle
{
    // (Penny Farthings are those bicycles with the big front wheel.
    // They have no gears.)

    // calling parent constructor
    public PennyFarthing(int startCadence, int startSpeed) :
        base(startCadence, startSpeed, 0, "PennyFarthing", true, BikeBrand.Electra)
    {
    }

    protected override int Gear
    {
        get
        {
            return 0;
        }
        set
        {
       throw new InvalidOperationException("You can't change gears on a PennyFarthing");
        }
    }

    public static PennyFarthing CreateWithGears(int gears)
    {
        var penny = new PennyFarthing(1, 1);
        penny.Gear = gears; // Oops, can't do this!
        return penny;
    }

    public override string Info()
    {
        string result = "PennyFarthing bicycle ";
        result += base.ToString(); // Calling the base version of the method
        return result;
    }
}

// Interfaces only contain signatures of the members, without the implementation.
interface IJumpable
{
    void Jump(int meters); // all interface members are implicitly public
}

interface IBreakable
{
   bool Broken { get; } // interfaces can contain properties as well as methods & events
}

// Class can inherit only one other class, but can implement any amount of interfaces, however
// the base class name must be the first in the list and all interfaces follow
class MountainBike : Bicycle, IJumpable, IBreakable
{
    int damage = 0;

    public void Jump(int meters)
    {
        damage += meters;
    }

    public bool Broken
    {
        get
        {
            return damage > 100;
        }
    }
}

/// <summary>
```

```
    /// Used to connect to DB for LinqToSql example.
    /// EntityFramework Code First is awesome (similar to Ruby's ActiveRecord, but bidirectional)
    /// http://msdn.microsoft.com/en-us/data/jj193542.aspx
    /// </summary>
    public class BikeRepository : DbContext
    {
        public BikeRepository()
            : base()
        {
        }

        public DbSet<Bicycle> Bikes { get; set; }
    }

    // Classes can be split across multiple .cs files
    // A1.cs
    public partial class A
    {
        public static void A1()
        {
            Console.WriteLine("Method A1 in class A");
        }
    }

    // A2.cs
    public partial class A
    {
        public static void A2()
        {
            Console.WriteLine("Method A2 in class A");
        }
    }

    // Program using the partial class "A"
    public class Program
    {
        static void Main()
        {
            A.A1();
            A.A2();
        }
    }
} // End Namespace
```

Topics Not Covered

- Attributes
- async/await, pragma directives
- Web Development
    - ASP.NET MVC & WebApi (new)
    - ASP.NET Web Forms (old)
    - WebMatrix (tool)
- Desktop Development

    - Windows Presentation Foundation (WPF) (new)
    - Winforms (old)

Further Reading

- DotNetPerls
- C# in Depth
- Programming C#
- LINQ
- MSDN Library
- ASP.NET MVC Tutorials
- ASP.NET Web Matrix Tutorials
- ASP.NET Web Forms Tutorials
- Windows Forms Programming in C#
- C# Coding Conventions

Css

Web pages are built with HTML, which specifies the content of a page. CSS (Cascading Style Sheets) is a separate language which specifies a page's **appearance**.

CSS code is made of static *rules*. Each rule takes one or more *selectors* and gives specific *values* to a number of visual *properties*. Those properties are then applied to the page elements indicated by the selectors.

This guide has been written with CSS 2 in mind, which is extended by the new features of CSS 3.

NOTE: Because CSS produces visual results, in order to learn it, you need to try everything in a CSS playground like dabblet. The main focus of this article is on the syntax and some general tips.

Syntax

```
/* comments appear inside slash-asterisk, just like this line!
   there are no "one-line comments"; this is the only comment style */

/* ####################
   ## SELECTORS
   #################### */

/* the selector is used to target an element on a page. */
selector { property: value; /* more properties...*/ }

/*
Here is an example element:

<div class='class1 class2' id='anID' attr='value' otherAttr='en-us foo bar' />
*/

/* You can target it using one of its CSS classes */
.class1 { }

/* or both classes! */
.class1.class2 { }
```

```
/* or its name */
div { }

/* or its id */
#anID { }

/* or using the fact that it has an attribute! */
[attr] { font-size:smaller; }

/* or that the attribute has a specific value */
[attr='value'] { font-size:smaller; }

/* starts with a value (CSS 3) */
[attr^='val'] { font-size:smaller; }

/* or ends with a value (CSS 3) */
[attr$='ue'] { font-size:smaller; }

/* or contains a value in a space-separated list */
[otherAttr~='foo'] { }
[otherAttr~='bar'] { }

/* or contains a value in a dash-separated list, ie, "-" (U+002D) */
[otherAttr|='en'] { font-size:smaller; }


/* You can combine different selectors to create a more focused selector. Don't
   put spaces between them. */
div.some-class[attr$='ue'] { }

/* You can select an element which is a child of another element */
div.some-parent > .class-name { }

/* or a descendant of another element. Children are the direct descendants of
   their parent element, only one level down the tree. Descendants can be any
   level down the tree. */
div.some-parent .class-name { }

/* Warning: the same selector without a space has another meaning.
   Can you guess what? */
div.some-parent.class-name { }

/* You may also select an element based on its adjacent sibling */
.i-am-just-before + .this-element { }

/* or any sibling preceding it */
.i-am-any-element-before ~ .this-element { }

/* There are some selectors called pseudo classes that can be used to select an
   element only when it is in a particular state */

/* for example, when the cursor hovers over an element */
selector:hover { }
```

```
/* or a link has been visited */
selector:visited { }

/* or hasn't been visited */
selected:link { }

/* or an element is in focus */
selected:focus { }

/* any element that is the first child of its parent */
selector:first-child {}

/* any element that is the last child of its parent */
selector:last-child {}

/* Just like pseudo classes, pseudo elements allow you to style certain parts of a document  */

/* matches a virtual first child of the selected element */
selector::before {}

/* matches a virtual last child of the selected element */
selector::after {}

/* At appropriate places, an asterisk may be used as a wildcard to select every
   element */
* { } /* all elements */
.parent * { } /* all descendants */
.parent > * { } /* all children */

/* ####################
   ## PROPERTIES
   #################### */

selector {

    /* Units of length can be absolute or relative. */

    /* Relative units */
    width: 50%;       /* percentage of parent element width */
    font-size: 2em;   /* multiples of element's original font-size */
    font-size: 2rem;  /* or the root element's font-size */
    font-size: 2vw;   /* multiples of 1% of the viewport's width (CSS 3) */
    font-size: 2vh;   /* or its height */
    font-size: 2vmin; /* whichever of a vh or a vw is smaller */
    font-size: 2vmax; /* or greater */

    /* Absolute units */
    width: 200px;     /* pixels */
    font-size: 20pt;  /* points */
    width: 5cm;       /* centimeters */
    min-width: 50mm;  /* millimeters */
    max-width: 5in;   /* inches */

    /* Colors */
    color: #F6E;                 /* short hex format */
    color: #FF66EE;              /* long hex format */
    color: tomato;               /* a named color */
    color: rgb(255, 255, 255);   /* as rgb values */
    color: rgb(10%, 20%, 50%);   /* as rgb percentages */
    color: rgba(255, 0, 0, 0.3); /* as rgba values (CSS 3) Note: 0 <= a <= 1 */
    color: transparent;          /* equivalent to setting the alpha to 0 */
    color: hsl(0, 100%, 50%);    /* as hsl percentages (CSS 3) */
    color: hsla(0, 100%, 50%, 0.3); /* as hsl percentages with alpha */

    /* Images as backgrounds of elements */
    background-image: url(/img-path/img.jpg); /* quotes inside url() optional */

    /* Fonts */
    font-family: Arial;
    /* if the font family name has a space, it must be quoted */
    font-family: "Courier New";
    /* if the first one is not found, the browser uses the next, and so on */
    font-family: "Courier New", Trebuchet, Arial, sans-serif;
}
```

Usage

Save a CSS stylesheet with the extension .css.

```
<!-- You need to include the css file in your page's <head>. This is the
     recommended method. Refer to http://stackoverflow.com/questions/8284365 -->
<link rel='stylesheet' type='text/css' href='path/to/style.css' />

<!-- You can also include some CSS inline in your markup. -->
<style>
   a { color: purple; }
</style>

<!-- Or directly set CSS properties on the element. -->
<div style="border: 1px solid red;">
</div>
```

Precedence or Cascade

An element may be targeted by multiple selectors and may have a property set on it in more than once. In these cases, one of the rules takes precedence over others. Rules with a more specific selector take precedence over a less specific one, and a rule occuring later in the stylesheet overwrites a previous one.

This process is called cascading, hence the name Cascading Style Sheets.

Given the following CSS:

```
/* A */
p.class1[attr='value']

/* B */
p.class1 { }

/* C */
```

```
p.class2 { }

/* D */
p { }

/* E */
p { property: value !important; }
```

and the following markup:

```
<p style='/*F*/ property:value;' class='class1 class2' attr='value' />
```

The precedence of style is as follows. Remember, the precedence is for each **property**, not for the entire block.

- `E` has the highest precedence because of the keyword `!important`. It is recommended that you avoid its usage.
- `F` is next, because it is an inline style.
- `A` is next, because it is more "specific" than anything else. It has 3 specifiers: The name of the element `p`, its class `class1`, an attribute `attr='value'`.
- `C` is next, even though it has the same specificity as `B`. This is because it appears after `B`.
- `B` is next.
- `D` is the last one.

Compatibility

Most of the features in CSS 2 (and many in CSS 3) are available across all browsers and devices. But it's always good practice to check before using a new feature.

Resources

- CanIUse (Detailed compatibility info)
- Dabblet (CSS playground)
- Mozilla Developer Network's CSS documentation (Tutorials and reference)
- Codrops' CSS Reference (Reference)

Further Reading

- Understanding Style Precedence in CSS: Specificity, Inheritance, and the Cascade
- Selecting elements using attributes
- QuirksMode CSS
- Z-Index - The stacking context
- SASS and LESS for CSS pre-processing
- CSS-Tricks

D

```
// You know what's coming...
module hello;

import std.stdio;

// args is optional
```

```
void main(string[] args) {
    writeln("Hello, World!");
}
```

If you're like me and spend way too much time on the internet, odds are you've heard about D. The D programming language is a modern, general-purpose, multi-paradigm language with support for everything from low-level features to expressive high-level abstractions.

D is actively developed by a large group of super-smart people and is spearheaded by Walter Bright and Andrei Alexandrescu. With all that out of the way, let's look at some examples!

```
import std.stdio;

void main() {

    // Conditionals and loops work as expected.
    for(int i = 0; i < 10000; i++) {
        writeln(i);
    }

    // 'auto' can be used for inferring types.
    auto n = 1;

    // Numeric literals can use '_' as a digit separator for clarity.
    while(n < 10_000) {
        n += n;
    }

    do {
        n -= (n / 2);
    } while(n > 0);

    // For and while are nice, but in D-land we prefer 'foreach' loops.
    // The '..' creates a continuous range, including the first value
    // but excluding the last.
    foreach(n; 1..1_000_000) {
        if(n % 2 == 0)
            writeln(n);
    }

    // There's also 'foreach_reverse' when you want to loop backwards.
    foreach_reverse(n; 1..int.max) {
        if(n % 2 == 1) {
            writeln(n);
        } else {
            writeln("No!");
        }
    }
}
```

We can define new types with `struct`, `class`, `union`, and `enum`. Structs and unions are passed to functions by value (i.e. copied) and classes are passed by reference. Furthermore, we can use templates to parameterize all of these on both types and values!

```
// Here, 'T' is a type parameter. Think '<T>' from C++/C#/Java.
struct LinkedList(T) {
    T data = null;
    // Use '!' to instantiate a parameterized type. Again, think '<T>'.
    LinkedList!(T)* next;
}

class BinTree(T) {
    T data = null;

    // If there is only one template parameter, we can omit the parentheses.
    BinTree!T left;
    BinTree!T right;
}

enum Day {
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
}

// Use alias to create abbreviations for types.
alias IntList = LinkedList!int;
alias NumTree = BinTree!double;

// We can create function templates as well!
T max(T)(T a, T b) {
    if(a < b)
        return b;

    return a;
}

// Use the ref keyword to ensure pass by reference. That is, even if 'a' and 'b'
// are value types, they will always be passed by reference to 'swap()'.
void swap(T)(ref T a, ref T b) {
    auto temp = a;

    a = b;
    b = temp;
}

// With templates, we can also parameterize on values, not just types.
class Matrix(uint m, uint n, T = int) {
    T[m] rows;
    T[n] columns;
}

auto mat = new Matrix!(3, 3); // We've defaulted type 'T' to 'int'.
```

Speaking of classes, let's talk about properties for a second. A property is roughly a function that may act like an lvalue, so we can have the syntax of POD structures (`structure.x = 7`) with the semantics of getter and setter methods (`object.setX(7)`)!

```
// Consider a class parameterized on types 'T' & 'U'.
class MyClass(T, U) {
    T _data;
    U _other;
}

// And "getter" and "setter" methods like so:
class MyClass(T, U) {
    T _data;
    U _other;

    // Constructors are always named 'this'.
    this(T t, U u) {
        // This will call the setter methods below.
        data = t;
        other = u;
    }

    // getters
    @property T data() {
        return _data;
    }

    @property U other() {
        return _other;
    }

    // setters
    @property void data(T t) {
        _data = t;
    }

    @property void other(U u) {
        _other = u;
    }
}

// And we use them in this manner:
void main() {
    auto mc = new MyClass!(int, string)(7, "seven");

    // Import the 'stdio' module from the standard library for writing to
    // console (imports can be local to a scope).
    import std.stdio;

    // Call the getters to fetch the values.
    writefln("Earlier: data = %d, str = %s", mc.data, mc.other);

    // Call the setters to assign new values.
    mc.data = 8;
    mc.other = "eight";
    // Call the getters again to fetch the new values.
    writefln("Later: data = %d, str = %s", mc.data, mc.other);
}
```

With properties, we can add any amount of logic to our getter and setter methods, and keep the clean syntax of accessing members directly!

Other object-oriented goodies at our disposal include interfaces, abstract classes, and overriding methods. D does inheritance just like Java: Extend one class, implement as many interfaces as you please.

We've seen D's OOP facilities, but let's switch gears. D offers functional programming with first-class functions, `pure` functions, and immutable data. In addition, all of your favorite functional algorithms (map, filter, reduce and friends) can be found in the wonderful `std.algorithm` module!

```
import std.algorithm : map, filter, reduce;
import std.range : iota; // builds an end-exclusive range

void main() {
    // We want to print the sum of a list of squares of even ints
    // from 1 to 100. Easy!

    // Just pass lambda expressions as template parameters!
    // You can pass any function you like, but lambdas are convenient here.
    auto num = iota(1, 101).filter!(x => x % 2 == 0)
                           .map!(y => y ^^ 2)
                           .reduce!((a, b) => a + b);

    writeln(num);
}
```

Notice how we got to build a nice Haskellian pipeline to compute num? That's thanks to a D innovation know as Uniform Function Call Syntax (UFCS). With UFCS, we can choose whether to write a function call as a method or free function call! Walter wrote a nice article on this here. In short, you can call functions whose first parameter is of some type A on any expression of type A as a method.

I like parallelism. Anyone else like parallelism? Sure you do. Let's do some!

```
// Let's say we want to populate a large array with the square root of all
// consecutive integers starting from 1 (up until the size of the array), and we
// want to do this concurrently taking advantage of as many cores as we have
// available.

import std.stdio;
import std.parallelism : parallel;
import std.math : sqrt;

void main() {
    // Create your large array
    auto arr = new double[1_000_000];

    // Use an index, access every array element by reference (because we're
    // going to change each element) and just call parallel on the array!
    foreach(i, ref elem; parallel(arr)) {
        elem = sqrt(i + 1.0);
    }
}
```

Dart

Dart is a newcomer into the realm of programming languages. It borrows a lot from other mainstream languages, having as a goal not to deviate too much from its JavaScript sibling. Like JavaScript, Dart aims for great browser integration.

Dart's most controversial feature must be its Optional Typing.

```
import "dart:collection";
import "dart:math" as DM;

// Welcome to Learn Dart in 15 minutes. http://www.dartlang.org/
// This is an executable tutorial. You can run it with Dart or on
// the Try Dart! site if you copy/paste it there. http://try.dartlang.org/

// Function declaration and method declaration look the same. Function
// declarations can be nested. The declaration takes the form of
// name() {} or name() => singleLineExpression;
// The fat arrow function declaration has an implicit return for the result of
// the expression.
example1() {
  example1nested1() {
    example1nested2() => print("Example1 nested 1 nested 2");
    example1nested2();
  }
  example1nested1();
}

// Anonymous functions don't include a name.
example2() {
  example2nested1(fn) {
    fn();
  }
  example2nested1(() => print("Example2 nested 1"));
}

// When a function parameter is declared, the declaration can include the
// number of parameters the function takes by specifying the names of the
// parameters it takes.
example3() {
  example3nested1(fn(informSomething)) {
    fn("Example3 nested 1");
  }
  example3planB(fn) { // Or don't declare number of parameters.
    fn("Example3 plan B");
  }
  example3nested1((s) => print(s));
  example3planB((s) => print(s));
}

// Functions have closure access to outer variables.
var example4Something = "Example4 nested 1";
example4() {
  example4nested1(fn(informSomething)) {
    fn(example4Something);
  }
  example4nested1((s) => print(s));
}

// Class declaration with a sayIt method, which also has closure access
// to the outer variable as though it were a function as seen before.
var example5method = "Example5 sayIt";
class Example5Class {
  sayIt() {
    print(example5method);
  }
}
example5() {
  // Create an anonymous instance of the Example5Class and call the sayIt
  // method on it.
  new Example5Class().sayIt();
}

// Class declaration takes the form of class name { [classBody] }.
// Where classBody can include instance methods and variables, but also
// class methods and variables.
class Example6Class {
  var example6InstanceVariable = "Example6 instance variable";
  sayIt() {
    print(example6InstanceVariable);
  }
}
example6() {
  new Example6Class().sayIt();
}

// Class methods and variables are declared with "static" terms.
class Example7Class {
  static var example7ClassVariable = "Example7 class variable";
  static sayItFromClass() {
    print(example7ClassVariable);
  }
  sayItFromInstance() {
    print(example7ClassVariable);
  }
}
example7() {
  Example7Class.sayItFromClass();
  new Example7Class().sayItFromInstance();
}

// Literals are great, but there's a restriction for what literals can be
// outside of function/method bodies. Literals on the outer scope of class
// or outside of class have to be constant. Strings and numbers are constant
// by default. But arrays and maps are not. They can be made constant by
// declaring them "const".
var example8A = const ["Example8 const array"],
  example8M = const {"someKey": "Example8 const map"};
example8() {
  print(example8A[0]);
  print(example8M["someKey"]);
}

// Loops in Dart take the form of standard for () {} or while () {} loops,
// slightly more modern for (.. in ..) {}, or functional callbacks with many
// supported features, starting with forEach.
var example9A = const ["a", "b"];
example9() {
  for (var i = 0; i < example9A.length; i++) {
    print("Example9 for loop '${example9A[i]}'");
  }
  var i = 0;
  while (i < example9A.length) {
    print("Example9 while loop '${example9A[i]}'");
    i++;
  }
  for (var e in example9A) {
    print("Example9 for-in loop '${e}'");
  }
  example9A.forEach((e) => print("Example9 forEach loop '${e}'"));
}

// To loop over the characters of a string or to extract a substring.
var example10S = "ab";
example10() {
  for (var i = 0; i < example10S.length; i++) {
    print("Example10 String character loop '${example10S[i]}'");
  }
  for (var i = 0; i < example10S.length; i++) {
    print("Example10 substring loop '${example10S.substring(i, i + 1)}'");
  }
}

// Int and double are the two supported number formats.
example11() {
  var i = 1 + 320, d = 3.2 + 0.01;
  print("Example11 int ${i}");
  print("Example11 double ${d}");
}

// DateTime provides date/time arithmetic.
example12() {
  var now = new DateTime.now();
  print("Example12 now '${now}'");
  now = now.add(new Duration(days: 1));
  print("Example12 tomorrow '${now}'");
}

// Regular expressions are supported.
example13() {
  var s1 = "some string", s2 = "some", re = new RegExp("^s.+?g\$");
  match(s) {
    if (re.hasMatch(s)) {
      print("Example13 regexp matches '${s}'");
    } else {
      print("Example13 regexp doesn't match '${s}'");
    }
  }
  match(s1);
  match(s2);
}

// Boolean expressions need to resolve to either true or false, as no
// implicit conversions are supported.
example14() {
  var v = true;
  if (v) {
    print("Example14 value is true");
  }
  v = null;
  try {
    if (v) {
      // Never runs
    } else {
      // Never runs
    }
  } catch (e) {
    print("Example14 null value causes an exception: '${e}'");
  }
}

// try/catch/finally and throw are used for exception handling.
// throw takes any object as parameter;
example15() {
  try {
    try {
      throw "Some unexpected error.";
    } catch (e) {
      print("Example15 an exception: '${e}'");
      throw e; // Re-throw
    }
  } catch (e) {
    print("Example15 catch exception being re-thrown: '${e}'");
  } finally {
    print("Example15 Still run finally");
  }
}

// To be efficient when creating a long string dynamically, use
// StringBuffer. Or you could join a string array.
example16() {
  var sb = new StringBuffer(), a = ["a", "b", "c", "d"], e;
  for (e in a) { sb.write(e); }
  print("Example16 dynamic string created with "
    "StringBuffer '${sb.toString()}'");
  print("Example16 join string array '${a.join()}'");
}
```

```
// Strings can be concatenated by just having string literals next to
// one another with no further operator needed.
example17() {
  print("Example17 "
      "concatenate "
      "strings "
      "just like that");
}

// Strings have single-quote or double-quote for delimiters with no
// actual difference between the two. The given flexibility can be good
// to avoid the need to escape content that matches the delimiter being
// used. For example, double-quotes of HTML attributes if the string
// contains HTML content.
example18() {
  print('Example18 <a href="etc">'
      "Don't can't I'm Etc"
      '</a>');
}

// Strings with triple single-quotes or triple double-quotes span
// multiple lines and include line delimiters.
example19() {
  print('''Example19 <a href="etc">
Example19 Don't can't I'm Etc
Example19 </a>''');
}

// Strings have the nice interpolation feature with the $ character.
// With $ { [expression] }, the return of the expression is interpolated.
// $ followed by a variable name interpolates the content of that variable.
// $ can be escaped like so \$ to just add it to the string instead.
example20() {
  var s1 = "'\${s}'", s2 = "'\$s'";
  print("Example20 \$ interpolation ${s1} or $s2 works.");
}

// Optional types allow for the annotation of APIs and come to the aid of
// IDEs so the IDEs can better refactor, auto-complete and check for
// errors. So far we haven't declared any types and the programs have
// worked just fine. In fact, types are disregarded during runtime.
// Types can even be wrong and the program will still be given the
// benefit of the doubt and be run as though the types didn't matter.
// There's a runtime parameter that checks for type errors which is
// the checked mode, which is said to be useful during development time,
// but which is also slower because of the extra checking and is thus
// avoided during deployment runtime.
class Example21 {
  List<String> _names;
  Example21() {
    _names = ["a", "b"];
  }
  List<String> get names => _names;
  set names(List<String> list) {
    _names = list;
  }
  int get length => _names.length;
  void add(String name) {
    _names.add(name);
  }
}
void example21() {
  Example21 o = new Example21();
  o.add("c");
  print("Example21 names '${o.names}' and length '${o.length}'");
  o.names = ["d", "e"];
  print("Example21 names '${o.names}' and length '${o.length}'");
}

// Class inheritance takes the form of class name extends AnotherClassName {}.
class Example22A {
  var _name = "Some Name!";
  get name => _name;
}
class Example22B extends Example22A {}
example22() {
  var o = new Example22B();
  print("Example22 class inheritance '${o.name}'");
}

// Class mixin is also available, and takes the form of
// class name extends SomeClass with AnotherClassName {}.
// It's necessary to extend some class to be able to mixin another one.
// The template class of mixin cannot at the moment have a constructor.
// Mixin is mostly used to share methods with distant classes, so the
// single inheritance doesn't get in the way of reusable code.
// Mixins follow the "with" statement during the class declaration.
class Example23A {}
class Example23Utils {
  addTwo(n1, n2) {
    return n1 + n2;
  }
}
class Example23B extends Example23A with Example23Utils {
  addThree(n1, n2, n3) {
    return addTwo(n1, n2) + n3;
  }
}
example23() {
  var o = new Example23B(), r1 = o.addThree(1, 2, 3),
    r2 = o.addTwo(1, 2);
  print("Example23 addThree(1, 2, 3) results in '${r1}'");
  print("Example23 addTwo(1, 2) results in '${r2}'");
}

// The Class constructor method uses the same name of the class and
// takes the form of SomeClass() : super() {}, where the ": super()"
// part is optional and it's used to delegate constant parameters to the
// super-parent's constructor.
class Example24A {
  var _value;
  Example24A({value: "someValue"}) {
    _value = value;
  }
  get value => _value;
}
class Example24B extends Example24A {
  Example24B({value: "someOtherValue"}) : super(value: value);
}
example24() {
  var o1 = new Example24B(),
    o2 = new Example24B(value: "evenMore");
  print("Example24 calling super during constructor '${o1.value}'");
  print("Example24 calling super during constructor '${o2.value}'");
}

// There's a shortcut to set constructor parameters in case of simpler classes.
// Just use the this.parameterName prefix and it will set the parameter on
// an instance variable of same name.
class Example25 {
  var value, anotherValue;
  Example25({this.value, this.anotherValue});
}
example25() {
  var o = new Example25(value: "a", anotherValue: "b");
  print("Example25 shortcut for constructor '${o.value}' and "
    "'${o.anotherValue}'");
}

// Named parameters are available when declared between {}.
// Parameter order can be optional when declared between {}.
// Parameters can be made optional when declared between [].
example26() {
  var _name, _surname, _email;
  setConfig1({name, surname}) {
    _name = name;
    _surname = surname;
  }
  setConfig2(name, [surname, email]) {
    _name = name;
    _surname = surname;
    _email = email;
  }
  setConfig1(surname: "Doe", name: "John");
  print("Example26 name '${_name}', surname '${_surname}', "
    "email '${_email}'");
  setConfig2("Mary", "Jane");
  print("Example26 name '${_name}', surname '${_surname}', "
  "email '${_email}'");
}
```

```
// Variables declared with final can only be set once.
// In case of classes, final instance variables can be set via constant
// constructor parameter.
class Example27 {
  final color1, color2;
  // A little flexibility to set final instance variables with syntax
  // that follows the :
  Example27({this.color1, color2}) : color2 = color2;
}
example27() {
  final color = "orange", o = new Example27(color1: "lilac", color2: "white");
  print("Example27 color is '${color}'");
  print("Example27 color is '${o.color1}' and '${o.color2}'");
}

// To import a library, use import "libraryPath" or if it's a core library,
// import "dart:libraryName". There's also the "pub" package management with
// its own convention of import "package:packageName".
// See import "dart:collection"; at the top. Imports must come before
// other code declarations. IterableBase comes from dart:collection.
class Example28 extends IterableBase {
  var names;
  Example28() {
    names = ["a", "b"];
  }
  get iterator => names.iterator;
}
example28() {
  var o = new Example28();
  o.forEach((name) => print("Example28 '${name}'"));
}

// For control flow we have:
// * standard switch with must break statements
// * if-else if-else and ternary ..?..:.. operator
// * closures and anonymous functions
// * break, continue and return statements
example29() {
  var v = true ? 30 : 60;
  switch (v) {
    case 30:
      print("Example29 switch statement");
      break;
  }
  if (v < 30) {
  } else if (v > 30) {
  } else {
    print("Example29 if-else statement");
  }
  callItForMe(fn()) {
    return fn();
  }
  rand() {
    v = new DM.Random().nextInt(50);
    return v;
  }
  while (true) {
    print("Example29 callItForMe(rand) '${callItForMe(rand)}'");
    if (v != 30) {
      break;
    } else {
      continue;
    }
    // Never gets here.
  }
}

// Parse int, convert double to int, or just keep int when dividing numbers
// by using the ~/ operation. Let's play a guess game too.
example30() {
  var gn, tooHigh = false,
    n, n2 = (2.0).toInt(), top = int.parse("123") ~/ n2, bottom = 0;
  top = top ~/ 6;
  gn = new DM.Random().nextInt(top + 1); // +1 because nextInt top is exclusive
  print("Example30 Guess a number between 0 and ${top}");
  guessNumber(i) {
    if (n == gn) {
      print("Example30 Guessed right! The number is ${gn}");
    } else {
      tooHigh = n > gn;
      print("Example30 Number ${n} is too "
        "${tooHigh ? 'high' : 'low'}. Try again");
    }
    return n == gn;
  }
  n = (top - bottom) ~/ 2;
  while (!guessNumber(n)) {
    if (tooHigh) {
      top = n - 1;
    } else {
      bottom = n + 1;
    }
    n = bottom + ((top - bottom) ~/ 2);
  }
}

// Programs have only one entry point in the main function.
// Nothing is expected to be executed on the outer scope before a program
// starts running with what's in its main function.
// This helps with faster loading and even lazily loading of just what
// the program needs to startup with.
main() {
  print("Learn Dart in 15 minutes!");
  [example1, example2, example3, example4, example5, example6, example7,
    example8, example9, example10, example11, example12, example13, example14,
    example15, example16, example17, example18, example19, example20,
    example21, example22, example23, example24, example25, example26,
    example27, example28, example29, example30
    ].forEach((ef) => ef());
}
```

Further Reading

Dart has a comprehensive web-site. It covers API reference, tutorials, articles and more, including a useful Try Dart online. http://www.dartlang.org/ http://try.dartlang.org/

Edn

Extensible Data Notation (EDN) is a format for serializing data.

The notation is used internally by Clojure to represent programs. It is also used as a data transfer format like JSON. Though it is more commonly used in Clojure, there are implementations of EDN for many other languages.

The main benefit of EDN over JSON and YAML is that it is extensible. We will see how it is extended later on.

```
; Comments start with a semicolon.
; Anything after the semicolon is ignored.

;;;;;;;;;;;;;;;;;;;
;;; Basic Types ;;;
;;;;;;;;;;;;;;;;;;;

nil         ; also known in other languages as null

; Booleans
true
false

; Strings are enclosed in double quotes
"hungarian breakfast"
"farmer's cheesy omelette"

; Characters are preceeded by backslashes
\g \r \a \c \e

; Keywords start with a colon. They behave like enums. Kind of
; like symbols in Ruby.
:eggs
:cheese
:olives

; Symbols are used to represent identifiers. They start with #.
; You can namespace symbols by using /. Whatever preceeds / is
; the namespace of the name.
#spoon
#kitchen/spoon ; not the same as #spoon
#kitchen/fork
#github/fork   ; you can't eat with this

; Integers and floats
```

```
42
3.14159

; Lists are sequences of values
(:bun :beef-patty 9 "yum!")

; Vectors allow random access
[:gelato 1 2 -2]

; Maps are associative data structures that associate the key with its value
{:eggs        2
 :lemon-juice 3.5
 :butter      1}

; You're not restricted to using keywords as keys
{[1 2 3 4] "tell the people what she wore",
 [5 6 7 8] "the more you see the more you hate"}

; You may use commas for readability. They are treated as whitespace.

; Sets are collections that contain unique elements.
#{:a :b 88 "huat"}

;;;;;;;;;;;;;;;;;;;;;;;
;;; Tagged Elements ;;;
;;;;;;;;;;;;;;;;;;;;;;;

; EDN can be extended by tagging elements with # symbols.

#MyYelpClone/MenuItem {:name "eggs-benedict" :rating 10}

; Let me explain this with a clojure example. Suppose I want to transform that
; piece of EDN into a MenuItem record.

(defrecord MenuItem [name rating])

; To transform EDN to clojure values, I will need to use the built in EDN
; reader, edn/read-string

(edn/read-string "{:eggs 2 :butter 1 :flour 5}")
; -> {:eggs 2 :butter 1 :flour 5}

; To transform tagged elements, define the reader function and pass a map
; that maps tags to reader functions to edn/read-string like so

(edn/read-string {:readers {'MyYelpClone/MenuItem map->menu-item}}
                 "#MyYelpClone/MenuItem {:name \"eggs-benedict\" :rating 10}")
; -> #user.MenuItem{:name "eggs-benedict", :rating 10}
```

References

- EDN spec
- Implementations

- Tagged Elements

Elisp

```
;; This gives an introduction to Emacs Lisp in 15 minutes (v0.2d)
;;
;; Author: Bastien / @bzg2 / http://bzg.fr
;;
;; First make sure you read this text by Peter Norvig:
;; http://norvig.com/21-days.html
;;
;; Then install GNU Emacs 24.3:
;;
;; Debian: apt-get install emacs (or see your distro instructions)
;; OSX: http://emacsformacosx.com/emacs-builds/Emacs-24.3-universal-10.6.8.dmg
;; Windows: http://ftp.gnu.org/gnu/windows/emacs/emacs-24.3-bin-i386.zip
;;
;; More general information can be found at:
;; http://www.gnu.org/software/emacs/#Obtaining

;; Important warning:
;;
;; Going through this tutorial won't damage your computer unless
;; you get so angry that you throw it on the floor.  In that case,
;; I hereby decline any responsibility.  Have fun!

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Fire up Emacs.
;;
;; Hit the `q' key to dismiss the welcome message.
;;
;; Now look at the gray line at the bottom of the window:
;;
;; "*scratch*" is the name of the editing space you are now in.
;; This editing space is called a "buffer".
;;
;; The scratch buffer is the default buffer when opening Emacs.
;; You are never editing files: you are editing buffers that you
;; can save to a file.
;;
;; "Lisp interaction" refers to a set of commands available here.
;;
;; Emacs has a built-in set of commands available in every buffer,
;; and several subsets of commands available when you activate a
;; specific mode.  Here we use the `lisp-interaction-mode', which
;; comes with commands to evaluate and navigate within Elisp code.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Semi-colons start comments anywhere on a line.
;;
;; Elisp programs are made of symbolic expressions ("sexps"):
```

```
(+ 2 2)

;; This symbolic expression reads as "Add 2 to 2".

;; Sexps are enclosed into parentheses, possibly nested:
(+ 2 (+ 1 1))

;; A symbolic expression contains atoms or other symbolic
;; expressions.  In the above examples, 1 and 2 are atoms,
;; (+ 2 (+ 1 1)) and (+ 1 1) are symbolic expressions.

;; From `lisp-interaction-mode' you can evaluate sexps.
;; Put the cursor right after the closing parenthesis then
;; hold down the control and hit the j keys ("C-j" for short).

(+ 3 (+ 1 2))
;;           ^ cursor here
;; `C-j' => 6

;; `C-j' inserts the result of the evaluation in the buffer.

;; `C-xC-e' displays the same result in Emacs bottom line,
;; called the "minibuffer".  We will generally use `C-xC-e',
;; as we don't want to clutter the buffer with useless text.

;; `setq' stores a value into a variable:
(setq my-name "Bastien")
;; `C-xC-e' => "Bastien" (displayed in the mini-buffer)

;; `insert' will insert "Hello!" where the cursor is:
(insert "Hello!")
;; `C-xC-e' => "Hello!"

;; We used `insert' with only one argument "Hello!", but
;; we can pass more arguments -- here we use two:

(insert "Hello" " world!")
;; `C-xC-e' => "Hello world!"

;; You can use variables instead of strings:
(insert "Hello, I am " my-name)
;; `C-xC-e' => "Hello, I am Bastien"

;; You can combine sexps into functions:
(defun hello () (insert "Hello, I am " my-name))
;; `C-xC-e' => hello

;; You can evaluate functions:
(hello)
;; `C-xC-e' => Hello, I am Bastien

;; The empty parentheses in the function's definition means that
;; it does not accept arguments.  But always using `my-name' is
;; boring, let's tell the function to accept one argument (here
;; the argument is called "name"):

(defun hello (name) (insert "Hello " name))
;; `C-xC-e' => hello

;; Now let's call the function with the string "you" as the value
;; for its unique argument:
(hello "you")
;; `C-xC-e' => "Hello you"

;; Yeah!

;; Take a breath.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Now switch to a new buffer named "*test*" in another window:

(switch-to-buffer-other-window "*test*")
;; `C-xC-e'
;; => [screen has two windows and cursor is in the *test* buffer]

;; Mouse over the top window and left-click to go back.  Or you can
;; use `C-xo' (i.e. hold down control-x and hit o) to go to the other
;; window interactively.

;; You can combine several sexps with `progn':
(progn
  (switch-to-buffer-other-window "*test*")
  (hello "you"))
;; `C-xC-e'
;; => [The screen has two windows and cursor is in the *test* buffer]

;; Now if you don't mind, I'll stop asking you to hit `C-xC-e': do it
;; for every sexp that follows.

;; Always go back to the *scratch* buffer with the mouse or `C-xo'.

;; It's often useful to erase the buffer:
(progn
  (switch-to-buffer-other-window "*test*")
  (erase-buffer)
  (hello "there"))

;; Or to go back to the other window:
(progn
  (switch-to-buffer-other-window "*test*")
  (erase-buffer)
  (hello "you")
  (other-window 1))

;; You can bind a value to a local variable with `let':
(let ((local-name "you"))
  (switch-to-buffer-other-window "*test*")
  (erase-buffer)
  (hello local-name)
  (other-window 1))

;; No need to use `progn' in that case, since `let' also combines
;; several sexps.

;; Let's format a string:
(format "Hello %s!\n" "visitor")

;; %s is a place-holder for a string, replaced by "visitor".
;; \n is the newline character.

;; Let's refine our function by using format:
(defun hello (name)
  (insert (format "Hello %s!\n" name)))

(hello "you")

;; Let's create another function which uses `let':
(defun greeting (name)
  (let ((your-name "Bastien"))
    (insert (format "Hello %s!\n\nI am %s."
                    name       ; the argument of the function
                    your-name  ; the let-bound variable "Bastien"
                    ))))

;; And evaluate it:
(greeting "you")

;; Some function are interactive:
(read-from-minibuffer "Enter your name: ")

;; Evaluating this function returns what you entered at the prompt.

;; Let's make our `greeting' function prompt for your name:
(defun greeting (from-name)
  (let ((your-name (read-from-minibuffer "Enter your name: ")))
    (insert (format "Hello!\n\nI am %s and you are %s."
                    from-name ; the argument of the function
                    your-name ; the let-bound var, entered at prompt
                    ))))

(greeting "Bastien")

;; Let's complete it by displaying the results in the other window:
(defun greeting (from-name)
  (let ((your-name (read-from-minibuffer "Enter your name: ")))
    (switch-to-buffer-other-window "*test*")
    (erase-buffer)
    (insert (format "Hello %s!\n\nI am %s." your-name from-name))
    (other-window 1)))

;; Now test it:
```

```
(greeting "Bastien")

;; Take a breath.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Let's store a list of names:
(setq list-of-names '("Sarah" "Chloe" "Mathilde"))

;; Get the first element of this list with `car':
(car list-of-names)

;; Get a list of all but the first element with `cdr':
(cdr list-of-names)

;; Add an element to the beginning of a list with `push':
(push "Stephanie" list-of-names)

;; NOTE: `car' and `cdr' don't modify the list, but `push' does.
;; This is an important difference: some functions don't have any
;; side-effects (like `car') while others have (like `push').

;; Let's call `hello' for each element in `list-of-names':
(mapcar 'hello list-of-names)

;; Refine `greeting' to say hello to everyone in `list-of-names':
(defun greeting ()
    (switch-to-buffer-other-window "*test*")
    (erase-buffer)
    (mapcar 'hello list-of-names)
    (other-window 1))

(greeting)

;; Remember the `hello' function we defined above?  It takes one
;; argument, a name.  `mapcar' calls `hello', successively using each
;; element of `list-of-names' as the argument for `hello'.

;; Now let's arrange a bit what we have in the displayed buffer:

(defun replace-hello-by-bonjour ()
    (switch-to-buffer-other-window "*test*")
    (goto-char (point-min))
    (while (search-forward "Hello")
      (replace-match "Bonjour"))
    (other-window 1))

;; (goto-char (point-min)) goes to the beginning of the buffer.
;; (search-forward "Hello") searches for the string "Hello".
;; (while x y) evaluates the y sexp(s) while x returns something.
;; If x returns `nil' (nothing), we exit the while loop.

(replace-hello-by-bonjour)
```

```
;; You should see all occurrences of "Hello" in the *test* buffer
;; replaced by "Bonjour".

;; You should also get an error: "Search failed: Hello".
;;
;; To avoid this error, you need to tell `search-forward' whether it
;; should stop searching at some point in the buffer, and whether it
;; should silently fail when nothing is found:

;; (search-forward "Hello" nil 't) does the trick:

;; The `nil' argument says: the search is not bound to a position.
;; The `'t' argument says: silently fail when nothing is found.

;; We use this sexp in the function below, which doesn't throw an error:

(defun hello-to-bonjour ()
    (switch-to-buffer-other-window "*test*")
    (erase-buffer)
    ;; Say hello to names in `list-of-names'
    (mapcar 'hello list-of-names)
    (goto-char (point-min))
    ;; Replace "Hello" by "Bonjour"
    (while (search-forward "Hello" nil 't)
      (replace-match "Bonjour"))
    (other-window 1))

(hello-to-bonjour)

;; Let's colorize the names:

(defun boldify-names ()
    (switch-to-buffer-other-window "*test*")
    (goto-char (point-min))
    (while (re-search-forward "Bonjour \\(.+\\)!" nil 't)
      (add-text-properties (match-beginning 1)
                           (match-end 1)
                           (list 'face 'bold)))
    (other-window 1))

;; This functions introduces `re-search-forward': instead of
;; searching for the string "Bonjour", you search for a pattern,
;; using a "regular expression" (abbreviated in the prefix "re-").

;; The regular expression is "Bonjour \\(.+\\)!" and it reads:
;; the string "Bonjour ", and
;; a group of            | this is the \\( ... \\) construct
;;   any character       | this is the .
;;   possibly repeated   | this is the +
;; and the "!" string.

;; Ready?  Test it!

(boldify-names)
```

```
;; `add-text-properties' adds... text properties, like a face.

;; OK, we are done.  Happy hacking!

;; If you want to know more about a variable or a function:
;;
;; C-h v a-variable RET
;; C-h f a-function RET
;;
;; To read the Emacs Lisp manual with Emacs:
;;
;; C-h i m elisp RET
;;
;; To read an online introduction to Emacs Lisp:
;; https://www.gnu.org/software/emacs/manual/html_node/eintr/index.html

;; Thanks to these people for their feedback and suggestions:
;; - Wes Hardaker
;; - notbob
;; - Kevin Montuori
;; - Arne Babenhauserheide
;; - Alan Schmitt
;; - LinXitoW
;; - Aaron Meurer
```

Elixir

Elixir is a modern functional language built on top of the Erlang VM. It's fully compatible with Erlang, but features a more standard syntax and many more features.

```
# Single line comments start with a number symbol.

# There's no multi-line comment,
# but you can stack multiple comments.

# To use the elixir shell use the `iex` command.
# Compile your modules with the `elixirc` command.

# Both should be in your path if you installed elixir correctly.

## ---------------------------
## -- Basic types
## ---------------------------

# There are numbers
3    # integer
0x1F # integer
3.0  # float

# Atoms, that are literals, a constant with name. They start with `:`.
:hello # atom
```

```
# Tuples that are stored contiguously in memory.
{1,2,3} # tuple

# We can access a tuple element with the `elem` function:
elem({1, 2, 3}, 0) #=> 1

# Lists that are implemented as linked lists.
[1,2,3] # list

# We can access the head and tail of a list as follows:
[head | tail] = [1,2,3]
head #=> 1
tail #=> [2,3]

# In elixir, just like in Erlang, the `=` denotes pattern matching and
# not an assignment.
#
# This means that the left-hand side (pattern) is matched against a
# right-hand side.
#
# This is how the above example of accessing the head and tail of a list works.

# A pattern match will error when the sides don't match, in this example
# the tuples have different sizes.
# {a, b, c} = {1, 2} #=> ** (MatchError) no match of right hand side value: {1,2}

# There are also binaries
<<1,2,3>> # binary

# Strings and char lists
"hello" # string
'hello' # char list

# Multi-line strings
"""
I'm a multi-line
string.
"""
#=> "I'm a multi-line\nstring.\n"

# Strings are all encoded in UTF-8:
"héllò" #=> "héllò"

# Strings are really just binaries, and char lists are just lists.
<<?a, ?b, ?c>> #=> "abc"
[?a, ?b, ?c]   #=> 'abc'

# `?a` in elixir returns the ASCII integer for the letter `a`
?a #=> 97

# To concatenate lists use `++`, for binaries use `<>`
[1,2,3] ++ [4,5]     #=> [1,2,3,4,5]
'hello ' ++ 'world'  #=> 'hello world'
```

```
<<1,2,3>> <> <<4,5>> #=> <<1,2,3,4,5>>
"hello " <> "world"  #=> "hello world"

# Ranges are represented as `start..end` (both inclusive)
1..10 #=> 1..10
lower..upper = 1..10 # Can use pattern matching on ranges as well
[lower, upper] #=> [1, 10]

## ---------------------------
## -- Operators
## ---------------------------

# Some math
1 + 1  #=> 2
10 - 5 #=> 5
5 * 2  #=> 10
10 / 2 #=> 5.0

# In elixir the operator `/` always returns a float.

# To do integer division use `div`
div(10, 2) #=> 5

# To get the division remainder use `rem`
rem(10, 3) #=> 1

# There are also boolean operators: `or`, `and` and `not`.
# These operators expect a boolean as their first argument.
true and true #=> true
false or true #=> true
# 1 and true    #=> ** (ArgumentError) argument error

# Elixir also provides `||`, `&&` and `!` which accept arguments of any type.
# All values except `false` and `nil` will evaluate to true.
1 || true  #=> 1
false && 1 #=> false
nil && 20  #=> nil
!true #=> false

# For comparisons we have: `==`, `!=`, `===`, `!==`, `<=`, `>=`, `<` and `>`
1 == 1 #=> true
1 != 1 #=> false
1 < 2  #=> true

# `===` and `!==` are more strict when comparing integers and floats:
1 == 1.0  #=> true
1 === 1.0 #=> false

# We can also compare two different data types:
1 < :hello #=> true

# The overall sorting order is defined below:
# number < atom < reference < functions < port < pid < tuple < list < bit string
# To quote Joe Armstrong on this: "The actual order is not important,
# but that a total ordering is well defined is important."

## ---------------------------
## -- Control Flow
## ---------------------------

# `if` expression
if false do
  "This will never be seen"
else
  "This will"
end

# There's also `unless`
unless true do
  "This will never be seen"
else
  "This will"
end

# Remember pattern matching? Many control-flow structures in elixir rely on it.

# `case` allows us to compare a value against many patterns:
case {:one, :two} do
  {:four, :five} ->
    "This won't match"
  {:one, x} ->
    "This will match and bind `x` to `:two`"
  _ ->
    "This will match any value"
end

# It's common to bind the value to `_` if we don't need it.
# For example, if only the head of a list matters to us:
[head | _] = [1,2,3]
head #=> 1

# For better readability we can do the following:
[head | _tail] = [:a, :b, :c]
head #=> :a

# `cond` lets us check for many conditions at the same time.
# Use `cond` instead of nesting many `if` expressions.
cond do
  1 + 1 == 3 ->
    "I will never be seen"
  2 * 5 == 12 ->
    "Me neither"
  1 + 2 == 3 ->
    "But I will"
end
```

```
# It is common to set the last condition equal to `true`, which will always match.
cond do
  1 + 1 == 3 ->
    "I will never be seen"
  2 * 5 == 12 ->
    "Me neither"
  true ->
    "But I will (this is essentially an else)"
end

# `try/catch` is used to catch values that are thrown, it also supports an
# `after` clause that is invoked whether or not a value is caught.
try do
  throw(:hello)
catch
  message -> "Got #{message}."
after
  IO.puts("I'm the after clause.")
end
#=> I'm the after clause
# "Got :hello"

## ---------------------------
## -- Modules and Functions
## ---------------------------

# Anonymous functions (notice the dot)
square = fn(x) -> x * x end
square.(5) #=> 25

# They also accept many clauses and guards.
# Guards let you fine tune pattern matching,
# they are indicated by the `when` keyword:
f = fn
  x, y when x > 0 -> x + y
  x, y -> x * y
end

f.(1, 3)  #=> 4
f.(-1, 3) #=> -3

# Elixir also provides many built-in functions.
# These are available in the current scope.
is_number(10)    #=> true
is_list("hello") #=> false
elem({1,2,3}, 0) #=> 1

# You can group several functions into a module. Inside a module use `def`
# to define your functions.
defmodule Math do
  def sum(a, b) do
    a + b
  end
  def square(x) do
    x * x
  end
end

Math.sum(1, 2)  #=> 3
Math.square(3) #=> 9

# To compile our simple Math module save it as `math.ex` and use `elixirc`
# in your terminal: elixirc math.ex

# Inside a module we can define functions with `def` and private functions with `defp`.
# A function defined with `def` is available to be invoked from other modules,
# a private function can only be invoked locally.
defmodule PrivateMath do
  def sum(a, b) do
    do_sum(a, b)
  end

  defp do_sum(a, b) do
    a + b
  end
end

PrivateMath.sum(1, 2)    #=> 3
# PrivateMath.do_sum(1, 2) #=> ** (UndefinedFunctionError)

# Function declarations also support guards and multiple clauses:
defmodule Geometry do
  def area({:rectangle, w, h}) do
    w * h
  end

  def area({:circle, r}) when is_number(r) do
    3.14 * r * r
  end
end

Geometry.area({:rectangle, 2, 3}) #=> 6
Geometry.area({:circle, 3})       #=> 28.25999999999999801048
# Geometry.area({:circle, "not_a_number"})
#=> ** (FunctionClauseError) no function clause matching in Geometry.area/1

# Due to immutability, recursion is a big part of elixir
defmodule Recursion do
  def sum_list([head | tail], acc) do
    sum_list(tail, acc + head)
  end

  def sum_list([], acc) do
    acc
  end
end
```

```
Recursion.sum_list([1,2,3], 0) #=> 6

# Elixir modules support attributes, there are built-in attributes and you
# may also add custom ones.
defmodule MyMod do
  @moduledoc """
  This is a built-in attribute on a example module.
  """

  @my_data 100 # This is a custom attribute.
  IO.inspect(@my_data) #=> 100
end

## ---------------------------
## -- Structs and Exceptions
## ---------------------------

# Structs are extensions on top of maps that bring default values,
# compile-time guarantees and polymorphism into Elixir.
defmodule Person do
  defstruct name: nil, age: 0, height: 0
end

joe_info = %Person{ name: "Joe", age: 30, height: 180 }
#=> %Person{age: 30, height: 180, name: "Joe"}

# Access the value of name
joe_info.name #=> "Joe"

# Update the value of age
older_joe_info = %{ joe_info | age: 31 }
#=> %Person{age: 31, height: 180, name: "Joe"}

# The `try` block with the `rescue` keyword is used to handle exceptions
try do
  raise "some error"
rescue
  RuntimeError -> "rescued a runtime error"
  _error -> "this will rescue any error"
end
#=> "rescued a runtime error"

# All exceptions have a message
try do
  raise "some error"
rescue
  x in [RuntimeError] ->
    x.message
end
#=> "some error"

## ---------------------------
## -- Concurrency
## ---------------------------
```

```
# Elixir relies on the actor model for concurrency. All we need to write
# concurrent programs in elixir are three primitives: spawning processes,
# sending messages and receiving messages.

# To start a new process we use the `spawn` function, which takes a function
# as argument.
f = fn -> 2 * 2 end #=> #Function<erl_eval.20.80484245>
spawn(f) #=> #PID<0.40.0>

# `spawn` returns a pid (process identifier), you can use this pid to send
# messages to the process. To do message passing we use the `send` operator.
# For all of this to be useful we need to be able to receive messages. This is
# achieved with the `receive` mechanism:

# The `receive do` block is used to listen for messages and process
# them when they are received. A `receive do` block will only
# process one received message. In order to process multiple
# messages, a function with a `receive do` block must recursively
# call itself to get into the `receive do` block again.

defmodule Geometry do
  def area_loop do
    receive do
      {:rectangle, w, h} ->
        IO.puts("Area = #{w * h}")
        area_loop()
      {:circle, r} ->
        IO.puts("Area = #{3.14 * r * r}")
        area_loop()
    end
  end
end

# Compile the module and create a process that evaluates `area_loop` in the shell
pid = spawn(fn -> Geometry.area_loop() end) #=> #PID<0.40.0>
# Alternatively
pid = spawn(Geometry, :area_loop, [])

# Send a message to `pid` that will match a pattern in the receive statement
send pid, {:rectangle, 2, 3}
#=> Area = 6
#   {:rectangle,2,3}

send pid, {:circle, 2}
#=> Area = 12.56000000000000049738
#   {:circle,2}

# The shell is also a process, you can use `self` to get the current pid
self() #=> #PID<0.27.0>
```

References

- Getting started guide from elixir webpage
- Elixir Documentation
- "Programming Elixir" by Dave Thomas
- Elixir Cheat Sheet
- "Learn You Some Erlang for Great Good!" by Fred Hebert
- "Programming Erlang: Software for a Concurrent World" by Joe Armstrong

Elm

Elm is a functional reactive programming language that compiles to (client-side) JavaScript. Elm is statically typed, meaning that the compiler catches most errors immediately and provides a clear and understandable error message. Elm is great for designing user interfaces and games for the web.

```
-- Single line comments start with two dashes.
{- Multiline comments can be enclosed in a block like this.
{- They can be nested. -}
-}

{-- The Basics --}

-- Arithmetic
1 + 1 -- 2
8 - 1 -- 7
10 * 2 -- 20

-- Every number literal without a decimal point can be either an Int or a Float.
33 / 2 -- 16.5 with floating point division
33 // 2 -- 16 with integer division

-- Exponents
5 ^ 2 -- 25

-- Booleans
not True -- False
not False -- True
1 == 1 -- True
1 /= 1 -- False
1 < 10 -- True

-- Strings and characters
"This is a string because it uses double quotes."
'a' -- characters in single quotes

-- Strings can be appended.
"Hello " ++ "world!" -- "Hello world!"

{-- Lists, Tuples, and Records --}

-- Every element in a list must have the same type.
["the", "quick", "brown", "fox"]
[1, 2, 3, 4, 5]
```

```
-- The second example can also be written with two dots.
[1..5]

-- Append lists just like strings.
[1..5] ++ [6..10] == [1..10] -- True

-- To add one item, use "cons".
0 :: [1..5] -- [0, 1, 2, 3, 4, 5]

-- The head and tail of a list are returned as a Maybe. Instead of checking
-- every value to see if it's null, you deal with missing values explicitly.
List.head [1..5] -- Just 1
List.tail [1..5] -- Just [2, 3, 4, 5]
List.head [] -- Nothing
-- List.functionName means the function lives in the List module.

-- Every element in a tuple can be a different type, but a tuple has a
-- fixed length.
("elm", 42)

-- Access the elements of a pair with the first and second functions.
-- (This is a shortcut; we'll come to the "real way" in a bit.)
fst ("elm", 42) -- "elm"
snd ("elm", 42) -- 42

-- The empty tuple, or "unit", is sometimes used as a placeholder.
-- It is the only value of its type, also called "Unit".
()

-- Records are like tuples but the fields have names. The order of fields
-- doesn't matter. Notice that record values use equals signs, not colons.
{ x = 3, y = 7 }

-- Access a field with a dot and the field name.
{ x = 3, y = 7 }.x -- 3

-- Or with an accessor fuction, which is a dot and the field name on its own.
.y { x = 3, y = 7 } -- 7

-- Update the fields of a record. (It must have the fields already.)
{ person |
  name = "George" }

-- Update multiple fields at once, using the current values.
{ particle |
  position = particle.position + particle.velocity,
  velocity = particle.velocity + particle.acceleration }

{-- Control Flow --}

-- If statements always have an else, and the branches must be the same type.
if powerLevel > 9000 then
  "WHOA!"
else
```

```
  "meh"

-- If statements can be chained.
if n < 0 then
  "n is negative"
else if n > 0 then
  "n is positive"
else
  "n is zero"

-- Use case statements to pattern match on different possibilities.
case aList of
  [] -> "matches the empty list"
  [x]-> "matches a list of exactly one item, " ++ toString x
  x::xs -> "matches a list of at least one item whose head is " ++ toString x
-- Pattern matches go in order. If we put [x] last, it would never match because
-- x::xs also matches (xs would be the empty list). Matches do not "fall through".
-- The compiler will alert you to missing or extra cases.

-- Pattern match on a Maybe.
case List.head aList of
  Just x -> "The head is " ++ toString x
  Nothing -> "The list was empty."

{-- Functions --}

-- Elm's syntax for functions is very minimal, relying mostly on whitespace
-- rather than parentheses and curly brackets. There is no "return" keyword.

-- Define a function with its name, arguments, an equals sign, and the body.
multiply a b =
  a * b

-- Apply (call) a function by passing it arguments (no commas necessary).
multiply 7 6 -- 42

-- Partially apply a function by passing only some of its arguments.
-- Then give that function a new name.
double =
  multiply 2

-- Constants are similar, except there are no arguments.
answer =
  42

-- Pass functions as arguments to other functions.
List.map double [1..4] -- [2, 4, 6, 8]

-- Or write an anonymous function.
List.map (\a -> a * 2) [1..4] -- [2, 4, 6, 8]

-- You can pattern match in function definitions when there's only one case.
-- This function takes one tuple rather than two arguments.
area (width, height) =
  width * height

area (6, 7) -- 42

-- Use curly brackets to pattern match record field names.
-- Use let to define intermediate values.
volume {width, height, depth} =
  let
    area = width * height
  in
    area * depth

volume { width = 3, height = 2, depth = 7 } -- 42

-- Functions can be recursive.
fib n =
  if n < 2 then
    1
  else
    fib (n - 1) + fib (n - 2)

List.map fib [0..8] -- [1, 1, 2, 3, 5, 8, 13, 21, 34]

-- Another recursive function (use List.length in real code).
listLength aList =
  case aList of
    [] -> 0
    x::xs -> 1 + listLength xs

-- Function calls happen before any infix operator. Parens indicate precedence.
cos (degrees 30) ^ 2 + sin (degrees 30) ^ 2 -- 1
-- First degrees is applied to 30, then the result is passed to the trig
-- functions, which is then squared, and the addition happens last.

{-- Types and Type Annotations --}

-- The compiler will infer the type of every value in your program.
-- Types are always uppercase. Read x : T as "x has type T".
-- Some common types, which you might see in Elm's REPL.
5 : Int
6.7 : Float
"hello" : String
True : Bool

-- Functions have types too. Read -> as "goes to". Think of the rightmost type
-- as the type of the return value, and the others as arguments.
not : Bool -> Bool
round : Float -> Int

-- When you define a value, it's good practice to write its type above it.
-- The annotation is a form of documentation, which is verified by the compiler.
double : Int -> Int
double x = x * 2
```

```
-- Function arguments are passed in parentheses.
-- Lowercase types are type variables: they can be any type, as long as each
-- call is consistent.
List.map : (a -> b) -> List a -> List b
-- "List dot map has type a-goes-to-b, goes to list of a, goes to list of b."

-- There are three special lowercase types: number, comparable, and appendable.
-- Numbers allow you to use arithmetic on Ints and Floats.
-- Comparable allows you to order numbers and strings, like a < b.
-- Appendable things can be combined with a ++ b.

{-- Type Aliases and Union Types --}

-- When you write a record or tuple, its type already exists.
-- (Notice that record types use colon and record values use equals.)
origin : { x : Float, y : Float, z : Float }
origin =
  { x = 0, y = 0, z = 0 }

-- You can give existing types a nice name with a type alias.
type alias Point3D =
  { x : Float, y : Float, z : Float }

-- If you alias a record, you can use the name as a constructor function.
otherOrigin : Point3D
otherOrigin =
  Point3D 0 0 0

-- But it's still the same type, so you can equate them.
origin == otherOrigin -- True

-- By contrast, defining a union type creates a type that didn't exist before.
-- A union type is so called because it can be one of many possibilities.
-- Each of the possibilities is represented as a "tag".
type Direction =
  North | South | East | West

-- Tags can carry other values of known type. This can work recursively.
type IntTree =
  Leaf | Node Int IntTree IntTree
-- "Leaf" and "Node" are the tags. Everything following a tag is a type.

-- Tags can be used as values or functions.
root : IntTree
root =
  Node 7 Leaf Leaf

-- Union types (and type aliases) can use type variables.
type Tree a =
  Leaf | Node a (Tree a) (Tree a)
-- "The type tree-of-a is a leaf, or a node of a, tree-of-a, and tree-of-a."

-- Pattern match union tags. The uppercase tags will be matched exactly. The
-- lowercase variables will match anything. Underscore also matches anything,
-- but signifies that you aren't using it.
leftmostElement : Tree a -> Maybe a
leftmostElement tree =
  case tree of
    Leaf -> Nothing
    Node x Leaf _ -> Just x
    Node _ subtree _ -> leftmostElement subtree

-- That's pretty much it for the language itself. Now let's see how to organize
-- and run your code.

{-- Modules and Imports --}

-- The core libraries are organized into modules, as are any third-party
-- libraries you may use. For large projects, you can define your own modules.

-- Put this at the top of the file. If omitted, you're in Main.
module Name where

-- By default, everything is exported. You can specify exports explicity.
module Name (MyType, myValue) where

-- One common pattern is to export a union type but not its tags. This is known
-- as an "opaque type", and is frequently used in libraries.

-- Import code from other modules to use it in this one.
-- Places Dict in scope, so you can call Dict.insert.
import Dict

-- Imports the Dict module and the Dict type, so your annotations don't have to
-- say Dict.Dict. You can still use Dict.insert.
import Dict exposing (Dict)

-- Rename an import.
import Graphics.Collage as C

{-- Ports --}

-- A port indicates that you will be communicating with the outside world.
-- Ports are only allowed in the Main module.

-- An incoming port is just a type signature.
port clientID : Int

-- An outgoing port has a defintion.
port clientOrders : List String
port clientOrders = ["Books", "Groceries", "Furniture"]

-- We won't go into the details, but you set up callbacks in JavaScript to send
-- on incoming ports and receive on outgoing ports.

{-- Command Line Tools --}

-- Compile a file.
```

```
$ elm make MyFile.elm

-- The first time you do this, Elm will install the core libraries and create
-- elm-package.json, where information about your project is kept.

-- The reactor is a server that compiles and runs your files.
-- Click the wrench next to file names to enter the time-travelling debugger!
$ elm reactor

-- Experiment with simple expressions in a Read-Eval-Print Loop.
$ elm repl

-- Packages are identified by GitHub username and repo name.
-- Install a new package, and record it in elm-package.json.
$ elm package install evancz/elm-html

-- See what changed between versions of a package.
$ elm package diff evancz/elm-html 3.0.0 4.0.2
-- Elm's package manager enforces semantic versioning, so minor version bumps
-- will never break your build!
```

The Elm language is surprisingly small. You can now look through almost any Elm source code and have a rough idea of what is going on. However, the possibilties for error-resistant and easy-to-refactor code are endless!

Here are some useful resources.

- The Elm website. Includes:
- Links to the installers
- Documentation guides, including the syntax reference
- Lots of helpful examples
- Documentation for Elm's core libraries. Take note of:
- Basics, which is imported by default
- Maybe and its cousin Result, commonly used for missing values or error handling
- Data structures like List, Array, Dict, and Set
- JSON encoding and decoding
- The Elm Architecture. An essay by Elm's creator with examples on how to organize code into components.
- The Elm mailing list. Everyone is friendly and helpful.
- Scope in Elm and How to Read a Type Annotation. More community guides on the basics of Elm, written for JavaScript developers.

Go out and write some Elm!

Erlang

```
% Percent sign starts a one-line comment.

%% Two percent characters shall be used to comment functions.
```

```erlang
%%% Three percent characters shall be used to comment modules.

% We use three types of punctuation in Erlang.
% Commas (`,`) separate arguments in function calls, data constructors, and
% patterns.
% Periods (`.`) (followed by whitespace) separate entire functions and
% expressions in the shell.
% Semicolons (`;`) separate clauses. We find clauses in several contexts:
% function definitions and in `case`, `if`, `try..catch`, and `receive`
% expressions.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 1. Variables and pattern matching.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% In Erlang new variables are bound with an `=` statement.
Num = 42.  % All variable names must start with an uppercase letter.

% Erlang has single-assignment variables; if you try to assign a different
% value to the variable `Num`, you’ll get an error.
Num = 43. % ** exception error: no match of right hand side value 43

% In most languages, `=` denotes an assignment statement. In Erlang, however,
% `=` denotes a pattern-matching operation. When an empty variable is used on the
% left hand side of the `=` operator to is bound (assigned), but when a bound
% variable is used on the left hand side the following behaviour is observed.
% `Lhs = Rhs` really means this: evaluate the right side (`Rhs`), and then
% match the result against the pattern on the left side (`Lhs`).
Num = 7 * 6.

% Floating-point number.
Pi = 3.14159.

% Atoms are used to represent different non-numerical constant values. Atoms
% start with lowercase letters, followed by a sequence of alphanumeric
% characters or the underscore (`_`) or at (`@`) sign.
Hello = hello.
OtherNode = example@node.

% Atoms with non alphanumeric values can be written by enclosing the atoms
% with apostrophes.
AtomWithSpace = 'some atom with space'.

% Tuples are similar to structs in C.
Point = {point, 10, 45}.

% If we want to extract some values from a tuple, we use the pattern-matching
% operator `=`.
{point, X, Y} = Point.  % X = 10, Y = 45

% We can use `_` as a placeholder for variables that we’re not interested in.
% The symbol `_` is called an anonymous variable. Unlike regular variables,
% several occurrences of `_` in the same pattern don’t have to bind to the
% same value.
```

```
Person = {person, {name, {first, joe}, {last, armstrong}}, {footsize, 42}}.
{_, {_, {_, Who}, _}, _} = Person.  % Who = joe

% We create a list by enclosing the list elements in square brackets and
% separating them with commas.
% The individual elements of a list can be of any type.
% The first element of a list is the head of the list. If you imagine removing
% the head from the list, what's left is called the tail of the list.
ThingsToBuy = [{apples, 10}, {pears, 6}, {milk, 3}].

% If `T` is a list, then `[H|T]` is also a list, with head `H` and tail `T`.
% The vertical bar (`|`) separates the head of a list from its tail.
% `[]` is the empty list.
% We can extract elements from a list with a pattern-matching operation. If we
% have a nonempty list `L`, then the expression `[X|Y] = L`, where `X` and `Y`
% are unbound variables, will extract the head of the list into `X` and the tail
% of the list into `Y`.
[FirstThing|OtherThingsToBuy] = ThingsToBuy.
% FirstThing = {apples, 10}
% OtherThingsToBuy = [{pears, 6}, {milk, 3}]

% There are no strings in Erlang. Strings are really just lists of integers.
% Strings are enclosed in double quotation marks (`"`).
Name = "Hello".
[72, 101, 108, 108, 111] = "Hello".


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 2. Sequential programming.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Modules are the basic unit of code in Erlang. All the functions we write are
% stored in modules. Modules are stored in files with `.erl` extensions.
% Modules must be compiled before the code can be run. A compiled module has the
% extension `.beam`.
-module(geometry).
-export([area/1]). % the list of functions exported from the module.

% The function `area` consists of two clauses. The clauses are separated by a
% semicolon, and the final clause is terminated by dot-whitespace.
% Each clause has a head and a body; the head consists of a function name
% followed by a pattern (in parentheses), and the body consists of a sequence of
% expressions, which are evaluated if the pattern in the head is successfully
% matched against the calling arguments. The patterns are matched in the order
% they appear in the function definition.
area({rectangle, Width, Ht}) -> Width * Ht;
area({circle, R})            -> 3.14159 * R * R.

% Compile the code in the file geometry.erl.
c(geometry).  % {ok,geometry}

% We need to include the module name together with the function name in order to
% identify exactly which function we want to call.
geometry:area({rectangle, 10, 5}).  % 50
```

```
geometry:area({circle, 1.4}).  % 6.15752

% In Erlang, two functions with the same name and different arity (number of
% arguments) in the same module represent entirely different functions.
-module(lib_misc).
-export([sum/1]). % export function `sum` of arity 1
                  % accepting one argument: list of integers.
sum(L) -> sum(L, 0).
sum([], N)    -> N;
sum([H|T], N) -> sum(T, H+N).

% Funs are "anonymous" functions. They are called this way because they have
% no name. However, they can be assigned to variables.
Double = fun(X) -> 2 * X end. % `Double` points to an anonymous function
                              % with handle: #Fun<erl_eval.6.17052888>
Double(2).  % 4

% Functions accept funs as their arguments and can return funs.
Mult = fun(Times) -> ( fun(X) -> X * Times end ) end.
Triple = Mult(3).
Triple(5).  % 15

% List comprehensions are expressions that create lists without having to use
% funs, maps, or filters.
% The notation `[F(X) || X <- L]` means "the list of `F(X)` where `X` is taken
% from the list `L`."
L = [1,2,3,4,5].
[2 * X || X <- L].  % [2,4,6,8,10]
% A list comprehension can have generators and filters, which select subset of
% the generated values.
EvenNumbers = [N || N <- [1, 2, 3, 4], N rem 2 == 0]. % [2, 4]

% Guards are constructs that we can use to increase the power of pattern
% matching. Using guards, we can perform simple tests and comparisons on the
% variables in a pattern.
% You can use guards in the heads of function definitions where they are
% introduced by the `when` keyword, or you can use them at any place in the
% language where an expression is allowed.
max(X, Y) when X > Y -> X;
max(X, Y) -> Y.

% A guard is a series of guard expressions, separated by commas (`,`).
% The guard `GuardExpr1, GuardExpr2, ..., GuardExprN` is true if all the guard
% expressions `GuardExpr1`, `GuardExpr2`, ..., `GuardExprN` evaluate to `true`.
is_cat(A) when is_atom(A), A =:= cat -> true;
is_cat(A) -> false.
is_dog(A) when is_atom(A), A =:= dog -> true;
is_dog(A) -> false.

% We won't dwell on the `=:=` operator here; just be aware that it is used to
% check whether two Erlang expressions have the same value *and* the same type.
% Contrast this behaviour to that of the `==` operator:
1 + 2 =:= 3.   % true
1 + 2 =:= 3.0. % false
```

```
1 + 2 ==  3.0. % true

% A guard sequence is either a single guard or a series of guards, separated
% by semicolons (`;`). The guard sequence `G1; G2; ...; Gn` is true if at
% least one of the guards `G1`, `G2`, ..., `Gn` evaluates to `true`.
is_pet(A) when is_atom(A), (A =:= dog);(A =:= cat) -> true;
is_pet(A)                                          -> false.

% Warning: not all valid Erlang expressions can be used as guard expressions;
% in particular, our `is_cat` and `is_dog` functions cannot be used within the
% guard sequence in `is_pet`'s definition. For a description of the
% expressions allowed in guard sequences, refer to this
% [section](http://erlang.org/doc/reference_manual/expressions.html#id81912)
% of the Erlang reference manual.

% Records provide a method for associating a name with a particular element in a
% tuple.
% Record definitions can be included in Erlang source code files or put in files
% with the extension `.hrl`, which are then included by Erlang source code
% files.
-record(todo, {
  status = reminder,  % Default value
  who = joe,
  text
}).

% We have to read the record definitions into the shell before we can define a
% record. We use the shell function `rr` (short for read records) to do this.
rr("records.hrl").  % [todo]

% Creating and updating records:
X = #todo{}.
% #todo{status = reminder, who = joe, text = undefined}
X1 = #todo{status = urgent, text = "Fix errata in book"}.
% #todo{status = urgent, who = joe, text = "Fix errata in book"}
X2 = X1#todo{status = done}.
% #todo{status = done, who = joe, text = "Fix errata in book"}

% `case` expressions.
% `filter` returns a list of all elements `X` in a list `L` for which `P(X)` is
% true.
filter(P, [H|T]) ->
  case P(H) of
    true -> [H|filter(P, T)];
    false -> filter(P, T)
  end;
filter(P, []) -> [].
filter(fun(X) -> X rem 2 == 0 end, [1, 2, 3, 4]). % [2, 4]

% `if` expressions.
max(X, Y) ->
  if
    X > Y -> X;
    X < Y -> Y;
    true -> nil
  end.

% Warning: at least one of the guards in the `if` expression must evaluate to
% `true`; otherwise, an exception will be raised.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 3. Exceptions.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Exceptions are raised by the system when internal errors are encountered or
% explicitly in code by calling `throw(Exception)`, `exit(Exception)`, or
% `erlang:error(Exception)`.
generate_exception(1) -> a;
generate_exception(2) -> throw(a);
generate_exception(3) -> exit(a);
generate_exception(4) -> {'EXIT', a};
generate_exception(5) -> erlang:error(a).

% Erlang has two methods of catching an exception. One is to enclose the call to
% the function that raises the exception within a `try...catch` expression.
catcher(N) ->
  try generate_exception(N) of
    Val -> {N, normal, Val}
  catch
    throw:X -> {N, caught, thrown, X};
    exit:X -> {N, caught, exited, X};
    error:X -> {N, caught, error, X}
  end.

% The other is to enclose the call in a `catch` expression. When you catch an
% exception, it is converted into a tuple that describes the error.
catcher(N) -> catch generate_exception(N).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 4. Concurrency
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Erlang relies on the actor model for concurrency. All we need to write
% concurrent programs in Erlang are three primitives: spawning processes,
% sending messages and receiving messages.

% To start a new process, we use the `spawn` function, which takes a function
% as argument.

F = fun() -> 2 + 2 end. % #Fun<erl_eval.20.67289768>
spawn(F). % <0.44.0>

% `spawn` returns a pid (process identifier); you can use this pid to send
% messages to the process. To do message passing, we use the `!` operator.
% For all of this to be useful, we need to be able to receive messages. This is
% achieved with the `receive` mechanism:
```

````erlang
-module(calculateGeometry).
-compile(export_all).
calculateArea() ->
    receive
      {rectangle, W, H} ->
        W * H;
      {circle, R} ->
        3.14 * R * R;
      _ ->
        io:format("We can only calculate area of rectangles or circles.")
    end.

% Compile the module and create a process that evaluates `calculateArea` in the
% shell.
c(calculateGeometry).
CalculateArea = spawn(calculateGeometry, calculateArea, []).
CalculateArea ! {circle, 2}. % 12.56000000000000049738

% The shell is also a process; you can use `self` to get the current pid.
self(). % <0.41.0>

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 5. Testing with EUnit
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Unit tests can be written using EUnits's test generators and assert macros
-module(fib).
-export([fib/1]).
-include_lib("eunit/include/eunit.hrl").

fib(0) -> 1;
fib(1) -> 1;
fib(N) when N > 1 -> fib(N-1) + fib(N-2).

fib_test_() ->
    [?_assert(fib(0) =:= 1),
     ?_assert(fib(1) =:= 1),
     ?_assert(fib(2) =:= 2),
     ?_assert(fib(3) =:= 3),
     ?_assert(fib(4) =:= 5),
     ?_assert(fib(5) =:= 8),
     ?_assertException(error, function_clause, fib(-1)),
     ?_assert(fib(31) =:= 2178309)
    ].

% EUnit will automatically export to a test() function to allow running the tests
% in the erlang shell
fib:test()

% The popular erlang build tool Rebar is also compatible with EUnit
% ```
% rebar eunit
% ```
````

References

- "Learn You Some Erlang for great good!"
- "Programming Erlang: Software for a Concurrent World" by Joe Armstrong
- Erlang/OTP Reference Documentation
- Erlang - Programming Rules and Conventions

Factor

Factor is a modern stack-based language, based on Forth, created by Slava Pestov.

Code in this file can be typed into Factor, but not directly imported because the vocabulary and import header would make the beginning thoroughly confusing.

```
! This is a comment

! Like Forth, all programming is done by manipulating the stack.
! Stating a literal value pushes it onto the stack.
5 2 3 56 76 23 65    ! No output, but stack is printed out in interactive mode

! Those numbers get added to the stack, from left to right.
! .s prints out the stack non-destructively.
.s     ! 5 2 3 56 76 23 65

! Arithmetic works by manipulating data on the stack.
5 4 +    ! No output

! `.` pops the top result from the stack and prints it.
.    ! 9

! More examples of arithmetic:
6 7 * .        ! 42
1360 23 - .    ! 1337
12 12 / .      ! 1
13 2 mod .     ! 1

99 neg .       ! -99
-99 abs .      ! 99
52 23 max .    ! 52
52 23 min .    ! 23

! A number of words are provided to manipulate the stack, collectively known as shuffle words.

3 dup -          ! duplicate the top item (1st now equals 2nd): 3 - 3
2 5 swap /       ! swap the top with the second element:        5 / 2
4 0 drop 2 /     ! remove the top item (dont print to screen):  4 / 2
1 2 3 nip .s     ! remove the second item (similar to drop):    1 3
1 2 clear .s     ! wipe out the entire stack
1 2 3 4 over .s  ! duplicate the second item to the top: 1 2 3 4 3
1 2 3 4 2 pick .s ! duplicate the third item to the top: 1 2 3 4 2 3

! Creating Words
! The `:` word sets Factor into compile mode until it sees the `;` word.
: square ( n -- n ) dup * ;    ! No output
```

```
5 square .                     ! 25

! We can view what a word does too.
! \ suppresses evaluation of a word and pushes its identifier on the stack instead.
\ square see    ! : square ( n -- n ) dup * ;

! After the name of the word to create, the declaration between brackets gives the stack effect.
! We can use whatever names we like inside the declaration:
: weirdsquare ( camel -- llama ) dup * ;

! Provided their count matches the word's stack effect:
: doubledup ( a -- b ) dup dup ; ! Error: Stack effect declaration is wrong
: doubledup ( a -- a a a ) dup dup ; ! Ok
: weirddoubledup ( i -- am a fish ) dup dup ; ! Also Ok

! Where Factor differs from Forth is in the use of quotations.
! A quotation is a block of code that is pushed on the stack as a value.
! [ starts quotation mode; ] ends it.
[ 2 + ]       ! Quotation that adds 2 is left on the stack
4 swap call . ! 6

! And thus, higher order words. TONS of higher order words.
2 3 [ 2 + ] dip .s      ! Pop top stack value, run quotation, push it back: 4 3
3 4 [ + ] keep .s       ! Copy top stack value, run quotation, push the copy: 7 4
1 [ 2 + ] [ 3 + ] bi .s ! Run each quotation on the top value, push both results: 3 4
4 3 1 [ + ] [ + ] bi .s ! Quotations in a bi can pull values from deeper on the stack: 4 5 ( 1+3 1+4 )
1 2 [ 2 + ] bi@ .s      ! Run the quotation on first and second values
2 [ + ] curry      ! Inject the given value at the start of the quotation: [ 2 + ] is left on the stack

! Conditionals
! Any value is true except the built-in value f.
! A built-in value t does exist, but its use isn't essential.
! Conditionals are higher order words as with the combinators above.

5 [ "Five is true" . ] when                      ! Five is true
0 [ "Zero is true" . ] when                      ! Zero is true
f [ "F is true" . ] when                         ! No output
f [ "F is false" . ] unless                      ! F is false
2 [ "Two is true" . ] [ "Two is false" . ] if   ! Two is true

! By default the conditionals consume the value under test, but starred variants
! leave it alone if it's true:

5 [ . ] when*      ! 5
f [ . ] when*      ! No output, empty stack, f is consumed because it's false


! Loops
! You've guessed it.. these are higher order words too.

5 [ . ] each-integer               ! 0 1 2 3 4
4 3 2 1 0 5 [ + . ] each-integer   ! 0 2 4 6 8
5 [ "Hello" . ] times              ! Hello Hello Hello Hello Hello
```

```
! Here's a list:
{ 2 4 6 8 }                        ! Goes on the stack as one item

! Loop through the list:
{ 2 4 6 8 } [ 1 + . ] each          ! Prints 3 5 7 9
{ 2 4 6 8 } [ 1 + ] map             ! Leaves { 3 5 7 9 } on stack

! Loop reducing or building lists:
{ 1 2 3 4 5 } [ 2 mod 0 = ] filter ! Keeps only list members for which quotation yields true: { 2 4 }
{ 2 4 6 8 } 0 [ + ] reduce .      ! Like "fold" in functional languages: prints 20 (0+2+4+6+8)
{ 2 4 6 8 } 0 [ + ] accumulate . . ! Like reduce but keeps the intermediate values in a list: prints { 0 2 6 12 }
1 5 [ 2 * dup ] replicate .      ! Loops the quotation 5 times and collects the results in a list: { 2 4 8 16 32
1 [ dup 100 < ] [ 2 * dup ] produce ! Loops the second quotation until the first returns false and collects the

! If all else fails, a general purpose while loop:
1 [ dup 10 < ] [ "Hello" . 1 + ] while  ! Prints "Hello" 10 times
                                        ! Yes, it's hard to read
                                        ! That's what all those variant loops are for

! Variables
! Usually Factor programs are expected to keep all data on the stack.
! Using named variables makes refactoring harder (and it's called Factor for a reason)
! Global variables, if you must:

SYMBOL: name            ! Creates name as an identifying word
"Bob" name set-global   ! No output
name get-global .       ! "Bob"

! Named local variables are considered an extension but are available
! In a quotation..
[| m n                  ! Quotation captures top two stack values into m and n
 | m n + ]              ! Read them

! Or in a word..
:: lword ( -- )           ! Note double colon to invoke lexical variable extension
   2 :> c                 ! Declares immutable variable c to hold 2
   c . ;                  ! Print it out

! In a word declared this way,  the input side of the stack declaration
! becomes meaningful and gives the variable names stack values are captured into
:: double ( a -- result ) a 2 * ;

! Variables are declared mutable by ending their name with a shriek
:: mword2 ( a! -- x y )   ! Capture top of stack in mutable variable a
   a                      ! Push a
   a 2 * a!               ! Multiply a by 2 and store result back in a
   a ;                    ! Push new value of a
5 mword2                  ! Stack: 5 10

! Lists and Sequences
! We saw above how to push a list onto the stack

0 { 1 2 3 4 } nth         ! Access a particular member of a list: 1
10 { 1 2 3 4 } nth        ! Error: sequence index out of bounds
```

```
1 { 1 2 3 4 } ?nth        ! Same as nth if index is in bounds: 2
10 { 1 2 3 4 } ?nth       ! No error if out of bounds: f

{ "at" "the" "beginning" } "Append" prefix    ! { "Append" "at" "the" "beginning" }
{ "Append" "at" "the" } "end" suffix          ! { "Append" "at" "the" "end" }
"in" 1 { "Insert" "the" "middle" } insert-nth ! { "Insert" "in" "the" "middle" }
"Concat" "enate" append                       ! "Concatenate" - strings are sequences too
"Concatenate" "Reverse " prepend              ! "Reverse Concatenate"
{ "Concatenate " "seq " "of " "seqs" } concat ! "Concatenate seq of seqs"
{ "Connect" "subseqs" "with" "separators" } " " join  ! "Connect subseqs with separators"

! And if you want to get meta, quotations are sequences and can be dismantled..
0 [ 2 + ] nth                              ! 2
1 [ 2 + ] nth                              ! +
[ 2 + ] \ - suffix                         ! Quotation [ 2 + - ]
```

Ready For More?

- Factor Documentation

Forth

Forth was created by Charles H. Moore in the 70s. It is an imperative, stack-based language and programming environment, being used in projects such as Open Firmware. It's also used by NASA.

Note: This article focuses predominantly on the Gforth implementation of Forth, but most of what is written here should work elsewhere.

```
\ This is a comment
( This is also a comment but it's only used when defining words )

\ --------------------------------- Precursor ----------------------------------

\ All programming in Forth is done by manipulating the parameter stack (more
\ commonly just referred to as "the stack").
5 2 3 56 76 23 65    \ ok

\ Those numbers get added to the stack, from left to right.
.s    \ <7> 5 2 3 56 76 23 65 ok

\ In Forth, everything is either a word or a number.

\ ------------------------------ Basic Arithmetic ------------------------------

\ Arithmetic (in fact most words requiring data) works by manipulating data on
\ the stack.
5 4 +    \ ok

\ `.` pops the top result from the stack:
.    \ 9 ok

\ More examples of arithmetic:
```

```
6 7 * .        \ 42 ok
1360 23 - .    \ 1337 ok
12 12 / .      \ 1 ok
13 2 mod .     \ 1 ok

99 negate .    \ -99 ok
-99 abs .      \ 99 ok
52 23 max .    \ 52 ok
52 23 min .    \ 23 ok

\ ----------------------------- Stack Manipulation ----------------------------

\ Naturally, as we work with the stack, we'll want some useful methods:

3 dup -          \ duplicate the top item (1st now equals 2nd): 3 - 3
2 5 swap /       \ swap the top with the second element:        5 / 2
6 4 5 rot .s     \ rotate the top 3 elements:                   4 5 6
4 0 drop 2 /     \ remove the top item (dont print to screen):  4 / 2
1 2 3 nip .s     \ remove the second item (similar to drop):    1 3

\ ---------------------- More Advanced Stack Manipulation ----------------------

1 2 3 4 tuck   \ duplicate the top item into the second slot:      1 2 4 3 4 ok
1 2 3 4 over   \ duplicate the second item to the top:             1 2 3 4 3 ok
1 2 3 4 2 roll \ *move* the item at that position to the top:      1 3 4 2 ok
1 2 3 4 2 pick \ *duplicate* the item at that position to the top: 1 2 3 4 2 ok

\ When referring to stack indexes, they are zero-based.

\ ------------------------------ Creating Words --------------------------------

\ The `:` word sets Forth into compile mode until it sees the `;` word.
: square ( n -- n ) dup * ;    \ ok
5 square .                     \ 25 ok

\ We can view what a word does too:
see square     \ : square dup * ; ok

\ -------------------------------- Conditionals --------------------------------

\ -1 == true, 0 == false. However, any non-zero value is usually treated as
\ being true:
42 42 =    \ -1 ok
12 53 =    \ 0 ok

\ `if` is a compile-only word. `if` <stuff to do> `then` <rest of program>.
: ?>64 ( n -- n ) dup 64 > if ." Greater than 64!" then ; \ ok
100 ?>64                                                  \ Greater than 64! ok

\ Else:
: ?>64 ( n -- n ) dup 64 > if ." Greater than 64!" else ." Less than 64!" then ;
100 ?>64    \ Greater than 64! ok
20 ?>64     \ Less than 64! ok
```

```
\ ------------------------------------ Loops -----------------------------------

\ `do` is also a compile-only word.
: myloop ( -- ) 5 0 do cr ." Hello!" loop ; \ ok
myloop
\ Hello!
\ Hello!
\ Hello!
\ Hello!
\ Hello! ok

\ `do` expects two numbers on the stack: the end number and the start number.

\ We can get the value of the index as we loop with `i`:
: one-to-12 ( -- ) 12 0 do i . loop ;     \ ok
one-to-12                                 \ 0 1 2 3 4 5 6 7 8 9 10 11 12 ok

\ `?do` works similarly, except it will skip the loop if the end and start
\ numbers are equal.
: squares ( n -- ) 0 ?do i square . loop ;   \ ok
10 squares                                   \ 0 1 4 9 16 25 36 49 64 81 ok

\ Change the "step" with `+loop`:
: threes ( n n -- ) ?do i . 3 +loop ;    \ ok
15 0 threes                             \ 0 3 6 9 12 ok

\ Indefinite loops with `begin` <stuff to do> <flag> `until`:
: death ( -- ) begin ." Are we there yet?" 0 until ;    \ ok

\ ---------------------------- Variables and Memory ----------------------------

\ Use `variable` to declare `age` to be a variable.
variable age    \ ok

\ Then we write 21 to age with the word `!`.
21 age !    \ ok

\ Finally we can print our variable using the "read" word `@`, which adds the
\ value to the stack, or use `?` that reads and prints it in one go.
age @ .    \ 21 ok
age ?      \ 21 ok

\ Constants are quite similar, except we don't bother with memory addresses:
100 constant WATER-BOILING-POINT    \ ok
WATER-BOILING-POINT .               \ 100 ok

\ ----------------------------------- Arrays -----------------------------------

\ Creating arrays is similar to variables, except we need to allocate more
\ memory to them.

\ You can use `2 cells allot` to create an array that's 3 cells long:
variable mynumbers 2 cells allot    \ ok
```

```
\ Initialize all the values to 0
mynumbers 3 cells erase    \ ok

\ Alternatively we could use `fill`:
mynumbers 3 cells 0 fill

\ or we can just skip all the above and initialize with specific values:
create mynumbers 64 , 9001 , 1337 , \ ok (the last `,` is important!)

\ ...which is equivalent to:

\ Manually writing values to each index:
64 mynumbers 0 cells + !      \ ok
9001 mynumbers 1 cells + !    \ ok
1337 mynumbers 2 cells + !    \ ok

\ Reading values at certain array indexes:
0 cells mynumbers + ?    \ 64 ok
1 cells mynumbers + ?    \ 9001 ok

\ We can simplify it a little by making a helper word for manipulating arrays:
: of-arr ( n n -- n ) cells + ;    \ ok
mynumbers 2 of-arr ?               \ 1337 ok

\ Which we can use for writing too:
20 mynumbers 1 of-arr !    \ ok
mynumbers 1 of-arr ?       \ 20 ok

\ ------------------------------ The Return Stack ------------------------------

\ The return stack is used to the hold pointers to things when words are
\ executing other words, e.g. loops.

\ We've already seen one use of it: `i`, which duplicates the top of the return
\ stack. `i` is equivalent to `r@`.
: myloop ( -- ) 5 0 do r@ . loop ;    \ ok

\ As well as reading, we can add to the return stack and remove from it:
5 6 4 >r swap r> .s    \ 6 5 4 ok

\ NOTE: Because Forth uses the return stack for word pointers,  `>r` should
\ always be followed by `r>`.

\ ------------------------- Floating Point Operations --------------------------

\ Most Forths tend to eschew the use of floating point operations.
8.3e 0.8e f+ f.    \ 9.1 ok

\ Usually we simply prepend words with 'f' when dealing with floats:
variable myfloatingvar     \ ok
4.4e myfloatingvar f!      \ ok
myfloatingvar f@ f.        \ 4.4 ok

\ --------------------------------- Final Notes --------------------------------
\ Typing a non-existent word will empty the stack. However, there's also a word
\ specifically for that:
clearstack

\ Clear the screen:
page

\ Loading Forth files:
\ s" forthfile.fs" included

\ You can list every word that's in Forth's dictionary (but it's a huge list!):
\ words

\ Exiting Gforth:
\ bye
```

Ready For More?

- Starting Forth
- Simple Forth
- Thinking Forth

Fsharp

F# is a general purpose functional/OO programming language. It's free and open source, and runs on Linux, Mac, Windows and more.

It has a powerful type system that traps many errors at compile time, but it uses type inference so that it reads more like a dynamic language.

The syntax of F# is different from C-style languages:

- Curly braces are not used to delimit blocks of code. Instead, indentation is used (like Python).
- Whitespace is used to separate parameters rather than commas.

If you want to try out the code below, you can go to tryfsharp.org and paste it into an interactive REPL.

```
// single line comments use a double slash
(* multi line comments use (* . . . *) pair

-end of multi line comment- *)

// ================================================
// Basic Syntax
// ================================================

// ------ "Variables" (but not really) ------
// The "let" keyword defines an (immutable) value
let myInt = 5
let myFloat = 3.14
let myString = "hello"           // note that no types needed

// ------ Lists ------
```

```
let twoToFive = [2; 3; 4; 5]        // Square brackets create a list with
                                 // semicolon delimiters.
let oneToFive = 1 :: twoToFive   // :: creates list with new 1st element
// The result is [1; 2; 3; 4; 5]
let zeroToFive = [0; 1] @ twoToFive   // @ concats two lists

// IMPORTANT: commas are never used as delimiters, only semicolons!

// ------ Functions ------
// The "let" keyword also defines a named function.
let square x = x * x          // Note that no parens are used.
square 3                      // Now run the function. Again, no parens.

let add x y = x + y           // don't use add (x,y)! It means something
                              // completely different.
add 2 3                       // Now run the function.

// to define a multiline function, just use indents. No semicolons needed.
let evens list =
   let isEven x = x % 2 = 0     // Define "isEven" as a sub function
   List.filter isEven list    // List.filter is a library function
                              // with two parameters: a boolean function
                              // and a list to work on

evens oneToFive               // Now run the function

// You can use parens to clarify precedence. In this example,
// do "map" first, with two args, then do "sum" on the result.
// Without the parens, "List.map" would be passed as an arg to List.sum
let sumOfSquaresTo100 =
   List.sum ( List.map square [1..100] )

// You can pipe the output of one operation to the next using "|>"
// Piping data around is very common in F#, similar to UNIX pipes.

// Here is the same sumOfSquares function written using pipes
let sumOfSquaresTo100piped =
   [1..100] |> List.map square |> List.sum  // "square" was defined earlier

// you can define lambdas (anonymous functions) using the "fun" keyword
let sumOfSquaresTo100withFun =
   [1..100] |> List.map (fun x -> x * x) |> List.sum

// In F# there is no "return" keyword. A function always
// returns the value of the last expression used.

// ------ Pattern Matching ------
// Match..with.. is a supercharged case/switch statement.
let simplePatternMatch =
   let x = "a"
   match x with
    | "a" -> printfn "x is a"
    | "b" -> printfn "x is b"
    | _ -> printfn "x is something else"   // underscore matches anything
```

```
// F# doesn't allow nulls by default -- you must use an Option type
// and then pattern match.
// Some(..) and None are roughly analogous to Nullable wrappers
let validValue = Some(99)
let invalidValue = None

// In this example, match..with matches the "Some" and the "None",
// and also unpacks the value in the "Some" at the same time.
let optionPatternMatch input =
   match input with
    | Some i -> printfn "input is an int=%d" i
    | None -> printfn "input is missing"

optionPatternMatch validValue
optionPatternMatch invalidValue

// ------ Printing ------
// The printf/printfn functions are similar to the
// Console.Write/WriteLine functions in C#.
printfn "Printing an int %i, a float %f, a bool %b" 1 2.0 true
printfn "A string %s, and something generic %A" "hello" [1; 2; 3; 4]

// There are also sprintf/sprintfn functions for formatting data
// into a string, similar to String.Format in C#.

// ================================================
// More on functions
// ================================================

// F# is a true functional language -- functions are first
// class entities and can be combined easily to make powerful
// constructs

// Modules are used to group functions together
// Indentation is needed for each nested module.
module FunctionExamples =

    // define a simple adding function
    let add x y = x + y

    // basic usage of a function
    let a = add 1 2
    printfn "1 + 2 = %i" a

    // partial application to "bake in" parameters
    let add42 = add 42
    let b = add42 1
    printfn "42 + 1 = %i" b

    // composition to combine functions
    let add1 = add 1
    let add2 = add 2
    let add3 = add1 >> add2
    let c = add3 7
    printfn "3 + 7 = %i" c

    // higher order functions
    [1..10] |> List.map add3 |> printfn "new list is %A"

    // lists of functions, and more
    let add6 = [add1; add2; add3] |> List.reduce (>>)
    let d = add6 7
    printfn "1 + 2 + 3 + 7 = %i" d

// ================================================
// Lists and collection
// ================================================

// There are three types of ordered collection:
// * Lists are most basic immutable collection.
// * Arrays are mutable and more efficient when needed.
// * Sequences are lazy and infinite (e.g. an enumerator).
//
// Other collections include immutable maps and sets
// plus all the standard .NET collections

module ListExamples =

    // lists use square brackets
    let list1 = ["a"; "b"]
    let list2 = "c" :: list1    // :: is prepending
    let list3 = list1 @ list2   // @ is concat

    // list comprehensions (aka generators)
    let squares = [for i in 1..10 do yield i * i]

    // prime number generator
    let rec sieve = function
        | (p::xs) -> p :: sieve [ for x in xs do if x % p > 0 then yield x ]
        | []      -> []
    let primes = sieve [2..50]
    printfn "%A" primes

    // pattern matching for lists
    let listMatcher aList =
        match aList with
        | [] -> printfn "the list is empty"
        | [first] -> printfn "the list has one element %A " first
        | [first; second] -> printfn "list is %A and %A" first second
        | _ -> printfn "the list has more than two elements"

    listMatcher [1; 2; 3; 4]
    listMatcher [1; 2]
    listMatcher [1]
    listMatcher []

    // recursion using lists
    let rec sum aList =
        match aList with
        | [] -> 0
        | x::xs -> x + sum xs
    sum [1..10]

    // -----------------------------------------
    // Standard library functions
    // -----------------------------------------

    // map
    let add3 x = x + 3
    [1..10] |> List.map add3

    // filter
    let even x = x % 2 = 0
    [1..10] |> List.filter even

    // many more -- see documentation

module ArrayExamples =

    // arrays use square brackets with bar
    let array1 = [| "a"; "b" |]
    let first = array1.[0]        // indexed access using dot

    // pattern matching for arrays is same as for lists
    let arrayMatcher aList =
        match aList with
        | [| |] -> printfn "the array is empty"
        | [| first |] -> printfn "the array has one element %A " first
        | [| first; second |] -> printfn "array is %A and %A" first second
        | _ -> printfn "the array has more than two elements"

    arrayMatcher [| 1; 2; 3; 4 |]

    // Standard library functions just as for List

    [| 1..10 |]
    |> Array.map (fun i -> i + 3)
    |> Array.filter (fun i -> i % 2 = 0)
    |> Array.iter (printfn "value is %i. ")


module SequenceExamples =

    // sequences use curly braces
    let seq1 = seq { yield "a"; yield "b" }

    // sequences can use yield and
    // can contain subsequences
    let strange = seq {
        // "yield" adds one element
        yield 1; yield 2;
        // "yield!" adds a whole subsequence
        yield! [5..10]
        yield! seq {
            for i in 1..10 do
              if i % 2 = 0 then yield i }}
    // test
    strange |> Seq.toList


    // Sequences can be created using "unfold"
    // Here's the fibonacci series
    let fib = Seq.unfold (fun (fst,snd) ->
        Some(fst + snd, (snd, fst + snd))) (0,1)

    // test
    let fib10 = fib |> Seq.take 10 |> Seq.toList
    printf "first 10 fibs are %A" fib10


// ================================================
// Data Types
// ================================================

module DataTypeExamples =

    // All data is immutable by default

    // Tuples are quick 'n easy anonymous types
    // -- Use a comma to create a tuple
    let twoTuple = 1, 2
    let threeTuple = "a", 2, true

    // Pattern match to unpack
    let x, y = twoTuple  // sets x = 1, y = 2

    // ------------------------------------
    // Record types have named fields
    // ------------------------------------

    // Use "type" with curly braces to define a record type
    type Person = {First:string; Last:string}

    // Use "let" with curly braces to create a record
    let person1 = {First="John"; Last="Doe"}

    // Pattern match to unpack
    let {First = first} = person1    // sets first="John"

    // ------------------------------------
    // Union types (aka variants) have a set of choices
    // Only case can be valid at a time.
    // ------------------------------------
    // Use "type" with bar/pipe to define a union type
    type Temp =
        | DegreesC of float
        | DegreesF of float

    // Use one of the cases to create one
    let temp1 = DegreesF 98.6
    let temp2 = DegreesC 37.0

    // Pattern match on all cases to unpack
    let printTemp = function
       | DegreesC t -> printfn "%f degC" t
       | DegreesF t -> printfn "%f degF" t

    printTemp temp1
    printTemp temp2

    // ------------------------------------
    // Recursive types
    // ------------------------------------

    // Types can be combined recursively in complex ways
    // without having to create subclasses
    type Employee =
      | Worker of Person
      | Manager of Employee list

    let jdoe = {First="John"; Last="Doe"}
    let worker = Worker jdoe

    // ------------------------------------
    // Modeling with types
    // ------------------------------------

    // Union types are great for modeling state without using flags
    type EmailAddress =
        | ValidEmailAddress of string
        | InvalidEmailAddress of string

    let trySendEmail email =
        match email with // use pattern matching
        | ValidEmailAddress address -> ()   // send
        | InvalidEmailAddress address -> () // dont send

    // The combination of union types and record types together
    // provide a great foundation for domain driven design.
    // You can create hundreds of little types that accurately
    // reflect the domain.

    type CartItem = { ProductCode: string; Qty: int }
    type Payment = Payment of float
    type ActiveCartData = { UnpaidItems: CartItem list }
    type PaidCartData = { PaidItems: CartItem list; Payment: Payment}
```

```
    type ShoppingCart =
        | EmptyCart  // no data
        | ActiveCart of ActiveCartData
        | PaidCart of PaidCartData

    // ------------------------------------
    // Built in behavior for types
    // ------------------------------------

    // Core types have useful "out-of-the-box" behavior, no coding needed.
    // * Immutability
    // * Pretty printing when debugging
    // * Equality and comparison
    // * Serialization

    // Pretty printing using %A
    printfn "twoTuple=%A,\nPerson=%A,\nTemp=%A,\nEmployee=%A"
             twoTuple person1 temp1 worker

    // Equality and comparison built in.
    // Here's an example with cards.
    type Suit = Club | Diamond | Spade | Heart
    type Rank = Two | Three | Four | Five | Six | Seven | Eight
                | Nine | Ten | Jack | Queen | King | Ace

    let hand = [ Club, Ace; Heart, Three; Heart, Ace;
                 Spade, Jack; Diamond, Two; Diamond, Ace ]

    // sorting
    List.sort hand |> printfn "sorted hand is (low to high) %A"
    List.max hand |> printfn "high card is %A"
    List.min hand |> printfn "low card is %A"


// ================================================
// Active patterns
// ================================================

module ActivePatternExamples =

    // F# has a special type of pattern matching called "active patterns"
    // where the pattern can be parsed or detected dynamically.

    // "banana clips" are the syntax for active patterns

    // for example, define an "active" pattern to match character types...
    let (|Digit|Letter|Whitespace|Other|) ch =
       if System.Char.IsDigit(ch) then Digit
       else if System.Char.IsLetter(ch) then Letter
       else if System.Char.IsWhiteSpace(ch) then Whitespace
       else Other

    // ... and then use it to make parsing logic much clearer
    let printChar ch =
      match ch with
      | Digit -> printfn "%c is a Digit" ch
      | Letter -> printfn "%c is a Letter" ch
      | Whitespace -> printfn "%c is a Whitespace" ch
      | _ -> printfn "%c is something else" ch

    // print a list
    ['a'; 'b'; '1'; ' '; '-'; 'c'] |> List.iter printChar

    // -----------------------------------
    // FizzBuzz using active patterns
    // -----------------------------------

    // You can create partial matching patterns as well
    // Just use underscore in the defintion, and return Some if matched.
    let (|MultOf3|_|) i = if i % 3 = 0 then Some MultOf3 else None
    let (|MultOf5|_|) i = if i % 5 = 0 then Some MultOf5 else None

    // the main function
    let fizzBuzz i =
      match i with
      | MultOf3 & MultOf5 -> printf "FizzBuzz, "
      | MultOf3 -> printf "Fizz, "
      | MultOf5 -> printf "Buzz, "
      | _ -> printf "%i, " i

    // test
    [1..20] |> List.iter fizzBuzz

// ================================================
// Conciseness
// ================================================

module AlgorithmExamples =

    // F# has a high signal/noise ratio, so code reads
    // almost like the actual algorithm

    // ------ Example: define sumOfSquares function ------
    let sumOfSquares n =
       [1..n]              // 1) take all the numbers from 1 to n
       |> List.map square  // 2) square each one
       |> List.sum         // 3) sum the results

    // test
    sumOfSquares 100 |> printfn "Sum of squares = %A"

    // ------ Example: define a sort function ------
    let rec sort list =
       match list with
       // If the list is empty
       | [] ->
            []                            // return an empty list
       // If the list is not empty
       | firstElem::otherElements ->        // take the first element
            let smallerElements =           // extract the smaller elements
                otherElements               // from the remaining ones
                |> List.filter (fun e -> e < firstElem)
                |> sort                     // and sort them
            let largerElements =            // extract the larger ones
                otherElements               // from the remaining ones
                |> List.filter (fun e -> e >= firstElem)
                |> sort                     // and sort them
            // Combine the 3 parts into a new list and return it
            List.concat [smallerElements; [firstElem]; largerElements]

    // test
    sort [1; 5; 23; 18; 9; 1; 3] |> printfn "Sorted = %A"

// ================================================
// Asynchronous Code
// ================================================

module AsyncExample =

    // F# has built-in features to help with async code
    // without encountering the "pyramid of doom"
    //
    // The following example downloads a set of web pages in parallel.

    open System.Net
    open System
    open System.IO
    open Microsoft.FSharp.Control.CommonExtensions

    // Fetch the contents of a URL asynchronously
    let fetchUrlAsync url =
        async {   // "async" keyword and curly braces
                  // creates an "async" object
            let req = WebRequest.Create(Uri(url))
            use! resp = req.AsyncGetResponse()
                // use! is async assignment
            use stream = resp.GetResponseStream()
                // "use" triggers automatic close()
                // on resource at end of scope
            use reader = new IO.StreamReader(stream)
            let html = reader.ReadToEnd()
            printfn "finished downloading %s" url
            }

    // a list of sites to fetch
    let sites = ["http://www.bing.com";
                 "http://www.google.com";
                 "http://www.microsoft.com";
                 "http://www.amazon.com";
                 "http://www.yahoo.com"]

    // do it
    sites
    |> List.map fetchUrlAsync  // make a list of async tasks
    |> Async.Parallel          // set up the tasks to run in parallel
    |> Async.RunSynchronously  // start them off

// ================================================
// .NET compatibility
// ================================================

module NetCompatibilityExamples =

    // F# can do almost everything C# can do, and it integrates
    // seamlessly with .NET or Mono libraries.

    // ------- work with existing library functions  -------

    let (i1success, i1) = System.Int32.TryParse("123");
    if i1success then printfn "parsed as %i" i1 else printfn "parse failed"

    // ------- Implement interfaces on the fly! -------

    // create a new object that implements IDisposable
    let makeResource name =
       { new System.IDisposable
         with member this.Dispose() = printfn "%s disposed" name }

    let useAndDisposeResources =
        use r1 = makeResource "first resource"
        printfn "using first resource"
        for i in [1..3] do
            let resourceName = sprintf "\tinner resource %d" i
            use temp = makeResource resourceName
            printfn "\tdo something with %s" resourceName
        use r2 = makeResource "second resource"
        printfn "using second resource"
        printfn "done."

    // ------- Object oriented code -------

    // F# is also a fully fledged OO language.
    // It supports classes, inheritance, virtual methods, etc.

    // interface with generic type
    type IEnumerator<'a> =
        abstract member Current : 'a
        abstract MoveNext : unit -> bool

    // abstract base class with virtual methods
    [<AbstractClass>]
    type Shape() =
        // readonly properties
        abstract member Width : int with get
        abstract member Height : int with get
        // non-virtual method
        member this.BoundingArea = this.Height * this.Width
        // virtual method with base implementation
        abstract member Print : unit -> unit
        default this.Print () = printfn "I'm a shape"

    // concrete class that inherits from base class and overrides
    type Rectangle(x:int, y:int) =
        inherit Shape()
        override this.Width = x
        override this.Height = y
        override this.Print ()  = printfn "I'm a Rectangle"

    // test
    let r = Rectangle(2, 3)
    printfn "The width is %i" r.Width
    printfn "The area is %i" r.BoundingArea
    r.Print()

    // ------- extension methods  -------

    // Just as in C#, F# can extend existing classes with extension methods.
    type System.String with
       member this.StartsWithA = this.StartsWith "A"

    // test
    let s = "Alice"
    printfn "'%s' starts with an 'A' = %A" s s.StartsWithA

    // ------- events  -------

    type MyButton() =
        let clickEvent = new Event<_>()

        [<CLIEvent>]
        member this.OnClick = clickEvent.Publish

        member this.TestEvent(arg) =
            clickEvent.Trigger(this, arg)

    // test
    let myButton = new MyButton()
    myButton.OnClick.Add(fun (sender, arg) ->
            printfn "Click event with arg=%O" arg)

    myButton.TestEvent("Hello World!")
```

More Information

For more demonstrations of F#, go to the Try F# site, or my why use F# series.

Read more about F# at fsharp.org.

Git

Git is a distributed version control and source code management system.

It does this through a series of snapshots of your project, and it works with those snapshots to provide you with functionality to version and manage your source code.

Versioning Concepts

What is version control?

Version control is a system that records changes to a file(s), over time.

Centralized Versioning VS Distributed Versioning

- Centralized version control focuses on synchronizing, tracking, and backing up files.
- Distributed version control focuses on sharing changes. Every change has a unique id.
- Distributed systems have no defined structure. You could easily have a SVN style, centralized system, with git.

Additional Information

Why Use Git?

- Can work offline.
- Collaborating with others is easy!
- Branching is easy!
- Merging is easy!
- Git is fast.
- Git is flexible.

Git Architecture

Repository

A set of files, directories, historical records, commits, and heads. Imagine it as a source code data structure, with the attribute that each source code "element" gives you access to its revision history, among other things.

A git repository is comprised of the .git directory & working tree.

.git Directory (component of repository)

The .git directory contains all the configurations, logs, branches, HEAD, and more. Detailed List.

Working Tree (component of repository)

This is basically the directories and files in your repository. It is often referred to as your working directory.

Index (component of .git dir)

The Index is the staging area in git. It's basically a layer that separates your working tree from the Git repository. This gives developers more power over what gets sent to the Git repository.

Commit

A git commit is a snapshot of a set of changes, or manipulations to your Working Tree. For example, if you added 5 files, and removed 2 others, these changes will be contained in a commit (or snapshot). This commit can then be pushed to other repositories, or not!

Branch

A branch is essentially a pointer to the last commit you made. As you go on committing, this pointer will automatically update to point the latest commit.

Tag

A tag is a mark on specific point in history. Typically people use this functionality to mark release points (v1.0, and so on)

HEAD and head (component of .git dir)

HEAD is a pointer that points to the current branch. A repository only has 1 *active* HEAD. head is a pointer that points to any commit. A repository can have any number of heads.

Stages of Git

- Modified - Changes have been made to a file but file has not been committed to Git Database yet
- Staged - Marks a modified file to go into your next commit snapshot
- Committed - Files have been committed to the Git Database

Conceptual Resources

- Git For Computer Scientists
- Git For Designers

Commands

init

Create an empty Git repository. The Git repository's settings, stored information, and more is stored in a directory (a folder) named ".git".

```
$ git init
```

config

To configure settings. Whether it be for the repository, the system itself, or global configurations (global config file is `~/.gitconfig`).

```
# Print & Set Some Basic Config Variables (Global)
$ git config --global user.email "MyEmail@Zoho.com"
$ git config --global user.name "My Name"
```

Learn More About git config.

help

To give you quick access to an extremely detailed guide of each command. Or to just give you a quick reminder of some semantics.

```
# Quickly check available commands
$ git help

# Check all available commands
$ git help -a

# Command specific help - user manual
# git help <command_here>
$ git help add
$ git help commit
$ git help init
# or git <command_here> --help
$ git add --help
$ git commit --help
$ git init --help
```

ignore files

To intentionally untrack file(s) & folder(s) from git. Typically meant for private & temp files which would otherwise be shared in the repository.

```
$ echo "temp/" >> .gitignore
$ echo "private_key" >> .gitignore
```

status

To show differences between the index file (basically your working copy/repo) and the current HEAD commit.

```
# Will display the branch, untracked files, changes and other differences
$ git status

# To learn other "tid bits" about git status
$ git help status
```

add

To add files to the staging area/index. If you do not `git add` new files to the staging area/index, they will not be included in commits!

```
# add a file in your current working directory
$ git add HelloWorld.java

# add a file in a nested dir
$ git add /path/to/file/HelloWorld.c

# Regular Expression support!
$ git add ./*.java
```

This only adds a file to the staging area/index, it doesn't commit it to the working directory/repo.

branch

Manage your branches. You can view, edit, create, delete branches using this command.

```
# list existing branches & remotes
$ git branch -a

# create a new branch
$ git branch myNewBranch

# delete a branch
$ git branch -d myBranch

# rename a branch
# git branch -m <oldname> <newname>
$ git branch -m myBranchName myNewBranchName

# edit a branch's description
$ git branch myBranchName --edit-description
```

tag

Manage your tags

```
# List tags
$ git tag
# Create a annotated tag
# The -m specifies a tagging message,which is stored with the tag.
# If you don’t specify a message for an annotated tag,
# Git launches your editor so you can type it in.
$ git tag -a v2.0 -m 'my version 2.0'
# Show info about tag
# That shows the tagger information, the date the commit was tagged,
# and the annotation message before showing the commit information.
$ git show v2.0
# Push a single tag to remote
$ git push origin v2.0
# Push a lot of tags to remote
$ git push origin --tags
```

checkout

Updates all files in the working tree to match the version in the index, or specified tree.

```
# Checkout a repo - defaults to master branch
$ git checkout
# Checkout a specified branch
$ git checkout branchName
# Create a new branch & switch to it
# equivalent to "git branch <name>; git checkout <name>"
$ git checkout -b newBranch
```

clone

Clones, or copies, an existing repository into a new directory. It also adds remote-tracking branches for each branch in the cloned repo, which allows you to push to a remote branch.

```
# Clone learnxinyminutes-docs
$ git clone https://github.com/adambard/learnxinyminutes-docs.git
# shallow clone - faster cloning that pulls only latest snapshot
$ git clone --depth 1 https://github.com/adambard/learnxinyminutes-docs.git
# clone only a specific branch
$ git clone -b master-cn https://github.com/adambard/learnxinyminutes-docs.git --single-branch
```

commit

Stores the current contents of the index in a new "commit." This commit contains the changes made and a message created by the user.

```
# commit with a message
$ git commit -m "Added multiplyNumbers() function to HelloWorld.c"

# automatically stage modified or deleted files, except new files, and then commit
$ git commit -a -m "Modified foo.php and removed bar.php"

# change last commit (this deletes previous commit with a fresh commit)
$ git commit --amend -m "Correct message"
```

diff

Shows differences between a file in the working directory, index and commits.

```
# Show difference between your working dir and the index
$ git diff

# Show differences between the index and the most recent commit.
$ git diff --cached

# Show differences between your working dir and the most recent commit
$ git diff HEAD
```

grep

Allows you to quickly search a repository.

Optional Configurations:

```
# Thanks to Travis Jeffery for these
# Set line numbers to be shown in grep search results
$ git config --global grep.lineNumber true

# Make search results more readable, including grouping
$ git config --global alias.g "grep --break --heading --line-number"
```

```
# Search for "variableName" in all java files
$ git grep 'variableName' -- '*.java'

# Search for a line that contains "arrayListName" and, "add" or "remove"
$ git grep -e 'arrayListName' --and \( -e add -e remove \)
```

Google is your friend; for more examples Git Grep Ninja

log

Display commits to the repository.

```
# Show all commits
$ git log

# Show only commit message & ref
$ git log --oneline

# Show merge commits only
$ git log --merges

# Show all commits represented by an ASCII graph
$ git log --graph
```

merge

"Merge" in changes from external commits into the current branch.

```
# Merge the specified branch into the current.
$ git merge branchName

# Always generate a merge commit when merging
$ git merge --no-ff branchName
```

mv

Rename or move a file

```
# Renaming a file
$ git mv HelloWorld.c HelloNewWorld.c

# Moving a file
$ git mv HelloWorld.c ./new/path/HelloWorld.c

# Force rename or move
# "existingFile" already exists in the directory, will be overwritten
$ git mv -f myFile existingFile
```

pull

Pulls from a repository and merges it with another branch.

```
# Update your local repo, by merging in new changes
# from the remote "origin" and "master" branch.
# git pull <remote> <branch>
$ git pull origin master

# By default, git pull will update your current branch
# by merging in new changes from its remote-tracking branch
$ git pull

# Merge in changes from remote branch and rebase
# branch commits onto your local repo, like: "git pull <remote> <branch>, git rebase <branch>"
$ git pull origin master --rebase
```

push

Push and merge changes from a branch to a remote & branch.

```
# Push and merge changes from a local repo to a
# remote named "origin" and "master" branch.
# git push <remote> <branch>
$ git push origin master

# By default, git push will push and merge changes from
# the current branch to its remote-tracking branch
$ git push

# To link up current local branch with a remote branch, add -u flag:
$ git push -u origin master
# Now, anytime you want to push from that same local branch, use shortcut:
$ git push
```

stash

Stashing takes the dirty state of your working directory and saves it on a stack of unfinished changes that you can reapply at any time.

Let's say you've been doing some work in your git repo, but you want to pull from the remote. Since you have dirty (uncommited) changes to some files, you are not able to run `git pull`. Instead, you can run `git stash` to save your changes onto a stack!

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
  HEAD is now at 049d078 added the index file
  (To restore them type "git stash apply")
```

Now you can pull!

```
git pull
```

```
...changes apply...
```

Now check that everything is OK

```
$ git status
# On branch master
nothing to commit, working directory clean
```

You can see what "hunks" you've stashed so far using `git stash list`. Since the "hunks" are stored in a Last-In-First-Out stack, our most recent change will be at top.

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

Now let's apply our dirty changes back by popping them off the stack.

```
$ git stash pop
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   index.html
#      modified:   lib/simplegit.rb
#
```

`git stash apply` does the same thing

Now you're ready to get back to work on your stuff!

Additional Reading.

rebase (caution)

Take all changes that were committed on one branch, and replay them onto another branch. *Do not rebase commits that you have pushed to a public repo.*

```
# Rebase experimentBranch onto master
# git rebase <basebranch> <topicbranch>
$ git rebase master experimentBranch
```

Additional Reading.

reset (caution)

Reset the current HEAD to the specified state. This allows you to undo merges, pulls, commits, adds, and more. It's a great command but also dangerous if you don't know what you are doing.

```
# Reset the staging area, to match the latest commit (leaves dir unchanged)
$ git reset

# Reset the staging area, to match the latest commit, and overwrite working dir
$ git reset --hard

# Moves the current branch tip to the specified commit (leaves dir unchanged)
# all changes still exist in the directory.
$ git reset 31f2bb1

# Moves the current branch tip backward to the specified commit
# and makes the working dir match (deletes uncommited changes and all commits
# after the specified commit).
$ git reset --hard 31f2bb1
```

revert

Revert can be used to undo a commit. It should not be confused with reset which restores the state of a project to a previous point. Revert will add a new commit which is the inverse of the specified commit, thus reverting it.

```
# Revert a specified commit
$ git revert <commit>
```

rm

The opposite of git add, git rm removes files from the current working tree.

```
# remove HelloWorld.c
$ git rm HelloWorld.c

# Remove a file from a nested dir
$ git rm /pather/to/the/file/HelloWorld.c
```

Further Information

- tryGit - A fun interactive way to learn Git.
- Udemy Git Tutorial: A Comprehensive Guide
- Git Immersion - A Guided tour that walks through the fundamentals of git
- git-scm - Video Tutorials
- git-scm - Documentation
- Atlassian Git - Tutorials & Workflows
- SalesForce Cheat Sheet
- GitGuys
- Git - the simple guide
- Pro Git
- An introduction to Git and GitHub for Beginners (Tutorial)

Go

Go was created out of the need to get work done. It's not the latest trend in computer science, but it is the newest fastest way to solve real-world problems.

It has familiar concepts of imperative languages with static typing. It's fast to compile and fast to execute, it adds easy-to-understand concurrency to leverage today's multi-core CPUs, and has features to help with large-scale programming.

Go comes with a great standard library and an enthusiastic community.

```
// Single line comment
/* Multi-
 line comment */

// A package clause starts every source file.
// Main is a special name declaring an executable rather than a library.
```

```
package main

// Import declaration declares library packages referenced in this file.
import (
    "fmt"       // A package in the Go standard library.
    "io/ioutil" // Implements some I/O utility functions.
    m "math"    // Math library with local alias m.
    "net/http"  // Yes, a web server!
    "strconv"   // String conversions.
)

// A function definition. Main is special. It is the entry point for the
// executable program. Love it or hate it, Go uses brace brackets.
func main() {
    // Println outputs a line to stdout.
    // Qualify it with the package name, fmt.
    fmt.Println("Hello world!")

    // Call another function within this package.
    beyondHello()
}

// Functions have parameters in parentheses.
// If there are no parameters, empty parentheses are still required.
func beyondHello() {
    var x int // Variable declaration. Variables must be declared before use.
    x = 3     // Variable assignment.
    // "Short" declarations use := to infer the type, declare, and assign.
    y := 4
    sum, prod := learnMultiple(x, y)        // Function returns two values.
    fmt.Println("sum:", sum, "prod:", prod) // Simple output.
    learnTypes()                            // < y minutes, learn more!
}

/* <- multiline comment
Functions can have parameters and (multiple!) return values.
Here `x`, `y` are the arguments and `sum`, `prod` is the signature (what's returned).
Note that `x` and `sum` receive the type `int`.
*/
func learnMultiple(x, y int) (sum, prod int) {
    return x + y, x * y // Return two values.
}

// Some built-in types and literals.
func learnTypes() {
    // Short declaration usually gives you what you want.
    str := "Learn Go!" // string type.

    s2 := `A "raw" string literal
can include line breaks.` // Same string type.

    // Non-ASCII literal. Go source is UTF-8.
    g := 'Σ' // rune type, an alias for int32, holds a unicode code point.
    f := 3.14195 // float64, an IEEE-754 64-bit floating point number.
    c := 3 + 4i  // complex128, represented internally with two float64's.

    // var syntax with initializers.
    var u uint = 7 // Unsigned, but implementation dependent size as with int.
    var pi float32 = 22. / 7

    // Conversion syntax with a short declaration.
    n := byte('\n') // byte is an alias for uint8.

    // Arrays have size fixed at compile time.
    var a4 [4]int           // An array of 4 ints, initialized to all 0.
    a3 := [...]int{3, 1, 5} // An array initialized with a fixed size of three
    // elements, with values 3, 1, and 5.

    // Slices have dynamic size. Arrays and slices each have advantages
    // but use cases for slices are much more common.
    s3 := []int{4, 5, 9}    // Compare to a3. No ellipsis here.
    s4 := make([]int, 4)    // Allocates slice of 4 ints, initialized to all 0.
    var d2 [][]float64      // Declaration only, nothing allocated here.
    bs := []byte("a slice") // Type conversion syntax.

    // Because they are dynamic, slices can be appended to on-demand.
    // To append elements to a slice, the built-in append() function is used.
    // First argument is a slice to which we are appending. Commonly,
    // the array variable is updated in place, as in example below.
    s := []int{1, 2, 3}     // Result is a slice of length 3.
    s = append(s, 4, 5, 6)  // Added 3 elements. Slice now has length of 6.
    fmt.Println(s) // Updated slice is now [1 2 3 4 5 6]

    // To append another slice, instead of list of atomic elements we can
    // pass a reference to a slice or a slice literal like this, with a
    // trailing ellipsis, meaning take a slice and unpack its elements,
    // appending them to slice s.
    s = append(s, []int{7, 8, 9}...) // Second argument is a slice literal.
    fmt.Println(s)  // Updated slice is now [1 2 3 4 5 6 7 8 9]

    p, q := learnMemory() // Declares p, q to be type pointer to int.
    fmt.Println(*p, *q)   // * follows a pointer. This prints two ints.

    // Maps are a dynamically growable associative array type, like the
    // hash or dictionary types of some other languages.
    m := map[string]int{"three": 3, "four": 4}
    m["one"] = 1

    // Unused variables are an error in Go.
    // The underscore lets you "use" a variable but discard its value.
    _, _, _, _, _, _, _, _, _, _ = str, s2, g, f, u, pi, n, a3, s4, bs
    // Output of course counts as using a variable.
    fmt.Println(s, c, a4, s3, d2, m)

    learnFlowControl() // Back in the flow.
}
```

```
// It is possible, unlike in many other languages for functions in go
// to have named return values.
// Assigning a name to the type being returned in the function declaration line
// allows us to easily return from multiple points in a function as well as to
// only use the return keyword, without anything further.
func learnNamedReturns(x, y int) (z int) {
    z = x * y
    return // z is implicit here, because we named it earlier.
}

// Go is fully garbage collected. It has pointers but no pointer arithmetic.
// You can make a mistake with a nil pointer, but not by incrementing a pointer.
func learnMemory() (p, q *int) {
    // Named return values p and q have type pointer to int.
    p = new(int) // Built-in function new allocates memory.
    // The allocated int is initialized to 0, p is no longer nil.
    s := make([]int, 20) // Allocate 20 ints as a single block of memory.
    s[3] = 7             // Assign one of them.
    r := -2              // Declare another local variable.
    return &s[3], &r     // & takes the address of an object.
}

func expensiveComputation() float64 {
    return m.Exp(10)
}

func learnFlowControl() {
    // If statements require brace brackets, and do not require parentheses.
    if true {
        fmt.Println("told ya")
    }
    // Formatting is standardized by the command line command "go fmt."
    if false {
        // Pout.
    } else {
        // Gloat.
    }
    // Use switch in preference to chained if statements.
    x := 42.0
    switch x {
    case 0:
    case 1:
    case 42:
        // Cases don't "fall through".
        /*
        There is a `fallthrough` keyword however, see:
          https://github.com/golang/go/wiki/Switch#fall-through
        */
    case 43:
        // Unreached.
    default:
        // Default case is optional.
    }
    // Like if, for doesn't use parens either.
    // Variables declared in for and if are local to their scope.
    for x := 0; x < 3; x++ { // ++ is a statement.
        fmt.Println("iteration", x)
    }
    // x == 42 here.

    // For is the only loop statement in Go, but it has alternate forms.
    for { // Infinite loop.
        break    // Just kidding.
        continue // Unreached.
    }

    // You can use range to iterate over an array, a slice, a string, a map, or a channel.
    // range returns one (channel) or two values (array, slice, string and map).
    for key, value := range map[string]int{"one": 1, "two": 2, "three": 3} {
        // for each pair in the map, print key and value
        fmt.Printf("key=%s, value=%d\n", key, value)
    }

    // As with for, := in an if statement means to declare and assign
    // y first, then test y > x.
    if y := expensiveComputation(); y > x {
        x = y
    }
    // Function literals are closures.
    xBig := func() bool {
        return x > 10000 // References x declared above switch statement.
    }
    fmt.Println("xBig:", xBig()) // true (we last assigned e^10 to x).
    x = 1.3e3                    // This makes x == 1300
    fmt.Println("xBig:", xBig()) // false now.

    // What's more is function literals may be defined and called inline,
    // acting as an argument to function, as long as:
    // a) function literal is called immediately (),
    // b) result type matches expected type of argument.
    fmt.Println("Add + double two numbers: ",
        func(a, b int) int {
            return (a + b) * 2
        }(10, 2)) // Called with args 10 and 2
    // => Add + double two numbers: 24

    // When you need it, you'll love it.
    goto love
love:

    learnFunctionFactory() // func returning func is fun(3)(3)
    learnDefer()      // A quick detour to an important keyword.
    learnInterfaces() // Good stuff coming up!
}

func learnFunctionFactory() {
    // Next two are equivalent, with second being more practical
    fmt.Println(sentenceFactory("summer")("A beautiful", "day!"))
    d := sentenceFactory("summer")
    fmt.Println(d("A beautiful", "day!"))
    fmt.Println(d("A lazy", "afternoon!"))
}

// Decorators are common in other languages. Same can be done in Go
// with function literals that accept arguments.
func sentenceFactory(mystring string) func(before, after string) string {
    return func(before, after string) string {
        return fmt.Sprintf("%s %s %s", before, mystring, after) // new string
    }
}

func learnDefer() (ok bool) {
    // Deferred statements are executed just before the function returns.
    defer fmt.Println("deferred statements execute in reverse (LIFO) order.")
    defer fmt.Println("\nThis line is being printed first because")
    // Defer is commonly used to close a file, so the function closing the
    // file stays close to the function opening the file.
    return true
}

// Define Stringer as an interface type with one method, String.
type Stringer interface {
    String() string
}

// Define pair as a struct with two fields, ints named x and y.
type pair struct {
    x, y int
}

// Define a method on type pair. Pair now implements Stringer.
func (p pair) String() string { // p is called the "receiver"
    // Sprintf is another public function in package fmt.
    // Dot syntax references fields of p.
    return fmt.Sprintf("(%d, %d)", p.x, p.y)
}

func learnInterfaces() {
    // Brace syntax is a "struct literal". It evaluates to an initialized
    // struct. The := syntax declares and initializes p to this struct.
    p := pair{3, 4}
    fmt.Println(p.String()) // Call String method of p, of type pair.
    var i Stringer          // Declare i of interface type Stringer.
    i = p                   // Valid because pair implements Stringer
    // Call String method of i, of type Stringer. Output same as above.
    fmt.Println(i.String())

    // Functions in the fmt package call the String method to ask an object
    // for a printable representation of itself.
    fmt.Println(p) // Output same as above. Println calls String method.
    fmt.Println(i) // Output same as above.
    learnVariadicParams("great", "learning", "here!")
}

// Functions can have variadic parameters.
func learnVariadicParams(myStrings ...interface{}) {
    // Iterate each value of the variadic.
    // The underbar here is ignoring the index argument of the array.
    for _, param := range myStrings {
        fmt.Println("param:", param)
    }

    // Pass variadic value as a variadic parameter.
    fmt.Println("params:", fmt.Sprintln(myStrings...))

    learnErrorHandling()
}

func learnErrorHandling() {
    // ", ok" idiom used to tell if something worked or not.
    m := map[int]string{3: "three", 4: "four"}
    if x, ok := m[1]; !ok { // ok will be false because 1 is not in the map.
        fmt.Println("no one there")
    } else {
        fmt.Print(x) // x would be the value, if it were in the map.
    }
    // An error value communicates not just "ok" but more about the problem.
    if _, err := strconv.Atoi("non-int"); err != nil { // _ discards value
        // prints 'strconv.ParseInt: parsing "non-int": invalid syntax'
        fmt.Println(err)
    }
    // We'll revisit interfaces a little later. Meanwhile,
    learnConcurrency()
}

// c is a channel, a concurrency-safe communication object.
func inc(i int, c chan int) {
    c <- i + 1 // <- is the "send" operator when a channel appears on the left.
}

// We'll use inc to increment some numbers concurrently.
func learnConcurrency() {
    // Same make function used earlier to make a slice. Make allocates and
    // initializes slices, maps, and channels.
    c := make(chan int)
    // Start three concurrent goroutines. Numbers will be incremented
    // concurrently, perhaps in parallel if the machine is capable and
    // properly configured. All three send to the same channel.
    go inc(0, c) // go is a statement that starts a new goroutine.
    go inc(10, c)
    go inc(-805, c)
    // Read three results from the channel and print them out.
    // There is no telling in what order the results will arrive!
    fmt.Println(<-c, <-c, <-c) // channel on right, <- is "receive" operator.
    cs := make(chan string)       // Another channel, this one handles strings.
    ccs := make(chan chan string) // A channel of string channels.
    go func() { c <- 84 }()       // Start a new goroutine just to send a value.
    go func() { cs <- "wordy" }() // Again, for cs this time.
    // Select has syntax like a switch statement but each case involves
    // a channel operation. It selects a case at random out of the cases
    // that are ready to communicate.
    select {
    case i := <-c: // The value received can be assigned to a variable,
        fmt.Printf("it's a %T", i)
    case <-cs: // or the value received can be discarded.
        fmt.Println("it's a string")
    case <-ccs: // Empty channel, not ready for communication.
        fmt.Println("didn't happen.")
    }
    // At this point a value was taken from either c or cs. One of the two
    // goroutines started above has completed, the other will remain blocked.

    learnWebProgramming() // Go does it. You want to do it too.
}

// A single function from package http starts a web server.
func learnWebProgramming() {

    // First parameter of ListenAndServe is TCP address to listen to.
    // Second parameter is an interface, specifically http.Handler.
    go func() {
        err := http.ListenAndServe(":8080", pair{})
        fmt.Println(err) // don't ignore errors
    }()

    requestServer()
}

// Make pair an http.Handler by implementing its only method, ServeHTTP.
func (p pair) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // Serve data with a method of http.ResponseWriter.
    w.Write([]byte("You learned Go in Y minutes!"))
}

func requestServer() {
    resp, err := http.Get("http://localhost:8080")
    fmt.Println(err)
    defer resp.Body.Close()
    body, err := ioutil.ReadAll(resp.Body)
    fmt.Printf("\nWebserver said: `%s`", string(body))
}
```

Further Reading

The root of all things Go is the official Go web site. There you can follow the tutorial, play interactively, and read lots. Aside from a tour, the docs contain information on how to write clean and effective Go code, package and command docs, and release history.

The language definition itself is highly recommended. It's easy to read and amazingly short (as language definitions go these days.)

You can play around with the code on Go playground. Try to change it and run it from your browser! Note that you can use https://play.golang.org as a REPL to test things and code in your browser, without even installing Go.

On the reading list for students of Go is the source code to the standard library. Comprehensively documented, it demonstrates the best of readable and understandable Go, Go style, and Go idioms. Or you can click on a function name in the documentation and the source code comes up!

Another great resource to learn Go is Go by example.

Go Mobile adds support for mobile platforms (Android and iOS). You can write all-Go native mobile apps or write a library that contains bindings from a Go package, which can be invoked via Java (Android) and Objective-C (iOS). Check out the Go Mobile page for more information.

Groovy

Groovy - A dynamic language for the Java platform Read more here.

```
/*
  Set yourself up:

  1) Install SDKMAN - http://sdkman.io/
  2) Install Groovy: sdk install groovy
  3) Start the groovy console by typing: groovyConsole

*/

//  Single line comments start with two forward slashes
/*
Multi line comments look like this.
*/

// Hello World
println "Hello world!"

/*
  Variables:

  You can assign values to variables for later use
*/

def x = 1
println x

x = new java.util.Date()
println x

x = -3.1499392
println x
```

```
x = false
println x

x = "Groovy!"
println x

/*
  Collections and maps
*/

//Creating an empty list
def technologies = []

/*** Adding a elements to the list ***/

// As with Java
technologies.add("Grails")

// Left shift adds, and returns the list
technologies << "Groovy"

// Add multiple elements
technologies.addAll(["Gradle","Griffon"])

/*** Removing elements from the list ***/

// As with Java
technologies.remove("Griffon")

// Subtraction works also
technologies = technologies - 'Grails'

/*** Iterating Lists ***/

// Iterate over elements of a list
technologies.each { println "Technology: $it"}
technologies.eachWithIndex { it, i -> println "$i: $it"}

/*** Checking List contents ***/

//Evaluate if a list contains element(s) (boolean)
contained = technologies.contains( 'Groovy' )

// Or
contained = 'Groovy' in technologies

// Check for multiple contents
technologies.containsAll(['Groovy','Grails'])

/*** Sorting Lists ***/

// Sort a list (mutates original list)
technologies.sort()
```

```
// To sort without mutating original, you can do:
sortedTechnologies = technologies.sort( false )

/*** Manipulating Lists ***/e

//Replace all elements in the list
Collections.replaceAll(technologies, 'Gradle', 'gradle')

//Shuffle a list
Collections.shuffle(technologies, new Random())

//Clear a list
technologies.clear()

//Creating an empty map
def devMap = [:]

//Add values
devMap = ['name':'Roberto', 'framework':'Grails', 'language':'Groovy']
devMap.put('lastName','Perez')

//Iterate over elements of a map
devMap.each { println "$it.key: $it.value" }
devMap.eachWithIndex { it, i -> println "$i: $it"}

//Evaluate if a map contains a key
assert devMap.containsKey('name')

//Evaluate if a map contains a value
assert devMap.containsValue('Roberto')

//Get the keys of a map
println devMap.keySet()

//Get the values of a map
println devMap.values()

/*
  Groovy Beans

  GroovyBeans are JavaBeans but using a much simpler syntax

  When Groovy is compiled to bytecode, the following rules are used.

    * If the name is declared with an access modifier (public, private or
      protected) then a field is generated.

    * A name declared with no access modifier generates a private field with
      public getter and setter (i.e. a property).

    * If a property is declared final the private field is created final and no
      setter is generated.

    * You can declare a property and also declare your own getter or setter.
    * You can declare a property and a field of the same name, the property will
      use that field then.

    * If you want a private or protected property you have to provide your own
      getter and setter which must be declared private or protected.

    * If you access a property from within the class the property is defined in
      at compile time with implicit or explicit this (for example this.foo, or
      simply foo), Groovy will access the field directly instead of going though
      the getter and setter.

    * If you access a property that does not exist using the explicit or
      implicit foo, then Groovy will access the property through the meta class,
      which may fail at runtime.

*/

class Foo {
    // read only property
    final String name = "Roberto"

    // read only property with public getter and protected setter
    String language
    protected void setLanguage(String language) { this.language = language }

    // dynamically typed property
    def lastName
}

/*
  Logical Branching and Looping
*/

//Groovy supports the usual if - else syntax
def x = 3

if(x==1) {
    println "One"
} else if(x==2) {
    println "Two"
} else {
    println "X greater than Two"
}

//Groovy also supports the ternary operator:
def y = 10
def x = (y > 1) ? "worked" : "failed"
assert x == "worked"

//Groovy supports 'The Elvis Operator' too!
//Instead of using the ternary operator:

displayName = user.name ? user.name : 'Anonymous'
```

```
//We can write it:
displayName = user.name ?: 'Anonymous'

//For loop
//Iterate over a range
def x = 0
for (i in 0 .. 30) {
    x += i
}

//Iterate over a list
x = 0
for( i in [5,3,2,1] ) {
    x += i
}

//Iterate over an array
array = (0..20).toArray()
x = 0
for (i in array) {
    x += i
}

//Iterate over a map
def map = ['name':'Roberto', 'framework':'Grails', 'language':'Groovy']
x = 0
for ( e in map ) {
    x += e.value
}

/*
  Operators

  Operator Overloading for a list of the common operators that Groovy supports:
  http://www.groovy-lang.org/operators.html#Operator-Overloading

  Helpful groovy operators
*/
//Spread operator:  invoke an action on all items of an aggregate object.
def technologies = ['Groovy','Grails','Gradle']
technologies*.toUpperCase() // = to technologies.collect { it?.toUpperCase() }

//Safe navigation operator: used to avoid a NullPointerException.
def user = User.get(1)
def username = user?.username


/*
  Closures
  A Groovy Closure is like a "code block" or a method pointer. It is a piece of
  code that is defined and then executed at a later point.

  More info at: http://www.groovy-lang.org/closures.html
```

```
*/
//Example:
def clos = { println "Hello World!" }

println "Executing the Closure:"
clos()

//Passing parameters to a closure
def sum = { a, b -> println a+b }
sum(2,4)

//Closures may refer to variables not listed in their parameter list.
def x = 5
def multiplyBy = { num -> num * x }
println multiplyBy(10)

// If you have a Closure that takes a single argument, you may omit the
// parameter definition of the Closure
def clos = { print it }
clos( "hi" )

/*
  Groovy can memorize closure results [1][2][3]
*/
def cl = {a, b ->
    sleep(3000) // simulate some time consuming processing
    a + b
}

mem = cl.memoize()

def callClosure(a, b) {
    def start = System.currentTimeMillis()
    mem(a, b)
    println "Inputs(a = $a, b = $b) - took ${System.currentTimeMillis() - start} msecs."
}

callClosure(1, 2)
callClosure(1, 2)
callClosure(2, 3)
callClosure(2, 3)
callClosure(3, 4)
callClosure(3, 4)
callClosure(1, 2)
callClosure(2, 3)
callClosure(3, 4)

/*
  Expando

  The Expando class is a dynamic bean so we can add properties and we can add
  closures as methods to an instance of this class

  http://mrhaki.blogspot.mx/2009/10/groovy-goodness-expando-as-dynamic-bean.html
```

```
*/
  def user = new Expando(name:"Roberto")
  assert 'Roberto' == user.name

  user.lastName = 'Pérez'
  assert 'Pérez' == user.lastName

  user.showInfo = { out ->
      out << "Name: $name"
      out << ", Last name: $lastName"
  }

  def sw = new StringWriter()
  println user.showInfo(sw)


/*
  Metaprogramming (MOP)
*/

//Using ExpandoMetaClass to add behaviour
String.metaClass.testAdd = {
    println "we added this"
}

String x = "test"
x?.testAdd()

//Intercepting method calls
class Test implements GroovyInterceptable {
    def sum(Integer x, Integer y) { x + y }

    def invokeMethod(String name, args) {
        System.out.println "Invoke method $name with args: $args"
    }
}

def test = new Test()
test?.sum(2,3)
test?.multiply(2,3)

//Groovy supports propertyMissing for dealing with property resolution attempts.
class Foo {
   def propertyMissing(String name) { name }
}
def f = new Foo()

assertEquals "boo", f.boo

/*
  TypeChecked and CompileStatic
  Groovy, by nature, is and will always be a dynamic language but it supports
  typechecked and compilestatic
  More info: http://www.infoq.com/articles/new-groovy-20
*/
//TypeChecked
import groovy.transform.TypeChecked

void testMethod() {}

@TypeChecked
void test() {
    testMeethod()

    def name = "Roberto"

    println naameee

}

//Another example:
import groovy.transform.TypeChecked

@TypeChecked
Integer test() {
    Integer num = "1"

    Integer[] numbers = [1,2,3,4]

    Date date = numbers[1]

    return "Test"

}

//CompileStatic example:
import groovy.transform.CompileStatic

@CompileStatic
int sum(int x, int y) {
    x + y
}

assert sum(2,5) == 7
```

Further resources

Groovy documentation

Groovy web console

Join a Groovy user group

Books

- [Groovy Goodness] (https://leanpub.com/groovy-goodness-notebook)

- [Groovy in Action] (http://manning.com/koenig2/)
- [Programming Groovy 2: Dynamic Productivity for the Java Developer] (http://shop.oreilly.com/product/978193778530

1 http://roshandawrani.wordpress.com/2010/10/18/groovy-new-feature-closures-can-now-memorize-their-results/ 2 http://www.solutionsiq.com/resources/agileiq-blog/bid/72880/Programming-with-Groovy-Trampoline-and-Memoize 3 http://mrhaki.blogspot.mx/2011/05/groovy-goodness-cache-closure-results.html

Hack

Hack is a superset of PHP that runs under a virtual machine called HHVM. Hack is almost completely interoperable with existing PHP code and adds a bunch of useful features from statically typed languages.

Only Hack-specific features are covered here. Details about PHP's syntax are available in the PHP article on this site.

```
<?hh

// Hack syntax is only enabled for files starting with an <?hh marker
// <?hh markers cannot be interspersed with HTML the way <?php can be.
// Using the marker "<?hh //strict" puts the type checker in strict mode.


// Scalar parameter type hints
function repeat(string $word, int $count)
{
    $word = trim($word);
    return str_repeat($word . ' ', $count);
}

// Type hints for return values
function add(...$numbers) : int
{
    return array_sum($numbers);
}

// Functions that return nothing are hinted as "void"
function truncate(resource $handle) : void
{
    // ...
}

// Type hints must explicitly allow being nullable
function identity(?string $stringOrNull) : ?string
{
    return $stringOrNull;
}

// Type hints can be specified on class properties
class TypeHintedProperties
{
    public ?string $name;

    protected int $id;
    private float $score = 100.0;

    // Hack's type checker enforces that typed properties either have a
    // default value or are set in the constructor.
    public function __construct(int $id)
    {
        $this->id = $id;
    }
}


// Concise anonymous functions (lambdas)
$multiplier = 5;
array_map($y ==> $y * $multiplier, [1, 2, 3]);


// Generics
class Box<T>
{
    protected T $data;

    public function __construct(T $data) {
        $this->data = $data;
    }

    public function getData(): T {
        return $this->data;
    }
}

function openBox(Box<int> $box) : int
{
    return $box->getData();
}


// Shapes
//
// Hack adds the concept of shapes for defining struct-like arrays with a
// guaranteed, type-checked set of keys
type Point2D = shape('x' => int, 'y' => int);

function distance(Point2D $a, Point2D $b) : float
{
    return sqrt(pow($b['x'] - $a['x'], 2) + pow($b['y'] - $a['y'], 2));
}

distance(
    shape('x' => -1, 'y' => 5),
    shape('x' => 2, 'y' => 50)
);
```

```
// Type aliasing
//
// Hack adds a bunch of type aliasing features for making complex types readable
newtype VectorArray = array<int, Vector<int>>;

// A tuple containing two integers
newtype Point = (int, int);

function addPoints(Point $p1, Point $p2) : Point
{
    return tuple($p1[0] + $p2[0], $p1[1] + $p2[1]);
}

addPoints(
    tuple(1, 2),
    tuple(5, 6)
);


// First-class enums
enum RoadType : int
{
    Road = 0;
    Street = 1;
    Avenue = 2;
    Boulevard = 3;
}

function getRoadType() : RoadType
{
    return RoadType::Avenue;
}


// Constructor argument promotion
//
// To avoid boilerplate property and constructor definitions that only set
// properties, Hack adds a concise syntax for defining properties and a
// constructor at the same time.
class ArgumentPromotion
{
    public function __construct(public string $name,
                                protected int $age,
                                private bool $isAwesome) {}
}

class WithoutArgumentPromotion
{
    public string $name;

    protected int $age;

    private bool $isAwesome;
    public function __construct(string $name, int $age, bool $isAwesome)
    {
        $this->name = $name;
        $this->age = $age;
        $this->isAwesome = $isAwesome;
    }
}


// Co-operative multi-tasking
//
// Two new keywords "async" and "await" can be used to perform multi-tasking
// Note that this does not involve threads - it just allows transfer of control
async function cooperativePrint(int $start, int $end) : Awaitable<void>
{
    for ($i = $start; $i <= $end; $i++) {
        echo "$i ";

        // Give other tasks a chance to do something
        await RescheduleWaitHandle::create(RescheduleWaitHandle::QUEUE_DEFAULT, 0);
    }
}

// This prints "1 4 7 2 5 8 3 6 9"
AwaitAllWaitHandle::fromArray([
    cooperativePrint(1, 3),
    cooperativePrint(4, 6),
    cooperativePrint(7, 9)
])->getWaitHandle()->join();


// Attributes
//
// Attributes are a form of metadata for functions. Hack provides some
// special built-in attributes that introduce useful behaviour.

// The __Memoize special attribute causes the result of a function to be cached
<<__Memoize>>
function doExpensiveTask() : ?string
{
    return file_get_contents('http://example.com');
}

// The function's body is only executed once here:
doExpensiveTask();
doExpensiveTask();


// The __ConsistentConstruct special attribute signals the Hack type checker to
// ensure that the signature of __construct is the same for all subclasses.
<<__ConsistentConstruct>>
class ConsistentFoo
{
    public function __construct(int $x, float $y)
    {
        // ...
    }

    public function someMethod()
    {
        // ...
    }
}

class ConsistentBar extends ConsistentFoo
{
    public function __construct(int $x, float $y)
    {
        // Hack's type checker enforces that parent constructors are called
        parent::__construct($x, $y);

        // ...
    }

    // The __Override annotation is an optional signal for the Hack type
    // checker to enforce that this method is overriding a method in a parent
    // or trait. If not, this will error.
    <<__Override>>
    public function someMethod()
    {
        // ...
    }
}

class InvalidFooSubclass extends ConsistentFoo
{
    // Not matching the parent constructor will cause a type checker error:
    //
    //  "This object is of type ConsistentBaz. It is incompatible with this object
    //   of type ConsistentFoo because some of their methods are incompatible"
    //
    public function __construct(float $x)
    {
        // ...
    }

    // Using the __Override annotation on a non-overriden method will cause a
    // type checker error:
    //
    //  "InvalidFooSubclass::otherMethod() is marked as override; no non-private
    //   parent definition found or overridden parent is defined in non-<?hh code"
    //
    <<__Override>>
    public function otherMethod()
    {
        // ...
    }
}
```

```
// Traits can implement interfaces (standard PHP does not support this)
interface KittenInterface
{
    public function play() : void;
}

trait CatTrait implements KittenInterface
{
    public function play() : void
    {
        // ...
    }
}

class Samuel
{
    use CatTrait;
}


$cat = new Samuel();
$cat instanceof KittenInterface === true; // True
```

More Information

Visit the Hack language reference for detailed explainations of the features Hack adds to PHP, or the official Hack website for more general information.

Visit the official HHVM website for HHVM installation instructions.

Visit Hack's unsupported PHP features article for details on the backwards incompatibility between Hack and PHP.

Haml

Haml is a markup language predominantly used with Ruby that cleanly and simply describes the HTML of any web document without the use of inline code. It is a popular alternative to using Rails templating language (.erb) and allows you to embed Ruby code into your markup.

It aims to reduce repetition in your markup by closing tags for you based on the structure of the indents in your code. The result is markup that is well-structured, DRY, logical, and easier to read.

You can also use Haml on a project independent of Ruby, by installing the Haml gem on your machine and using the command line to convert it to html.

$ haml input_file.haml output_file.html

```
/ -------------------------------------------
/ Indenting
/ -------------------------------------------

/
  Because of the importance indentation has on how your code is rendered, the
  indents should be consistent throughout the document. Any differences in
  indentation will throw an error. It's common-practice to use two spaces,
  but it's really up to you, as long as they're constant.


/ -------------------------------------------
/ Comments
/ -------------------------------------------

/ This is what a comment looks like in Haml.

/
  To write a multi line comment, indent your commented code to be
  wrapped by the forward slash

-# This is a silent comment, which means it wont be rendered into the doc at all


/ -------------------------------------------
/ Html elements
/ -------------------------------------------

/ To write your tags, use the percent sign followed by the name of the tag
%body
  %header
    %nav

/ Notice no closing tags. The above code would output
  <body>
    <header>
      <nav></nav>
    </header>
  </body>

/ The div tag is the default element, so they can be written simply like this
.foo

/ To add content to a tag, add the text directly after the declaration
%h1 Headline copy

/ To write multiline content, nest it instead
%p
  This is a lot of content that we could probably split onto two
  separate lines.

/
  You can escape html by using the ampersand and equals sign ( &= ). This
  converts html-sensitive characters (&, /, :) into their html encoded
  equivalents. For example

%p
  &= "Yes & yes"

/ would output 'Yes &amp; yes'
/ You can unescape html by using the bang and equals sign ( != )
%p
  != "This is how you write a paragraph tag <p></p>"

/ which would output 'This is how you write a paragraph tag <p></p>'

/ CSS classes can be added to your tags either by chaining .classnames to the tag
%div.foo.bar

/ or as part of a Ruby hash
%div{:class => 'foo bar'}

/ Attributes for any tag can be added in the hash
%a{:href => '#', :class => 'bar', :title => 'Bar'}

/ For boolean attributes assign the value 'true'
%input{:selected => true}

/ To write data-attributes, use the :data key with its value as another hash
%div{:data => {:attribute => 'foo'}}


/ -------------------------------------------
/ Inserting Ruby
/ -------------------------------------------

/
  To output a Ruby value as the contents of a tag, use an equals sign followed
  by the Ruby code

%h1= book.name

%p
  = book.author
  = book.publisher


/ To run some Ruby code without rendering it to the html, use a hyphen instead
- books = ['book 1', 'book 2', 'book 3']

/ Allowing you to do all sorts of awesome, like Ruby blocks
- books.shuffle.each_with_index do |book, index|
  %h1= book

  if book do
    %p This is a book

/ Adding ordered / unordered list
%ul
  %li
    =item1
    =item2
```

```
/
  Again, no need to add the closing tags to the block, even for the Ruby.
  Indentation will take care of that for you.

/ -------------------------------------------
/ Inserting Table with bootstrap classes
/ -------------------------------------------

%table.table.table-hover
  %thead
    %tr
      %th Header 1
      %th Header 2

    %tr
      %td Value1
      %td value2

  %tfoot
    %tr
      %td
        Foot value


/ -------------------------------------------
/ Inline Ruby / Ruby interpolation
/ -------------------------------------------

/ Include a Ruby variable in a line of plain text using #{}
%p Your highest scoring game is #{best_game}


/ -------------------------------------------
/ Filters
/ -------------------------------------------

/
  Use the colon to define Haml filters, one example of a filter you can
  use is :javascript, which can be used for writing inline js

:javascript
  console.log('This is inline <script>');
```

Additional resources

- What is HAML? - A good introduction that does a much better job of explaining the benefits of using HAML.
- Official Docs - If you'd like to go a little deeper.

Haskell

Haskell was designed as a practical, purely functional programming language. It's famous for its monads and its type system, but I keep coming back to it because of its elegance. Haskell makes coding a real joy for me.

```haskell
-- Single line comments start with two dashes.
{- Multiline comments can be enclosed
in a block like this.
-}

----------------------------------------------------
-- 1. Primitive Datatypes and Operators
----------------------------------------------------

-- You have numbers
3 -- 3

-- Math is what you would expect
1 + 1 -- 2
8 - 1 -- 7
10 * 2 -- 20
35 / 5 -- 7.0

-- Division is not integer division by default
35 / 4 -- 8.75

-- integer division
35 `div` 4 -- 8

-- Boolean values are primitives
True
False

-- Boolean operations
not True -- False
not False -- True
1 == 1 -- True
1 /= 1 -- False
1 < 10 -- True

-- In the above examples, `not` is a function that takes one value.
-- Haskell doesn't need parentheses for function calls...all the arguments
-- are just listed after the function. So the general pattern is:
-- func arg1 arg2 arg3...
-- See the section on functions for information on how to write your own.

-- Strings and characters
"This is a string."
'a' -- character
'You cant use single quotes for strings.' -- error!

-- Strings can be concatenated
"Hello " ++ "world!" -- "Hello world!"
```

```haskell
-- A string is a list of characters
['H', 'e', 'l', 'l', 'o'] -- "Hello"
"This is a string" !! 0 -- 'T'


----------------------------------------------------
-- Lists and Tuples
----------------------------------------------------

-- Every element in a list must have the same type.
-- These two lists are the same:
[1, 2, 3, 4, 5]
[1..5]

-- Ranges are versatile.
['A'..'F'] -- "ABCDEF"

-- You can create a step in a range.
[0,2..10] -- [0, 2, 4, 6, 8, 10]
[5..1] -- This doesn't work because Haskell defaults to incrementing.
[5,4..1] -- [5, 4, 3, 2, 1]

-- indexing into a list
[1..10] !! 3 -- 4

-- You can also have infinite lists in Haskell!
[1..] -- a list of all the natural numbers

-- Infinite lists work because Haskell has "lazy evaluation". This means
-- that Haskell only evaluates things when it needs to. So you can ask for
-- the 1000th element of your list and Haskell will give it to you:

[1..] !! 999 -- 1000

-- And now Haskell has evaluated elements 1 - 1000 of this list...but the
-- rest of the elements of this "infinite" list don't exist yet! Haskell won't
-- actually evaluate them until it needs to.

-- joining two lists
[1..5] ++ [6..10]

-- adding to the head of a list
0:[1..5] -- [0, 1, 2, 3, 4, 5]

-- more list operations
head [1..5] -- 1
tail [1..5] -- [2, 3, 4, 5]
init [1..5] -- [1, 2, 3, 4]
last [1..5] -- 5

-- list comprehensions
[x*2 | x <- [1..5]] -- [2, 4, 6, 8, 10]
```

```haskell
-- with a conditional
[x*2 | x <- [1..5], x*2 > 4] -- [6, 8, 10]

-- Every element in a tuple can be a different type, but a tuple has a
-- fixed length.
-- A tuple:
("haskell", 1)

-- accessing elements of a pair (i.e. a tuple of length 2)
fst ("haskell", 1) -- "haskell"
snd ("haskell", 1) -- 1

----------------------------------------------------
-- 3. Functions
----------------------------------------------------
-- A simple function that takes two variables
add a b = a + b

-- Note that if you are using ghci (the Haskell interpreter)
-- You'll need to use `let`, i.e.
-- let add a b = a + b

-- Using the function
add 1 2 -- 3

-- You can also put the function name between the two arguments
-- with backticks:
1 `add` 2 -- 3

-- You can also define functions that have no letters! This lets
-- you define your own operators! Here's an operator that does
-- integer division
(//) a b = a `div` b
35 // 4 -- 8

-- Guards: an easy way to do branching in functions
fib x
  | x < 2 = 1
  | otherwise = fib (x - 1) + fib (x - 2)

-- Pattern matching is similar. Here we have given three different
-- definitions for fib. Haskell will automatically call the first
-- function that matches the pattern of the value.
fib 1 = 1
fib 2 = 2
fib x = fib (x - 1) + fib (x - 2)

-- Pattern matching on tuples:
foo (x, y) = (x + 1, y + 2)

-- Pattern matching on lists. Here `x` is the first element
-- in the list, and `xs` is the rest of the list. We can write
-- our own map function:
myMap func [] = []
myMap func (x:xs) = func x:(myMap func xs)

-- Anonymous functions are created with a backslash followed by
-- all the arguments.
myMap (\x -> x + 2) [1..5] -- [3, 4, 5, 6, 7]

-- using fold (called `inject` in some languages) with an anonymous
-- function. foldl1 means fold left, and use the first value in the
-- list as the initial value for the accumulator.
foldl1 (\acc x -> acc + x) [1..5] -- 15

----------------------------------------------------
-- 4. More functions
----------------------------------------------------

-- partial application: if you don't pass in all the arguments to a function,
-- it gets "partially applied". That means it returns a function that takes the
-- rest of the arguments.

add a b = a + b
foo = add 10 -- foo is now a function that takes a number and adds 10 to it
foo 5 -- 15

-- Another way to write the same thing
foo = (+10)
foo 5 -- 15

-- function composition
-- the (.) function chains functions together.
-- For example, here foo is a function that takes a value. It adds 10 to it,
-- multiplies the result of that by 4, and then returns the final value.
foo = (*4) . (+10)

-- (5 + 10) * 4 = 60
foo 5 -- 60

-- fixing precedence
-- Haskell has another operator called `$`. This operator applies a function
-- to a given parameter. In contrast to standard function application, which
-- has highest possible priority of 10 and is left-associative, the `$` operator
-- has priority of 0 and is right-associative. Such a low priority means that
-- the expression on its right is applied as the parameter to the function on its left.

-- before
even (fib 7) -- false

-- equivalently
even $ fib 7 -- false

-- composing functions
even . fib $ 7 -- false


----------------------------------------------------
```

```
-- 5. Type signatures
----------------------------------------------------

-- Haskell has a very strong type system, and everything has a type signature.

-- Some basic types:
5 :: Integer
"hello" :: String
True :: Bool

-- Functions have types too.
-- `not` takes a boolean and returns a boolean:
-- not :: Bool -> Bool

-- Here's a function that takes two arguments:
-- add :: Integer -> Integer -> Integer

-- When you define a value, it's good practice to write its type above it:
double :: Integer -> Integer
double x = x * 2

----------------------------------------------------
-- 6. Control Flow and If Expressions
----------------------------------------------------

-- if expressions
haskell = if 1 == 1 then "awesome" else "awful" -- haskell = "awesome"

-- if expressions can be on multiple lines too, indentation is important
haskell = if 1 == 1
            then "awesome"
            else "awful"

-- case expressions: Here's how you could parse command line arguments
case args of
  "help" -> printHelp
  "start" -> startProgram
  _ -> putStrLn "bad args"

-- Haskell doesn't have loops; it uses recursion instead.
-- map applies a function over every element in an array

map (*2) [1..5] -- [2, 4, 6, 8, 10]

-- you can make a for function using map
for array func = map func array

-- and then use it
for [0..5] $ \i -> show i

-- we could've written that like this too:
for [0..5] show

-- You can use foldl or foldr to reduce a list
-- foldl <fn> <initial value> <list>
foldl (\x y -> 2*x + y) 4 [1,2,3] -- 43

-- This is the same as
(2 * (2 * (2 * 4 + 1) + 2) + 3)

-- foldl is left-handed, foldr is right-
foldr (\x y -> 2*x + y) 4 [1,2,3] -- 16

-- This is now the same as
(2 * 1 + (2 * 2 + (2 * 3 + 4)))

----------------------------------------------------
-- 7. Data Types
----------------------------------------------------

-- Here's how you make your own data type in Haskell

data Color = Red | Blue | Green

-- Now you can use it in a function:


say :: Color -> String
say Red = "You are Red!"
say Blue = "You are Blue!"
say Green =  "You are Green!"

-- Your data types can have parameters too:

data Maybe a = Nothing | Just a

-- These are all of type Maybe
Just "hello"    -- of type `Maybe String`
Just 1          -- of type `Maybe Int`
Nothing         -- of type `Maybe a` for any `a`

----------------------------------------------------
-- 8. Haskell IO
----------------------------------------------------

-- While IO can't be explained fully without explaining monads,
-- it is not hard to explain enough to get going.

-- When a Haskell program is executed, `main` is
-- called. It must return a value of type `IO ()`. For example:

main :: IO ()
main = putStrLn $ "Hello, sky! " ++ (say Blue)
-- putStrLn has type String -> IO ()

-- It is easiest to do IO if you can implement your program as
-- a function from String to String. The function
--    interact :: (String -> String) -> IO ()
```

```haskell
-- inputs some text, runs a function on it, and prints out the
-- output.

countLines :: String -> String
countLines = show . length . lines

main' = interact countLines

-- You can think of a value of type `IO ()` as representing a
-- sequence of actions for the computer to do, much like a
-- computer program written in an imperative language. We can use
-- the `do` notation to chain actions together. For example:

sayHello :: IO ()
sayHello = do
   putStrLn "What is your name?"
   name <- getLine -- this gets a line and gives it the name "name"
   putStrLn $ "Hello, " ++ name

-- Exercise: write your own version of `interact` that only reads
--           one line of input.

-- The code in `sayHello` will never be executed, however. The only
-- action that ever gets executed is the value of `main`.
-- To run `sayHello` comment out the above definition of `main`
-- and replace it with:
--   main = sayHello

-- Let's understand better how the function `getLine` we just
-- used works. Its type is:
--    getLine :: IO String
-- You can think of a value of type `IO a` as representing a
-- computer program that will generate a value of type `a`
-- when executed (in addition to anything else it does). We can
-- store and reuse this value using `<-`. We can also
-- make our own action of type `IO String`:

action :: IO String
action = do
   putStrLn "This is a line. Duh"
   input1 <- getLine
   input2 <- getLine
   -- The type of the `do` statement is that of its last line.
   -- `return` is not a keyword, but merely a function
   return (input1 ++ "\n" ++ input2) -- return :: String -> IO String

-- We can use this just like we used `getLine`:

main'' = do
    putStrLn "I will echo two lines!"
    result <- action
    putStrLn result
    putStrLn "This was all, folks!"
```

```
-- The type `IO` is an example of a "monad". The way Haskell uses a monad to
-- do IO allows it to be a purely functional language. Any function that
-- interacts with the outside world (i.e. does IO) gets marked as `IO` in its
-- type signature. This lets us reason about what functions are "pure" (don't
-- interact with the outside world or modify state) and what functions aren't.

-- This is a powerful feature, because it's easy to run pure functions
-- concurrently; so, concurrency in Haskell is very easy.


----------------------------------------------------
-- 9. The Haskell REPL
----------------------------------------------------

-- Start the repl by typing `ghci`.
-- Now you can type in Haskell code. Any new values
-- need to be created with `let`:

let foo = 5

-- You can see the type of any value with `:t`:

>:t foo
foo :: Integer

-- You can also run any action of type `IO ()`

> sayHello
What is your name?
Friend!
Hello, Friend!
```

There's a lot more to Haskell, including typeclasses and monads. These are the big ideas that make Haskell such fun to code in. I'll leave you with one final Haskell example: an implementation of quicksort in Haskell:

```
qsort [] = []
qsort (p:xs) = qsort lesser ++ [p] ++ qsort greater
    where lesser  = filter (< p) xs
          greater = filter (>= p) xs
```

There are two popular ways to install Haskell: The traditional Cabal-based installation, and the newer Stack-based process.

You can find a much gentler introduction from the excellent Learn you a Haskell or Real World Haskell.

Haxe

Haxe is a web-oriented language that provides platform support for C++, C#, Swf/ActionScript, Javascript, Java, and Neko byte code (also written by the Haxe author). Note that this guide is for Haxe version 3. Some of the guide may be applicable to older versions, but it is recommended to use other references.

```
/*
   Welcome to Learn Haxe 3 in 15 minutes.  http://www.haxe.org
   This is an executable tutorial.  You can compile and run it using the haxe
   compiler, while in the same directory as LearnHaxe.hx:
```

```
   $> haxe -main LearnHaxe3 -x out

   Look for the slash-star marks surrounding these paragraphs.  We are inside
   a "Multiline comment".  We can leave some notes here that will get ignored
   by the compiler.

   Multiline comments are also used to generate javadoc-style documentation for
   haxedoc.  They will be used for haxedoc if they immediately precede a class,
   class function, or class variable.

 */

// Double slashes like this will give a single-line comment


/*
   This is your first actual haxe code coming up, it's declaring an empty
   package.  A package isn't necessary, but it's useful if you want to create a
   namespace for your code (e.g. org.yourapp.ClassName).

   Omitting package declaration is the same as declaring an empty package.
 */
package; // empty package, no namespace.

/*
   Packages are directories that contain modules. Each module is a .hx file
   that contains types defined in a package. Package names (e.g. org.yourapp)
   must be lower case while module names are capitalized. A module contain one
   or more types whose names are also capitalized.

   E.g, the class "org.yourapp.Foo" should have the folder structure org/module/Foo.hx,
   as accessible from the compiler's working directory or class path.

   If you import code from other files, it must be declared before the rest of
   the code.  Haxe provides a lot of common default classes to get you started:
 */
import haxe.ds.ArraySort;

// you can import many classes/modules at once with "*"
import haxe.ds.*;

// you can import static fields
import Lambda.array;

// you can also use "*" to import all static fields
import Math.*;

/*
   You can also import classes in a special way, enabling them to extend the
   functionality of other classes like a "mixin".  More on 'using' later.
 */
using StringTools;

/*
```

```
   Typedefs are like variables... for types.  They must be declared before any
   code.  More on this later.
 */
typedef FooString = String;

// Typedefs can also reference "structural" types, more on that later as well.
typedef FooObject = { foo: String };

/*
   Here's the class definition.  It's the main class for the file, since it has
   the same name (LearnHaxe3).
 */
class LearnHaxe3{
    /*
       If you want certain code to run automatically, you need to put it in
       a static main function, and specify the class in the compiler arguments.
       In this case, we've specified the "LearnHaxe3" class in the compiler
       arguments above.
     */
    static function main(){

        /*
           Trace is the default method of printing haxe expressions to the
           screen.  Different targets will have different methods of
           accomplishing this.  E.g., java, c++, c#, etc. will print to std
           out.  Javascript will print to console.log, and flash will print to
           an embedded TextField.  All traces come with a default newline.
           Finally, It's possible to prevent traces from showing by using the
           "--no-traces" argument on the compiler.
         */
        trace("Hello World, with trace()!");

        /*
           Trace can handle any type of value or object.  It will try to print
           a representation of the expression as best it can.  You can also
           concatenate strings with the "+" operator:
         */
        trace( " Integer: " + 10 + " Float: " + 3.14 + " Boolean: " + true);

        /*
           In Haxe, it's required to separate expressions in the same block with
           semicolons.  But, you can put two expressions on one line:
         */
        trace('two expressions..'); trace('one line');


        //////////////////////////////////////////////////////////////////
        // Types & Variables
        //////////////////////////////////////////////////////////////////
        trace("***Types & Variables***");

        /*
           You can save values and references to data structures using the
           "var" keyword:
```

```
 */
var an_integer:Int = 1;
trace(an_integer + " is the value for an_integer");


/*
   Haxe is statically typed, so "an_integer" is declared to have an
   "Int" type, and the rest of the expression assigns the value "1" to
   it.  It's not necessary to declare the type in many cases.  Here,
   the haxe compiler is inferring that the type of another_integer
   should be "Int".
 */
var another_integer = 2;
trace(another_integer + " is the value for another_integer");

// The $type() method prints the type that the compiler assigns:
$type(another_integer);

// You can also represent integers with hexadecimal:
var hex_integer = 0xffffff;

/*
   Haxe uses platform precision for Int and Float sizes.  It also
   uses the platform behavior for overflow.
   (Other numeric types and behavior are possible using special
   libraries)
 */

/*
   In addition to simple values like Integers, Floats, and Booleans,
   Haxe provides standard library implementations for common data
   structures like strings, arrays, lists, and maps:
 */

var a_string = "some" + 'string';  // strings can have double or single quotes
trace(a_string + " is the value for a_string");

/*
   Strings can be "interpolated" by inserting variables into specific
   positions.  The string must be single quoted, and the variable must
   be preceded with "$".  Expressions can be enclosed in ${...}.
 */
var x = 1;
var an_interpolated_string = 'the value of x is $x';
var another_interpolated_string = 'the value of x + 1 is ${x + 1}';

/*
   Strings are immutable, instance methods will return a copy of
   parts or all of the string.
   (See also the StringBuf class).
 */
var a_sub_string = a_string.substr(0,4);
trace(a_sub_string + " is the value for a_sub_string");
```

```
/*
   Regexes are also supported, but there's not enough space to go into
   much detail.
 */
var re = ~/foobar/;
trace(re.match('foo') + " is the value for (~/foobar/.match('foo')))");

/*
   Arrays are zero-indexed, dynamic, and mutable.  Missing values are
   defined as null.
 */
var a = new Array<String>(); // an array that contains Strings
a[0] = 'foo';
trace(a.length + " is the value for a.length");
a[9] = 'bar';
trace(a.length + " is the value for a.length (after modification)");
trace(a[3] + " is the value for a[3]"); //null

/*
   Arrays are *generic*, so you can indicate which values they contain
   with a type parameter:
 */
var a2 = new Array<Int>(); // an array of Ints
var a3 = new Array<Array<String>>(); // an Array of Arrays (of Strings).

/*
   Maps are simple key/value data structures.  The key and the value
   can be of any type.
 */
var m = new Map<String, Int>();  // The keys are strings, the values are Ints.
m.set('foo', 4);
// You can also use array notation;
m['bar'] = 5;
trace(m.exists('bar') + " is the value for m.exists('bar')");
trace(m.get('bar') + " is the value for m.get('bar')");
trace(m['bar'] + " is the value for m['bar']");

var m2 =  ['foo' => 4, 'baz' => 6]; // Alternative map syntax
trace(m2 + " is the value for m2");

/*
   Remember, you can use type inference.  The Haxe compiler will
   decide the type of the variable the first time you pass an
   argument that sets a type parameter.
 */
var m3 = new Map();
m3.set(6, 'baz'); // m3 is now a Map<Int,String>
trace(m3 + " is the value for m3");

/*
   Haxe has some more common datastructures in the haxe.ds module, such as
   List, Stack, and BalancedTree
 */
```

```
//////////////////////////////////////////////////////////////////
// Operators
//////////////////////////////////////////////////////////////////
trace("***OPERATORS***");

// basic arithmetic
trace((4 + 3) + " is the value for (4 + 3)");
trace((5 - 1) + " is the value for (5 - 1)");
trace((2 * 4) + " is the value for (2 * 4)");
trace((8 / 3) + " is the value for (8 / 3) (division always produces Floats)");
trace((12 % 4) + " is the value for (12 % 4)");


//basic comparison
trace((3 == 2) + " is the value for 3 == 2");
trace((3 != 2) + " is the value for 3 != 2");
trace((3 >  2) + " is the value for 3 > 2");
trace((3 <  2) + " is the value for 3 < 2");
trace((3 >= 2) + " is the value for 3 >= 2");
trace((3 <= 2) + " is the value for 3 <= 2");

// standard bitwise operators
/*
~       Unary bitwise complement
<<      Signed left shift
>>      Signed right shift
>>>     Unsigned right shift
&       Bitwise AND
^       Bitwise exclusive OR
|       Bitwise inclusive OR
*/

//increments
var i = 0;
trace("Increments and decrements");
trace(i++); //i = 1. Post-Incrementation
trace(++i); //i = 2. Pre-Incrementation
trace(i--); //i = 1. Post-Decrementation
trace(--i); //i = 0. Pre-Decrementation

//////////////////////////////////////////////////////////////////
// Control Structures
//////////////////////////////////////////////////////////////////
trace("***CONTROL STRUCTURES***");

// if statements
var j = 10;
if (j == 10){
    trace("this is printed");
} else if (j > 10){
    trace("not greater than 10, so not printed");
} else {
    trace("also not printed.");
```

```
        }

        // there is also a "ternary" if:
        (j == 10) ?  trace("equals 10") : trace("not equals 10");

        /*
           Finally, there is another form of control structures that operates
           at compile time:  conditional compilation.
         */
#if neko
        trace('hello from neko');
#elseif js
        trace('hello from js');
#else
        trace('hello from another platform!');
#end
        /*
           The compiled code will change depending on the platform target.
           Since we're compiling for neko (-x or -neko), we only get the neko
           greeting.
         */


        trace("Looping and Iteration");

        // while loop
        var k = 0;
        while(k < 100){
            // trace(counter); // will print out numbers 0-99
            k++;
        }

        // do-while loop
        var  l = 0;
        do{
            trace("do statement always runs at least once");
        } while (l > 0);

        // for loop
        /*
           There is no c-style for loop in Haxe, because they are prone
           to error, and not necessary.  Instead, Haxe has a much simpler
           and safer version that uses Iterators (more on those later).
         */
        var m = [1,2,3];
        for (val in m){
            trace(val + " is the value for val in the m array");
        }

        // Note that you can iterate on an index using a range
        // (more on ranges later as well)
        var n = ['foo', 'bar', 'baz'];
        for (val in 0...n.length){
            trace(val + " is the value for val (an index for n)");
```

```
}


trace("Array Comprehensions");

// Array comprehensions give you the ability to iterate over arrays
// while also creating filters and modifications.
var filtered_n = [for (val in n) if (val != "foo") val];
trace(filtered_n + " is the value for filtered_n");

var modified_n = [for (val in n) val += '!'];
trace(modified_n + " is the value for modified_n");

var filtered_and_modified_n = [for (val in n) if (val != "foo") val += "!"];
trace(filtered_and_modified_n + " is the value for filtered_and_modified_n");

//////////////////////////////////////////////////////////////////
// Switch Statements (Value Type)
//////////////////////////////////////////////////////////////////
trace("***SWITCH STATEMENTS (VALUE TYPES)***");

/*
   Switch statements in Haxe are very powerful.  In addition to working
   on basic values like strings and ints, they can also work on the
   generalized algebraic data types in enums (more on enums later).
   Here's some basic value examples for now:
 */
var my_dog_name = "fido";
var favorite_thing  = "";
switch(my_dog_name){
    case "fido" : favorite_thing = "bone";
    case "rex"  : favorite_thing = "shoe";
    case "spot" : favorite_thing = "tennis ball";
    default     : favorite_thing = "some unknown treat";
    // case _   : favorite_thing = "some unknown treat"; // same as default
}
// The "_" case above is a "wildcard" value
// that will match anything.

trace("My dog's name is" + my_dog_name
        + ", and his favorite thing is a: "
        + favorite_thing);

//////////////////////////////////////////////////////////////////
// Expression Statements
//////////////////////////////////////////////////////////////////
trace("***EXPRESSION STATEMENTS***");

/*
   Haxe control statements are very powerful because every statement
   is also an expression, consider:
*/

// if statements
```

```
var k = if (true) 10 else 20;

trace("k equals ", k); // outputs 10

var other_favorite_thing = switch(my_dog_name) {
    case "fido" : "teddy";
    case "rex"  : "stick";
    case "spot" : "football";
    default     : "some unknown treat";
}

trace("My dog's name is" + my_dog_name
        + ", and his other favorite thing is a: "
        + other_favorite_thing);

//////////////////////////////////////////////////////////////////
// Converting Value Types
//////////////////////////////////////////////////////////////////
trace("***CONVERTING VALUE TYPES***");

// You can convert strings to ints fairly easily.

// string to integer
Std.parseInt("0"); // returns 0
Std.parseFloat("0.4"); // returns 0.4;

// integer to string
Std.string(0); // returns "0";
// concatenation with strings will auto-convert to string.
0 + "";  // returns "0";
true + ""; // returns "true";
// See documentation for parsing in Std for more details.


//////////////////////////////////////////////////////////////////
// Dealing with Types
//////////////////////////////////////////////////////////////////

/*

   As mentioned before, Haxe is a statically typed language.  All in
   all, static typing is a wonderful thing.  It enables
   precise autocompletions, and can be used to thoroughly check the
   correctness of a program.  Plus, the Haxe compiler is super fast.

   *HOWEVER*, there are times when you just wish the compiler would let
   something slide, and not throw a type error in a given case.

   To do this, Haxe has two separate keywords.  The first is the
   "Dynamic" type:
 */
var dyn: Dynamic = "any type of variable, such as this string";

/*
```

```
   All that you know for certain with a Dynamic variable is that the
   compiler will no longer worry about what type it is. It is like a
   wildcard variable:  You can pass it instead of any variable type,
   and you can assign any variable type you want.

   The other more extreme option is the "untyped" keyword:
 */

    untyped {
        var x:Int = 'foo'; // this can't be right!
        var y:String = 4; // madness!
    }

/*
   The untyped keyword operates on entire *blocks* of code, skipping
   any type checks that might be otherwise required. This keyword should
   be used very sparingly, such as in limited conditionally-compiled
   situations where type checking is a hinderance.

   In general, skipping type checks is *not* recommended.  Use the
   enum, inheritance, or structural type models in order to help ensure
   the correctness of your program.  Only when you're certain that none
   of the type models work should you resort to "Dynamic" or "untyped".
 */

//////////////////////////////////////////////////////////////////
// Basic Object Oriented Programming
//////////////////////////////////////////////////////////////////
trace("***BASIC OBJECT ORIENTED PROGRAMMING***");


/*
   Create an instance of FooClass.  The classes for this are at the
   end of the file.
 */
var foo_instance = new FooClass(3);

// read the public variable normally
trace(foo_instance.public_any + " is the value for foo_instance.public_any");

// we can read this variable
trace(foo_instance.public_read + " is the value for foo_instance.public_read");
// but not write it
// foo_instance.public_read = 4; // this will throw an error if uncommented:
// trace(foo_instance.public_write); // as will this.

// calls the toString method:
trace(foo_instance + " is the value for foo_instance");
// same thing:
trace(foo_instance.toString() + " is the value for foo_instance.toString()");


/*
   The foo_instance has the "FooClass" type, while acceptBarInstance
```

```
           has the BarClass type.  However, since FooClass extends BarClass, it
           is accepted.
         */
        BarClass.acceptBarInstance(foo_instance);

        /*
           The classes below have some more advanced examples, the "example()"
           method will just run them here.
         */
        SimpleEnumTest.example();
        ComplexEnumTest.example();
        TypedefsAndStructuralTypes.example();
        UsingExample.example();

    }

}

/*
   This is the "child class" of the main LearnHaxe3 Class
 */
class FooClass extends BarClass implements BarInterface{
    public var public_any:Int; // public variables are accessible anywhere
    public var public_read (default, null): Int; // enable only public read
    public var public_write (null, default): Int; // or only public write
    public var property (get, set): Int; // use this style to enable getters/setters

    // private variables are not available outside the class.
    // see @:allow for ways around this.
    var _private:Int; // variables are private if they are not marked public

    // a public constructor
    public function new(arg:Int){
        // call the constructor of the parent object, since we extended BarClass:
        super();

        this.public_any = 0;
        this._private = arg;

    }

    // getter for _private
    function get_property() : Int {
        return _private;
    }

    // setter for _private
    function set_property(val:Int) : Int {
        _private = val;
        return val;
    }

    // special function that is called whenever an instance is cast to a string.
    public function toString(){
        return _private + " with toString() method!";
    }

    // this class needs to have this function defined, since it implements
    // the BarInterface interface.
    public function baseFunction(x: Int) : String{
        // convert the int to string automatically
        return x + " was passed into baseFunction!";
    }
}

/*
    A simple class to extend
*/
class BarClass {
    var base_variable:Int;
    public function new(){
        base_variable = 4;
    }
    public static function acceptBarInstance(b:BarClass){
    }
}

/*
    A simple interface to implement
*/
interface BarInterface{
    public function baseFunction(x:Int):String;
}

//////////////////////////////////////////////////////////////////
// Enums and Switch Statements
//////////////////////////////////////////////////////////////////

/*
   Enums in Haxe are very powerful.  In their simplest form, enums
   are a type with a limited number of states:
 */

enum SimpleEnum {
    Foo;
    Bar;
    Baz;
}

//   Here's a class that uses it:

class SimpleEnumTest{
    public static function example(){
        var e_explicit:SimpleEnum = SimpleEnum.Foo; // you can specify the "full" name
        var e = Foo; // but inference will work as well.
        switch(e){
            case Foo: trace("e was Foo");
            case Bar: trace("e was Bar");
            case Baz: trace("e was Baz"); // comment this line to throw an error.
        }

        /*
           This doesn't seem so different from simple value switches on strings.
           However, if we don't include *all* of the states, the compiler will
           complain.  You can try it by commenting out a line above.

           You can also specify a default for enum switches as well:
         */
        switch(e){
            case Foo: trace("e was Foo again");
            default : trace("default works here too");
        }
    }
}

/*
    Enums go much further than simple states, we can also enumerate
    *constructors*, but we'll need a more complex enum example
 */
enum ComplexEnum{
    IntEnum(i:Int);
    MultiEnum(i:Int, j:String, k:Float);
    SimpleEnumEnum(s:SimpleEnum);
    ComplexEnumEnum(c:ComplexEnum);
}
// Note: The enum above can include *other* enums as well, including itself!
// Note: This is what's called *Algebraic data type* in some other languages.

class ComplexEnumTest{
    public static function example(){
        var e1:ComplexEnum = IntEnum(4); // specifying the enum parameter
        /*
           Now we can switch on the enum, as well as extract any parameters
           it might of had.
         */
        switch(e1){
            case IntEnum(x) : trace('$x was the parameter passed to e1');
            default: trace("Shouldn't be printed");
        }

        // another parameter here that is itself an enum... an enum enum?
        var e2 = SimpleEnumEnum(Foo);
        switch(e2){
            case SimpleEnumEnum(s): trace('$s was the parameter passed to e2');
            default: trace("Shouldn't be printed");
        }

        // enums all the way down
        var e3 = ComplexEnumEnum(ComplexEnumEnum(MultiEnum(4, 'hi', 4.3)));
        switch(e3){
            // You can look for certain nested enums by specifying them explicitly:
            case ComplexEnumEnum(ComplexEnumEnum(MultiEnum(i,j,k))) : {
                trace('$i, $j, and $k were passed into this nested monster');
            }
            default: trace("Shouldn't be printed");
        }
        /*
           Check out "generalized algebraic data types" (GADT) for more details
           on why these are so great.
         */
    }
}

class TypedefsAndStructuralTypes {
    public static function example(){
        /*
           Here we're going to use typedef types, instead of base types.
           At the top we've declared the type "FooString" to mean a "String" type.
         */
        var t1:FooString = "some string";

        /*
           We can use typedefs for "structural types" as well.  These types are
           defined by their field structure, not by class inheritance.  Here's
           an anonymous object with a String field named "foo":
         */

        var anon_obj = { foo: 'hi' };

        /*
           The anon_obj variable doesn't have a type declared, and is an
           anonymous object according to the compiler.  However, remember back at
           the top where we declared the FooObj typedef?  Since anon_obj matches
           that structure, we can use it anywhere that a "FooObject" type is
           expected.
         */

        var f = function(fo:FooObject){
            trace('$fo was passed in to this function');
        }
        f(anon_obj); // call the FooObject signature function with anon_obj.

        /*
           Note that typedefs can have optional fields as well, marked with "?"

           typedef OptionalFooObj = {
                ?optionalString: String,
                requiredInt: Int
           }
         */

        /*
           Typedefs work well with conditional compilation.  For instance,
           we could have included this at the top of the file:

#if( js )
```

```
        typedef Surface = js.html.CanvasRenderingContext2D;
#elseif( nme )
        typedef Surface = nme.display.Graphics;
#elseif( !flash9 )
        typedef Surface = flash8.MovieClip;
#elseif( java )
        typedef Surface = java.awt.geom.GeneralPath;
#end

        That would give us a single "Surface" type to work with across
        all of those platforms.
        */
    }
}

class UsingExample {
    public static function example() {

        /*
           The "using" import keyword is a special type of class import that
           alters the behavior of any static methods in the class.

           In this file, we've applied "using" to "StringTools", which contains
           a number of static methods for dealing with String types.
         */
        trace(StringTools.endsWith("foobar", "bar") + " should be true!");

        /*
           With a "using" import, the first argument type is extended with the
           method.  What does that mean?  Well, since "endsWith" has a first
           argument type of "String", that means all String types now have the
           "endsWith" method:
         */
        trace("foobar".endsWith("bar") + " should be true!");

        /*
           This technique enables a good deal of expression for certain types,
           while limiting the scope of modifications to a single file.

           Note that the String instance is *not* modified in the run time.
           The newly attached method is not really part of the attached
           instance, and the compiler still generates code equivalent to a
           static method.
         */
      }

}
```

We're still only scratching the surface here of what Haxe can do. For a formal overiew of all Haxe features, checkout the online manual, the online api, and "haxelib", the [haxe library repo] (http://lib.haxe.org/).

For more advanced topics, consider checking out:

- Abstract types
- Macros, and Compiler Macros
- Tips and Tricks

Finally, please join us on the mailing list, on IRC #haxe on freenode, or on Google+.

Hy

Hy is a lisp dialect built on top of python. This is achieved by converting hy code to python's abstract syntax tree (ast). This allows hy to call native python code or python to call native hy code as well

This tutorial works for hy   0.9.12

```
;; this gives an gentle introduction to hy for a quick trial head to
;; http://try-hy.appspot.com
;;
; Semicolon comments, like other LISPS

;; s-expression basics
; lisp programs are made of symbolic expressions or sexps which
; resemble
(some-function args)
; now the quintessential hello world
(print "hello world")

;; simple data types
; All simple data types are exactly similar to their python counterparts
; which
42 ; => 42
3.14 ; => 3.14
True ; => True
4+10j ; => (4+10j) a complex number

; lets start with some really simple arithmetic
(+ 4 1) ;=> 5
; the operator is applied to all arguments, like other lisps
(+ 4 1 2 3) ;=> 10
(- 2 1) ;=> 1
(* 4 2) ;=> 8
(/ 4 1) ;=> 4
(% 4 2) ;=> 0 the modulo operator
; power is represented by ** operator like python
(** 3 2) ;=> 9
; nesting forms will do the expected thing
(+ 2 (* 4 2)) ;=> 10
; also logical operators and or not and equal to etc. do as expected
(= 5 4) ;=> False
(not (= 5 4)) ;=> True

;; variables
; variables are set using setv, variable names can use utf-8 except
; for ()[]{}",'`;#|
(setv a 42)
(setv   3.14159)
(def *foo* 42)
;; other container data types
; strings, lists, tuples & dicts
; these are exactly same as python's container types
```

```
"hello world" ;=> "hello world"
; string operations work similar to python
(+ "hello " "world") ;=> "hello world"
; lists are created using [], indexing starts at 0
(setv mylist [1 2 3 4])
; tuples are immutable data structures
(setv mytuple (, 1 2))
; dictionaries are key value pairs
(setv dict1 {"key1" 42 "key2" 21})
; :name can be used to define keywords in hy which can be used for keys
(setv dict2 {:key1 41 :key2 20})
; use `get' to get the element at an index/key
(get mylist 1) ;=> 2
(get dict1 "key1") ;=> 42
; Alternatively if keywords were used they can directly be called
(:key1 dict2) ;=> 41

;; functions and other program constructs
; functions are defined using defn, the last sexp is returned by default
(defn greet [name]
  "A simple greeting" ; an optional docstring
  (print "hello " name))

(greet "bilbo") ;=> "hello bilbo"

; functions can take optional arguments as well as keyword arguments
(defn foolists [arg1 &optional [arg2 2]]
  [arg1 arg2])

(foolists 3) ;=> [3 2]
(foolists 10 3) ;=> [10 3]

; anonymous functions are created using `fn' or `lambda' constructs
; which are similiar to `defn'
(map (fn [x] (* x x)) [1 2 3 4]) ;=> [1 4 9 16]

;; Sequence operations
; hy has some builtin utils for sequence operations etc.
; retrieve the first element using `first' or `car'
(setv mylist [1 2 3 4])
(setv mydict {"a" 1 "b" 2})
(first mylist) ;=> 1

; slice lists using slice
(slice mylist 1 3) ;=> [2 3]

; get elements from a list or dict using `get'
(get mylist 1) ;=> 2
(get mydict "b") ;=> 2
; list indexing starts from 0 same as python
; assoc can set elements at keys/indexes
(assoc mylist 2 10) ; makes mylist [1 2 10 4]
(assoc mydict "c" 3) ; makes mydict {"a" 1 "b" 2 "c" 3}
; there are a whole lot of other core functions which makes working with
; sequences fun

;; Python interop
;; import works just like in python
(import datetime)
(import [functools [partial reduce]]) ; imports fun1 and fun2 from module1
(import [matplotlib.pyplot :as plt]) ; doing an import foo as bar
; all builtin python methods etc. are accessible from hy
; a.foo(arg) is called as (.foo a arg)
(.split (.strip "hello world  ")) ;=> ["hello" "world"]

;; Conditionals
; (if condition (body-if-true) (body-if-false)
(if (= passcode "moria")
  (print "welcome")
  (print "Speak friend, and Enter!"))

; nest multiple if else if clauses with cond
(cond
 [(= someval 42)
  (print "Life, universe and everything else!")]
 [(> someval 42)
  (print "val too large")]
 [(< someval 42)
  (print "val too small")])

; group statements with do, these are executed sequentially
; forms like defn have an implicit do
(do
 (setv someval 10)
 (print "someval is set to " someval)) ;=> 10

; create lexical bindings with `let', all variables defined thusly
; have local scope
(let [[nemesis {"superman" "lex luther"
                "sherlock" "moriarty"
                "seinfeld" "newman"}]]
  (for [(, h v) (.items nemesis)]
    (print (.format "{0}'s nemesis was {1}" h v))))

;; classes
; classes are defined in the following way
(defclass Wizard [object]
  [[--init-- (fn [self spell]
             (setv self.spell spell) ; init the spell attr
             None)]
   [get-spell (fn [self]
              self.spell)]])

;; do checkout hylang.org
```

Further Reading

This tutorial is just a very basic introduction to hy/lisp/python.

Hy docs are here: http://hy.readthedocs.org

Hy's Github repo: http://github.com/hylang/hy

On freenode irc #hy, twitter hashtag #hylang

Java

Java is a general-purpose, concurrent, class-based, object-oriented computer programming language. Read more here.

```
// Single-line comments start with //
/*
Multi-line comments look like this.
*/
/**
JavaDoc comments look like this. Used to describe the Class or various
attributes of a Class.
*/

// Import ArrayList class inside of the java.util package
import java.util.ArrayList;
// Import all classes inside of java.security package
import java.security.*;

// Each .java file contains one outer-level public class, with the same name as
// the file.
public class LearnJava {

    // In order to run a java program, it must have a main method as an entry point.
    public static void main (String[] args) {

        // Use System.out.println() to print lines.
        System.out.println("Hello World!");
        System.out.println(
            "Integer: " + 10 +
            " Double: " + 3.14 +
            " Boolean: " + true);

        // To print without a newline, use System.out.print().
        System.out.print("Hello ");
        System.out.print("World");

        // Use System.out.printf() for easy formatted printing.
        System.out.printf("pi = %.5f", Math.PI); // => pi = 3.14159

        ///////////////////////////////////////
        // Variables
        ///////////////////////////////////////

        /*
```

```
*  Variable Declaration
*/
// Declare a variable using <type> <name>
int fooInt;
// Declare multiple variables of the same type <type> <name1>, <name2>, <name3>
int fooInt1, fooInt2, fooInt3;

/*
*  Variable Initialization
*/

// Initialize a variable using <type> <name> = <val>
int fooInt = 1;
// Initialize multiple variables of same type with same value <type> <name1>, <name2>, <name3>
int fooInt1, fooInt2, fooInt3;
fooInt1 = fooInt2 = fooInt3 = 1;

/*
*  Variable types
*/
// Byte - 8-bit signed two's complement integer
// (-128 <= byte <= 127)
byte fooByte = 100;

// Short - 16-bit signed two's complement integer
// (-32,768 <= short <= 32,767)
short fooShort = 10000;

// Integer - 32-bit signed two's complement integer
// (-2,147,483,648 <= int <= 2,147,483,647)
int fooInt = 1;

// Long - 64-bit signed two's complement integer
// (-9,223,372,036,854,775,808 <= long <= 9,223,372,036,854,775,807)
long fooLong = 100000L;
// L is used to denote that this variable value is of type Long;
// anything without is treated as integer by default.

// Note: Java has no unsigned types.

// Float - Single-precision 32-bit IEEE 754 Floating Point
// 2^-149 <= float <= (2-2^-23) * 2^127
float fooFloat = 234.5f;
// f or F is used to denote that this variable value is of type float;
// otherwise it is treated as double.

// Double - Double-precision 64-bit IEEE 754 Floating Point
// 2^-1074 <= x <= (2-2^-52) * 2^1023
double fooDouble = 123.4;

// Boolean - true & false
boolean fooBoolean = true;
boolean barBoolean = false;
```

```
// Char - A single 16-bit Unicode character
char fooChar = 'A';

// final variables can't be reassigned to another object,
final int HOURS_I_WORK_PER_WEEK = 9001;
// but they can be initialized later.
final double E;
E = 2.71828;


// BigInteger - Immutable arbitrary-precision integers
//
// BigInteger is a data type that allows programmers to manipulate
// integers longer than 64-bits. Integers are stored as an array of
// of bytes and are manipulated using functions built into BigInteger
//
// BigInteger can be initialized using an array of bytes or a string.

BigInteger fooBigInteger = new BigInteger(fooByteArray);


// BigDecimal - Immutable, arbitrary-precision signed decimal number
//
// A BigDecimal takes two parts: an arbitrary precision integer
// unscaled value and a 32-bit integer scale
//
// BigDecimal allows the programmer complete control over decimal
// rounding. It is recommended to use BigDecimal with currency values
// and where exact decimal precision is required.
//
// BigDecimal can be initialized with an int, long, double or String
// or by initializing the unscaled value (BigInteger) and scale (int).

BigDecimal fooBigDecimal = new BigDecimal(fooBigInteger, fooInt);

// Be wary of the constructor that takes a float or double as
// the inaccuracy of the float/double will be copied in BigDecimal.
// Prefer the String constructor when you need an exact value.

BigDecimal tenCents = new BigDecimal("0.1");


// Strings
String fooString = "My String Is Here!";

// \n is an escaped character that starts a new line
String barString = "Printing on a new line?\nNo Problem!";
// \t is an escaped character that adds a tab character
String bazString = "Do you want to add a tab?\tNo Problem!";
System.out.println(fooString);
System.out.println(barString);
System.out.println(bazString);

// Arrays
// The array size must be decided upon instantiation
// The following formats work for declaring an array
// <datatype>[] <var name> = new <datatype>[<array size>];
// <datatype> <var name>[] = new <datatype>[<array size>];
int[] intArray = new int[10];
String[] stringArray = new String[1];
boolean boolArray[] = new boolean[100];

// Another way to declare & initialize an array
int[] y = {9000, 1000, 1337};
String names[] = {"Bob", "John", "Fred", "Juan Pedro"};
boolean bools[] = new boolean[] {true, false, false};

// Indexing an array - Accessing an element
System.out.println("intArray @ 0: " + intArray[0]);

// Arrays are zero-indexed and mutable.
intArray[1] = 1;
System.out.println("intArray @ 1: " + intArray[1]); // => 1

// Others to check out
// ArrayLists - Like arrays except more functionality is offered, and
//              the size is mutable.
// LinkedLists - Implementation of doubly-linked list. All of the
//               operations perform as could be expected for a
//               doubly-linked list.
// Maps - A set of objects that map keys to values. Map is
//        an interface and therefore cannot be instantiated.
//        The type of keys and values contained in a Map must
//        be specified upon instantiation of the implementing
//        class. Each key may map to only one corresponding value,
//        and each key may appear only once (no duplicates).
// HashMaps - This class uses a hashtable to implement the Map
//            interface. This allows the execution time of basic
//            operations, such as get and insert element, to remain
//            constant even for large sets.

///////////////////////////////////////
// Operators
///////////////////////////////////////
System.out.println("\n->Operators");

int i1 = 1, i2 = 2; // Shorthand for multiple declarations

// Arithmetic is straightforward
System.out.println("1+2 = " + (i1 + i2)); // => 3
System.out.println("2-1 = " + (i2 - i1)); // => 1
System.out.println("2*1 = " + (i2 * i1)); // => 2
System.out.println("1/2 = " + (i1 / i2)); // => 0 (int/int returns an int)
System.out.println("1/2 = " + (i1 / (double)i2)); // => 0.5

// Modulo
System.out.println("11%3 = "+(11 % 3)); // => 2
```

```
// Comparison operators
System.out.println("3 == 2? " + (3 == 2)); // => false
System.out.println("3 != 2? " + (3 != 2)); // => true
System.out.println("3 > 2? " + (3 > 2)); // => true
System.out.println("3 < 2? " + (3 < 2)); // => false
System.out.println("2 <= 2? " + (2 <= 2)); // => true
System.out.println("2 >= 2? " + (2 >= 2)); // => true

// Boolean operators
System.out.println("3 > 2 && 2 > 3? " + ((3 > 2) && (2 > 3))); // => false
System.out.println("3 > 2 || 2 > 3? " + ((3 > 2) || (2 > 3))); // => true
System.out.println("!(3 == 2)? " + (!(3 == 2))); // => true

// Bitwise operators!
/*
~       Unary bitwise complement
<<      Signed left shift
>>      Signed/Arithmetic right shift
>>>     Unsigned/Logical right shift
&       Bitwise AND
^       Bitwise exclusive OR
|       Bitwise inclusive OR
*/

// Incrementations
int i = 0;
System.out.println("\n->Inc/Dec-rementation");
// The ++ and -- operators increment and decrement by 1 respectively.
// If they are placed before the variable, they increment then return;
// after the variable they return then increment.
System.out.println(i++); // i = 1, prints 0 (post-increment)
System.out.println(++i); // i = 2, prints 2 (pre-increment)
System.out.println(i--); // i = 1, prints 2 (post-decrement)
System.out.println(--i); // i = 0, prints 0 (pre-decrement)

///////////////////////////////////////
// Control Structures
///////////////////////////////////////
System.out.println("\n->Control Structures");

// If statements are c-like
int j = 10;
if (j == 10) {
    System.out.println("I get printed");
} else if (j > 10) {
    System.out.println("I don't");
} else {
    System.out.println("I also don't");
}

// While loop
int fooWhile = 0;
while(fooWhile < 100) {
    System.out.println(fooWhile);
    // Increment the counter
    // Iterated 100 times, fooWhile 0,1,2...99
    fooWhile++;
}
System.out.println("fooWhile Value: " + fooWhile);

// Do While Loop
int fooDoWhile = 0;
do {
    System.out.println(fooDoWhile);
    // Increment the counter
    // Iterated 99 times, fooDoWhile 0->99
    fooDoWhile++;
} while(fooDoWhile < 100);
System.out.println("fooDoWhile Value: " + fooDoWhile);

// For Loop
// for loop structure => for(<start_statement>; <conditional>; <step>)
for (int fooFor = 0; fooFor < 10; fooFor++) {
    System.out.println(fooFor);
    // Iterated 10 times, fooFor 0->9
}
System.out.println("fooFor Value: " + fooFor);

// Nested For Loop Exit with Label
outer:
for (int i = 0; i < 10; i++) {
  for (int j = 0; j < 10; j++) {
    if (i == 5 && j ==5) {
      break outer;
      // breaks out of outer loop instead of only the inner one
    }
  }
}

// For Each Loop
// The for loop is also able to iterate over arrays as well as objects
// that implement the Iterable interface.
int[] fooList = {1, 2, 3, 4, 5, 6, 7, 8, 9};
// for each loop structure => for (<object> : <iterable>)
// reads as: for each element in the iterable
// note: the object type must match the element type of the iterable.

for (int bar : fooList) {
    System.out.println(bar);
    //Iterates 9 times and prints 1-9 on new lines
}

// Switch Case
// A switch works with the byte, short, char, and int data types.
// It also works with enumerated types (discussed in Enum Types), the
// String class, and a few special classes that wrap primitive types:
// Character, Byte, Short, and Integer.
int month = 3;
```

```
String monthString;
switch (month) {
    case 1: monthString = "January";
            break;
    case 2: monthString = "February";
            break;
    case 3: monthString = "March";
            break;
    default: monthString = "Some other month";
             break;
}
System.out.println("Switch Case Result: " + monthString);

// Starting in Java 7 and above, switching Strings works like this:
String myAnswer = "maybe";
switch(myAnswer) {
    case "yes":
        System.out.println("You answered yes.");
        break;
    case "no":
        System.out.println("You answered no.");
        break;
    case "maybe":
        System.out.println("You answered maybe.");
        break;
    default:
        System.out.println("You answered " + myAnswer);
        break;
}

// Conditional Shorthand
// You can use the '?' operator for quick assignments or logic forks.
// Reads as "If (statement) is true, use <first value>, otherwise, use
// <second value>"
int foo = 5;
String bar = (foo < 10) ? "A" : "B";
System.out.println(bar); // Prints A, because the statement is true


////////////////////////////////////////
// Converting Data Types And Typecasting
////////////////////////////////////////

// Converting data

// Convert String To Integer
Integer.parseInt("123");//returns an integer version of "123"

// Convert Integer To String
Integer.toString(123);//returns a string version of 123

// For other conversions check out the following classes:
// Double
// Long
```

```
    // String

    // Typecasting
    // You can also cast Java objects, there's a lot of details and deals
    // with some more intermediate concepts. Feel free to check it out here:
    // http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html


    ///////////////////////////////////////
    // Classes And Functions
    ///////////////////////////////////////

    System.out.println("\n->Classes & Functions");

    // (definition of the Bicycle class follows)

    // Use new to instantiate a class
    Bicycle trek = new Bicycle();

    // Call object methods
    trek.speedUp(3); // You should always use setter and getter methods
    trek.setCadence(100);

    // toString returns this Object's string representation.
    System.out.println("trek info: " + trek.toString());

    // Double Brace Initialization
    // The Java Language has no syntax for how to create static Collections
    // in an easy way. Usually you end up in the following way:

    private static final Set<String> COUNTRIES = new HashSet<String>();
    static {
       validCodes.add("DENMARK");
       validCodes.add("SWEDEN");
       validCodes.add("FINLAND");
    }

    // But there's a nifty way to achieve the same thing in an
    // easier way, by using something that is called Double Brace
    // Initialization.

    private static final Set<String> COUNTRIES = new HashSet<String>() {{
        add("DENMARK");
        add("SWEDEN");
        add("FINLAND");
    }}

    // The first brace is creating a new AnonymousInnerClass and the
    // second one declares an instance initializer block. This block
    // is called when the anonymous inner class is created.
    // This does not only work for Collections, it works for all
    // non-final classes.

} // End main method
} // End LearnJava class


// You can include other, non-public outer-level classes in a .java file,
// but it is good practice. Instead split classes into separate files.


// Class Declaration Syntax:
// <public/private/protected> class <class name> {
//    // data fields, constructors, functions all inside.
//    // functions are called as methods in Java.
// }

class Bicycle {

    // Bicycle's Fields/Variables
    public int cadence; // Public: Can be accessed from anywhere
    private int speed;  // Private: Only accessible from within the class
    protected int gear; // Protected: Accessible from the class and subclasses
    String name; // default: Only accessible from within this package

    static String className; // Static class variable

    // Static block
    // Java has no implementation of static constructors, but
    // has a static block that can be used to initialize class variables
    // (static variables).
    // This block will be called when the class is loaded.
    static {
        className = "Bicycle";
    }

    // Constructors are a way of creating classes
    // This is a constructor
    public Bicycle() {
        // You can also call another constructor:
        // this(1, 50, 5, "Bontrager");
        gear = 1;
        cadence = 50;
        speed = 5;
        name = "Bontrager";
    }

    // This is a constructor that takes arguments
    public Bicycle(int startCadence, int startSpeed, int startGear,
        String name) {
        this.gear = startGear;
        this.cadence = startCadence;
        this.speed = startSpeed;
        this.name = name;
    }

    // Method Syntax:
    // <public/private/protected> <return type> <function name>(<args>)
```

```java
    // Java classes often implement getters and setters for their fields

    // Method declaration syntax:
    // <access modifier> <return type> <method name>(<args>)
    public int getCadence() {
        return cadence;
    }

    // void methods require no return statement
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

    public void slowDown(int decrement) {
        speed -= decrement;
    }

    public void setName(String newName) {
        name = newName;
    }

    public String getName() {
        return name;
    }

    //Method to display the attribute values of this Object.
    @Override // Inherited from the Object class.
    public String toString() {
        return "gear: " + gear + " cadence: " + cadence + " speed: " + speed +
            " name: " + name;
    }
} // end class Bicycle

// PennyFarthing is a subclass of Bicycle
class PennyFarthing extends Bicycle {
    // (Penny Farthings are those bicycles with the big front wheel.
    // They have no gears.)

    public PennyFarthing(int startCadence, int startSpeed) {
        // Call the parent constructor with super
        super(startCadence, startSpeed, 0, "PennyFarthing");
    }

    // You should mark a method you're overriding with an @annotation.
    // To learn more about what annotations are and their purpose check this
    // out: http://docs.oracle.com/javase/tutorial/java/annotations/
    @Override
    public void setGear(int gear) {
        gear = 0;
    }
}

// Interfaces
// Interface declaration syntax
// <access-level> interface <interface-name> extends <super-interfaces> {
//     // Constants
//     // Method declarations
// }

// Example - Food:
public interface Edible {
    public void eat(); // Any class that implements this interface, must
                       // implement this method.
}

public interface Digestible {
    public void digest();
}


// We can now create a class that implements both of these interfaces.
public class Fruit implements Edible, Digestible {

    @Override
    public void eat() {
        // ...
    }

    @Override
    public void digest() {
        // ...
    }
}

// In Java, you can extend only one class, but you can implement many
// interfaces. For example:
public class ExampleClass extends ExampleClassParent implements InterfaceOne,
    InterfaceTwo {

    @Override
    public void InterfaceOneMethod() {
    }

    @Override
    public void InterfaceTwoMethod() {
    }

}
```

```
// Abstract Classes

// Abstract Class declaration syntax
// <access-level> abstract <abstract-class-name> extends <super-abstract-classes> {
//     // Constants and variables
//     // Method declarations
// }

// Marking a class as abstract means that it contains abstract methods that must
// be defined in a child class. Similar to interfaces, abstract classes cannot
// be instantiated, but instead must be extended and the abstract methods
// defined. Different from interfaces, abstract classes can contain a mixture of
// concrete and abstract methods. Methods in an interface cannot have a body,
// unless the method is static, and variables are final by default, unlike an
// abstract class. Also abstract classes CAN have the "main" method.

public abstract class Animal
{
    public abstract void makeSound();

    // Method can have a body
    public void eat()
    {
        System.out.println("I am an animal and I am Eating.");
        // Note: We can access private variable here.
        age = 30;
    }

    // No need to initialize, however in an interface
    // a variable is implicitly final and hence has
    // to be initialized.
    protected int age;

    public void printAge()
    {
        System.out.println(age);
    }

    // Abstract classes can have main function.
    public static void main(String[] args)
    {
        System.out.println("I am abstract");
    }
}

class Dog extends Animal
{
    // Note still have to override the abstract methods in the
    // abstract class.
    @Override
    public void makeSound()
    {
        System.out.println("Bark");
        // age = 30;    ==> ERROR!  age is private to Animal
    }

    // NOTE: You will get an error if you used the
    // @Override annotation here, since java doesn't allow
    // overriding of static methods.
    // What is happening here is called METHOD HIDING.
    // Check out this awesome SO post: http://stackoverflow.com/questions/16313649/
    public static void main(String[] args)
    {
        Dog pluto = new Dog();
        pluto.makeSound();
        pluto.eat();
        pluto.printAge();
    }
}

// Final Classes

// Final Class declaration syntax
// <access-level> final <final-class-name> {
//     // Constants and variables
//     // Method declarations
// }

// Final classes are classes that cannot be inherited from and are therefore a
// final child. In a way, final classes are the opposite of abstract classes
// because abstract classes must be extended, but final classes cannot be
// extended.
public final class SaberToothedCat extends Animal
{
    // Note still have to override the abstract methods in the
    // abstract class.
    @Override
    public void makeSound()
    {
        System.out.println("Roar");
    }
}

// Final Methods
public abstract class Mammal()
{
    // Final Method Syntax:
    // <access modifier> final <return type> <function name>(<args>)

    // Final methods, like, final classes cannot be overridden by a child class,
    // and are therefore the final implementation of the method.
    public final boolean isWarmBlooded()
    {
        return true;
    }
}
```

```java
// Enum Type
//
// An enum type is a special data type that enables for a variable to be a set
// of predefined constants. The variable must be equal to one of the values that
// have been predefined for it. Because they are constants, the names of an enum
// type's fields are in uppercase letters. In the Java programming language, you
// define an enum type by using the enum keyword. For example, you would specify
// a days-of-the-week enum type as:

public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}

// We can use our enum Day like that:

public class EnumTest {

    // Variable Enum
    Day day;

    public EnumTest(Day day) {
        this.day = day;
    }

    public void tellItLikeItIs() {
        switch (day) {
            case MONDAY:
                System.out.println("Mondays are bad.");
                break;

            case FRIDAY:
                System.out.println("Fridays are better.");
                break;

            case SATURDAY:
            case SUNDAY:
                System.out.println("Weekends are best.");
                break;

            default:
                System.out.println("Midweek days are so-so.");
                break;
        }
    }

    public static void main(String[] args) {
        EnumTest firstDay = new EnumTest(Day.MONDAY);
        firstDay.tellItLikeItIs(); // => Mondays are bad.
        EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
        thirdDay.tellItLikeItIs(); // => Midweek days are so-so.
    }
}
```

```
// Enum types are much more powerful than we show above.
// The enum body can include methods and other fields.
// You can se more at https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
```

Further Reading

The links provided here below are just to get an understanding of the topic, feel free to Google and find specific examples.

Official Oracle Guides:

- Java Tutorial Trail from Sun / Oracle
- Java Access level modifiers
- Object-Oriented Programming Concepts:
    - Inheritance
    - Polymorphism
    - Abstraction
- Exceptions
- Interfaces
- Generics
- Java Code Conventions

Online Practice and Tutorials

- Learneroo.com - Learn Java
- Codingbat.com

Books:

- Head First Java
- Thinking in Java
- Objects First with Java
- Java The Complete Reference

Javascript

JavaScript was created by Netscape's Brendan Eich in 1995. It was originally intended as a simpler scripting language for websites, complementing the use of Java for more complex web applications, but its tight integration with Web pages and built-in support in browsers has caused it to become far more common than Java in web frontends.

JavaScript isn't just limited to web browsers, though: Node.js, a project that provides a standalone runtime for Google Chrome's V8 JavaScript engine, is becoming more and more popular.

JavaScript has a C-like syntax, so if you've used languages like C or Java, a lot of the basic syntax will already be familiar. Despite this, and despite the similarity in name, JavaScript's object model is significantly different to Java's.

```
// Single-line comments start with two slashes.
/* Multiline comments start with slash-star,
   and end with star-slash */
```

```
// Statements can be terminated by ;
doStuff();

// ... but they don't have to be, as semicolons are automatically inserted
// wherever there's a newline, except in certain cases.
doStuff()

// Because those cases can cause unexpected results, we'll keep on using
// semicolons in this guide.

///////////////////////////////////
// 1. Numbers, Strings and Operators

// JavaScript has one number type (which is a 64-bit IEEE 754 double).
// Doubles have a 52-bit mantissa, which is enough to store integers
// up to about 9 10^15 precisely.
3; // = 3
1.5; // = 1.5

// Some basic arithmetic works as you'd expect.
1 + 1; // = 2
0.1 + 0.2; // = 0.30000000000000004
8 - 1; // = 7
10 * 2; // = 20
35 / 5; // = 7

// Including uneven division.
5 / 2; // = 2.5

// And modulo division.
10 % 2; // = 0
30 % 4; // = 2
18.5 % 7; // = 4.5

// Bitwise operations also work; when you perform a bitwise operation your float
// is converted to a signed int *up to* 32 bits.
1 << 2; // = 4

// Precedence is enforced with parentheses.
(1 + 3) * 2; // = 8

// There are three special not-a-real-number values:
Infinity; // result of e.g. 1/0
-Infinity; // result of e.g. -1/0
NaN; // result of e.g. 0/0, stands for 'Not a Number'

// There's also a boolean type.
true;
false;

// Strings are created with ' or ".
'abc';
"Hello, world";
```

```
// Negation uses the ! symbol
!true; // = false
!false; // = true

// Equality is ===
1 === 1; // = true
2 === 1; // = false

// Inequality is !==
1 !== 1; // = false
2 !== 1; // = true

// More comparisons
1 < 10; // = true
1 > 10; // = false
2 <= 2; // = true
2 >= 2; // = true

// Strings are concatenated with +
"Hello " + "world!"; // = "Hello world!"

// ... which works with more than just strings
"1, 2, " + 3; // = "1, 2, 3"
"Hello " + ["world", "!"] // = "Hello world,!"

// and are compared with < and >
"a" < "b"; // = true

// Type coercion is performed for comparisons with double equals...
"5" == 5; // = true
null == undefined; // = true

// ...unless you use ===
"5" === 5; // = false
null === undefined; // = false

// ...which can result in some weird behaviour...
13 + !0; // 14
"13" + !0; // '13true'

// You can access characters in a string with `charAt`
"This is a string".charAt(0);  // = 'T'

// ...or use `substring` to get larger pieces.
"Hello world".substring(0, 5); // = "Hello"

// `length` is a property, so don't use ().
"Hello".length; // = 5

// There's also `null` and `undefined`.
null;      // used to indicate a deliberate non-value
undefined; // used to indicate a value is not currently present (although
           // `undefined` is actually a value itself)
// false, null, undefined, NaN, 0 and "" are falsy; everything else is truthy.
// Note that 0 is falsy and "0" is truthy, even though 0 == "0".

///////////////////////////////////
// 2. Variables, Arrays and Objects

// Variables are declared with the `var` keyword. JavaScript is dynamically
// typed, so you don't need to specify type. Assignment uses a single `=`
// character.
var someVar = 5;

// If you leave the var keyword off, you won't get an error...
someOtherVar = 10;

// ...but your variable will be created in the global scope, not in the scope
// you defined it in.

// Variables declared without being assigned to are set to undefined.
var someThirdVar; // = undefined

// If you want to declare a couple of variables, then you could use a comma
// separator
var someFourthVar = 2, someFifthVar = 4;

// There's shorthand for performing math operations on variables:
someVar += 5; // equivalent to someVar = someVar + 5; someVar is 10 now
someVar *= 10; // now someVar is 100

// and an even-shorter-hand for adding or subtracting 1
someVar++; // now someVar is 101
someVar--; // back to 100

// Arrays are ordered lists of values, of any type.
var myArray = ["Hello", 45, true];

// Their members can be accessed using the square-brackets subscript syntax.
// Array indices start at zero.
myArray[1]; // = 45

// Arrays are mutable and of variable length.
myArray.push("World");
myArray.length; // = 4

// Add/Modify at specific index
myArray[3] = "Hello";

// JavaScript's objects are equivalent to "dictionaries" or "maps" in other
// languages: an unordered collection of key-value pairs.
var myObj = {key1: "Hello", key2: "World"};

// Keys are strings, but quotes aren't required if they're a valid
// JavaScript identifier. Values can be any type.
var myObj = {myKey: "myValue", "my other key": 4};
```

```
// Object attributes can also be accessed using the subscript syntax,
myObj["my other key"]; // = 4

// ... or using the dot syntax, provided the key is a valid identifier.
myObj.myKey; // = "myValue"

// Objects are mutable; values can be changed and new keys added.
myObj.myThirdKey = true;

// If you try to access a value that's not yet set, you'll get undefined.
myObj.myFourthKey; // = undefined

///////////////////////////////////
// 3. Logic and Control Structures

// The `if` structure works as you'd expect.
var count = 1;
if (count == 3){
    // evaluated if count is 3
} else if (count == 4){
    // evaluated if count is 4
} else {
    // evaluated if it's not either 3 or 4
}

// As does `while`.
while (true){
    // An infinite loop!
}

// Do-while loops are like while loops, except they always run at least once.
var input;
do {
    input = getInput();
} while (!isValid(input))

// The `for` loop is the same as C and Java:
// initialisation; continue condition; iteration.
for (var i = 0; i < 5; i++){
    // will run 5 times
}

// The for/in statement iterates over every property across the entire prototype chain.
var description = "";
var person = {fname:"Paul", lname:"Ken", age:18};
for (var x in person){
    description += person[x] + " ";
}

// To only consider properties attached to the object itself
// and not its prototypes, use the `hasOwnProperty()` check.
var description = "";
var person = {fname:"Paul", lname:"Ken", age:18};
for (var x in person){
    if (person.hasOwnProperty(x)){
        description += person[x] + " ";
    }
}

// For/in should not be used to iterate over an Array where the index order
// is important, as there is no guarantee that for/in will return the indexes
// in any particular order.

// && is logical and, || is logical or
if (house.size == "big" && house.colour == "blue"){
    house.contains = "bear";
}
if (colour == "red" || colour == "blue"){
    // colour is either red or blue
}

// && and || "short circuit", which is useful for setting default values.
var name = otherName || "default";


// The `switch` statement checks for equality with `===`.
// Use 'break' after each case
// or the cases after the correct one will be executed too.
grade = 'B';
switch (grade) {
  case 'A':
    console.log("Great job");
    break;
  case 'B':
    console.log("OK job");
    break;
  case 'C':
    console.log("You can do better");
    break;
  default:
    console.log("Oy vey");
    break;
}


///////////////////////////////////
// 4. Functions, Scope and Closures

// JavaScript functions are declared with the `function` keyword.
function myFunction(thing){
    return thing.toUpperCase();
}
myFunction("foo"); // = "FOO"

// Note that the value to be returned must start on the same line as the
// `return` keyword, otherwise you'll always return `undefined` due to
// automatic semicolon insertion. Watch out for this when using Allman style.
function myFunction(){
    return // <- semicolon automatically inserted here
    {thisIsAn: 'object literal'}
}
myFunction(); // = undefined

// JavaScript functions are first class objects, so they can be reassigned to
// different variable names and passed to other functions as arguments - for
// example, when supplying an event handler:
function myFunction(){
    // this code will be called in 5 seconds' time
}
setTimeout(myFunction, 5000);
// Note: setTimeout isn't part of the JS language, but is provided by browsers
// and Node.js.

// Another function provided by browsers is setInterval
function myFunction(){
    // this code will be called every 5 seconds
}
setInterval(myFunction, 5000);

// Function objects don't even have to be declared with a name - you can write
// an anonymous function definition directly into the arguments of another.
setTimeout(function(){
    // this code will be called in 5 seconds' time
}, 5000);

// JavaScript has function scope; functions get their own scope but other blocks
// do not.
if (true){
    var i = 5;
}
i; // = 5 - not undefined as you'd expect in a block-scoped language

// This has led to a common pattern of "immediately-executing anonymous
// functions", which prevent temporary variables from leaking into the global
// scope.
(function(){
    var temporary = 5;
    // We can access the global scope by assigning to the "global object", which
    // in a web browser is always `window`. The global object may have a
    // different name in non-browser environments such as Node.js.
    window.permanent = 10;
})();
temporary; // raises ReferenceError
permanent; // = 10

// One of JavaScript's most powerful features is closures. If a function is
// defined inside another function, the inner function has access to all the
// outer function's variables, even after the outer function exits.
function sayHelloInFiveSeconds(name){
    var prompt = "Hello, " + name + "!";
    // Inner functions are put in the local scope by default, as if they were
    // declared with `var`.
    function inner(){
        alert(prompt);
    }
    setTimeout(inner, 5000);
    // setTimeout is asynchronous, so the sayHelloInFiveSeconds function will
    // exit immediately, and setTimeout will call inner afterwards. However,
    // because inner is "closed over" sayHelloInFiveSeconds, inner still has
    // access to the `prompt` variable when it is finally called.
}
sayHelloInFiveSeconds("Adam"); // will open a popup with "Hello, Adam!" in 5s

///////////////////////////////////
// 5. More about Objects; Constructors and Prototypes

// Objects can contain functions.
var myObj = {
    myFunc: function(){
        return "Hello world!";
    }
};
myObj.myFunc(); // = "Hello world!"

// When functions attached to an object are called, they can access the object
// they're attached to using the `this` keyword.
myObj = {
    myString: "Hello world!",
    myFunc: function(){
        return this.myString;
    }
};
myObj.myFunc(); // = "Hello world!"

// What this is set to has to do with how the function is called, not where
// it's defined. So, our function doesn't work if it isn't called in the
// context of the object.
var myFunc = myObj.myFunc;
myFunc(); // = undefined

// Inversely, a function can be assigned to the object and gain access to it
// through `this`, even if it wasn't attached when it was defined.
var myOtherFunc = function(){
    return this.myString.toUpperCase();
}
myObj.myOtherFunc = myOtherFunc;
myObj.myOtherFunc(); // = "HELLO WORLD!"

// We can also specify a context for a function to execute in when we invoke it
// using `call` or `apply`.

var anotherFunc = function(s){
    return this.myString + s;
}
anotherFunc.call(myObj, " And Hello Moon!"); // = "Hello World! And Hello Moon!"
```

```
// The `apply` function is nearly identical, but takes an array for an argument
// list.

anotherFunc.apply(myObj, [" And Hello Sun!"]); // = "Hello World! And Hello Sun!"

// This is useful when working with a function that accepts a sequence of
// arguments and you want to pass an array.

Math.min(42, 6, 27); // = 6
Math.min([42, 6, 27]); // = NaN (uh-oh!)
Math.min.apply(Math, [42, 6, 27]); // = 6

// But, `call` and `apply` are only temporary. When we want it to stick, we can
// use `bind`.

var boundFunc = anotherFunc.bind(myObj);
boundFunc(" And Hello Saturn!"); // = "Hello World! And Hello Saturn!"

// `bind` can also be used to partially apply (curry) a function.

var product = function(a, b){ return a * b; }
var doubler = product.bind(this, 2);
doubler(8); // = 16

// When you call a function with the `new` keyword, a new object is created, and
// made available to the function via the `this` keyword. Functions designed to be
// called like that are called constructors.

var MyConstructor = function(){
    this.myNumber = 5;
}
myNewObj = new MyConstructor(); // = {myNumber: 5}
myNewObj.myNumber; // = 5

// Every JavaScript object has a 'prototype'. When you go to access a property
// on an object that doesn't exist on the actual object, the interpreter will
// look at its prototype.

// Some JS implementations let you access an object's prototype on the magic
// property `__proto__`. While this is useful for explaining prototypes it's not
// part of the standard; we'll get to standard ways of using prototypes later.
var myObj = {
    myString: "Hello world!"
};
var myPrototype = {
    meaningOfLife: 42,
    myFunc: function(){
        return this.myString.toLowerCase()
    }
};

myObj.__proto__ = myPrototype;
myObj.meaningOfLife; // = 42
```

```
// This works for functions, too.
myObj.myFunc(); // = "hello world!"

// Of course, if your property isn't on your prototype, the prototype's
// prototype is searched, and so on.
myPrototype.__proto__ = {
    myBoolean: true
};
myObj.myBoolean; // = true

// There's no copying involved here; each object stores a reference to its
// prototype. This means we can alter the prototype and our changes will be
// reflected everywhere.
myPrototype.meaningOfLife = 43;
myObj.meaningOfLife; // = 43

// We mentioned that `__proto__` was non-standard, and there's no standard way to
// change the prototype of an existing object. However, there are two ways to
// create a new object with a given prototype.

// The first is Object.create, which is a recent addition to JS, and therefore
// not available in all implementations yet.
var myObj = Object.create(myPrototype);
myObj.meaningOfLife; // = 43

// The second way, which works anywhere, has to do with constructors.
// Constructors have a property called prototype. This is *not* the prototype of
// the constructor function itself; instead, it's the prototype that new objects
// are given when they're created with that constructor and the new keyword.
MyConstructor.prototype = {
    myNumber: 5,
    getMyNumber: function(){
        return this.myNumber;
    }
};
var myNewObj2 = new MyConstructor();
myNewObj2.getMyNumber(); // = 5
myNewObj2.myNumber = 6
myNewObj2.getMyNumber(); // = 6

// Built-in types like strings and numbers also have constructors that create
// equivalent wrapper objects.
var myNumber = 12;
var myNumberObj = new Number(12);
myNumber == myNumberObj; // = true

// Except, they aren't exactly equivalent.
typeof myNumber; // = 'number'
typeof myNumberObj; // = 'object'
myNumber === myNumberObj; // = false
if (0){
    // This code won't execute, because 0 is falsy.
}
if (new Number(0)){
    // This code will execute, because wrapped numbers are objects, and objects
    // are always truthy.
}

// However, the wrapper objects and the regular builtins share a prototype, so
// you can actually add functionality to a string, for instance.
String.prototype.firstCharacter = function(){
    return this.charAt(0);
}
"abc".firstCharacter(); // = "a"

// This fact is often used in "polyfilling", which is implementing newer
// features of JavaScript in an older subset of JavaScript, so that they can be
// used in older environments such as outdated browsers.

// For instance, we mentioned that Object.create isn't yet available in all
// implementations, but we can still use it with this polyfill:
if (Object.create === undefined){ // don't overwrite it if it exists
    Object.create = function(proto){
        // make a temporary constructor with the right prototype
        var Constructor = function(){};
        Constructor.prototype = proto;
        // then use it to create a new, appropriately-prototyped object
        return new Constructor();
    }
}
```

Further Reading

The Mozilla Developer Network provides excellent documentation for JavaScript as it's used in browsers. Plus, it's a wiki, so as you learn more you can help others out by sharing your own knowledge.

MDN's A re-introduction to JavaScript covers much of the concepts covered here in more detail. This guide has quite deliberately only covered the JavaScript language itself; if you want to learn more about how to use JavaScript in web pages, start by learning about the Document Object Model.

Learn Javascript by Example and with Challenges is a variant of this reference with built-in challenges.

JavaScript Garden is an in-depth guide of all the counter-intuitive parts of the language.

JavaScript: The Definitive Guide is a classic guide and reference book.

Eloquent Javascript by Marijn Haverbeke is an excellent JS book/ebook with attached terminal

Eloquent Javascript - The Annotated Version by Gordon Zhu is also a great derivative of Eloquent Javascript with extra explanations and clarifications for some of the more complicated examples.

Javascript: The Right Way is a guide intended to introduce new developers to JavaScript and help experienced developers learn more about its best practices.

In addition to direct contributors to this article, some content is adapted from Louie Dinh's Python tutorial on this site, and the JS Tutorial on the Mozilla Developer Network.

Json

JSON is an extremely simple data-interchange format. As json.org says, it is easy for humans to read and write and for machines to parse and generate.

A piece of JSON must represent either: * A collection of name/value pairs ({ }). In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array. * An ordered list of values ([ ]). In various languages, this is realized as an array, vector, list, or sequence. an array/list/sequence ([ ]) or a dictionary/object/associated array ({ }).

JSON in its purest form has no actual comments, but most parsers will accept C-style (//, /* */) comments. Some parsers also tolerate a trailing comma (i.e. a comma after the last element of an array or the after the last property of an object), but they should be avoided for better compatibility.

For the purposes of this tutorial, everything is going to be 100% valid JSON. Luckily, it kind of speaks for itself.

Supported data types:

- Strings: `"hello"`, `"\"A quote.\""`, `"\u0abe"`, `"Newline.\n"`
- Numbers: `23`, `0.11`, `12e10`, `3.141e-10`, `1.23e+4`
- Objects: `{ "key": "value" }`
- Arrays: `["Values"]`
- Miscellaneous: `true`, `false`, `null`

```json
{
  "key": "value",

  "keys": "must always be enclosed in double quotes",
  "numbers": 0,
  "strings": "Hellø, wørld. All unicode is allowed, along with \"escaping\".",
  "has bools?": true,
  "nothingness": null,

  "big number": 1.2e+100,

  "objects": {
    "comment": "Most of your structure will come from objects.",

    "array": [0, 1, 2, 3, "Arrays can have anything in them.", 5],

    "another object": {
      "comment": "These things can be nested, very useful."
    }
  },

  "silliness": [
    {
      "sources of potassium": ["bananas"]
    },
    [
      [1, 0, 0, 0],
      [0, 1, 0, 0],
      [0, 0, 1, "neo"],
      [0, 0, 0, 1]
    ]
  ],

  "alternative style": {
    "comment": "check this out!"
  , "comma position": "doesn't matter, if it's before the next key, it's valid"
  , "another comment": "how nice"
  },



  "whitespace": "Does not matter.",



  "that was short": "And done. You now know everything JSON has to offer."
}
```

Further Reading

- JSON.org All of JSON beautifully explained using flowchart-like graphics.

Julia

Julia is a new homoiconic functional language focused on technical computing. While having the full power of homoiconic macros, first-class functions, and low-level control, Julia is as easy to learn and use as Python.

This is based on Julia 0.3.

```
# Single line comments start with a hash (pound) symbol.
#= Multiline comments can be written
   by putting '#=' before the text  and '=#'
   after the text. They can also be nested.
=#

####################################################
## 1. Primitive Datatypes and Operators
####################################################

# Everything in Julia is a expression.

# There are several basic types of numbers.
3 # => 3 (Int64)
3.2 # => 3.2 (Float64)
2 + 1im # => 2 + 1im (Complex{Int64})
2//3 # => 2//3 (Rational{Int64})

# All of the normal infix operators are available.
1 + 1 # => 2
8 - 1 # => 7
10 * 2 # => 20
35 / 5 # => 7.0
5 / 2 # => 2.5 # dividing an Int by an Int always results in a Float
div(5, 2) # => 2 # for a truncated result, use div
5 \ 35 # => 7.0
2 ^ 2 # => 4 # power, not bitwise xor
12 % 10 # => 2

# Enforce precedence with parentheses
```

```
(1 + 3) * 2 # => 8

# Bitwise Operators
~2 # => -3   # bitwise not
3 & 5 # => 1 # bitwise and
2 | 4 # => 6 # bitwise or
2 $ 4 # => 6 # bitwise xor
2 >>> 1 # => 1 # logical shift right
2 >> 1  # => 1 # arithmetic shift right
2 << 1  # => 4 # logical/arithmetic shift left

# You can use the bits function to see the binary representation of a number.
bits(12345)
# => "0000000000000000000000000000000000000000000000000011000000111001"
bits(12345.0)
# => "0100000011001000000111001000000000000000000000000000000000000000"

# Boolean values are primitives
true
false

# Boolean operators
!true # => false
!false # => true
1 == 1 # => true
2 == 1 # => false
1 != 1 # => false
2 != 1 # => true
1 < 10 # => true
1 > 10 # => false
2 <= 2 # => true
2 >= 2 # => true
# Comparisons can be chained
1 < 2 < 3 # => true
2 < 3 < 2 # => false

# Strings are created with "
"This is a string."

# Julia has several types of strings, including ASCIIString and UTF8String.
# More on this in the Types section.

# Character literals are written with '
'a'

# Some strings can be indexed like an array of characters
"This is a string"[1] # => 'T' # Julia indexes from 1
# However, this is will not work well for UTF8 strings,
# so iterating over strings is recommended (map, for loops, etc).

# $ can be used for string interpolation:
"2 + 2 = $(2 + 2)" # => "2 + 2 = 4"
# You can put any Julia expression inside the parentheses.
# Another way to format strings is the printf macro.
@printf "%d is less than %f" 4.5 5.3 # 5 is less than 5.300000

# Printing is easy
println("I'm Julia. Nice to meet you!")

# String can be compared lexicographically
"good" > "bye" # => true
"good" == "good" # => true
"1 + 2 = 3" == "1 + 2 = $(1+2)" # => true

####################################################
## 2. Variables and Collections
####################################################

# You don't declare variables before assigning to them.
some_var = 5 # => 5
some_var # => 5

# Accessing a previously unassigned variable is an error
try
    some_other_var # => ERROR: some_other_var not defined
catch e
    println(e)
end

# Variable names start with a letter or underscore.
# After that, you can use letters, digits, underscores, and exclamation points.
SomeOtherVar123! = 6 # => 6

# You can also use certain unicode characters
  = 8 # => 8
# These are especially handy for mathematical notation
2 *   # => 6.283185307179586

# A note on naming conventions in Julia:
#
# * Word separation can be indicated by underscores ('_'), but use of
#   underscores is discouraged unless the name would be hard to read
#   otherwise.
#
# * Names of Types begin with a capital letter and word separation is shown
#   with CamelCase instead of underscores.
#
# * Names of functions and macros are in lower case, without underscores.
#
# * Functions that modify their inputs have names that end in !. These
#   functions are sometimes called mutating functions or in-place functions.

# Arrays store a sequence of values indexed by integers 1 through n:
a = Int64[] # => 0-element Int64 Array

# 1-dimensional array literals can be written with comma-separated values.
b = [4, 5, 6] # => 3-element Int64 Array: [4, 5, 6]
```

```
b = [4; 5; 6] # => 3-element Int64 Array: [4, 5, 6]
b[1] # => 4
b[end] # => 6

# 2-dimentional arrays use space-separated values and semicolon-separated rows.
matrix = [1 2; 3 4] # => 2x2 Int64 Array: [1 2; 3 4]

# Arrays of a particular Type
b = Int8[4, 5, 6] # => 3-element Int8 Array: [4, 5, 6]

# Add stuff to the end of a list with push! and append!
push!(a,1)     # => [1]
push!(a,2)     # => [1,2]
push!(a,4)     # => [1,2,4]
push!(a,3)     # => [1,2,4,3]
append!(a,b) # => [1,2,4,3,4,5,6]

# Remove from the end with pop
pop!(b)        # => 6 and b is now [4,5]

# Let's put it back
push!(b,6)   # b is now [4,5,6] again.

a[1] # => 1 # remember that Julia indexes from 1, not 0!

# end is a shorthand for the last index. It can be used in any
# indexing expression
a[end] # => 6

# we also have shift and unshift
shift!(a) # => 1 and a is now [2,4,3,4,5,6]
unshift!(a,7) # => [7,2,4,3,4,5,6]

# Function names that end in exclamations points indicate that they modify
# their argument.
arr = [5,4,6] # => 3-element Int64 Array: [5,4,6]
sort(arr) # => [4,5,6]; arr is still [5,4,6]
sort!(arr) # => [4,5,6]; arr is now [4,5,6]

# Looking out of bounds is a BoundsError
try
    a[0] # => ERROR: BoundsError() in getindex at array.jl:270
    a[end+1] # => ERROR: BoundsError() in getindex at array.jl:270
catch e
    println(e)
end

# Errors list the line and file they came from, even if it's in the standard
# library. If you built Julia from source, you can look in the folder base
# inside the julia folder to find these files.

# You can initialize arrays from ranges
a = [1:5;] # => 5-element Int64 Array: [1,2,3,4,5]
```

```
# You can look at ranges with slice syntax.
a[1:3] # => [1, 2, 3]
a[2:end] # => [2, 3, 4, 5]

# Remove elements from an array by index with splice!
arr = [3,4,5]
splice!(arr,2) # => 4 ; arr is now [3,5]

# Concatenate lists with append!
b = [1,2,3]
append!(a,b) # Now a is [1, 2, 3, 4, 5, 1, 2, 3]

# Check for existence in a list with in
in(1, a) # => true

# Examine the length with length
length(a) # => 8

# Tuples are immutable.
tup = (1, 2, 3) # => (1,2,3) # an (Int64,Int64,Int64) tuple.
tup[1] # => 1
try:
    tup[1] = 3 # => ERROR: no method setindex!((Int64,Int64,Int64),Int64,Int64)
catch e
    println(e)
end

# Many list functions also work on tuples
length(tup) # => 3
tup[1:2] # => (1,2)
in(2, tup) # => true

# You can unpack tuples into variables
a, b, c = (1, 2, 3) # => (1,2,3)  # a is now 1, b is now 2 and c is now 3

# Tuples are created even if you leave out the parentheses
d, e, f = 4, 5, 6 # => (4,5,6)

# A 1-element tuple is distinct from the value it contains
(1,) == 1 # => false
(1) == 1 # => true

# Look how easy it is to swap two values
e, d = d, e  # => (5,4) # d is now 5 and e is now 4


# Dictionaries store mappings
empty_dict = Dict() # => Dict{Any,Any}()

# You can create a dictionary using a literal
filled_dict = ["one"=> 1, "two"=> 2, "three"=> 3]
# => Dict{ASCIIString,Int64}

# Look up values with []
```

```
filled_dict["one"] # => 1

# Get all keys
keys(filled_dict)
# => KeyIterator{Dict{ASCIIString,Int64}}(["three"=>3,"one"=>1,"two"=>2])
# Note - dictionary keys are not sorted or in the order you inserted them.

# Get all values
values(filled_dict)
# => ValueIterator{Dict{ASCIIString,Int64}}(["three"=>3,"one"=>1,"two"=>2])
# Note - Same as above regarding key ordering.

# Check for existence of keys in a dictionary with in, haskey
in(("one", 1), filled_dict) # => true
in(("two", 3), filled_dict) # => false
haskey(filled_dict, "one") # => true
haskey(filled_dict, 1) # => false

# Trying to look up a non-existent key will raise an error
try
    filled_dict["four"] # => ERROR: key not found: four in getindex at dict.jl:489
catch e
    println(e)
end

# Use the get method to avoid that error by providing a default value
# get(dictionary,key,default_value)
get(filled_dict,"one",4) # => 1
get(filled_dict,"four",4) # => 4

# Use Sets to represent collections of unordered, unique values
empty_set = Set() # => Set{Any}()
# Initialize a set with values
filled_set = Set(1,2,2,3,4) # => Set{Int64}(1,2,3,4)

# Add more values to a set
push!(filled_set,5) # => Set{Int64}(5,4,2,3,1)

# Check if the values are in the set
in(2, filled_set) # => true
in(10, filled_set) # => false

# There are functions for set intersection, union, and difference.
other_set = Set(3, 4, 5, 6) # => Set{Int64}(6,4,5,3)
intersect(filled_set, other_set) # => Set{Int64}(3,4,5)
union(filled_set, other_set) # => Set{Int64}(1,2,3,4,5,6)
setdiff(Set(1,2,3,4),Set(2,3,5)) # => Set{Int64}(1,4)


####################################################
## 3. Control Flow
####################################################

# Let's make a variable
```

```
some_var = 5

# Here is an if statement. Indentation is not meaningful in Julia.
if some_var > 10
    println("some_var is totally bigger than 10.")
elseif some_var < 10    # This elseif clause is optional.
    println("some_var is smaller than 10.")
else                    # The else clause is optional too.
    println("some_var is indeed 10.")
end
# => prints "some var is smaller than 10"


# For loops iterate over iterables.
# Iterable types include Range, Array, Set, Dict, and AbstractString.
for animal=["dog", "cat", "mouse"]
    println("$animal is a mammal")
    # You can use $ to interpolate variables or expression into strings
end
# prints:
#    dog is a mammal
#    cat is a mammal
#    mouse is a mammal

# You can use 'in' instead of '='.
for animal in ["dog", "cat", "mouse"]
    println("$animal is a mammal")
end
# prints:
#    dog is a mammal
#    cat is a mammal
#    mouse is a mammal

for a in ["dog"=>"mammal","cat"=>"mammal","mouse"=>"mammal"]
    println("$(a[1]) is a $(a[2])")
end
# prints:
#    dog is a mammal
#    cat is a mammal
#    mouse is a mammal

for (k,v) in ["dog"=>"mammal","cat"=>"mammal","mouse"=>"mammal"]
    println("$k is a $v")
end
# prints:
#    dog is a mammal
#    cat is a mammal
#    mouse is a mammal

# While loops loop while a condition is true
x = 0
while x < 4
    println(x)
    x += 1  # Shorthand for x = x + 1
```

```
end
# prints:
#   0
#   1
#   2
#   3

# Handle exceptions with a try/catch block
try
   error("help")
catch e
   println("caught it $e")
end
# => caught it ErrorException("help")


####################################################
## 4. Functions
####################################################

# The keyword 'function' creates new functions
#function name(arglist)
#  body...
#end
function add(x, y)
    println("x is $x and y is $y")

    # Functions return the value of their last statement
    x + y
end

add(5, 6) # => 11 after printing out "x is 5 and y is 6"

# Compact assignment of functions
f_add(x, y) = x + y # => "f (generic function with 1 method)"
f_add(3, 4) # => 7

# Function can also return multiple values as tuple
f(x, y) = x + y, x - y
f(3, 4) # => (7, -1)

# You can define functions that take a variable number of
# positional arguments
function varargs(args...)
    return args
    # use the keyword return to return anywhere in the function
end
# => varargs (generic function with 1 method)

varargs(1,2,3) # => (1,2,3)

# The ... is called a splat.
# We just used it in a function definition.
# It can also be used in a fuction call,
# where it will splat an Array or Tuple's contents into the argument list.
Set([1,2,3])    # => Set{Array{Int64,1}}([1,2,3]) # produces a Set of Arrays
Set([1,2,3]...) # => Set{Int64}(1,2,3) # this is equivalent to Set(1,2,3)

x = (1,2,3)     # => (1,2,3)
Set(x)          # => Set{(Int64,Int64,Int64)}((1,2,3)) # a Set of Tuples
Set(x...)       # => Set{Int64}(2,3,1)


# You can define functions with optional positional arguments
function defaults(a,b,x=5,y=6)
    return "$a $b and $x $y"
end

defaults('h','g') # => "h g and 5 6"
defaults('h','g','j') # => "h g and j 6"
defaults('h','g','j','k') # => "h g and j k"
try
    defaults('h') # => ERROR: no method defaults(Char,)
    defaults() # => ERROR: no methods defaults()
catch e
    println(e)
end

# You can define functions that take keyword arguments
function keyword_args(;k1=4,name2="hello") # note the ;
    return ["k1"=>k1,"name2"=>name2]
end

keyword_args(name2="ness") # => ["name2"=>"ness","k1"=>4]
keyword_args(k1="mine") # => ["k1"=>"mine","name2"=>"hello"]
keyword_args() # => ["name2"=>"hello","k1"=>4]

# You can combine all kinds of arguments in the same function
function all_the_args(normal_arg, optional_positional_arg=2; keyword_arg="foo")
    println("normal arg: $normal_arg")
    println("optional arg: $optional_positional_arg")
    println("keyword arg: $keyword_arg")
end

all_the_args(1, 3, keyword_arg=4)
# prints:
#   normal arg: 1
#   optional arg: 3
#   keyword arg: 4

# Julia has first class functions
function create_adder(x)
    adder = function (y)
        return x + y
    end
    return adder
end
```

```
# This is "stabby lambda syntax" for creating anonymous functions
(x -> x > 2)(3) # => true

# This function is identical to create_adder implementation above.
function create_adder(x)
    y -> x + y
end

# You can also name the internal function, if you want
function create_adder(x)
    function adder(y)
        x + y
    end
    adder
end

add_10 = create_adder(10)
add_10(3) # => 13


# There are built-in higher order functions
map(add_10, [1,2,3]) # => [11, 12, 13]
filter(x -> x > 5, [3, 4, 5, 6, 7]) # => [6, 7]

# We can use list comprehensions for nicer maps
[add_10(i) for i=[1, 2, 3]] # => [11, 12, 13]
[add_10(i) for i in [1, 2, 3]] # => [11, 12, 13]

####################################################
## 5. Types
####################################################

# Julia has a type system.
# Every value has a type; variables do not have types themselves.
# You can use the `typeof` function to get the type of a value.
typeof(5) # => Int64

# Types are first-class values
typeof(Int64) # => DataType
typeof(DataType) # => DataType
# DataType is the type that represents types, including itself.

# Types are used for documentation, optimizations, and dispatch.
# They are not statically checked.

# Users can define types
# They are like records or structs in other languages.
# New types are defined using the `type` keyword.

# type Name
#   field::OptionalType
#   ...
# end
type Tiger
```

```
  taillength::Float64
  coatcolor # not including a type annotation is the same as `::Any`
end

# The default constructor's arguments are the properties
# of the type, in the order they are listed in the definition
tigger = Tiger(3.5,"orange") # => Tiger(3.5,"orange")

# The type doubles as the constructor function for values of that type
sherekhan = typeof(tigger)(5.6,"fire") # => Tiger(5.6,"fire")

# These struct-style types are called concrete types
# They can be instantiated, but cannot have subtypes.
# The other kind of types is abstract types.

# abstract Name
abstract Cat # just a name and point in the type hierarchy

# Abstract types cannot be instantiated, but can have subtypes.
# For example, Number is an abstract type
subtypes(Number) # => 6-element Array{Any,1}:
                 #     Complex{Float16}
                 #     Complex{Float32}
                 #     Complex{Float64}
                 #     Complex{T<:Real}
                 #     ImaginaryUnit
                 #     Real
subtypes(Cat) # => 0-element Array{Any,1}

# AbstractString, as the name implies, is also an abstract type
subtypes(AbstractString)    # 8-element Array{Any,1}:
                            #  Base.SubstitutionString{T<:AbstractString}
                            #  DirectIndexString
                            #  RepString
                            #  RevString{T<:AbstractString}
                            #  RopeString
                            #  SubString{T<:AbstractString}
                            #  UTF16String
                            #  UTF8String

# Every type has a super type; use the `super` function to get it.
typeof(5) # => Int64
super(Int64) # => Signed
super(Signed) # => Real
super(Real) # => Number
super(Number) # => Any
super(super(Signed)) # => Number
super(Any) # => Any
# All of these type, except for Int64, are abstract.
typeof("fire") # => ASCIIString
super(ASCIIString) # => DirectIndexString
super(DirectIndexString) # => AbstractString
# Likewise here with ASCIIString
# <: is the subtyping operator
type Lion <: Cat # Lion is a subtype of Cat
  mane_color
  roar::AbstractString
end

# You can define more constructors for your type
# Just define a function of the same name as the type
# and call an existing constructor to get a value of the correct type
Lion(roar::AbstractString) = Lion("green",roar)
# This is an outer constructor because it's outside the type definition

type Panther <: Cat # Panther is also a subtype of Cat
  eye_color
  Panther() = new("green")
  # Panthers will only have this constructor, and no default constructor.
end
# Using inner constructors, like Panther does, gives you control
# over how values of the type can be created.
# When possible, you should use outer constructors rather than inner ones.

####################################################
## 6. Multiple-Dispatch
####################################################

# In Julia, all named functions are generic functions
# This means that they are built up from many small methods
# Each constructor for Lion is a method of the generic function Lion.

# For a non-constructor example, let's make a function meow:

# Definitions for Lion, Panther, Tiger
function meow(animal::Lion)
  animal.roar # access type properties using dot notation
end

function meow(animal::Panther)
  "grrr"
end

function meow(animal::Tiger)
  "rawwwr"
end

# Testing the meow function
meow(tigger) # => "rawwr"
meow(Lion("brown","ROAAR")) # => "ROAAR"
meow(Panther()) # => "grrr"

# Review the local type hierarchy
issubtype(Tiger,Cat) # => false
issubtype(Lion,Cat) # => true
issubtype(Panther,Cat) # => true
```

```
# Defining a function that takes Cats
function pet_cat(cat::Cat)
  println("The cat says $(meow(cat))")
end

pet_cat(Lion("42")) # => prints "The cat says 42"
try
    pet_cat(tigger) # => ERROR: no method pet_cat(Tiger,)
catch e
    println(e)
end

# In OO languages, single dispatch is common;
# this means that the method is picked based on the type of the first argument.
# In Julia, all of the argument types contribute to selecting the best method.

# Let's define a function with more arguments, so we can see the difference
function fight(t::Tiger,c::Cat)
  println("The $(t.coatcolor) tiger wins!")
end
# => fight (generic function with 1 method)

fight(tigger,Panther()) # => prints The orange tiger wins!
fight(tigger,Lion("ROAR")) # => prints The orange tiger wins!

# Let's change the behavior when the Cat is specifically a Lion
fight(t::Tiger,l::Lion) = println("The $(l.mane_color)-maned lion wins!")
# => fight (generic function with 2 methods)

fight(tigger,Panther()) # => prints The orange tiger wins!
fight(tigger,Lion("ROAR")) # => prints The green-maned lion wins!

# We don't need a Tiger in order to fight
fight(l::Lion,c::Cat) = println("The victorious cat says $(meow(c))")
# => fight (generic function with 3 methods)

fight(Lion("balooga!"),Panther()) # => prints The victorious cat says grrr
try
  fight(Panther(),Lion("RAWR")) # => ERROR: no method fight(Panther,Lion)
catch
end

# Also let the cat go first
fight(c::Cat,l::Lion) = println("The cat beats the Lion")
# => Warning: New definition
#    fight(Cat,Lion) at none:1
# is ambiguous with
#    fight(Lion,Cat) at none:2.
# Make sure
#    fight(Lion,Lion)
# is defined first.
#fight (generic function with 4 methods)

# This warning is because it's unclear which fight will be called in:
```

```
fight(Lion("RAR"),Lion("brown","rarrr")) # => prints The victorious cat says rarrr
# The result may be different in other versions of Julia

fight(l::Lion,l2::Lion) = println("The lions come to a tie")
fight(Lion("RAR"),Lion("brown","rarrr")) # => prints The lions come to a tie


# Under the hood
# You can take a look at the llvm  and the assembly code generated.

square_area(l) = l * l      # square_area (generic function with 1 method)

square_area(5) #25

# What happens when we feed square_area an integer?
code_native(square_area, (Int32,))
    #       .section    __TEXT,__text,regular,pure_instructions
    #   Filename: none
    #   Source line: 1              # Prologue
    #       push    RBP
    #       mov RBP, RSP
    #   Source line: 1
    #       movsxd  RAX, EDI        # Fetch l from memory?
    #       imul    RAX, RAX        # Square l and store the result in RAX
    #       pop RBP                 # Restore old base pointer
    #       ret                     # Result will still be in RAX

code_native(square_area, (Float32,))
    #       .section    __TEXT,__text,regular,pure_instructions
    #   Filename: none
    #   Source line: 1
    #       push    RBP
    #       mov RBP, RSP
    #   Source line: 1
    #       vmulss  XMM0, XMM0, XMM0  # Scalar single precision multiply (AVX)
    #       pop RBP
    #       ret

code_native(square_area, (Float64,))
    #       .section    __TEXT,__text,regular,pure_instructions
    #   Filename: none
    #   Source line: 1
    #       push    RBP
    #       mov RBP, RSP
    #   Source line: 1
    #       vmulsd  XMM0, XMM0, XMM0 # Scalar double precision multiply (AVX)
    #       pop RBP
    #       ret
    #
# Note that julia will use floating point instructions if any of the
# arguments are floats.
# Let's calculate the area of a circle
circle_area(r) = pi * r * r     # circle_area (generic function with 1 method)
circle_area(5)                  # 78.53981633974483
```

```
code_native(circle_area, (Int32,))
    #       .section    __TEXT,__text,regular,pure_instructions
    #   Filename: none
    #   Source line: 1
    #       push    RBP
    #       mov RBP, RSP
    #   Source line: 1
    #       vcvtsi2sd   XMM0, XMM0, EDI          # Load integer (r) from memory
    #       movabs  RAX, 4593140240              # Load pi
    #       vmulsd  XMM1, XMM0, QWORD PTR [RAX]  # pi * r
    #       vmulsd  XMM0, XMM0, XMM1             # (pi * r) * r
    #       pop RBP
    #       ret
    #

code_native(circle_area, (Float64,))
    #       .section    __TEXT,__text,regular,pure_instructions
    #   Filename: none
    #   Source line: 1
    #       push    RBP
    #       mov RBP, RSP
    #       movabs  RAX, 4593140496
    #   Source line: 1
    #       vmulsd  XMM1, XMM0, QWORD PTR [RAX]
    #       vmulsd  XMM0, XMM1, XMM0
    #       pop RBP
    #       ret
    #
```

Further Reading

You can get a lot more detail from The Julia Manual

The best place to get help with Julia is the (very friendly) mailing list.

Less

Less is a CSS pre-processor, that adds features such as variables, nesting, mixins and more. Less (and other preprocessors, such as Sass help developers to write maintainable and DRY (Don't Repeat Yourself) code.

```
//Single line comments are removed when Less is compiled to CSS.

/*Multi line comments are preserved. */


/*Variables
==============================*/
```

```
/* You can store a CSS value (such as a color) in a variable.
Use the '@' symbol to create a variable. */

@primary-color: #A3A4FF;
@secondary-color: #51527F;
@body-font: 'Roboto', sans-serif;

/* You can use the variables throughout your stylesheet.
Now if you want to change a color, you only have to make the change once.*/

body {
    background-color: @primary-color;
    color: @secondary-color;
    font-family: @body-font;
}

/* This would compile to: */
body {
    background-color: #A3A4FF;
    color: #51527F;
    font-family: 'Roboto', sans-serif;
}


/* This is much more maintainable than having to change the color
each time it appears throughout your stylesheet. */


/*Mixins
==============================*/



/* If you find you are writing the same code for more than one
element, you might want to reuse that easily.*/

.center {
    display: block;
    margin-left: auto;
    margin-right: auto;
    left: 0;
    right: 0;
}

/* You can use the mixin by simply adding the selector as a style */

div {
    .center;
    background-color: @primary-color;
}

/*Which would compile to: */
.center {
  display: block;
  margin-left: auto;
  margin-right: auto;
  left: 0;
  right: 0;
}
div {
    display: block;
    margin-left: auto;
    margin-right: auto;
    left: 0;
    right: 0;
    background-color: #A3A4FF;
}

/* You can omit the mixin code from being compiled by adding paranthesis
   after the selector */

.center() {
  display: block;
  margin-left: auto;
  margin-right: auto;
  left: 0;
  right: 0;
}

div {
  .center;
  background-color: @primary-color;
}

/*Which would compile to: */
div {
  display: block;
  margin-left: auto;
  margin-right: auto;
  left: 0;
  right: 0;
  background-color: #A3A4FF;
}


/*Functions
==============================*/



/* Less provides functions that can be used to accomplish a variety of
   tasks. Consider the following */

/* Functions can be invoked by using their name and passing in the
   required arguments */
body {
  width: round(10.25px);
}
```

```
.footer {
  background-color: fadeout(#000000, 0.25)
}

/* Compiles to: */

body {
  width: 10px;
}

.footer {
  background-color: rgba(0, 0, 0, 0.75);
}

/* You may also define your own functions. Functions are very similar to
   mixins. When trying to choose between a function or a mixin, remember
   that mixins are best for generating CSS while functions are better for
   logic that might be used throughout your Less code. The examples in
   the Math Operators' section are ideal candidates for becoming a reusable
   function. */

/* This function will take a target size and the parent size and calculate
   and return the percentage */

.average(@x, @y) {
  @average_result: ((@x + @y) / 2);
}

div {
  .average(16px, 50px); // "call" the mixin
  padding: @average_result;    // use its "return" value
}

/* Compiles to: */

div {
  padding: 33px;
}

/*Extend (Inheritance)
==============================*/



/*Extend is a way to share the properties of one selector with another. */

.display {
  height: 50px;
}

.display-success {
  &:extend(.display);
    border-color: #22df56;
```

```
}

/* Compiles to: */
.display,
.display-success {
  height: 50px;
}
.display-success {
  border-color: #22df56;
}

/* Extending a CSS statement is preferable to creating a mixin
   because of the way it groups together the classes that all share
   the same base styling. If this was done with a mixin, the properties
   would be duplicated for each statement that
   called the mixin. While it won't affect your workflow, it will
   add unnecessary bloat to the files created by the Less compiler. */



/*Nesting
==============================*/



/*Less allows you to nest selectors within selectors */

ul {
    list-style-type: none;
    margin-top: 2em;

    li {
        background-color: #FF0000;
    }
}

/* '&' will be replaced by the parent selector. */
/* You can also nest pseudo-classes. */
/* Keep in mind that over-nesting will make your code less maintainable.
Best practices recommend going no more than 3 levels deep when nesting.
For example: */

ul {
    list-style-type: none;
    margin-top: 2em;

    li {
        background-color: red;

        &:hover {
          background-color: blue;
        }

        a {
          color: white;
        }
    }
}

/* Compiles to: */

ul {
  list-style-type: none;
  margin-top: 2em;
}

ul li {
  background-color: red;
}

ul li:hover {
  background-color: blue;
}

ul li a {
  color: white;
}



/*Partials and Imports
==============================*/



/* Less allows you to create partial files. This can help keep your Less
   code modularized. Partial files conventionally begin with an '_',
   e.g. _reset.less. and are imported into a main less file that gets
   compiled into CSS */

/* Consider the following CSS which we'll put in a file called _reset.less */

html,
body,
ul,
ol {
  margin: 0;
  padding: 0;
}

/* Less offers @import which can be used to import partials into a file.
   This differs from the traditional CSS @import statement which makes
   another HTTP request to fetch the imported file. Less takes the
   imported file and combines it with the compiled code. */

@import 'reset';

body {
  font-size: 16px;
  font-family: Helvetica, Arial, Sans-serif;
}

/* Compiles to: */

html, body, ul, ol {
  margin: 0;
  padding: 0;
}

body {
  font-size: 16px;
  font-family: Helvetica, Arial, Sans-serif;
}


/*Math Operations
==============================*/



/* Less provides the following operators: +, -, *, /, and %. These can
   be useful for calculating values directly in your Less files instead
   of using values that you've already calculated by hand. Below is an example
   of a setting up a simple two column design. */

@content-area: 960px;
@main-content: 600px;
@sidebar-content: 300px;

@main-size: @main-content / @content-area * 100%;
@sidebar-size: @sidebar-content / @content-area * 100%;
@gutter: 100% - (@main-size + @sidebar-size);

body {
  width: 100%;
}

.main-content {
  width: @main-size;
}

.sidebar {
  width: @sidebar-size;
}

.gutter {
  width: @gutter;
}

/* Compiles to: */

body {
  width: 100%;
}

.main-content {
  width: 62.5%;
}

.sidebar {
  width: 31.25%;
}

.gutter {
  width: 6.25%;
}
```

Practice Less

If you want to play with Less in your browser, check out LESS2CSS.

Compatibility

Less can be used in any project as long as you have a program to compile it into CSS. You'll want to verify that the CSS you're using is compatible with your target browsers.

QuirksMode CSS and CanIUse are great resources for checking compatibility.

Further reading

- Official Documentation

Livescript

LiveScript is a functional compile-to-JavaScript language which shares most of the underlying semantics with its host language. Nice additions comes with currying, function composition, pattern matching and lots of other goodies heavily borrowed from languages like Haskell, F# and Scala.

LiveScript is a fork of Coco, which is itself a fork of CoffeeScript. The language is stable, and a new version is in active development to bring a plethora of new niceties!

Feedback is always welcome, so feel free to reach me over at [@kurisuwhyte](https://twitter.com/kurisuwhyte) :)

```
# Just like its CoffeeScript cousin, LiveScript uses number symbols for
# single-line comments.

/*
 Multi-line comments are written C-style. Use them if you want comments
 to be preserved in the JavaScript output.
 */

# As far as syntax goes, LiveScript uses indentation to delimit blocks,
# rather than curly braces, and whitespace to apply functions, rather
# than parenthesis.


########################################################################
## 1. Basic values
########################################################################

# Lack of value is defined by the keyword `void` instead of `undefined`
void            # same as `undefined` but safer (can't be overridden)

# No valid value is represented by Null.
null


# The most basic actual value is the logical type:
true
false

# And it has a plethora of aliases that mean the same thing:
on; off
yes; no


# Then you get numbers. These are double-precision floats like in JS.
10
0.4     # Note that the leading `0` is required

# For readability, you may use underscores and letter suffixes in a
# number, and these will be ignored by the compiler.
12_344km


# Strings are immutable sequences of characters, like in JS:
"Christina"             # apostrophes are okay too!
"""Multi-line
   strings
   are
   okay
   too."""

# Sometimes you want to encode a keyword, the backslash notation makes
# this easy:
\keyword                # => 'keyword'


# Arrays are ordered collections of values.
fruits =
  * \apple
  * \orange
  * \pear

# They can be expressed more concisely with square brackets:
fruits = [ \apple, \orange, \pear ]
```

```
# You also get a convenient way to create a list of strings, using
# white space to delimit the items.
fruits = <[ apple orange pear ]>

# You can retrieve an item by their 0-based index:
fruits[0]       # => "apple"

# Objects are a collection of unordered key/value pairs, and a few other
# things (more on that later).
person =
  name: "Christina"
  likes:
    * "kittens"
    * "and other cute stuff"

# Again, you can express them concisely with curly brackets:
person = {name: "Christina", likes: ["kittens", "and other cute stuff"]}

# You can retrieve an item by their key:
person.name     # => "Christina"
person["name"]  # => "Christina"


# Regular expressions use the same syntax as JavaScript:
trailing-space = /\s$/          # dashed-words become dashedWords

# Except you can do multi-line expressions too!
# (comments and whitespace just gets ignored)
funRE = //
        function\s+(.+)         # name
        \s* \((.*)\) \s*        # arguments
        { (.*) }                # body
        //


########################################################################
## 2. Basic operations
########################################################################

# Arithmetic operators are the same as JavaScript's:
1 + 2   # => 3
2 - 1   # => 1
2 * 3   # => 6
4 / 2   # => 2
3 % 2   # => 1


# Comparisons are mostly the same too, except that `==` is the same as
# JS's `===`, where JS's `==` in LiveScript is `~=`, and `===` enables
# object and array comparisons, and also stricter comparisons:
2 == 2          # => true
2 == "2"        # => false
2 ~= "2"        # => true
2 === "2"       # => false
```

```
[1,2,3] == [1,2,3]        # => false
[1,2,3] === [1,2,3]       # => true

+0 == -0     # => true
+0 === -0    # => false

# Other relational operators include <, <=, > and >=

# Logical values can be combined through the logical operators `or`,
# `and` and `not`
true and false  # => false
false or true   # => true
not false       # => true


# Collections also get some nice additional operators
[1, 2] ++ [3, 4]                # => [1, 2, 3, 4]
'a' in <[ a b c ]>              # => true
'name' of { name: 'Chris' }     # => true


########################################################################
## 3. Functions
########################################################################

# Since LiveScript is functional, you'd expect functions to get a nice
# treatment. In LiveScript it's even more apparent that functions are
# first class:
add = (left, right) -> left + right
add 1, 2        # => 3

# Functions which take no arguments are called with a bang!
two = -> 2
two!

# LiveScript uses function scope, just like JavaScript, and has proper
# closures too. Unlike JavaScript, the `=` works as a declaration
# operator, and will always declare the variable on the left hand side.

# The `:=` operator is available to *reuse* a name from the parent
# scope.


# You can destructure arguments of a function to quickly get to
# interesting values inside a complex data structure:
tail = ([head, ...rest]) -> rest
tail [1, 2, 3]  # => [2, 3]

# You can also transform the arguments using binary or unary
# operators. Default arguments are also possible.
foo = (a = 1, b = 2) -> a + b
foo!    # => 3
```

```
# You could use it to clone a particular argument to avoid side-effects,
# for example:
copy = (^^target, source) ->
  for k,v of source => target[k] = v
  target
a = { a: 1 }
copy a, { b: 2 }        # => { a: 1, b: 2 }
a                       # => { a: 1 }


# A function may be curried by using a long arrow rather than a short
# one:
add = (left, right) --> left + right
add1 = add 1
add1 2          # => 3

# Functions get an implicit `it` argument, even if you don't declare
# any.
identity = -> it
identity 1      # => 1

# Operators are not functions in LiveScript, but you can easily turn
# them into one! Enter the operator sectioning:
divide-by-two = (/ 2)
[2, 4, 8, 16].map(divide-by-two) .reduce (+)


# Not only of function application lives LiveScript, as in any good
# functional language you get facilities for composing them:
double-minus-one = (- 1) . (* 2)

# Other than the usual `f . g` mathematical formulae, you get the `>>`
# and `<<` operators, that describe how the flow of values through the
# functions.
double-minus-one = (* 2) >> (- 1)
double-minus-one = (- 1) << (* 2)


# And talking about flow of value, LiveScript gets the `|>` and `<|`
# operators that apply a value to a function:
map = (f, xs) --> xs.map f
[1 2 3] |> map (* 2)            # => [2 4 6]

# You can also choose where you want the value to be placed, just mark
# the place with an underscore (_):
reduce = (f, xs, initial) --> xs.reduce f, initial
[1 2 3] |> reduce (+), _, 0     # => 6


# The underscore is also used in regular partial application, which you
# can use for any function:
div = (left, right) -> left / right
div-by-two = div _, 2
div-by-two 4      # => 2
```

```
# Last, but not least, LiveScript has back-calls, which might help
# with some callback-based code (though you should try more functional
# approaches, like Promises):
readFile = (name, f) -> f name
a <- readFile 'foo'
b <- readFile 'bar'
console.log a + b

# Same as:
readFile 'foo', (a) -> readFile 'bar', (b) -> console.log a + b


########################################################################
## 4. Patterns, guards and control-flow
########################################################################

# You can branch computations with the `if...else` expression:
x = if n > 0 then \positive else \negative

# Instead of `then`, you can use `=>`
x = if n > 0 => \positive
    else        \negative

# Complex conditions are better-off expressed with the `switch`
# expression, though:
y = {}
x = switch
  | (typeof y) is \number => \number
  | (typeof y) is \string => \string
  | 'length' of y         => \array
  | otherwise             => \object      # `otherwise` and `_` always matches.

# Function bodies, declarations and assignments get a free `switch`, so
# you don't need to type it again:
take = (n, [x, ...xs]) -->
                        | n == 0 => []
                        | _      => [x] ++ take (n - 1), xs


########################################################################
## 5. Comprehensions
########################################################################

# While the functional helpers for dealing with lists and objects are
# right there in the JavaScript's standard library (and complemented on
# the prelude-ls, which is a "standard library" for LiveScript),
# comprehensions will usually allow you to do this stuff faster and with
# a nice syntax:
oneToTwenty = [1 to 20]
evens       = [x for x in oneToTwenty when x % 2 == 0]

# `when` and `unless` can be used as filters in the comprehension.
# Object comprehension works in the same way, except that it gives you
# back an object rather than an Array:
copy = { [k, v] for k, v of source }


########################################################################
## 4. OOP
########################################################################

# While LiveScript is a functional language in most aspects, it also has
# some niceties for imperative and object oriented programming. One of
# them is class syntax and some class sugar inherited from CoffeeScript:
class Animal
  (@name, kind) ->
    @kind = kind
  action: (what) -> "*#{@name} (a #{@kind}) #{what}*"

class Cat extends Animal
  (@name) -> super @name, 'cat'
  purr: -> @action 'purrs'

kitten = new Cat 'Mei'
kitten.purr!      # => "*Mei (a cat) purrs*"

# Besides the classical single-inheritance pattern, you can also provide
# as many mixins as you would like for a class. Mixins are just plain
# objects:
Huggable =
  hug: -> @action 'is hugged'

class SnugglyCat extends Cat implements Huggable

kitten = new SnugglyCat 'Purr'
kitten.hug!     # => "*Mei (a cat) is hugged*"
```

Further reading

There's just so much more to LiveScript, but this should be enough to get you started writing little functional things in it. The official website has a lot of information on the language, and a nice online compiler for you to try stuff out!

You may also want to grab yourself some prelude.ls, and check out the `#livescript` channel on the Freenode network.

Lua

```
-- Two dashes start a one-line comment.

--[[
     Adding two ['s and ]'s makes it a
     multi-line comment.
--]]
--------------------------------------------------------------------------------
-- 1. Variables and flow control.
--------------------------------------------------------------------------------

num = 42  -- All numbers are doubles.
-- Don't freak out, 64-bit doubles have 52 bits for storing exact int
-- values; machine precision is not a problem for ints that need < 52 bits.

s = 'walternate'  -- Immutable strings like Python.
t = "double-quotes are also fine"
u = [[ Double brackets
       start and end
       multi-line strings.]]
t = nil  -- Undefines t; Lua has garbage collection.

-- Blocks are denoted with keywords like do/end:
while num < 50 do
  num = num + 1  -- No ++ or += type operators.
end

-- If clauses:
if num > 40 then
  print('over 40')
elseif s ~= 'walternate' then  -- ~= is not equals.
  -- Equality check is == like Python; ok for strs.
  io.write('not over 40\n')  -- Defaults to stdout.
else
  -- Variables are global by default.
  thisIsGlobal = 5  -- Camel case is common.

  -- How to make a variable local:
  local line = io.read()  -- Reads next stdin line.

  -- String concatenation uses the .. operator:
  print('Winter is coming, ' .. line)
end

-- Undefined variables return nil.
-- This is not an error:
foo = anUnknownVariable  -- Now foo = nil.

aBoolValue = false

-- Only nil and false are falsy; 0 and '' are true!
if not aBoolValue then print('twas false') end

-- 'or' and 'and' are short-circuited. This is similar to the a?b:c operator
-- in C/js:
ans = aBoolValue and 'yes' or 'no'  --> 'no'

karlSum = 0
for i = 1, 100 do  -- The range includes both ends.
  karlSum = karlSum + i
```

```
end

-- Use "100, 1, -1" as the range to count down:
fredSum = 0
for j = 100, 1, -1 do fredSum = fredSum + j end

-- In general, the range is begin, end[, step].

-- Another loop construct:
repeat
  print('the way of the future')
  num = num - 1
until num == 0

----------------------------------------------------------------------------------
-- 2. Functions.
----------------------------------------------------------------------------------

function fib(n)
  if n < 2 then return n end
  return fib(n - 2) + fib(n - 1)
end

-- Closures and anonymous functions are ok:
function adder(x)
  -- The returned function is created when adder is called, and remembers the
  -- value of x:
  return function (y) return x + y end
end
a1 = adder(9)
a2 = adder(36)
print(a1(16))  --> 25
print(a2(64))  --> 100

-- Returns, func calls, and assignments all work with lists that may be
-- mismatched in length. Unmatched receivers are nil; unmatched senders are
-- discarded.

x, y, z = 1, 2, 3, 4
-- Now x = 1, y = 2, z = 3, and 4 is thrown away.

function bar(a, b, c)
  print(a, b, c)
  return 4, 8, 15, 16, 23, 42
end

x, y = bar('zaphod')  --> prints "zaphod  nil nil"
-- Now x = 4, y = 8, values 15..42 are discarded.

-- Functions are first-class, may be local/global. These are the same:
function f(x) return x * x end
f = function (x) return x * x end

-- And so are these:
```

```lua
local function g(x) return math.sin(x) end
local g = function(x) return math.sin(x) end
-- Equivalent to local function g(x)..., except referring to g in the function
-- body won't work as expected.
local g; g  = function (x) return math.sin(x) end
-- the 'local g' decl makes g-self-references ok.

-- Trig funcs work in radians, by the way.

-- Calls with one string param don't need parens:
print 'hello'  -- Works fine.

-- Calls with one table param don't need parens either (more on tables below):
print {} -- Works fine too.

--------------------------------------------------------------------------------
-- 3. Tables.
--------------------------------------------------------------------------------

-- Tables = Lua's only compound data structure; they are associative arrays.
-- Similar to php arrays or js objects, they are hash-lookup dicts that can
-- also be used as lists.

-- Using tables as dictionaries / maps:

-- Dict literals have string keys by default:
t = {key1 = 'value1', key2 = false}

-- String keys can use js-like dot notation:
print(t.key1)  -- Prints 'value1'.
t.newKey = {}  -- Adds a new key/value pair.
t.key2 = nil   -- Removes key2 from the table.

-- Literal notation for any (non-nil) value as key:
u = {['@!#'] = 'qbert', [{}] = 1729, [6.28] = 'tau'}
print(u[6.28])  -- prints "tau"

-- Key matching is basically by value for numbers and strings, but by identity
-- for tables.
a = u['@!#']  -- Now a = 'qbert'.
b = u[{}]     -- We might expect 1729, but it's nil:
-- b = nil since the lookup fails. It fails because the key we used is not the
-- same object as the one used to store the original value. So strings &
-- numbers are more portable keys.

-- A one-table-param function call needs no parens:
function h(x) print(x.key1) end
h{key1 = 'Sonmi~451'}  -- Prints 'Sonmi~451'.

for key, val in pairs(u) do  -- Table iteration.
  print(key, val)
end

-- _G is a special table of all globals.
```

```
print(_G['_G'] == _G)  -- Prints 'true'.

-- Using tables as lists / arrays:

-- List literals implicitly set up int keys:
v = {'value1', 'value2', 1.21, 'gigawatts'}
for i = 1, #v do  -- #v is the size of v for lists.
  print(v[i])  -- Indices start at 1 !! SO CRAZY!
end
-- A 'list' is not a real type. v is just a table with consecutive integer
-- keys, treated as a list.

--------------------------------------------------------------------------------
-- 3.1 Metatables and metamethods.
--------------------------------------------------------------------------------

-- A table can have a metatable that gives the table operator-overloadish
-- behaviour. Later we'll see how metatables support js-prototypey behaviour.

f1 = {a = 1, b = 2}  -- Represents the fraction a/b.
f2 = {a = 2, b = 3}

-- This would fail:
-- s = f1 + f2

metafraction = {}
function metafraction.__add(f1, f2)
  local sum = {}
  sum.b = f1.b * f2.b
  sum.a = f1.a * f2.b + f2.a * f1.b
  return sum
end

setmetatable(f1, metafraction)
setmetatable(f2, metafraction)

s = f1 + f2  -- call __add(f1, f2) on f1's metatable

-- f1, f2 have no key for their metatable, unlike prototypes in js, so you must
-- retrieve it as in getmetatable(f1). The metatable is a normal table with
-- keys that Lua knows about, like __add.

-- But the next line fails since s has no metatable:
-- t = s + s
-- Class-like patterns given below would fix this.

-- An __index on a metatable overloads dot lookups:
defaultFavs = {animal = 'gru', food = 'donuts'}
myFavs = {food = 'pizza'}
setmetatable(myFavs, {__index = defaultFavs})
eatenBy = myFavs.animal  -- works! thanks, metatable

--------------------------------------------------------------------------------
-- Direct table lookups that fail will retry using the metatable's __index
-- value, and this recurses.

-- An __index value can also be a function(tbl, key) for more customized
-- lookups.

-- Values of __index,add, .. are called metamethods.
-- Full list. Here a is a table with the metamethod.

-- __add(a, b)                     for a + b
-- __sub(a, b)                     for a - b
-- __mul(a, b)                     for a * b
-- __div(a, b)                     for a / b
-- __mod(a, b)                     for a % b
-- __pow(a, b)                     for a ^ b
-- __unm(a)                        for -a
-- __concat(a, b)                  for a .. b
-- __len(a)                        for #a
-- __eq(a, b)                      for a == b
-- __lt(a, b)                      for a < b
-- __le(a, b)                      for a <= b
-- __index(a, b)  <fn or a table>  for a.b
-- __newindex(a, b, c)             for a.b = c
-- __call(a, ...)                  for a(...)

--------------------------------------------------------------------------------
-- 3.2 Class-like tables and inheritance.
--------------------------------------------------------------------------------

-- Classes aren't built in; there are different ways to make them using
-- tables and metatables.

-- Explanation for this example is below it.

Dog = {}                                   -- 1.

function Dog:new()                         -- 2.
  local newObj = {sound = 'woof'}          -- 3.
  self.__index = self                      -- 4.
  return setmetatable(newObj, self)        -- 5.
end

function Dog:makeSound()                   -- 6.
  print('I say ' .. self.sound)
end

mrDog = Dog:new()                          -- 7.
mrDog:makeSound()  -- 'I say woof'         -- 8.

-- 1. Dog acts like a class; it's really a table.
-- 2. "function tablename:fn(...)" is the same as
--    "function tablename.fn(self, ...)", The : just adds a first arg called
--    self. Read 7 & 8 below for how self gets its value.
-- 3. newObj will be an instance of class Dog.
-- 4. "self" is the class being instantiated. Often self = Dog, but inheritance
--    can change it. newObj gets self's functions when we set both newObj's
--    metatable and self's __index to self.
-- 5. Reminder: setmetatable returns its first arg.
-- 6. The : works as in 2, but this time we expect self to be an instance
--    instead of a class.
-- 7. Same as Dog.new(Dog), so self = Dog in new().
-- 8. Same as mrDog.makeSound(mrDog); self = mrDog.

--------------------------------------------------------------------------------

-- Inheritance example:

LoudDog = Dog:new()                           -- 1.

function LoudDog:makeSound()
  local s = self.sound .. ' '                 -- 2.
  print(s .. s .. s)
end

seymour = LoudDog:new()                       -- 3.
seymour:makeSound()  -- 'woof woof woof'      -- 4.

--------------------------------------------------------------------------------
-- 1. LoudDog gets Dog's methods and variables.
-- 2. self has a 'sound' key from new(), see 3.
-- 3. Same as "LoudDog.new(LoudDog)", and converted to "Dog.new(LoudDog)" as
--    LoudDog has no 'new' key, but does have "__index = Dog" on its metatable.
--    Result: seymour's metatable is LoudDog, and "LoudDog.__index = Dog". So
--    seymour.key will equal seymour.key, LoudDog.key, Dog.key, whichever
--    table is the first with the given key.
-- 4. The 'makeSound' key is found in LoudDog; this is the same as
--    "LoudDog.makeSound(seymour)".

-- If needed, a subclass's new() is like the base's:
function LoudDog:new()
  local newObj = {}
  -- set up newObj
  self.__index = self
  return setmetatable(newObj, self)
end

--------------------------------------------------------------------------------
-- 4. Modules.
--------------------------------------------------------------------------------


--[[ I'm commenting out this section so the rest of this script remains
--   runnable.
-- Suppose the file mod.lua looks like this:
local M = {}

local function sayMyName()
  print('Hrunkner')
end

function M.sayHello()
  print('Why hello there')
  sayMyName()
end

return M

-- Another file can use mod.lua's functionality:
local mod = require('mod')  -- Run the file mod.lua.

-- require is the standard way to include modules.
-- require acts like:     (if not cached; see below)
local mod = (function ()
  <contents of mod.lua>
end)()
-- It's like mod.lua is a function body, so that locals inside mod.lua are
-- invisible outside it.

-- This works because mod here = M in mod.lua:
mod.sayHello()  -- Says hello to Hrunkner.

-- This is wrong; sayMyName only exists in mod.lua:
mod.sayMyName()  -- error

-- require's return values are cached so a file is run at most once, even when
-- require'd many times.

-- Suppose mod2.lua contains "print('Hi!')".
local a = require('mod2')  -- Prints Hi!
local b = require('mod2')  -- Doesn't print; a=b.

-- dofile is like require without caching:
dofile('mod2')  --> Hi!
dofile('mod2')  --> Hi! (runs again, unlike require)

-- loadfile loads a lua file but doesn't run it yet.
f = loadfile('mod2')  -- Calling f() runs mod2.lua.

-- loadstring is loadfile for strings.
g = loadstring('print(343)')  -- Returns a function.
g()  -- Prints out 343; nothing printed before now.

--]]
```

References

I was excited to learn Lua so I could make games with the Love 2D game engine. That's the why.

I started with BlackBulletIV's Lua for programmers. Next I read the official Programming in Lua book. That's the how.

It might be helpful to check out the Lua short reference on lua-users.org.

The main topics not covered are standard libraries:

- string library
- table library
- math library
- io library
- os library

By the way, the entire file is valid Lua; save it as learn.lua and run it with "lua learn.lua" !

This was first written for tylerneylon.com, and is also available as a github gist. Have fun with Lua!

Make

A Makefile defines a graph of rules for creating a target (or targets). Its purpose is to do the minimum amount of work needed to update a target to the most recent version of the source. Famously written over a weekend by Stuart Feldman in 1976, it is still widely used (particularly on Unix) despite many competitors and criticisms.

There are many varieties of make in existance, this article assumes that we are using GNU make which is the standard on Linux.

```
# Comments can be written like this.

# Files should be named Makefile and then be can run as `make <target>`.
# Otherwise we use `make -f "filename" <target>`.

# Warning - only use TABS to indent in Makefiles, never spaces!

#-----------------------------------------------------------------------
# Basics
#-----------------------------------------------------------------------

# A rule - this rule will only run if file0.txt doesn't exist.
file0.txt:
    echo "foo" > file0.txt
    # Even comments in these 'recipe' sections get passed to the shell.
    # Try `make file0.txt` or simply `make` - first rule is the default.


# This rule will only run if file0.txt is newer than file1.txt.
file1.txt: file0.txt
    cat file0.txt > file1.txt
    # use the same quoting rules as in the shell.
    @cat file0.txt >> file1.txt
    # @ stops the command from being echoed to stdout.
    -@echo 'hello'
    # - means that make will keep going in the case of an error.
    # Try `make file1.txt` on the commandline.

# A rule can have multiple targets and multiple prerequisites
file2.txt file3.txt: file0.txt file1.txt
    touch file2.txt
    touch file3.txt
```

```
# Make will complain about multiple recipes for the same rule. Empty
# recipes don't count though and can be used to add new dependencies.

#-----------------------------------------------------------------------
# Phony Targets
#-----------------------------------------------------------------------

# A phony target. Any target that isn't a file.
# It will never be up to date so make will always try to run it.
all: maker process

# We can declare things out of order.
maker:
    touch ex0.txt ex1.txt

# Can avoid phony rules breaking when a real file has the same name by
.PHONY: all maker process
# This is a special target. There are several others.

# A rule with a dependency on a phony target will always run
ex0.txt ex1.txt: maker

# Common phony targets are: all make clean install ...

#-----------------------------------------------------------------------
# Automatic Variables & Wildcards
#-----------------------------------------------------------------------

process: file*.txt  #using a wildcard to match filenames
    @echo $^    # $^ is a variable containing the list of prerequisites
    @echo $@    # prints the target name
    #(for multiple target rules, $@ is whichever caused the rule to run)
    @echo $<    # the first prerequisite listed
    @echo $?    # only the dependencies that are out of date
    @echo $+    # all dependencies including duplicates (unlike normal)
    #@echo $|   # all of the 'order only' prerequisites

# Even if we split up the rule dependency definitions, $^ will find them
process: ex1.txt file0.txt
# ex1.txt will be found but file0.txt will be deduplicated.

#-----------------------------------------------------------------------
# Patterns
#-----------------------------------------------------------------------

# Can teach make how to convert certain files into other files.

%.png: %.svg
    inkscape --export-png $^

# Pattern rules will only do anything if make decides to create the \
target.
```

```
# Directory paths are normally ignored when matching pattern rules. But
# make will try to use the most appropriate rule available.
small/%.png: %.svg
    inkscape --export-png --export-dpi 30 $^

# make will use the last version for a pattern rule that it finds.
%.png: %.svg
    @echo this rule is chosen

# However make will use the first pattern rule that can make the target
%.png: %.ps
    @echo this rule is not chosen if *.svg and *.ps are both present

# make already has some pattern rules built-in. For instance, it knows
# how to turn *.c files into *.o files.

# Older makefiles might use suffix rules instead of pattern rules
.png.ps:
    @echo this rule is similar to a pattern rule.

# Tell make about the suffix rule
.SUFFIXES: .png

#-----------------------------------------------------------------------
# Variables
#-----------------------------------------------------------------------
# aka. macros

# Variables are basically all string types

name = Ted
name2="Sarah"

echo:
    @echo $(name)
    @echo ${name2}
    @echo $name    # This won't work, treated as $(n)ame.
    @echo $(name3) # Unknown variables are treated as empty strings.

# There are 4 places to set variables.
# In order of priority from highest to lowest:
# 1: commandline arguments
# 2: Makefile
# 3: shell enviroment variables - make imports these automatically.
# 4: make has some predefined variables

name4 ?= Jean
# Only set the variable if enviroment variable is not already defined.

override name5 = David
# Stops commandline arguments from changing this variable.

name4 +=grey
# Append values to variable (includes a space).
# Pattern-specific variable values (GNU extension).
echo: name2 = Sara # True within the matching rule
    # and also within its remade recursive dependencies
    # (except it can break when your graph gets too complicated!)

# Some variables defined automatically by make.
echo_inbuilt:
    echo $(CC)
    echo ${CXX)}
    echo $(FC)
    echo ${CFLAGS)}
    echo $(CPPFLAGS)
    echo ${CXXFLAGS}
    echo $(LDFLAGS)
    echo ${LDLIBS}

#-----------------------------------------------------------------------
# Variables 2
#-----------------------------------------------------------------------

# The first type of variables are evaluated each time they are used.
# This can be expensive, so a second type of variable exists which is
# only evaluated once. (This is a GNU make extension)

var := hello
var2 ::=  $(var) hello
#:= and ::= are equivalent.

# These variables are evaluated procedurely (in the order that they
# appear), thus breaking with the rest of the language !

# This doesn't work
var3 ::= $(var4) and good luck
var4 ::= good night

#-----------------------------------------------------------------------
# Functions
#-----------------------------------------------------------------------

# make has lots of functions available.

sourcefiles = $(wildcard *.c */*.c)
objectfiles = $(patsubst %.c,%.o,$(sourcefiles))

# Format is $(func arg0,arg1,arg2...)

# Some examples
ls: * src/*
    @echo $(filter %.txt, $^)
    @echo $(notdir $^)
    @echo $(join $(dir $^),$(notdir $^))

#-----------------------------------------------------------------------
```

```
# Directives
#-----------------------------------------------------------------------

# Include other makefiles, useful for platform specific code
include foo.mk

sport = tennis
# Conditional compilation
report:
ifeq ($(sport),tennis)
    @echo 'game, set, match'
else
    @echo "They think it's all over; it is now"
endif

# There are also ifneq, ifdef, ifndef

foo = true

ifdef $(foo)
bar = 'hello'
endif
```

More Resources

- gnu make documentation
- software carpentry tutorial
- learn C the hard way ex2 ex28

Matlab

MATLAB stands for MATrix LABoratory. It is a powerful numerical computing language commonly used in engineering and mathematics.

If you have any feedback please feel free to reach me at [@the_ozzinator](https://twitter.com/the_ozzinator), or osvaldo.t.mendoza@gmail.com.

```
%% Code sections start with two percent signs. Section titles go on the same line.
% Comments start with a percent sign.

%{
Multi line comments look
something
like
this
%}

% commands can span multiple lines, using '...':
 a = 1 + 2 + ...
 + 4

% commands can be passed to the operating system
!ping google.com
```

```
who % Displays all variables in memory
whos % Displays all variables in memory, with their types
clear % Erases all your variables from memory
clear('A') % Erases a particular variable
openvar('A') % Open variable in variable editor

clc % Erases the writing on your Command Window
diary % Toggle writing Command Window text to file
ctrl-c % Abort current computation

edit('myfunction.m') % Open function/script in editor
type('myfunction.m') % Print the source of function/script to Command Window

profile on  % turns on the code profiler
profile off     % turns off the code profiler
profile viewer  % Open profiler

help command    % Displays documentation for command in Command Window
doc command     % Displays documentation for command in Help Window
lookfor command % Searches for command in the first commented line of all functions
lookfor command -all % searches for command in all functions


% Output formatting
format short    % 4 decimals in a floating number
format long     % 15 decimals
format bank     % only two digits after decimal point - for financial calculations
fprintf('text') % print "text" to the screen
disp('text')    % print "text" to the screen

% Variables & Expressions
myVariable = 4  % Notice Workspace pane shows newly created variable
myVariable = 4; % Semi colon suppresses output to the Command Window
4 + 6       % ans = 10
8 * myVariable  % ans = 32
2 ^ 3       % ans = 8
a = 2; b = 3;
c = exp(a)*sin(pi/2) % c = 7.3891

% Calling functions can be done in either of two ways:
% Standard function syntax:
load('myFile.mat', 'y') % arguments within parentheses, separated by commas
% Command syntax:
load myFile.mat y   % no parentheses, and spaces instead of commas
% Note the lack of quote marks in command form: inputs are always passed as
% literal text - cannot pass variable values. Also, can't receive output:
[V,D] = eig(A);  % this has no equivalent in command form
[~,D] = eig(A);  % if you only want D and not V



% Logicals
1 > 5 % ans = 0
```

```
10 >= 10 % ans = 1
3 ~= 4 % Not equal to -> ans = 1
3 == 3 % equal to -> ans = 1
3 > 1 && 4 > 1 % AND -> ans = 1
3 > 1 || 4 > 1 % OR -> ans = 1
~1 % NOT -> ans = 0

% Logicals can be applied to matrices:
A > 5
% for each element, if condition is true, that element is 1 in returned matrix
A( A > 5 )
% returns a vector containing the elements in A for which condition is true

% Strings
a = 'MyString'
length(a) % ans = 8
a(2) % ans = y
[a,a] % ans = MyStringMyString


% Cells
a = {'one', 'two', 'three'}
a(1) % ans = 'one' - returns a cell
char(a(1)) % ans = one - returns a string

% Structures
A.b = {'one','two'};
A.c = [1 2];
A.d.e = false;

% Vectors
x = [4 32 53 7 1]
x(2) % ans = 32, indices in Matlab start 1, not 0
x(2:3) % ans = 32 53
x(2:end) % ans = 32 53 7 1

x = [4; 32; 53; 7; 1] % Column vector

x = [1:10] % x = 1 2 3 4 5 6 7 8 9 10
x = [1:2:10] % Increment by 2, i.e. x = 1 3 5 7 9

% Matrices
A = [1 2 3; 4 5 6; 7 8 9]
% Rows are separated by a semicolon; elements are separated with space or comma
% A =

%     1     2     3
%     4     5     6
%     7     8     9

A(2,3) % ans = 6, A(row, column)
A(6) % ans = 8
% (implicitly concatenates columns into vector, then indexes into that)
```

```
A(2,3) = 42 % Update row 2 col 3 with 42
% A =

%     1     2     3
%     4     5     42
%     7     8     9

A(2:3,2:3) % Creates a new matrix from the old one
%ans =

%     5     42
%     8     9

A(:,1) % All rows in column 1
%ans =

%     1
%     4
%     7

A(1,:) % All columns in row 1
%ans =

%     1     2     3

[A ; A] % Concatenation of matrices (vertically)
%ans =

%     1     2     3
%     4     5    42
%     7     8     9
%     1     2     3
%     4     5    42
%     7     8     9

% this is the same as
vertcat(A,A);


[A , A] % Concatenation of matrices (horizontally)

%ans =

%     1     2     3     1     2     3
%     4     5    42     4     5    42
%     7     8     9     7     8     9

% this is the same as
horzcat(A,A);


A(:, [3 1 2]) % Rearrange the columns of original matrix
%ans =
%     3     1     2
%    42     4     5
%     9     7     8

size(A) % ans = 3 3

A(1, :) =[] % Delete the first row of the matrix
A(:, 1) =[] % Delete the first column of the matrix

transpose(A) % Transpose the matrix, which is the same as:
A one
ctranspose(A) % Hermitian transpose the matrix
% (the transpose, followed by taking complex conjugate of each element)
A' % Concise version of complex transpose
A.' % Concise version of transpose (without taking complex conjugate)




% Element by Element Arithmetic vs. Matrix Arithmetic
% On their own, the arithmetic operators act on whole matrices. When preceded
% by a period, they act on each element instead. For example:
A * B % Matrix multiplication
A .* B % Multiple each element in A by its corresponding element in B

% There are several pairs of functions, where one acts on each element, and
% the other (whose name ends in m) acts on the whole matrix.
exp(A) % exponentiate each element
expm(A) % calculate the matrix exponential
sqrt(A) % take the square root of each element
sqrtm(A) %  find the matrix whose square is A


% Plotting
x = 0:.10:2*pi; % Creates a vector that starts at 0 and ends at 2*pi with increments of .1
y = sin(x);
plot(x,y)
xlabel('x axis')
ylabel('y axis')
title('Plot of y = sin(x)')
axis([0 2*pi -1 1]) % x range from 0 to 2*pi, y range from -1 to 1

plot(x,y1,'-',x,y2,'--',x,y3,':') % For multiple functions on one plot
legend('Line 1 label', 'Line 2 label') % Label curves with a legend

% Alternative method to plot multiple functions in one plot.
% while 'hold' is on, commands add to existing graph rather than replacing it
plot(x, y)
hold on
plot(x, z)
hold off

loglog(x, y) % A log-log plot
```

```
semilogx(x, y) % A plot with logarithmic x-axis
semilogy(x, y) % A plot with logarithmic y-axis

fplot (@(x) x^2, [2,5]) % plot the function x^2 from x=2 to x=5

grid on % Show grid; turn off with 'grid off'
axis square % Makes the current axes region square
axis equal % Set aspect ratio so data units are the same in every direction

scatter(x, y); % Scatter-plot
hist(x); % Histogram
stem(x); % Plot values as stems, useful for displaying discrete data
bar(x); % Plot bar graph

z = sin(x);
plot3(x,y,z); % 3D line plot

pcolor(A) % Heat-map of matrix: plot as grid of rectangles, coloured by value
contour(A) % Contour plot of matrix
mesh(A) % Plot as a mesh surface

h = figure % Create new figure object, with handle h
figure(h) % Makes the figure corresponding to handle h the current figure
close(h) % close figure with handle h
close all % close all open figure windows
close % close current figure window

shg % bring an existing graphics window forward, or create new one if needed
clf clear % clear current figure window, and reset most figure properties

% Properties can be set and changed through a figure handle.
% You can save a handle to a figure when you create it.
% The function get returns a handle to the current figure
h = plot(x, y); % you can save a handle to a figure when you create it
set(h, 'Color', 'r')
% 'y' yellow; 'm' magenta, 'c' cyan, 'r' red, 'g' green, 'b' blue, 'w' white, 'k' black
set(h, 'LineStyle', '--')
 % '--' is solid line, '---' dashed, ':' dotted, '-.' dash-dot, 'none' is no line
get(h, 'LineStyle')


% The function gca returns a handle to the axes for the current figure
set(gca, 'XDir', 'reverse'); % reverse the direction of the x-axis

% To create a figure that contains several axes in tiled positions, use subplot
subplot(2,3,1); % select the first position in a 2-by-3 grid of subplots
plot(x1); title('First Plot') % plot something in this position
subplot(2,3,2); % select second position in the grid
plot(x2); title('Second Plot') % plot something there


% To use functions or scripts, they must be on your path or current directory
path % display current path
addpath /path/to/dir % add to path
```

```
rmpath /path/to/dir % remove from path
cd /path/to/move/into % change directory


% Variables can be saved to .mat files
save('myFileName.mat') % Save the variables in your Workspace
load('myFileName.mat') % Load saved variables into Workspace

% M-file Scripts
% A script file is an external file that contains a sequence of statements.
% They let you avoid repeatedly typing the same code in the Command Window
% Have .m extensions

% M-file Functions
% Like scripts, and have the same .m extension
% But can accept input arguments and return an output
% Also, they have their own workspace (ie. different variable scope).
% Function name should match file name (so save this example as double_input.m).
% 'help double_input.m' returns the comments under line beginning function
function output = double_input(x)
    %double_input(x) returns twice the value of x
    output = 2*x;
end
double_input(6) % ans = 12


% You can also have subfunctions and nested functions.
% Subfunctions are in the same file as the primary function, and can only be
% called by functions in the file. Nested functions are defined within another
% functions, and have access to both its workspace and their own workspace.

% If you want to create a function without creating a new file you can use an
% anonymous function. Useful when quickly defining a function to pass to
% another function (eg. plot with fplot, evaluate an indefinite integral
% with quad, find roots with fzero, or find minimum with fminsearch).
% Example that returns the square of it's input, assigned to the handle sqr:
sqr = @(x) x.^2;
sqr(10) % ans = 100
doc function_handle % find out more

% User input
a = input('Enter the value: ')

% Stops execution of file and gives control to the keyboard: user can examine
% or change variables. Type 'return' to continue execution, or 'dbquit' to exit
keyboard

% Reading in data (also xlsread/importdata/imread for excel/CSV/image files)
fopen(filename)

% Output
disp(a) % Print out the value of variable a
disp('Hello World') % Print out a string
fprintf % Print to Command Window with more control
```

```
% Conditional statements (the parentheses are optional, but good style)
if (a > 15)
    disp('Greater than 15')
elseif (a == 23)
    disp('a is 23')
else
    disp('neither condition met')
end

% Looping
% NB. looping over elements of a vector/matrix is slow!
% Where possible, use functions that act on whole vector/matrix at once
for k = 1:5
    disp(k)
end

k = 0;
while (k < 5)
    k = k + 1;
end

% Timing code execution: 'toc' prints the time since 'tic' was called
tic
A = rand(1000);
A*A*A*A*A*A*A;
toc

% Connecting to a MySQL Database
dbname = 'database_name';
username = 'root';
password = 'root';
driver = 'com.mysql.jdbc.Driver';
dburl = ['jdbc:mysql://localhost:8889/' dbname];
javaclasspath('mysql-connector-java-5.1.xx-bin.jar'); %xx depends on version, download available at http
conn = database(dbname, username, password, driver, dburl);
sql = ['SELECT * from table_name where id = 22'] % Example sql statement
a = fetch(conn, sql) %a will contain your data


% Common math functions
sin(x)
cos(x)
tan(x)
asin(x)
acos(x)
atan(x)
exp(x)
sqrt(x)
log(x)
log10(x)
abs(x) %If x is complex, returns magnitude
min(x)
max(x)
```

```
ceil(x)
floor(x)
round(x)
rem(x)
rand % Uniformly distributed pseudorandom numbers
randi % Uniformly distributed pseudorandom integers
randn % Normally distributed pseudorandom numbers

%Complex math operations
abs(x)   % Magnitude of complex variable x
phase(x) % Phase (or angle) of complex variable x
real(x)  % Returns the real part of x (i.e returns a if x = a +jb)
imag(x)  % Returns the imaginary part of x (i.e returns b if x = a+jb)
conj(x)  % Returns the complex conjugate


% Common constants
pi
NaN
inf

% Solving matrix equations (if no solution, returns a least squares solution)
% The \ and / operators are equivalent to the functions mldivide and mrdivide
x=A\b % Solves Ax=b. Faster and more numerically accurate than using inv(A)*b.
x=b/A % Solves xA=b

inv(A) % calculate the inverse matrix
pinv(A) % calculate the pseudo-inverse

% Common matrix functions
zeros(m,n) % m x n matrix of 0's
ones(m,n) % m x n matrix of 1's
diag(A) % Extracts the diagonal elements of a matrix A
diag(x) % Construct a matrix with diagonal elements listed in x, and zeroes elsewhere
eye(m,n) % Identity matrix
linspace(x1, x2, n) % Return n equally spaced points, with min x1 and max x2
inv(A) % Inverse of matrix A
det(A) % Determinant of A
eig(A) % Eigenvalues and eigenvectors of A
trace(A) % Trace of matrix - equivalent to sum(diag(A))
isempty(A) % Tests if array is empty
all(A) % Tests if all elements are nonzero or true
any(A) % Tests if any elements are nonzero or true
isequal(A, B) % Tests equality of two arrays
numel(A) % Number of elements in matrix
triu(x) % Returns the upper triangular part of x
tril(x) % Returns the lower triangular part of x
cross(A,B) %  Returns the cross product of the vectors A and B
dot(A,B) % Returns scalar product of two vectors (must have the same length)
transpose(A) % Returns the transpose of A
fliplr(A) % Flip matrix left to right
flipud(A) % Flip matrix up to down

% Matrix Factorisations
```

```
[L, U, P] = lu(A) % LU decomposition: PA = LU,L is lower triangular, U is upper triangular, P is permut
[P, D] = eig(A) % eigen-decomposition: AP = PD, P's columns are eigenvectors and D's diagonals are eige
[U,S,V] = svd(X) % SVD: XV = US, U and V are unitary matrices, S has non-negative diagonal elements in

% Common vector functions
max     % largest component
min     % smallest component
length  % length of a vector
sort    % sort in ascending order
sum     % sum of elements
prod    % product of elements
mode    % modal value
median  % median value
mean    % mean value
std     % standard deviation
perms(x) % list all permutations of elements of x
find(x) % Finds all non-zero elements of x and returns their indexes, can use comparison operators,
        % i.e. find( x == 3 ) returns indexes of elements that are equal to 3
        % i.e. find( x >= 3 ) returns indexes of elements greater than or equal to 3


% Classes
% Matlab can support object-oriented programming.
% Classes must be put in a file of the class name with a .m extension.
% To begin, we create a simple class to store GPS waypoints.
% Begin WaypointClass.m
classdef WaypointClass % The class name.
  properties % The properties of the class behave like Structures
    latitude
    longitude
  end
  methods
    % This method that has the same name of the class is the constructor.
    function obj = WaypointClass(lat, lon)
      obj.latitude = lat;
      obj.longitude = lon;
    end

    % Other functions that use the Waypoint object
    function r = multiplyLatBy(obj, n)
      r = n*[obj.latitude];
    end

    % If we want to add two Waypoint objects together without calling
    % a special function we can overload Matlab's arithmetic like so:
    function r = plus(o1,o2)
      r = WaypointClass([o1.latitude] +[o2.latitude], ...
                        [o1.longitude]+[o2.longitude]);
    end
  end
end
% End WaypointClass.m

% We can create an object of the class using the constructor
```

```
a = WaypointClass(45.0, 45.0)

% Class properties behave exactly like Matlab Structures.
a.latitude = 70.0
a.longitude = 25.0

% Methods can be called in the same way as functions
ans = multiplyLatBy(a,3)

% The method can also be called using dot notation. In this case, the object
% does not need to be passed to the method.
ans = a.multiplyLatBy(a,1/3)

% Matlab functions can be overloaded to handle objects.
% In the method above, we have overloaded how Matlab handles
% the addition of two Waypoint objects.
b = WaypointClass(15.0, 32.0)
c = a + b
```

More on Matlab

- The official website http://http://www.mathworks.com/products/matlab/
- The official MATLAB Answers forum: http://www.mathworks.com/matlabcentral/answers/

Neat

Neat is basically a smaller version of D1 with some experimental syntax and a focus on terseness without losing the basic C-like syntax.

Read more here.

```
// single line comments start with //
/*
  multiline comments look like this
*/
/+
  or this
  /+ these can be nested too, same as D +/
+/

// Module name. This has to match the filename/directory.
module LearnNeat;

// Make names from another module visible in this one.
import std.file;
// You can import multiple things at once.
import std.math, std.util;
// You can even group up imports!
import std.(process, socket);

// Global functions!
void foo() { }
```

```
// Main function, same as in C.
// string[] == "array of strings".
// "string" is just an alias for char[],
void main(string[] args) {
  // Call functions with "function expression".
  writeln "Hello World";
  // You can do it like in C too... if you really want.
  writeln ("Hello World");
  // Declare a variable with "type identifier"
  string arg = ("Hello World");
  writeln arg;
  // (expression, expression) forms a tuple.
  // There are no one-value tuples though.
  // So you can always use () in the mathematical sense.
  // (string) arg; <- is an error

  /*
    byte: 8 bit signed integer
      char: 8 bit UTF-8 byte component.
    short: 16 bit signed integer
    int: 32 bit signed integer
    long: 64 bit signed integer

    float: 32 bit floating point
    double: 64 bit floating point
    real: biggest native size floating point (80 bit on x86).

    bool: true or false
  */
  int a = 5;
  bool b = true;
  // as in C, && and || are short-circuit evaluating.
  b = b && false;
  assert(b == false);
  // "" are "format strings". So $variable will be substituted at runtime
  // with a formatted version of the variable.
  writeln "$a";
  // This will just print $a.
  writeln `$a`;
  // you can format expressions with $()
  writeln "$(2+2)";
  // Note: there is no special syntax for characters.
  char c = "a";
  // Cast values by using type: expression.
  // There are three kinds of casts:
  // casts that just specify conversions that would be happening automatically
  // (implicit casts)
  float f = float:5;
  float f2 = 5; // would also work
  // casts that require throwing away information or complicated computation -
  // those must always be done explicitly
  // (conversion casts)
  int i = int:f;
  // int i = f; // would not work!
  // and, as a last attempt, casts that just reinterpret the raw data.
  // Those only work if the types have the same size.
  string s = "Hello World";
  // Arrays are (length, pointer) pairs.
  // This is a tuple type. Tuple types are (type, type, type).
  // The type of a tuple expression is a tuple type. (duh)
  (int, char*) array = (int, char*): s;
  // You can index arrays and tuples using the expression[index] syntax.
  writeln "pointer is $(array[1]) and length is $(array[0])";
  // You can slice them using the expression[from .. to] syntax.
  // Slicing an array makes another array.
  writeln "$(s[0..5]) World";
  // Alias name = expression gives the expression a name.
  // As opposed to a variable, aliases do not have an address
  // and can not be assigned to. (Unless the expression is assignable)
  alias range = 0 .. 5;
  writeln "$(s[range]) World";
  // You can iterate over ranges.
  for int i <- range {
    write "$(s[i])";
  }
  writeln " World";
  // Note that if "range" had been a variable, it would be 'empty' now!
  // Range variables can only be iterated once.
  // The syntax for iteration is "expression <- iterable".
  // Lots of things are iterable.
  for char c <- "Hello" { write "$c"; }
  writeln " World";
  // For loops are "for test statement";
  alias test = char d <- "Hello";
  for test write "$d";
  writeln " World\t\x05"; // note: escapes work
  // Pointers: function the same as in C, btw. The usual.
  // Do note: the pointer star sticks with the TYPE, not the VARIABLE!
  string* p;
  assert(p == null); // default initializer
  p = &s;
  writeln "$(*p)";
  // Math operators are (almost) standard.
  int x = 2 + 3 * 4 << 5;
  // Note: XOR is "xor". ^ is reserved for exponentiation (once I implement that).
  int y = 3 xor 5;
  int z = 5;
  assert(z++ == 5);
  assert(++z == 7);
  writeln "x $x y $y z $z";
  // As in D, ~ concatenates.
  string hewo = "Hello " ~ "World";
  // == tests for equality, "is" tests for identity.
  assert  (hewo == s);
  assert !(hewo is s);
  // same as
  assert  (hewo !is s);
```

```
// Allocate arrays using "new array length"
int[] integers = new int[] 10;
assert(integers.length == 10);
assert(integers[0] == 0); // zero is default initializer
integers = integers ~ 5; // This allocates a new array!
assert(integers.length == 11);

// This is an appender array.
// Instead of (length, pointer), it tracks (capacity, length, pointer).
// When you append to it, it will use the free capacity if it can.
// If it runs out of space, it reallocates - but it will free the old array automatically.
// This makes it convenient for building arrays.
int[auto~] appender;
appender ~= 2;
appender ~= 3;
appender.free(); // same as {mem.free(appender.ptr); appender = null;}

// Scope variables are automatically freed at the end of the current scope.
scope int[auto~] someOtherAppender;
// This is the same as:
int[auto~] someOtherAppender2;
onExit { someOtherAppender2.free; }

// You can do a C for loop too
// - but why would you want to?
for (int i = 0; i < 5; ++i) { }
// Otherwise, for and while are the same.
while int i <- 0..4 {
  assert(i == 0);
  break; // continue works too
} then assert(false); // if we hadn't break'd, this would run at the end
// This is the height of loopdom - the produce-test-consume loop.
do {
  int i = 5;
} while (i == 5) {
  assert(i == 5);
  break; // otherwise we'd go back up to do {
}

// This is a nested function.
// Nested functions can access the surrounding function.
string returnS() { return s; }
writeln returnS();

// Take the address of a function using &
// The type of a global function is ReturnType function(ParameterTypeTuple).
void function() foop = &foo;

// Similarly, the type of a nested function is ReturnType delegate(ParameterTypeTuple).
string delegate() returnSp = &returnS;
writeln returnSp();
// Class member functions and struct member functions also fit into delegate variables.
// In general, delegates are functions that carry an additional context pointer.
// ("fat pointers" in C)
// Allocate a "snapshot" with "new delegate".
// Snapshots are not closures! I used to call them closures too,
// but then my Haskell-using friends yelled at me so I had to stop.
// The difference is that snapshots "capture" their surrounding context
// when "new" is used.
// This allows things like this
int delegate(int) add(int a) {
  int add_a(int b) { return a + b; }
  // This does not work - the context of add_a becomes invalid
  // when add returns.
  // return &add_a;
  // Instead:
  return new &add_a;
}
int delegate(int) dg = add 2;
assert (dg(3) == 5);
// or
assert (((add 2) 3) == 5);
// or
assert (add 2 3 == 5);
// add can also be written as
int delegate(int) add2(int a) {
  // this is an implicit, nameless nested function.
  return new  (int b) { return a + b; }
}
// or even
auto add3(int a) { return new  (int b) -> a + b; }
// hahahaaa
auto add4 =  (int a) -> new  (int b) -> a + b;
assert(add4 2 3 == 5);
// If your keyboard doesn't have a   (you poor sod)
// you can use \ too.
auto add5 = \(int a) -> new \(int b) -> a + b;
// Note!
auto nestfun =  () { } // There is NO semicolon needed here!
// "}" can always substitute for "};".
// This provides syntactic consistency with built-in statements.


// This is a class.
// Note: almost all elements of Neat can be used on the module level
//       or just as well inside a function.
class C {
  int a;
  void writeA() { writeln "$a"; }
  // It's a nested class - it exists in the context of main().
  // so if you leave main(), any instances of C become invalid.
  void writeS() { writeln "$s"; }
}
C cc = new C;
// cc is a *reference* to C. Classes are always references.
cc.a = 5; // Always used for property access.
auto ccp = &cc;
```

```
  (*ccp).a = 6;
  // or just
  ccp.a = 7;
  cc.writeA();
  cc.writeS(); // to prove I'm not making things up
  // Interfaces work same as in D, basically. Or Java.
  interface E { void doE(); }
  // Inheritance works same as in D, basically. Or Java.
  class D : C, E {
    override void writeA() { writeln "hahahahaha no"; }
    override void doE() { writeln "eeeee"; }
    // all classes inherit from Object. (toString is defined in Object)
    override string toString() { return "I am a D"; }
  }
  C cd = new D;
  // all methods are always virtual.
  cd.writeA();
  E e = E:cd; // dynamic class cast!
  e.doE();
  writeln "$e"; // all interfaces convert to Object implicitly.

  // Templates!
  // Templates are parameterized namespaces, taking a type as a parameter.
  template Templ(T) {
    alias hi = 5, hii = 8;
    // Templates always have to include something with the same name as the template
    // - this will become the template's _value_.
    // Static ifs are evaluated statically, at compile-time.
    // Because of this, the test has to be a constant expression,
    // or something that can be optimized to a constant.
    static if (types-equal (T, int)) {
      alias Templ = hi;
    } else {
      alias Templ = hii;
    }
  }
  assert(Templ!int == 5);
  assert(Templ!float == 8);
}
```

Topics Not Covered

- Extended iterator types and expressions
- Standard library
- Conditions (error handling)
- Macros

Nim

Nim (formerly Nimrod) is a statically typed, imperative programming language that gives the programmer power without compromises on runtime efficiency.

Nim is efficient, expressive, and elegant.

```
var                     # Declare (and assign) variables,
  letter: char = 'n'    # with or without type annotations
  lang = "N" & "im"
  nLength : int = len(lang)
  boat: float
  truth: bool = false

let            # Use let to declare and bind variables *once*.
  legs = 400   # legs is immutable.
  arms = 2_000 # _ are ignored and are useful for long numbers.
  aboutPi = 3.15

const            # Constants are computed at compile time. This provides
  debug = true   # performance and is useful in compile time expressions.
  compileBadCode = false

when compileBadCode:            # `when` is a compile time `if`
  legs = legs + 1               # This error will never be compiled.
  const input = readline(stdin) # Const values must be known at compile time.

discard 1 > 2 # Note: The compiler will complain if the result of an expression
              # is unused. `discard` bypasses this.

discard """
This can work as a multiline comment.
Or for unparsable, broken code
"""

#
# Data Structures
#

# Tuples

var
  child: tuple[name: string, age: int]   # Tuples have *both* field names
  today: tuple[sun: string, temp: float] # *and* order.

child = (name: "Rudiger", age: 2) # Assign all at once with literal ()
today.sun = "Overcast"            # or individual fields.
today.temp = 70.1

# Sequences

var
  drinks: seq[string]

drinks = @["Water", "Juice", "Chocolate"] # @[V1,..,Vn] is the sequence literal

drinks.add("Milk")

if "Milk" in drinks:
  echo "We have Milk and ", drinks.len - 1, " other drinks"

let myDrink = drinks[2]

#
# Defining Types
#

# Defining your own types puts the compiler to work for you. It's what makes
# static typing powerful and useful.

type
  Name = string # A type alias gives you a new type that is interchangable
  Age = int     # with the old type but is more descriptive.
  Person = tuple[name: Name, age: Age] # Define data structures too.
  AnotherSyntax = tuple
    fieldOne: string
    secondField: int

var
  john: Person = (name: "John B.", age: 17)
  newage: int = 18 # It would be better to use Age than int

john.age = newage # But still works because int and Age are synonyms

type
  Cash = distinct int    # `distinct` makes a new type incompatible with its
  Desc = distinct string # base type.

var
  money: Cash = 100.Cash # `.Cash` converts the int to our type
  description: Desc  = "Interesting".Desc

when compileBadCode:
  john.age  = money        # Error! age is of type int and money is Cash
  john.name = description  # Compiler says: "No way!"

#
# More Types and Data Structures
#

# Enumerations allow a type to have one of a limited number of values

type
  Color = enum cRed, cBlue, cGreen
  Direction = enum # Alternative formating
    dNorth
    dWest
    dEast
    dSouth
var
  orient = dNorth # `orient` is of type Direction, with the value `dNorth`
  pixel = cGreen # `pixel` is of type Color, with the value `cGreen`
```

```
discard dNorth > dEast # Enums are usually an "ordinal" type

# Subranges specify a limited valid range

type
  DieFaces = range[1..20] # Only an int from 1 to 20 is a valid value
var
  my_roll: DieFaces = 13

when compileBadCode:
  my_roll = 23 # Error!

# Arrays

type
  RollCounter = array[DieFaces, int]  # Array's are fixed length and
  DirNames = array[Direction, string] # indexed by any ordinal type.
  Truths = array[42..44, bool]
var
  counter: RollCounter
  directions: DirNames
  possible: Truths

possible = [false, false, false] # Literal arrays are created with [V1,..,Vn]
possible[42] = true

directions[dNorth] = "Ahh. The Great White North!"
directions[dWest] = "No, don't go there."

my_roll = 13
counter[my_roll] += 1
counter[my_roll] += 1

var anotherArray = ["Default index", "starts at", "0"]

# More data structures are available, including tables, sets, lists, queues,
# and crit bit trees.
# http://nim-lang.org/docs/lib.html#collections-and-algorithms

#
# IO and Control Flow
#

# `case`, `readLine()`

echo "Read any good books lately?"
case readLine(stdin)
of "no", "No":
  echo "Go to your local library."
of "yes", "Yes":
  echo "Carry on, then."
else:
  echo "That's great; I assume."
```

```
# `while`, `if`, `continue`, `break`

import strutils as str # http://nim-lang.org/docs/strutils.html
echo "I'm thinking of a number between 41 and 43. Guess which!"
let number: int = 42
var
  raw_guess: string
  guess: int
while guess != number:
  raw_guess = readLine(stdin)
  if raw_guess == "": continue # Skip this iteration
  guess = str.parseInt(raw_guess)
  if guess == 1001:
    echo("AAAAAAGGG!")
    break
  elif guess > number:
    echo("Nope. Too high.")
  elif guess < number:
    echo(guess, " is too low")
  else:
    echo("Yeeeeeehaw!")

#
# Iteration
#

for i, elem in ["Yes", "No", "Maybe so"]: # Or just `for elem in`
  echo(elem, " is at index: ", i)

for k, v in items(@[(person: "You", power: 100), (person: "Me", power: 9000)]):
  echo v

let myString = """
an <example>
`string` to
play with
""" # Multiline raw string

for line in splitLines(myString):
  echo(line)

for i, c in myString:       # Index and letter. Or `for j in` for just letter
  if i mod 2 == 0: continue # Compact `if` form
  elif c == 'X': break
  else: echo(c)

#
# Procedures
#

type Answer = enum aYes, aNo

proc ask(question: string): Answer =
  echo(question, " (y/n)")
  while true:
    case readLine(stdin)
    of "y", "Y", "yes", "Yes":
      return Answer.aYes  # Enums can be qualified
    of "n", "N", "no", "No":
      return Answer.aNo
    else: echo("Please be clear: yes or no")

proc addSugar(amount: int = 2) = # Default amount is 2, returns nothing
  assert(amount > 0 and amount < 9000, "Crazy Sugar")
  for a in 1..amount:
    echo(a, " sugar...")

case ask("Would you like sugar in your tea?")
of aYes:
  addSugar(3)
of aNo:
  echo "Oh do take a little!"
  addSugar()
# No need for an `else` here. Only `yes` and `no` are possible.

#
# FFI
#

# Because Nim compiles to C, FFI is easy:

proc strcmp(a, b: cstring): cint {.importc: "strcmp", nodecl.}

let cmp = strcmp("C?", "Easy!")
```

Additionally, Nim separates itself from its peers with metaprogramming, performance, and compile-time features.

Further Reading

- Home Page
- Download
- Community
- FAQ
- Documentation
- Manual
- Standard Library
- Rosetta Code

Objective-C

Objective-C is the main programming language used by Apple for the OS X and iOS operating systems and their respective frameworks, Cocoa and Cocoa Touch. It is a general-purpose, object-oriented programming language that adds Smalltalk-style messaging to the C programming language.

```
// Single-line comments start with //
```

```
/*
Multi-line comments look like this
*/

// XCode supports pragma mark directive that improve jump bar readability
#pragma mark Navigation Functions // New tag on jump bar named 'Navigation Functions'
#pragma mark - Navigation Functions // Same tag, now with a separator

// Imports the Foundation headers with #import
// Use <> to import global files (in general frameworks)
// Use "" to import local files (from project)
#import <Foundation/Foundation.h>
#import "MyClass.h"

// If you enable modules for iOS >= 7.0 or OS X >= 10.9 projects in
// Xcode 5 you can import frameworks like that:
@import Foundation;

// Your program's entry point is a function called
// main with an integer return type
int main (int argc, const char * argv[])
{
    // Create an autorelease pool to manage the memory into the program
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // If using automatic reference counting (ARC), use @autoreleasepool instead:
    @autoreleasepool {

    // Use NSLog to print lines to the console
    NSLog(@"Hello World!"); // Print the string "Hello World!"

    ///////////////////////////////////////
    // Types & Variables
    ///////////////////////////////////////

    // Primitive declarations
    int myPrimitive1  = 1;
    long myPrimitive2 = 234554664565;

    // Object declarations
    // Put the * in front of the variable names for strongly-typed object declarations
    MyClass *myObject1 = nil;  // Strong typing
    id       myObject2 = nil;  // Weak typing
    // %@ is an object
    // 'description' is a convention to display the value of the Objects
    NSLog(@"%@ and %@", myObject1, [myObject2 description]); // prints => "(null) and (null)"

    // String
    NSString *worldString = @"World";
    NSLog(@"Hello %@!", worldString); // prints => "Hello World!"
    // NSMutableString is a mutable version of the NSString object
    NSMutableString *mutableString = [NSMutableString stringWithString:@"Hello"];
    [mutableString appendString:@" World!"];
    NSLog(@"%@", mutableString); // prints => "Hello World!"
```

```
// Character literals
NSNumber *theLetterZNumber = @'Z';
char theLetterZ            = [theLetterZNumber charValue]; // or 'Z'
NSLog(@"%c", theLetterZ);

// Integral literals
NSNumber *fortyTwoNumber = @42;
int fortyTwo             = [fortyTwoNumber intValue]; // or 42
NSLog(@"%i", fortyTwo);

NSNumber *fortyTwoUnsignedNumber = @42U;
unsigned int fortyTwoUnsigned    = [fortyTwoUnsignedNumber unsignedIntValue]; // or 42
NSLog(@"%u", fortyTwoUnsigned);

NSNumber *fortyTwoShortNumber = [NSNumber numberWithShort:42];
short fortyTwoShort           = [fortyTwoShortNumber shortValue]; // or 42
NSLog(@"%hi", fortyTwoShort);

NSNumber *fortyOneShortNumber   = [NSNumber numberWithShort:41];
unsigned short fortyOneUnsigned = [fortyOneShortNumber unsignedShortValue]; // or 41
NSLog(@"%u", fortyOneUnsigned);

NSNumber *fortyTwoLongNumber = @42L;
long fortyTwoLong            = [fortyTwoLongNumber longValue]; // or 42
NSLog(@"%li", fortyTwoLong);

NSNumber *fiftyThreeLongNumber   = @53L;
unsigned long fiftyThreeUnsigned = [fiftyThreeLongNumber unsignedLongValue]; // or 53
NSLog(@"%lu", fiftyThreeUnsigned);

// Floating point literals
NSNumber *piFloatNumber = @3.141592654F;
float piFloat           = [piFloatNumber floatValue]; // or 3.141592654f
NSLog(@"%f", piFloat); // prints => 3.141592654
NSLog(@"%5.2f", piFloat); // prints => " 3.14"

NSNumber *piDoubleNumber = @3.1415926535;
double piDouble          = [piDoubleNumber doubleValue]; // or 3.1415926535
NSLog(@"%f", piDouble);
NSLog(@"%4.2f", piDouble); // prints => "3.14"

// NSDecimalNumber is a fixed-point class that's more precise then float or double
NSDecimalNumber *oneDecNum = [NSDecimalNumber decimalNumberWithString:@"10.99"];
NSDecimalNumber *twoDecNum = [NSDecimalNumber decimalNumberWithString:@"5.002"];
// NSDecimalNumber isn't able to use standard +, -, *, / operators so it provides its own:
[oneDecNum decimalNumberByAdding:twoDecNum];
[oneDecNum decimalNumberBySubtracting:twoDecNum];
[oneDecNum decimalNumberByMultiplyingBy:twoDecNum];
[oneDecNum decimalNumberByDividingBy:twoDecNum];
NSLog(@"%@", oneDecNum); // prints => 10.99 as NSDecimalNumber is immutable

// BOOL literals
NSNumber *yesNumber = @YES;
NSNumber *noNumber  = @NO;
```

```
// or
BOOL yesBool = YES;
BOOL noBool  = NO;
NSLog(@"%i", yesBool); // prints => 1

// Array object
// May contain different data types, but must be an Objective-C object
NSArray *anArray      = @[@1, @2, @3, @4];
NSNumber *thirdNumber = anArray[2];
NSLog(@"Third number = %@", thirdNumber); // prints => "Third number = 3"
// NSMutableArray is a mutable version of NSArray, allowing you to change
// the items in the array and to extend or shrink the array object.
// Convenient, but not as efficient as NSArray.
NSMutableArray *mutableArray = [NSMutableArray arrayWithCapacity:2];
[mutableArray addObject:@"Hello"];
[mutableArray addObject:@"World"];
[mutableArray removeObjectAtIndex:0];
NSLog(@"%@", [mutableArray objectAtIndex:0]); // prints => "World"

// Dictionary object
NSDictionary *aDictionary = @{ @"key1" : @"value1", @"key2" : @"value2" };
NSObject *valueObject     = aDictionary[@"A Key"];
NSLog(@"Object = %@", valueObject); // prints => "Object = (null)"
// NSMutableDictionary also available as a mutable dictionary object
NSMutableDictionary *mutableDictionary = [NSMutableDictionary dictionaryWithCapacity:2];
[mutableDictionary setObject:@"value1" forKey:@"key1"];
[mutableDictionary setObject:@"value2" forKey:@"key2"];
[mutableDictionary removeObjectForKey:@"key1"];

// Change types from Mutable To Immutable
//In general [object mutableCopy] will make the object mutable whereas [object copy] will make the
NSMutableDictionary *aMutableDictionary = [aDictionary mutableCopy];
NSDictionary *mutableDictionaryChanged = [mutableDictionary copy];


// Set object
NSSet *set = [NSSet setWithObjects:@"Hello", @"Hello", @"World", nil];
NSLog(@"%@", set); // prints => {(Hello, World)} (may be in different order)
// NSMutableSet also available as a mutable set object
NSMutableSet *mutableSet = [NSMutableSet setWithCapacity:2];
[mutableSet addObject:@"Hello"];
[mutableSet addObject:@"Hello"];
NSLog(@"%@", mutableSet); // prints => {(Hello)}

///////////////////////////////////////
// Operators
///////////////////////////////////////

// The operators works like in the C language
// For example:
2 + 5; // => 7
4.2f + 5.1f; // => 9.3f
3 == 2; // => 0 (NO)
3 != 2; // => 1 (YES)
```

```
1 && 1; // => 1 (Logical and)
0 || 1; // => 1 (Logical or)
~0x0F; // => 0xF0 (bitwise negation)
0x0F & 0xF0; // => 0x00 (bitwise AND)
0x01 << 1; // => 0x02 (bitwise left shift (by 1))

///////////////////////////////////////
// Control Structures
///////////////////////////////////////

// If-Else statement
if (NO)
{
    NSLog(@"I am never run");
} else if (0)
{
    NSLog(@"I am also never run");
} else
{
    NSLog(@"I print");
}

// Switch statement
switch (2)
{
    case 0:
    {
        NSLog(@"I am never run");
    } break;
    case 1:
    {
        NSLog(@"I am also never run");
    } break;
    default:
    {
        NSLog(@"I print");
    } break;
}

// While loops statements
int ii = 0;
while (ii < 4)
{
    NSLog(@"%d,", ii++); // ii++ increments ii in-place, after using its value
} // prints => "0,"
  //           "1,"
  //           "2,"
  //           "3,"

// For loops statements
int jj;
for (jj=0; jj < 4; jj++)
{
    NSLog(@"%d,", jj);
} // prints => "0,"
  //           "1,"
  //           "2,"
  //           "3,"

// Foreach statements
NSArray *values = @[@0, @1, @2, @3];
for (NSNumber *value in values)
{
    NSLog(@"%@,", value);
} // prints => "0,"
  //           "1,"
  //           "2,"
  //           "3,"

// Object for loop statement. Can be used with any Objective-C object type
for (id item in values) {
    NSLog(@"%@,", item);
} // prints => "0,"
  //           "1,"
  //           "2,"
  //           "3,"

// Try-Catch-Finally statements
@try
{
    // Your statements here
    @throw [NSException exceptionWithName:@"FileNotFoundException"
                        reason:@"File Not Found on System" userInfo:nil];
} @catch (NSException * e) // use: @catch (id exceptionName) to catch all objects.
{
    NSLog(@"Exception: %@", e);
} @finally
{
    NSLog(@"Finally. Time to clean up.");
} // prints => "Exception: File Not Found on System"
  //           "Finally. Time to clean up."

// NSError objects are useful for function arguments to populate on user mistakes.
NSError *error = [NSError errorWithDomain:@"Invalid email." code:4 userInfo:nil];

///////////////////////////////////////
// Objects
///////////////////////////////////////

// Create an object instance by allocating memory and initializing it
// An object is not fully functional until both steps have been completed
MyClass *myObject = [[MyClass alloc] init];

// The Objective-C model of object-oriented programming is based on message
// passing to object instances
// In Objective-C one does not simply call a method; one sends a message
[myObject instanceMethodWithParameter:@"Steve Jobs"];
```

```
    // Clean up the memory you used into your program
    [pool drain];

    // End of @autoreleasepool
    }

    // End the program
    return 0;
}

///////////////////////////////////////
// Classes And Functions
///////////////////////////////////////

// Declare your class in a header file (MyClass.h):
// Class declaration syntax:
// @interface ClassName : ParentClassName <ImplementedProtocols>
// {
//    type name; <= variable declarations;
// }
// @property type name; <= property declarations
// -/+ (type) Method declarations; <= Method declarations
// @end
@interface MyClass : NSObject <MyProtocol> // NSObject is Objective-C's base object class.
{
    // Instance variable declarations (can exist in either interface or implementation file)
    int count; // Protected access by default.
    @private id data; // Private access (More convenient to declare in implementation file)
    NSString *name;
}
// Convenient notation for public access variables to auto generate a setter method
// By default, setter method name is 'set' followed by @property variable name
@property int propInt; // Setter method name = 'setPropInt'
@property (copy) id copyId; // (copy) => Copy the object during assignment
// (readonly) => Cannot set value outside @interface
@property (readonly) NSString *roString; // Use @synthesize in @implementation to create accessor
// You can customize the getter and setter names instead of using default 'set' name:
@property (getter=lengthGet, setter=lengthSet:) int length;

// Methods
+/- (return type)methodSignature:(Parameter Type *)parameterName;

// + for class methods:
+ (NSString *)classMethod;
+ (MyClass *)myClassFromHeight:(NSNumber *)defaultHeight;

// - for instance methods:
- (NSString *)instanceMethodWithParameter:(NSString *)string;
- (NSNumber *)methodAParameterAsString:(NSString*)string andAParameterAsNumber:(NSNumber *)number;

// Constructor methods with arguments:
- (id)initWithDistance:(int)defaultDistance;
// Objective-C method names are very descriptive. Always name methods according to their arguments
```

```
@end // States the end of the interface


// To access public variables from the implementation file, @property generates a setter method
// automatically. Method name is 'set' followed by @property variable name:
MyClass *myClass = [[MyClass alloc] init]; // create MyClass object instance
[myClass setCount:10];
NSLog(@"%d", [myClass count]); // prints => 10
// Or using the custom getter and setter method defined in @interface:
[myClass lengthSet:32];
NSLog(@"%i", [myClass lengthGet]); // prints => 32
// For convenience, you may use dot notation to set and access object instance variables:
myClass.count = 45;
NSLog(@"%i", myClass.count); // prints => 45

// Call class methods:
NSString *classMethodString = [MyClass classMethod];
MyClass *classFromName = [MyClass myClassFromName:@"Hello"];

// Call instance methods:
MyClass *myClass = [[MyClass alloc] init]; // Create MyClass object instance
NSString *stringFromInstanceMethod = [myClass instanceMethodWithParameter:@"Hello"];

// Selectors
// Way to dynamically represent methods. Used to call methods of a class, pass methods
// through functions to tell other classes they should call it, and to save methods
// as a variable
// SEL is the data type. @selector() returns a selector from method name provided
// methodAParameterAsString:andAParameterAsNumber: is method name for method in MyClass
SEL selectorVar = @selector(methodAParameterAsString:andAParameterAsNumber:);
if ([myClass respondsToSelector:selectorVar]) { // Checks if class contains method
    // Must put all method arguments into one object to send to performSelector function
    NSArray *arguments = [NSArray arrayWithObjects:@"Hello", @4, nil];
    [myClass performSelector:selectorVar withObject:arguments]; // Calls the method
} else {
    // NSStringFromSelector() returns a NSString of the method name of a given selector
    NSLog(@"MyClass does not have method: %@", NSStringFromSelector(selectedVar));
}

// Implement the methods in an implementation (MyClass.m) file:
@implementation MyClass {
    long distance; // Private access instance variable
    NSNumber height;
}

// To access a public variable from the interface file, use '_' followed by variable name:
_count = 5; // References "int count" from MyClass interface
// Access variables defined in implementation file:
distance = 18; // References "long distance" from MyClass implementation
// To use @property variable in implementation, use @synthesize to create accessor variable:
@synthesize roString = _roString; // _roString available now in @implementation

// Called before calling any class methods or instantiating any objects
+ (void)initialize
```

```
{
    if (self == [MyClass class]) {
        distance = 0;
    }
}

// Counterpart to initialize method. Called when an object's reference count is zero
- (void)dealloc
{
    [height release]; // If not using ARC, make sure to release class variable objects
    [super dealloc];  // and call parent class dealloc
}

// Constructors are a way of creating instances of a class
// This is a default constructor which is called when the object is initialized.
- (id)init
{
    if ((self = [super init])) // 'super' used to access methods from parent class
    {
        self.count = 1; // 'self' used for object to call itself
    }
    return self;
}
// Can create constructors that contain arguments:
- (id)initWithDistance:(int)defaultDistance
{
    distance = defaultDistance;
    return self;
}

+ (NSString *)classMethod
{
    return @"Some string";
}

+ (MyClass *)myClassFromHeight:(NSNumber *)defaultHeight
{
    height = defaultHeight;
    return [[self alloc] init];
}

- (NSString *)instanceMethodWithParameter:(NSString *)string
{
    return @"New string";
}

- (NSNumber *)methodAParameterAsString:(NSString*)string andAParameterAsNumber:(NSNumber *)number
{
    return @42;
}

// Objective-C does not have private method declarations, but you can simulate them.
// To simulate a private method, create the method in the @implementation but not in the @interface.
- (NSNumber *)secretPrivateMethod {
    return @72;
}
[self secretPrivateMethod]; // Calls private method

// Methods declared into MyProtocol
- (void)myProtocolMethod
{
    // statements
}

@end // States the end of the implementation

///////////////////////////////////////
// Categories
///////////////////////////////////////
// A category is a group of methods designed to extend a class. They allow you to add new methods
// to an existing class for organizational purposes. This is not to be mistaken with subclasses.
// Subclasses are meant to CHANGE functionality of an object while categories instead ADD
// functionality to an object.
// Categories allow you to:
// -- Add methods to an existing class for organizational purposes.
// -- Allow you to extend Objective-C object classes (ex: NSString) to add your own methods.
// -- Add ability to create protected and private methods to classes.
// NOTE: Do not override methods of the base class in a category even though you have the ability
// to. Overriding methods may cause compiler errors later between different categories and it
// ruins the purpose of categories to only ADD functionality. Subclass instead to override methods.

// Here is a simple Car base class.
@interface Car : NSObject

@property NSString *make;
@property NSString *color;

- (void)turnOn;
- (void)accelerate;

@end

// And the simple Car base class implementation:
#import "Car.h"

@implementation Car

@synthesize make = _make;
@synthesize color = _color;

- (void)turnOn {
    NSLog(@"Car is on.");
}
- (void)accelerate {
    NSLog(@"Accelerating.");
}

@end
```

```
// Now, if we wanted to create a Truck object, we would instead create a subclass of Car as it would
// be changing the functionality of the Car to behave like a truck. But lets say we want to just add
// functionality to this existing Car. A good example would be to clean the car. So we would create
// a category to add these cleaning methods:
// @interface filename: Car+Clean.h (BaseClassName+CategoryName.h)
#import "Car.h" // Make sure to import base class to extend.

@interface Car (Clean) // The category name is inside () following the name of the base class.

- (void)washWindows; // Names of the new methods we are adding to our Car object.
- (void)wax;

@end

// @implementation filename: Car+Clean.m (BaseClassName+CategoryName.m)
#import "Car+Clean.h" // Import the Clean category's @interface file.

@implementation Car (Clean)

- (void)washWindows {
    NSLog(@"Windows washed.");
}
- (void)wax {
    NSLog(@"Waxed.");
}

@end

// Any Car object instance has the ability to use a category. All they need to do is import it:
#import "Car+Clean.h" // Import as many different categories as you want to use.
#import "Car.h" // Also need to import base class to use it's original functionality.

int main (int argc, const char * argv[]) {
    @autoreleasepool {
        Car *mustang = [[Car alloc] init];
        mustang.color = @"Red";
        mustang.make = @"Ford";

        [mustang turnOn]; // Use methods from base Car class.
        [mustang washWindows]; // Use methods from Car's Clean category.
    }
    return 0;
}

// Objective-C does not have protected method declarations but you can simulate them.
// Create a category containing all of the protected methods, then import it ONLY into the
// @implementation file of a class belonging to the Car class:
@interface Car (Protected) // Naming category 'Protected' to remember methods are protected.

- (void)lockCar; // Methods listed here may only be created by Car objects.

@end
//To use protected methods, import the category, then implement the methods:
```

```
#import "Car+Protected.h" // Remember, import in the @implementation file only.

@implementation Car

- (void)lockCar {
    NSLog(@"Car locked."); // Instances of Car can't use lockCar because it's not in the @interface.
}

@end

///////////////////////////////////////
// Extensions
///////////////////////////////////////
// Extensions allow you to override public access property attributes and methods of an @interface.
// @interface filename: Shape.h
@interface Shape : NSObject // Base Shape class extension overrides below.

@property (readonly) NSNumber *numOfSides;

- (int)getNumOfSides;

@end
// You can override numOfSides variable or getNumOfSides method to edit them with an extension:
// @implementation filename: Shape.m
#import "Shape.h"
// Extensions live in the same file as the class @implementation.
@interface Shape () // () after base class name declares an extension.

@property (copy) NSNumber *numOfSides; // Make numOfSides copy instead of readonly.
-(NSNumber)getNumOfSides; // Make getNumOfSides return a NSNumber instead of an int.
-(void)privateMethod; // You can also create new private methods inside of extensions.

@end
// The main @implementation:
@implementation Shape

@synthesize numOfSides = _numOfSides;

-(NSNumber)getNumOfSides { // All statements inside of extension must be in the @implementation.
    return _numOfSides;
}
-(void)privateMethod {
    NSLog(@"Private method created by extension. Shape instances cannot call me.");
}

@end

// Starting in Xcode 7.0, you can create Generic classes,
// allowing you to provide greater type safety and clarity
// without writing excessive boilerplate.
@interface Result<__covariant A> : NSObject

- (void)handleSuccess:(void(^)(A))success
              failure:(void(^)(NSError *))failure;
```

```
@property (nonatomic) A object;

@end

// we can now declare instances of this class like
Result<NSNumber *> *result;
Result<NSArray *> *result;

// Each of these cases would be equivalent to rewriting Result's interface
// and substituting the appropriate type for A
@interface Result : NSObject
- (void)handleSuccess:(void(^)(NSArray *))success
              failure:(void(^)(NSError *))failure;
@property (nonatomic) NSArray * object;
@end

@interface Result : NSObject
- (void)handleSuccess:(void(^)(NSNumber *))success
              failure:(void(^)(NSError *))failure;
@property (nonatomic) NSNumber * object;
@end

// It should be obvious, however, that writing one
//  Class to solve a problem is always preferable to writing two

// Note that Clang will not accept generic types in @implementations,
// so your @implemnation of Result would have to look like this:

@implementation Result

- (void)handleSuccess:(void (^)(id))success
              failure:(void (^)(NSError *))failure {
  // Do something
}

@end


///////////////////////////////////////
// Protocols
///////////////////////////////////////
// A protocol declares methods that can be implemented by any class.
// Protocols are not classes themselves. They simply define an interface
// that other objects are responsible for implementing.
// @protocol filename: "CarUtilities.h"
@protocol CarUtilities <NSObject> // <NSObject> => Name of another protocol this protocol includes.
    @property BOOL engineOn; // Adopting class must @synthesize all defined @properties and
    - (void)turnOnEngine; // all defined methods.
@end
// Below is an example class implementing the protocol.
#import "CarUtilities.h" // Import the @protocol file.

@interface Car : NSObject <CarUtilities> // Name of protocol goes inside <>
```

```
    // You don't need the @property or method names here for CarUtilities. Only @implementation does.
- (void)turnOnEngineWithUtilities:(id <CarUtilities>)car; // You can use protocols as data too.
@end
// The @implementation needs to implement the @properties and methods for the protocol.
@implementation Car : NSObject <CarUtilities>

@synthesize engineOn = _engineOn; // Create a @synthesize statement for the engineOn @property.

- (void)turnOnEngine { // Implement turnOnEngine however you would like. Protocols do not define
    _engineOn = YES; // how you implement a method, it just requires that you do implement it.
}
// You may use a protocol as data as you know what methods and variables it has implemented.
- (void)turnOnEngineWithCarUtilities:(id <CarUtilities>)objectOfSomeKind {
    [objectOfSomeKind engineOn]; // You have access to object variables
    [objectOfSomeKind turnOnEngine]; // and the methods inside.
    [objectOfSomeKind engineOn]; // May or may not be YES. Class implements it however it wants.
}

@end
// Instances of Car now have access to the protocol.
Car *carInstance = [[Car alloc] init];
[carInstance setEngineOn:NO];
[carInstance turnOnEngine];
if ([carInstance engineOn]) {
    NSLog(@"Car engine is on."); // prints => "Car engine is on."
}
// Make sure to check if an object of type 'id' implements a protocol before calling protocol methods:
if ([myClass conformsToProtocol:@protocol(CarUtilities)]) {
    NSLog(@"This does not run as the MyClass class does not implement the CarUtilities protocol.");
} else if ([carInstance conformsToProtocol:@protocol(CarUtilities)]) {
    NSLog(@"This does run as the Car class implements the CarUtilities protocol.");
}
// Categories may implement protocols as well: @interface Car (CarCategory) <CarUtilities>
// You may implement many protocols: @interface Car : NSObject <CarUtilities, CarCleaning>
// NOTE: If two or more protocols rely on each other, make sure to forward-declare them:
#import "Brother.h"

@protocol Brother; // Forward-declare statement. Without it, compiler would through error.

@protocol Sister <NSObject>

- (void)beNiceToBrother:(id <Brother>)brother;

@end

// See the problem is that Sister relies on Brother, and Brother relies on Sister.
#import "Sister.h"

@protocol Sister; // These lines stop the recursion, resolving the issue.

@protocol Brother <NSObject>

- (void)beNiceToSister:(id <Sister>)sister;
```

```
@end


///////////////////////////////////////
// Blocks
///////////////////////////////////////
// Blocks are statements of code, just like a function, that are able to be used as data.
// Below is a simple block with an integer argument that returns the argument plus 4.
int (^addUp)(int n); // Declare a variable to store the block.
void (^noParameterBlockVar)(void); // Example variable declaration of block with no arguments.
// Blocks have access to variables in the same scope. But the variables are readonly and the
// value passed to the block is the value of the variable when the block is created.
int outsideVar = 17; // If we edit outsideVar after declaring addUp, outsideVar is STILL 17.
__block long mutableVar = 3; // __block makes variables writable to blocks, unlike outsideVar.
addUp = ^(int n) { // Remove (int n) to have a block that doesn't take in any parameters.
    NSLog(@"You may have as many lines in a block as you would like.");
    NSSet *blockSet; // Also, you can declare local variables.
    mutableVar = 32; // Assigning new value to __block variable.
    return n + outsideVar; // Return statements are optional.
}
int addUp = addUp(10 + 16); // Calls block code with arguments.
// Blocks are often used as arguments to functions to be called later, or for callbacks.
@implementation BlockExample : NSObject

 - (void)runBlock:(void (^)(NSString))block {
    NSLog(@"Block argument returns nothing and takes in a NSString object.");
    block(@"Argument given to block to execute."); // Calling block.
 }

 @end


///////////////////////////////////////
// Memory Management
///////////////////////////////////////
/*
For each object used in an application, memory must be allocated for that object. When the application
is done using that object, memory must be deallocated to ensure application efficiency.
Objective-C does not use garbage collection and instead uses reference counting. As long as
there is at least one reference to an object (also called "owning" an object), then the object
will be available to use (known as "ownership").

When an instance owns an object, its reference counter is increments by one. When the
object is released, the reference counter decrements by one. When reference count is zero,
the object is removed from memory.

With all object interactions, follow the pattern of:
(1) create the object, (2) use the object, (3) then free the object from memory.
*/

MyClass *classVar = [MyClass alloc]; // 'alloc' sets classVar's reference count to one. Returns pointer
[classVar release]; // Decrements classVar's reference count
// 'retain' claims ownership of existing object instance and increments reference count. Returns pointe
MyClass *newVar = [classVar retain]; // If classVar is released, object is still in memory because newV
```

```
[classVar autorelease]; // Removes ownership of object at end of @autoreleasepool block. Returns pointe

// @property can use 'retain' and 'assign' as well for small convenient definitions
@property (retain) MyClass *instance; // Release old value and retain a new one (strong reference)
@property (assign) NSSet *set; // Pointer to new value without retaining/releasing old (weak reference)

// Automatic Reference Counting (ARC)
// Because memory management can be a pain, Xcode 4.2 and iOS 4 introduced Automatic Reference Counting
// ARC is a compiler feature that inserts retain, release, and autorelease automatically for you, so wh
// you must not use retain, relase, or autorelease
MyClass *arcMyClass = [[MyClass alloc] init];
// ... code using arcMyClass
// Without ARC, you will need to call: [arcMyClass release] after you're done using arcMyClass. But wit
// there is no need. It will insert this release statement for you

// As for the 'assign' and 'retain' @property attributes, with ARC you use 'weak' and 'strong'
@property (weak) MyClass *weakVar; // 'weak' does not take ownership of object. If original instance's
// is set to zero, weakVar will automatically receive value of nil to avoid application crashing
@property (strong) MyClass *strongVar; // 'strong' takes ownership of object. Ensures object will stay

// For regular variables (not @property declared variables), use the following:
__strong NSString *strongString; // Default. Variable is retained in memory until it leaves it's scope
__weak NSSet *weakSet; // Weak reference to existing object. When existing object is released, weakSet
__unsafe_unretained NSArray *unsafeArray; // Like __weak, but unsafeArray not set to nil when existing
```

Further Reading

Wikipedia Objective-C

Programming with Objective-C. Apple PDF book

Programming with Objective-C for iOS

Programming with Objective-C for Mac OSX

iOS For High School Students: Getting Started

Ocaml

OCaml is a strictly evaluated functional language with some imperative features.

Along with StandardML and its dialects it belongs to ML language family. F# is also heavily influenced by OCaml.

Just like StandardML, OCaml features both an interpreter, that can be used interactively, and a compiler. The interpreter binary is normally called "ocaml" and the compiler is "ocamlopt". There is also a bytecode compiler, "ocamlc", but there are few reasons to use it.

It is strongly and statically typed, but instead of using manually written type annotations, it infers types of expressions using Hindley-Milner algorithm. It makes type annotations unnecessary in most cases, but can be a major source of confusion for beginners.

When you are in the top level loop, OCaml will print the inferred type after you enter an expression.

```
# let inc x = x + 1 ;;
val inc : int -> int = <fun>
# let a = 99 ;;
```

```
val a : int = 99
```

For a source file you can use “ocamlc -i /path/to/file.ml” command to print all names and type signatures.

```
$ cat sigtest.ml
let inc x = x + 1
let add x y = x + y

let a = 1

$ ocamlc -i ./sigtest.ml
val inc : int -> int
val add : int -> int -> int
val a : int
```

Note that type signatures of functions of multiple arguments are written in curried form. A function that takes multiple arguments can be represented as a composition of functions that take only one argument. The “f(x,y) = x + y” function from the example above applied to arguments 2 and 3 is equivalent to the “f0(y) = 2 + y” function applied to 3. Hence the “int -> int -> int” signature.

```
(*** Comments ***)

(* Comments are enclosed in (* and *). It's fine to nest comments. *)

(* There are no single-line comments. *)


(*** Variables and functions ***)

(* Expressions can be separated by a double semicolon symbol, ";;".
   In many cases it's redundant, but in this tutorial we use it after
   every expression for easy pasting into the interpreter shell.
   Unnecessary use of expression separators in source code files
   is often considered to be a bad style. *)

(* Variable and function declarations use "let" keyword. *)
let x = 10 ;;

(* OCaml allows single quote characters in identifiers.
   Single quote doesn't have a special meaning in this case, it's often used
   in cases when in other languages one would use names like "foo_tmp". *)
let foo = 1 ;;
let foo' = foo * 2 ;;

(* Since OCaml compiler infers types automatically, you normally don't need to
   specify argument types explicitly. However, you can do it if
   you want or need to. *)
let inc_int (x: int) : int = x + 1 ;;

(* One of the cases when explicit type annotations may be needed is
   resolving ambiguity between two record types that have fields with
   the same name. The alternative is to encapsulate those types in
   modules, but both topics are a bit out of scope of this
   tutorial. *)

(* You need to mark recursive function definitions as such with "rec" keyword. *)
```

```
let rec factorial n =
    if n = 0 then 1
    else n * factorial (n-1)
;;

(* Function application usually doesn't need parentheses around arguments *)
let fact_5 = factorial 5 ;;

(* ...unless the argument is an expression. *)
let fact_4 = factorial (5-1) ;;
let sqr2 = sqr (-2) ;;

(* Every function must have at least one argument.
   Since some funcions naturally don't take any arguments, there's
   "unit" type for it that has the only one value written as "()" *)
let print_hello () = print_endline "hello world" ;;

(* Note that you must specify "()" as argument when calling it. *)
print_hello () ;;

(* Calling a function with insufficient number of arguments
   does not cause an error, it produces a new function. *)
let make_inc x y = x + y ;; (* make_inc is int -> int -> int *)
let inc_2 = make_inc 2 ;;   (* inc_2 is int -> int *)
inc_2 3 ;; (* Evaluates to 5 *)

(* You can use multiple expressions in function body.
   The last expression becomes the return value. All other
   expressions must be of the "unit" type.
   This is useful when writing in imperative style, the simplest
   form of it is inserting a debug print. *)
let print_and_return x =
    print_endline (string_of_int x);
    x
;;

(* Since OCaml is a functional language, it lacks "procedures".
   Every function must return something. So functions that
   do not really return anything and are called solely for their
   side effects, like print_endline, return value of "unit" type. *)


(* Definitions can be chained with "let ... in" construct.
   This is roughly the same to assigning values to multiple
   variables before using them in expressions in imperative
   languages. *)
let x = 10 in
let y = 20 in
x + y ;;

(* Alternatively you can use "let ... and ... in" construct.
   This is especially useful for mutually recursive functions,
   with ordinary "let .. in" the compiler will complain about
   unbound values. *)
```

```
let rec
  is_even = function
  | 0 -> true
  | n -> is_odd (n-1)
and
  is_odd = function
  | 0 -> false
  | n -> is_even (n-1)
;;

(* Anonymous functions use the following syntax: *)
let my_lambda = fun x -> x * x ;;

(*** Operators ***)

(* There is little distintion between operators and functions.
   Every operator can be called as a function. *)

(+) 3 4  (* Same as 3 + 4 *)

(* There's a number of built-in operators. One unusual feature is
   that OCaml doesn't just refrain from any implicit conversions
   between integers and floats, it also uses different operators
   for floats. *)
12 + 3 ;; (* Integer addition. *)
12.0 +. 3.0 ;; (* Floating point addition. *)

12 / 3 ;; (* Integer division. *)
12.0 /. 3.0 ;; (* Floating point division. *)
5 mod 2 ;; (* Remainder. *)

(* Unary minus is a notable exception, it's polymorphic.
   However, it also has "pure" integer and float forms. *)
- 3 ;; (* Polymorphic, integer *)
- 4.5 ;; (* Polymorphic, float *)
~- 3 (* Integer only *)
~- 3.4 (* Type error *)
~-. 3.4 (* Float only *)

(* You can define your own operators or redefine existing ones.
   Unlike SML or Haskell, only selected symbols can be used
   for operator names and first symbol defines associativity
   and precedence rules. *)
let (+) a b = a - b ;; (* Surprise maintenance programmers. *)

(* More useful: a reciprocal operator for floats.
   Unary operators must start with "~". *)
let (~/) x = 1.0 /. x ;;
~/4.0 (* = 0.25 *)


(*** Built-in data structures ***)

(* Lists are enclosed in square brackets, items are separated by
   semicolons. *)
let my_list = [1; 2; 3] ;;

(* Tuples are (optionally) enclosed in parentheses, items are separated
   by commas. *)
let first_tuple = 3, 4 ;; (* Has type "int * int". *)
let second_tuple = (4, 5) ;;

(* Corollary: if you try to separate list items by commas, you get a list
   with a tuple inside, probably not what you want. *)
let bad_list = [1, 2] ;; (* Becomes [(1, 2)] *)

(* You can access individual list items with the List.nth function. *)
List.nth my_list 1 ;;

(* There are higher-order functions for lists such as map and filter. *)
List.map (fun x -> x * 2) [1; 2; 3] ;;
List.filter (fun x -> if x mod 2 = 0 then true else false) [1; 2; 3; 4] ;;

(* You can add an item to the beginning of a list with the "::" constructor
   often referred to as "cons". *)
1 :: [2; 3] ;; (* Gives [1; 2; 3] *)

(* Arrays are enclosed in [| |] *)
let my_array = [| 1; 2; 3 |] ;;

(* You can access array items like this: *)
my_array.(0) ;;


(*** Strings and characters ***)

(* Use double quotes for string literals. *)
let my_str = "Hello world" ;;

(* Use single quotes for character literals. *)
let my_char = 'a' ;;

(* Single and double quotes are not interchangeable. *)
let bad_str = 'syntax error' ;; (* Syntax error. *)

(* This will give you a single character string, not a character. *)
let single_char_str = "w" ;;

(* Strings can be concatenated with the "^" operator. *)
let some_str = "hello" ^ "world" ;;

(* Strings are not arrays of characters.
   You can't mix characters and strings in expressions.
   You can convert a character to a string with "String.make 1 my_char".
   There are more convenient functions for this purpose in additional
   libraries such as Core.Std that may not be installed and/or loaded
   by default. *)
let ocaml = (String.make 1 'O') ^ "Caml" ;;
```

```
(* There is a printf function. *)
Printf.printf "%d %s" 99 "bottles of beer" ;;

(* Unformatted read and write functions are there too. *)
print_string "hello world\n" ;;
print_endline "hello world" ;;
let line = read_line () ;;


(*** User-defined data types ***)

(* You can define types with the "type some_type =" construct. Like in this
   useless type alias: *)
type my_int = int ;;

(* More interesting types include so called type constructors.
   Constructors must start with a capital letter. *)
type ml = OCaml | StandardML ;;
let lang = OCaml ;;  (* Has type "ml". *)

(* Type constructors don't need to be empty. *)
type my_number = PlusInfinity | MinusInfinity | Real of float ;;
let r0 = Real (-3.4) ;; (* Has type "my_number". *)

(* Can be used to implement polymorphic arithmetics. *)
type number = Int of int | Float of float ;;

(* Point on a plane, essentially a type-constrained tuple *)
type point2d = Point of float * float ;;
let my_point = Point (2.0, 3.0) ;;

(* Types can be parameterized, like in this type for "list of lists
   of anything". 'a can be substituted with any type. *)
type 'a list_of_lists = 'a list list ;;
type int_list_list = int list_of_lists ;;

(* Types can also be recursive. Like in this type analogous to
   built-in list of integers. *)
type my_int_list = EmptyList | IntList of int * my_int_list ;;
let l = IntList (1, EmptyList) ;;


(*** Pattern matching ***)

(* Pattern matching is somewhat similar to switch statement in imperative
   languages, but offers a lot more expressive power.

   Even though it may look complicated, it really boils down to matching
   an argument against an exact value, a predicate, or a type constructor.
   The type system is what makes it so powerful. *)

(** Matching exact values.  **)
```

```
let is_zero x =
    match x with
    | 0 -> true
    | _ -> false  (* The "_" pattern means "anything else". *)
;;

(* Alternatively, you can use the "function" keyword. *)
let is_one = function
| 1 -> true
| _ -> false
;;

(* Matching predicates, aka "guarded pattern matching". *)
let abs x =
    match x with
    | x when x < 0 -> -x
    | _ -> x
;;

abs 5 ;; (* 5 *)
abs (-5) (* 5 again *)

(** Matching type constructors **)

type animal = Dog of string | Cat of string ;;

let say x =
    match x with
    | Dog x -> x ^ " says woof"
    | Cat x -> x ^ " says meow"
;;

say (Cat "Fluffy") ;; (* "Fluffy says meow". *)

(** Traversing data structures with pattern matching **)

(* Recursive types can be traversed with pattern matching easily.
   Let's see how we can traverse a data structure of the built-in list type.
   Even though the built-in cons ("::") looks like an infix operator,
   it's actually a type constructor and can be matched like any other. *)
let rec sum_list l =
    match l with
    | [] -> 0
    | head :: tail -> head + (sum_list tail)
;;

sum_list [1; 2; 3] ;; (* Evaluates to 6 *)

(* Built-in syntax for cons obscures the structure a bit, so we'll make
   our own list for demonstration. *)

type int_list = Nil | Cons of int * int_list ;;
let rec sum_int_list l =
  match l with
```

```
      | Nil -> 0
      | Cons (head, tail) -> head + (sum_int_list tail)
;;

let t = Cons (1, Cons (2, Cons (3, Nil))) ;;
sum_int_list t ;;
```

Further reading

- Visit the official website to get the compiler and read the docs: http://ocaml.org/
- Try interactive tutorials and a web-based interpreter by OCaml Pro: http://try.ocamlpro.com/
- Read "OCaml for the skeptical" course: http://www2.lib.uchicago.edu/keith/ocaml-class/home.html

Paren

Paren is a dialect of Lisp. It is designed to be an embedded language.

Some examples are from http://learnxinyminutes.com/docs/racket/.

```
;;; Comments
# comments

;; Single line comments start with a semicolon or a sharp sign

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 1. Primitive Datatypes and Operators
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Numbers
123 ; int
3.14 ; double
6.02e+23 ; double
(int 3.14) ; => 3 : int
(double 123) ; => 123 : double

;; Function application is written (f x y z ...)
;; where f is a function and x, y, z, ... are operands
;; If you want to create a literal list of data, use (quote) to stop it from
;; being evaluated
(quote (+ 1 2)) ; => (+ 1 2)
;; Now, some arithmetic operations
(+ 1 1)  ; => 2
(- 8 1)  ; => 7
(* 10 2) ; => 20
(^ 2 3) ; => 8
(/ 5 2) ; => 2
(% 5 2) ; => 1
(/ 5.0 2) ; => 2.5

;;; Booleans
true ; for true
false ; for false
(! true) ; => false
```

```
(&& true false (prn "doesn't get here")) ; => false
(|| false true (prn "doesn't get here")) ; => true

;;; Characters are ints.
(char-at "A" 0) ; => 65
(chr 65) ; => "A"

;;; Strings are fixed-length array of characters.
"Hello, world!"
"Benjamin \"Bugsy\" Siegel"   ; backslash is an escaping character
"Foo\tbar\r\n" ; includes C escapes: \t \r \n

;; Strings can be added too!
(strcat "Hello " "world!") ; => "Hello world!"

;; A string can be treated like a list of characters
(char-at "Apple" 0) ; => 65

;; Printing is pretty easy
(pr "I'm" "Paren. ") (prn "Nice to meet you!")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 2. Variables
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; You can create or set a variable using (set)
;; a variable name can use any character except: ();#"
(set some-var 5) ; => 5
some-var ; => 5

;; Accessing a previously unassigned variable is an exception
; x ; => Unknown variable: x : nil

;; Local binding: Use lambda calculus! `a' and `b' are bound to `1' and `2' only within the (fn ...)
((fn (a b) (+ a b)) 1 2) ; => 3

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 3. Collections
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Lists

;; Lists are vector-like data structures. (Random access is O(1).)
(cons 1 (cons 2 (cons 3 (list)))) ; => (1 2 3)
;; `list' is a convenience variadic constructor for lists
(list 1 2 3) ; => (1 2 3)
;; and a quote can also be used for a literal list value
(quote (+ 1 2)) ; => (+ 1 2)

;; Can still use `cons' to add an item to the beginning of a list
(cons 0 (list 1 2 3)) ; => (0 1 2 3)

;; Lists are a very basic type, so there is a *lot* of functionality for
;; them, a few examples:
(map inc (list 1 2 3))          ; => (2 3 4)
```

```
(filter (fn (x) (== 0 (% x 2))) (list 1 2 3 4))    ; => (2 4)
(length (list 1 2 3 4))     ; => 4

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 3. Functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Use `fn' to create functions.
;; A function always returns the value of its last expression
(fn () "Hello World") ; => (fn () Hello World) : fn

;; Use parentheses to call all functions, including a lambda expression
((fn () "Hello World")) ; => "Hello World"

;; Assign a function to a var
(set hello-world (fn () "Hello World"))
(hello-world) ; => "Hello World"

;; You can shorten this using the function definition syntatcic sugae:
(defn hello-world2 () "Hello World")

;; The () in the above is the list of arguments for the function
(set hello
  (fn (name)
    (strcat "Hello " name)))
(hello "Steve") ; => "Hello Steve"

;; ... or equivalently, using a sugared definition:
(defn hello2 (name)
  (strcat "Hello " name))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 4. Equality
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; for numbers use `=='
(== 3 3.0) ; => true
(== 2 1) ; => false

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 5. Control Flow
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Conditionals

(if true               ; test expression
    "this is true"   ; then expression
    "this is false") ; else expression
; => "this is true"

;;; Loops

;; for loop is for number
;; (for SYMBOL START END STEP EXPR ..)
```

```
(for i 0 10 2 (pr i "")) ; => prints 0 2 4 6 8 10
(for i 0.0 10 2.5 (pr i "")) ; => prints 0 2.5 5 7.5 10

;; while loop
((fn (i)
  (while (< i 10)
    (pr i)
    (++ i))) 0) ; => prints 0123456789

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 6. Mutation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Use `set' to assign a new value to a variable or a place
(set n 5) ; => 5
(set n (inc n)) ; => 6
n ; => 6
(set a (list 1 2)) ; => (1 2)
(set (nth 0 a) 3) ; => 3
a ; => (3 2)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 7. Macros
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Macros let you extend the syntax of the language.
;; Paren macros are easy.
;; In fact, (defn) is a macro.
(defmacro setfn (name ...) (set name (fn ...)))
(defmacro defn (name ...) (def name (fn ...)))

;; Let's add an infix notation
(defmacro infix (a op ...) (op a ...))
(infix 1 + 2 (infix 3 * 4)) ; => 15

;; Macros are not hygienic, you can clobber existing variables!
;; They are code transformations.
```

Perl

Perl 5 is a highly capable, feature-rich programming language with over 25 years of development.

Perl 5 runs on over 100 platforms from portables to mainframes and is suitable for both rapid prototyping and large scale development projects.

```
# Single line comments start with a number sign.


#### Perl variable types

#  Variables begin with a sigil, which is a symbol showing the type.
#  A valid variable name starts with a letter or underscore,
#  followed by any number of letters, numbers, or underscores.
```

```perl
### Perl has three main variable types: $scalar, @array, and %hash.

## Scalars
#  A scalar represents a single value:
my $animal = "camel";
my $answer = 42;

# Scalar values can be strings, integers or floating point numbers, and
# Perl will automatically convert between them as required.

## Arrays
#  An array represents a list of values:
my @animals = ("camel", "llama", "owl");
my @numbers = (23, 42, 69);
my @mixed   = ("camel", 42, 1.23);



## Hashes
#   A hash represents a set of key/value pairs:

my %fruit_color = ("apple", "red", "banana", "yellow");

#  You can use whitespace and the "=>" operator to lay them out more nicely:

my %fruit_color = (
  apple  => "red",
  banana => "yellow",
);
# Scalars, arrays and hashes are documented more fully in perldata.
# (perldoc perldata).

# More complex data types can be constructed using references, which allow you
# to build lists and hashes within lists and hashes.

#### Conditional and looping constructs

# Perl has most of the usual conditional and looping constructs.

if ($var) {
  ...
} elsif ($var eq 'bar') {
  ...
} else {
  ...
}

unless (condition) {
  ...
}
# This is provided as a more readable version of "if (!condition)"

# the Perlish post-condition way
print "Yow!" if $zippy;
print "We have no bananas" unless $bananas;

#  while
while (condition) {
  ...
}


# for loops and iteration
for (my $i = 0; $i < $max; $i++) {
  print "index is $i";
}

for (my $i = 0; $i < @elements; $i++) {
  print "Current element is " . $elements[$i];
}

for my $element (@elements) {
  print $element;
}

# implicitly

for (@elements) {
  print;
}

# the Perlish post-condition way again
print for @elements;

#### Regular expressions

# Perl's regular expression support is both broad and deep, and is the subject
# of lengthy documentation in perlrequick, perlretut, and elsewhere.
# However, in short:

# Simple matching
if (/foo/)       { ... }  # true if $_ contains "foo"
if ($a =~ /foo/) { ... }  # true if $a contains "foo"

# Simple substitution

$a =~ s/foo/bar/;         # replaces foo with bar in $a
$a =~ s/foo/bar/g;        # replaces ALL INSTANCES of foo with bar in $a


#### Files and I/O

# You can open a file for input or output using the "open()" function.

open(my $in,  "<",  "input.txt")  or die "Can't open input.txt: $!";
open(my $out, ">",  "output.txt") or die "Can't open output.txt: $!";
open(my $log, ">>", "my.log")     or die "Can't open my.log: $!";
```

```
# You can read from an open filehandle using the "<>" operator.  In scalar
# context it reads a single line from the filehandle, and in list context it
# reads the whole file in, assigning each line to an element of the list:

my $line  = <$in>;
my @lines = <$in>;

#### Writing subroutines

# Writing subroutines is easy:

sub logger {
  my $logmessage = shift;

  open my $logfile, ">>", "my.log" or die "Could not open my.log: $!";

  print $logfile $logmessage;
}

# Now we can use the subroutine just as any other built-in function:

logger("We have a logger subroutine!");
```

Using Perl modules

Perl modules provide a range of features to help you avoid reinventing the wheel, and can be downloaded from CPAN (http://www.cpan.org/). A number of popular modules are included with the Perl distribution itself.

perlfaq contains questions and answers related to many common tasks, and often provides suggestions for good CPAN modules to use.

Further Reading

- perl-tutorial
- Learn at www.perl.com
- perldoc
- and perl built-in : `perldoc perlintro`

Perl6

Perl 6 is a highly capable, feature-rich programming language made for at least the next hundred years.

The primary Perl 6 compiler is called Rakudo, which runs on the JVM and the MoarVM.

Meta-note : the triple pound signs are here to denote headlines, double paragraphs, and single notes.

`#=>` represents the output of a command.

```
# Single line comment start with a pound

#`(
  Multiline comments use #` and a quoting construct.
  (), [], {},  , etc, will work.
)
```

```
### Variables

# In Perl 6, you declare a lexical variable using `my`
my $variable;
# Perl 6 has 4 kinds of variables:

## * Scalars. They represent a single value. They start with a `$`

my $str = 'String';
# double quotes allow for interpolation (which we'll see later):
my $str2 = "String";

# variable names can contain but not end with simple quotes and dashes,
#  and can contain (and end with) underscores :
# my $weird'variable-name_ = 5; # works !

my $bool = True; # `True` and `False` are Perl 6's boolean
my $inverse = !$bool; # You can invert a bool with the prefix `!` operator
my $forced-bool = so $str; # And you can use the prefix `so` operator
                           # which turns its operand into a Bool

## * Lists. They represent multiple values. Their name start with `@`.

my @array = 'a', 'b', 'c';
# equivalent to :
my @letters = <a b c>; # array of words, delimited by space.
                     # Similar to perl5's qw, or Ruby's %w.
my @array = 1, 2, 3;

say @array[2]; # Array indices start at 0 -- This is the third element

say "Interpolate an array using [] : @array[]";
#=> Interpolate an array using [] : 1 2 3

@array[0] = -1; # Assign a new value to an array index
@array[0, 1] = 5, 6; # Assign multiple values

my @keys = 0, 2;
@array[@keys] = @letters; # Assign using an array
say @array; #=> a 6 b

## * Hashes, or key-value Pairs.
# Hashes are actually arrays of Pairs
# (you can construct a Pair object using the syntax `Key => Value`),
#  except they get "flattened" (hash context), removing duplicated keys.
my %hash = 1 => 2,
           3 => 4;
my %hash = foo => "bar", # keys get auto-quoted
            "some other" => "value", # trailing commas are okay
            ;
my %hash = <key1 value1 key2 value2>; # you can also create a hash
                                      # from an even-numbered array
my %hash = key1 => 'value1', key2 => 'value2'; # same as this
```

```
# You can also use the "colon pair" syntax:
# (especially handy for named parameters that you'll see later)
my %hash = :w(1), # equivalent to `w => 1`
           # this is useful for the `True` shortcut:
           :truey, # equivalent to `:truey(True)`, or `truey => True`
           # and for the `False` one:
           :!falsey, # equivalent to `:falsey(False)`, or `falsey => False`
           ;

say %hash{'key1'}; # You can use {} to get the value from a key
say %hash<key2>;   # If it's a string, you can actually use <>
                   # (`{key1}` doesn't work, as Perl6 doesn't have barewords)

## * Subs (subroutines, or functions in most other languages).
sub say-hello { say "Hello, world" }

sub say-hello-to(Str $name) { # You can provide the type of an argument
                              # and it'll be checked at compile-time.

    say "Hello, $name !";
}

## It can also have optional arguments:
sub with-optional($arg?) { # the "?" marks the argument optional
  say "I might return `(Any)` (Perl's 'null'-like value) if I don't have
        an argument passed, or I'll return my argument";
  $arg;
}
with-optional; # returns Any
with-optional(); # returns Any
with-optional(1); # returns 1

## You can also give them a default value when they're not passed:
sub hello-to($name = "World") {
  say "Hello, $name !";
}
hello-to; #=> Hello, World !
hello-to(); #=> Hello, World !
hello-to('You'); #=> Hello, You !

## You can also, by using a syntax akin to the one of hashes (yay unified syntax !),
##  pass *named* arguments to a `sub`.
# They're optional, and will default to "Any".
sub with-named($normal-arg, :$named) {
  say $normal-arg + $named;
}
with-named(1, named => 6); #=> 7
# There's one gotcha to be aware of, here:
# If you quote your key, Perl 6 won't be able to see it at compile time,
#  and you'll have a single Pair object as a positional parameter,
#  which means this fails:
with-named(1, 'named' => 6);
```

```
with-named(2, :named(5)); #=> 7

# To make a named argument mandatory, you can use `?`'s inverse, `!`
sub with-mandatory-named(:$str!)  {
  say "$str !";
}
with-mandatory-named(str => "My String"); #=> My String !
with-mandatory-named; # run time error: "Required named parameter not passed"
with-mandatory-named(3); # run time error: "Too many positional parameters passed"

## If a sub takes a named boolean argument ...
sub takes-a-bool($name, :$bool) {
  say "$name takes $bool";
}
# ... you can use the same "short boolean" hash syntax:
takes-a-bool('config', :bool); # config takes True
takes-a-bool('config', :!bool); # config takes False

## You can also provide your named arguments with defaults:
sub named-def(:$def = 5) {
  say $def;
}
named-def; #=> 5
named-def(def => 15); #=> 15

# Since you can omit parenthesis to call a function with no arguments,
#  you need "&" in the name to store `say-hello` in a variable.
my &s = &say-hello;
my &other-s = sub { say "Anonymous function !" }

# A sub can have a "slurpy" parameter, or "doesn't-matter-how-many"
sub as-many($head, *@rest) { # `*@` (slurpy) will basically "take everything else".
                             # Note: you can have parameters *before* (like here)
                             # a slurpy one, but not *after*.
  say @rest.join(' / ') ~ " !";
}
say as-many('Happy', 'Happy', 'Birthday'); #=> Happy / Birthday !
                                           # Note that the splat (the *) did not
                                           # consume the parameter before.

## You can call a function with an array using the
# "argument list flattening" operator `|`
# (it's not actually the only role of this operator, but it's one of them)
sub concat3($a, $b, $c) {
  say "$a, $b, $c";
}
concat3(|@array); #=> a, b, c
                  # `@array` got "flattened" as a part of the argument list

### Containers
# In Perl 6, values are actually stored in "containers".
# The assignment operator asks the container on the left to store the value on
#  its right. When passed around, containers are marked as immutable.
# Which means that, in a function, you'll get an error if you try to
#  mutate one of your arguments.
# If you really need to, you can ask for a mutable container using `is rw`:
sub mutate($n is rw) {
  $n++;
  say "\$n is now $n !";
}

# If what you want a copy instead, use `is copy`.

# A sub itself returns a container, which means it can be marked as rw:
my $x = 42;
sub x-store() is rw { $x }
x-store() = 52; # in this case, the parentheses are mandatory
                # (else Perl 6 thinks `x-store` is an identifier)
say $x; #=> 52


### Control Flow Structures
## Conditionals

# - `if`
# Before talking about `if`, we need to know which values are "Truthy"
#  (represent True), and which are "Falsey" (or "Falsy") -- represent False.
# Only these values are Falsey: 0, (), {}, "", Nil, A type (like `Str` or `Int`),
#  and of course False itself.
# Every other value is Truthy.
if True {
  say "It's true !";
}

unless False {
  say "It's not false !";
}

# As you can see, you don't need parentheses around conditions.
# However, you do need the brackets around the "body" block:
# if (true) say; # This doesn't work !

# You can also use their postfix versions, with the keyword after:
say "Quite truthy" if True;

# - Ternary conditional, "?? !!" (like `x ? y : z` in some other languages)
my $a = $condition ?? $value-if-true !! $value-if-false;

# - `given`-`when` looks like other languages' `switch`, but much more
# powerful thanks to smart matching and thanks to Perl 6's "topic variable", $_.
#
# This variable contains the default argument of a block,
#  a loop's current iteration (unless explicitly named), etc.
#
# `given` simply puts its argument into `$_` (like a block would do),
#  and `when` compares it using the "smart matching" (`~~`) operator.
#
# Since other Perl 6 constructs use this variable (as said before, like `for`,
# blocks, etc), this means the powerful `when` is not only applicable along with
# a `given`, but instead anywhere a `$_` exists.
given "foo bar" {
  say $_; #=> foo bar
  when /foo/ { # Don't worry about smart matching yet - just know `when` uses it.
               # This is equivalent to `if $_ ~~ /foo/`.
    say "Yay !";
  }
  when $_.chars > 50 { # smart matching anything with True (`$a ~~ True`) is True,
                       # so you can also put "normal" conditionals.
                       # This when is equivalent to this `if`:
                       #  if $_ ~~ ($_.chars > 50) {...}
                       # Which means:
                       #  if $_.chars > 50 {...}
    say "Quite a long string !";
  }
  default { # same as `when *` (using the Whatever Star)
    say "Something else"
  }
}

## Looping constructs

# - `loop` is an infinite loop if you don't pass it arguments,
# but can also be a C-style `for` loop:
loop {
  say "This is an infinite loop !";
  last; # last breaks out of the loop, like the `break` keyword in other languages
}

loop (my $i = 0; $i < 5; $i++) {
  next if $i == 3; # `next` skips to the next iteration, like `continue`
                   # in other languages. Note that you can also use postfix
                   # conditionals, loops, etc.
  say "This is a C-style for loop !";
}

# - `for` - Passes through an array
for @array -> $variable {
  say "I've got $variable !";
}

# As we saw with given, for's default "current iteration" variable is `$_`.
# That means you can use `when` in a `for` just like you were in a `given`.
for @array {
  say "I've got $_";

  .say; # This is also allowed.
        # A dot call with no "topic" (receiver) is sent to `$_` by default
  $_.say; # the above and this are equivalent.
}

for @array {
  # You can...
  next if $_ == 3; # Skip to the next iteration (`continue` in C-like languages).
  redo if $_ == 4; # Re-do the iteration, keeping the same topic variable (`$_`).
  last if $_ == 5; # Or break out of a loop (like `break` in C-like languages).
}

# The "pointy block" syntax isn't specific to for.
# It's just a way to express a block in Perl6.
if long-computation() -> $result {
  say "The result is $result";
}

### Operators

## Since Perl languages are very much operator-based languages,
## Perl 6 operators are actually just funny-looking subroutines, in syntactic
##  categories, like infix:<+> (addition) or prefix:<!> (bool not).

## The categories are:
# - "prefix": before (like `!` in `!True`).
# - "postfix": after (like `++` in `$a++`).
# - "infix": in between (like `*` in `4 * 3`).
# - "circumfix": around (like `[`-`]` in `[1, 2]`).
# - "post-circumfix": around, after another term (like `{`-`}` in `%hash{'key'}`)

## The associativity and precedence list are explained below.

# Alright, you're set to go !

## * Equality Checking

# - `==` is numeric comparison
3 == 4; # False
3 != 4; # True

# - `eq` is string comparison
'a' eq 'b';
'a' ne 'b'; # not equal
'a' !eq 'b'; # same as above

# - `eqv` is canonical equivalence (or "deep equality")
(1, 2) eqv (1, 3);

# - `~~` is smart matching
# For a complete list of combinations, use this table:
# http://perlcabal.org/syn/S03.html#Smart_matching
'a' ~~ /a/; # true if matches regexp
'key' ~~ %hash; # true if key exists in hash
$arg ~~ &bool-returning-function; # `True` if the function, passed `$arg`
                                  # as an argument, returns `True`.
1 ~~ Int; # "has type" (check superclasses and roles)
1 ~~ True; # smart-matching against a boolean always returns that boolean
           # (and will warn).

# You also, of course, have `<`, `<=`, `>`, `>=`.
# Their string equivalent are also avaiable : `lt`, `le`, `gt`, `ge`.
3 > 4;

## * Range constructors
3 .. 7; # 3 to 7, both included
# `^` on either side them exclusive on that side :
3 ^..^ 7; # 3 to 7, not included (basically `4 .. 6`)
# This also works as a shortcut for `0..^N`:
^10; # means 0..^10

# This also allows us to demonstrate that Perl 6 has lazy/infinite arrays,
#  using the Whatever Star:
my @array = 1..*; # 1 to Infinite ! `1..Inf` is the same.
say @array[^10]; # you can pass arrays as subscripts and it'll return
                 #  an array of results. This will print
                 # "1 2 3 4 5 6 7 8 9 10" (and not run out of memory !)
# Note : when reading an infinite list, Perl 6 will "reify" the elements
# it needs, then keep them in memory. They won't be calculated more than once.
# It also will never calculate more elements that are needed.

# An array subscript can also be a closure.
# It'll be called with the length as the argument
say join(' ', @array[15..*]); #=> 15 16 17 18 19
# which is equivalent to:
say join(' ', @array[-> $n { 15..$n }]);
# Note: if you try to do either of those with an infinite array,
#       you'll trigger an infinite loop (your program won't finish)

# You can use that in most places you'd expect, even assigning to an array
my @numbers = ^20;

# Here numbers increase by "6"; more on `...` operator later.
my @seq =  3, 9 ... * > 95; # 3 9 15 21 27 [...] 81 87 93 99;
@numbers[5..*] = 3, 9 ... *; # even though the sequence is infinite,
                             # only the 15 needed values will be calculated.
say @numbers; #=> 0 1 2 3 4 3 9 15 21 [...] 81 87
              # (only 20 values)

## * And, Or
3 && 4; # 4, which is Truthy. Calls `.Bool` on `4` and gets `True`.
0 || False; # False. Calls `.Bool` on `0`

## * Short-circuit (and tight) versions of the above
$a && $b && $c; # Returns the first argument that evaluates to False,
                # or the last argument.
$a || $b;

# And because you're going to want them,
#  you also have compound assignment operators:
$a *= 2; # multiply and assignment
$b %%= 5; # divisible by and assignment
@array .= sort; # calls the `sort` method and assigns the result back

### More on subs !
```

```
# As we said before, Perl 6 has *really* powerful subs. We're going to see
# a few more key concepts that make them better than in any other language :-).

## Unpacking !
# It's the ability to "extract" arrays and keys (AKA "destructuring").
# It'll work in `my`s and in parameter lists.
my ($a, $b) = 1, 2;
say $a; #=> 1
my ($, $, $c) = 1, 2, 3; # keep the non-interesting anonymous
say $c; #=> 3

my ($head, *@tail) = 1, 2, 3; # Yes, it's the same as with "slurpy subs"
my (*@small) = 1;

sub foo(@array [$fst, $snd]) {
  say "My first is $fst, my second is $snd ! All in all, I'm @array[].";
  # (^ remember the `[]` to interpolate the array)
}
foo(@tail); #=> My first is 2, my second is 3 ! All in all, I'm 2 3


# If you're not using the array itself, you can also keep it anonymous,
#  much like a scalar:
sub first-of-array(@ [$fst]) { $fst }
first-of-array(@small); #=> 1
first-of-array(@tail); # Throws an error "Too many positional parameters passed"
                       # (which means the array is too big).

# You can also use a slurp ...
sub slurp-in-array(@ [$fst, *@rest]) { # You could keep `*@rest` anonymous
  say $fst + @rest.elems; # `.elems` returns a list's length.
                          # Here, `@rest` is `(3,)`, since `$fst` holds the `2`.
}
slurp-in-array(@tail); #=> 3

# You could even extract on a slurpy (but it's pretty useless ;-).)
sub fst(*@ [$fst]) { # or simply : `sub fst($fst) { ... }`
  say $fst;
}
fst(1); #=> 1
fst(1, 2); # errors with "Too many positional parameters passed"

# You can also destructure hashes (and classes, which you'll learn about later !)
# The syntax is basically `%hash-name (:key($variable-to-store-value-in))`.
# The hash can stay anonymous if you only need the values you extracted.
sub key-of(% (:value($val), :qua($qua))) {
  say "Got val $val, $qua times.";
}

# Then call it with a hash: (you need to keep the brackets for it to be a hash)
key-of({value => 'foo', qua => 1});
#key-of(%hash); # the same (for an equivalent `%hash`)

## The last expression of a sub is returned automatically
```

```
# (though you may use the `return` keyword, of course):
sub next-index($n) {
  $n + 1;
}
my $new-n = next-index(3); # $new-n is now 4

# This is true for everything, except for the looping constructs
# (due to performance reasons): there's reason to build a list
#  if we're just going to discard all the results.
# If you still want to build one, you can use the `do` statement prefix:
#  (or the `gather` prefix, which we'll see later)
sub list-of($n) {
  do for ^$n { # note the use of the range-to prefix operator `^` (`0..^N`)
    $_ # current loop iteration
  }
}
my @list3 = list-of(3); #=> (0, 1, 2)

## You can create a lambda with `-> {}` ("pointy block") or `{}` ("block")
my &lambda = -> $argument { "The argument passed to this lambda is $argument" }
# `-> {}` and `{}` are pretty much the same thing, except that the former can
# take arguments, and that the latter can be mistaken as a hash by the parser.

# We can, for example, add 3 to each value of an array using map:
my @arrayplus3 = map({ $_ + 3 }, @array); # $_ is the implicit argument

# A sub (`sub {}`) has different semantics than a block (`{}` or `-> {}`):
# A block doesn't have a "function context" (though it can have arguments),
#  which means that if you return from it,
#  you're going to return from the parent function. Compare:
sub is-in(@array, $elem) {
  # this will `return` out of the `is-in` sub
  # once the condition evaluated to True, the loop won't be run anymore
  map({ return True if $_ == $elem }, @array);
}
sub truthy-array(@array) {
  # this will produce an array of `True` and `False`:
  # (you can also say `anon sub` for "anonymous subroutine")
  map(sub ($i) { if $i { return True } else { return False } }, @array);
  # ^ the `return` only returns from the anonymous `sub`
}

# You can also use the "whatever star" to create an anonymous function
# (it'll stop at the furthest operator in the current expression)
my @arrayplus3 = map(*+3, @array); # `*+3` is the same as `{ $_ + 3 }`
my @arrayplus3 = map(*+*+3, @array); # Same as `-> $a, $b { $a + $b + 3 }`
                                     # also `sub ($a, $b) { $a + $b + 3 }`
say (*/2)(4); #=> 2
              # Immediatly execute the function Whatever created.
say ((*+3)/5)(5); #=> 1.6
                  # works even in parens !

# But if you need to have more than one argument (`$_`)
#  in a block (without wanting to resort to `-> {}`),
#  you can also use the implicit argument syntax, `$^`` :
map({ $^a + $^b + 3 }, @array); # equivalent to following:
map(sub ($a, $b) { $a + $b + 3 }, @array); # (here with `sub`)

# Note : those are sorted lexicographically.
# `{ $^b / $^a }` is like `-> $a, $b { $b / $a }`

## About types...
# Perl6 is gradually typed. This means you can specify the type
#  of your variables/arguments/return types, or you can omit them
#  and they'll default to "Any".
# You obviously get access to a few base types, like Int and Str.
# The constructs for declaring types are "class", "role",
#  which you'll see later.

# For now, let us examine "subset":
# a "subset" is a "sub-type" with additional checks.
# For example: "a very big integer is an Int that's greater than 500"
# You can specify the type you're subtyping (by default, Any),
#  and add additional checks with the "where" keyword:
subset VeryBigInteger of Int where * > 500;

## Multiple Dispatch
# Perl 6 can decide which variant of a `sub` to call based on the type of the
# arguments, or on arbitrary preconditions, like with a type or a `where`:

# with types
multi sub sayit(Int $n) { # note the `multi` keyword here
  say "Number: $n";
}
multi sayit(Str $s) { # a multi is a `sub` by default
  say "String: $s";
}
sayit("foo"); # prints "String: foo"
sayit(True); # fails at *compile time* with
             # "calling 'sayit' will never work with arguments of types ..."

# with arbitrary precondition (remember subsets?):
multi is-big(Int $n where * > 50) { "Yes !" } # using a closure
multi is-big(Int $ where 10..50) { "Quite." } # Using smart-matching
                                              # (could use a regexp, etc)
multi is-big(Int $) { "No" }

subset Even of Int where * %% 2;

multi odd-or-even(Even) { "Even" } # The main case using the type.
                                   # We don't name the argument.
multi odd-or-even($) { "Odd" } # "else"

# You can even dispatch based on a positional's argument presence !
multi with-or-without-you(:$with!) { # You need make it mandatory to
                                     # be able to dispatch against it.
  say "I can live ! Actually, I can't.";
}
```

```
multi with-or-without-you {
  say "Definitely can't live.";
}
# This is very, very useful for many purposes, like `MAIN` subs (covered later),
#  and even the language itself is using it in several places.
#
# - `is`, for example, is actually a `multi sub` named `trait_mod:<is>`,
#  and it works off that.
# - `is rw`, is simply a dispatch to a function with this signature:
# sub trait_mod:<is>(Routine $r, :$rw!) {}
#
# (commented because running this would be a terrible idea !)


### Scoping
# In Perl 6, contrarily to many scripting languages (like Python, Ruby, PHP),
#  you are to declare your variables before using them. You know `my`.
# (there are other declarators, `our`, `state`, ..., which we'll see later).
# This is called "lexical scoping", where in inner blocks,
#  you can access variables from outer blocks.
my $foo = 'Foo';
sub foo {
  my $bar = 'Bar';
  sub bar {
    say "$foo $bar";
  }
  &bar; # return the function
}
foo()(); #=> 'Foo Bar'

# As you can see, `$foo` and `$bar` were captured.
# But if we were to try and use `$bar` outside of `foo`,
# the variable would be undefined (and you'd get a compile time error).

# Perl 6 has another kind of scope : dynamic scope.
# They use the twigil (composed sigil) `*` to mark dynamically-scoped variables:
my $*a = 1;
# Dyamically-scoped variables depend on the current call stack,
#  instead of the current block depth.
sub foo {
  my $*foo = 1;
  bar(); # call `bar` in-place
}
sub bar {
  say $*foo; # `$*foo` will be looked in the call stack, and find `foo`'s,
             #  even though the blocks aren't nested (they're call-nested).
             #=> 1
}

### Object Model

# You declare a class with the keyword `class`, fields with `has`,
# methods with `method`. Every attribute that is private is named `$!attr`.
# Immutable public attributes are named `$.attr`
#   (you can make them mutable with `is rw`)

# Perl 6's object model ("SixModel") is very flexible,
# and allows you to dynamically add methods, change semantics, etc ...
# (this will not be covered here, and you should refer to the Synopsis).

class A {
  has $.field; # `$.field` is immutable.
               # From inside the class, use `$!field` to modify it.
  has $.other-field is rw; # You can mark a public attribute `rw`.
  has Int $!private-field = 10;

  method get-value {
    $.field + $!private-field;
  }

  method set-value($n) {
    # $.field = $n; # As stated before, you can't use the `$.` immutable version.
    $!field = $n;   # This works, because `$!` is always mutable.

    $.other-field = 5; # This works, because `$.other-field` is `rw`.
  }

  method !private-method {
    say "This method is private to the class !";
  }
};

# Create a new instance of A with $.field set to 5 :
# Note: you can't set private-field from here (more later on).
my $a = A.new(field => 5);
$a.get-value; #=> 15
#$a.field = 5; # This fails, because the `has $.field` is immutable
$a.other-field = 10; # This, however, works, because the public field
                     #  is mutable (`rw`).

## Perl 6 also has inheritance (along with multiple inheritance)

class A {
  has $.val;

  submethod not-inherited {
    say "This method won't be available on B.";
    say "This is most useful for BUILD, which we'll see later";
  }

  method bar { $.val * 5 }
}
class B is A { # inheritance uses `is`
  method foo {
    say $.val;
  }

  method bar { $.val * 10 } # this shadows A's `bar`
```

```
}

# When you use `my T $var`, `$var` starts off with `T` itself in it,
# so you can call `new` on it.
# (`.=` is just the dot-call and the assignment operator:
#  `$a .= b` is the same as `$a = $a.b`)
# Also note that `BUILD` (the method called inside `new`)
#  will set parent properties too, so you can pass `val => 5`.
my B $b .= new(val => 5);

# $b.not-inherited; # This won't work, for reasons explained above
$b.foo; # prints 5
$b.bar; #=> 50, since it calls B's `bar`

## Roles are supported too (also called Mixins in other languages)
role PrintableVal {
  has $!counter = 0;
  method print {
    say $.val;
  }
}

# you "import" a mixin (a "role") with "does":
class Item does PrintableVal {
  has $.val;

  # When `does`-ed, a `role` literally "mixes in" the class:
  #  the methods and fields are put together, which means a class can access
  #  the private fields/methods of its roles (but not the inverse !):
  method access {
    say $!counter++;
  }

  # However, this:
  # method print {}
  # is ONLY valid when `print` isn't a `multi` with the same dispatch.
  # (this means a parent class can shadow a child class's `multi print() {}`,
  #  but it's an error if a role does)

  # NOTE: You can use a role as a class (with `is ROLE`). In this case, methods
  # will be shadowed, since the compiler will consider `ROLE` to be a class.
}

### Exceptions
# Exceptions are built on top of classes, in the package `X` (like `X::IO`).
# Unlike many other languages, in Perl 6, you put the `CATCH` block *within* the
#  block to `try`. By default, a `try` has a `CATCH` block that catches
#  any exception (`CATCH { default {} }`).
# You can redefine it using `when`s (and `default`)
#  to handle the exceptions you want:
try {
  open 'foo';
  CATCH {
    when X::AdHoc { say "unable to open file !" }
    # Any other exception will be re-raised, since we don't have a `default`
    # Basically, if a `when` matches (or there's a `default`) marks the exception as
    #  "handled" so that it doesn't get re-thrown from the `CATCH`.
    # You still can re-throw the exception (see below) by hand.
  }
}

# You can throw an exception using `die`:
die X::AdHoc.new(payload => 'Error !');

# You can access the last exception with `$!` (use `$_` in a `CATCH` block)

# There are also some subtelties to exceptions. Some Perl 6 subs return a `Failure`,
#  which is a kind of "unthrown exception". They're not thrown until you tried to look
#  at their content, unless you call `.Bool`/`.defined` on them - then they're handled.
#  (the `.handled` method is `rw`, so you can mark it as `False` back yourself)
#
# You can throw a `Failure` using `fail`. Note that if the pragma `use fatal` is on,
#  `fail` will throw an exception (like `die`).
fail "foo"; # We're not trying to access the value, so no problem.
try {
  fail "foo";
  CATCH {
    default { say "It threw because we tried to get the fail's value!" }
  }
}

# There is also another kind of exception: Control exceptions.
# Those are "good" exceptions, which happen when you change your program's flow,
#  using operators like `return`, `next` or `last`.
# You can "catch" those with `CONTROL` (not 100% working in Rakudo yet).

### Packages
# Packages are a way to reuse code. Packages are like "namespaces", and any
#  element of the six model (`module`, `role`, `class`, `grammar`, `subset`
#  and `enum`) are actually packages. (Packages are the lowest common denominator)
# Packages are important - especially as Perl is well-known for CPAN,
#  the Comprehensive Perl Archive Network.
# You're not supposed to use the package keyword, usually:
#  you use `class Package::Name::Here;` to declare a class,
#  or if you only want to export variables/subs, you can use `module`:
module Hello::World { # Bracketed form
                      # If `Hello` doesn't exist yet, it'll just be a "stub",
                      #  that can be redeclared as something else later.
  # ... declarations here ...
}
unit module Parse::Text; # file-scoped form
grammar Parse::Text::Grammar { # A grammar is a package, which you could `use`
}

# You can use a module (bring its declarations into scope) with `use`
use JSON::Tiny; # if you installed Rakudo* or Panda, you'll have this module
say from-json('[1]').perl; #=> [1]
```

```
# As said before, any part of the six model is also a package.
# Since `JSON::Tiny` uses (its own) `JSON::Tiny::Actions` class, you can use it:
my $actions = JSON::Tiny::Actions.new;

# We'll see how to export variables and subs in the next part:

### Declarators
# In Perl 6, you get different behaviors based on how you declare a variable.
# You've already seen `my` and `has`, we'll now explore the others.

## * `our` (happens at `INIT` time -- see "Phasers" below)
# It's like `my`, but it also creates a package variable.
# (All packagish things (`class`, `role`, etc) are `our` by default)
module Foo::Bar {
  our $n = 1; # note: you can't put a type constraint on an `our` variable
  our sub inc {
    our sub available { # If you try to make inner `sub`s `our`...
                        # Better know what you're doing (Don't !).
      say "Don't do that. Seriously. You'd get burned.";
    }
    my sub unavailable { # `my sub` is the default
      say "Can't access me from outside, I'm my !";
    }
    say ++$n; # increment the package variable and output its value
  }
}
say $Foo::Bar::n; #=> 1
Foo::Bar::inc; #=> 2
Foo::Bar::inc; #=> 3

## * `constant` (happens at `BEGIN` time)
# You can use the `constant` keyword to declare a compile-time variable/symbol:
constant Pi = 3.14;
constant $var = 1;

# And if you're wondering, yes, it can also contain infinite lists.
constant why-not = 5, 15 ... *;
say why-not[^5]; #=> 5 15 25 35 45

## * `state` (happens at run time, but only once)
# State variables are only initialized one time
# (they exist in other langages such as C as `static`)
sub fixed-rand {
  state $val = rand;
  say $rand;
}
fixed-rand for ^10; # will print the same number 10 times

# Note, however, that they exist separately in different enclosing contexts.
# If you declare a function with a `state` within a loop, it'll re-create the
#  variable for each iteration of the loop. See:
for ^5 -> $a {
  sub foo {
    state $val = rand; # This will be a different value for every value of `$a`
  }
  for ^5 -> $b {
    say foo; # This will print the same value 5 times, but only 5.
             # Next iteration will re-run `rand`.
  }
}



### Phasers
# Phasers in Perl 6 are blocks that happen at determined points of time in your
#  program. When the program is compiled, when a for loop runs, when you leave a
#  block, when an exception gets thrown ... (`CATCH` is actually a phaser !)
# Some of them can be used for their return values, some of them can't
#  (those that can have a "[*]" in the beginning of their explanation text).
# Let's have a look !

## * Compile-time phasers
BEGIN { say "[*] Runs at compile time, as soon as possible, only once" }
CHECK { say "[*] Runs at compile time, as late as possible, only once" }

## * Run-time phasers
INIT { say "[*] Runs at run time, as soon as possible, only once" }
END { say "Runs at run time, as late as possible, only once" }

## * Block phasers
ENTER { say "[*] Runs everytime you enter a block, repeats on loop blocks" }
LEAVE { say "Runs everytime you leave a block, even when an exception
    happened. Repeats on loop blocks." }

PRE { say "Asserts a precondition at every block entry,
    before ENTER (especially useful for loops)" }
# exemple:
for 0..2 {
    PRE { $_ > 1 } # This is going to blow up with "Precondition failed"
}

POST { say "Asserts a postcondition at every block exit,
    after LEAVE (especially useful for loops)" }
for 0..2 {
    POST { $_ < 2 } # This is going to blow up with "Postcondition failed"
}

## * Block/exceptions phasers
sub {
    KEEP { say "Runs when you exit a block successfully (without throwing an exception)" }
    UNDO { say "Runs when you exit a block unsuccessfully (by throwing an exception)" }
}

## * Loop phasers
for ^5 {
  FIRST { say "[*] The first time the loop is run, before ENTER" }
  NEXT { say "At loop continuation time, before LEAVE" }
  LAST { say "At loop termination time, after LEAVE" }
```

```
}

## * Role/class phasers
COMPOSE { "When a role is composed into a class. /!\ NOT YET IMPLEMENTED" }

# They allow for cute tricks or clever code ...:
say "This code took " ~ (time - CHECK time) ~ "s to compile";

# ... or clever organization:
sub do-db-stuff {
  $db.start-transaction; # start a new transaction
  KEEP $db.commit; # commit the transaction if all went well
  UNDO $db.rollback; # or rollback if all hell broke loose
}

### Statement prefixes
# Those act a bit like phasers: they affect the behavior of the following code.
# Though, they run in-line with the executable code, so they're in lowercase.
# (`try` and `start` are theoretically in that list, but explained somewhere else)
# Note: all of these (except start) don't need explicit brackets `{` and `}`.

# - `do` (that you already saw) - runs a block or a statement as a term
# You can't normally use a statement as a value (or "term"):
#
#    my $value = if True { 1 } # `if` is a statement - parse error
#
# This works:
my $a = do if True { 5 } # with `do`, `if` is now a term.

# - `once` - Makes sure a piece of code only runs once
for ^5 { once say 1 }; #=> 1
                       # Only prints ... once.
# Like `state`, they're cloned per-scope
for ^5 { sub { once say 1 }() } #=> 1 1 1 1 1
                                # Prints once per lexical scope

# - `gather` - Co-routine thread
# Gather allows you to `take` several values in an array,
#  much like `do`, but allows you to take any expression.
say gather for ^5 {
  take $_ * 3 - 1;
  take $_ * 3 + 1;
} #=> -1 1 2 4 5 7 8 10 11 13
say join ',', gather if False {
  take 1;
  take 2;
  take 3;
} # Doesn't print anything.

# - `eager` - Evaluate statement eagerly (forces eager context)
# Don't try this at home:
#
#    eager 1..*; # this will probably hang for a while (and might crash ...).
#
```

```
# But consider:
constant thrice = gather for ^3 { say take $_ }; # Doesn't print anything
# versus:
constant thrice = eager gather for ^3 { say take $_ }; #=> 0 1 2

# - `lazy` - Defer actual evaluation until value is fetched (forces lazy context)
# Not yet implemented !!

# - `sink` - An `eager` that discards the results (forces sink context)
constant nilthingie = sink for ^3 { .say } #=> 0 1 2
say nilthingie.perl; #=> Nil

# - `quietly` - Supresses warnings
# Not yet implemented !

# - `contend` - Attempts side effects under STM
# Not yet implemented !

### More operators thingies !

## Everybody loves operators ! Let's get more of them

# The precedence list can be found here:
# http://perlcabal.org/syn/S03.html#Operator_precedence
# But first, we need a little explanation about associativity:

# * Binary operators:
$a ! $b ! $c; # with a left-associative `!`, this is `($a ! $b) ! $c`
$a ! $b ! $c; # with a right-associative `!`, this is `$a ! ($b ! $c)`
$a ! $b ! $c; # with a non-associative `!`, this is illegal
$a ! $b ! $c; # with a chain-associative `!`, this is `($a ! $b) and ($b ! $c)`
$a ! $b ! $c; # with a list-associative `!`, this is `infix:<>`

# * Unary operators:
!$a! # with left-associative `!`, this is `(!$a)!`
!$a! # with right-associative `!`, this is `!($a!)`
!$a! # with non-associative `!`, this is illegal

## Create your own operators !
# Okay, you've been reading all of that, so I guess I should try
#  to show you something exciting.
# I'll tell you a little secret (or not-so-secret):
# In Perl 6, all operators are actually just funny-looking subroutines.

# You can declare an operator just like you declare a sub:
sub prefix:<win>($winner) { # refer to the operator categories
                            # (yes, it's the "words operator" `<>`)
  say "$winner Won !";
}
win "The King"; #=> The King Won !
                # (prefix is before)

# you can still call the sub with its "full name"
say prefix:<!>(True); #=> False
```

```
sub postfix:<!>(Int $n) {
  [*] 2..$n; # using the reduce meta-operator ... See below ;-) !
}
say 5!; #=> 120
        # Postfix operators (after) have to come *directly* after the term.
        # No whitespace. You can use parentheses to disambiguate, i.e. `(5!)!`


sub infix:<times>(Int $n, Block $r) { # infix in the middle
  for ^$n {
    $r(); # You need the explicit parentheses to call the function in `$r`,
          #  else you'd be referring at the variable itself, like with `&r`.
  }
}
3 times -> { say "hello" }; #=> hello
                            #=> hello
                            #=> hello
                            # You're very recommended to put spaces
                            # around your infix operator calls.

# For circumfix and post-circumfix ones
sub circumfix:<[ ]>(Int $n) {
  $n ** $n
}
say [5]; #=> 3125
         # circumfix is around. Again, no whitespace.

sub postcircumfix:<{ }>(Str $s, Int $idx) {
  # post-circumfix is
  # "after a term, around something"
  $s.substr($idx, 1);
}
say "abc"{1}; #=> b
              # after the term `"abc"`, and around the index (1)

# This really means a lot -- because everything in Perl 6 uses this.
# For example, to delete a key from a hash, you use the `:delete` adverb
#  (a simple named argument underneath):
%h{$key}:delete;
# equivalent to:
postcircumfix:<{ }>(%h, $key, :delete); # (you can call operators like that)
# It's *all* using the same building blocks!
# Syntactic categories (prefix infix ...), named arguments (adverbs), ...,
#  - used to build the language - are available to you.

# (you are, obviously, recommended against making an operator out of
# *everything* -- with great power comes great responsibility)

## Meta operators !
# Oh boy, get ready. Get ready, because we're delving deep
#  into the rabbit's hole, and you probably won't want to go
#  back to other languages after reading that.
#  (I'm guessing you don't want to already at that point).
# Meta-operators, as their name suggests, are *composed* operators.
# Basically, they're operators that apply another operator.

## * Reduce meta-operator
# It's a prefix meta-operator that takes a binary function and
#  one or many lists. If it doesn't get passed any argument,
#  it either returns a "default value" for this operator
#  (a meaningless value) or `Any` if there's none (examples below).
#
# Otherwise, it pops an element from the list(s) one at a time, and applies
#  the binary function to the last result (or the list's first element)
#  and the popped element.
#
# To sum a list, you could use the reduce meta-operator with `+`, i.e.:
say [+] 1, 2, 3; #=> 6
# equivalent to `(1+2)+3`
say [*] 1..5; #=> 120
# equivalent to `((((1*2)*3)*4)*5)`.

# You can reduce with any operator, not just with mathematical ones.
# For example, you could reduce with `//` to get
#  the first defined element of a list:
say [//] Nil, Any, False, 1, 5; #=> False
                                # (Falsey, but still defined)


# Default value examples:
say [*] (); #=> 1
say [+] (); #=> 0
            # meaningless values, since N*1=N and N+0=N.
say [//];   #=> (Any)
            # There's no "default value" for `//`.

# You can also call it with a function you made up, using double brackets:
sub add($a, $b) { $a + $b }
say [[&add]] 1, 2, 3; #=> 6

## * Zip meta-operator
# This one is an infix meta-operator than also can be used as a "normal" operator.
# It takes an optional binary function (by default, it just creates a pair),
#  and will pop one value off of each array and call its binary function on these
#  until it runs out of elements. It returns an array with all of these new elements.
(1, 2) Z (3, 4); # ((1, 3), (2, 4)), since by default, the function makes an array
1..3 Z+ 4..6; # (5, 7, 9), using the custom infix:<+> function

# Since `Z` is list-associative (see the list above),
#  you can use it on more than one list
(True, False) Z|| (False, False) Z|| (False, False); # (True, False)

# And, as it turns out, you can also use the reduce meta-operator with it:
[Z||] (True, False), (False, False), (False, False); # (True, False)


## And to end the operator list:
```

```
## * Sequence operator
# The sequence operator is one of Perl 6's most powerful features:
# it's composed of first, on the left, the list you want Perl 6 to deduce from
#  (and might include a closure), and on the right, a value or the predicate
#  that says when to stop (or Whatever for a lazy infinite list).
my @list = 1, 2, 3 ... 10; # basic deducing
#my @list = 1, 3, 6 ... 10; # this dies because Perl 6 can't figure out the end
my @list = 1, 2, 3 ...^ 10; # as with ranges, you can exclude the last element
                            # (the iteration when the predicate matches).
my @list = 1, 3, 9 ... * > 30; # you can use a predicate
                               # (with the Whatever Star, here).
my @list = 1, 3, 9 ... { $_ > 30 }; # (equivalent to the above)

my @fib = 1, 1, *+* ... *; # lazy infinite list of fibonacci series,
                           #  computed using a closure!
my @fib = 1, 1, -> $a, $b { $a + $b } ... *; # (equivalent to the above)
my @fib = 1, 1, { $^a + $^b } ... *; #(... also equivalent to the above)
# $a and $b will always take the previous values, meaning here
#  they'll start with $a = 1 and $b = 1 (values we set by hand).
#  then $a = 1 and $b = 2 (result from previous $a+$b), and so on.

say @fib[^10]; #=> 1 1 2 3 5 8 13 21 34 55
               # (using a range as the index)
# Note : as for ranges, once reified, elements aren't re-calculated.
# That's why `@primes[^100]` will take a long time the first time you print
#  it, then be instant.

### Regular Expressions
# I'm sure a lot of you have been waiting for this one.
# Well, now that you know a good deal of Perl 6 already, we can get started.
# First off, you'll have to forget about "PCRE regexps" (perl-compatible regexps).
#
# IMPORTANT: Don't skip them because you know PCRE. They're different.
# Some things are the same (like `?`, `+`, and `*`),
#  but sometimes the semantics change (`|`).
# Make sure you read carefully, because you might trip over a new behavior.
#
# Perl 6 has many features related to RegExps. After all, Rakudo parses itself.
# We're first going to look at the syntax itself,
#  then talk about grammars (PEG-like), differences between
#  `token`, `regex` and `rule` declarators, and some more.
# Side note: you still have access to PCRE regexps using the `:P5` modifier.
#  (we won't be discussing this in this tutorial, however)
#
# In essence, Perl 6 natively implements PEG ("Parsing Expression Grammars").
# The pecking order for ambiguous parses is determined by a multi-level
#  tie-breaking test:
#  - Longest token matching. `foo\s+` beats `foo` (by 2 or more positions)
#  - Longest literal prefix. `food\w*` beats `foo\w*` (by 1)
#  - Declaration from most-derived to less derived grammars
#     (grammars are actually classes)
#  - Earliest declaration wins
say so 'a' ~~ /a/; #=> True
say so 'a' ~~ / a /; # More readable with some spaces!

# In all our examples, we're going to use the smart-matching operator against
#  a regexp. We're converting the result using `so`, but in fact, it's
#  returning a `Match` object. They know how to respond to list indexing,
#  hash indexing, and return the matched string.
# The results of the match are available as `$/` (implicitly lexically-scoped).
# You can also use the capture variables (`$0`, `$1`, ... starting at 0, not 1 !).
#
# You can also note that `~~` does not perform start/end checking
#  (meaning the regexp can be matched with just one char of the string),
#  we're going to explain later how you can do it.

# In Perl 6, you can have any alphanumeric as a literal,
# everything else has to be escaped, using a backslash or quotes.
say so 'a|b' ~~ / a '|' b /; # `True`. Wouln't mean the same if `|` wasn't escaped
say so 'a|b' ~~ / a \| b /; # `True`. Another way to escape it.

# The whitespace in a regexp is actually not significant,
#  unless you use the `:s` (`:sigspace`, significant space) modifier.
say so 'a b c' ~~ / a b c /; # `False`. Space is not significant here
say so 'a b c' ~~ /:s a b c /; # `True`. We added the modifier `:s` here.

# It is, however, important as for how modifiers (that you're gonna see just below)
#  are applied ...

## Quantifying - `?`, `+`, `*` and `**`.
# - `?` - 0 or 1
so 'ac' ~~ / a b c /; # `False`
so 'ac' ~~ / a b? c /; # `True`, the "b" matched 0 times.
so 'abc' ~~ / a b? c /; # `True`, the "b" matched 1 time.

# ... As you read just before, whitespace is important because it determines
#  which part of the regexp is the target of the modifier:
so 'def' ~~ / a b c? /; # `False`. Only the `c` is optional
so 'def' ~~ / ab?c /; # `False`. Whitespace is not significant
so 'def' ~~ / 'abc'? /; # `True`. The whole "abc" group is optional.

# Here (and below) the quantifier applies only to the `b`

# - `+` - 1 or more
so 'ac' ~~ / a b+ c /; # `False`; `+` wants at least one matching
so 'abc' ~~ / a b+ c /; # `True`; one is enough
so 'abbbbc' ~~ / a b+ c /; # `True`, matched 4 "b"s

# - `*` - 0 or more
so 'ac' ~~ / a b* c /; # `True`, they're all optional.
so 'abc' ~~ / a b* c /; # `True`
so 'abbbbc' ~~ / a b* c /; # `True`
so 'aec' ~~ / a b* c /; # `False`. "b"(s) are optional, not replaceable.

# - `**` - (Unbound) Quantifier
# If you squint hard enough, you might understand
#  why exponentation is used for quantity.
```

```
so 'abc' ~~ / a b ** 1 c /; # `True` (exactly one time)
so 'abc' ~~ / a b ** 1..3 c /; # `True` (one to three times)
so 'abbbc' ~~ / a b ** 1..3 c /; # `True`
so 'abbbbbbc' ~~ / a b ** 1..3 c /; # `False` (too much)
so 'abbbbbbc' ~~ / a b ** 3..* c /; # `True` (infinite ranges are okay)

# - `<[]>` - Character classes
# Character classes are the equivalent of PCRE's `[]` classes, but
#  they use a more perl6-ish syntax:
say 'fooa' ~~ / f <[ o a ]>+ /; #=> 'fooa'
# You can use ranges:
say 'aeiou' ~~ / a <[ e..w ]> /; #=> 'ae'
# Just like in normal regexes, if you want to use a special character, escape it
#  (the last one is escaping a space)
say 'he-he !' ~~ / 'he-' <[ a..z \! \  ]> + /; #=> 'he-he !'
# You'll get a warning if you put duplicate names
#  (which has the nice effect of catching the wrote quoting:)
'he he' ~~ / <[ h e ' ' ]> /; # Warns "Repeated characters found in characters class"

# You can also negate them ... (equivalent to `[^]` in PCRE)
so 'foo' ~~ / <-[ f o ]> + /; # False

# ... and compose them: :
so 'foo' ~~ / <[ a..z ] - [ f o ]> + /; # False (any letter except f and o)
so 'foo' ~~ / <-[ a..z ] + [ f o ]> + /; # True (no letter except f and o)
so 'foo!' ~~ / <-[ a..z ] + [ f o ]> + /; # True (the + doesn't replace the left part)

## Grouping and capturing
# Group: you can group parts of your regexp with `[]`.
# These groups are *not* captured (like PCRE's `(?:)`).
so 'abc' ~~ / a [ b ] c /; # `True`. The grouping does pretty much nothing
so 'foo012012bar' ~~ / foo [ '01' <[0..9]> ] + bar /;
# The previous line returns `True`.
# We match the "012" 1 or more time (the `+` was applied to the group).

# But this does not go far enough, because we can't actually get back what
#  we matched.
# Capture: We can actually *capture* the results of the regexp, using parentheses.
so 'fooABCABCbar' ~~ / foo ( 'A' <[A..Z]> 'C' ) + bar /; # `True`. (using `so` here, `$/` below)

# So, starting with the grouping explanations.
# As we said before, our `Match` object is available as `$/`:
say $/; # Will print some weird stuff (we'll explain) (or "Nil" if nothing matched).

# As we also said before, it has array indexing:
say $/[0]; #=> ABC  ABC
           # These weird brackets are `Match` objects.
           # Here, we have an array of these.
say $0; # The same as above.

# Our capture is `$0` because it's the first and only one capture in the regexp.
# You might be wondering why it's an array, and the answer is simple:
# Some capture (indexed using `$0`, `$/[0]` or a named one) will be an array
#  IFF it can have more than one element
#  (so, with `*`, `+` and `**` (whatever the operands), but not with `?`).
# Let's use examples to see that:
so 'fooABCbar' ~~ / foo ( A B C )? bar /; # `True`
say $/[0]; #=> ABC
say $0.WHAT; #=> (Match)
             # It can't be more than one, so it's only a single match object.
so 'foobar' ~~ / foo ( A B C )? bar /; #=> True
say $0.WHAT; #=> (Any)
             # This capture did not match, so it's empty
so 'foobar' ~~ / foo ( A B C ) ** 0..1 bar /; # `True`
say $0.WHAT; #=> (Array)
             # A specific quantifier will always capture an Array,
             #  may it be a range or a specific value (even 1).

# The captures are indexed per nesting. This means a group in a group will be nested
#  under its parent group: `$/[0][0]`, for this code:
'hello-~-world' ~~ / ( 'hello' ( <[ \- \~ ]> + ) ) 'world' /;
say $/[0].Str; #=> hello~
say $/[0][0].Str; #=> ~

# This stems from a very simple fact: `$/` does not contain strings, integers or arrays,
#  it only contains match objects. These contain the `.list`, `.hash` and `.Str` methods.
#  (but you can also just use `match<key>` for hash access
#    and `match[idx]` for array access)
say $/[0].list.perl; #=> (Match.new(...),).list
                     # We can see it's a list of Match objects. Those contain
                     # a bunch of infos: where the match started/ended,
                     #    the "ast" (see actions later), etc.
                     # You'll see named capture below with grammars.

## Alternatives - the `or` of regexps
# WARNING: They are DIFFERENT from PCRE regexps.
so 'abc' ~~ / a [ b | y ] c /; # `True`. Either "b" or "y".
so 'ayc' ~~ / a [ b | y ] c /; # `True`. Obviously enough ...

# The difference between this `|` and the one you're used to is LTM.
# LTM means "Longest Token Matching". This means that the engine will always
#  try to match as much as possible in the strng
'foo' ~~ / fo | foo /; # `foo`, because it's longer.
# To decide which part is the "longest", it first splits the regex in two parts:
# The "declarative prefix" (the part that can be statically analyzed)
#  and the procedural parts.
# Declarative prefixes include alternations (`|`), conjuctions (`&`),
#  sub-rule calls (not yet introduced), literals, characters classes and quantifiers.
# The latter include everything else: back-references, code assertions,
#  and other things that can't traditionnaly be represented by normal regexps.
#
# Then, all the alternatives are tried at once, and the longest wins.
# Exemples:
# DECLARATIVE | PROCEDURAL
/ 'foo' \d+     [ <subrule1> || <subrule2> ] /;
# DECLARATIVE (nested groups are not a problem)
/ \s* [ \w & b ] [ c | d ] /;
# However, closures and recursion (of named regexps) are procedural.
# ... There are also more complicated rules, like specificity
#  (literals win over character classes)

# Note: the first-matching `or` still exists, but is now spelled `||`
'foo' ~~ / fo || foo /; # `fo` now.




### Extra: the MAIN subroutine
# The `MAIN` subroutine is called when you run a Perl 6 file directly.
# It's very powerful, because Perl 6 actually parses the arguments
#  and pass them as such to the sub. It also handles named argument (`--foo`)
#  and will even go as far as to autogenerate a `--help`
sub MAIN($name) { say "Hello, $name !" }
# This produces:
#    $ perl6 cli.pl
#    Usage:
#      t.pl <name>

# And since it's a regular Perl 6 sub, you can haz multi-dispatch:
# (using a "Bool" for the named argument so that we can do `--replace`
#  instead of `--replace=1`)
subset File of Str where *.IO.d; # convert to IO object to check the file exists

multi MAIN('add', $key, $value, Bool :$replace) { ... }
multi MAIN('remove', $key) { ... }
multi MAIN('import', File, Str :$as) { ... } # omitting parameter name
# This produces:
#    $ perl6 cli.pl
#    Usage:
#      t.pl [--replace] add <key> <value>
#      t.pl remove <key>
#      t.pl [--as=<Str>] import (File)
# As you can see, this is *very* powerful.
# It even went as far as to show inline the constants.
# (the type is only displayed if the argument is `$`/is named)

###
### APPENDIX A:
###
### List of things
###

# It's considered by now you know the Perl6 basics.
# This section is just here to list some common operations,
#  but which are not in the "main part" of the tutorial to bloat it up

## Operators


## * Sort comparison
# They return one value of the `Order` enum : `Less`, `Same` and `More`
#  (which numerify to -1, 0 or +1).
```

```
1 <=> 4; # sort comparison for numerics
'a' leg 'b'; # sort comparison for string
$obj eqv $obj2; # sort comparison using eqv semantics

## * Generic ordering
3 before 4; # True
'b' after 'a'; # True

## * Short-circuit default operator
# Like `or` and `||`, but instead returns the first *defined* value :
say Any // Nil // 0 // 5; #=> 0

## * Short-circuit exclusive or (XOR)
# Returns `True` if one (and only one) of its arguments is true
say True ^^ False; #=> True
## * Flip Flop
# The flip flop operators (`ff` and `fff`, equivalent to P5's `..`/`...`).
#  are operators that take two predicates to test:
# They are `False` until their left side returns `True`, then are `True` until
#  their right side returns `True`.
# Like for ranges, you can exclude the iteration when it became `True`/`False`
#  by using `^` on either side.
# Let's start with an example :
for <well met young hero we shall meet later> {
  # by default, `ff`/`fff` smart-match (`~~`) against `$_`:
  if 'met' ^ff 'meet' { # Won't enter the if for "met"
                        #  (explained in details below).
    .say
  }

  if rand == 0 ff rand == 1 { # compare variables other than `$_`
    say "This ... probably will never run ...";
  }
}
# This will print "young hero we shall meet" (excluding "met"):
#  the flip-flop will start returning `True` when it first encounters "met"
#  (but will still return `False` for "met" itself, due to the leading `^`
#   on `ff`), until it sees "meet", which is when it'll start returning `False`.

# The difference between `ff` (awk-style) and `fff` (sed-style) is that
#  `ff` will test its right side right when its left side changes to `True`,
#  and can get back to `False` right away
#  (*except* it'll be `True` for the iteration that matched) -
# While `fff` will wait for the next iteration to
#  try its right side, once its left side changed:
.say if 'B' ff 'B' for <A B C B A>; #=> B B
                                    # because the right-hand-side was tested
                                    # directly (and returned `True`).
                                    # "B"s are printed since it matched that time
                                    #  (it just went back to `False` right away).
.say if 'B' fff 'B' for <A B C B A>; #=> B C B
                                    # The right-hand-side wasn't tested until
                                    #  `$_` became "C"
                                    # (and thus did not match instantly).
# A flip-flop can change state as many times as needed:
for <test start print it stop not printing start print again stop not anymore> {
  .say if $_ eq 'start' ^ff^ $_ eq 'stop'; # exclude both "start" and "stop",
                                           #=> "print it print again"
}

# you might also use a Whatever Star,
# which is equivalent to `True` for the left side or `False` for the right:
for (1, 3, 60, 3, 40, 60) { # Note: the parenthesis are superfluous here
                            # (sometimes called "superstitious parentheses")
 .say if $_ > 50 ff *; # Once the flip-flop reaches a number greater than 50,
                       #  it'll never go back to `False`
                       #=> 60 3 40 60
}

# You can also use this property to create an `If`
#  that'll not go through the first time :
for <a b c> {
  .say if * ^ff *; # the flip-flop is `True` and never goes back to `False`,
                   #  but the `^` makes it *not run* on the first iteration
                   #=> b c
}


# - `===` is value identity and uses `.WHICH` on the objects to compare them
# - `=:=` is container identity and uses `VAR()` on the objects to compare them
```

If you want to go further, you can:

- Read the Perl 6 Advent Calendar. This is probably the greatest source of Perl 6 information, snippets and such.
- Come along on `#perl6` at `irc.freenode.net`. The folks here are always helpful.
- Check the source of Perl 6's functions and classes. Rakudo is mainly written in Perl 6 (with a lot of NQP, "Not Quite Perl", a Perl 6 subset easier to implement and optimize).
- Read the language design documents. They explain P6 from an implementor point-of-view, but it's still very interesting.

Php

This document describes PHP 5+.

```
<?php // PHP code must be enclosed with <?php tags

// If your php file only contains PHP code, it is best practice
// to omit the php closing tag to prevent accidental output.

// Two forward slashes start a one-line comment.

# So will a hash (aka pound symbol) but // is more common

/*
     Surrounding text in slash-asterisk and asterisk-slash
     makes it a multi-line comment.
```

```
*/

// Use "echo" or "print" to print output
print('Hello '); // Prints "Hello " with no line break

// () are optional for print and echo
echo "World\n"; // Prints "World" with a line break
// (all statements must end with a semicolon)

// Anything outside <?php tags is echoed automatically
?>
Hello World Again!
<?php


/************************************
 * Types & Variables
 */

// Variables begin with the $ symbol.
// A valid variable name starts with a letter or underscore,
// followed by any number of letters, numbers, or underscores.

// Boolean values are case-insensitive
$boolean = true;  // or TRUE or True
$boolean = false; // or FALSE or False

// Integers
$int1 = 12;   // => 12
$int2 = -12;  // => -12
$int3 = 012;  // => 10 (a leading 0 denotes an octal number)
$int4 = 0x0F; // => 15 (a leading 0x denotes a hex literal)
// Binary integer literals are available since PHP 5.4.0.
$int5 = 0b11111111; // 255 (a leading 0b denotes a binary number)

// Floats (aka doubles)
$float = 1.234;
$float = 1.2e3;
$float = 7E-10;

// Delete variable
unset($int1);

// Arithmetic
$sum        = 1 + 1; // 2
$difference = 2 - 1; // 1
$product    = 2 * 2; // 4
$quotient   = 2 / 1; // 2

// Shorthand arithmetic
$number = 0;
$number += 1;      // Increment $number by 1
echo $number++;    // Prints 1 (increments after evaluation)
echo ++$number;    // Prints 3 (increments before evaluation)
```

```
$number /= $float; // Divide and assign the quotient to $number

// Strings should be enclosed in single quotes;
$sgl_quotes = '$String'; // => '$String'

// Avoid using double quotes except to embed other variables
$dbl_quotes = "This is a $sgl_quotes."; // => 'This is a $String.'

// Special characters are only escaped in double quotes
$escaped   = "This contains a \t tab character.";
$unescaped = 'This just contains a slash and a t: \t';

// Enclose a variable in curly braces if needed
$money = "I have $${number} in the bank.";

// Since PHP 5.3, nowdocs can be used for uninterpolated multi-liners
$nowdoc = <<<'END'
Multi line
string
END;

// Heredocs will do string interpolation
$heredoc = <<<END
Multi line
$sgl_quotes
END;

// String concatenation is done with .
echo 'This string ' . 'is concatenated';

// Strings can be passed in as parameters to echo
echo 'Multiple', 'Parameters', 'Valid';  // Returns 'MultipleParametersValid'


/********************************
 * Constants
 */

// A constant is defined by using define()
// and can never be changed during runtime!

// a valid constant name starts with a letter or underscore,
// followed by any number of letters, numbers, or underscores.
define("FOO", "something");

// access to a constant is possible by calling the choosen name without a $
echo FOO; // Returns 'something'
echo 'This outputs ' . FOO;  // Returns 'This ouputs something'



/********************************
 * Arrays
 */
```

```
// All arrays in PHP are associative arrays (hashmaps),

// Associative arrays, known as hashmaps in some languages.

// Works with all PHP versions
$associative = array('One' => 1, 'Two' => 2, 'Three' => 3);

// PHP 5.4 introduced a new syntax
$associative = ['One' => 1, 'Two' => 2, 'Three' => 3];

echo $associative['One']; // prints 1

// List literals implicitly assign integer keys
$array = ['One', 'Two', 'Three'];
echo $array[0]; // => "One"

// Add an element to the end of an array
$array[] = 'Four';
// or
array_push($array, 'Five');

// Remove element from array
unset($array[3]);

/********************************
 * Output
 */

echo('Hello World!');
// Prints Hello World! to stdout.
// Stdout is the web page if running in a browser.

print('Hello World!'); // The same as echo

// echo and print are language constructs too, so you can drop the parentheses
echo 'Hello World!';
print 'Hello World!';

$paragraph = 'paragraph';

echo 100;        // Echo scalar variables directly
echo $paragraph; // or variables

// If short open tags are configured, or your PHP version is
// 5.4.0 or greater, you can use the short echo syntax
?>
<p><?= $paragraph ?></p>
<?php

$x = 1;
$y = 2;
$x = $y; // $x now contains the same value as $y
$z = &$y;
```

```
// $z now contains a reference to $y. Changing the value of
// $z will change the value of $y also, and vice-versa.
// $x will remain unchanged as the original value of $y

echo $x; // => 2
echo $z; // => 2
$y = 0;
echo $x; // => 2
echo $z; // => 0

// Dumps type and value of variable to stdout
var_dump($z); // prints int(0)

// Prints variable to stdout in human-readable format
print_r($array); // prints: Array ( [0] => One [1] => Two [2] => Three )

/********************************
 * Logic
 */
$a = 0;
$b = '0';
$c = '1';
$d = '1';

// assert throws a warning if its argument is not true

// These comparisons will always be true, even if the types aren't the same.
assert($a == $b); // equality
assert($c != $a); // inequality
assert($c <> $a); // alternative inequality
assert($a < $c);
assert($c > $b);
assert($a <= $b);
assert($c >= $d);

// The following will only be true if the values match and are the same type.
assert($c === $d);
assert($a !== $d);
assert(1 === '1');
assert(1 !== '1');

// 'Spaceship' operator (since PHP 7)
// Returns 0 if values on either side are equal
// Returns 1 if value on the left is greater
// Returns -1 if the value on the right is greater

$a = 100;
$b = 1000;

echo $a <=> $a; // 0 since they are equal
echo $a <=> $b; // -1 since $a < $b
echo $b <=> $a; // 1 since $b > $a

// Variables can be converted between types, depending on their usage.
```

```
$integer = 1;
echo $integer + $integer; // => 2

$string = '1';
echo $string + $string; // => 2 (strings are coerced to integers)

$string = 'one';
echo $string + $string; // => 0
// Outputs 0 because the + operator cannot cast the string 'one' to a number

// Type casting can be used to treat a variable as another type

$boolean = (boolean) 1; // => true

$zero = 0;
$boolean = (boolean) $zero; // => false

// There are also dedicated functions for casting most types
$integer = 5;
$string = strval($integer);

$var = null; // Null value


/********************************
 * Control Structures
 */

if (true) {
    print 'I get printed';
}

if (false) {
    print 'I don\'t';
} else {
    print 'I get printed';
}

if (false) {
    print 'Does not get printed';
} elseif(true) {
    print 'Does';
}

// ternary operator
print (false ? 'Does not get printed' : 'Does');

// ternary shortcut operator since PHP 5.3
// equivalent of "$x ? $x : 'Does'""
$x = false;
print($x ?: 'Does');

// null coalesce operator since php 7
```

```
$a = null;
$b = 'Does print';
echo $a ?? 'a is not set'; // prints 'a is not set'
echo $b ?? 'b is not set'; // prints 'Does print'


$x = 0;
if ($x === '0') {
    print 'Does not print';
} elseif($x == '1') {
    print 'Does not print';
} else {
    print 'Does print';
}



// This alternative syntax is useful for templates:
?>

<?php if ($x): ?>
This is displayed if the test is truthy.
<?php else: ?>
This is displayed otherwise.
<?php endif; ?>

<?php

// Use switch to save some logic.
switch ($x) {
    case '0':
        print 'Switch does type coercion';
        break; // You must include a break, or you will fall through
               // to cases 'two' and 'three'
    case 'two':
    case 'three':
        // Do something if $variable is either 'two' or 'three'
        break;
    default:
        // Do something by default
}

// While, do...while and for loops are probably familiar
$i = 0;
while ($i < 5) {
    echo $i++;
}; // Prints "01234"

echo "\n";

$i = 0;
do {
    echo $i++;
} while ($i < 5); // Prints "01234"
```

```
echo "\n";

for ($x = 0; $x < 10; $x++) {
    echo $x;
} // Prints "0123456789"

echo "\n";

$wheels = ['bicycle' => 2, 'car' => 4];

// Foreach loops can iterate over arrays
foreach ($wheels as $wheel_count) {
    echo $wheel_count;
} // Prints "24"

echo "\n";

// You can iterate over the keys as well as the values
foreach ($wheels as $vehicle => $wheel_count) {
    echo "A $vehicle has $wheel_count wheels";
}

echo "\n";

$i = 0;
while ($i < 5) {
    if ($i === 3) {
        break; // Exit out of the while loop
    }
    echo $i++;
} // Prints "012"

for ($i = 0; $i < 5; $i++) {
    if ($i === 3) {
        continue; // Skip this iteration of the loop
    }
    echo $i;
} // Prints "0124"


/********************************
 * Functions
 */

// Define a function with "function":
function my_function () {
    return 'Hello';
}

echo my_function(); // => "Hello"

// A valid function name starts with a letter or underscore, followed by any
// number of letters, numbers, or underscores.
```

```php
function add ($x, $y = 1) { // $y is optional and defaults to 1
    $result = $x + $y;
    return $result;
}

echo add(4); // => 5
echo add(4, 2); // => 6

// $result is not accessible outside the function
// print $result; // Gives a warning.

// Since PHP 5.3 you can declare anonymous functions;
$inc = function ($x) {
    return $x + 1;
};

echo $inc(2); // => 3

function foo ($x, $y, $z) {
    echo "$x - $y - $z";
}

// Functions can return functions
function bar ($x, $y) {
    // Use 'use' to bring in outside variables
    return function ($z) use ($x, $y) {
        foo($x, $y, $z);
    };
}

$bar = bar('A', 'B');
$bar('C'); // Prints "A - B - C"

// You can call named functions using strings
$function_name = 'add';
echo $function_name(1, 2); // => 3
// Useful for programatically determining which function to run.
// Or, use call_user_func(callable $callback [, $parameter [, ... ]]);


// You can get the all the parameters passed to a function
function parameters() {
    $numargs = func_num_args();
    if ($numargs > 0) {
        echo func_get_arg(0) . ' | ';
    }
    $args_array = func_get_args();
    foreach ($args_array as $key => $arg) {
        echo $key . ' - ' . $arg . ' | ';
    }
}

parameters('Hello', 'World'); // Hello | 0 - Hello | 1 - World |
// Since PHP 5.6 you can get a variable number of arguments
function variable($word, ...$list) {
    echo $word . " || ";
    foreach ($list as $item) {
        echo $item . ' | ';
    }
}

variable("Separate", "Hello", "World") // Separate || Hello | World |

/********************************
 * Includes
 */

<?php
// PHP within included files must also begin with a PHP open tag.

include 'my-file.php';
// The code in my-file.php is now available in the current scope.
// If the file cannot be included (e.g. file not found), a warning is emitted.

include_once 'my-file.php';
// If the code in my-file.php has been included elsewhere, it will
// not be included again. This prevents multiple class declaration errors

require 'my-file.php';
require_once 'my-file.php';
// Same as include(), except require() will cause a fatal error if the
// file cannot be included.

// Contents of my-include.php:
<?php

return 'Anything you like.';
// End file

// Includes and requires may also return a value.
$value = include 'my-include.php';

// Files are included based on the file path given or, if none is given,
// the include_path configuration directive. If the file isn't found in
// the include_path, include will finally check in the calling script's
// own directory and the current working directory before failing.
/* */

/********************************
 * Classes
 */

// Classes are defined with the class keyword

class MyClass
{
    const MY_CONST      = 'value'; // A constant

    static $staticVar  = 'static';

    // Static variables and their visibility
    public static $publicStaticVar = 'publicStatic';
    // Accessible within the class only
    private static $privateStaticVar = 'privateStatic';
    // Accessible from the class and subclasses
    protected static $protectedStaticVar = 'protectedStatic';

    // Properties must declare their visibility
    public $property    = 'public';
    public $instanceProp;
    protected $prot = 'protected'; // Accessible from the class and subclasses
    private $priv   = 'private';   // Accessible within the class only

    // Create a constructor with __construct
    public function __construct($instanceProp) {
        // Access instance variables with $this
        $this->instanceProp = $instanceProp;
    }

    // Methods are declared as functions inside a class
    public function myMethod()
    {
        print 'MyClass';
    }

    //final keyword would make a function unoverridable
    final function youCannotOverrideMe()
    {
    }

/*
 * Declaring class properties or methods as static makes them accessible without
 * needing an instantiation of the class. A property declared as static can not
 * be accessed with an instantiated class object (though a static method can).
 */

    public static function myStaticMethod()
    {
        print 'I am static';
    }
}

// Class constants can always be accessed statically
echo MyClass::MY_CONST;    // Outputs 'value';

echo MyClass::$staticVar;  // Outputs 'static';
MyClass::myStaticMethod(); // Outputs 'I am static';

// Instantiate classes using new
$my_class = new MyClass('An instance property');
```

```
// The parentheses are optional if not passing in an argument.

// Access class members using ->
echo $my_class->property;     // => "public"
echo $my_class->instanceProp; // => "An instance property"
$my_class->myMethod();        // => "MyClass"


// Extend classes using "extends"
class MyOtherClass extends MyClass
{
    function printProtectedProperty()
    {
        echo $this->prot;
    }

    // Override a method
    function myMethod()
    {
        parent::myMethod();
        print ' > MyOtherClass';
    }
}

$my_other_class = new MyOtherClass('Instance prop');
$my_other_class->printProtectedProperty(); // => Prints "protected"
$my_other_class->myMethod();               // Prints "MyClass > MyOtherClass"

final class YouCannotExtendMe
{
}

// You can use "magic methods" to create getters and setters
class MyMapClass
{
    private $property;

    public function __get($key)
    {
        return $this->$key;
    }

    public function __set($key, $value)
    {
        $this->$key = $value;
    }
}

$x = new MyMapClass();
echo $x->property; // Will use the __get() method
$x->property = 'Something'; // Will use the __set() method

// Classes can be abstract (using the abstract keyword) or
// implement interfaces (using the implements keyword).
// An interface is declared with the interface keyword.

interface InterfaceOne
{
    public function doSomething();
}

interface InterfaceTwo
{
    public function doSomethingElse();
}

// interfaces can be extended
interface InterfaceThree extends InterfaceTwo
{
    public function doAnotherContract();
}

abstract class MyAbstractClass implements InterfaceOne
{
    public $x = 'doSomething';
}

class MyConcreteClass extends MyAbstractClass implements InterfaceTwo
{
    public function doSomething()
    {
        echo $x;
    }

    public function doSomethingElse()
    {
        echo 'doSomethingElse';
    }
}


// Classes can implement more than one interface
class SomeOtherClass implements InterfaceOne, InterfaceTwo
{
    public function doSomething()
    {
        echo 'doSomething';
    }

    public function doSomethingElse()
    {
        echo 'doSomethingElse';
    }
}


/********************************
 * Traits
 */

// Traits are available from PHP 5.4.0 and are declared using "trait"

trait MyTrait
{
    public function myTraitMethod()
    {
        print 'I have MyTrait';
    }
}

class MyTraitfulClass
{
    use MyTrait;
}

$cls = new MyTraitfulClass();
$cls->myTraitMethod(); // Prints "I have MyTrait"


/********************************
 * Namespaces
 */

// This section is separate, because a namespace declaration
// must be the first statement in a file. Let's pretend that is not the case

<?php

// By default, classes exist in the global namespace, and can
// be explicitly called with a backslash.

$cls = new \MyClass();



// Set the namespace for a file
namespace My\Namespace;

class MyClass
{
}

// (from another file)
$cls = new My\Namespace\MyClass;

//Or from within another namespace.
namespace My\Other\Namespace;

use My\Namespace\MyClass;

$cls = new MyClass();
```

```
// Or you can alias the namespace;

namespace My\Other\Namespace;

use My\Namespace as SomeOtherNamespace;

$cls = new SomeOtherNamespace\MyClass();


/**********************
* Late Static Binding
*
*/

class ParentClass {
    public static function who() {
        echo "I'm a " . __CLASS__ . "\n";
    }
    public static function test() {
        // self references the class the method is defined within
        self::who();
        // static references the class the method was invoked on
        static::who();
    }
}

ParentClass::test();
/*
I'm a ParentClass
I'm a ParentClass
*/

class ChildClass extends ParentClass {
    public static function who() {
        echo "But I'm " . __CLASS__ . "\n";
    }
}

ChildClass::test();
/*
I'm a ParentClass
But I'm ChildClass
*/

/**********************
*  Magic constants
*
*/

// Get current class name. Must be used inside a class declaration.
echo "Current class name is " . __CLASS__;

// Get full path directory of a file
echo "Current directory is " . __DIR__;
```

```
    // Typical usage
    require __DIR__ . '/vendor/autoload.php';

// Get full path of a file
echo "Current file path is " . __FILE__;

// Get current function name
echo "Current function name is " . __FUNCTION__;

// Get current line number
echo "Current line number is " . __LINE__;

// Get the name of the current method. Only returns a value when used inside a trait or object declarati
echo "Current method is " . __METHOD__;

// Get the name of the current namespace
echo "Current namespace is " . __NAMESPACE__;

// Get the name of the current trait. Only returns a value when used inside a trait or object declaratio
echo "Current namespace is " . __TRAIT__;

/**********************
*  Error Handling
*
*/

// Simple error handling can be done with try catch block

try {
    // Do something
} catch (Exception $e) {
    // Handle exception
}

// When using try catch blocks in a namespaced enviroment use the following

try {
    // Do something
} catch (\Exception $e) {
    // Handle exception
}

// Custom exceptions

class MyException extends Exception {}

try {

    $condition = true;

    if ($condition) {
        throw new MyException('Something just happend');
    }
} catch (MyException $e) {
    // Handle my exception
}
```

More Information

Visit the official PHP documentation for reference and community input.

If you're interested in up-to-date best practices, visit PHP The Right Way.

If you're coming from a language with good package management, check out Composer.

For common standards, visit the PHP Framework Interoperability Group's PSR standards.

Pogo

Pogoscript is a little language that emphasises readability, DSLs and provides excellent asynchronous primitives for writing connected JavaScript applications for the browser or server.

```
// defining a variable
water temperature = 24

// re-assigning a variable after its definition
water temperature := 26

// functions allow their parameters to be placed anywhere
temperature at (a) altitude = 32 - a / 100

// longer functions are just indented
temperature at (a) altitude :=
    if (a < 0)
        water temperature
    else
        32 - a / 100

// calling a function
current temperature = temperature at 3200 altitude

// this function constructs a new object with methods
position (x, y) = {
    x = x
    y = y

    distance from position (p) =
        dx = self.x - p.x
        dy = self.y - p.y
        Math.sqrt (dx * dx + dy * dy)
}

// `self` is similar to `this` in JavaScript with the
// exception that `self` isn't redefined in each new
// function definition
// `self` just does what you expect
// calling methods
position (7, 2).distance from position (position (5, 1))

// as in JavaScript, objects are hashes too
position.'x' == position.x == position.('x')

// arrays
positions = [
    position (1, 1)
    position (1, 2)
    position (1, 3)
]

// indexing an array
positions.0.y

n = 2
positions.(n).y

// strings
poem = 'Tail turned to red sunset on a juniper crown a lone magpie cawks.
        Mad at Oryoki in the shrine-room -- Thistles blossomed late afternoon.
        Put on my shirt and took it off in the sun walking the path to lunch.
        A dandelion seed floats above the marsh grass with the mosquitos.
        At 4 A.M. the two middleaged men sleeping together holding hands.
        In the half-light of dawn a few birds warble under the Pleiades.
        Sky reddens behind fir trees, larks twitter, sparrows cheep cheep cheep
        cheep cheep.'

// that's Allen Ginsburg

// interpolation
outlook = 'amazing!'
console.log "the weather tomorrow is going to be #(outlook)"

// regular expressions
r/(\d+)m/i
r/(\d+) degrees/mg

// operators
true @and true
false @or true
@not false
2 < 4
2 >= 2
2 > 1

// plus all the javascript ones

// to define your own
(p1) plus (p2) =
    position (p1.x + p2.x, p1.y + p2.y)
```

```
// `plus` can be called as an operator
position (1, 1) @plus position (0, 2)
// or as a function
(position (1, 1)) plus (position (0, 2))

// explicit return
(x) times (y) = return (x * y)

// new
now = @new Date ()

// functions can take named optional arguments
spark (position, color: 'black', velocity: {x = 0, y = 0}) = {
    color = color
    position = position
    velocity = velocity
}

red = spark (position 1 1, color: 'red')
fast black = spark (position 1 1, velocity: {x = 10, y = 0})

// functions can unsplat arguments too
log (messages, ...) =
    console.log (messages, ...)

// blocks are functions passed to other functions.
// This block takes two parameters, `spark` and `c`,
// the body of the block is the indented code after the
// function call

render each @(spark) into canvas context @(c)
    ctx.begin path ()
    ctx.stroke style = spark.color
    ctx.arc (
        spark.position.x + canvas.width / 2
        spark.position.y
        3
        0
        Math.PI * 2
    )
    ctx.stroke ()

// asynchronous calls

// JavaScript both in the browser and on the server (with Node.js)
// makes heavy use of asynchronous IO with callbacks. Async IO is
// amazing for performance and making concurrency simple but it
// quickly gets complicated.
// Pogoscript has a few things to make async IO much much easier

// Node.js includes the `fs` module for accessing the file system.
// Let's list the contents of a directory

fs = require 'fs'
```

```
directory listing = fs.readdir! '.'

// `fs.readdir()` is an asynchronous function, so we can call it
// using the `!` operator. The `!` operator allows you to call
// async functions with the same syntax and largely the same
// semantics as normal synchronous functions. Pogoscript rewrites
// it so that all subsequent code is placed in the callback function
// to `fs.readdir()`.

// to catch asynchronous errors while calling asynchronous functions

try
    another directory listing = fs.readdir! 'a-missing-dir'
catch (ex)
    console.log (ex)

// in fact, if you don't use `try catch`, it will raise the error up the
// stack to the outer-most `try catch` or to the event loop, as you'd expect
// with non-async exceptions

// all the other control structures work with asynchronous calls too
// here's `if else`
config =
    if (fs.stat! 'config.json'.is file ())
        JSON.parse (fs.read file! 'config.json' 'utf-8')
    else
        {
            color: 'red'
        }

// to run two asynchronous calls concurrently, use the `?` operator.
// The `?` operator returns a *future* which can be executed to
// wait for and obtain the result, again using the `!` operator

// we don't wait for either of these calls to finish
a = fs.stat? 'a.txt'
b = fs.stat? 'b.txt'

// now we wait for the calls to finish and print the results
console.log "size of a.txt is #(a!.size)"
console.log "size of b.txt is #(b!.size)"

// futures in Pogoscript are analogous to Promises
```

That’s it.

Download Node.js and `npm install pogo`.

There is plenty of documentation on http://pogoscript.org/, including a cheat sheet, a guide, and how Pogoscript translates to Javascript. Get in touch on the google group if you have questions!

Powershell

PowerShell is the Windows scripting language and configuration management framework from Microsoft built on the .NET Framework. Windows 7 and up ship with PowerShell.
Nearly all examples below can be a part of a shell script or executed directly in the shell.

A key difference with Bash is that it is mostly objects that you manipulate rather than plain text.

Read more here.

If you are uncertain about your environment:

```
Get-ExecutionPolicy -List
Set-ExecutionPolicy AllSigned
# Execution policies include:
# - Restricted: Scripts won't run.
# - RemoteSigned: Downloaded scripts run only if signed by a trusted publisher.
# - AllSigned: Scripts need to be signed by a trusted publisher.
# - Unrestricted: Run all scripts.
help about_Execution_Policies # for more info

# Current PowerShell version:
$PSVersionTable
```

Getting help:

```
# Find commands
Get-Command about_* # alias: gcm
Get-Command -Verb Add
Get-Alias ps
Get-Alias -Definition Get-Process

Get-Help ps | less # alias: help
ps | Get-Member # alias: gm

Show-Command Get-EventLog # Display GUI to fill in the parameters

Update-Help # Run as admin
```

The tutorial starts here:

```
# As you already figured, comments start with #

# Simple hello world example:
echo Hello world!
# echo is an alias for Write-Output (=cmdlet)
# Most cmdlets and functions follow the Verb-Noun naming convention

# Each command starts on a new line, or after a semicolon:
echo 'This is the first line'; echo 'This is the second line'

# Declaring a variable looks like this:
$aString="Some string"
# Or like this:
$aNumber = 5 -as [double]
$aList = 1,2,3,4,5
$aString = $aList -join '--' # yes, -split exists also
$aHashtable = @{name1='val1'; name2='val2'}
```

```
# Using variables:
echo $aString
echo "Interpolation: $aString"
echo "`$aString has length of $($aString.Length)"
echo '$aString'
echo @"
This is a Here-String
$aString
"@
# Note that ' (single quote) won't expand the variables!
# Here-Strings also work with single quote

# Builtin variables:
# There are some useful builtin variables, like
echo "Booleans: $TRUE and $FALSE"
echo "Empty value: $NULL"
echo "Last program's return value: $?"
echo "Exit code of last run Windows-based program: $LastExitCode"
echo "The last token in the last line received by the session: $$"
echo "The first token: $^"
echo "Script's PID: $PID"
echo "Full path of current script directory: $PSScriptRoot"
echo 'Full path of current script: ' + $MyInvocation.MyCommand.Path
echo "FUll path of current directory: $Pwd"
echo "Bound arguments in a function, script or code block: $PSBoundParameters"
echo "Unbound arguments: $($Args -join ', ')."
# More builtins: `help about_Automatic_Variables`

# Inline another file (dot operator)
. .\otherScriptName.ps1


### Control Flow
# We have the usual if structure:
if ($Age -is [string]) {
    echo 'But.. $Age cannot be a string!'
} elseif ($Age -lt 12 -and $Age -gt 0) {
    echo 'Child (Less than 12. Greater than 0)'
} else {
    echo 'Adult'
}

# Switch statements are more powerfull compared to most languages
$val = "20"
switch($val) {
  { $_ -eq 42 }           { "The answer equals 42"; break }
  '20'                    { "Exactly 20"; break }
  { $_ -like 's*' }       { "Case insensitive"; break }
  { $_ -clike 's*'}       { "clike, ceq, cne for case sensitive"; break }
  { $_ -notmatch '^.*$'}  { "Regex matching. cnotmatch, cnotlike, ..."; break }
  { 'x' -contains 'x'}    { "FALSE! -contains is for lists!"; break }
  default                 { "Others" }
}
```

```
# The classic for
for($i = 1; $i -le 10; $i++) {
  "Loop number $i"
}
# Or shorter
1..10 | % { "Loop number $_" }

# PowerShell also offers
foreach ($var in 'val1','val2','val3') { echo $var }
# while () {}
# do {} while ()
# do {} until ()

# Exception handling
try {} catch {} finally {}
try {} catch [System.NullReferenceException] {
    echo $_.Exception | Format-List -Force
}


### Providers
# List files and directories in the current directory
ls # or `dir`
cd ~ # goto home

Get-Alias ls # -> Get-ChildItem
# Uh!? These cmdlets have generic names because unlike other scripting
# languages, PowerShell does not only operate in the current directory.
cd HKCU: # go to the HKEY_CURRENT_USER registry hive

# Get all providers in your session
Get-PSProvider


### Pipeline
# Cmdlets have parameters that control their execution:
Get-ChildItem -Filter *.txt -Name # Get just the name of all txt files
# Only need to type as much of a parameter name until it is no longer ambiguous
ls -fi *.txt -n # -f is not possible because -Force also exists
# Use `Get-Help Get-ChildItem -Full` for a complete overview

# Results of the previous cmdlet can be passed to the next as input.
# `$_` is the current object in the pipeline object.
ls | Where-Object { $_.Name -match 'c' } | Export-CSV export.txt
ls | ? { $_.Name -match 'c' } | ConvertTo-HTML | Out-File export.html

# If you get confused in the pipeline use `Get-Member` for an overview
# of the available methods and properties of the pipelined objects:
ls | Get-Member
Get-Date | gm

# ` is the line continuation character. Or end the line with a |
Get-Process | Sort-Object ID -Descending | Select-Object -First 10 Name,ID,VM `
```

```
    | Stop-Process -WhatIf

Get-EventLog Application -After (Get-Date).AddHours(-2) | Format-List

# Use % as a shorthand for ForEach-Object
(a,b,c) | ForEach-Object `
    -Begin { "Starting"; $counter = 0 } `
    -Process { "Processing $_"; $counter++ } `
    -End { "Finishing: $counter" }

# Get-Process as a table with three columns
# The third column is the value of the VM property in MB and 2 decimal places
# Computed columns can be written more verbose as:
# `@{name='lbl';expression={$_}`
ps | Format-Table ID,Name,@{n='VM(MB)';e={'{0:n2}' -f ($_.VM / 1MB)}} -autoSize


### Functions
# The [string] attribute is optional.
function foo([string]$name) {
    echo "Hey $name, have a function"
}

# Calling your function
foo "Say my name"

# Functions with named parameters, parameter attributes, parsable documention
<#
.SYNOPSIS
Setup a new website
.DESCRIPTION
Creates everything your new website needs for much win
.PARAMETER siteName
The name for the new website
.EXAMPLE
New-Website -Name FancySite -Po 5000
New-Website SiteWithDefaultPort
New-Website siteName 2000 # ERROR! Port argument could not be validated
('name1','name2') | New-Website -Verbose
#>
function New-Website() {
    [CmdletBinding()]
    param (
        [Parameter(ValueFromPipeline=$true, Mandatory=$true)]
        [Alias('name')]
        [string]$siteName,
        [ValidateSet(3000,5000,8000)]
        [int]$port = 3000
    )
    BEGIN { Write-Verbose 'Creating new website(s)' }
    PROCESS { echo "name: $siteName, port: $port" }
    END { Write-Verbose 'Website(s) created' }
}
```

```
### It's all .NET
# A PS string is in fact a .NET System.String
# All .NET methods and properties are thus available
'string'.ToUpper().Replace('G', 'ggg')
# Or more powershellish
'string'.ToUpper() -replace 'G', 'ggg'

# Unsure how that .NET method is called again?
'string' | gm

# Syntax for calling static .NET methods
[System.Reflection.Assembly]::LoadWithPartialName('Microsoft.VisualBasic')

# Note that .NET functions MUST be called with parentheses
# while PS functions CANNOT be called with parentheses.
# If you do call a cmdlet/PS function with parentheses,
# it is the same as passing a single parameter list
$writer = New-Object System.IO.StreamWriter($path, $true)
$writer.Write([Environment]::NewLine)
$writer.Dispose()

### IO
# Reading a value from input:
$Name = Read-Host "What's your name?"
echo "Hello, $Name!"
[int]$Age = Read-Host "What's your age?"

# Test-Path, Split-Path, Join-Path, Resolve-Path
# Get-Content filename # returns a string[]
# Set-Content, Add-Content, Clear-Content
Get-Command ConvertTo-*,ConvertFrom-*


### Useful stuff
# Refresh your PATH
$env:PATH = [System.Environment]::GetEnvironmentVariable("Path", "Machine") +
    ";" + [System.Environment]::GetEnvironmentVariable("Path", "User")

# Find Python in path
$env:PATH.Split(";") | Where-Object { $_ -like "*python*"}

# Change working directory without having to remember previous path
Push-Location c:\temp # change working directory to c:\temp
Pop-Location # change back to previous working directory
# Aliases are: pushd and popd

# Unblock a directory after download
Get-ChildItem -Recurse | Unblock-File

# Open Windows Explorer in working directory
ii .

# Any key to exit
```

```
$host.UI.RawUI.ReadKey()
return

# Create a shortcut
$WshShell = New-Object -comObject WScript.Shell
$Shortcut = $WshShell.CreateShortcut($link)
$Shortcut.TargetPath = $file
$Shortcut.WorkingDirectory = Split-Path $file
$Shortcut.Save()
```

Configuring your shell

```
# $Profile is the full path for your `Microsoft.PowerShell_profile.ps1`
# All code there will be executed when the PS session starts
if (-not (Test-Path $Profile)) {
    New-Item -Type file -Path $Profile -Force
    notepad $Profile
}
# More info: `help about_profiles`
# For a more usefull shell, be sure to check the project PSReadLine below
```

Interesting Projects

- Channel9 PowerShell tutorials
- PSGet NuGet for PowerShell
- PSReadLine A bash inspired readline implementation for PowerShell (So good that it now ships with Windows10 by default!)
- Posh-Git Fancy Git Prompt (Recommended!)
- PSake Build automation tool
- Pester BDD Testing Framework
- Jump-Location Powershell `cd` that reads your mind
- PowerShell Community Extensions (Dead)

Not covered

- WMI: Windows Management Intrumentation (Get-CimInstance)

- Multitasking: Start-Job -scriptBlock {…},
- Code Signing
- Remoting (Enter-PSSession/Exit-PSSession; Invoke-Command)

Purescript

PureScript is a small strongly, statically typed language compiling to Javascript.

- Learn more at http://www.purescript.org/
- Documentation: http://pursuit.purescript.org/
- Book: Purescript by Example, https://leanpub.com/purescript/

All the noncommented lines of code can be run in the PSCI REPL, though some will require the `--multi-line-mode` flag.

```
--
-- 1. Primitive datatypes that corresponds to their Javascript
-- equivalents at runtime.

import Prelude
```

```
-- Numbers
1.0 + 7.2*5.5 :: Number -- 40.6
-- Ints
1 + 2*5 :: Int -- 11
-- Types are inferred, so the following works fine
9.0/2.5 + 4.4 -- 8.0
-- But Ints and Numbers don't mix, so the following won't
5/2 + 2.5 -- Expression 2.5 does not have type Int
-- Hexadecimal literals
0xff + 1 -- 256
-- Unary negation
6 * -3 -- -18
6 * negate 3 -- -18
-- Modulus, from purescript-math (Math)
3.0 % 2.0 -- 1.0
4.0 % 2.0 -- 0.0
-- Inspect the type of an expression in psci
:t 9.5/2.5 + 4.4 -- Prim.Number

-- Booleans
true :: Boolean -- true
false :: Boolean -- false
-- Negation
not true -- false
23 == 23 -- true
1 /= 4 -- true
1 >= 4 -- false
-- Comparisions < <= > >=
-- are defined in terms of compare
compare 1 2 -- LT
compare 2 2 -- EQ
compare 3 2 -- GT
-- Conjunction and Disjunction
true && (9 >= 19 || 1 < 2) -- true

-- Strings
"Hellow" :: String -- "Hellow"
-- Multiline string without newlines, to run in psci use the --multi-line-mode flag
"Hellow\
\orld" -- "Helloworld"
-- Multiline string with newlines
"""Hello
world""" -- "Hello\nworld"
-- Concatenate
"such " ++ "amaze" -- "such amaze"

--
-- 2. Arrays are Javascript arrays, but must be homogeneous

[1,1,2,3,5,8] :: Array Number -- [1,1,2,3,5,8]
[true, true, false] :: Array Boolean -- [true,true,false]
-- [1,2, true, "false"] won't work
-- `Cannot unify Prim.Int with Prim.Boolean`
-- Cons (prepend)
```

```
1 : [2,4,3] -- [1,2,4,3]

-- Requires purescript-arrays (Data.Array)
-- and purescript-maybe (Data.Maybe)

-- Safe access return Maybe a
head [1,2,3] -- Just (1)
tail [3,2,1] -- Just ([2,1])
init [1,2,3] -- Just ([1,2])
last [3,2,1] -- Just (1)
-- Random access - indexing
[3,4,5,6,7] !! 2 -- Just (5)
-- Range
1..5 -- [1,2,3,4,5]
length [2,2,2] -- 3
drop 3 [5,4,3,2,1] -- [2,1]
take 3 [5,4,3,2,1] -- [5,4,3]
append [1,2,3] [4,5,6] -- [1,2,3,4,5,6]

--
-- 3. Records are Javascript objects, with zero or more fields, which
-- can have different types.
-- In psci you have to write `let` in front of the function to get a
-- top level binding.
let book = {title: "Foucault's pendulum", author: "Umberto Eco"}
-- Access properties
book.title -- "Foucault's pendulum"

let getTitle b = b.title
-- Works on all records with a title (but doesn't require any other field)
getTitle book -- "Foucault's pendulum"
getTitle {title: "Weekend in Monaco", artist: "The Rippingtons"} -- "Weekend in Monaco"
-- Can use underscores as shorthand
_.title book -- "Foucault's pendulum"
-- Update a record
let changeTitle b t = b {title = t}
getTitle (changeTitle book "Ill nome della rosa") -- "Ill nome della rosa"

--
-- 4. Functions
-- In psci's multiline mode
let sumOfSquares :: Int -> Int -> Int
    sumOfSquares x y = x*x + y*y
sumOfSquares 3 4 -- 25
let myMod x y = x % y
myMod 3.0 2.0 -- 1.0
-- Infix application of function
3 `mod` 2 -- 1

-- function application has higher precedence than all other
-- operators
sumOfSquares 3 4 * sumOfSquares 4 5 -- 1025

-- Conditional
```

```
let abs' n = if n>=0 then n else -n
abs' (-3) -- 3

-- Guarded equations
let abs'' n | n >= 0    = n
            | otherwise = -n

-- Pattern matching

-- Note the type signature, input is a list of numbers. The pattern matching
-- destructures and binds the list into parts.
-- Requires purescript-lists (Data.List)
let first :: forall a. List a -> a
    first (Cons x _) = x
first (toList [3,4,5]) -- 3
let second :: forall a. List a -> a
    second (Cons _ (Cons y _)) = y
second (toList [3,4,5]) -- 4
let sumTwo :: List Int -> List Int
    sumTwo (Cons x (Cons y rest)) = x + y : rest
fromList (sumTwo (toList [2,3,4,5,6])) :: Array Int -- [5,4,5,6]

-- sumTwo doesn't handle when the list is empty or there's only one element in
-- which case you get an error.
sumTwo [1] -- Failed pattern match

-- Complementing patterns to match
-- Good ol' Fibonacci
let fib 1 = 1
    fib 2 = 2
    fib x = fib (x-1) + fib (x-2)
fib 10 -- 89

-- Use underscore to match any, where you don't care about the binding name
let isZero 0 = true
    isZero _ = false

-- Pattern matching on records
let ecoTitle {author = "Umberto Eco", title = t} = Just t
    ecoTitle _ = Nothing

ecoTitle book -- Just ("Foucault's pendulum")
ecoTitle {title: "The Quantum Thief", author: "Hannu Rajaniemi"} -- Nothing
-- ecoTitle requires both field to type check:
ecoTitle {title: "The Quantum Thief"} -- Object lacks required property "author"

-- Lambda expressions
(\x -> x*x) 3 -- 9
(\x y -> x*x + y*y) 4 5 -- 41
let sqr = \x -> x*x

-- Currying
let myAdd x y = x + y -- is equivalent with
let myAdd' = \x -> \y -> x + y
let add3 = myAdd 3
:t add3 -- Prim.Int -> Prim.Int

-- Forward and backward function composition
-- drop 3 followed by taking 5
(drop 3 >>> take 5) (1..20) -- [4,5,6,7,8]
-- take 5 followed by dropping 3
(drop 3 <<< take 5) (1..20) -- [4,5]

-- Operations using higher order functions
let even x = x `mod` 2 == 0
filter even (1..10) -- [2,4,6,8,10]
map (\x -> x + 11) (1..5) -- [12,13,14,15,16]

-- Requires purescript-foldable-traversabe (Data.Foldable)

foldr (+) 0 (1..10) -- 55
sum (1..10) -- 55
product (1..10) -- 3628800

-- Testing with predicate
any even [1,2,3] -- true
all even [1,2,3] -- false
```

Python

Python was created by Guido Van Rossum in the early 90s. It is now one of the most popular languages in existence. I fell in love with Python for its syntactic clarity. It's basically executable pseudocode.

Feedback would be highly appreciated! You can reach me at [@louiedinh](http://twitter.com/louiedinh) or louiedinh [at] [google's email service]

Note: This article applies to Python 2.7 specifically, but should be applicable to Python 2.x. Python 2.7 is reaching end of life and will stop being maintained in 2020, it is though recommended to start learning Python with Python 3. For Python 3.x, take a look at the Python 3 tutorial.

It is also possible to write Python code which is compatible with Python 2.7 and 3.x at the same time, using Python `__future__` imports. `__future__` imports allow you to write Python 3 code that will run on Python 2, so check out the Python 3 tutorial.

```
# Single line comments start with a number symbol.

""" Multiline strings can be written
    using three "s, and are often used
    as comments
"""

####################################################
## 1. Primitive Datatypes and Operators
####################################################

# You have numbers
3  # => 3
```

```
# Math is what you would expect
1 + 1  # => 2
8 - 1  # => 7
10 * 2  # => 20
35 / 5  # => 7

# Division is a bit tricky. It is integer division and floors the results
# automatically.
5 / 2  # => 2

# To fix division we need to learn about floats.
2.0     # This is a float
11.0 / 4.0  # => 2.75 ahhh...much better

# Result of integer division truncated down both for positive and negative.
5 // 3     # => 1
5.0 // 3.0 # => 1.0 # works on floats too
-5 // 3  # => -2
-5.0 // 3.0 # => -2.0

# Note that we can also import division module(Section 6 Modules)
# to carry out normal division with just one '/'.
from __future__ import division
11/4    # => 2.75  ...normal division
11//4   # => 2 ...floored division

# Modulo operation
7 % 3 # => 1

# Exponentiation (x to the yth power)
2**4 # => 16

# Enforce precedence with parentheses
(1 + 3) * 2  # => 8

# Boolean Operators
# Note "and" and "or" are case-sensitive
True and False #=> False
False or True #=> True

# Note using Bool operators with ints
0 and 2 #=> 0
-5 or 0 #=> -5
0 == False #=> True
2 == True #=> False
1 == True #=> True

# negate with not
not True  # => False
not False  # => True

# Equality is ==
1 == 1  # => True
2 == 1  # => False
```

```
# Inequality is !=
1 != 1  # => False
2 != 1  # => True

# More comparisons
1 < 10  # => True
1 > 10  # => False
2 <= 2  # => True
2 >= 2  # => True

# Comparisons can be chained!
1 < 2 < 3  # => True
2 < 3 < 2  # => False

# Strings are created with " or '
"This is a string."
'This is also a string.'

# Strings can be added too!
"Hello " + "world!"  # => "Hello world!"
# Strings can be added without using '+'
"Hello " "world!"  # => "Hello world!"

# ... or multiplied
"Hello" * 3  # => "HelloHelloHello"

# A string can be treated like a list of characters
"This is a string"[0]  # => 'T'

#String formatting with %
#Even though the % string operator will be deprecated on Python 3.1 and removed
#later at some time, it may still be good to know how it works.
x = 'apple'
y = 'lemon'
z = "The items in the basket are %s and %s" % (x,y)

# A newer way to format strings is the format method.
# This method is the preferred way
"{} is a {}".format("This", "placeholder")
"{0} can be {1}".format("strings", "formatted")
# You can use keywords if you don't want to count.
"{name} wants to eat {food}".format(name="Bob", food="lasagna")

# None is an object
None  # => None

# Don't use the equality "==" symbol to compare objects to None
# Use "is" instead
"etc" is None  # => False
None is None  # => True

# The 'is' operator tests for object identity. This isn't
# very useful when dealing with primitive values, but is
# very useful when dealing with objects.

# Any object can be used in a Boolean context.
# The following values are considered falsey:
#    - None
#    - zero of any numeric type (e.g., 0, 0L, 0.0, 0j)
#    - empty sequences (e.g., '', (), [])
#    - empty containers (e.g., {}, set())
#    - instances of user-defined classes meeting certain conditions
#      see: https://docs.python.org/2/reference/datamodel.html#object.__nonzero__
#
# All other values are truthy (using the bool() function on them returns True).
bool(0)  # => False
bool("")  # => False


####################################################
## 2. Variables and Collections
####################################################

# Python has a print statement
print "I'm Python. Nice to meet you!" # => I'm Python. Nice to meet you!

# Simple way to get input data from console
input_string_var = raw_input("Enter some data: ") # Returns the data as a string
input_var = input("Enter some data: ") # Evaluates the data as python code
# Warning: Caution is recommended for input() method usage
# Note: In python 3, input() is deprecated and raw_input() is renamed to input()

# No need to declare variables before assigning to them.
some_var = 5    # Convention is to use lower_case_with_underscores
some_var  # => 5

# Accessing a previously unassigned variable is an exception.
# See Control Flow to learn more about exception handling.
some_other_var  # Raises a name error

# if can be used as an expression
# Equivalent of C's '?:' ternary operator
"yahoo!" if 3 > 2 else 2  # => "yahoo!"

# Lists store sequences
li = []
# You can start with a prefilled list
other_li = [4, 5, 6]

# Add stuff to the end of a list with append
li.append(1)    # li is now [1]
li.append(2)    # li is now [1, 2]
li.append(4)    # li is now [1, 2, 4]
li.append(3)    # li is now [1, 2, 4, 3]
# Remove from the end with pop
li.pop()        # => 3 and li is now [1, 2, 4]
# Let's put it back
```

```python
li.append(3)    # li is now [1, 2, 4, 3] again.

# Access a list like you would any array
li[0]  # => 1
# Assign new values to indexes that have already been initialized with =
li[0] = 42
li[0]  # => 42
li[0] = 1  # Note: setting it back to the original value
# Look at the last element
li[-1]  # => 3

# Looking out of bounds is an IndexError
li[4]  # Raises an IndexError

# You can look at ranges with slice syntax.
# (It's a closed/open range for you mathy types.)
li[1:3]  # => [2, 4]
# Omit the beginning
li[2:]  # => [4, 3]
# Omit the end
li[:3]  # => [1, 2, 4]
# Select every second entry
li[::2]   # =>[1, 4]
# Reverse a copy of the list
li[::-1]   # => [3, 4, 2, 1]
# Use any combination of these to make advanced slices
# li[start:end:step]

# Remove arbitrary elements from a list with "del"
del li[2]   # li is now [1, 2, 3]

# You can add lists
li + other_li   # => [1, 2, 3, 4, 5, 6]
# Note: values for li and for other_li are not modified.

# Concatenate lists with "extend()"
li.extend(other_li)   # Now li is [1, 2, 3, 4, 5, 6]

# Remove first occurrence of a value
li.remove(2)  # li is now [1, 3, 4, 5, 6]
li.remove(2)  # Raises a ValueError as 2 is not in the list

# Insert an element at a specific index
li.insert(1, 2)  # li is now [1, 2, 3, 4, 5, 6] again

# Get the index of the first item found
li.index(2)  # => 1
li.index(7)  # Raises a ValueError as 7 is not in the list

# Check for existence in a list with "in"
1 in li   # => True

# Examine the length with "len()"
len(li)   # => 6
```

```
# Tuples are like lists but are immutable.
tup = (1, 2, 3)
tup[0]   # => 1
tup[0] = 3  # Raises a TypeError

# You can do all those list thingies on tuples too
len(tup)   # => 3
tup + (4, 5, 6)   # => (1, 2, 3, 4, 5, 6)
tup[:2]   # => (1, 2)
2 in tup   # => True

# You can unpack tuples (or lists) into variables
a, b, c = (1, 2, 3)     # a is now 1, b is now 2 and c is now 3
d, e, f = 4, 5, 6       # you can leave out the parentheses
# Tuples are created by default if you leave out the parentheses
g = 4, 5, 6             # => (4, 5, 6)
# Now look how easy it is to swap two values
e, d = d, e     # d is now 5 and e is now 4


# Dictionaries store mappings
empty_dict = {}
# Here is a prefilled dictionary
filled_dict = {"one": 1, "two": 2, "three": 3}

# Look up values with []
filled_dict["one"]   # => 1

# Get all keys as a list with "keys()"
filled_dict.keys()   # => ["three", "two", "one"]
# Note - Dictionary key ordering is not guaranteed.
# Your results might not match this exactly.

# Get all values as a list with "values()"
filled_dict.values()   # => [3, 2, 1]
# Note - Same as above regarding key ordering.

# Check for existence of keys in a dictionary with "in"
"one" in filled_dict   # => True
1 in filled_dict   # => False

# Looking up a non-existing key is a KeyError
filled_dict["four"]   # KeyError

# Use "get()" method to avoid the KeyError
filled_dict.get("one")   # => 1
filled_dict.get("four")   # => None
# The get method supports a default argument when the value is missing
filled_dict.get("one", 4)   # => 1
filled_dict.get("four", 4)   # => 4
# note that filled_dict.get("four") is still => None
# (get doesn't set the value in the dictionary)
# set the value of a key with a syntax similar to lists
filled_dict["four"] = 4  # now, filled_dict["four"] => 4

# "setdefault()" inserts into a dictionary only if the given key isn't present
filled_dict.setdefault("five", 5)  # filled_dict["five"] is set to 5
filled_dict.setdefault("five", 6)  # filled_dict["five"] is still 5


# Sets store ... well sets (which are like lists but can contain no duplicates)
empty_set = set()
# Initialize a "set()" with a bunch of values
some_set = set([1, 2, 2, 3, 4])   # some_set is now set([1, 2, 3, 4])

# order is not guaranteed, even though it may sometimes look sorted
another_set = set([4, 3, 2, 2, 1])  # another_set is now set([1, 2, 3, 4])

# Since Python 2.7, {} can be used to declare a set
filled_set = {1, 2, 2, 3, 4}   # => {1, 2, 3, 4}

# Add more items to a set
filled_set.add(5)   # filled_set is now {1, 2, 3, 4, 5}

# Do set intersection with &
other_set = {3, 4, 5, 6}
filled_set & other_set   # => {3, 4, 5}

# Do set union with |
filled_set | other_set   # => {1, 2, 3, 4, 5, 6}

# Do set difference with -
{1, 2, 3, 4} - {2, 3, 5}   # => {1, 4}

# Do set symmetric difference with ^
{1, 2, 3, 4} ^ {2, 3, 5}  # => {1, 4, 5}

# Check if set on the left is a superset of set on the right
{1, 2} >= {1, 2, 3} # => False

# Check if set on the left is a subset of set on the right
{1, 2} <= {1, 2, 3} # => True

# Check for existence in a set with in
2 in filled_set   # => True
10 in filled_set   # => False


####################################################
## 3. Control Flow
####################################################

# Let's just make a variable
some_var = 5
```

```
# Here is an if statement. Indentation is significant in python!
# prints "some_var is smaller than 10"
if some_var > 10:
    print "some_var is totally bigger than 10."
elif some_var < 10:    # This elif clause is optional.
    print "some_var is smaller than 10."
else:           # This is optional too.
    print "some_var is indeed 10."


"""
For loops iterate over lists
prints:
    dog is a mammal
    cat is a mammal
    mouse is a mammal
"""
for animal in ["dog", "cat", "mouse"]:
    # You can use {0} to interpolate formatted strings. (See above.)
    print "{0} is a mammal".format(animal)

"""
"range(number)" returns a list of numbers
from zero to the given number
prints:
    0
    1
    2
    3
"""
for i in range(4):
    print i

"""
"range(lower, upper)" returns a list of numbers
from the lower number to the upper number
prints:
    4
    5
    6
    7
"""
for i in range(4, 8):
    print i

"""
While loops go until a condition is no longer met.
prints:
    0
    1
    2
    3
"""
x = 0
```

```
while x < 4:
    print x
    x += 1  # Shorthand for x = x + 1

# Handle exceptions with a try/except block

# Works on Python 2.6 and up:
try:
    # Use "raise" to raise an error
    raise IndexError("This is an index error")
except IndexError as e:
    pass    # Pass is just a no-op. Usually you would do recovery here.
except (TypeError, NameError):
    pass    # Multiple exceptions can be handled together, if required.
else:   # Optional clause to the try/except block. Must follow all except blocks
    print "All good!"   # Runs only if the code in try raises no exceptions
finally: #  Execute under all circumstances
    print "We can clean up resources here"

# Instead of try/finally to cleanup resources you can use a with statement
with open("myfile.txt") as f:
    for line in f:
        print line

####################################################
## 4. Functions
####################################################

# Use "def" to create new functions
def add(x, y):
    print "x is {0} and y is {1}".format(x, y)
    return x + y    # Return values with a return statement

# Calling functions with parameters
add(5, 6)   # => prints out "x is 5 and y is 6" and returns 11

# Another way to call functions is with keyword arguments
add(y=6, x=5)   # Keyword arguments can arrive in any order.


# You can define functions that take a variable number of
# positional args, which will be interpreted as a tuple by using *
def varargs(*args):
    return args

varargs(1, 2, 3)   # => (1, 2, 3)


# You can define functions that take a variable number of
# keyword args, as well, which will be interpreted as a dict by using **
def keyword_args(**kwargs):
    return kwargs

# Let's call it to see what happens
```

```
keyword_args(big="foot", loch="ness")   # => {"big": "foot", "loch": "ness"}


# You can do both at once, if you like
def all_the_args(*args, **kwargs):
    print args
    print kwargs
"""
all_the_args(1, 2, a=3, b=4) prints:
    (1, 2)
    {"a": 3, "b": 4}
"""

# When calling functions, you can do the opposite of args/kwargs!
# Use * to expand positional args and use ** to expand keyword args.
args = (1, 2, 3, 4)
kwargs = {"a": 3, "b": 4}
all_the_args(*args)   # equivalent to foo(1, 2, 3, 4)
all_the_args(**kwargs)   # equivalent to foo(a=3, b=4)
all_the_args(*args, **kwargs)   # equivalent to foo(1, 2, 3, 4, a=3, b=4)

# you can pass args and kwargs along to other functions that take args/kwargs
# by expanding them with * and ** respectively
def pass_all_the_args(*args, **kwargs):
    all_the_args(*args, **kwargs)
    print varargs(*args)
    print keyword_args(**kwargs)

# Function Scope
x = 5

def set_x(num):
    # Local var x not the same as global variable x
    x = num # => 43
    print x # => 43

def set_global_x(num):
    global x
    print x # => 5
    x = num # global var x is now set to 6
    print x # => 6

set_x(43)
set_global_x(6)

# Python has first class functions
def create_adder(x):
    def adder(y):
        return x + y
    return adder

add_10 = create_adder(10)
add_10(3)   # => 13
```

```
# There are also anonymous functions
(lambda x: x > 2)(3)    # => True
(lambda x, y: x ** 2 + y ** 2)(2, 1) # => 5

# There are built-in higher order functions
map(add_10, [1, 2, 3])   # => [11, 12, 13]
map(max, [1, 2, 3], [4, 2, 1])   # => [4, 2, 3]

filter(lambda x: x > 5, [3, 4, 5, 6, 7])   # => [6, 7]

# We can use list comprehensions for nice maps and filters
[add_10(i) for i in [1, 2, 3]]  # => [11, 12, 13]
[x for x in [3, 4, 5, 6, 7] if x > 5]   # => [6, 7]


####################################################
## 5. Classes
####################################################

# We subclass from object to get a class.
class Human(object):

    # A class attribute. It is shared by all instances of this class
    species = "H. sapiens"

    # Basic initializer, this is called when this class is instantiated.
    # Note that the double leading and trailing underscores denote objects
    # or attributes that are used by python but that live in user-controlled
    # namespaces. You should not invent such names on your own.
    def __init__(self, name):
        # Assign the argument to the instance's name attribute
        self.name = name

        # Initialize property
        self.age = 0


    # An instance method. All methods take "self" as the first argument
    def say(self, msg):
        return "{0}: {1}".format(self.name, msg)

    # A class method is shared among all instances
    # They are called with the calling class as the first argument
    @classmethod
    def get_species(cls):
        return cls.species

    # A static method is called without a class or instance reference
    @staticmethod
    def grunt():
        return "*grunt*"

    # A property is just like a getter.
    # It turns the method age() into an read-only attribute
    # of the same name.
    @property
    def age(self):
        return self._age

    # This allows the property to be set
    @age.setter
    def age(self, age):
        self._age = age

    # This allows the property to be deleted
    @age.deleter
    def age(self):
        del self._age


# Instantiate a class
i = Human(name="Ian")
print i.say("hi")     # prints out "Ian: hi"

j = Human("Joel")
print j.say("hello")  # prints out "Joel: hello"

# Call our class method
i.get_species()   # => "H. sapiens"

# Change the shared attribute
Human.species = "H. neanderthalensis"
i.get_species()   # => "H. neanderthalensis"
j.get_species()   # => "H. neanderthalensis"

# Call the static method
Human.grunt()   # => "*grunt*"

# Update the property
i.age = 42

# Get the property
i.age # => 42

# Delete the property
del i.age
i.age  # => raises an AttributeError


####################################################
## 6. Modules
####################################################

# You can import modules
import math
print math.sqrt(16)  # => 4

# You can get specific functions from a module
```

```
from math import ceil, floor
print ceil(3.7)  # => 4.0
print floor(3.7)   # => 3.0

# You can import all functions from a module.
# Warning: this is not recommended
from math import *

# You can shorten module names
import math as m
math.sqrt(16) == m.sqrt(16)   # => True
# you can also test that the functions are equivalent
from math import sqrt
math.sqrt == m.sqrt == sqrt  # => True

# Python modules are just ordinary python files. You
# can write your own, and import them. The name of the
# module is the same as the name of the file.

# You can find out which functions and attributes
# defines a module.
import math
dir(math)


####################################################
## 7. Advanced
####################################################

# Generators help you make lazy code
def double_numbers(iterable):
    for i in iterable:
        yield i + i

# A generator creates values on the fly.
# Instead of generating and returning all values at once it creates one in each
# iteration.  This means values bigger than 15 wont be processed in
# double_numbers.
# Note xrange is a generator that does the same thing range does.
# Creating a list 1-900000000 would take lot of time and space to be made.
# xrange creates an xrange generator object instead of creating the entire list
# like range does.
# We use a trailing underscore in variable names when we want to use a name that
# would normally collide with a python keyword
xrange_ = xrange(1, 900000000)

# will double all numbers until a result >=30 found
for i in double_numbers(xrange_):
    print i
    if i >= 30:
        break


# Decorators
```

```python
# in this example beg wraps say
# Beg will call say. If say_please is True then it will change the returned
# message
from functools import wraps


def beg(target_function):
    @wraps(target_function)
    def wrapper(*args, **kwargs):
        msg, say_please = target_function(*args, **kwargs)
        if say_please:
            return "{} {}".format(msg, "Please! I am poor :(")
        return msg

    return wrapper


@beg
def say(say_please=False):
    msg = "Can you buy me a beer?"
    return msg, say_please


print say()  # Can you buy me a beer?
print say(say_please=True)  # Can you buy me a beer? Please! I am poor :(
```

Ready For More?

Free Online

- Automate the Boring Stuff with Python
- Learn Python The Hard Way
- Dive Into Python
- The Official Docs
- Hitchhiker's Guide to Python
- Python Module of the Week
- A Crash Course in Python for Scientists
- First Steps With Python
- Fullstack Python

Dead Tree

- Programming Python
- Dive Into Python
- Python Essential Reference

Python3

Python was created by Guido Van Rossum in the early 90s. It is now one of the most popular languages in existence. I fell in love with Python for its syntactic clarity. It's basically executable pseudocode.

Feedback would be highly appreciated! You can reach me at [@louiedinh](http://twitter.com/louiedinh) or louiedinh [at] [google's email service]

Note: This article applies to Python 3 specifically. Check out here if you want to learn the old Python 2.7

```
# Single line comments start with a number symbol.

""" Multiline strings can be written
    using three "s, and are often used
    as comments
"""

####################################################
## 1. Primitive Datatypes and Operators
####################################################

# You have numbers
3  # => 3

# Math is what you would expect
1 + 1   # => 2
8 - 1   # => 7
10 * 2  # => 20

# Except division which returns floats, real numbers, by default
35 / 5  # => 7.0

# Result of integer division truncated down both for positive and negative.
5 // 3       # => 1
5.0 // 3.0   # => 1.0 # works on floats too
-5 // 3      # => -2
-5.0 // 3.0  # => -2.0

# When you use a float, results are floats
3 * 2.0  # => 6.0

# Modulo operation
7 % 3  # => 1

# Exponentiation (x**y, x to the yth power)
2**4  # => 16

# Enforce precedence with parentheses
(1 + 3) * 2  # => 8

# Boolean values are primitives (Note: the capitalization)
True
False

# negate with not
not True   # => False
not False  # => True

# Boolean Operators
# Note "and" and "or" are case-sensitive
True and False  # => False
False or True   # => True

# Note using Bool operators with ints
0 and 2     # => 0
-5 or 0     # => -5
0 == False  # => True
2 == True   # => False
1 == True   # => True

# Equality is ==
1 == 1  # => True
2 == 1  # => False

# Inequality is !=
1 != 1  # => False
2 != 1  # => True

# More comparisons
1 < 10  # => True
1 > 10  # => False
2 <= 2  # => True
2 >= 2  # => True

# Comparisons can be chained!
1 < 2 < 3  # => True
2 < 3 < 2  # => False

# (is vs. ==) is checks if two variables refer to the same object, but == checks
# if the objects pointed to have the same values.
a = [1, 2, 3, 4]  # Point a at a new list, [1, 2, 3, 4]
b = a             # Point b at what a is pointing to
b is a            # => True, a and b refer to the same object
b == a            # => True, a's and b's objects are equal
b = [1, 2, 3, 4]  # Point b at a new list, [1, 2, 3, 4]
b is a            # => False, a and b do not refer to the same object
b == a            # => True, a's and b's objects are equal

# Strings are created with " or '
"This is a string."
'This is also a string.'

# Strings can be added too! But try not to do this.
"Hello " + "world!"  # => "Hello world!"
# Strings can be added without using '+'
"Hello " "world!"    # => "Hello world!"

# A string can be treated like a list of characters
"This is a string"[0]  # => 'T'

# .format can be used to format strings, like this:
"{} can be {}".format("Strings", "interpolated")  # => "Strings can be interpolated"
```

```
# You can repeat the formatting arguments to save some typing.
"{0} be nimble, {0} be quick, {0} jump over the {1}".format("Jack", "candle stick")
# => "Jack be nimble, Jack be quick, Jack jump over the candle stick"

# You can use keywords if you don't want to count.
"{name} wants to eat {food}".format(name="Bob", food="lasagna")  # => "Bob wants to eat lasagna"

# If your Python 3 code also needs to run on Python 2.5 and below, you can also
# still use the old style of formatting:
"%s can be %s the %s way" % ("Strings", "interpolated", "old")  # => "Strings can be interpolated the o


# None is an object
None  # => None

# Don't use the equality "==" symbol to compare objects to None
# Use "is" instead. This checks for equality of object identity.
"etc" is None  # => False
None is None   # => True

# None, 0, and empty strings/lists/dicts all evaluate to False.
# All other values are True
bool(0)   # => False
bool("")  # => False
bool([])  # => False
bool({})  # => False


####################################################
## 2. Variables and Collections
####################################################

# Python has a print function
print("I'm Python. Nice to meet you!")  # => I'm Python. Nice to meet you!

# By default the print function also prints out a newline at the end.
# Use the optional argument end to change the end character.
print("Hello, World", end="!")  # => Hello, World!

# Simple way to get input data from console
input_string_var = input("Enter some data: ") # Returns the data as a string
# Note: In earlier versions of Python, input() method was named as raw_input()

# No need to declare variables before assigning to them.
# Convention is to use lower_case_with_underscores
some_var = 5
some_var  # => 5

# Accessing a previously unassigned variable is an exception.
# See Control Flow to learn more about exception handling.
some_unknown_var  # Raises a NameError

# Lists store sequences
li = []
```

```
# You can start with a prefilled list
other_li = [4, 5, 6]

# Add stuff to the end of a list with append
li.append(1)    # li is now [1]
li.append(2)    # li is now [1, 2]
li.append(4)    # li is now [1, 2, 4]
li.append(3)    # li is now [1, 2, 4, 3]
# Remove from the end with pop
li.pop()        # => 3 and li is now [1, 2, 4]
# Let's put it back
li.append(3)    # li is now [1, 2, 4, 3] again.

# Access a list like you would any array
li[0]   # => 1
# Look at the last element
li[-1]  # => 3

# Looking out of bounds is an IndexError
li[4]  # Raises an IndexError

# You can look at ranges with slice syntax.
# (It's a closed/open range for you mathy types.)
li[1:3]   # => [2, 4]
# Omit the beginning
li[2:]    # => [4, 3]
# Omit the end
li[:3]    # => [1, 2, 4]
# Select every second entry
li[::2]   # =>[1, 4]
# Return a reversed copy of the list
li[::-1]  # => [3, 4, 2, 1]
# Use any combination of these to make advanced slices
# li[start:end:step]

# Make a one layer deep copy using slices
li2 = li[:]  # => li2 = [1, 2, 4, 3] but (li2 is li) will result in false.

# Remove arbitrary elements from a list with "del"
del li[2]  # li is now [1, 2, 3]

# Remove first occurrence of a value
li.remove(2)  # li is now [1, 3]
li.remove(2)  # Raises a ValueError as 2 is not in the list

# Insert an element at a specific index
li.insert(1, 2)  # li is now [1, 2, 3] again

# Get the index of the first item found matching the argument
li.index(2)  # => 1
li.index(4)  # Raises a ValueError as 4 is not in the list

# You can add lists
# Note: values for li and for other_li are not modified.
```

```
li + other_li  # => [1, 2, 3, 4, 5, 6]

# Concatenate lists with "extend()"
li.extend(other_li)  # Now li is [1, 2, 3, 4, 5, 6]

# Check for existence in a list with "in"
1 in li  # => True

# Examine the length with "len()"
len(li)  # => 6


# Tuples are like lists but are immutable.
tup = (1, 2, 3)
tup[0]      # => 1
tup[0] = 3  # Raises a TypeError

# Note that a tuple of length one has to have a comma after the last element but
# tuples of other lengths, even zero, do not.
type((1))   # => <class 'int'>
type((1,))  # => <class 'tuple'>
type(())    # => <class 'tuple'>

# You can do most of the list operations on tuples too
len(tup)         # => 3
tup + (4, 5, 6)  # => (1, 2, 3, 4, 5, 6)
tup[:2]          # => (1, 2)
2 in tup         # => True

# You can unpack tuples (or lists) into variables
a, b, c = (1, 2, 3)  # a is now 1, b is now 2 and c is now 3
# You can also do extended unpacking
a, *b, c = (1, 2, 3, 4)  # a is now 1, b is now [2, 3] and c is now 4
# Tuples are created by default if you leave out the parentheses
d, e, f = 4, 5, 6
# Now look how easy it is to swap two values
e, d = d, e  # d is now 5 and e is now 4


# Dictionaries store mappings
empty_dict = {}
# Here is a prefilled dictionary
filled_dict = {"one": 1, "two": 2, "three": 3}

# Note keys for dictionaries have to be immutable types. This is to ensure that
# the key can be converted to a constant hash value for quick look-ups.
# Immutable types include ints, floats, strings, tuples.
invalid_dict = {[1,2,3]: "123"}  # => Raises a TypeError: unhashable type: 'list'
valid_dict = {(1,2,3):[1,2,3]}   # Values can be of any type, however.

# Look up values with []
filled_dict["one"]  # => 1

# Get all keys as an iterable with "keys()". We need to wrap the call in list()
# to turn it into a list. We'll talk about those later.  Note - Dictionary key
# ordering is not guaranteed. Your results might not match this exactly.
list(filled_dict.keys())  # => ["three", "two", "one"]


# Get all values as an iterable with "values()". Once again we need to wrap it
# in list() to get it out of the iterable. Note - Same as above regarding key
# ordering.
list(filled_dict.values())  # => [3, 2, 1]


# Check for existence of keys in a dictionary with "in"
"one" in filled_dict  # => True
1 in filled_dict      # => False

# Looking up a non-existing key is a KeyError
filled_dict["four"]  # KeyError

# Use "get()" method to avoid the KeyError
filled_dict.get("one")      # => 1
filled_dict.get("four")     # => None
# The get method supports a default argument when the value is missing
filled_dict.get("one", 4)   # => 1
filled_dict.get("four", 4)  # => 4

# "setdefault()" inserts into a dictionary only if the given key isn't present
filled_dict.setdefault("five", 5)  # filled_dict["five"] is set to 5
filled_dict.setdefault("five", 6)  # filled_dict["five"] is still 5

# Adding to a dictionary
filled_dict.update({"four":4})  # => {"one": 1, "two": 2, "three": 3, "four": 4}
#filled_dict["four"] = 4        #another way to add to dict

# Remove keys from a dictionary with del
del filled_dict["one"]  # Removes the key "one" from filled dict

# From Python 3.5 you can also use the additional unpacking options
{'a': 1, **{'b': 2}}  # => {'a': 1, 'b': 2}
{'a': 1, **{'a': 2}}  # => {'a': 2}



# Sets store ... well sets
empty_set = set()
# Initialize a set with a bunch of values. Yeah, it looks a bit like a dict. Sorry.
some_set = {1, 1, 2, 2, 3, 4}  # some_set is now {1, 2, 3, 4}

# Similar to keys of a dictionary, elements of a set have to be immutable.
invalid_set = {[1], 1}  # => Raises a TypeError: unhashable type: 'list'
valid_set = {(1,), 1}

# Can set new variables to a set
filled_set = some_set
```

```
# Add one more item to the set
filled_set.add(5)  # filled_set is now {1, 2, 3, 4, 5}

# Do set intersection with &
other_set = {3, 4, 5, 6}
filled_set & other_set  # => {3, 4, 5}

# Do set union with |
filled_set | other_set  # => {1, 2, 3, 4, 5, 6}

# Do set difference with -
{1, 2, 3, 4} - {2, 3, 5}  # => {1, 4}

# Do set symmetric difference with ^
{1, 2, 3, 4} ^ {2, 3, 5}  # => {1, 4, 5}

# Check if set on the left is a superset of set on the right
{1, 2} >= {1, 2, 3} # => False

# Check if set on the left is a subset of set on the right
{1, 2} <= {1, 2, 3} # => True

# Check for existence in a set with in
2 in filled_set   # => True
10 in filled_set  # => False


####################################################
## 3. Control Flow and Iterables
####################################################

# Let's just make a variable
some_var = 5

# Here is an if statement. Indentation is significant in python!
# prints "some_var is smaller than 10"
if some_var > 10:
    print("some_var is totally bigger than 10.")
elif some_var < 10:    # This elif clause is optional.
    print("some_var is smaller than 10.")
else:                  # This is optional too.
    print("some_var is indeed 10.")


"""
For loops iterate over lists
prints:
    dog is a mammal
    cat is a mammal
    mouse is a mammal
"""
for animal in ["dog", "cat", "mouse"]:
    # You can use format() to interpolate formatted strings
    print("{} is a mammal".format(animal))

"""
"range(number)" returns an iterable of numbers
from zero to the given number
prints:
    0
    1
    2
    3
"""
for i in range(4):
    print(i)

"""
"range(lower, upper)" returns an iterable of numbers
from the lower number to the upper number
prints:
    4
    5
    6
    7
"""
for i in range(4, 8):
    print(i)

"""
"range(lower, upper, step)" returns an iterable of numbers
from the lower number to the upper number, while incrementing
by step. If step is not indicated, the default value is 1.
prints:
    4
    6
"""
for i in range(4, 8, 2):
    print(i)
"""

While loops go until a condition is no longer met.
prints:
    0
    1
    2
    3
"""
x = 0
while x < 4:
    print(x)
    x += 1  # Shorthand for x = x + 1

# Handle exceptions with a try/except block
try:
    # Use "raise" to raise an error
    raise IndexError("This is an index error")
```

```
except IndexError as e:
    pass                 # Pass is just a no-op. Usually you would do recovery here.
except (TypeError, NameError):
    pass                 # Multiple exceptions can be handled together, if required.
else:                    # Optional clause to the try/except block. Must follow all except blocks
    print("All good!")   # Runs only if the code in try raises no exceptions
finally:                 #  Execute under all circumstances
    print("We can clean up resources here")

# Instead of try/finally to cleanup resources you can use a with statement
with open("myfile.txt") as f:
    for line in f:
        print(line)

# Python offers a fundamental abstraction called the Iterable.
# An iterable is an object that can be treated as a sequence.
# The object returned the range function, is an iterable.

filled_dict = {"one": 1, "two": 2, "three": 3}
our_iterable = filled_dict.keys()
print(our_iterable)  # => dict_keys(['one', 'two', 'three']). This is an object that implements our Ite

# We can loop over it.
for i in our_iterable:
    print(i)  # Prints one, two, three

# However we cannot address elements by index.
our_iterable[1]  # Raises a TypeError

# An iterable is an object that knows how to create an iterator.
our_iterator = iter(our_iterable)

# Our iterator is an object that can remember the state as we traverse through it.
# We get the next object with "next()".
next(our_iterator)  # => "one"

# It maintains state as we iterate.
next(our_iterator)  # => "two"
next(our_iterator)  # => "three"

# After the iterator has returned all of its data, it gives you a StopIterator Exception
next(our_iterator)  # Raises StopIteration

# You can grab all the elements of an iterator by calling list() on it.
list(filled_dict.keys())  # => Returns ["one", "two", "three"]


####################################################
## 4. Functions
####################################################

# Use "def" to create new functions
def add(x, y):
    print("x is {} and y is {}".format(x, y))
    return x + y  # Return values with a return statement

# Calling functions with parameters
add(5, 6)  # => prints out "x is 5 and y is 6" and returns 11

# Another way to call functions is with keyword arguments
add(y=6, x=5)  # Keyword arguments can arrive in any order.

# You can define functions that take a variable number of
# positional arguments
def varargs(*args):
    return args

varargs(1, 2, 3)  # => (1, 2, 3)

# You can define functions that take a variable number of
# keyword arguments, as well
def keyword_args(**kwargs):
    return kwargs

# Let's call it to see what happens
keyword_args(big="foot", loch="ness")  # => {"big": "foot", "loch": "ness"}


# You can do both at once, if you like
def all_the_args(*args, **kwargs):
    print(args)
    print(kwargs)
"""
all_the_args(1, 2, a=3, b=4) prints:
    (1, 2)
    {"a": 3, "b": 4}
"""

# When calling functions, you can do the opposite of args/kwargs!
# Use * to expand tuples and use ** to expand kwargs.
args = (1, 2, 3, 4)
kwargs = {"a": 3, "b": 4}
all_the_args(*args)            # equivalent to foo(1, 2, 3, 4)
all_the_args(**kwargs)         # equivalent to foo(a=3, b=4)
all_the_args(*args, **kwargs)  # equivalent to foo(1, 2, 3, 4, a=3, b=4)

# Returning multiple values (with tuple assignments)
def swap(x, y):
    return y, x  # Return multiple values as a tuple without the parenthesis.
                 # (Note: parenthesis have been excluded but can be included)

x = 1
y = 2
x, y = swap(x, y)     # => x = 2, y = 1
# (x, y) = swap(x,y)  # Again parenthesis have been excluded but can be included.

# Function Scope
x = 5
```

```
def set_x(num):
    # Local var x not the same as global variable x
    x = num     # => 43
    print (x)   # => 43

def set_global_x(num):
    global x
    print (x)   # => 5
    x = num     # global var x is now set to 6
    print (x)   # => 6

set_x(43)
set_global_x(6)


# Python has first class functions
def create_adder(x):
    def adder(y):
        return x + y
    return adder

add_10 = create_adder(10)
add_10(3)   # => 13

# There are also anonymous functions
(lambda x: x > 2)(3)                  # => True
(lambda x, y: x ** 2 + y ** 2)(2, 1)  # => 5

# TODO - Fix for iterables
# There are built-in higher order functions
map(add_10, [1, 2, 3])          # => [11, 12, 13]
map(max, [1, 2, 3], [4, 2, 1])  # => [4, 2, 3]

filter(lambda x: x > 5, [3, 4, 5, 6, 7])  # => [6, 7]

# We can use list comprehensions for nice maps and filters
# List comprehension stores the output as a list which can itself be a nested list
[add_10(i) for i in [1, 2, 3]]         # => [11, 12, 13]
[x for x in [3, 4, 5, 6, 7] if x > 5]  # => [6, 7]

####################################################
## 5. Classes
####################################################


# We use the "class" operator to get a class
class Human:

    # A class attribute. It is shared by all instances of this class
    species = "H. sapiens"

    # Basic initializer, this is called when this class is instantiated.
    # Note that the double leading and trailing underscores denote objects
    # or attributes that are used by python but that live in user-controlled
    # namespaces. Methods(or objects or attributes) like: __init__, __str__,
    # __repr__ etc. are called magic methods (or sometimes called dunder methods)
    # You should not invent such names on your own.
    def __init__(self, name):
        # Assign the argument to the instance's name attribute
        self.name = name

        # Initialize property
        self.age = 0

    # An instance method. All methods take "self" as the first argument
    def say(self, msg):
        return "{name}: {message}".format(name=self.name, message=msg)

    # A class method is shared among all instances
    # They are called with the calling class as the first argument
    @classmethod
    def get_species(cls):
        return cls.species

    # A static method is called without a class or instance reference
    @staticmethod
    def grunt():
        return "*grunt*"

    # A property is just like a getter.
    # It turns the method age() into an read-only attribute
    # of the same name.
    @property
    def age(self):
        return self._age

    # This allows the property to be set
    @age.setter
    def age(self, age):
        self._age = age

    # This allows the property to be deleted
    @age.deleter
    def age(self):
        del self._age


# Instantiate a class
i = Human(name="Ian")
print(i.say("hi"))     # prints out "Ian: hi"

j = Human("Joel")
print(j.say("hello"))  # prints out "Joel: hello"

# Call our class method
i.get_species()   # => "H. sapiens"
```

```python
# Change the shared attribute
Human.species = "H. neanderthalensis"
i.get_species()  # => "H. neanderthalensis"
j.get_species()  # => "H. neanderthalensis"

# Call the static method
Human.grunt()    # => "*grunt*"

# Update the property
i.age = 42

# Get the property
i.age # => 42

# Delete the property
del i.age
i.age  # => raises an AttributeError



####################################################
## 6. Modules
####################################################

# You can import modules
import math
print(math.sqrt(16))  # => 4.0

# You can get specific functions from a module
from math import ceil, floor
print(ceil(3.7))   # => 4.0
print(floor(3.7))  # => 3.0

# You can import all functions from a module.
# Warning: this is not recommended
from math import *

# You can shorten module names
import math as m
math.sqrt(16) == m.sqrt(16)  # => True

# Python modules are just ordinary python files. You
# can write your own, and import them. The name of the
# module is the same as the name of the file.

# You can find out which functions and attributes
# defines a module.
import math
dir(math)


####################################################
## 7. Advanced
####################################################
```

```python
# Generators help you make lazy code
def double_numbers(iterable):
    for i in iterable:
        yield i + i

# A generator creates values on the fly.
# Instead of generating and returning all values at once it creates one in each
# iteration.  This means values bigger than 15 wont be processed in
# double_numbers.
# We use a trailing underscore in variable names when we want to use a name that
# would normally collide with a python keyword
range_ = range(1, 900000000)
# will double all numbers until a result >=30 found
for i in double_numbers(range_):
    print(i)
    if i >= 30:
        break


# Decorators
# in this example beg wraps say
# Beg will call say. If say_please is True then it will change the returned
# message
from functools import wraps


def beg(target_function):
    @wraps(target_function)
    def wrapper(*args, **kwargs):
        msg, say_please = target_function(*args, **kwargs)
        if say_please:
            return "{} {}".format(msg, "Please! I am poor :(")
        return msg

    return wrapper


@beg
def say(say_please=False):
    msg = "Can you buy me a beer?"
    return msg, say_please


print(say())                 # Can you buy me a beer?
print(say(say_please=True))  # Can you buy me a beer? Please! I am poor :(
```

Ready For More?

Free Online

- Automate the Boring Stuff with Python
- Learn Python The Hard Way

- Dive Into Python
- Ideas for Python Projects
- The Official Docs
- Hitchhiker's Guide to Python
- A Crash Course in Python for Scientists
- Python Course
- First Steps With Python
- A curated list of awesome Python frameworks, libraries and software
- 30 Python Language Features and Tricks You May Not Know About
- Official Style Guide for Python

Dead Tree

- Programming Python
- Dive Into Python
- Python Essential Reference

Pythonstatcomp

This is a tutorial on how to do some typical statistical programming tasks using Python. It's intended for people basically familiar with Python and experienced at statistical programming in a language like R, Stata, SAS, SPSS, or MATLAB.

```
# 0. Getting set up ====

""" Get set up with IPython and pip install the following: numpy, scipy, pandas,
    matplotlib, seaborn, requests.
        Make sure to do this tutorial in the IPython notebook so that you get
    the inline plots and easy documentation lookup.
"""

# 1. Data acquisition ====

""" One reason people choose Python over R is that they intend to interact a lot
    with the web, either by scraping pages directly or requesting data through
    an API. You can do those things in R, but in the context of a project
    already using Python, there's a benefit to sticking with one language.
"""

import requests  # for HTTP requests (web scraping, APIs)
import os

# web scraping
r = requests.get("https://github.com/adambard/learnxinyminutes-docs")
r.status_code  # if 200, request was successful
r.text  # raw page source
print(r.text)  # prettily formatted
# save the page source in a file:
os.getcwd()  # check what's the working directory
f = open("learnxinyminutes.html", "wb")
```

```
f.write(r.text.encode("UTF-8"))
f.close()

# downloading a csv
fp = "https://raw.githubusercontent.com/adambard/learnxinyminutes-docs/master/"
fn = "pets.csv"
r = requests.get(fp + fn)
print(r.text)
f = open(fn, "wb")
f.write(r.text.encode("UTF-8"))
f.close()

""" for more on the requests module, including APIs, see
    http://docs.python-requests.org/en/latest/user/quickstart/
"""

# 2. Reading a CSV file ====

""" Wes McKinney's pandas package gives you 'DataFrame' objects in Python. If
    you've used R, you will be familiar with the idea of the "data.frame" already.
"""

import pandas as pd
import numpy as np
import scipy as sp
pets = pd.read_csv(fn)
pets
#        name  age  weight species
# 0    fluffy    3      14     cat
# 1  vesuvius    6      23    fish
# 2       rex    5      34     dog

""" R users: note that Python, like most normal programming languages, starts
    indexing from 0. R is the unusual one for starting from 1.
"""

# two different ways to print out a column
pets.age
pets["age"]

pets.head(2)  # prints first 2 rows
pets.tail(1)  # prints last row

pets.name[1]  # 'vesuvius'
pets.species[0]  # 'cat'
pets["weight"][2]  # 34

# in R, you would expect to get 3 rows doing this, but here you get 2:
pets.age[0:2]
# 0    3
# 1    6

sum(pets.age) * 2  # 28
max(pets.weight) - min(pets.weight)  # 20
```

```
""" If you are doing some serious linear algebra and number-crunching, you may
    just want arrays, not DataFrames. DataFrames are ideal for combining columns
    of different types.
"""

# 3. Charts ====

import matplotlib as mpl
import matplotlib.pyplot as plt
%matplotlib inline

# To do data vizualization in Python, use matplotlib

plt.hist(pets.age);

plt.boxplot(pets.weight);

plt.scatter(pets.age, pets.weight)
plt.xlabel("age")
plt.ylabel("weight");

# seaborn sits atop matplotlib and makes plots prettier

import seaborn as sns

plt.scatter(pets.age, pets.weight)
plt.xlabel("age")
plt.ylabel("weight");

# there are also some seaborn-specific plotting functions
# notice how seaborn automatically labels the x-axis on this barplot
sns.barplot(pets["age"])

# R veterans can still use ggplot
from ggplot import *
ggplot(aes(x="age",y="weight"), data=pets) + geom_point() + labs(title="pets")
# source: https://pypi.python.org/pypi/ggplot

# there's even a d3.js port: https://github.com/mikedewar/d3py

# 4. Simple data cleaning and exploratory analysis ====

""" Here's a more complicated example that demonstrates a basic data
    cleaning workflow leading to the creation of some exploratory plots
    and the running of a linear regression.
        The data set was transcribed from Wikipedia by hand. It contains
    all the Holy Roman Emperors and the important milestones in their lives
    (birth, death, coronation, etc.).
        The goal of the analysis will be to explore whether a relationship
    exists between emperor birth year and emperor lifespan.
    data source: https://en.wikipedia.org/wiki/Holy_Roman_Emperor
"""
```

```
# load some data on Holy Roman Emperors
url = "https://raw.githubusercontent.com/e99n09/R-notes/master/data/hre.csv"
r = requests.get(url)
fp = "hre.csv"
f = open(fp, "wb")
f.write(r.text.encode("UTF-8"))
f.close()

hre = pd.read_csv(fp)

hre.head()
"""
   Ix      Dynasty         Name        Birth             Death Election 1
0 NaN  Carolingian   Charles I  2 April 742    28 January 814        NaN
1 NaN  Carolingian     Louis I         778       20 June 840        NaN
2 NaN  Carolingian   Lothair I         795  29 September 855        NaN
3 NaN  Carolingian    Louis II         825     12 August 875        NaN
4 NaN  Carolingian  Charles II  13 June 823     6 October 877        NaN

  Election 2     Coronation 1   Coronation 2 Ceased to be Emperor
0        NaN  25 December 800            NaN       28 January 814
1        NaN 11 September 813  5 October 816          20 June 840
2        NaN      5 April 823            NaN     29 September 855
3        NaN       Easter 850     18 May 872        12 August 875
4        NaN  29 December 875            NaN        6 October 877

  Descent from whom 1 Descent how 1 Descent from whom 2 Descent how 2
0                 NaN           NaN                 NaN           NaN
1           Charles I           son                 NaN           NaN
2             Louis I           son                 NaN           NaN
3           Lothair I           son                 NaN           NaN
4             Louis I           son                 NaN           NaN
"""

# clean the Birth and Death columns

import re  # module for regular expressions

rx = re.compile(r'\d+$')  # match trailing digits

""" This function applies the regular expression to an input column (here Birth,
    Death), flattens the resulting list, converts it to a Series object, and
    finally converts the type of the Series object from string to integer. For
    more information into what different parts of the code do, see:
      - https://docs.python.org/2/howto/regex.html
      - http://stackoverflow.com/questions/11860476/how-to-unlist-a-python-list
      - http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html
"""

def extractYear(v):
    return(pd.Series(reduce(lambda x, y: x + y, map(rx.findall, v), [])).astype(int))

hre["BirthY"] = extractYear(hre.Birth)
hre["DeathY"] = extractYear(hre.Death)
```

```
# make a column telling estimated age
hre["EstAge"] = hre.DeathY.astype(int) - hre.BirthY.astype(int)

# simple scatterplot, no trend line, color represents dynasty
sns.lmplot("BirthY", "EstAge", data=hre, hue="Dynasty", fit_reg=False);

# use scipy to run a linear regression
from scipy import stats
(slope, intercept, rval, pval, stderr) = stats.linregress(hre.BirthY, hre.EstAge)
# code source: http://wiki.scipy.org/Cookbook/LinearRegression

# check the slope
slope  # 0.0057672618839073328

# check the R^2 value:
rval**2  # 0.020363950027333586

# check the p-value
pval  # 0.34971812581498452

# use seaborn to make a scatterplot and plot the linear regression trend line
sns.lmplot("BirthY", "EstAge", data=hre);

""" For more information on seaborn, see
      - http://web.stanford.edu/~mwaskom/software/seaborn/
      - https://github.com/mwaskom/seaborn
    For more information on SciPy, see
      - http://wiki.scipy.org/SciPy
      - http://wiki.scipy.org/Cookbook/
    To see a version of the Holy Roman Emperors analysis using R, see
      - http://github.com/e99n09/R-notes/blob/master/holy_roman_emperors_dates.R
"""
```

If you want to learn more, get *Python for Data Analysis* by Wes McKinney. It's a superb resource and I used it as a reference when writing this tutorial.

You can also find plenty of interactive IPython tutorials on subjects specific to your interests, like Cam Davidson-Pilon's Probabilistic Programming and Bayesian Methods for Hackers.

Some more modules to research: - text analysis and natural language processing: nltk, http://www.nltk.org - social network analysis: igraph, http://igraph.org/python/

R

R is a statistical computing language. It has lots of libraries for uploading and cleaning data sets, running statistical procedures, and making graphs. You can also run `R` commands within a LaTeX document.

```
# Comments start with number symbols.

# You can't make multi-line comments,
# but you can stack multiple comments like so.

# in Windows you can use CTRL-ENTER to execute a line.
# on Mac it is COMMAND-ENTER



#############################################################################
# Stuff you can do without understanding anything about programming
#############################################################################

# In this section, we show off some of the cool stuff you can do in
# R without understanding anything about programming. Do not worry
# about understanding everything the code does. Just enjoy!

data()           # browse pre-loaded data sets
data(rivers)     # get this one: "Lengths of Major North American Rivers"
ls()             # notice that "rivers" now appears in the workspace
head(rivers)     # peek at the data set
# 735 320 325 392 524 450

length(rivers)  # how many rivers were measured?
# 141
summary(rivers) # what are some summary statistics?
#   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#  135.0   310.0   425.0   591.2   680.0  3710.0

# make a stem-and-leaf plot (a histogram-like data visualization)
stem(rivers)

#  The decimal point is 2 digit(s) to the right of the |
#
#   0 | 4
#   2 | 011223334555566667778888899900001111223333344455555666688888999
#   4 | 111222333445566779001233344567
#   6 | 000112233578012234468
#   8 | 045790018
#  10 | 04507
#  12 | 1471
#  14 | 56
#  16 | 7
#  18 | 9
#  20 |
#  22 | 25
#  24 | 3
#  26 |
#  28 |
#  30 |
#  32 |
#  34 |
#  36 | 1

stem(log(rivers)) # Notice that the data are neither normal nor log-normal!
# Take that, Bell curve fundamentalists.

#  The decimal point is 1 digit(s) to the left of the |
#
```

```
#   48 | 1
#   50 |
#   52 | 15578
#   54 | 44571222466689
#   56 | 023334677000124455789
#   58 | 00122366666999933445777
#   60 | 122445567800133459
#   62 | 112666799035
#   64 | 00011334581257889
#   66 | 003683579
#   68 | 0019156
#   70 | 079357
#   72 | 89
#   74 | 84
#   76 | 56
#   78 | 4
#   80 |
#   82 | 2

# make a histogram:
hist(rivers, col="#333333", border="white", breaks=25) # play around with these parameters
hist(log(rivers), col="#333333", border="white", breaks=25) # you'll do more plotting later

# Here's another neat data set that comes pre-loaded. R has tons of these.
data(discoveries)
plot(discoveries, col="#333333", lwd=3, xlab="Year",
     main="Number of important discoveries per year")
plot(discoveries, col="#333333", lwd=3, type = "h", xlab="Year",
     main="Number of important discoveries per year")

# Rather than leaving the default ordering (by year),
# we could also sort to see what's typical:
sort(discoveries)
#  [1]  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  1  1  2  2  2  2
# [26]  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  3  3  3
# [51]  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  4  4  4  4  4  4  4  4
# [76]  4  4  4  4  5  5  5  5  5  5  5  6  6  6  6  6  6  7  7  7  7  8  9 10 12

stem(discoveries, scale=2)
#
#  The decimal point is at the |
#
#    0 | 000000000
#    1 | 000000000000
#    2 | 00000000000000000000000000
#    3 | 00000000000000000000
#    4 | 000000000000
#    5 | 0000000
#    6 | 000000
#    7 | 0000
#    8 | 0
#    9 | 0
#   10 | 0
#   11 |
#  12 | 0

max(discoveries)
# 12
summary(discoveries)
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#     0.0     2.0     3.0     3.1     4.0    12.0

# Roll a die a few times
round(runif(7, min=.5, max=6.5))
# 1 4 6 1 4 6 4
# Your numbers will differ from mine unless we set the same random.seed(31337)

# Draw from a standard Gaussian 9 times
rnorm(9)
# [1]  0.07528471  1.03499859  1.34809556 -0.82356087  0.61638975 -1.88757271
# [7] -0.59975593  0.57629164  1.08455362



##################################################
# Data types and basic arithmetic
##################################################

# Now for the programming-oriented part of the tutorial.
# In this section you will meet the important data types of R:
# integers, numerics, characters, logicals, and factors.
# There are others, but these are the bare minimum you need to
# get started.

# INTEGERS
# Long-storage integers are written with L
5L # 5
class(5L) # "integer"
# (Try ?class for more information on the class() function.)
# In R, every single value, like 5L, is considered a vector of length 1
length(5L) # 1
# You can have an integer vector with length > 1 too:
c(4L, 5L, 8L, 3L) # 4 5 8 3
length(c(4L, 5L, 8L, 3L)) # 4
class(c(4L, 5L, 8L, 3L)) # "integer"

# NUMERICS
# A "numeric" is a double-precision floating-point number
5 # 5
class(5) # "numeric"
# Again, everything in R is a vector;
# you can make a numeric vector with more than one element
c(3,3,3,2,2,1) # 3 3 3 2 2 1
# You can use scientific notation too
5e4 # 50000
6.02e23 # Avogadro's number
1.6e-35 # Planck length
# You can also have infinitely large or small numbers
```

```
class(Inf)  # "numeric"
class(-Inf) # "numeric"
# You might use "Inf", for example, in integrate(dnorm, 3, Inf);
# this obviates Z-score tables.

# BASIC ARITHMETIC
# You can do arithmetic with numbers
# Doing arithmetic on a mix of integers and numerics gives you another numeric
10L + 66L # 76      # integer plus integer gives integer
53.2 - 4  # 49.2    # numeric minus numeric gives numeric
2.0 * 2L  # 4       # numeric times integer gives numeric
3L / 4    # 0.75    # integer over numeric gives numeric
3 %% 2    # 1       # the remainder of two numerics is another numeric
# Illegal arithmetic yeilds you a "not-a-number":
0 / 0 # NaN
class(NaN) # "numeric"
# You can do arithmetic on two vectors with length greater than 1,
# so long as the larger vector's length is an integer multiple of the smaller
c(1,2,3) + c(1,2,3) # 2 4 6
# Since a single number is a vector of length one, scalars are applied
# elementwise to vectors
(4 * c(1,2,3) - 2) / 2 # 1 3 5
# Except for scalars, use caution when performing arithmetic on vectors with
# different lengths. Although it can be done,
c(1,2,3,1,2,3) * c(1,2) # 1 4 3 2 2 6
# Matching lengths is better practice and easier to read
c(1,2,3,1,2,3) * c(1,2,1,2,1,2)

# CHARACTERS
# There's no difference between strings and characters in R
"Horatio" # "Horatio"
class("Horatio") # "character"
class('H') # "character"
# Those were both character vectors of length 1
# Here is a longer one:
c('alef', 'bet', 'gimmel', 'dalet', 'he')
# =>
# "alef"   "bet"    "gimmel" "dalet"  "he"
length(c("Call","me","Ishmael")) # 3
# You can do regex operations on character vectors:
substr("Fortuna multis dat nimis, nulli satis.", 9, 15) # "multis "
gsub('u', 'ø', "Fortuna multis dat nimis, nulli satis.") # "Fortøna møltis dat nimis, nølli satis."
# R has several built-in character vectors:
letters
# =>
#  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
# [20] "t" "u" "v" "w" "x" "y" "z"
month.abb # "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"

# LOGICALS
# In R, a "logical" is a boolean
class(TRUE) # "logical"
class(FALSE)    # "logical"
# Their behavior is normal
```

```
TRUE == TRUE    # TRUE
TRUE == FALSE   # FALSE
FALSE != FALSE  # FALSE
FALSE != TRUE   # TRUE
# Missing data (NA) is logical, too
class(NA)   # "logical"
# Use | and & for logic operations.
# OR
TRUE | FALSE    # TRUE
# AND
TRUE & FALSE    # FALSE
# Applying | and & to vectors returns elementwise logic operations
c(TRUE,FALSE,FALSE) | c(FALSE,TRUE,FALSE) # TRUE TRUE FALSE
c(TRUE,FALSE,TRUE) & c(FALSE,TRUE,TRUE) # FALSE FALSE TRUE
# You can test if x is TRUE
isTRUE(TRUE)    # TRUE
# Here we get a logical vector with many elements:
c('Z', 'o', 'r', 'r', 'o') == "Zorro" # FALSE FALSE FALSE FALSE FALSE
c('Z', 'o', 'r', 'r', 'o') == "Z" # TRUE FALSE FALSE FALSE FALSE

# FACTORS
# The factor class is for categorical data
# Factors can be ordered (like childrens' grade levels) or unordered (like gender)
factor(c("female", "female", "male", NA, "female"))
#  female female male   <NA>   female
# Levels: female male
# The "levels" are the values the categorical data can take
# Note that missing data does not enter the levels
levels(factor(c("male", "male", "female", NA, "female"))) # "female" "male"
# If a factor vector has length 1, its levels will have length 1, too
length(factor("male")) # 1
length(levels(factor("male"))) # 1
# Factors are commonly seen in data frames, a data structure we will cover later
data(infert) # "Infertility after Spontaneous and Induced Abortion"
levels(infert$education) # "0-5yrs"  "6-11yrs" "12+ yrs"

# NULL
# "NULL" is a weird one; use it to "blank out" a vector
class(NULL) # NULL
parakeet = c("beak", "feathers", "wings", "eyes")
parakeet
# =>
# [1] "beak"     "feathers" "wings"    "eyes"
parakeet <- NULL
parakeet
# =>
# NULL

# TYPE COERCION
# Type-coercion is when you force a value to take on a different type
as.character(c(6, 8)) # "6" "8"
as.logical(c(1,0,1,1)) # TRUE FALSE  TRUE  TRUE
# If you put elements of different types into a vector, weird coercions happen:
c(TRUE, 4) # 1 4
```

```r
c("dog", TRUE, 4) # "dog"  "TRUE" "4"
as.numeric("Bilbo")
# =>
# [1] NA
# Warning message:
# NAs introduced by coercion

# Also note: those were just the basic data types
# There are many more data types, such as for dates, time series, etc.



###################################################
# Variables, loops, if/else
###################################################

# A variable is like a box you store a value in for later use.
# We call this "assigning" the value to the variable.
# Having variables lets us write loops, functions, and if/else statements

# VARIABLES
# Lots of way to assign stuff:
x = 5 # this is possible
y <- "1" # this is preferred
TRUE -> z # this works but is weird

# LOOPS
# We've got for loops
for (i in 1:4) {
  print(i)
}
# We've got while loops
a <- 10
while (a > 4) {
    cat(a, "...", sep = "")
    a <- a - 1
}
# Keep in mind that for and while loops run slowly in R
# Operations on entire vectors (i.e. a whole row, a whole column)
# or apply()-type functions (we'll discuss later) are preferred

# IF/ELSE
# Again, pretty standard
if (4 > 3) {
    print("4 is greater than 3")
} else {
    print("4 is not greater than 3")
}
# =>
# [1] "4 is greater than 3"

# FUNCTIONS
# Defined like so:
jiggle <- function(x) {
    x = x + rnorm(1, sd=.1) #add in a bit of (controlled) noise
    return(x)
}
# Called like any other R function:
jiggle(5)   # 5± . After set.seed(2716057), jiggle(5)==5.005043



###########################################################################
# Data structures: Vectors, matrices, data frames, and arrays
###########################################################################

# ONE-DIMENSIONAL

# Let's start from the very beginning, and with something you already know: vectors.
vec <- c(8, 9, 10, 11)
vec #  8  9 10 11
# We ask for specific elements by subsetting with square brackets
# (Note that R starts counting from 1)
vec[1]      # 8
letters[18] # "r"
LETTERS[13] # "M"
month.name[9]   # "September"
c(6, 8, 7, 5, 3, 0, 9)[3]   # 7
# We can also search for the indices of specific components,
which(vec %% 2 == 0)    # 1 3
# grab just the first or last few entries in the vector,
head(vec, 1)    # 8
tail(vec, 2)    # 10 11
# or figure out if a certain value is in the vector
any(vec == 10) # TRUE
# If an index "goes over" you'll get NA:
vec[6]  # NA
# You can find the length of your vector with length()
length(vec) # 4
# You can perform operations on entire vectors or subsets of vectors
vec * 4 # 16 20 24 28
vec[2:3] * 5    # 25 30
any(vec[2:3] == 8) # FALSE
# and R has many built-in functions to summarize vectors
mean(vec)   # 9.5
var(vec)    # 1.666667
sd(vec)     # 1.290994
max(vec)    # 11
min(vec)    # 8
sum(vec)    # 38
# Some more nice built-ins:
5:15    # 5  6  7  8  9 10 11 12 13 14 15
seq(from=0, to=31337, by=1337)
# =>
#  [1]     0  1337  2674  4011  5348  6685  8022  9359 10696 12033 13370 14707
# [13] 16044 17381 18718 20055 21392 22729 24066 25403 26740 28077 29414 30751

# TWO-DIMENSIONAL (ALL ONE CLASS)
# You can make a matrix out of entries all of the same type like so:
mat <- matrix(nrow = 3, ncol = 2, c(1,2,3,4,5,6))
mat
# =>
#      [,1] [,2]
# [1,]    1    4
# [2,]    2    5
# [3,]    3    6
# Unlike a vector, the class of a matrix is "matrix", no matter what's in it
class(mat) # => "matrix"
# Ask for the first row
mat[1,] # 1 4
# Perform operation on the first column
3 * mat[,1] # 3 6 9
# Ask for a specific cell
mat[3,2]    # 6

# Transpose the whole matrix
t(mat)
# =>
#      [,1] [,2] [,3]
# [1,]    1    2    3
# [2,]    4    5    6

# Matrix multiplication
mat %*% t(mat)
# =>
#      [,1] [,2] [,3]
# [1,]   17   22   27
# [2,]   22   29   36
# [3,]   27   36   45

# cbind() sticks vectors together column-wise to make a matrix
mat2 <- cbind(1:4, c("dog", "cat", "bird", "dog"))
mat2
# =>
#      [,1] [,2]
# [1,] "1"  "dog"
# [2,] "2"  "cat"
# [3,] "3"  "bird"
# [4,] "4"  "dog"
class(mat2) # matrix
# Again, note what happened!
# Because matrices must contain entries all of the same class,
# everything got converted to the character class
c(class(mat2[,1]), class(mat2[,2]))

# rbind() sticks vectors together row-wise to make a matrix
mat3 <- rbind(c(1,2,4,5), c(6,7,0,4))
mat3
# =>
#      [,1] [,2] [,3] [,4]
# [1,]    1    2    4    5
# [2,]    6    7    0    4
# Ah, everything of the same class. No coercions. Much better.

# TWO-DIMENSIONAL (DIFFERENT CLASSES)

# For columns of different types, use a data frame
# This data structure is so useful for statistical programming,
# a version of it was added to Python in the package "pandas".

students <- data.frame(c("Cedric","Fred","George","Cho","Draco","Ginny"),
                       c(3,2,2,1,0,-1),
                       c("H", "G", "G", "R", "S", "G"))
names(students) <- c("name", "year", "house") # name the columns
class(students) # "data.frame"
students
# =>
#     name year house
# 1 Cedric    3     H
# 2   Fred    2     G
# 3 George    2     G
# 4    Cho    1     R
# 5  Draco    0     S
# 6  Ginny   -1     G
class(students$year)    # "numeric"
class(students[,3]) # "factor"
# find the dimensions
nrow(students)  # 6
ncol(students)  # 3
dim(students)   # 6 3
# The data.frame() function converts character vectors to factor vectors
# by default; turn this off by setting stringsAsFactors = FALSE when
# you create the data.frame
?data.frame

# There are many twisty ways to subset data frames, all subtly unalike
students$year   # 3  2  2  1  0 -1
students[,2]    # 3  2  2  1  0 -1
students[,"year"]   # 3  2  2  1  0 -1

# An augmented version of the data.frame structure is the data.table
# If you're working with huge or panel data, or need to merge a few data
# sets, data.table can be a good choice. Here's a whirlwind tour:
install.packages("data.table") # download the package from CRAN
require(data.table) # load it
students <- as.data.table(students)
students # note the slightly different print-out
# =>
#      name year house
# 1: Cedric    3     H
# 2:   Fred    2     G
# 3: George    2     G
# 4:    Cho    1     R
# 5:  Draco    0     S
# 6:  Ginny   -1     G
```

```
students[name=="Ginny"] # get rows with name == "Ginny"
# =>
#     name year house
# 1: Ginny   -1     G
students[year==2] # get rows with year == 2
# =>
#      name year house
# 1:   Fred    2     G
# 2: George    2     G
# data.table makes merging two data sets easy
# let's make another data.table to merge with students
founders <- data.table(house=c("G","H","R","S"),
                       founder=c("Godric","Helga","Rowena","Salazar"))
founders
# =>
#    house founder
# 1:     G  Godric
# 2:     H   Helga
# 3:     R  Rowena
# 4:     S Salazar
setkey(students, house)
setkey(founders, house)
students <- founders[students] # merge the two data sets by matching "house"
setnames(students, c("house","houseFounderName","studentName","year"))
students[,order(c("name","year","house","houseFounderName")), with=F]
# =>
#    studentName year house houseFounderName
# 1:        Fred    2     G           Godric
# 2:      George    2     G           Godric
# 3:       Ginny   -1     G           Godric
# 4:      Cedric    3     H            Helga
# 5:         Cho    1     R           Rowena
# 6:       Draco    0     S          Salazar

# data.table makes summary tables easy
students[,sum(year),by=house]
# =>
#    house V1
# 1:     G  3
# 2:     H  3
# 3:     R  1
# 4:     S  0

# To drop a column from a data.frame or data.table,
# assign it the NULL value
students$houseFounderName <- NULL
students
# =>
#    studentName year house
# 1:        Fred    2     G
# 2:      George    2     G
# 3:       Ginny   -1     G
# 4:      Cedric    3     H
# 5:         Cho    1     R
# 6:       Draco    0     S

# Drop a row by subsetting
# Using data.table:
students[studentName != "Draco"]
# =>
#    house studentName year
# 1:     G        Fred    2
# 2:     G      George    2
# 3:     G       Ginny   -1
# 4:     H      Cedric    3
# 5:     R         Cho    1
# Using data.frame:
students <- as.data.frame(students)
students[students$house != "G",]
# =>
#   house houseFounderName studentName year
# 4     H            Helga      Cedric    3
# 5     R           Rowena         Cho    1
# 6     S          Salazar       Draco    0

# MULTI-DIMENSIONAL (ALL ELEMENTS OF ONE TYPE)

# Arrays creates n-dimensional tables
# All elements must be of the same type
# You can make a two-dimensional table (sort of like a matrix)
array(c(c(1,2,4,5),c(8,9,3,6)), dim=c(2,4))
# =>
#      [,1] [,2] [,3] [,4]
# [1,]    1    4    8    3
# [2,]    2    5    9    6
# You can use array to make three-dimensional matrices too
array(c(c(c(2,300,4),c(8,9,0)),c(c(5,60,0),c(66,7,847))), dim=c(3,2,2))
# =>
# , , 1
#
#      [,1] [,2]
# [1,]    2    8
# [2,]  300    9
# [3,]    4    0
#
# , , 2
#
#      [,1] [,2]
# [1,]    5   66
# [2,]   60    7
# [3,]    0  847

# LISTS (MULTI-DIMENSIONAL, POSSIBLY RAGGED, OF DIFFERENT TYPES)

# Finally, R has lists (of vectors)
list1 <- list(time = 1:40)
list1$price = c(rnorm(40,.5*list1$time,4)) # random
list1
```

```r
# You can get items in the list like so
list1$time # one way
list1[["time"]] # another way
list1[[1]] # yet another way
# =>
#  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
# [34] 34 35 36 37 38 39 40
# You can subset list items like any other vector
list1$price[4]

# Lists are not the most efficient data structure to work with in R;
# unless you have a very good reason, you should stick to data.frames
# Lists are often returned by functions that perform linear regressions

##################################################
# The apply() family of functions
##################################################

# Remember mat?
mat
# =>
#      [,1] [,2]
# [1,]    1    4
# [2,]    2    5
# [3,]    3    6
# Use apply(X, MARGIN, FUN) to apply function FUN to a matrix X
# over rows (MAR = 1) or columns (MAR = 2)
# That is, R does FUN to each row (or column) of X, much faster than a
# for or while loop would do
apply(mat, MAR = 2, jiggle)
# =>
#      [,1] [,2]
# [1,]    3   15
# [2,]    7   19
# [3,]   11   23
# Other functions: ?lapply, ?sapply

# Don't feel too intimidated; everyone agrees they are rather confusing

# The plyr package aims to replace (and improve upon!) the *apply() family.
install.packages("plyr")
require(plyr)
?plyr



#########################
# Loading data
#########################

# "pets.csv" is a file on the internet
# (but it could just as easily be be a file on your own computer)
pets <- read.csv("http://learnxinyminutes.com/docs/pets.csv")
pets
```

```
head(pets, 2) # first two rows
tail(pets, 1) # last row

# To save a data frame or matrix as a .csv file
write.csv(pets, "pets2.csv") # to make a new .csv file
# set working directory with setwd(), look it up with getwd()

# Try ?read.csv and ?write.csv for more information



#########################
# Statistical Analysis
#########################

# Linear regression!
linearModel <- lm(price  ~ time, data = list1)
linearModel # outputs result of regression
# =>
# Call:
# lm(formula = price ~ time, data = list1)
#
# Coefficients:
# (Intercept)         time
#      0.1453       0.4943
summary(linearModel) # more verbose output from the regression
# =>
# Call:
# lm(formula = price ~ time, data = list1)
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -8.3134 -3.0131 -0.3606  2.8016 10.3992
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)  0.14527    1.50084   0.097    0.923
# time         0.49435    0.06379   7.749 2.44e-09 ***
# ---
# Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
#
# Residual standard error: 4.657 on 38 degrees of freedom
# Multiple R-squared:  0.6124,  Adjusted R-squared:  0.6022
# F-statistic: 60.05 on 1 and 38 DF,  p-value: 2.44e-09
coef(linearModel) # extract estimated parameters
# =>
# (Intercept)        time
#   0.1452662   0.4943490
summary(linearModel)$coefficients # another way to extract results
# =>
#              Estimate Std. Error    t value     Pr(>|t|)
# (Intercept) 0.1452662 1.50084246 0.09678975 9.234021e-01
# time        0.4943490 0.06379348 7.74920901 2.440008e-09
summary(linearModel)$coefficients[,4] # the p-values
```

```
# =>
#  (Intercept)         time
# 9.234021e-01 2.440008e-09

# GENERAL LINEAR MODELS
# Logistic regression
set.seed(1)
list1$success = rbinom(length(list1$time), 1, .5) # random binary
glModel <- glm(success  ~ time, data = list1,
    family=binomial(link="logit"))
glModel # outputs result of logistic regression
# =>
# Call:  glm(formula = success ~ time,
#   family = binomial(link = "logit"), data = list1)
#
# Coefficients:
# (Intercept)         time
#     0.17018     -0.01321
#
# Degrees of Freedom: 39 Total (i.e. Null);  38 Residual
# Null Deviance:        55.35
# Residual Deviance: 55.12   AIC: 59.12
summary(glModel) # more verbose output from the regression
# =>
# Call:
# glm(formula = success ~ time,
#   family = binomial(link = "logit"), data = list1)

# Deviance Residuals:
#    Min      1Q  Median      3Q     Max
# -1.245  -1.118  -1.035   1.202   1.327
#
# Coefficients:
#             Estimate Std. Error z value Pr(>|z|)
# (Intercept)  0.17018    0.64621   0.263    0.792
# time        -0.01321    0.02757  -0.479    0.632
#
# (Dispersion parameter for binomial family taken to be 1)
#
#     Null deviance: 55.352  on 39  degrees of freedom
# Residual deviance: 55.121  on 38  degrees of freedom
# AIC: 59.121
#
# Number of Fisher Scoring iterations: 3


#########################
# Plots
#########################

# BUILT-IN PLOTTING FUNCTIONS
# Scatterplots!
plot(list1$time, list1$price, main = "fake data")
# Plot regression line on existing plot
```

```
abline(linearModel, col = "red")
# Get a variety of nice diagnostics
plot(linearModel)
# Histograms!
hist(rpois(n = 10000, lambda = 5), col = "thistle")
# Barplots!
barplot(c(1,4,5,1,2), names.arg = c("red","blue","purple","green","yellow"))

# GGPLOT2
# But these are not even the prettiest of R's plots
# Try the ggplot2 package for more and better graphics
install.packages("ggplot2")
require(ggplot2)
?ggplot2
pp <- ggplot(students, aes(x=house))
pp + geom_histogram()
ll <- as.data.table(list1)
pp <- ggplot(ll, aes(x=time,price))
pp + geom_point()
# ggplot2 has excellent documentation (available http://docs.ggplot2.org/current/)
```

How do I get R?

- Get R and the R GUI from http://www.r-project.org/
- RStudio is another GUI

Racket

Racket is a general purpose, multi-paradigm programming language in the Lisp/Scheme family.

Feedback is appreciated! You can reach me at [@th3rac25](http://twitter.com/th3rac25) or th3rac25 [at] [google's email service]

```
#lang racket ; defines the language we are using

;;; Comments

;; Single line comments start with a semicolon

#| Block comments
   can span multiple lines and...
    #|
       they can be nested!
    |#
|#

;; S-expression comments discard the following expression,
;; useful to comment expressions when debugging
#; (this expression is discarded)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; 1. Primitive Datatypes and Operators
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Numbers
9999999999999999999999 ; integers
#b111                  ; binary => 7
#o111                  ; octal => 73
#x111                  ; hexadecimal => 273
3.14                   ; reals
6.02e+23
1/2                    ; rationals
1+2i                   ; complex numbers

;; Function application is written (f x y z ...)
;; where f is a function and x, y, z, ... are operands
;; If you want to create a literal list of data, use ' to stop it from
;; being evaluated
'(+ 1 2) ; => (+ 1 2)
;; Now, some arithmetic operations
(+ 1 1)  ; => 2
(- 8 1)  ; => 7
(* 10 2) ; => 20
(expt 2 3) ; => 8
(quotient 5 2) ; => 2
(remainder 5 2) ; => 1
(/ 35 5) ; => 7
(/ 1 3) ; => 1/3
(exact->inexact 1/3) ; => 0.3333333333333333
(+ 1+2i  2-3i) ; => 3-1i

;;; Booleans
#t ; for true
#f ; for false -- any value other than #f is true
(not #t) ; => #f
(and 0 #f (error "doesn't get here")) ; => #f
(or #f 0 (error "doesn't get here"))  ; => 0

;;; Characters
#\A ; => #\A
#\  ; => #\
#\u03BB ; => #\

;;; Strings are fixed-length array of characters.
"Hello, world!"
"Benjamin \"Bugsy\" Siegel"   ; backslash is an escaping character
"Foo\tbar\41\x21\u0021\a\r\n" ; includes C escapes, Unicode
" x:( . → ).xx"          ; can include Unicode characters

;; Strings can be added too!
(string-append "Hello " "world!") ; => "Hello world!"

;; A string can be treated like a list of characters
(string-ref "Apple" 0) ; => #\A
```

```
;; format can be used to format strings:
(format "~a can be ~a" "strings" "formatted")

;; Printing is pretty easy
(printf "I'm Racket. Nice to meet you!\n")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 2. Variables
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; You can create a variable using define
;; a variable name can use any character except: ()[]{}",'`;#|\
(define some-var 5)
some-var ; => 5

;; You can also use unicode characters
(define   subset?)
(  (set 3 2) (set 1 2 3)) ; => #t

;; Accessing a previously unassigned variable is an exception
; x ; => x: undefined ...

;; Local binding: `me' is bound to "Bob" only within the (let ...)
(let ([me "Bob"])
  "Alice"
  me) ; => "Bob"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 3. Structs and Collections
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Structs
(struct dog (name breed age))
(define my-pet
  (dog "lassie" "collie" 5))
my-pet ; => #<dog>
(dog? my-pet) ; => #t
(dog-name my-pet) ; => "lassie"

;;; Pairs (immutable)
;; `cons' constructs pairs, `car' and `cdr' extract the first
;; and second elements
(cons 1 2) ; => '(1 . 2)
(car (cons 1 2)) ; => 1
(cdr (cons 1 2)) ; => 2

;;; Lists

;; Lists are linked-list data structures, made of `cons' pairs and end
;; with a `null' (or '()) to mark the end of the list
(cons 1 (cons 2 (cons 3 null))) ; => '(1 2 3)
;; `list' is a convenience variadic constructor for lists
(list 1 2 3) ; => '(1 2 3)
;; and a quote can also be used for a literal list value
'(1 2 3) ; => '(1 2 3)
```

```
;; Can still use `cons' to add an item to the beginning of a list
(cons 4 '(1 2 3)) ; => '(4 1 2 3)

;; Use `append' to add lists together
(append '(1 2) '(3 4)) ; => '(1 2 3 4)

;; Lists are a very basic type, so there is a *lot* of functionality for
;; them, a few examples:
(map add1 '(1 2 3))          ; => '(2 3 4)
(map + '(1 2 3) '(10 20 30)) ; => '(11 22 33)
(filter even? '(1 2 3 4))    ; => '(2 4)
(count even? '(1 2 3 4))     ; => 2
(take '(1 2 3 4) 2)          ; => '(1 2)
(drop '(1 2 3 4) 2)          ; => '(3 4)

;;; Vectors

;; Vectors are fixed-length arrays
#(1 2 3) ; => '#(1 2 3)

;; Use `vector-append' to add vectors together
(vector-append #(1 2 3) #(4 5 6)) ; => #(1 2 3 4 5 6)

;;; Sets

;; Create a set from a list
(list->set '(1 2 3 1 2 3 3 2 1 3 2 1)) ; => (set 1 2 3)

;; Add a member with `set-add'
;; (Functional: returns the extended set rather than mutate the input)
(set-add (set 1 2 3) 4) ; => (set 1 2 3 4)

;; Remove one with `set-remove'
(set-remove (set 1 2 3) 1) ; => (set 2 3)

;; Test for existence with `set-member?'
(set-member? (set 1 2 3) 1) ; => #t
(set-member? (set 1 2 3) 4) ; => #f

;;; Hashes

;; Create an immutable hash table (mutable example below)
(define m (hash 'a 1 'b 2 'c 3))

;; Retrieve a value
(hash-ref m 'a) ; => 1

;; Retrieving a non-present value is an exception
; (hash-ref m 'd) => no value found

;; You can provide a default value for missing keys
(hash-ref m 'd 0) ; => 0
```

```
;; Use `hash-set' to extend an immutable hash table
;; (Returns the extended hash instead of mutating it)
(define m2 (hash-set m 'd 4))
m2 ; => '#hash((b . 2) (a . 1) (d . 4) (c . 3))

;; Remember, these hashes are immutable!
m ; => '#hash((b . 2) (a . 1) (c . 3))  <-- no `d'

;; Use `hash-remove' to remove keys (functional too)
(hash-remove m 'a) ; => '#hash((b . 2) (c . 3))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 3. Functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Use `lambda' to create functions.
;; A function always returns the value of its last expression
(lambda () "Hello World") ; => #<procedure>
;; Can also use a unicode ` '
(  () "Hello World")     ; => same function

;; Use parens to call all functions, including a lambda expression
((lambda () "Hello World")) ; => "Hello World"
((  () "Hello World"))      ; => "Hello World"

;; Assign a function to a var
(define hello-world (lambda () "Hello World"))
(hello-world) ; => "Hello World"

;; You can shorten this using the function definition syntactic sugar:
(define (hello-world2) "Hello World")

;; The () in the above is the list of arguments for the function
(define hello
  (lambda (name)
    (string-append "Hello " name)))
(hello "Steve") ; => "Hello Steve"
;; ... or equivalently, using a sugared definition:
(define (hello2 name)
  (string-append "Hello " name))

;; You can have multi-variadic functions too, using `case-lambda'
(define hello3
  (case-lambda
    [() "Hello World"]
    [(name) (string-append "Hello " name)]))
(hello3 "Jake") ; => "Hello Jake"
(hello3) ; => "Hello World"
;; ... or specify optional arguments with a default value expression
(define (hello4 [name "World"])
  (string-append "Hello " name))

;; Functions can pack extra arguments up in a list
(define (count-args . args)
  (format "You passed ~a args: ~a" (length args) args))
(count-args 1 2 3) ; => "You passed 3 args: (1 2 3)"
;; ... or with the unsugared `lambda' form:
(define count-args2
  (lambda args
    (format "You passed ~a args: ~a" (length args) args)))

;; You can mix regular and packed arguments
(define (hello-count name . args)
  (format "Hello ~a, you passed ~a extra args" name (length args)))
(hello-count "Finn" 1 2 3)
; => "Hello Finn, you passed 3 extra args"
;; ... unsugared:
(define hello-count2
  (lambda (name . args)
    (format "Hello ~a, you passed ~a extra args" name (length args))))

;; And with keywords
(define (hello-k #:name [name "World"] #:greeting [g "Hello"] . args)
  (format "~a ~a, ~a extra args" g name (length args)))
(hello-k)                 ; => "Hello World, 0 extra args"
(hello-k 1 2 3)           ; => "Hello World, 3 extra args"
(hello-k #:greeting "Hi") ; => "Hi World, 0 extra args"
(hello-k #:name "Finn" #:greeting "Hey") ; => "Hey Finn, 0 extra args"
(hello-k 1 2 3 #:greeting "Hi" #:name "Finn" 4 5 6)
                                         ; => "Hi Finn, 6 extra args"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 4. Equality
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; for numbers use `='
(= 3 3.0) ; => #t
(= 2 1)   ; => #f

;; `eq?' returns #t if 2 arguments refer to the same object (in memory),
;; #f otherwise.
;; In other words, it's a simple pointer comparison.
(eq? '() '()) ; => #t, since there exists only one empty list in memory
(let ([x '()] [y '()])
  (eq? x y))  ; => #t, same as above

(eq? (list 3) (list 3)) ; => #f
(let ([x (list 3)] [y (list 3)])
  (eq? x y))            ; => #f - not the same list in memory!

(let* ([x (list 3)] [y x])
  (eq? x y)) ; => #t, since x and y now point to the same stuff

(eq? 'yes 'yes) ; => #t
(eq? 'yes 'no)  ; => #f

(eq? 3 3)   ; => #t - be careful here
            ; It’s better to use `=' for number comparisons.
```

```
(eq? 3 3.0) ; => #f

(eq? (expt 2 100) (expt 2 100))               ; => #f
(eq? (integer->char 955) (integer->char 955)) ; => #f

(eq? (string-append "foo" "bar") (string-append "foo" "bar")) ; => #f

;; `eqv?' supports the comparison of number and character datatypes.
;; for other datatypes, `eqv?' and `eq?' return the same result.
(eqv? 3 3.0)                                   ; => #f
(eqv? (expt 2 100) (expt 2 100))               ; => #t
(eqv? (integer->char 955) (integer->char 955)) ; => #t

(eqv? (string-append "foo" "bar") (string-append "foo" "bar"))   ; => #f

;; `equal?' supports the comparison of the following datatypes:
;; strings, byte strings, pairs, mutable pairs, vectors, boxes,
;; hash tables, and inspectable structures.
;; for other datatypes, `equal?' and `eqv?' return the same result.
(equal? 3 3.0)                                                   ; => #f
(equal? (string-append "foo" "bar") (string-append "foo" "bar")) ; => #t
(equal? (list 3) (list 3))                                       ; => #t

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 5. Control Flow
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Conditionals

(if #t               ; test expression
    "this is true"   ; then expression
    "this is false") ; else expression
; => "this is true"

;; In conditionals, all non-#f values are treated as true
(member 'Groucho '(Harpo Groucho Zeppo)) ; => '(Groucho Zeppo)
(if (member 'Groucho '(Harpo Groucho Zeppo))
    'yep
    'nope)
; => 'yep

;; `cond' chains a series of tests to select a result
(cond [(> 2 2) (error "wrong!")]
      [(< 2 2) (error "wrong again!")]
      [else 'ok]) ; => 'ok

;;; Pattern Matching

(define (fizzbuzz? n)
  (match (list (remainder n 3) (remainder n 5))
    [(list 0 0) 'fizzbuzz]
    [(list 0 _) 'fizz]
    [(list _ 0) 'buzz]
    [_          #f]))
```

```
(fizzbuzz? 15) ; => 'fizzbuzz
(fizzbuzz? 37) ; => #f

;;; Loops

;; Looping can be done through (tail-) recursion
(define (loop i)
  (when (< i 10)
    (printf "i=~a\n" i)
    (loop (add1 i))))
(loop 5) ; => i=5, i=6, ...

;; Similarly, with a named let
(let loop ((i 0))
  (when (< i 10)
    (printf "i=~a\n" i)
    (loop (add1 i)))) ; => i=0, i=1, ...

;; See below how to add a new `loop' form, but Racket already has a very
;; flexible `for' form for loops:
(for ([i 10])
  (printf "i=~a\n" i)) ; => i=0, i=1, ...
(for ([i (in-range 5 10)])
  (printf "i=~a\n" i)) ; => i=5, i=6, ...

;;; Iteration Over Other Sequences
;; `for' allows iteration over many other kinds of sequences:
;; lists, vectors, strings, sets, hash tables, etc...

(for ([i (in-list '(l i s t))])
  (displayln i))

(for ([i (in-vector #(v e c t o r))])
  (displayln i))

(for ([i (in-string "string")])
  (displayln i))

(for ([i (in-set (set 'x 'y 'z))])
  (displayln i))

(for ([(k v) (in-hash (hash 'a 1 'b 2 'c 3 ))])
  (printf "key:~a value:~a\n" k v))

;;; More Complex Iterations

;; Parallel scan of multiple sequences (stops on shortest)
(for ([i 10] [j '(x y z)]) (printf "~a:~a\n" i j))
; => 0:x 1:y 2:z

;; Nested loops
(for* ([i 2] [j '(x y z)]) (printf "~a:~a\n" i j))
; => 0:x, 0:y, 0:z, 1:x, 1:y, 1:z
```

```
;; Conditions
(for ([i 1000]
      #:when (> i 5)
      #:unless (odd? i)
      #:break (> i 10))
  (printf "i=~a\n" i))
; => i=6, i=8, i=10

;;; Comprehensions
;; Very similar to `for' loops -- just collect the results

(for/list ([i '(1 2 3)])
  (add1 i)) ; => '(2 3 4)

(for/list ([i '(1 2 3)] #:when (even? i))
  i) ; => '(2)

(for/list ([i 10] [j '(x y z)])
  (list i j)) ; => '((0 x) (1 y) (2 z))

(for/list ([i 1000] #:when (> i 5) #:unless (odd? i) #:break (> i 10))
  i) ; => '(6 8 10)

(for/hash ([i '(1 2 3)])
  (values i (number->string i)))
; => '#hash((1 . "1") (2 . "2") (3 . "3"))

;; There are many kinds of other built-in ways to collect loop values:
(for/sum ([i 10]) (* i i)) ; => 285
(for/product ([i (in-range 1 11)]) (* i i)) ; => 13168189440000
(for/and ([i 10] [j (in-range 10 20)]) (< i j)) ; => #t
(for/or ([i 10] [j (in-range 0 20 2)]) (= i j)) ; => #t
;; And to use any arbitrary combination, use `for/fold'
(for/fold ([sum 0]) ([i '(1 2 3 4)]) (+ sum i)) ; => 10
;; (This can often replace common imperative loops)

;;; Exceptions

;; To catch exceptions, use the `with-handlers' form
(with-handlers ([exn:fail? (lambda (exn) 999)])
  (+ 1 "2")) ; => 999
(with-handlers ([exn:break? (lambda (exn) "no time")])
  (sleep 3)
  "phew") ; => "phew", but if you break it => "no time"

;; Use `raise' to throw exceptions or any other value
(with-handlers ([number?    ; catch numeric values raised
                 identity]) ; return them as plain values
  (+ 1 (raise 2))) ; => 2

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 6. Mutation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; Use `set!' to assign a new value to an existing variable
(define n 5)
(set! n (add1 n))
n ; => 6

;; Use boxes for explicitly mutable values (similar to pointers or
;; references in other languages)
(define n* (box 5))
(set-box! n* (add1 (unbox n*)))
(unbox n*) ; => 6

;; Many Racket datatypes are immutable (pairs, lists, etc), some come in
;; both mutable and immutable flavors (strings, vectors, hash tables,
;; etc...)

;; Use `vector' or `make-vector' to create mutable vectors
(define vec (vector 2 2 3 4))
(define wall (make-vector 100 'bottle-of-beer))
;; Use vector-set! to update a slot
(vector-set! vec 0 1)
(vector-set! wall 99 'down)
vec ; => #(1 2 3 4)

;; Create an empty mutable hash table and manipulate it
(define m3 (make-hash))
(hash-set! m3 'a 1)
(hash-set! m3 'b 2)
(hash-set! m3 'c 3)
(hash-ref m3 'a)   ; => 1
(hash-ref m3 'd 0) ; => 0
(hash-remove! m3 'a)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 7. Modules
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Modules let you organize code into multiple files and reusable
;; libraries; here we use sub-modules, nested in the whole module that
;; this text makes (starting from the "#lang" line)

(module cake racket/base ; define a `cake' module based on racket/base

  (provide print-cake) ; function exported by the module

  (define (print-cake n)
    (show "   ~a   " n #\.)
    (show " .-~a-. " n #\|)
    (show " | ~a | " n #\space)
    (show "---~a---" n #\-))

  (define (show fmt n ch) ; internal function
    (printf fmt (make-string n ch))
    (newline)))
```

```
;; Use `require' to get all `provide'd names from a module
(require 'cake) ; the ' is for a local submodule
(print-cake 3)
; (show "~a" 1 #\A) ; => error, `show' was not exported

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 8. Classes and Objects
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Create a class fish% (-% is idiomatic for class bindings)
(define fish%
  (class object%
    (init size) ; initialization argument
    (super-new) ; superclass initialization
    ;; Field
    (define current-size size)
    ;; Public methods
    (define/public (get-size)
      current-size)
    (define/public (grow amt)
      (set! current-size (+ amt current-size)))
    (define/public (eat other-fish)
      (grow (send other-fish get-size)))))

;; Create an instance of fish%
(define charlie
  (new fish% [size 10]))

;; Use `send' to call an object's methods
(send charlie get-size) ; => 10
(send charlie grow 6)
(send charlie get-size) ; => 16

;; `fish%' is a plain "first class" value, which can get us mixins
(define (add-color c%)
  (class c%
    (init color)
    (super-new)
    (define my-color color)
    (define/public (get-color) my-color)))
(define colored-fish% (add-color fish%))
(define charlie2 (new colored-fish% [size 10] [color 'red]))
(send charlie2 get-color)
;; or, with no names:
(send (new (add-color fish%) [size 10] [color 'red]) get-color)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 9. Macros
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Macros let you extend the syntax of the language

;; Let's add a while loop
```

```
(define-syntax-rule (while condition body ...)
  (let loop ()
    (when condition
      body ...
      (loop))))

(let ([i 0])
  (while (< i  10)
    (displayln i)
    (set! i (add1 i))))

;; Macros are hygienic, you cannot clobber existing variables!
(define-syntax-rule (swap! x y) ; -! is idiomatic for mutation
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))

(define tmp 2)
(define other 3)
(swap! tmp other)
(printf "tmp = ~a; other = ~a\n" tmp other)
;; The variable `tmp` is renamed to `tmp_1`
;; in order to avoid name conflict
;; (let ([tmp_1 tmp])
;;   (set! tmp other)
;;   (set! other tmp_1))

;; But they are still code transformations, for example:
(define-syntax-rule (bad-while condition body ...)
  (when condition
    body ...
    (bad-while condition body ...)))
;; this macro is broken: it generates infinite code, if you try to use
;; it, the compiler will get in an infinite loop

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 10. Contracts
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Contracts impose constraints on values exported from modules

(module bank-account racket
  (provide (contract-out
            [deposit (-> positive? any)] ; amounts are always positive
            [balance (-> positive?)]))

  (define amount 0)
  (define (deposit a) (set! amount (+ amount a)))
  (define (balance) amount)
  )

(require 'bank-account)
(deposit 5)
```

```
(balance) ; => 5

;; Clients that attempt to deposit a non-positive amount are blamed
;; (deposit -5) ; => deposit: contract violation
;; expected: positive?
;; given: -5
;; more details....

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 11. Input & output
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Racket has this concept of "port", which is very similar to file
;; descriptors in other languages

;; Open "/tmp/tmp.txt" and write "Hello World"
;; This would trigger an error if the file's already existed
(define out-port (open-output-file "/tmp/tmp.txt"))
(displayln "Hello World" out-port)
(close-output-port out-port)

;; Append to "/tmp/tmp.txt"
(define out-port (open-output-file "/tmp/tmp.txt"
                                   #:exists 'append))
(displayln "Hola mundo" out-port)
(close-output-port out-port)

;; Read from the file again
(define in-port (open-input-file "/tmp/tmp.txt"))
(displayln (read-line in-port))
; => "Hello World"
(displayln (read-line in-port))
; => "Hola mundo"
(close-input-port in-port)

;; Alternatively, with call-with-output-file you don't need to explicitly
;; close the file
(call-with-output-file "/tmp/tmp.txt"
  #:exists 'update ; Rewrite the content
  (  (out-port)
    (displayln "World Hello!" out-port)))

;; And call-with-input-file does the same thing for input
(call-with-input-file "/tmp/tmp.txt"
  (  (in-port)
    (displayln (read-line in-port))))
```

Further Reading

Still up for more? Try Getting Started with Racket

Red

Red was created out of the need to get work done, and the tool the author wanted to use, the language of REBOL, had a couple of drawbacks. It was not Open Sourced at that time and it is an interpreted language, what means that it is on average slow compared to a compiled language.

Red, together with its C-level dialect Red/System, provides a language that covers the entire programming space you ever need to program something in. Red is a language heavily based on the language of REBOL. Where Red itself reproduces the flexibility of the REBOL language, the underlying language Red will be built upon, Red/System, covers the more basic needs of programming like C can, being closer to the metal.

Red will be the world's first Full Stack Programming Language. This means that it will be an effective tool to do (almost) any programming task on every level from the metal to the meta without the aid of other stack tools. Furthermore Red will be able to cross-compile Red source code without using any GCC like toolchain from any platform to any other platform. And it will do this all from a binary executable that is supposed to stay under 1 MB.

Ready to learn your first Red?

```
All text before the header will be treated as comment, as long as you avoid using the
word "red" starting with a capital "R" in this pre-header text. This is a temporary
shortcoming of the used lexer but most of the time you start your script or program
with the header itself.
The header of a red script is the capitalized word "red" followed by a
whitespace character followed by a block of square brackets [].
The block of brackets can be filled with useful information about this script or
program: the author's name, the filename, the version, the license, a summary of
what the program does or any other files it needs.
The red/System header is just like the red header, only saying "red/System" and
not "red".

Red []

;this is a commented line

print "Hello Red World"    ; this is another comment

comment {
    This is a multiline comment.
    You just saw the Red version of the "Hello World" program.
}

; Your program's entry point is the first executable code that is found
; no need to restrict this to a 'main' function.

; Valid variable names start with a letter and can contain numbers,
; variables containing only capital A thru F and numbers and ending with 'h' are
; forbidden, because that is how hexadecimal numbers are expressed in Red and
; Red/System.

; assign a value to a variable using a colon ":"
my-name: "Red"
reason-for-using-the-colon: {Assigning values using the colon makes
 the equality sign "=" exclusively usable for comparisons purposes,
 exactly what "=" was intended for in the first place!
 Remember this y = x + 1 and x = 1 => y = 2 stuff from school?
}
is-this-name-valid?: true

; print output using print, or prin for printing without a newline or linefeed at the
; end of the printed text.

prin " My name is " print my-name
My name is Red

print ["My name is " my-name lf]
My name is Red

; In case you haven't already noticed: statements do NOT end with a semicolon ;-)

;
; Datatypes
;
; If you know Rebol, you probably have noticed it has lots of datatypes. Red
; does not have yet all those types, but as Red want to be close to Rebol it
; will have a lot of datatypes.
; You can recognize types by the exclamation sign at the end. But beware
; names ending with an exclamation sign are allowed.
; Some of the available types are integer! string! block!

; Declaring variables before using them?
; Red knows by itself what variable is best to use for the data you want to use it
; for.
; A variable declaration is not always necessary.
; It is considered good coding practise to declare your variables,
; but it is not forced upon you by Red.
; You can declare a variable and specify its type. a variable's type determines its
; size in bytes.

; Variables of integer! type are usually 4 bytes or 32 bits
my-integer: 0
; Red's integers are signed. No support for unsigned atm but that will come.

; To find out the type of variable use type?
type? my-integer
integer!

; A variable can be initialized using another variable that gets initialized
; at the same time.
i2: 1 + i1: 1

; Arithmetic is straightforward
i1 + i2 ; result 3
i2 - i1 ; result 1
i2 * i1 ; result 2
i1 / i2 ; result 0 (0.5, but truncated towards 0)

; Comparison operators are probably familiar, and unlike in other languages you
; only need a single '=' sign for comparison.
; There is a boolean like type in Red. It has values true and false, but also the
; values on/off or yes/no can be used

3 = 2 ; result false
3 != 2 ; result true
3 > 2 ; result true
3 < 2 ; result false
2 <= 2 ; result true
2 >= 2 ; result true

;
; Control Structures
;
; if
; Evaluate a block of code if a given condition is true. IF does not return any value,
; so cannot be used in an expression.
if a < 0 [print "a is negative"]

; either
; Evaluate a block of code if a given condition is true, else evaluate an alternative
; block of code. If the last expressions in both blocks have the same type, EITHER can
; be used inside an expression.
either a < 0 [
   either a = 0 [
       msg: "zero"
   ][
       msg: "negative"
   ]
][
   msg: "positive"
]

print ["a is " msg lf]

; There is an alternative way to write this
; (Which is allowed because all code paths return a value of the same type):

msg: either a < 0 [
   either a = 0 [
       "zero"
   ][
       "negative"
   ]
 ][
   "positive"
]
print ["a is " msg lf]

; until
; Loop over a block of code until the condition at end of block, is met.
; UNTIL does not return any value, so it cannot be used in an expression.
c: 5
until [
   prin "o"
   c: c - 1
   c = 0    ; the condition to end the until loop
]
;   will output:
ooooo
; Note that the loop will always be evaluated at least once, even if the condition is
; not met from the beginning.

; while
; While a given condition is met, evaluate a block of code.
; WHILE does not return any value, so it cannot be used in an expression.
c: 5
while [c > 0][
   prin "o"
   c: c - 1
]
; will output:
ooooo

;
; Functions
;
; function example
twice: function [a [integer!] /one return: [integer!]][
        c: 2
        a: a * c
        either one [a + 1][a]
]
b: 3
print twice b   ; will output 6.

; Import external files with #include and filenames start with a % sign
#include %includefile.red
; Now the functions in the included file can be used too.
```

Further Reading

The main source for information about Red is the Red language homepage.

The source can be found on github.

The Red/System language specification can be found here.

To learn more about Rebol and Red join the chat on Gitter. And if that is not working for you drop a mail to us on the Red mailing list (remove NO_SPAM).

Browse or ask questions on Stack Overflow.

Maybe you want to try Red right away? That is possible on the try Rebol and Red site.

You can also learn Red by learning some Rebol.

Ruby-Ecosystem

People using Ruby generally have a way to install different Ruby versions, manage their packages (or gems), and manage their gem dependencies.

Ruby Managers

Some platforms have Ruby pre-installed or available as a package. Most rubyists do not use these, or if they do, they only use them to bootstrap another Ruby installer or implementation. Instead rubyists tend to install a Ruby manager to install and switch between many versions of Ruby and their projects' Ruby environments.

The following are the popular Ruby environment managers:

- RVM - Installs and switches between rubies. RVM also has the concept of gemsets to isolate projects' environments completely.
- ruby-build - Only installs rubies. Use this for finer control over your rubies' installations.
- rbenv - Only switches between rubies. Used with ruby-build. Use this for finer control over how rubies load.
- chruby - Only switches between rubies. Similar in spirit to rbenv. Unopinionated about how rubies are installed.

Ruby Versions

Ruby was created by Yukihiro "Matz" Matsumoto, who remains somewhat of a BDFL, although that is changing recently. As a result, the reference implementation of Ruby is called MRI (Matz' Reference Implementation), and when you hear a Ruby version, it is referring to the release version of MRI.

The three major version of Ruby in use are:

- 2.0.0 - Released in February 2013. Most major libraries and frameworks support 2.0.0.
- 1.9.3 - Released in October 2011. This is the version most rubyists use currently. Also retired
- 1.8.7 - Ruby 1.8.7 has been retired.

The change between 1.8.7 to 1.9.x is a much larger change than 1.9.3 to 2.0.0. For instance, the 1.9 series introduced encodings and a bytecode VM. There are projects still on 1.8.7, but they are becoming a small minority, as most of the community has moved to at least 1.9.2 or 1.9.3.

Ruby Implementations

The Ruby ecosystem enjoys many different implementations of Ruby, each with unique strengths and states of compatibility. To be clear, the different implementations are written in different languages, but *they are all Ruby*. Each implementation has special hooks and extra features, but they all run normal Ruby files well. For instance, JRuby is written in Java, but you do not need to know Java to use it.

Very mature/compatible:

- MRI - Written in C, this is the reference implementation of Ruby. By definition it is 100% compatible (with itself). All other rubies maintain compatibility with MRI (see RubySpec below).
- JRuby - Written in Java and Ruby, this robust implementation is quite fast. Most importantly, JRuby's strength is JVM/Java interop, leveraging existing JVM tools, projects, and languages.
- Rubinius - Written primarily in Ruby itself with a C++ bytecode VM. Also mature and fast. Because it is implemented in Ruby itself, it exposes many VM features into rubyland.

Medium mature/compatible:

- Maglev - Built on top of Gemstone, a Smalltalk VM. Smalltalk has some impressive tooling, and this project tries to bring that into Ruby development.
- RubyMotion - Brings Ruby to iOS development.

Less mature/compatible:

- Topaz - Written in RPython (using the PyPy toolchain), Topaz is fairly young and not yet compatible. It shows promise to be a high-performance Ruby implementation.

- IronRuby - Written in C# targeting the .NET platform, work on IronRuby seems to have stopped since Microsoft pulled their support.

Ruby implementations may have their own release version numbers, but they always target a specific version of MRI for compatability. Many implementations have the ability to enter different modes (for example, 1.8 or 1.9 mode) to specify which MRI version to target.

RubySpec

Most Ruby implementations rely heavily on RubySpec. Ruby has no official specification, so the community has written executable specs in Ruby to test their implementations' compatibility with MRI.

RubyGems

RubyGems is a community-run package manager for Ruby. RubyGems ships with Ruby, so there is no need to download it separately.

Ruby packages are called "gems," and they can be hosted by the community at RubyGems.org. Each gem contains its source code and some metadata, including things like version, dependencies, author(s), and license(s).

Bundler

Bundler is a gem dependency resolver. It uses a project's Gemfile to find dependencies, and then fetches those dependencies' dependencies recursively. It does this until all dependencies are resolved and downloaded, or it will stop if a conflict has been found.

Bundler will raise an error if it finds conflicting dependencies. For example, if gem A requires version 3 or greater of gem Z, but gem B requires version 2, Bundler will notify you of the conflict. This becomes extremely helpful as many gems refer to other gems (which refer to other gems), which can form a large dependency graph to resolve.

Testing

Testing is a large part of Ruby culture. Ruby comes with its own Unit-style testing framework called minitest (Or TestUnit for Ruby version 1.8.x). There are many testing libraries with different goals.

- TestUnit - Ruby 1.8's built-in "Unit-style" testing framework
- minitest - Ruby 1.9/2.0's built-in testing framework
- RSpec - A testing framework that focuses on expressivity
- Cucumber - A BDD testing framework that parses Gherkin formatted tests

Be Nice

The Ruby community takes pride in being an open, diverse, welcoming community. Matz himself is extremely friendly, and the generosity of rubyists on the whole is amazing.

Ruby

```
# This is a comment
```

```
=begin
This is a multiline comment
No-one uses them
You shouldn't either
=end

# First and foremost: Everything is an object.

# Numbers are objects

3.class #=> Fixnum

3.to_s #=> "3"


# Some basic arithmetic
1 + 1 #=> 2
8 - 1 #=> 7
10 * 2 #=> 20
35 / 5 #=> 7
2**5 #=> 32
5 % 3 #=> 2

# Bitwise operators
3 & 5 #=> 1
3 | 5 #=> 7
3 ^ 5 #=> 6

# Arithmetic is just syntactic sugar
# for calling a method on an object
1.+(3) #=> 4
10.* 5 #=> 50

# Special values are objects
nil # equivalent to null in other languages
true # truth
false # falsehood

nil.class #=> NilClass
true.class #=> TrueClass
false.class #=> FalseClass

# Equality
1 == 1 #=> true
2 == 1 #=> false

# Inequality
1 != 1 #=> false
2 != 1 #=> true

# apart from false itself, nil is the only other 'falsey' value

!nil   #=> true
!false #=> true
```

```
!0     #=> false

# More comparisons
1 < 10 #=> true
1 > 10 #=> false
2 <= 2 #=> true
2 >= 2 #=> true

# Combined comparison operator
1 <=> 10 #=> -1
10 <=> 1 #=> 1
1 <=> 1 #=> 0

# Logical operators
true && false #=> false
true || false #=> true
!true #=> false

# There are alternate versions of the logical operators with much lower
# precedence. These are meant to be used as flow-control constructs to chain
# statements together until one of them returns true or false.

# `do_something_else` only called if `do_something` succeeds.
do_something() and do_something_else()
# `log_error` only called if `do_something` fails.
do_something() or log_error()


# Strings are objects

'I am a string'.class #=> String
"I am a string too".class #=> String

placeholder = 'use string interpolation'
"I can #{placeholder} when using double quoted strings"
#=> "I can use string interpolation when using double quoted strings"

# Prefer single quoted strings to double quoted ones where possible
# Double quoted strings perform additional inner calculations

# Combine strings, but not with numbers
'hello ' + 'world'  #=> "hello world"
'hello ' + 3 #=> TypeError: can't convert Fixnum into String
'hello ' + 3.to_s #=> "hello 3"

# Combine strings and operators
'hello ' * 3 #=> "hello hello hello "

# Append to string
'hello' << ' world' #=> "hello world"

# print to the output with a newline at the end
puts "I'm printing!"
#=> I'm printing!
#=> nil

# print to the output without a newline
print "I'm printing!"
#=> I'm printing! => nil

# Variables
x = 25 #=> 25
x #=> 25

# Note that assignment returns the value assigned
# This means you can do multiple assignment:

x = y = 10 #=> 10
x #=> 10
y #=> 10

# By convention, use snake_case for variable names
snake_case = true

# Use descriptive variable names
path_to_project_root = '/good/name/'
path = '/bad/name/'

# Symbols (are objects)
# Symbols are immutable, reusable constants represented internally by an
# integer value. They're often used instead of strings to efficiently convey
# specific, meaningful values

:pending.class #=> Symbol

status = :pending

status == :pending #=> true

status == 'pending' #=> false

status == :approved #=> false

# Arrays

# This is an array
array = [1, 2, 3, 4, 5] #=> [1, 2, 3, 4, 5]

# Arrays can contain different types of items

[1, 'hello', false] #=> [1, "hello", false]

# Arrays can be indexed
# From the front
array[0] #=> 1
array.first #=> 1
array[12] #=> nil
```

```ruby
# Like arithmetic, [var] access
# is just syntactic sugar
# for calling a method [] on an object
array.[] 0 #=> 1
array.[] 12 #=> nil

# From the end
array[-1] #=> 5
array.last #=> 5

# With a start index and length
array[2, 3] #=> [3, 4, 5]

# Reverse an Array
a=[1,2,3]
a.reverse! #=> [3,2,1]

# Or with a range
array[1..3] #=> [2, 3, 4]

# Add to an array like this
array << 6 #=> [1, 2, 3, 4, 5, 6]
# Or like this
array.push(6) #=> [1, 2, 3, 4, 5, 6]

# Check if an item exists in an array
array.include?(1) #=> true

# Hashes are Ruby's primary dictionary with keys/value pairs.
# Hashes are denoted with curly braces:
hash = { 'color' => 'green', 'number' => 5 }

hash.keys #=> ['color', 'number']

# Hashes can be quickly looked up by key:
hash['color'] #=> 'green'
hash['number'] #=> 5

# Asking a hash for a key that doesn't exist returns nil:
hash['nothing here'] #=> nil

# Since Ruby 1.9, there's a special syntax when using symbols as keys:

new_hash = { defcon: 3, action: true }

new_hash.keys #=> [:defcon, :action]

# Check existence of keys and values in hash
new_hash.key?(:defcon) #=> true
new_hash.value?(3) #=> true

# Tip: Both Arrays and Hashes are Enumerable
# They share a lot of useful methods such as each, map, count, and more
# Control structures

if true
  'if statement'
elsif false
  'else if, optional'
else
  'else, also optional'
end

for counter in 1..5
  puts "iteration #{counter}"
end
#=> iteration 1
#=> iteration 2
#=> iteration 3
#=> iteration 4
#=> iteration 5

# HOWEVER, No-one uses for loops.
# Instead you should use the "each" method and pass it a block.
# A block is a bunch of code that you can pass to a method like "each".
# It is analogous to lambdas, anonymous functions or closures in other
# programming languages.
#
# The "each" method of a range runs the block once for each element of the range.
# The block is passed a counter as a parameter.
# Calling the "each" method with a block looks like this:

(1..5).each do |counter|
  puts "iteration #{counter}"
end
#=> iteration 1
#=> iteration 2
#=> iteration 3
#=> iteration 4
#=> iteration 5

# You can also surround blocks in curly brackets:
(1..5).each { |counter| puts "iteration #{counter}" }

# The contents of data structures can also be iterated using each.
array.each do |element|
  puts "#{element} is part of the array"
end
hash.each do |key, value|
  puts "#{key} is #{value}"
end

# If you still need and index you can use "each_with_index" and define an index
# variable
array.each_with_index do |element, index|
  puts "#{element} is number #{index} in the array"
end
```

```
counter = 1
while counter <= 5 do
  puts "iteration #{counter}"
  counter += 1
end
#=> iteration 1
#=> iteration 2
#=> iteration 3
#=> iteration 4
#=> iteration 5

# There are a bunch of other helpful looping functions in Ruby,
# for example "map", "reduce", "inject", the list goes on. Map,
# for instance, takes the array it's looping over, does something
# to it as defined in your block, and returns an entirely new array.
array = [1,2,3,4,5]
doubled = array.map do |element|
  element * 2
end
puts doubled
#=> [2,4,6,8,10]
puts array
#=> [1,2,3,4,5]

grade = 'B'

case grade
when 'A'
  puts 'Way to go kiddo'
when 'B'
  puts 'Better luck next time'
when 'C'
  puts 'You can do better'
when 'D'
  puts 'Scraping through'
when 'F'
  puts 'You failed!'
else
  puts 'Alternative grading system, eh?'
end
#=> "Better luck next time"

# cases can also use ranges
grade = 82
case grade
when 90..100
  puts 'Hooray!'
when 80...90
  puts 'OK job'
else
  puts 'You failed!'
end
#=> "OK job"
```

```ruby
# exception handling:
begin
  # code here that might raise an exception
  raise NoMemoryError, 'You ran out of memory.'
rescue NoMemoryError => exception_variable
  puts 'NoMemoryError was raised', exception_variable
rescue RuntimeError => other_exception_variable
  puts 'RuntimeError was raised now'
else
  puts 'This runs if no exceptions were thrown at all'
ensure
  puts 'This code always runs no matter what'
end

# Functions

def double(x)
  x * 2
end

# Functions (and all blocks) implicitly return the value of the last statement
double(2) #=> 4

# Parentheses are optional where the result is unambiguous
double 3 #=> 6

double double 3 #=> 12

def sum(x, y)
  x + y
end

# Method arguments are separated by a comma
sum 3, 4 #=> 7

sum sum(3, 4), 5 #=> 12

# yield
# All methods have an implicit, optional block parameter
# it can be called with the 'yield' keyword

def surround
  puts '{'
  yield
  puts '}'
end

surround { puts 'hello world' }

# {
# hello world
# }
# You can pass a block to a function
# "&" marks a reference to a passed block
def guests(&block)
  block.call 'some_argument'
end

# You can pass a list of arguments, which will be converted into an array
# That's what splat operator ("*") is for
def guests(*array)
  array.each { |guest| puts guest }
end

# Define a class with the class keyword
class Human

  # A class variable. It is shared by all instances of this class.
  @@species = 'H. sapiens'

  # Basic initializer
  def initialize(name, age = 0)
    # Assign the argument to the "name" instance variable for the instance
    @name = name
    # If no age given, we will fall back to the default in the arguments list.
    @age = age
  end

  # Basic setter method
  def name=(name)
    @name = name
  end

  # Basic getter method
  def name
    @name
  end

  # The above functionality can be encapsulated using the attr_accessor method as follows
  attr_accessor :name

  # Getter/setter methods can also be created individually like this
  attr_reader :name
  attr_writer :name

  # A class method uses self to distinguish from instance methods.
  # It can only be called on the class, not an instance.
  def self.say(msg)
    puts msg
  end

  def species
    @@species
  end
end
```

```
# Instantiate a class
jim = Human.new('Jim Halpert')

dwight = Human.new('Dwight K. Schrute')

# Let's call a couple of methods
jim.species #=> "H. sapiens"
jim.name #=> "Jim Halpert"
jim.name = "Jim Halpert II" #=> "Jim Halpert II"
jim.name #=> "Jim Halpert II"
dwight.species #=> "H. sapiens"
dwight.name #=> "Dwight K. Schrute"

# Call the class method
Human.say('Hi') #=> "Hi"

# Variable's scopes are defined by the way we name them.
# Variables that start with $ have global scope
$var = "I'm a global var"
defined? $var #=> "global-variable"

# Variables that start with @ have instance scope
@var = "I'm an instance var"
defined? @var #=> "instance-variable"

# Variables that start with @@ have class scope
@@var = "I'm a class var"
defined? @@var #=> "class variable"

# Variables that start with a capital letter are constants
Var = "I'm a constant"
defined? Var #=> "constant"

# Class is also an object in ruby. So class can have instance variables.
# Class variable is shared among the class and all of its descendants.

# base class
class Human
  @@foo = 0

  def self.foo
    @@foo
  end

  def self.foo=(value)
    @@foo = value
  end
end

# derived class
class Worker < Human
end
```

```ruby
Human.foo # 0
Worker.foo # 0

Human.foo = 2 # 2
Worker.foo # 2

# Class instance variable is not shared by the class's descendants.

class Human
  @bar = 0

  def self.bar
    @bar
  end

  def self.bar=(value)
    @bar = value
  end
end

class Doctor < Human
end

Human.bar # 0
Doctor.bar # nil

module ModuleExample
  def foo
    'foo'
  end
end

# Including modules binds their methods to the class instances
# Extending modules binds their methods to the class itself

class Person
  include ModuleExample
end

class Book
  extend ModuleExample
end

Person.foo     # => NoMethodError: undefined method `foo' for Person:Class
Person.new.foo # => 'foo'
Book.foo       # => 'foo'
Book.new.foo   # => NoMethodError: undefined method `foo'

# Callbacks are executed when including and extending a module

module ConcernExample
  def self.included(base)
    base.extend(ClassMethods)
    base.send(:include, InstanceMethods)
  end

  module ClassMethods
    def bar
      'bar'
    end
  end

  module InstanceMethods
    def qux
      'qux'
    end
  end
end

class Something
  include ConcernExample
end

Something.bar     # => 'bar'
Something.qux     # => NoMethodError: undefined method `qux'
Something.new.bar # => NoMethodError: undefined method `bar'
Something.new.qux # => 'qux'
```

Additional resources

- Learn Ruby by Example with Challenges - A variant of this reference with in-browser challenges.
- An Interactive Tutorial for Ruby - Learn Ruby through a series of interactive tutorials.
- Official Documentation
- Ruby from other languages
- Programming Ruby - An older free edition is available online.
- Ruby Style Guide - A community-driven Ruby coding style guide.
- Try Ruby - Learn the basic of Ruby programming language, interactive in the browser.

Rust

Rust is a programming language developed by Mozilla Research. Rust combines low-level control over performance with high-level convenience and safety guarantees.

It achieves these goals without requiring a garbage collector or runtime, making it possible to use Rust libraries as a “drop-in replacement” for C.

Rust’s first release, 0.1, occurred in January 2012, and for 3 years development moved so quickly that until recently the use of stable releases was discouraged and instead the general advice was to use nightly builds.

On May 15th 2015, Rust 1.0 was released with a complete guarantee of backward compatibility. Improvements to compile times and other aspects of the compiler are currently available in the nightly builds. Rust has adopted a train-based release model with regular releases every six weeks. Rust 1.1 beta was made available at the same time of the release of Rust 1.0.

Although Rust is a relatively low-level language, Rust has some functional concepts that are generally found in higher-level languages. This makes Rust not only fast, but also easy and efficient to code in.

```
// This is a comment. Single-line look like this...
/* ...and multi-line comment look like this */

///////////////
// 1. Basics //
///////////////

// Functions
// `i32` is the type for 32-bit signed integers
fn add2(x: i32, y: i32) -> i32 {
    // Implicit return (no semicolon)
    x + y
}

// Main function
fn main() {
    // Numbers //

    // Immutable bindings
    let x: i32 = 1;

    // Integer/float suffixes
    let y: i32 = 13i32;
    let f: f64 = 1.3f64;

    // Type inference
    // Most of the time, the Rust compiler can infer what type a variable is, so
    // you don’t have to write an explicit type annotation.
    // Throughout this tutorial, types are explicitly annotated in many places,
    // but only for demonstrative purposes. Type inference can handle this for
    // you most of the time.
    let implicit_x = 1;
    let implicit_f = 1.3;

    // Arithmetic
    let sum = x + y + 13;

    // Mutable variable
    let mut mutable = 1;
    mutable = 4;
    mutable += 2;

    // Strings //

    // String literals
    let x: &str = "hello world!";

    // Printing
    println!("{} {}", f, x); // 1.3 hello world

    // A `String` - a heap-allocated string
    let s: String = "hello world".to_string();

    // A string slice - an immutable view into another string
    // This is basically an immutable pointer to a string - it doesn’t
    // actually contain the contents of a string, just a pointer to
    // something that does (in this case, `s`)
    let s_slice: &str = &s;

    println!("{} {}", s, s_slice); // hello world hello world

    // Vectors/arrays //

    // A fixed-size array
    let four_ints: [i32; 4] = [1, 2, 3, 4];

    // A dynamic array (vector)
    let mut vector: Vec<i32> = vec![1, 2, 3, 4];
    vector.push(5);

    // A slice - an immutable view into a vector or array
    // This is much like a string slice, but for vectors
    let slice: &[i32] = &vector;

    // Use `{:?}` to print something debug-style
    println!("{:?} {:?}", vector, slice); // [1, 2, 3, 4, 5] [1, 2, 3, 4, 5]

    // Tuples //

    // A tuple is a fixed-size set of values of possibly different types
    let x: (i32, &str, f64) = (1, "hello", 3.4);

    // Destructuring `let`
    let (a, b, c) = x;
    println!("{} {} {}", a, b, c); // 1 hello 3.4

    // Indexing
    println!("{}", x.1); // hello

    //////////////
    // 2. Types //
    //////////////

    // Struct
    struct Point {
        x: i32,
        y: i32,
    }

    let origin: Point = Point { x: 0, y: 0 };

    // A struct with unnamed fields, called a ‘tuple struct’
    struct Point2(i32, i32);

    let origin2 = Point2(0, 0);

    // Basic C-like enum
    enum Direction {
        Left,
        Right,
        Up,
        Down,
    }

    let up = Direction::Up;

    // Enum with fields
    enum OptionalI32 {
        AnI32(i32),
        Nothing,
    }

    let two: OptionalI32 = OptionalI32::AnI32(2);
    let nothing = OptionalI32::Nothing;

    // Generics //

    struct Foo<T> { bar: T }

    // This is defined in the standard library as `Option`
    enum Optional<T> {
        SomeVal(T),
        NoVal,
    }

    // Methods //

    impl<T> Foo<T> {
        // Methods take an explicit `self` parameter
        fn get_bar(self) -> T {
            self.bar
        }
    }

    let a_foo = Foo { bar: 1 };
    println!("{}", a_foo.get_bar()); // 1

    // Traits (known as interfaces or typeclasses in other languages) //

    trait Frobnicate<T> {
        fn frobnicate(self) -> Option<T>;
    }

    impl<T> Frobnicate<T> for Foo<T> {
        fn frobnicate(self) -> Option<T> {
            Some(self.bar)
        }
    }

    let another_foo = Foo { bar: 1 };
    println!("{:?}", another_foo.frobnicate()); // Some(1)
```

```
/////////////////////////
// 3. Pattern matching //
/////////////////////////

let foo = OptionalI32::AnI32(1);
match foo {
    OptionalI32::AnI32(n) => println!("it’s an i32: {}", n),
    OptionalI32::Nothing  => println!("it’s nothing!"),
}

// Advanced pattern matching
struct FooBar { x: i32, y: OptionalI32 }
let bar = FooBar { x: 15, y: OptionalI32::AnI32(32) };

match bar {
    FooBar { x: 0, y: OptionalI32::AnI32(0) } =>
        println!("The numbers are zero!"),
    FooBar { x: n, y: OptionalI32::AnI32(m) } if n == m =>
        println!("The numbers are the same"),
    FooBar { x: n, y: OptionalI32::AnI32(m) } =>
        println!("Different numbers: {} {}", n, m),
    FooBar { x: _, y: OptionalI32::Nothing } =>
        println!("The second number is Nothing!"),
}

/////////////////////
// 4. Control flow //
/////////////////////

// `for` loops/iteration
let array = [1, 2, 3];
for i in array.iter() {
    println!("{}", i);
}

// Ranges
for i in 0u32..10 {
    print!("{} ", i);
}
println!("");
// prints `0 1 2 3 4 5 6 7 8 9 `

// `if`
if 1 == 1 {
    println!("Maths is working!");
} else {
    println!("Oh no...");
}

// `if` as expression
let value = if true {
    "good"
} else {
    "bad"
    };

    // `while` loop
    while 1 == 1 {
        println!("The universe is operating normally.");
    }

    // Infinite loop
    loop {
        println!("Hello!");
    }

    /////////////////////////////////
    // 5. Memory safety & pointers //
    /////////////////////////////////

    // Owned pointer - only one thing can ‘own’ this pointer at a time
    // This means that when the `Box` leaves its scope, it can be automatically deallocated safely.
    let mut mine: Box<i32> = Box::new(3);
    *mine = 5; // dereference
    // Here, `now_its_mine` takes ownership of `mine`. In other words, `mine` is moved.
    let mut now_its_mine = mine;
    *now_its_mine += 2;

    println!("{}", now_its_mine); // 7
    // println!("{}", mine); // this would not compile because `now_its_mine` now owns the pointer

    // Reference - an immutable pointer that refers to other data
    // When a reference is taken to a value, we say that the value has been ‘borrowed’.
    // While a value is borrowed immutably, it cannot be mutated or moved.
    // A borrow lasts until the end of the scope it was created in.
    let mut var = 4;
    var = 3;
    let ref_var: &i32 = &var;

    println!("{}", var); // Unlike `box`, `var` can still be used
    println!("{}", *ref_var);
    // var = 5; // this would not compile because `var` is borrowed
    // *ref_var = 6; // this would not too, because `ref_var` is an immutable reference

    // Mutable reference
    // While a value is mutably borrowed, it cannot be accessed at all.
    let mut var2 = 4;
    let ref_var2: &mut i32 = &mut var2;
    *ref_var2 += 2;         // '*' is used to point to the mutably borrowed var2

    println!("{}", *ref_var2); // 6 , //var2 would not compile. //ref_var2 is of type &mut i32, so
    // var2 = 2; // this would not compile because `var2` is borrowed
}
```

Further reading

There's a lot more to Rust—this is just the basics of Rust so you can understand the most important things. To learn more about Rust, read The Rust Programming Language and check out the /r/rust subreddit. The folks on the #rust channel on irc.mozilla.org are also always keen to help newcomers.

You can also try out features of Rust with an online compiler at the official Rust playpen or on the main Rust website.

Sass

Sass is a CSS extension language that adds features such as variables, nesting, mixins and more. Sass (and other preprocessors, such as Less) help developers write maintainable and DRY (Don't Repeat Yourself) code.

Sass has two different syntax options to choose from. SCSS, which has the same syntax as CSS but with the added features of Sass. Or Sass (the original syntax), which uses indentation rather than curly braces and semicolons. This tutorial is written using SCSS.

If you're already familiar with CSS3, you'll be able to pick up Sass relatively quickly. It does not provide any new styling properties but rather the tools to write your CSS more efficiently and make maintenance much easier.

```
//Single line comments are removed when Sass is compiled to CSS.

/* Multi line comments are preserved. */



/* Variables
============================== */



/* You can store a CSS value (such as a color) in a variable.
Use the '$' symbol to create a variable. */

$primary-color: #A3A4FF;
$secondary-color: #51527F;
$body-font: 'Roboto', sans-serif;

/* You can use the variables throughout your stylesheet.
Now if you want to change a color, you only have to make the change once. */

body {
    background-color: $primary-color;
    color: $secondary-color;
    font-family: $body-font;
}

/* This would compile to: */
body {
    background-color: #A3A4FF;
    color: #51527F;
    font-family: 'Roboto', sans-serif;
}


/* This is much more maintainable than having to change the color
each time it appears throughout your stylesheet. */



/* Mixins
============================== */



/* If you find you are writing the same code for more than one
element, you might want to store that code in a mixin.

Use the '@mixin' directive, plus a name for your mixin. */

@mixin center {
    display: block;
    margin-left: auto;
    margin-right: auto;
    left: 0;
    right: 0;
}

/* You can use the mixin with '@include' and the mixin name. */

div {
    @include center;
    background-color: $primary-color;
}

/* Which would compile to: */
div {
    display: block;
    margin-left: auto;
    margin-right: auto;
    left: 0;
    right: 0;
    background-color: #A3A4FF;
}


/* You can use mixins to create a shorthand property. */

@mixin size($width, $height) {
    width: $width;
    height: $height;
}

/* Which you can invoke by passing width and height arguments. */
```

```
.rectangle {
    @include size(100px, 60px);
}

.square {
    @include size(40px, 40px);
}

/* Compiles to: */
.rectangle {
  width: 100px;
  height: 60px;
}

.square {
  width: 40px;
  height: 40px;
}



/* Functions
============================== */



/* Sass provides functions that can be used to accomplish a variety of
   tasks. Consider the following */

/* Functions can be invoked by using their name and passing in the
   required arguments */
body {
  width: round(10.25px);
}

.footer {
  background-color: fade_out(#000000, 0.25)
}

/* Compiles to: */

body {
  width: 10px;
}

.footer {
  background-color: rgba(0, 0, 0, 0.75);
}

/* You may also define your own functions. Functions are very similar to
   mixins. When trying to choose between a function or a mixin, remember
   that mixins are best for generating CSS while functions are better for
   logic that might be used throughout your Sass code. The examples in
   the Math Operators' section are ideal candidates for becoming a reusable
   function. */

/* This function will take a target size and the parent size and calculate
   and return the percentage */

@function calculate-percentage($target-size, $parent-size) {
  @return $target-size / $parent-size * 100%;
}

$main-content: calculate-percentage(600px, 960px);

.main-content {
  width: $main-content;
}

.sidebar {
  width: calculate-percentage(300px, 960px);
}

/* Compiles to: */

.main-content {
  width: 62.5%;
}

.sidebar {
  width: 31.25%;
}



/* Extend (Inheritance)
============================== */



/* Extend is a way to share the properties of one selector with another. */

.display {
    @include size(5em, 5em);
    border: 5px solid $secondary-color;
}

.display-success {
    @extend .display;
    border-color: #22df56;
}

/* Compiles to: */
.display, .display-success {
  width: 5em;
  height: 5em;
  border: 5px solid #51527F;
}

.display-success {
  border-color: #22df56;
}

/* Extending a CSS statement is preferable to creating a mixin
   because of the way Sass groups together the classes that all share
   the same base styling. If this was done with a mixin, the width,
   height, and border would be duplicated for each statement that
   called the mixin. While it won't affect your workflow, it will
   add unnecessary bloat to the files created by the Sass compiler. */



/* Nesting
============================== */



/* Sass allows you to nest selectors within selectors */

ul {
    list-style-type: none;
    margin-top: 2em;

    li {
        background-color: #FF0000;
    }
}

/* '&' will be replaced by the parent selector. */
/* You can also nest pseudo-classes. */
/* Keep in mind that over-nesting will make your code less maintainable.
Best practices recommend going no more than 3 levels deep when nesting.
For example: */

ul {
    list-style-type: none;
    margin-top: 2em;

    li {
        background-color: red;

        &:hover {
          background-color: blue;
        }

        a {
          color: white;
        }
    }
}
```

```
/* Compiles to: */

ul {
  list-style-type: none;
  margin-top: 2em;
}

ul li {
  background-color: red;
}

ul li:hover {
  background-color: blue;
}

ul li a {
  color: white;
}



/* Partials and Imports
============================== */



/* Sass allows you to create partial files. This can help keep your Sass
   code modularized. Partial files should begin with an '_', e.g. _reset.css.
   Partials are not generated into CSS. */

/* Consider the following CSS which we'll put in a file called _reset.css */

html,
body,
ul,
ol {
  margin: 0;
  padding: 0;
}

/* Sass offers @import which can be used to import partials into a file.
   This differs from the traditional CSS @import statement which makes
   another HTTP request to fetch the imported file. Sass takes the
   imported file and combines it with the compiled code. */

@import 'reset';

body {
  font-size: 16px;
  font-family: Helvetica, Arial, Sans-serif;
}

/* Compiles to: */
```

```
html, body, ul, ol {
  margin: 0;
  padding: 0;
}

body {
  font-size: 16px;
  font-family: Helvetica, Arial, Sans-serif;
}



/* Placeholder Selectors
============================== */



/* Placeholders are useful when creating a CSS statement to extend. If you
   wanted to create a CSS statement that was exclusively used with @extend,
   you can do so using a placeholder. Placeholders begin with a '%' instead
   of '.' or '#'. Placeholders will not appear in the compiled CSS. */

%content-window {
  font-size: 14px;
  padding: 10px;
  color: #000;
  border-radius: 4px;
}

.message-window {
  @extend %content-window;
  background-color: #0000ff;
}

/* Compiles to: */

.message-window {
  font-size: 14px;
  padding: 10px;
  color: #000;
  border-radius: 4px;
}

.message-window {
  background-color: #0000ff;
}



/* Math Operations
============================== */
```

```
/* Sass provides the following operators: +, -, *, /, and %. These can
   be useful for calculating values directly in your Sass files instead
   of using values that you've already calculated by hand. Below is an example
   of a setting up a simple two column design. */

$content-area: 960px;
$main-content: 600px;
$sidebar-content: 300px;

$main-size: $main-content / $content-area * 100%;
$sidebar-size: $sidebar-content / $content-area * 100%;
$gutter: 100% - ($main-size + $sidebar-size);

body {
  width: 100%;
}

.main-content {
  width: $main-size;
}

.sidebar {
  width: $sidebar-size;
}

.gutter {
  width: $gutter;
}

/* Compiles to: */

body {
  width: 100%;
}

.main-content {
  width: 62.5%;
}

.sidebar {
  width: 31.25%;
}

.gutter {
  width: 6.25%;
}
```

SASS or Sass?

Have you ever wondered whether Sass is an acronym or not? You probably haven't, but I'll tell you anyway. The name of the language is a word, "Sass", and not an acronym. Because people were constantly writing it as "SASS", the creator of the language jokingly called it "Syntactically Awesome StyleSheets".

Practice Sass

If you want to play with Sass in your browser, check out SassMeister. You can use either syntax, just go into the settings and select either Sass or SCSS.

Compatibility

Sass can be used in any project as long as you have a program to compile it into CSS. You'll want to verify that the CSS you're using is compatible with your target browsers.

QuirksMode CSS and CanIUse are great resources for checking compatibility.

Further reading

- Official Documentation
- The Sass Way provides tutorials (beginner-advanced) and articles.

Scala

Scala - the scalable language

```
/*
  Set yourself up:

  1) Download Scala - http://www.scala-lang.org/downloads
  2) Unzip/untar to your favourite location and put the bin subdir in your `PATH` environment variable
  3) Start a Scala REPL by running `scala`. You should see the prompt:

  scala>

  This is the so called REPL (Read-Eval-Print Loop). You may type any Scala
  expression, and the result will be printed. We will explain what Scala files
  look like further into this tutorial, but for now, let's start with some
  basics.

*/


/////////////////////////////////////////////////
// 1. Basics
/////////////////////////////////////////////////

// Single-line comments start with two forward slashes

/*
  Multi-line comments, as you can already see from above, look like this.
*/

// Printing, and forcing a new line on the next print
println("Hello world!")
println(10)
```

```scala
// Hello world!
// 10

// Printing, without forcing a new line on next print
print("Hello world")
print(10)
// Hello world!10

// Declaring values is done using either var or val.
// val declarations are immutable, whereas vars are mutable. Immutability is
// a good thing.
val x = 10 // x is now 10
x = 20     // error: reassignment to val
var y = 10
y = 20     // y is now 20

/*
  Scala is a statically typed language, yet note that in the above declarations,
  we did not specify a type. This is due to a language feature called type
  inference. In most cases, Scala compiler can guess what the type of a variable
  is, so you don't have to type it every time. We can explicitly declare the
  type of a variable like so:
*/
val z: Int = 10
val a: Double = 1.0

// Notice automatic conversion from Int to Double, result is 10.0, not 10
val b: Double = 10

// Boolean values
true
false

// Boolean operations
!true         // false
!false        // true
true == false // false
10 > 5        // true

// Math is as per usual
1 + 1   // 2
2 - 1   // 1
5 * 3   // 15
6 / 2   // 3
6 / 4   // 1
6.0 / 4 // 1.5


// Evaluating an expression in the REPL gives you the type and value of the result

1 + 7

/* The above line results in:
  scala> 1 + 7
  res29: Int = 8

  This means the result of evaluating 1 + 7 is an object of type Int with a
  value of 8

  Note that "res29" is a sequentially generated variable name to store the
  results of the expressions you typed, your output may differ.
*/

"Scala strings are surrounded by double quotes"
'a' // A Scala Char
// 'Single quote strings don't exist' <= This causes an error

// Strings have the usual Java methods defined on them
"hello world".length
"hello world".substring(2, 6)
"hello world".replace("C", "3")

// They also have some extra Scala methods. See also: scala.collection.immutable.StringOps
"hello world".take(5)
"hello world".drop(5)

// String interpolation: notice the prefix "s"
val n = 45
s"We have $n apples" // => "We have 45 apples"

// Expressions inside interpolated strings are also possible
val a = Array(11, 9, 6)
s"My second daughter is ${a(0) - a(2)} years old."    // => "My second daughter is 5 years old."
s"We have double the amount of ${n / 2.0} in apples." // => "We have double the amount of 22.5 in apple
s"Power of 2: ${math.pow(2, 2)}"                      // => "Power of 2: 4"

// Formatting with interpolated strings with the prefix "f"
f"Power of 5: ${math.pow(5, 2)}%1.0f"         // "Power of 5: 25"
f"Square root of 122: ${math.sqrt(122)}%1.4f" // "Square root of 122: 11.0454"

// Raw strings, ignoring special characters.
raw"New line feed: \n. Carriage return: \r." // => "New line feed: \n. Carriage return: \r."

// Some characters need to be "escaped", e.g. a double quote inside a string:
"They stood outside the \"Rose and Crown\"" // => "They stood outside the "Rose and Crown""

// Triple double-quotes let strings span multiple rows and contain quotes
val html = """<form id="daform">
                <p>Press belo', Joe</p>
                <input type="submit">
              </form>"""


/////////////////////////////////////////////////
// 2. Functions
/////////////////////////////////////////////////
```

```
// Functions are defined like so:
//
//   def functionName(args...): ReturnType = { body... }
//
// If you come from more traditional languages, notice the omission of the
// return keyword. In Scala, the last expression in the function block is the
// return value.
def sumOfSquares(x: Int, y: Int): Int = {
  val x2 = x * x
  val y2 = y * y
  x2 + y2
}

// The { } can be omitted if the function body is a single expression:
def sumOfSquaresShort(x: Int, y: Int): Int = x * x + y * y

// Syntax for calling functions is familiar:
sumOfSquares(3, 4)  // => 25

// You can use parameters names to specify them in different order
def subtract(x: Int, y: Int): Int = x - y

subtract(10, 3)     // => 7
subtract(y=10, x=3) // => -7

// In most cases (with recursive functions the most notable exception), function
// return type can be omitted, and the same type inference we saw with variables
// will work with function return values:
def sq(x: Int) = x * x  // Compiler can guess return type is Int

// Functions can have default parameters:
def addWithDefault(x: Int, y: Int = 5) = x + y
addWithDefault(1, 2) // => 3
addWithDefault(1)    // => 6


// Anonymous functions look like this:
(x: Int) => x * x

// Unlike defs, even the input type of anonymous functions can be omitted if the
// context makes it clear. Notice the type "Int => Int" which means a function
// that takes Int and returns Int.
val sq: Int => Int = x => x * x

// Anonymous functions can be called as usual:
sq(10)   // => 100

// If each argument in your anonymous function is
// used only once, Scala gives you an even shorter way to define them. These
// anonymous functions turn out to be extremely common, as will be obvious in
// the data structure section.
val addOne: Int => Int = _ + 1
val weirdSum: (Int, Int) => Int = (_ * 2 + _ * 3)
```

```
addOne(5)      // => 6
weirdSum(2, 4) // => 16


// The return keyword exists in Scala, but it only returns from the inner-most
// def that surrounds it.
// WARNING: Using return in Scala is error-prone and should be avoided.
// It has no effect on anonymous functions. For example:
def foo(x: Int): Int = {
  val anonFunc: Int => Int = { z =>
    if (z > 5)
      return z // This line makes z the return value of foo!
    else
      z + 2    // This line is the return value of anonFunc
  }
  anonFunc(x)  // This line is the return value of foo
}


/////////////////////////////////////////////////
// 3. Flow Control
/////////////////////////////////////////////////

1 to 5
val r = 1 to 5
r.foreach(println)

r foreach println
// NB: Scala is quite lenient when it comes to dots and brackets - study the
// rules separately. This helps write DSLs and APIs that read like English

(5 to 1 by -1) foreach (println)

// A while loops
var i = 0
while (i < 10) { println("i " + i); i += 1 }

while (i < 10) { println("i " + i); i += 1 }   // Yes, again. What happened? Why?

i    // Show the value of i. Note that while is a loop in the classical sense -
     // it executes sequentially while changing the loop variable. while is very
     // fast, faster that Java loops, but using the combinators and
     // comprehensions above is easier to understand and parallelize

// A do while loop
i = 0
do {
  println("i is still less than 10")
  i += 1
} while (i < 10)

// Tail recursion is an idiomatic way of doing recurring things in Scala.
// Recursive functions need an explicit return type, the compiler can't infer it.
// Here it's Unit.
```

```
def showNumbersInRange(a: Int, b: Int): Unit = {
  print(a)
  if (a < b)
    showNumbersInRange(a + 1, b)
}
showNumbersInRange(1, 14)


// Conditionals

val x = 10

if (x == 1) println("yeah")
if (x == 10) println("yeah")
if (x == 11) println("yeah")
if (x == 11) println ("yeah") else println("nay")

println(if (x == 10) "yeah" else "nope")
val text = if (x == 10) "yeah" else "nope"


/////////////////////////////////////////////////
// 4. Data Structures
/////////////////////////////////////////////////

val a = Array(1, 2, 3, 5, 8, 13)
a(0)     // Int = 1
a(3)     // Int = 5
a(21)    // Throws an exception

val m = Map("fork" -> "tenedor", "spoon" -> "cuchara", "knife" -> "cuchillo")
m("fork")         // java.lang.String = tenedor
m("spoon")        // java.lang.String = cuchara
m("bottle")       // Throws an exception

val safeM = m.withDefaultValue("no lo se")
safeM("bottle")   // java.lang.String = no lo se

val s = Set(1, 3, 7)
s(0)      // Boolean = false
s(1)      // Boolean = true

/* Look up the documentation of map here -
 * http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.Map
 * and make sure you can read it
 */


// Tuples

(1, 2)

(4, 3, 2)
```

```
(1, 2, "three")

(a, 2, "three")

// Why have this?
val divideInts = (x: Int, y: Int) => (x / y, x % y)

// The function divideInts gives you the result and the remainder
divideInts(10, 3)    // (Int, Int) = (3,1)

// To access the elements of a tuple, use _._n where n is the 1-based index of
// the element
val d = divideInts(10, 3)    // (Int, Int) = (3,1)

d._1    // Int = 3
d._2    // Int = 1

// Alternatively you can do multiple-variable assignment to tuple, which is more
// convenient and readable in many cases
val (div, mod) = divideInts(10, 3)

div     // Int = 3
mod     // Int = 1


/////////////////////////////////////////////////
// 5. Object Oriented Programming
/////////////////////////////////////////////////

/*
  Aside: Everything we've done so far in this tutorial has been simple
  expressions (values, functions, etc). These expressions are fine to type into
  the command-line interpreter for quick tests, but they cannot exist by
  themselves in a Scala file. For example, you cannot have just "val x = 5" in
  a Scala file. Instead, the only top-level constructs allowed in Scala are:

  - objects
  - classes
  - case classes
  - traits

  And now we will explain what these are.
*/

// classes are similar to classes in other languages. Constructor arguments are
// declared after the class name, and initialization is done in the class body.
class Dog(br: String) {
  // Constructor code here
  var breed: String = br

  // Define a method called bark, returning a String
  def bark = "Woof, woof!"

  // Values and methods are assumed public. "protected" and "private" keywords
  // are also available.
  private def sleep(hours: Int) =
    println(s"I'm sleeping for $hours hours")

  // Abstract methods are simply methods with no body. If we uncomment the next
  // line, class Dog would need to be declared abstract
  //   abstract class Dog(...) { ... }
  // def chaseAfter(what: String): String
}

val mydog = new Dog("greyhound")
println(mydog.breed) // => "greyhound"
println(mydog.bark)  // => "Woof, woof!"


// The "object" keyword creates a type AND a singleton instance of it. It is
// common for Scala classes to have a "companion object", where the per-instance
// behavior is captured in the classes themselves, but behavior related to all
// instance of that class go in objects. The difference is similar to class
// methods vs static methods in other languages. Note that objects and classes
// can have the same name.
object Dog {
  def allKnownBreeds = List("pitbull", "shepherd", "retriever")
  def createDog(breed: String) = new Dog(breed)
}


// Case classes are classes that have extra functionality built in. A common
// question for Scala beginners is when to use classes and when to use case
// classes. The line is quite fuzzy, but in general, classes tend to focus on
// encapsulation, polymorphism, and behavior. The values in these classes tend
// to be private, and only methods are exposed. The primary purpose of case
// classes is to hold immutable data. They often have few methods, and the
// methods rarely have side-effects.
case class Person(name: String, phoneNumber: String)

// Create a new instance. Note cases classes don't need "new"
val george = Person("George", "1234")
val kate = Person("Kate", "4567")

// With case classes, you get a few perks for free, like getters:
george.phoneNumber  // => "1234"

// Per field equality (no need to override .equals)
Person("George", "1234") == Person("Kate", "1236")  // => false

// Easy way to copy
// otherGeorge == Person("george", "9876")
val otherGeorge = george.copy(phoneNumber = "9876")

// And many others. Case classes also get pattern matching for free, see below.


// Traits coming soon!
```

```
/////////////////////////////////////////////////
// 6. Pattern Matching
/////////////////////////////////////////////////

// Pattern matching is a powerful and commonly used feature in Scala. Here's how
// you pattern match a case class. NB: Unlike other languages, Scala cases do
// not need breaks, fall-through does not happen.

def matchPerson(person: Person): String = person match {
  // Then you specify the patterns:
  case Person("George", number) => "We found George! His number is " + number
  case Person("Kate", number)   => "We found Kate! Her number is " + number
  case Person(name, number)     => "We matched someone : " + name + ", phone : " + number
}

val email = "(.*)@(.*)".r  // Define a regex for the next example.

// Pattern matching might look familiar to the switch statements in the C family
// of languages, but this is much more powerful. In Scala, you can match much
// more:
def matchEverything(obj: Any): String = obj match {
  // You can match values:
  case "Hello world" => "Got the string Hello world"

  // You can match by type:
  case x: Double => "Got a Double: " + x

  // You can specify conditions:
  case x: Int if x > 10000 => "Got a pretty big number!"

  // You can match case classes as before:
  case Person(name, number) => s"Got contact info for $name!"

  // You can match regular expressions:
  case email(name, domain) => s"Got email address $name@$domain"

  // You can match tuples:
  case (a: Int, b: Double, c: String) => s"Got a tuple: $a, $b, $c"

  // You can match data structures:
  case List(1, b, c) => s"Got a list with three elements and starts with 1: 1, $b, $c"

  // You can nest patterns:
  case List(List((1, 2, "YAY"))) => "Got a list of list of tuple"

  // Match any case (default) if all previous haven't matched
  case _ => "Got unknown object"
}

// In fact, you can pattern match any object with an "unapply" method. This
// feature is so powerful that Scala lets you define whole functions as
// patterns:
```

```
val patternFunc: Person => String = {
  case Person("George", number) => s"George's number: $number"
  case Person(name, number) => s"Random person's number: $number"
}


/////////////////////////////////////////////////
// 7. Functional Programming
/////////////////////////////////////////////////

// Scala allows methods and functions to return, or take as parameters, other
// functions or methods.

val add10: Int => Int = _ + 10 // A function taking an Int and returning an Int
List(1, 2, 3) map add10 // List(11, 12, 13) - add10 is applied to each element

// Anonymous functions can be used instead of named functions:
List(1, 2, 3) map (x => x + 10)

// And the underscore symbol, can be used if there is just one argument to the
// anonymous function. It gets bound as the variable
List(1, 2, 3) map (_ + 10)

// If the anonymous block AND the function you are applying both take one
// argument, you can even omit the underscore
List("Dom", "Bob", "Natalia") foreach println


// Combinators

s.map(sq)

val sSquared = s. map(sq)

sSquared.filter(_ < 10)

sSquared.reduce (_+_)

// The filter function takes a predicate (a function from A -> Boolean) and
// selects all elements which satisfy the predicate
List(1, 2, 3) filter (_ > 2) // List(3)
case class Person(name: String, age: Int)
List(
  Person(name = "Dom", age = 23),
  Person(name = "Bob", age = 30)
).filter(_.age > 25) // List(Person("Bob", 30))


// Scala a foreach method defined on certain collections that takes a type
// returning Unit (a void method)
val aListOfNumbers = List(1, 2, 3, 4, 10, 20, 100)
aListOfNumbers foreach (x => println(x))
aListOfNumbers foreach println
```

```
// For comprehensions

for { n <- s } yield sq(n)

val nSquared2 = for { n <- s } yield sq(n)

for { n <- nSquared2 if n < 10 } yield n

for { n <- s; nSquared = n * n if nSquared < 10} yield nSquared

/* NB Those were not for loops. The semantics of a for loop is 'repeat', whereas
   a for-comprehension defines a relationship between two sets of data. */


/////////////////////////////////////////////////
// 8. Implicits
/////////////////////////////////////////////////

/* WARNING WARNING: Implicits are a set of powerful features of Scala, and
 * therefore it is easy to abuse them. Beginners to Scala should resist the
 * temptation to use them until they understand not only how they work, but also
 * best practices around them. We only include this section in the tutorial
 * because they are so commonplace in Scala libraries that it is impossible to
 * do anything meaningful without using a library that has implicits. This is
 * meant for you to understand and work with implicts, not declare your own.
 */

// Any value (vals, functions, objects, etc) can be declared to be implicit by
// using the, you guessed it, "implicit" keyword. Note we are using the Dog
// class from section 5 in these examples.
implicit val myImplicitInt = 100
implicit def myImplicitFunction(breed: String) = new Dog("Golden " + breed)

// By itself, implicit keyword doesn't change the behavior of the value, so
// above values can be used as usual.
myImplicitInt + 2                   // => 102
myImplicitFunction("Pitbull").breed // => "Golden Pitbull"

// The difference is that these values are now eligible to be used when another
// piece of code "needs" an implicit value. One such situation is implicit
// function arguments:
def sendGreetings(toWhom: String)(implicit howMany: Int) =
  s"Hello $toWhom, $howMany blessings to you and yours!"

// If we supply a value for "howMany", the function behaves as usual
sendGreetings("John")(1000)  // => "Hello John, 1000 blessings to you and yours!"

// But if we omit the implicit parameter, an implicit value of the same type is
// used, in this case, "myImplicitInt":
sendGreetings("Jane")  // => "Hello Jane, 100 blessings to you and yours!"

// Implicit function parameters enable us to simulate type classes in other
// functional languages. It is so often used that it gets its own shorthand. The
// following two lines mean the same thing:
```

```
// def foo[T](implicit c: C[T]) = ...
// def foo[T : C] = ...


// Another situation in which the compiler looks for an implicit is if you have
//   obj.method(...)
// but "obj" doesn't have "method" as a method. In this case, if there is an
// implicit conversion of type A => B, where A is the type of obj, and B has a
// method called "method", that conversion is applied. So having
// myImplicitFunction above in scope, we can say:
"Retriever".breed // => "Golden Retriever"
"Sheperd".bark    // => "Woof, woof!"

// Here the String is first converted to Dog using our function above, and then
// the appropriate method is called. This is an extremely powerful feature, but
// again, it is not to be used lightly. In fact, when you defined the implicit
// function above, your compiler should have given you a warning, that you
// shouldn't do this unless you really know what you're doing.


/////////////////////////////////////////////////
// 9. Misc
/////////////////////////////////////////////////

// Importing things
import scala.collection.immutable.List

// Import all "sub packages"
import scala.collection.immutable._

// Import multiple classes in one statement
import scala.collection.immutable.{List, Map}

// Rename an import using '=>'
import scala.collection.immutable.{List => ImmutableList}

// Import all classes, except some. The following excludes Map and Set:
import scala.collection.immutable.{Map => _, Set => _, _}

// Java classes can also be imported. Scala syntax can be used
import java.swing.{JFrame, JWindow}

// Your programs entry point is defined in an scala file using an object, with a
// single method, main:
object Application {
  def main(args: Array[String]): Unit = {
    // stuff goes here.
  }
}

// Files can contain multiple classes and objects. Compile with scalac
// Input and output

// To read a file line by line
import scala.io.Source
for(line <- Source.fromFile("myfile.txt").getLines())
  println(line)

// To write a file use Java's PrintWriter
val writer = new PrintWriter("myfile.txt")
writer.write("Writing line for line" + util.Properties.lineSeparator)
writer.write("Another line here" + util.Properties.lineSeparator)
writer.close()
```

Further resources

- Scala for the impatient
- Twitter Scala school
- The scala documentation
- Try Scala in your browser
- Join the Scala user group

Self

Self is a fast prototype based OO language which runs in its own JIT vm. Most development is done through interacting with live objects through a visual development environment called *morphic* with integrated browsers and debugger.

Everything in Self is an object. All computation is done by sending messages to objects. Objects in Self can be understood as sets of key-value slots.

Constructing objects

The inbuild Self parser can construct objects, including method objects.

```
"This is a comment"

"A string:"
'This is a string with \'escaped\' characters.\n'

"A 30 bit integer"
23

"A 30 bit float"
3.2

"-20"
-14r16

"An object which only understands one message, 'x' which returns 20"
(|
  x = 20.
|)

"An object which also understands 'x:' which sets the x slot"
(|
  x <- 20.
|)

"An object which understands the method 'doubleX' which
doubles the value of x and then returns the object"
(|
  x <- 20.
  doubleX = (x: x * 2. self)
|)

"An object which understands all the messages
that 'traits point' understands". The parser
looks up 'traits point' by sending the messages
'traits' then 'point' to a known object called
the 'lobby'. It looks up the 'true' object by
also sending the message 'true' to the lobby."
(|     parent* = traits point.
       x = 7.
       y <- 5.
       isNice = true.
|)
```

Sending messages to objects

Messages can either be unary, binary or keyword. Precedence is in that order. Unlike Smalltalk, the precedence of binary messages must be specified, and all keywords after the first must start with a capital letter. Messages are separeated from their destination by whitespace.

```
"unary message, sends 'printLine' to the object '23'
which prints the string '23' to stdout and returns the receiving object (ie 23)"
23 printLine

"sends the message '+' with '7' to '23', then the message '*' with '8' to the result"
(23 + 7) * 8

"sends 'power:' to '2' with '8' returns 256"
2 power: 8

"sends 'keyOf:IfAbsent:' to 'hello' with arguments 'e' and '-1'.
Returns 1, the index of 'e' in 'hello'."
'hello' keyOf: 'e' IfAbsent: -1
```

Blocks

Self defines flow control like Smalltalk and Ruby by way of blocks. Blocks are delayed computations of the form:

```
[|:x. localVar| x doSomething with: localVar]
```

Examples of the use of a block:

```
"returns 'HELLO'"
'hello' copyMutable mapBy: [|:c| c capitalize]

"returns 'Nah'"
'hello' size > 5 ifTrue: ['Yay'] False: ['Nah']

"returns 'HaLLO'"
'hello' copyMutable mapBy: [|:c|
   c = 'e' ifTrue: [c capitalize]
            False: ['a']]
```

Multiple expressions are separated by a period. ^ returns immediately.

```
"returns An 'E'! How icky!"
'hello' copyMutable mapBy: [|:c. tmp <- ''|
   tmp: c capitalize.
   tmp = 'E' ifTrue: [^ 'An \'E\'! How icky!'].
   c capitalize
   ]
```

Blocks are performed by sending them the message ‘value’ and inherit (delegate to) their contexts:

```
"returns 0"
[|x|
    x: 15.
    "Repeatedly sends 'value' to the first block while the result of sending 'value' to the
     second block is the 'true' object"
    [x > 0] whileTrue: [x: x - 1].
    x
] value
```

Methods

Methods are like blocks but they are not within a context but instead are stored as values of slots. Unlike Smalltalk, methods by default return their final value not ‘self’.

```
"Here is an object with one assignable slot 'x' and a method 'reduceXTo: y'.
Sending the message 'reduceXTo: 10' to this object will put
the object '10' in the 'x' slot and return the original object"
(|
    x <- 50.
    reduceXTo: y = (
        [x > y] whileTrue: [x: x - 1].
        self)
|)
.
```

Prototypes

Self has no classes. The way to get an object is to find a prototype and copy it.

```
| d |
d: dictionary copy.
d at: 'hello' Put: 23 + 8.
d at: 'goodbye' Put: 'No!.
"Prints No!"
```

```
( d at: 'goodbye' IfAbsent: 'Yes! ) printLine.
"Prints 31"
( d at: 'hello' IfAbsent: -1 ) printLine.
```

Further information

The Self handbook has much more information, and nothing beats hand-on experience with Self by downloading it from the homepage.

Smalltalk

- Smalltalk is an object-oriented, dynamically typed, reflective programming language.
- Smalltalk was created as the language to underpin the "new world" of computing exemplified by "human–computer symbiosis."
- It was designed and created in part for educational use, more so for constructionist learning, at the Learning Research Group (LRG) of Xerox PARC by Alan Kay, Dan Ingalls, Adele Goldberg, Ted Kaehler, Scott Wallace, and others during the 1970s.

Feedback highly appreciated! Reach me at [@jigyasa_grover](https://twitter.com/jigyasa_grover) or send me an e-mail at `grover.jigyasa1@gmail.com`.

Allowable characters:

- a-z
- A-Z
- 0-9
- .+/*~<>@%|&?
- blank, tab, cr, ff, lf

Variables:

- variables must be declared before use
- shared vars must begin with uppercase
- local vars must begin with lowercase
- reserved names: `nil`, `true`, `false`, `self`, `super`, and `Smalltalk`

Variable scope:

- Global: defined in Dictionary Smalltalk and accessible by all objects in system - Special: (reserved) `Smalltalk`, `super`, `self`, `true`, `false`, & `nil`
- Method Temporary: local to a method
- Block Temporary: local to a block
- Pool: variables in a Dictionary object
- Method Parameters: automatic local vars created as a result of message call with params
- Block Parameters: automatic local vars created as a result of value: message call
- Class: shared with all instances of one class & its subclasses
- Class Instance: unique to each instance of a class
- Instance Variables: unique to each instance

```
"Comments are enclosed in quotes"
```

```
"Period (.) is the statement seperator"
```

Transcript:

```
Transcript clear.                                              "clear to transcript window"
Transcript show: 'Hello World'.                           "output string in transcript window"
Transcript nextPutAll: 'Hello World'.                     "output string in transcript window"
Transcript nextPut: $A.                                "output character in transcript window"
Transcript space.                                "output space character in transcript window"
Transcript tab.                                    "output tab character in transcript window"
Transcript cr.                                                 "carriage return / linefeed"
'Hello' printOn: Transcript.                             "append print string into the window"
'Hello' storeOn: Transcript.                             "append store string into the window"
Transcript endEntry.                                           "flush the output buffer"
```

Assignment:

```
| x y |
x _ 4.                                                         "assignment (Squeak) <-"
x := 5.                                                        "assignment"
x := y := z := 6.                                              "compound assignment"
x := (y := 6) + 1.
x := Object new.                                       "bind to allocated instance of a class"
x := 123 class.                                                "discover the object class"
x := Integer superclass.                                  "discover the superclass of a class"
x := Object allInstances.                           "get an array of all instances of a class"
x := Integer allSuperclasses.                                "get all superclasses of a class"
x := 1.2 hash.                                                 "hash value for object"
y := x copy.                                                   "copy object"
y := x shallowCopy.                                             "copy object (not overridden)"
y := x deepCopy.                                               "copy object and instance vars"
y := x veryDeepCopy.                                   "complete tree copy using a dictionary"
```

Constants:

```
| b |
b := true.                                                     "true constant"
b := false.                                                    "false constant"
x := nil.                                                      "nil object constant"
x := 1.                                                        "integer constants"
x := 3.14.                                                     "float constants"
x := 2e-2.                                                     "fractional constants"
x := 16r0F.                                                    "hex constant".
x := -1.                                                       "negative constants"
x := 'Hello'.                                                  "string constant"
x := 'I''m here'.                                              "single quote escape"
x := $A.                                                       "character constant"
x := $ .                                                       "character constant (space)"
x := #aSymbol.                                                 "symbol constants"
x := #(3 2 1).                                                 "array constants"
x := #('abc' 2 $a).                                            "mixing of types allowed"
```

Booleans:

```
| b x y |
x := 1. y := 2.
b := (x = y).                                               "equals"
b := (x ~= y).                                              "not equals"
b := (x == y).                                              "identical"
b := (x ~~ y).                                              "not identical"
b := (x > y).                                               "greater than"
b := (x < y).                                               "less than"
b := (x >= y).                                              "greater than or equal"
b := (x <= y).                                              "less than or equal"
b := b not.                                                 "boolean not"
b := (x < 5) & (y > 1).                                     "boolean and"
b := (x < 5) | (y > 1).                                     "boolean or"
b := (x < 5) and: [y > 1].                                  "boolean and (short-circuit)"
b := (x < 5) or: [y > 1].                                   "boolean or (short-circuit)"
b := (x < 5) eqv: (y > 1).                               "test if both true or both false"
b := (x < 5) xor: (y > 1).                              "test if one true and other false"
b := 5 between: 3 and: 12.                                  "between (inclusive)"
b := 123 isKindOf: Number.                        "test if object is class or subclass of"
b := 123 isMemberOf: SmallInteger.                       "test if object is type of class"
b := 123 respondsTo: sqrt.                            "test if object responds to message"
b := x isNil.                                               "test if object is nil"
b := x isZero.                                              "test if number is zero"
b := x positive.                                            "test if number is positive"
b := x strictlyPositive.                             "test if number is greater than zero"
b := x negative.                                            "test if number is negative"
b := x even.                                                "test if number is even"
b := x odd.                                                 "test if number is odd"
b := x isLiteral.                                           "test if literal constant"
b := x isInteger.                                           "test if object is integer"
b := x isFloat.                                             "test if object is float"
b := x isNumber.                                            "test if object is number"
b := $A isUppercase.                                        "test if upper case character"
b := $A isLowercase.                                        "test if lower case character"
```

Arithmetic expressions:

```
| x |
x := 6 + 3.                                                 "addition"
x := 6 - 3.                                                 "subtraction"
x := 6 * 3.                                                 "multiplication"
x := 1 + 2 * 3.                              "evaluation always left to right (1 + 2) * 3"
x := 5 / 3.                                              "division with fractional result"
x := 5.0 / 3.0.                                             "division with float result"
x := 5.0 // 3.0.                                            "integer divide"
x := 5.0 \\ 3.0.                                            "integer remainder"
x := -5.                                                    "unary minus"
x := 5 sign.                                                "numeric sign (1, -1 or 0)"
x := 5 negated.                                             "negate receiver"
x := 1.2 integerPart.                                       "integer part of number (1.0)"
x := 1.2 fractionPart.                                   "fractional part of number (0.2)"
```

```
x := 5 reciprocal.                                          "reciprocal function"
x := 6 * 3.1.                                               "auto convert to float"
x := 5 squared.                                             "square function"
x := 25 sqrt.                                               "square root"
x := 5 raisedTo: 2.                                         "power function"
x := 5 raisedToInteger: 2.                                  "power function with integer"
x := 5 exp.                                                 "exponential"
x := -5 abs.                                                "absolute value"
x := 3.99 rounded.                                          "round"
x := 3.99 truncated.                                        "truncate"
x := 3.99 roundTo: 1.                                  "round to specified decimal places"
x := 3.99 truncateTo: 1.                            "truncate to specified decimal places"
x := 3.99 floor.                                            "truncate"
x := 3.99 ceiling.                                          "round up"
x := 5 factorial.                                           "factorial"
x := -5 quo: 3.                                       "integer divide rounded toward zero"
x := -5 rem: 3.                                    "integer remainder rounded toward zero"
x := 28 gcd: 12.                                            "greatest common denominator"
x := 28 lcm: 12.                                            "least common multiple"
x := 100 ln.                                                "natural logarithm"
x := 100 log.                                               "base 10 logarithm"
x := 100 log: 10.                                          "logarithm with specified base"
x := 100 floorLog: 10.                                      "floor of the log"
x := 180 degreesToRadians.                                  "convert degrees to radians"
x := 3.14 radiansToDegrees.                                 "convert radians to degrees"
x := 0.7 sin.                                               "sine"
x := 0.7 cos.                                               "cosine"
x := 0.7 tan.                                               "tangent"
x := 0.7 arcSin.                                            "arcsine"
x := 0.7 arcCos.                                            "arccosine"
x := 0.7 arcTan.                                            "arctangent"
x := 10 max: 20.                                            "get maximum of two numbers"
x := 10 min: 20.                                            "get minimum of two numbers"
x := Float pi.                                              "pi"
x := Float e.                                               "exp constant"
x := Float infinity.                                        "infinity"
x := Float nan.                                             "not-a-number"
x := Random new next; yourself. x next.                  "random number stream (0.0 to 1.0)
x := 100 atRandom.                                          "quick random number"
```

Bitwise Manipulation:

```
| b x |
x := 16rFF bitAnd: 16r0F.                                   "and bits"
x := 16rF0 bitOr: 16r0F.                                    "or bits"
x := 16rFF bitXor: 16r0F.                                   "xor bits"
x := 16rFF bitInvert.                                       "invert bits"
x := 16r0F bitShift: 4.                                     "left shift"
x := 16rF0 bitShift: -4.                                    "right shift"
"x := 16r80 bitAt: 7."                                   "bit at position (0|1) [!Squeak]"
x := 16r80 highbit.                                         "position of highest bit set"
b := 16rFF allMask: 16r0F.                   "test if all bits set in mask set in receiver"
b := 16rFF anyMask: 16r0F.                   "test if any bits set in mask set in receiver"
```

```
b := 16rFF noMask: 16r0F.                    "test if all bits set in mask clear in receiver"
```

Conversion:

```
| x |
x := 3.99 asInteger.                        "convert number to integer (truncates in Squeak)"
x := 3.99 asFraction.                                  "convert number to fraction"
x := 3 asFloat.                                        "convert number to float"
x := 65 asCharacter.                                   "convert integer to character"
x := $A asciiValue.                                    "convert character to integer"
x := 3.99 printString.                           "convert object to string via printOn:"
x := 3.99 storeString.                           "convert object to string via storeOn:"
x := 15 radix: 16.                                   "convert to string in given base"
x := 15 printStringBase: 16.
x := 15 storeStringBase: 16.
```

Blocks:

- blocks are objects and may be assigned to a variable
- value is last expression evaluated unless explicit return
- blocks may be nested
- specification [arguments | | localvars | expressions]
- Squeak does not currently support localvars in blocks
- max of three arguments allowed
- ^expression terminates block & method (exits all nested blocks)
- blocks intended for long term storage should not contain ^

```
| x y z |
x := [ y := 1. z := 2. ]. x value.                          "simple block usage"
x := [ :argOne :argTwo |  argOne, ' and ' , argTwo.].     "set up block with argument passing"
Transcript show: (x value: 'First' value: 'Second'); cr.   "use block with argument passing"
"x := [ | z | z := 1.]. *** localvars not available in squeak blocks"
```

Method calls:

- unary methods are messages with no arguments
- binary methods
- keyword methods are messages with selectors including colons standard categories/protocols: - initialize-release (methods called for new instance)
- accessing (get/set methods)
- testing (boolean tests - is)
- comparing (boolean tests with parameter
- displaying (gui related methods)
- printing (methods for printing)
- updating (receive notification of changes)
- private (methods private to class)
- instance-creation (class methods for creating instance)

```
| x |
x := 2 sqrt.                                                "unary message"
x := 2 raisedTo: 10.                                        "keyword message"
x := 194 * 9.                                               "binary message"
Transcript show: (194 * 9) printString; cr.                 "combination (chaining)"
x := 2 perform: #sqrt.                                   "indirect method invocation"
Transcript                                 "Cascading - send multiple messages to receiver"
   show: 'hello ';
   show: 'world';
   cr.
x := 3 + 2; * 100.                          "result=300. Sends message to same receiver (3)"
```

Conditional Statements:

```
| x |
x > 10 ifTrue: [Transcript show: 'ifTrue'; cr].             "if then"
x > 10 ifFalse: [Transcript show: 'ifFalse'; cr].           "if else"
x > 10                                                      "if then else"
   ifTrue: [Transcript show: 'ifTrue'; cr]
   ifFalse: [Transcript show: 'ifFalse'; cr].
x > 10                                                      "if else then"
   ifFalse: [Transcript show: 'ifFalse'; cr]
   ifTrue: [Transcript show: 'ifTrue'; cr].
Transcript
   show:
      (x > 10
         ifTrue: ['ifTrue']
         ifFalse: ['ifFalse']);
   cr.
Transcript                                                  "nested if then else"
   show:
      (x > 10
         ifTrue: [x > 5
            ifTrue: ['A']
            ifFalse: ['B']]
         ifFalse: ['C']);
   cr.
switch := Dictionary new.                                   "switch functionality"
switch at: $A put: [Transcript show: 'Case A'; cr].
switch at: $B put: [Transcript show: 'Case B'; cr].
switch at: $C put: [Transcript show: 'Case C'; cr].
result := (switch at: $B) value.
```

Iteration statements:

```
| x y |
x := 4. y := 1.
[x > 0] whileTrue: [x := x - 1. y := y * 2].                "while true loop"
[x >= 4] whileFalse: [x := x + 1. y := y * 2].              "while false loop"
x timesRepeat: [y := y * 2].                             "times repear loop (i := 1 to x)"
1 to: x do: [:a | y := y * 2].                              "for loop"
1 to: x by: 2 do: [:a | y := y / 2].                   "for loop with specified increment"
#(5 4 3) do: [:a | x := x + a].                             "iterate over array elements"
```

Character:

```
| x y |
x := $A.                                                    "character assignment"
y := x isLowercase.                                         "test if lower case"
y := x isUppercase.                                         "test if upper case"
y := x isLetter.                                            "test if letter"
y := x isDigit.                                             "test if digit"
y := x isAlphaNumeric.                                      "test if alphanumeric"
y := x isSeparator.                                         "test if seperator char"
y := x isVowel.                                             "test if vowel"
y := x digitValue.                                        "convert to numeric digit value"
y := x asLowercase.                                         "convert to lower case"
y := x asUppercase.                                         "convert to upper case"
y := x asciiValue.                                        "convert to numeric ascii value"
y := x asString.                                            "convert to string"
b := $A <= $B.                                              "comparison"
y := $A max: $B.
```

Symbol:

```
| b x y |
x := #Hello.                                                "symbol assignment"
y := 'String', 'Concatenation'.                  "symbol concatenation (result is string)"
b := x isEmpty.                                             "test if symbol is empty"
y := x size.                                                "string size"
y := x at: 2.                                               "char at location"
y := x copyFrom: 2 to: 4.                                   "substring"
y := x indexOf: $e ifAbsent: [0].             "first position of character within string"
x do: [:a | Transcript show: a printString; cr].            "iterate over the string"
b := x conform: [:a | (a >= $a) & (a <= $z)].       "test if all elements meet condition"
y := x select: [:a | a > $a].                    "return all elements that meet condition"
y := x asString.                                            "convert symbol to string"
y := x asText.                                              "convert symbol to text"
y := x asArray.                                             "convert symbol to array"
y := x asOrderedCollection.                         "convert symbol to ordered collection"
y := x asSortedCollection.                           "convert symbol to sorted collection"
y := x asBag.                                           "convert symbol to bag collection"
y := x asSet.                                           "convert symbol to set collection"
```

String:

```
| b x y |
x := 'This is a string'.                                    "string assignment"
x := 'String', 'Concatenation'.                             "string concatenation"
b := x isEmpty.                                             "test if string is empty"
y := x size.                                                "string size"
y := x at: 2.                                               "char at location"
y := x copyFrom: 2 to: 4.                                   "substring"
y := x indexOf: $a ifAbsent: [0].             "first position of character within string"
x := String new: 4.                                         "allocate string object"
x                                                           "set string elements"
   at: 1 put: $a;
   at: 2 put: $b;
   at: 3 put: $c;
   at: 4 put: $e.
x := String with: $a with: $b with: $c with: $d.            "set up to 4 elements at a time"
x do: [:a | Transcript show: a printString; cr].               "iterate over the string"
b := x conform: [:a | (a >= $a) & (a <= $z)].          "test if all elements meet condition"
y := x select: [:a | a > $a].                       "return all elements that meet condition"
y := x asSymbol.                                               "convert string to symbol"
y := x asArray.                                                "convert string to array"
x := 'ABCD' asByteArray.                                      "convert string to byte array"
y := x asOrderedCollection.                            "convert string to ordered collection"
y := x asSortedCollection.                              "convert string to sorted collection"
y := x asBag.                                             "convert string to bag collection"
y := x asSet.                                             "convert string to set collection"
y := x shuffled.                                               "randomly shuffle string"
```

Array: Fixed length collection

- ByteArray: Array limited to byte elements (0-255)
- WordArray: Array limited to word elements (0-2^32)

```
| b x y sum max |
x := #(4 3 2 1).                                               "constant array"
x := Array with: 5 with: 4 with: 3 with: 2.              "create array with up to 4 elements"
x := Array new: 4.                                    "allocate an array with specified size"
x                                                              "set array elements"
   at: 1 put: 5;
   at: 2 put: 4;
   at: 3 put: 3;
   at: 4 put: 2.
b := x isEmpty.                                                "test if array is empty"
y := x size.                                                   "array size"
y := x at: 4.                                                  "get array element at index"
b := x includes: 3.                                            "test if element is in array"
y := x copyFrom: 2 to: 4.                                      "subarray"
y := x indexOf: 3 ifAbsent: [0].                     "first position of element within array"
y := x occurrencesOf: 3.                               "number of times object in collection"
x do: [:a | Transcript show: a printString; cr].               "iterate over the array"
b := x conform: [:a | (a >= 1) & (a <= 4)].            "test if all elements meet condition"
y := x select: [:a | a > 2].                     "return collection of elements that pass test"
y := x reject: [:a | a < 2].                     "return collection of elements that fail test"
y := x collect: [:a | a + a].                     "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].             "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.                    "sum array elements"
sum := 0. 1 to: (x size) do: [:a | sum := sum + (x at: a)]. "sum array elements"
sum := x inject: 0 into: [:a :c | a + c].                      "sum array elements"
max := x inject: 0 into: [:a :c | (a > c)                      "find max element in array"
   ifTrue: [a]
   ifFalse: [c]].
y := x shuffled.                                               "randomly shuffle collection"
y := x asArray.                                                "convert to array"
"y := x asByteArray."                           "note: this instruction not available on Squeak"
y := x asWordArray.                                            "convert to word array"
y := x asOrderedCollection.                                  "convert to ordered collection"
```

```
y := x asSortedCollection.                              "convert to sorted collection"
y := x asBag.                                            "convert to bag collection"
y := x asSet.                                            "convert to set collection"
```

OrderedCollection: acts like an expandable array

```
| b x y sum max |
x := OrderedCollection with: 4 with: 3 with: 2 with: 1.   "create collection with up to 4 elements"
x := OrderedCollection new.                              "allocate collection"
x add: 3; add: 2; add: 1; add: 4; yourself.              "add element to collection"
y := x addFirst: 5.                             "add element at beginning of collection"
y := x removeFirst.                               "remove first element in collection"
y := x addLast: 6.                                  "add element at end of collection"
y := x removeLast.                                 "remove last element in collection"
y := x addAll: #(7 8 9).                         "add multiple elements to collection"
y := x removeAll: #(7 8 9).                 "remove multiple elements from collection"
x at: 2 put: 3.                                          "set element at index"
y := x remove: 5 ifAbsent: [].                       "remove element from collection"
b := x isEmpty.                                          "test if empty"
y := x size.                                             "number of elements"
y := x at: 2.                                            "retrieve element at index"
y := x first.                                   "retrieve first element in collection"
y := x last.                                     "retrieve last element in collection"
b := x includes: 5.                                 "test if element is in collection"
y := x copyFrom: 2 to: 3.                                "subcollection"
y := x indexOf: 3 ifAbsent: [0].            "first position of element within collection"
y := x occurrencesOf: 3.                         "number of times object in collection"
x do: [:a | Transcript show: a printString; cr].         "iterate over the collection"
b := x conform: [:a | (a >= 1) & (a <= 4)].        "test if all elements meet condition"
y := x select: [:a | a > 2].                "return collection of elements that pass test"
y := x reject: [:a | a < 2].                "return collection of elements that fail test"
y := x collect: [:a | a + a].                "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].      "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.              "sum elements"
sum := 0. 1 to: (x size) do: [:a | sum := sum + (x at: a)]. "sum elements"
sum := x inject: 0 into: [:a :c | a + c].                "sum elements"
max := x inject: 0 into: [:a :c | (a > c)             "find max element in collection"
   ifTrue: [a]
   ifFalse: [c]].
y := x shuffled.                                         "randomly shuffle collection"
y := x asArray.                                          "convert to array"
y := x asOrderedCollection.                             "convert to ordered collection"
y := x asSortedCollection.                              "convert to sorted collection"
y := x asBag.                                            "convert to bag collection"
y := x asSet.                                            "convert to set collection"
```

SortedCollection: like OrderedCollection except order of elements determined by sorting criteria

```
| b x y sum max |
x := SortedCollection with: 4 with: 3 with: 2 with: 1.    "create collection with up to 4 elements"
x := SortedCollection new.                               "allocate collection"
```

```
x := SortedCollection sortBlock: [:a :c | a > c].           "set sort criteria"
x add: 3; add: 2; add: 1; add: 4; yourself.                 "add element to collection"
y := x addFirst: 5.                          "add element at beginning of collection"
y := x removeFirst.                                "remove first element in collection"
y := x addLast: 6.                                   "add element at end of collection"
y := x removeLast.                                  "remove last element in collection"
y := x addAll: #(7 8 9).                          "add multiple elements to collection"
y := x removeAll: #(7 8 9).                  "remove multiple elements from collection"
y := x remove: 5 ifAbsent: [].                          "remove element from collection"
b := x isEmpty.                                             "test if empty"
y := x size.                                                "number of elements"
y := x at: 2.                                               "retrieve element at index"
y := x first.                                    "retrieve first element in collection"
y := x last.                                      "retrieve last element in collection"
b := x includes: 4.                                  "test if element is in collection"
y := x copyFrom: 2 to: 3.                                   "subcollection"
y := x indexOf: 3 ifAbsent: [0].              "first position of element within collection"
y := x occurrencesOf: 3.                         "number of times object in collection"
x do: [:a | Transcript show: a printString; cr].            "iterate over the collection"
b := x conform: [:a | (a >= 1) & (a <= 4)].        "test if all elements meet condition"
y := x select: [:a | a > 2].               "return collection of elements that pass test"
y := x reject: [:a | a < 2].               "return collection of elements that fail test"
y := x collect: [:a | a + a].                 "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].     "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.                 "sum elements"
sum := 0. 1 to: (x size) do: [:a | sum := sum + (x at: a)]. "sum elements"
sum := x inject: 0 into: [:a :c | a + c].                   "sum elements"
max := x inject: 0 into: [:a :c | (a > c)              "find max element in collection"
   ifTrue: [a]
   ifFalse: [c]].
y := x asArray.                                             "convert to array"
y := x asOrderedCollection.                              "convert to ordered collection"
y := x asSortedCollection.                                "convert to sorted collection"
y := x asBag.                                               "convert to bag collection"
y := x asSet.                                               "convert to set collection"
```

Bag: like OrderedCollection except elements are in no particular order

```
| b x y sum max |
x := Bag with: 4 with: 3 with: 2 with: 1.        "create collection with up to 4 elements"
x := Bag new.                                               "allocate collection"
x add: 4; add: 3; add: 1; add: 2; yourself.                 "add element to collection"
x add: 3 withOccurrences: 2.                        "add multiple copies to collection"
y := x addAll: #(7 8 9).                          "add multiple elements to collection"
y := x removeAll: #(7 8 9).                  "remove multiple elements from collection"
y := x remove: 4 ifAbsent: [].                          "remove element from collection"
b := x isEmpty.                                             "test if empty"
y := x size.                                                "number of elements"
b := x includes: 3.                                  "test if element is in collection"
y := x occurrencesOf: 3.                         "number of times object in collection"
x do: [:a | Transcript show: a printString; cr].            "iterate over the collection"
b := x conform: [:a | (a >= 1) & (a <= 4)].        "test if all elements meet condition"
y := x select: [:a | a > 2].               "return collection of elements that pass test"
```

```
y := x reject: [:a | a < 2].                   "return collection of elements that fail test"
y := x collect: [:a | a + a].                    "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].           "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.                 "sum elements"
sum := x inject: 0 into: [:a :c | a + c].                   "sum elements"
max := x inject: 0 into: [:a :c | (a > c)                 "find max element in collection"
   ifTrue: [a]
   ifFalse: [c]].
y := x asOrderedCollection.                                "convert to ordered collection"
y := x asSortedCollection.                                  "convert to sorted collection"
y := x asBag.                                               "convert to bag collection"
y := x asSet.                                               "convert to set collection"
```

Set: like Bag except duplicates not allowed

IdentitySet: uses identity test (== rather than =)

```
| b x y sum max |
x := Set with: 4 with: 3 with: 2 with: 1.          "create collection with up to 4 elements"
x := Set new.                                               "allocate collection"
x add: 4; add: 3; add: 1; add: 2; yourself.                 "add element to collection"
y := x addAll: #(7 8 9).                             "add multiple elements to collection"
y := x removeAll: #(7 8 9).                      "remove multiple elements from collection"
y := x remove: 4 ifAbsent: [].                            "remove element from collection"
b := x isEmpty.                                             "test if empty"
y := x size.                                                "number of elements"
x includes: 4.                                          "test if element is in collection"
x do: [:a | Transcript show: a printString; cr].            "iterate over the collection"
b := x conform: [:a | (a >= 1) & (a <= 4)].          "test if all elements meet condition"
y := x select: [:a | a > 2].                   "return collection of elements that pass test"
y := x reject: [:a | a < 2].                   "return collection of elements that fail test"
y := x collect: [:a | a + a].                    "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].           "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.                 "sum elements"
sum := x inject: 0 into: [:a :c | a + c].                   "sum elements"
max := x inject: 0 into: [:a :c | (a > c)                 "find max element in collection"
   ifTrue: [a]
   ifFalse: [c]].
y := x asArray.                                             "convert to array"
y := x asOrderedCollection.                                "convert to ordered collection"
y := x asSortedCollection.                                  "convert to sorted collection"
y := x asBag.                                               "convert to bag collection"
y := x asSet.                                               "convert to set collection"
```

Interval:

```
| b x y sum max |
x := Interval from: 5 to: 10.                               "create interval object"
x := 5 to: 10.
x := Interval from: 5 to: 10 by: 2.              "create interval object with specified increment"
x := 5 to: 10 by: 2.
b := x isEmpty.                                             "test if empty"
```

```
y := x size.                                                 "number of elements"
x includes: 9.                                           "test if element is in collection"
x do: [:k | Transcript show: k printString; cr].             "iterate over interval"
b := x conform: [:a | (a >= 1) & (a <= 4)].            "test if all elements meet condition"
y := x select: [:a | a > 7].                      "return collection of elements that pass test"
y := x reject: [:a | a < 2].                      "return collection of elements that fail test"
y := x collect: [:a | a + a].                      "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].             "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.                  "sum elements"
sum := 0. 1 to: (x size) do: [:a | sum := sum + (x at: a)]. "sum elements"
sum := x inject: 0 into: [:a :c | a + c].                    "sum elements"
max := x inject: 0 into: [:a :c | (a > c)                  "find max element in collection"
   ifTrue: [a]
   ifFalse: [c]].
y := x asArray.                                              "convert to array"
y := x asOrderedCollection.                                 "convert to ordered collection"
y := x asSortedCollection.                                   "convert to sorted collection"
y := x asBag.                                                "convert to bag collection"
y := x asSet.                                                "convert to set collection"
```

Associations:

```
| x y |
x := #myVar->'hello'.
y := x key.
y := x value.
```

Dictionary:

IdentityDictionary: uses identity test (== rather than =)

```
| b x y |
x := Dictionary new.                                         "allocate collection"
x add: #a->4; add: #b->3; add: #c->1; add: #d->2; yourself. "add element to collection"
x at: #e put: 3.                                             "set element at index"
b := x isEmpty.                                              "test if empty"
y := x size.                                                 "number of elements"
y := x at: #a ifAbsent: [].                                  "retrieve element at index"
y := x keyAtValue: 3 ifAbsent: [].                "retrieve key for given value with error block"
y := x removeKey: #e ifAbsent: [].                          "remove element from collection"
b := x includes: 3.                                "test if element is in values collection"
b := x includesKey: #a.                              "test if element is in keys collection"
y := x occurrencesOf: 3.                              "number of times object in collection"
y := x keys.                                                 "set of keys"
y := x values.                                               "bag of values"
x do: [:a | Transcript show: a printString; cr].           "iterate over the values collection"
x keysDo: [:a | Transcript show: a printString; cr].         "iterate over the keys collection"
x associationsDo: [:a | Transcript show: a printString; cr]."iterate over the associations"
x keysAndValuesDo: [:aKey :aValue | Transcript               "iterate over keys and values"
   show: aKey printString; space;
   show: aValue printString; cr].
b := x conform: [:a | (a >= 1) & (a <= 4)].            "test if all elements meet condition"
```

```
y := x select: [:a | a > 2].                   "return collection of elements that pass test"
y := x reject: [:a | a < 2].                   "return collection of elements that fail test"
y := x collect: [:a | a + a].                    "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].           "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.                  "sum elements"
sum := x inject: 0 into: [:a :c | a + c].                    "sum elements"
max := x inject: 0 into: [:a :c | (a > c)                  "find max element in collection"
   ifTrue: [a]
   ifFalse: [c]].
y := x asArray.                                              "convert to array"
y := x asOrderedCollection.                                 "convert to ordered collection"
y := x asSortedCollection.                                  "convert to sorted collection"
y := x asBag.                                                "convert to bag collection"
y := x asSet.                                                "convert to set collection"

Smalltalk at: #CMRGlobal put: 'CMR entry'.             "put global in Smalltalk Dictionary"
x := Smalltalk at: #CMRGlobal.                      "read global from Smalltalk Dictionary"
Transcript show: (CMRGlobal printString).            "entries are directly accessible by name"
Smalltalk keys do: [ :k |                                    "print out all classes"
   ((Smalltalk at: k) isKindOf: Class)
      ifFalse: [Transcript show: k printString; cr]].
Smalltalk at: #CMRDictionary put: (Dictionary new).        "set up user defined dictionary"
CMRDictionary at: #MyVar1 put: 'hello1'.                     "put entry in dictionary"
CMRDictionary add: #MyVar2->'hello2'.               "add entry to dictionary use key->value combo"
CMRDictionary size.                                          "dictionary size"
CMRDictionary keys do: [ :k |                               "print out keys in dictionary"
   Transcript show: k printString; cr].
CMRDictionary values do: [ :k |                            "print out values in dictionary"
   Transcript show: k printString; cr].
CMRDictionary keysAndValuesDo: [:aKey :aValue |              "print out keys and values"
   Transcript
      show: aKey printString;
      space;
      show: aValue printString;
      cr].
CMRDictionary associationsDo: [:aKeyValue |           "another iterator for printing key values"
   Transcript show: aKeyValue printString; cr].
Smalltalk removeKey: #CMRGlobal ifAbsent: [].         "remove entry from Smalltalk dictionary"
Smalltalk removeKey: #CMRDictionary ifAbsent: [].       "remove user dictionary from Smalltalk dictionary"
```

Internal Stream:

```
| b x ios |
ios := ReadStream on: 'Hello read stream'.
ios := ReadStream on: 'Hello read stream' from: 1 to: 5.
[(x := ios nextLine) notNil]
   whileTrue: [Transcript show: x; cr].
ios position: 3.
ios position.
x := ios next.
x := ios peek.
x := ios contents.
b := ios atEnd.
```

```
ios := ReadWriteStream on: 'Hello read stream'.
ios := ReadWriteStream on: 'Hello read stream' from: 1 to: 5.
ios := ReadWriteStream with: 'Hello read stream'.
ios := ReadWriteStream with: 'Hello read stream' from: 1 to: 10.
ios position: 0.
[(x := ios nextLine) notNil]
   whileTrue: [Transcript show: x; cr].
ios position: 6.
ios position.
ios nextPutAll: 'Chris'.
x := ios next.
x := ios peek.
x := ios contents.
b := ios atEnd.
```

FileStream:

```
| b x ios |
ios := FileStream newFileNamed: 'ios.txt'.
ios nextPut: $H; cr.
ios nextPutAll: 'Hello File'; cr.
'Hello File' printOn: ios.
'Hello File' storeOn: ios.
ios close.

ios := FileStream oldFileNamed: 'ios.txt'.
[(x := ios nextLine) notNil]
   whileTrue: [Transcript show: x; cr].
ios position: 3.
x := ios position.
x := ios next.
x := ios peek.
b := ios atEnd.
ios close.
```

Date:

```
| x y |
x := Date today.                                            "create date for today"
x := Date dateAndTimeNow.                             "create date from current time/date"
x := Date readFromString: '01/02/1999'.                "create date from formatted string"
x := Date newDay: 12 month: #July year: 1999                "create date from parts"
x := Date fromDays: 36000.                     "create date from elapsed days since 1/1/1901"
y := Date dayOfWeek: #Monday.                               "day of week as int (1-7)"
y := Date indexOfMonth: #January.                           "month of year as int (1-12)"
y := Date daysInMonth: 2 forYear: 1996.                     "day of month as int (1-31)"
y := Date daysInYear: 1996.                                 "days in year (365|366)"
y := Date nameOfDay: 1                                      "weekday name (#Monday,...)"
y := Date nameOfMonth: 1.                                   "month name (#January,...)"
y := Date leapYear: 1996.                              "1 if leap year; 0 if not leap year"
y := x weekday.                                             "day of week (#Monday,...)"
y := x previous: #Monday.                                  "date for previous day of week"
```

```
y := x dayOfMonth.                                           "day of month (1-31)"
y := x day.                                                  "day of year (1-366)"
y := x firstDayOfMonth.                                "day of year for first day of month"
y := x monthName.                                           "month of year (#January,...)"
y := x monthIndex.                                           "month of year (1-12)"
y := x daysInMonth.                                          "days in month (1-31)"
y := x year.                                                 "year (19xx)"
y := x daysInYear.                                           "days in year (365|366)"
y := x daysLeftInYear.                                       "days left in year (364|365)"
y := x asSeconds.                                         "seconds elapsed since 1/1/1901"
y := x addDays: 10.                                          "add days to date object"
y := x subtractDays: 10.                                    "subtract days to date object"
y := x subtractDate: (Date today).                        "subtract date (result in days)"
y := x printFormat: #(2 1 3 $/ 1 1).                         "print formatted date"
b := (x <= Date today).                                      "comparison"
```

Time:

```
| x y |
x := Time now.                                             "create time from current time"
x := Time dateAndTimeNow.                              "create time from current time/date"
x := Time readFromString: '3:47:26 pm'.                 "create time from formatted string"
x := Time fromSeconds: (60 * 60 * 4).          "create time from elapsed time from midnight"
y := Time millisecondClockValue.                             "milliseconds since midnight"
y := Time totalSeconds.                                     "total seconds since 1/1/1901"
y := x seconds.                                              "seconds past minute (0-59)"
y := x minutes.                                              "minutes past hour (0-59)"
y := x hours.                                                "hours past midnight (0-23)"
y := x addTime: (Time now).                                  "add time to time object"
y := x subtractTime: (Time now).                            "subtract time to time object"
y := x asSeconds.                                            "convert time to seconds"
x := Time millisecondsToRun: [                               "timing facility"
   1 to: 1000 do: [:index | y := 3.14 * index]].
b := (x <= Time now).                                        "comparison"
```

Point:

```
| x y |
x := 200@100.                                                "obtain a new point"
y := x x.                                                    "x coordinate"
y := x y.                                                    "y coordinate"
x := 200@100 negated.                                        "negates x and y"
x := (-200@-100) abs.                                        "absolute value of x and y"
x := (200.5@100.5) rounded.                                  "round x and y"
x := (200.5@100.5) truncated.                                "truncate x and y"
x := 200@100 + 100.                                          "add scale to both x and y"
x := 200@100 - 100.                                      "subtract scale from both x and y"
x := 200@100 * 2.                                            "multiply x and y by scale"
x := 200@100 / 2.                                            "divide x and y by scale"
x := 200@100 // 2.                                           "divide x and y by scale"
x := 200@100 \\ 3.                                          "remainder of x and y by scale"
x := 200@100 + 50@25.                                        "add points"
x := 200@100 - 50@25.                                        "subtract points"
```

```
x := 200@100 * 3@4.                                         "multiply points"
x := 200@100 // 3@4.                                        "divide points"
x := 200@100 max: 50@200.                                   "max x and y"
x := 200@100 min: 50@200.                                   "min x and y"
x := 20@5 dotProduct: 10@2.                               "sum of product (x1*x2 + y1*y2)"
```

Rectangle:

```
Rectangle fromUser.
```

Pen:

```
| myPen |
Display restoreAfter: [
   Display fillWhite.

myPen := Pen new.                                           "get graphic pen"
myPen squareNib: 1.
myPen color: (Color blue).                                  "set pen color"
myPen home.                                            "position pen at center of display"
myPen up.                                                   "makes nib unable to draw"
myPen down.                                                 "enable the nib to draw"
myPen north.                                               "points direction towards top"
myPen turn: -180.                                     "add specified degrees to direction"
myPen direction.                                            "get current angle of pen"
myPen go: 50.                                        "move pen specified number of pixels"
myPen location.                                             "get the pen position"
myPen goto: 200@200.                                        "move to specified point"
myPen place: 250@250.                             "move to specified point without drawing"
myPen print: 'Hello World' withFont: (TextStyle default fontAt: 1).
Display extent.                                             "get display width@height"
Display width.                                              "get display width"
Display height.                                             "get display height"

].
```

Dynamic Message Calling/Compiling:

```
| receiver message result argument keyword1 keyword2 argument1 argument2 |
"unary message"
receiver := 5.
message := 'factorial' asSymbol.
result := receiver perform: message.
result := Compiler evaluate: ((receiver storeString), ' ', message).
result := (Message new setSelector: message arguments: #()) sentTo: receiver.

"binary message"
receiver := 1.
message := '+' asSymbol.
argument := 2.
result := receiver perform: message withArguments: (Array with: argument).
result := Compiler evaluate: ((receiver storeString), ' ', message, ' ', (argument storeString)).
result := (Message new setSelector: message arguments: (Array with: argument)) sentTo: receiver.

"keyword messages"
receiver := 12.
keyword1 := 'between:' asSymbol.
keyword2 := 'and:' asSymbol.
argument1 := 10.
argument2 := 20.
result := receiver
   perform: (keyword1, keyword2) asSymbol
   withArguments: (Array with: argument1 with: argument2).
result := Compiler evaluate:
  ((receiver storeString), ' ', keyword1, (argument1 storeString) , ' ', keyword2, (argument2 storeString))
result := (Message
   new
      setSelector: (keyword1, keyword2) asSymbol
      arguments: (Array with: argument1 with: argument2))
   sentTo: receiver.
```

Class/Meta-class:

```
| b x |
x := String name.                                           "class name"
x := String category.                                       "organization category"
x := String comment.                                        "class comment"
x := String kindOfSubclass.                     "subclass type - subclass: variableSubclass, etc"
x := String definition.                                     "class definition"
x := String instVarNames.                              "immediate instance variable names"
x := String allInstVarNames.                         "accumulated instance variable names"
x := String classVarNames.                                "immediate class variable names"
x := String allClassVarNames.                           "accumulated class variable names"
x := String sharedPools.                       "immediate dictionaries used as shared pools"
x := String allSharedPools.                     "accumulated dictionaries used as shared pools"
x := String selectors.                                      "message selectors for class"
x := String sourceCodeAt: #size.                        "source code for specified method"
x := String allInstances.                            "collection of all instances of class"
x := String superclass.                                     "immediate superclass"
x := String allSuperclasses.                                "accumulated superclasses"
x := String withAllSuperclasses.                  "receiver class and accumulated superclasses"
x := String subclasses.                                     "immediate subclasses"
x := String allSubclasses.                                  "accumulated subclasses"
x := String withAllSubclasses.                   "receiver class and accumulated subclasses"
b := String instSize.                                 "number of named instance variables"
b := String isFixed.                                "true if no indexed instance variables"
b := String isVariable.                            "true if has indexed instance variables"
b := String isPointers.                        "true if index instance vars contain objects"
b := String isBits.                          "true if index instance vars contain bytes/words"
b := String isBytes.                            "true if index instance vars contain bytes"
b := String isWords.                             true if index instance vars contain words"
Object withAllSubclasses size.                         "get total number of class entries"
```

Debuging:

```
| a b x |
x yourself.                                                 "returns receiver"
String browse.                                              "browse specified class"
x inspect.                                                  "open object inspector window"
x confirm: 'Is this correct?'.
x halt.                                               "breakpoint to open debugger window"
x halt: 'Halt message'.
x notify: 'Notify text'.
x error: 'Error string'.                                 "open up error window with title"
x doesNotUnderstand: #cmrMessage.                           "flag message is not handled"
x shouldNotImplement.                               "flag message should not be implemented"
x subclassResponsibility.                                   "flag message as abstract"
x errorImproperStore.                           "flag an improper store into indexable object"
x errorNonIntegerIndex.                          "flag only integers should be used as index"
x errorSubscriptBounds.                                     "flag subscript out of bounds"
x primitiveFailed.                                          "system primitive failed"

a := 'A1'. b := 'B2'. a become: b.                          "switch two objects"
Transcript show: a, b; cr.
```

Misc

```
| x |
"Smalltalk condenseChanges."                                "compress the change file"
x := FillInTheBlank request: 'Prompt Me'.                   "prompt user for input"
Utilities openCommandKeyHelp
```

Ready For More?

Free Online

- GNU Smalltalk User's Guide
- smalltalk dot org
- Computer Programming using GNU Smalltalk
- Smalltalk Cheatsheet
- Smalltalk-72 Manual
- BYTE: A Special issue on Smalltalk
- Smalltalk, Objects, and Design
- Smalltalk: An Introduction to Application Development Using VisualWorks

Standard-Ml

Standard ML is a functional programming language with type inference and some side-effects. Some of the hard parts of learning Standard ML are: Recursion, pattern matching, type inference (guessing the right types but never allowing implicit type conversion). Standard ML is distinguished from Haskell by including references, allowing variables to be updated.

```
(* Comments in Standard ML begin with (* and end with *).  Comments can be
   nested which means that all (* tags must end with a *) tag.  This comment,
   for example, contains two nested comments. *)
```

```
(* A Standard ML program consists of declarations, e.g. value declarations: *)
val rent = 1200
val phone_no = 5551337
val pi = 3.14159
val negative_number = ~15  (* Yeah, unary minus uses the 'tilde' symbol *)

(* And just as importantly, functions: *)
fun is_large(x : int) = if x > 37 then true else false

(* Floating-point numbers are called "reals". *)
val tau = 2.0 * pi         (* You can multiply two reals *)
val twice_rent = 2 * rent  (* You can multiply two ints *)
(* val meh = 1.25 * 10 *)  (* But you can't multiply an int and a real *)

(* +, - and * are overloaded so they work for both int and real. *)
(* The same cannot be said for division which has separate operators: *)
val real_division = 14.0 / 4.0  (* gives 3.5 *)
val int_division  = 14 div 4    (* gives 3, rounding down *)
val int_remainder = 14 mod 4    (* gives 2, since 3*4 = 12 *)

(* ~ is actually sometimes a function (e.g. when put in front of variables) *)
val negative_rent = ~(rent)  (* Would also have worked if rent were a "real" *)

(* There are also booleans and boolean operators *)
val got_milk = true
val got_bread = false
val has_breakfast = got_milk andalso got_bread  (* 'andalso' is the operator *)
val has_something = got_milk orelse got_bread   (* 'orelse' is the operator *)
val is_sad = not(has_something)                 (* not is a function *)

(* Many values can be compared using equality operators: = and <> *)
val pays_same_rent = (rent = 1300)  (* false *)
val is_wrong_phone_no = (phone_no <> 5551337)  (* false *)

(* The operator <> is what most other languages call !=. *)
(* 'andalso' and 'orelse' are called && and || in many other languages. *)

(* Actually, most of the parentheses above are unnecessary.  Here are some
   different ways to say some of the things mentioned above: *)
fun is_large x = x > 37  (* The parens above were necessary because of ': int' *)
val is_sad = not has_something
val pays_same_rent = rent = 1300  (* Looks confusing, but works *)
val is_wrong_phone_no = phone_no <> 5551337
val negative_rent = ~rent  (* ~ rent (notice the space) would also work *)

(* Parentheses are mostly necessary when grouping things: *)
val some_answer = is_large (5 + 5)      (* Without parens, this would break! *)
(* val some_answer = is_large 5 + 5 *)  (* Read as: (is_large 5) + 5. Bad! *)


(* Besides booleans, ints and reals, Standard ML also has chars and strings: *)
val foo = "Hello, World!\n"  (* The \n is the escape sequence for linebreaks *)
val one_letter = #"a"        (* That funky syntax is just one character, a *)
val combined = "Hello " ^ "there, " ^ "fellow!\n"  (* Concatenate strings *)

val _ = print foo       (* You can print things. We are not interested in the *)
val _ = print combined  (* result of this computation, so we throw it away. *)
(* val _ = print one_letter *)  (* Only strings can be printed this way *)


val bar = [ #"H", #"e", #"l", #"l", #"o" ]  (* SML also has lists! *)
(* val _ = print bar *)  (* Lists are unfortunately not the same as strings *)

(* Fortunately they can be converted.  String is a library and implode and size
   are functions available in that library that take strings as argument. *)
val bob = String.implode bar          (* gives "Hello" *)
val bob_char_count = String.size bob  (* gives 5 *)
val _ = print (bob ^ "\n")            (* For good measure, add a linebreak *)

(* You can have lists of any kind *)
val numbers = [1, 3, 3, 7, 229, 230, 248]  (* : int list *)
val names = [ "Fred", "Jane", "Alice" ]    (* : string list *)

(* Even lists of lists of things *)
val groups = [ [ "Alice", "Bob" ],
               [ "Huey", "Dewey", "Louie" ],
               [ "Bonnie", "Clyde" ] ]     (* : string list list *)

val number_count = List.length numbers     (* gives 7 *)

(* You can put single values in front of lists of the same kind using
   the :: operator, called "the cons operator" (known from Lisp). *)
val more_numbers = 13 :: numbers  (* gives [13, 1, 3, 3, 7, ...] *)
val more_groups  = ["Batman","Superman"] :: groups

(* Lists of the same kind can be appended using the @ ("append") operator *)
val guest_list = [ "Mom", "Dad" ] @ [ "Aunt", "Uncle" ]

(* This could have been done with the "cons" operator.  It is tricky because the
   left-hand-side must be an element whereas the right-hand-side must be a list
   of those elements. *)
val guest_list = "Mom" :: "Dad" :: [ "Aunt", "Uncle" ]
val guest_list = "Mom" :: ("Dad" :: ("Aunt" :: ("Uncle" :: [])))

(* If you have many lists of the same kind, you can concatenate them all *)
val everyone = List.concat groups  (* [ "Alice", "Bob", "Huey", ... ] *)

(* A list can contain any (finite) number of values *)
val lots = [ 5, 5, 5, 6, 4, 5, 6, 5, 4, 5, 7, 3 ]  (* still just an int list *)

(* Lists can only contain one kind of thing... *)
(* val bad_list = [ 1, "Hello", 3.14159 ] : ??? list *)


(* Tuples, on the other hand, can contain a fixed number of different things *)
val person1 = ("Simon", 28, 3.14159)  (* : string * int * real *)
```

```
(* You can even have tuples inside lists and lists inside tuples *)
val likes = [ ("Alice", "ice cream"),
              ("Bob",   "hot dogs"),
              ("Bob",   "Alice") ]     (* : (string * string) list *)

val mixup = [ ("Alice", 39),
              ("Bob",   37),
              ("Eve",   41) ]  (* : (string * int) list *)

val good_bad_stuff =
  (["ice cream", "hot dogs", "chocolate"],
   ["liver", "paying the rent" ])           (* : string list * string list *)


(* Records are tuples with named slots *)

val rgb = { r=0.23, g=0.56, b=0.91 } (* : {b:real, g:real, r:real} *)

(* You don't need to declare their slots ahead of time. Records with
   different slot names are considered different types, even if their
   slot value types match up. For instance... *)

val Hsl = { H=310.3, s=0.51, l=0.23 } (* : {H:real, l:real, s:real} *)
val Hsv = { H=310.3, s=0.51, v=0.23 } (* : {H:real, s:real, v:real} *)

(* ...trying to evaluate `Hsv = Hsl` or `rgb = Hsl` would give a type
   error. While they're all three-slot records composed only of `real`s,
   they each have different names for at least some slots. *)

(* You can use hash notation to get values out of tuples. *)

val H = #H Hsv (* : real *)
val s = #s Hsl (* : real *)

(* Functions! *)
fun add_them (a, b) = a + b    (* A simple function that adds two numbers *)
val test_it = add_them (3, 4)  (* gives 7 *)

(* Larger functions are usually broken into several lines for readability *)
fun thermometer temp =
    if temp < 37
    then "Cold"
    else if temp > 37
         then "Warm"
         else "Normal"

val test_thermo = thermometer 40  (* gives "Warm" *)

(* if-sentences are actually expressions and not statements/declarations.
   A function body can only contain one expression.  There are some tricks
   for making a function do more than just one thing, though. *)

(* A function can call itself as part of its result (recursion!) *)
```

```
fun fibonacci n =
    if n = 0 then 0 else                   (* Base case *)
    if n = 1 then 1 else                   (* Base case *)
    fibonacci (n - 1) + fibonacci (n - 2)  (* Recursive case *)

(* Sometimes recursion is best understood by evaluating a function by hand:

 fibonacci 4
   ~> fibonacci (4 - 1) + fibonacci (4 - 2)
   ~> fibonacci 3 + fibonacci 2
   ~> (fibonacci (3 - 1) + fibonacci (3 - 2)) + fibonacci 2
   ~> (fibonacci 2 + fibonacci 1) + fibonacci 2
   ~> ((fibonacci (2 - 1) + fibonacci (2 - 2)) + fibonacci 1) + fibonacci 2
   ~> ((fibonacci 1 + fibonacci 0) + fibonacci 1) + fibonacci 2
   ~> ((1 + fibonacci 0) + fibonacci 1) + fibonacci 2
   ~> ((1 + 0) + fibonacci 1) + fibonacci 2
   ~> (1 + fibonacci 1) + fibonacci 2
   ~> (1 + 1) + fibonacci 2
   ~> 2 + fibonacci 2
   ~> 2 + (fibonacci (2 - 1) + fibonacci (2 - 2))
   ~> 2 + (fibonacci (2 - 1) + fibonacci (2 - 2))
   ~> 2 + (fibonacci 1 + fibonacci 0)
   ~> 2 + (1 + fibonacci 0)
   ~> 2 + (1 + 0)
   ~> 2 + 1
   ~> 3  which is the 4th Fibonacci number, according to this definition

 *)

(* A function cannot change the variables it can refer to.  It can only
   temporarily shadow them with new variables that have the same names.  In this
   sense, variables are really constants and only behave like variables when
   dealing with recursion.  For this reason, variables are also called value
   bindings. An example of this: *)

val x = 42
fun answer(question) =
    if question = "What is the meaning of life, the universe and everything?"
    then x
    else raise Fail "I'm an exception. Also, I don't know what the answer is."
val x = 43
val hmm = answer "What is the meaning of life, the universe and everything?"
(* Now, hmm has the value 42.  This is because the function answer refers to
   the copy of x that was visible before its own function definition. *)


(* Functions can take several arguments by taking one tuples as argument: *)
fun solve2 (a : real, b : real, c : real) =
    ( (~b + Math.sqrt(b * b - 4.0*a*c)) / (2.0 * a),
      (~b - Math.sqrt(b * b - 4.0*a*c)) / (2.0 * a) )

(* Sometimes, the same computation is carried out several times. It makes sense
   to save and re-use the result the first time. We can use "let-bindings": *)
fun solve2 (a : real, b : real, c : real) =
    let val discr  = b * b - 4.0*a*c
        val sqr = Math.sqrt discr
        val denom = 2.0 * a
    in ((~b + sqr) / denom,
        (~b - sqr) / denom) end


(* Pattern matching is a funky part of functional programming.  It is an
   alternative to if-sentences.  The fibonacci function can be rewritten: *)
fun fibonacci 0 = 0  (* Base case *)
  | fibonacci 1 = 1  (* Base case *)
  | fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)  (* Recursive case *)

(* Pattern matching is also possible on composite types like tuples, lists and
   records. Writing "fun solve2 (a, b, c) = ..." is in fact a pattern match on
   the one three-tuple solve2 takes as argument. Similarly, but less intuitively,
   you can match on a list consisting of elements in it (from the beginning of
   the list only). *)
fun first_elem (x::xs) = x
fun second_elem (x::y::xs) = y
fun evenly_positioned_elems (odd::even::xs) = even::evenly_positioned_elems xs
  | evenly_positioned_elems [odd] = []  (* Base case: throw away *)
  | evenly_positioned_elems []    = []  (* Base case *)

(* When matching on records, you must use their slot names, and you must bind
   every slot in a record. The order of the slots doesn't matter though. *)

fun rgbToTup {r, g, b} = (r, g, b)    (* fn : {b:'a, g:'b, r:'c} -> 'c * 'b * 'a *)
fun mixRgbToTup {g, b, r} = (r, g, b) (* fn : {b:'a, g:'b, r:'c} -> 'c * 'b * 'a *)

(* If called with {r=0.1, g=0.2, b=0.3}, either of the above functions
   would return (0.1, 0.2, 0.3). But it would be a type error to call them
   with {r=0.1, g=0.2, b=0.3, a=0.4} *)

(* Higher order functions: Functions can take other functions as arguments.
   Functions are just other kinds of values, and functions don't need names
   to exist.  Functions without names are called "anonymous functions" or
   lambda expressions or closures (since they also have a lexical scope). *)
val is_large = (fn x => x > 37)
val add_them = fn (a,b) => a + b
val thermometer =
    fn temp => if temp < 37
               then "Cold"
               else if temp > 37
                    then "Warm"
                    else "Normal"

(* The following uses an anonymous function directly and gives "ColdWarm" *)
val some_result = (fn x => thermometer (x - 5) ^ thermometer (x + 5)) 37

(* Here is a higher-order function that works on lists (a list combinator) *)
val readings = [ 34, 39, 37, 38, 35, 36, 37, 37, 37 ]  (* first an int list *)
val opinions = List.map thermometer readings (* gives [ "Cold", "Warm", ... ] *)
```

```
(* And here is another one for filtering lists *)
val warm_readings = List.filter is_large readings  (* gives [39, 38] *)

(* You can create your own higher-order functions, too.  Functions can also take
   several arguments by "currying" them. Syntax-wise this means adding spaces
   between function arguments instead of commas and surrounding parentheses. *)
fun map f [] = []
  | map f (x::xs) = f(x) :: map f xs

(* map has type ('a -> 'b) -> 'a list -> 'b list and is called polymorphic. *)
(* 'a is called a type variable. *)


(* We can declare functions as infix *)
val plus = add_them   (* plus is now equal to the same function as add_them *)
infix plus            (* plus is now an infix operator *)
val seven = 2 plus 5  (* seven is now bound to 7 *)

(* Functions can also be made infix before they are declared *)
infix minus
fun x minus y = x - y (* It becomes a little hard to see what's the argument *)
val four = 8 minus 4  (* four is now bound to 4 *)

(* An infix function/operator can be made prefix with 'op' *)
val n = op + (5, 5)   (* n is now 10 *)

(* 'op' is useful when combined with high order functions because they expect
   functions and not operators as arguments. Most operators are really just
   infix functions. *)
val sum_of_numbers = foldl op+ 0 [1,2,3,4,5]


(* Datatypes are useful for creating both simple and complex structures *)
datatype color = Red | Green | Blue

(* Here is a function that takes one of these as argument *)
fun say(col) =
    if col = Red then "You are red!" else
    if col = Green then "You are green!" else
    if col = Blue then "You are blue!" else
    raise Fail "Unknown color"

val _ = print (say(Red) ^ "\n")

(* Datatypes are very often used in combination with pattern matching *)
fun say Red   = "You are red!"
  | say Green = "You are green!"
  | say Blue  = "You are blue!"
  | say _     = raise Fail "Unknown color"


(* Here is a binary tree datatype *)
datatype 'a btree = Leaf of 'a
                  | Node of 'a btree * 'a * 'a btree (* three-arg constructor *)
```

```
(* Here is a binary tree *)
val myTree = Node (Leaf 9, 8, Node (Leaf 3, 5, Leaf 7))

(* Drawing it, it might look something like...

           8
          / \
 leaf -> 9   5
            / \
   leaf -> 3   7 <- leaf
 *)

(* This function counts the sum of all the elements in a tree *)
fun count (Leaf n) = n
  | count (Node (leftTree, n, rightTree)) = count leftTree + n + count rightTree

val myTreeCount = count myTree  (* myTreeCount is now bound to 32 *)


(* Exceptions! *)
(* Exceptions can be raised/thrown using the reserved word 'raise' *)
fun calculate_interest(n) = if n < 0.0
                            then raise Domain
                            else n * 1.04

(* Exceptions can be caught using "handle" *)
val balance = calculate_interest ~180.0
              handle Domain => ~180.0    (* x now has the value ~180.0 *)

(* Some exceptions carry extra information with them *)
(* Here are some examples of built-in exceptions *)
fun failing_function []    = raise Empty  (* used for empty lists *)
  | failing_function [x]   = raise Fail "This list is too short!"
  | failing_function [x,y] = raise Overflow  (* used for arithmetic *)
  | failing_function xs    = raise Fail "This list is too long!"

(* We can pattern match in 'handle' to make sure
   a specfic exception was raised, or grab the message *)
val err_msg = failing_function [1,2] handle Fail _ => "Fail was raised"
                                          | Domain => "Domain was raised"
                                          | Empty  => "Empty was raised"
                                          | _      => "Unknown exception"

(* err_msg now has the value "Unknown exception" because Overflow isn't
   listed as one of the patterns -- thus, the catch-all pattern _ is used. *)

(* We can define our own exceptions like this *)
exception MyException
exception MyExceptionWithMessage of string
exception SyntaxError of string * (int * int)

(* File I/O! *)
(* Write a nice poem to a file *)
```

```
fun writePoem(filename) =
    let val file = TextIO.openOut(filename)
        val _ = TextIO.output(file, "Roses are red,\nViolets are blue.\n")
        val _ = TextIO.output(file, "I have a gun.\nGet in the van.\n")
    in TextIO.closeOut(file) end

(* Read a nice poem from a file into a list of strings *)
fun readPoem(filename) =
    let val file = TextIO.openIn filename
        val poem = TextIO.inputAll file
        val _ = TextIO.closeIn file
    in String.tokens (fn c => c = #"\n") poem
    end

val _ = writePoem "roses.txt"
val test_poem = readPoem "roses.txt"  (* gives [ "Roses are red,",
                                                 "Violets are blue.",
                                                 "I have a gun.",
                                                 "Get in the van." ] *)

(* We can create references to data which can be updated *)
val counter = ref 0 (* Produce a reference with the ref function *)

(* Assign to a reference with the assignment operator *)
fun set_five reference = reference := 5

(* Read a reference with the dereference operator *)
fun equals_five reference = !reference = 5

(* We can use while loops for when recursion is messy *)
fun decrement_to_zero r = if !r < 0
                          then r := 0
                          else while !r >= 0 do r := !r - 1

(* This returns the unit value (in practical terms, nothing, a 0-tuple) *)

(* To allow returning a value, we can use the semicolon to sequence evaluations *)
fun decrement_ret x y = (x := !x - 1; y)
```

Further learning

- Install an interactive compiler (REPL), for example Poly/ML, Moscow ML, SML/NJ.
- Follow the Coursera course Programming Languages.
- Get the book *ML for the Working Programmer* by Larry C. Paulson.
- Use StackOverflow's sml tag.

Swift

Swift is a programming language for iOS and OS X development created by Apple. Designed to coexist with Objective-C and to be more resilient against erroneous code, Swift was introduced in 2014 at Apple's developer conference WWDC. It is built with the LLVM compiler included in Xcode 6+.

The official Swift Programming Language book from Apple is now available via iBooks.

See also Apple's getting started guide, which has a complete tutorial on Swift.

```
// import a module
import UIKit

//
// MARK: Basics
//

// Xcode supports landmarks to annotate your code and lists them in the jump bar
// MARK: Section mark
// MARK: - Section mark with a separator line
// TODO: Do something soon
// FIXME: Fix this code

// In Swift 2, println and print were combined into one print method. Print automatically appends a new line.
print("Hello, world") // println is now print
print("Hello, world", terminator: "") // printing without appending a newline

// variables (var) value can change after being set
// constants (let) value can NOT be changed after being set

var myVariable = 42
let ø Ω = "value" // unicode variable names
let   = 3.1415926
let convenience = "keyword" // contextual variable name
let weak = "keyword"; let override = "another keyword" // statements can be separated by a semi-colon
let `class` = "keyword" // backticks allow keywords to be used as variable names
let explicitDouble: Double = 70
let intValue = 0007 // 7
let largeIntValue = 77_000 // 77000
let label = "some text " + String(myVariable) // String construction
let piText = "Pi = \( ), Pi 2 = \(  * 2)" // String interpolation

// Build Specific values
// uses -D build configuration
#if false
    print("Not printed")
    let buildValue = 3
#else
    let buildValue = 7
#endif
print("Build value: \(buildValue)") // Build value: 7

/*
Optionals are a Swift language feature that either contains a value,
or contains nil (no value) to indicate that a value is missing.
A question mark (?) after the type marks the value as optional.

Because Swift requires every property to have a value, even nil must be
explicitly stored as an Optional value.

Optional<T> is an enum.
```

```
*/
var someOptionalString: String? = "optional" // Can be nil
// same as above, but ? is a postfix operator (syntax candy)
var someOptionalString2: Optional<String> = "optional"

if someOptionalString != nil {
    // I am not nil
    if someOptionalString!.hasPrefix("opt") {
        print("has the prefix")
    }

    let empty = someOptionalString?.isEmpty
}
someOptionalString = nil

/*
Trying to use ! to access a non-existent optional value triggers a runtime
error. Always make sure that an optional contains a non-nil value before
using ! to force-unwrap its value.
*/

// implicitly unwrapped optional
var unwrappedString: String! = "Value is expected."
// same as above, but ! is a postfix operator (more syntax candy)
var unwrappedString2: ImplicitlyUnwrappedOptional<String> = "Value is expected."

if let someOptionalStringConstant = someOptionalString {
    // has `Some` value, non-nil
    if !someOptionalStringConstant.hasPrefix("ok") {
        // does not have the prefix
    }
}

// Swift has support for storing a value of any type.
// AnyObject == id
// Unlike Objective-C `id`, AnyObject works with any value (Class, Int, struct, etc.)
var anyObjectVar: AnyObject = 7
anyObjectVar = "Changed value to a string, not good practice, but possible."

/*
    Comment here

    /*
        Nested comments are also supported
    */
*/

//
// MARK: Collections
//

/*
Array and Dictionary types are structs. So `let` and `var` also indicate
that they are mutable (var) or immutable (let) when declaring these types.
```

```
*/

// Array
var shoppingList = ["catfish", "water", "lemons"]
shoppingList[1] = "bottle of water"
let emptyArray = [String]() // let == immutable
let emptyArray2 = Array<String>() // same as above
var emptyMutableArray = [String]() // var == mutable
var explicitEmptyMutableStringArray: [String] = [] // same as above


// Dictionary
var occupations = [
    "Malcolm": "Captain",
    "kaylee": "Mechanic"
]
occupations["Jayne"] = "Public Relations"
let emptyDictionary = [String: Float]() // let == immutable
let emptyDictionary2 = Dictionary<String, Float>() // same as above
var emptyMutableDictionary = [String: Float]() // var == mutable
var explicitEmptyMutableDictionary: [String: Float] = [:] // same as above


//
// MARK: Control Flow
//

// Condition statements support "where" clauses, which can be used
// to help provide conditions on optional values.
// Both the assignment and the "where" clause must pass.
let someNumber = Optional<Int>(7)
if let num = someNumber where num > 3 {
    print("num is greater than 3")
}

// for loop (array)
let myArray = [1, 1, 2, 3, 5]
for value in myArray {
    if value == 1 {
        print("One!")
    } else {
        print("Not one!")
    }
}

// for loop (dictionary)
var dict = ["one": 1, "two": 2]
for (key, value) in dict {
    print("\(key): \(value)")
}

// for loop (range)
for i in -1...shoppingList.count {
    print(i)
}
shoppingList[1...2] = ["steak", "peacons"]
// use ..< to exclude the last number

// while loop
var i = 1
while i < 1000 {
    i *= 2
}

// repeat-while loop
repeat {
    print("hello")
} while 1 == 2

// Switch
// Very powerful, think `if` statements with syntax candy
// They support String, object instances, and primitives (Int, Double, etc)
let vegetable = "red pepper"
switch vegetable {
case "celery":
    let vegetableComment = "Add some raisins and make ants on a log."
case "cucumber", "watercress":
    let vegetableComment = "That would make a good tea sandwich."
case let localScopeValue where localScopeValue.hasSuffix("pepper"):
    let vegetableComment = "Is it a spicy \(localScopeValue)?"
default: // required (in order to cover all possible input)
    let vegetableComment = "Everything tastes good in soup."
}

//
// MARK: Functions
//

// Functions are a first-class type, meaning they can be nested
// in functions and can be passed around

// Function with Swift header docs (format as Swift-modified Markdown syntax)

/**
A greet operation

- A bullet in docs
- Another bullet in the docs

- Parameter name    : A name
- Parameter day : A day
- Returns : A string containing the name and day value.
*/
func greet(name: String, day: String) -> String {
    return "Hello \(name), today is \(day)."
}
greet("Bob", day: "Tuesday")
```

```
// similar to above except for the function parameter behaviors
func greet2(requiredName requiredName: String, externalParamName localParamName: String) -> String {
    return "Hello \(requiredName), the day is \(localParamName)"
}
greet2(requiredName: "John", externalParamName: "Sunday")

// Function that returns multiple items in a tuple
func getGasPrices() -> (Double, Double, Double) {
    return (3.59, 3.69, 3.79)
}
let pricesTuple = getGasPrices()
let price = pricesTuple.2 // 3.79
// Ignore Tuple (or other) values by using _ (underscore)
let (_, price1, _) = pricesTuple // price1 == 3.69
print(price1 == pricesTuple.1) // true
print("Gas price: \(price)")

// Labeled/named tuple params
func getGasPrices2() -> (lowestPrice: Double, highestPrice: Double, midPrice: Double) {
    return (1.77, 37.70, 7.37)
}
let pricesTuple2 = getGasPrices2()
let price2 = pricesTuple2.lowestPrice
let (_, price3, _) = pricesTuple2
print(pricesTuple2.highestPrice == pricesTuple2.1) // true
print("Highest gas price: \(pricesTuple2.highestPrice)")

// guard statements
func testGuard() {
   // guards provide early exits or breaks, placing the error handler code near the conditions.
    // it places variables it declares in the same scope as the guard statement.
    guard let aNumber = Optional<Int>(7) else {
        return
    }

    print("number is \(aNumber)")
}
testGuard()

// Variadic Args
func setup(numbers: Int...) {
    // its an array
    let _ = numbers[0]
    let _ = numbers.count
}

// Passing and returning functions
func makeIncrementer() -> (Int -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
```

```
increment(7)

// pass by ref
func swapTwoInts(inout a: Int, inout b: Int) {
    let tempA = a
    a = b
    b = tempA
}
var someIntA = 7
var someIntB = 3
swapTwoInts(&someIntA, b: &someIntB)
print(someIntB) // 7


//
// MARK: Closures
//
var numbers = [1, 2, 6]

// Functions are special case closures ({})

// Closure example.
// `->` separates the arguments and return type
// `in` separates the closure header from the closure body
numbers.map({
    (number: Int) -> Int in
    let result = 3 * number
    return result
})

// When the type is known, like above, we can do this
numbers = numbers.map({ number in 3 * number })
// Or even this
//numbers = numbers.map({ $0 * 3 })

print(numbers) // [3, 6, 18]

// Trailing closure
numbers = numbers.sort { $0 > $1 }

print(numbers) // [18, 6, 3]

//
// MARK: Structures
//

// Structures and classes have very similar capabilities
struct NamesTable {
    let names: [String]

    // Custom subscript
    subscript(index: Int) -> String {
        return names[index]
    }
```

```
}

// Structures have an auto-generated (implicit) designated initializer
let namesTable = NamesTable(names: ["Me", "Them"])
let name = namesTable[1]
print("Name is \(name)") // Name is Them

//
// MARK: Error Handling
//

// The `ErrorType` protocol is used when throwing errors to catch
enum MyError: ErrorType {
    case BadValue(msg: String)
    case ReallyBadValue(msg: String)
}

// functions marked with `throws` must be called using `try`
func fakeFetch(value: Int) throws -> String {
    guard 7 == value else {
        throw MyError.ReallyBadValue(msg: "Some really bad value")
    }

    return "test"
}

func testTryStuff() {
    // assumes there will be no error thrown, otherwise a runtime exception is raised
    let _ = try! fakeFetch(7)

    // if an error is thrown, then it proceeds, but if the value is nil
    // it also wraps every return value in an optional, even if its already optional
    let _ = try? fakeFetch(7)

    do {
        // normal try operation that provides error handling via `catch` block
        try fakeFetch(1)
    } catch MyError.BadValue(let msg) {
        print("Error message: \(msg)")
    } catch {
        // must be exhaustive
    }
}
testTryStuff()

//
// MARK: Classes
//

// Classes, structures and its members have three levels of access control
// They are: internal (default), public, private

public class Shape {
    public func getArea() -> Int {
        return 0
    }
}

// All methods and properties of a class are public.
// If you just need to store data in a
// structured object, you should use a `struct`

internal class Rect: Shape {
    var sideLength: Int = 1

    // Custom getter and setter property
    private var perimeter: Int {
        get {
            return 4 * sideLength
        }
        set {
            // `newValue` is an implicit variable available to setters
            sideLength = newValue / 4
        }
    }

    // Computed properties must be declared as `var`, you know, cause' they can change
    var smallestSideLength: Int {
        return self.sideLength - 1
    }

    // Lazily load a property
    // subShape remains nil (uninitialized) until getter called
    lazy var subShape = Rect(sideLength: 4)

    // If you don't need a custom getter and setter,
    // but still want to run code before and after getting or setting
    // a property, you can use `willSet` and `didSet`
    var identifier: String = "defaultID" {
        // the `willSet` arg will be the variable name for the new value
        willSet(someIdentifier) {
            print(someIdentifier)
        }
    }

    init(sideLength: Int) {
        self.sideLength = sideLength
        // always super.init last when init custom properties
        super.init()
    }

    func shrink() {
        if sideLength > 0 {
            --sideLength
        }
    }

    override func getArea() -> Int {
        return sideLength * sideLength
    }
}

// A simple class `Square` extends `Rect`
class Square: Rect {
    convenience init() {
        self.init(sideLength: 5)
    }
}

var mySquare = Square()
print(mySquare.getArea()) // 25
mySquare.shrink()
print(mySquare.sideLength) // 4

// cast instance
let aShape = mySquare as Shape

// compare instances, not the same as == which compares objects (equal to)
if mySquare === mySquare {
    print("Yep, it's mySquare")
}

// Optional init
class Circle: Shape {
    var radius: Int
    override func getArea() -> Int {
        return 3 * radius * radius
    }

    // Place a question mark postfix after `init` is an optional init
    // which can return nil
    init?(radius: Int) {
        self.radius = radius
        super.init()

        if radius <= 0 {
            return nil
        }
    }
}

var myCircle = Circle(radius: 1)
print(myCircle?.getArea())    // Optional(3)
print(myCircle!.getArea())    // 3
var myEmptyCircle = Circle(radius: -1)
print(myEmptyCircle?.getArea())    // "nil"
if let circle = myEmptyCircle {
    // will not execute since myEmptyCircle is nil
    print("circle is not nil")
}
```

```
//
// MARK: Enums
//

// Enums can optionally be of a specific type or on their own.
// They can contain methods like classes.

enum Suit {
    case Spades, Hearts, Diamonds, Clubs
    func getIcon() -> String {
        switch self {
        case .Spades: return " "
        case .Hearts: return " "
        case .Diamonds: return " "
        case .Clubs: return " "
        }
    }
}

// Enum values allow short hand syntax, no need to type the enum type
// when the variable is explicitly declared
var suitValue: Suit = .Hearts

// String enums can have direct raw value assignments
// or their raw values will be derived from the Enum field
enum BookName: String {
    case John
    case Luke = "Luke"
}
print("Name: \(BookName.John.rawValue)")

// Enum with associated Values
enum Furniture {
    // Associate with Int
    case Desk(height: Int)
    // Associate with String and Int
    case Chair(String, Int)

    func description() -> String {
        switch self {
        case .Desk(let height):
            return "Desk with \(height) cm"
        case .Chair(let brand, let height):
            return "Chair of \(brand) with \(height) cm"
        }
    }
}

var desk: Furniture = .Desk(height: 80)
print(desk.description())     // "Desk with 80 cm"
var chair = Furniture.Chair("Foo", 40)
print(chair.description())    // "Chair of Foo with 40 cm"
```

```
//
// MARK: Protocols
//

// `protocol`s can require that conforming types have specific
// instance properties, instance methods, type methods,
// operators, and subscripts.

protocol ShapeGenerator {
    var enabled: Bool { get set }
    func buildShape() -> Shape
}

// Protocols declared with @objc allow optional functions,
// which allow you to check for conformance
@objc protocol TransformShape {
    optional func reshape()
    optional func canReshape() -> Bool
}

class MyShape: Rect {
    var delegate: TransformShape?

    func grow() {
        sideLength += 2

        // Place a question mark after an optional property, method, or
        // subscript to gracefully ignore a nil value and return nil
        // instead of throwing a runtime error ("optional chaining").
        if let reshape = self.delegate?.canReshape?() where reshape {
            // test for delegate then for method
            self.delegate?.reshape?()
        }
    }
}


//
// MARK: Other
//

// `extension`s: Add extra functionality to an already existing type

// Square now "conforms" to the `CustomStringConvertible` protocol
extension Square: CustomStringConvertible {
    var description: String {
        return "Area: \(self.getArea()) - ID: \(self.identifier)"
    }
}

print("Square: \(mySquare)")

// You can also extend built-in types
extension Int {
    var customProperty: String {
        return "This is \(self)"
    }

    func multiplyBy(num: Int) -> Int {
        return num * self
    }
}

print(7.customProperty) // "This is 7"
print(14.multiplyBy(3)) // 42

// Generics: Similar to Java and C#. Use the `where` keyword to specify the
//   requirements of the generics.

func findIndex<T: Equatable>(array: [T], _ valueToFind: T) -> Int? {
    for (index, value) in array.enumerate() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
let foundAtIndex = findIndex([1, 2, 3, 4], 3)
print(foundAtIndex == 2) // true

// Operators:
// Custom operators can start with the characters:
//      / = - + * % < > ! & | ^ . ~
// or
// Unicode math, symbol, arrow, dingbat, and line/box drawing characters.
prefix operator !!! {}

// A prefix operator that triples the side length when used
prefix func !!! (inout shape: Square) -> Square {
    shape.sideLength *= 3
    return shape
}

// current value
print(mySquare.sideLength) // 4

// change side length using custom !!! operator, increases size by 3
!!!mySquare
print(mySquare.sideLength) // 12

// Operators can also be generics
infix operator <-> {}
func <-><T: Equatable> (inout a: T, inout b: T) {
    let c = a
    a = b
    b = c
}
```

```
var foo: Float = 10
var bar: Float = 20

foo <-> bar
print("foo is \(foo), bar is \(bar)") // "foo is 20.0, bar is 10.0"
```

Tcl

Tcl was created by John Ousterhout as a reusable scripting language for chip design tools he was creating. In 1997 he was awarded the ACM Software System Award for Tcl. Tcl can be used both as an embeddable scripting language and as a general programming language. It can also be used as a portable C library, even in cases where no scripting capability is needed, as it provides data structures such as dynamic strings, lists, and hash tables. The C library also provides portable functionality for loading dynamic libraries, string formatting and code conversion, filesystem operations, network operations, and more. Various features of Tcl stand out:

- Convenient cross-platform networking API
- Fully virtualized filesystem
- Stackable I/O channels
- Asynchronous to the core
- Full coroutines
- A threading model recognized as robust and easy to use

If Lisp is a list processor, then Tcl is a string processor. All values are strings. A list is a string format. A procedure definition is a string format. To achieve performance, Tcl internally caches structured representations of these values. The list commands, for example, operate on the internal cached representation, and Tcl takes care of updating the string representation if it is ever actually needed in the script. The copy-on-write design of Tcl allows script authors can pass around large data values without actually incurring additional memory overhead. Procedures are automatically byte-compiled unless they use the more dynamic commands such as “uplevel”, “upvar”, and “trace”.

Tcl is a pleasure to program in. It will appeal to hacker types who find Lisp, Forth, or Smalltalk interesting, as well as to engineers and scientists who just want to get down to business with a tool that bends to their will. Its discipline of exposing all programmatic functionality as commands, including things like loops and mathematical operations that are usually baked into the syntax of other languages, allows it to fade into the background of whatever domain-specific functionality a project needs. It’s syntax, which is even lighter that that of Lisp, just gets out of the way.

```
#! /bin/env tclsh

################################################################################
## 1. Guidelines
################################################################################

# Tcl is not Bash or C!  This needs to be said because standard shell quoting
# habits almost work in Tcl and it is common for people to pick up Tcl and try
# to get by with syntax they know from another language.  It works at first,
# but soon leads to frustration with more complex scripts.

# Braces are just a quoting mechanism, not a code block constructor or a list
# constructor.  Tcl doesn't have either of those things.  Braces are used,
# though, to escape special characters in procedure bodies and in strings that
# are formatted as lists.
```

```
################################################################################
## 2. Syntax
################################################################################

# Every line is a command.  The first word is the name of the command, and
# subsequent words are arguments to the command. Words are delimited by
# whitespace.  Since every word is a string, in the simple case no special
# markup such as quotes, braces, or backslash, is necessary.  Even when quotes
# are used, they are not a string constructor, but just another escaping
# character.

set greeting1 Sal
set greeting2 ut
set greeting3 ations


#semicolon also delimits commands
set greeting1 Sal; set greeting2 ut; set greeting3 ations


# Dollar sign introduces variable substitution
set greeting $greeting1$greeting2$greeting3


# Bracket introduces command substitution.  The result of the command is
# substituted in place of the bracketed script.  When the "set" command is
# given only the name of a variable, it returns the value of that variable.
set greeting $greeting1$greeting2[set greeting3]


# Command substitution should really be called script substitution, because an
# entire script, not just a command, can be placed between the brackets. The
# "incr" command increments the value of a variable and returns its value.
set greeting $greeting[
    incr i
    incr i
    incr i
]


# backslash suppresses the special meaning of characters
set amount \$16.42


# backslash adds special meaning to certain characters
puts lots\nof\n\n\n\n\n\nnewlines


# A word enclosed in braces is not subject to any special interpretation or
# substitutions, except that a backslash before a brace is not counted when
# looking for the closing brace
set somevar {
    This is a literal $ sign, and this \} escaped
    brace remains uninterpreted
}


# In a word enclosed in double quotes, whitespace characters lose their special
# meaning
set name Neo
set greeting "Hello, $name"


#variable names can be any string
set {first name} New


# The brace form of variable substitution handles more complex variable names
set greeting "Hello, ${first name}"


# The "set" command can always be used instead of variable substitution
set greeting "Hello, [set {first name}]"


# To promote the words within a word to individual words of the current
# command, use the expansion operator, "{*}".
set {*}{name Neo}

# is equivalent to
set name Neo


# An array is a special variable that is a container for other variables.
set person(name) Neo
set person(gender) male
set greeting "Hello, $person(name)"


# A namespace holds commands and variables
namespace eval people {
    namespace eval person1 {
        variable name Neo
    }
}


#The full name of a variable includes its enclosing namespace(s), delimited by two colons:
set greeting "Hello $people::person1::name"



################################################################################
## 3. A Few Notes
################################################################################
```

```
# All other functionality is implemented via commands.  From this point on,
# there is no new syntax.  Everything else there is to learn about Tcl is about
# the behaviour of individual commands, and what meaning they assign to their
# arguments.


# To end up with an interpreter that can do nothing, delete the global
# namespace.  It's not very useful to do such a thing, but it illustrates the
# nature of Tcl.
namespace delete ::


# Because of name resolution behaviour, it's safer to use the "variable" command to
# declare or to assign a value to a namespace. If a variable called "name" already
# exists in the global namespace, using "set" here will assign a value to the global variable
# instead of creating a new variable in the local namespace.
namespace eval people {
    namespace eval person1 {
        variable name Neo
    }
}


# The full name of a variable can always be used, if desired.
set people::person1::name Neo



################################################################################
## 4. Commands
################################################################################

# Math can be done with the "expr" command.
set a 3
set b 4
set c [expr {$a + $b}]

# Since "expr" performs variable substitution on its own, brace the expression
# to prevent Tcl from performing variable substitution first.  See
# "http://wiki.tcl.tk/Brace%20your%20#%20expr-essions" for details.


# The "expr" command understands variable and command substitution
set c [expr {$a + [set b]}]


# The "expr" command provides a set of mathematical functions
set c [expr {pow($a,$b)}]


# Mathematical operators are available as commands in the ::tcl::mathop
# namespace
::tcl::mathop::+ 5 3
```

```
# Commands can be imported from other namespaces
namespace import ::tcl::mathop::+
set result [+ 5 3]


# New commands can be created via the "proc" command.
proc greet name {
    return "Hello, $name!"
}

#multiple parameters can be specified
proc greet {greeting name} {
    return "$greeting, $name!"
}


# As noted earlier, braces do not construct a code block.  Every value, even
# the third argument of the "proc" command, is a string.  The previous command
# rewritten to not use braces at all:
proc greet greeting\ name return\ \"Hello,\ \$name!



# When the last parameter is the literal value, "args", it collects all extra
# arguments when the command is invoked
proc fold {cmd args} {
    set res 0
    foreach arg $args {
        set res [$cmd $res $arg]
    }
}
fold ::tcl::mathop::* 5 3 3 ;# ->  45


# Conditional execution is implemented as a command
if {3 > 4} {
    puts {This will never happen}
} elseif {4 > 4} {
    puts {This will also never happen}
} else {
    puts {This will always happen}
}


# Loops are implemented as commands.  The first, second, and third
# arguments of the "for" command are treated as mathematical expressions
for {set i 0} {$i < 10} {incr i} {
    set res [expr {$res + $i}]
}


# The first argument of the "while" command is also treated as a mathematical
# expression
```

```
set i 0
while {$i < 10} {
    incr i 2
}


# A list is a specially-formatted string.  In the simple case, whitespace is sufficient to delimit valu
set amounts 10\ 33\ 18
set amount [lindex $amounts 1]


# Braces and backslash can be used to format more complex values in a list.  A
# list looks exactly like a script, except that the newline character and the
# semicolon character lose their special meanings.  This feature makes Tcl
# homoiconic.  There are three items in the following list.
set values {

    one\ two

    {three four}

    five\{six

}


# Since a list is a string, string operations could be performed on it, at the
# risk of corrupting the formatting of the list.
set values {one two three four}
set values [string map {two \{} $values] ;# $values is no-longer a \
    properly-formatted listwell-formed list


# The sure-fire way to get a properly-formmated list is to use "list" commands
set values [list one \{ three four]
lappend values { } ;# add a single space as an item in the list


# Use "eval" to evaluate a value as a script
eval {
    set name Neo
    set greeting "Hello, $name"
}


# A list can always be passed to "eval" as a script composed of a single
# command.
eval {set name Neo}
eval [list set greeting "Hello, $name"]


# Therefore, when using "eval", use [list] to build up a desired command
set command {set name}
lappend command {Archibald Sorbisol}
```

```
eval $command


# A common mistake is not to use list functions when building up a command
set command {set name}
append command { Archibald Sorbisol}
eval $command ;# There is an error here, because there are too many arguments \
    to "set" in {set name Archibald Sorbisol}


# This mistake can easily occur with the "subst" command.
set replacement {Archibald Sorbisol}
set command {set name $replacement}
set command [subst $command]
eval $command ;# The same error as before: too many arguments to "set" in \
    {set name Archibald Sorbisol}


# The proper way is to format the substituted value using use the "list"
# command.
set replacement [list {Archibald Sorbisol}]
set command {set name $replacement}
set command [subst $command]
eval $command


# It is extremely common to see the "list" command being used to properly
# format values that are substituted into Tcl script templates.  There are
# several examples of this, below.


# The "apply" command evaluates a string as a command.
set cmd {{greeting name} {
    return "$greeting, $name!"
}}
apply $cmd Whaddup Neo


# The "uplevel" command evaluates a script in some enclosing scope.
proc greet {} {
    uplevel {puts "$greeting, $name"}
}

proc set_double {varname value} {
    if {[string is double $value]} {
        uplevel [list variable $varname $value]
    } else {
        error [list {not a double} $value]
    }
}


# The "upvar" command links a variable in the current scope to a variable in
# some enclosing scope
```

```
proc set_double {varname value} {
    if {[string is double $value]} {
        upvar 1 $varname var
        set var $value
    } else {
        error [list {not a double} $value]
    }
}


#get rid of the built-in "while" command.
rename ::while {}


# Define a new while command with the "proc" command.  More sophisticated error
# handling is left as an exercise.
proc while {condition script} {
    if {[uplevel 1 [list expr $condition]]} {
        uplevel 1 $script
        tailcall [namespace which while] $condition $script
    }
}


# The "coroutine" command creates a separate call stack, along with a command
# to enter that call stack. The "yield" command suspends execution in that
# stack.
proc countdown {} {
    #send something back to the initial "coroutine" command
    yield

    set count 3
    while {$count > 1} {
        yield [incr count -1]
    }
    return 0
}
coroutine countdown1 countdown
coroutine countdown2 countdown
puts [countdown 1] ;# -> 2
puts [countdown 2] ;# -> 2
puts [countdown 1] ;# -> 1
puts [countdown 1] ;# -> 0
puts [coundown 1] ;# -> invalid command name "countdown1"
puts [countdown 2] ;# -> 1
```

Reference

Official Tcl Documentation

Tcl Wiki

Tcl Subreddit

Tmux

tmux is a terminal multiplexer: it enables a number of terminals to be created, accessed, and controlled from a single screen. tmux may be detached from a screen and continue running in the background then later reattached.

```
  tmux [command]     # Run a command
                     # 'tmux' with no commands will create a new session

    new              # Create a new session
     -s "Session"    # Create named session
     -n "Window"     # Create named Window
     -c "/dir"       # Start in target directory

    attach           # Attach last/available session
     -t "#"          # Attach target session
     -d              # Detach the session from other instances

    ls               # List open sessions
     -a              # List all open sessions

    lsw              # List windows
     -a              # List all windows
     -s              # List all windows in session

    lsp              # List panes
     -a              # List all panes
     -s              # List all panes in session
     -t              # List all panes in target

    kill-window      # Kill current window
     -t "#"          # Kill target window
     -a              # Kill all windows
     -a -t "#"       # Kill all windows but the target

    kill-session     # Kill current session
     -t "#"          # Kill target session
     -a              # Kill all sessions
     -a -t "#"       # Kill all sessions but the target
```

Key Bindings

The method of controlling an attached tmux session is via key combinations called ‘Prefix’ keys.

```
----------------------------------------------------------------------
  (C-b) = Ctrl + b    # 'Prefix' combination required to use keybinds

  (M-1) = Meta + 1 -or- Alt + 1
----------------------------------------------------------------------


  ?                  # List all key bindings
  :                  # Enter the tmux command prompt
  r                  # Force redraw of the attached client
  c                  # Create a new window

  !                  # Break the current pane out of the window.
  %                  # Split the current pane into two, left and right
  "                  # Split the current pane into two, top and bottom

  n                  # Change to the next window
  p                  # Change to the previous window
  {                  # Swap the current pane with the previous pane
  }                  # Swap the current pane with the next pane

  s                  # Select a new session for the attached client
                     interactively
  w                  # Choose the current window interactively
  0 to 9             # Select windows 0 to 9

  d                  # Detach the current client
  D                  # Choose a client to detach

  &                  # Kill the current window
  x                  # Kill the current pane

  Up, Down           # Change to the pane above, below, left, or right
  Left, Right

  M-1 to M-5         # Arrange panes:
                       # 1) even-horizontal
                       # 2) even-vertical
                       # 3) main-horizontal
                       # 4) main-vertical
                       # 5) tiled

  C-Up, C-Down       # Resize the current pane in steps of one cell
  C-Left, C-Right

  M-Up, M-Down       # Resize the current pane in steps of five cells
  M-Left, M-Right
```

Configuring ~/.tmux.conf

tmux.conf can be used to set options automatically on start up, much like how .vimrc or init.el are used.

```
# Example tmux.conf
# 2014.10


### General
###########################################################################

# Enable UTF-8
setw -g utf8 on
set-option -g status-utf8 on

# Scrollback/History limit
```

```
set -g history-limit 2048

# Index Start
set -g base-index 1

# Mouse
set-option -g mouse-select-pane on

# Force reload of config file
unbind r
bind r source-file ~/.tmux.conf


### Keybinds
###########################################################################

# Unbind C-b as the default prefix
unbind C-b

# Set new default prefix
set-option -g prefix `

# Return to previous window when prefix is pressed twice
bind C-a last-window
bind ` last-window

# Allow swapping C-a and ` using F11/F12
bind F11 set-option -g prefix C-a
bind F12 set-option -g prefix `

# Keybind preference
setw -g mode-keys vi
set-option -g status-keys vi

# Moving between panes with vim movement keys
bind h select-pane -L
bind j select-pane -D
bind k select-pane -U
bind l select-pane -R

# Window Cycle/Swap
bind e previous-window
bind f next-window
bind E swap-window -t -1
bind F swap-window -t +1

# Easy split pane commands
bind = split-window -h
bind - split-window -v
unbind '"'
unbind %

# Activate inner-most session (when nesting tmux) to send commands
bind a send-prefix
```

```
### Theme
###########################################################################

# Statusbar Color Palatte
set-option -g status-justify left
set-option -g status-bg black
set-option -g status-fg white
set-option -g status-left-length 40
set-option -g status-right-length 80

# Pane Border Color Palette
set-option -g pane-active-border-fg green
set-option -g pane-active-border-bg black
set-option -g pane-border-fg white
set-option -g pane-border-bg black

# Message Color Palette
set-option -g message-fg black
set-option -g message-bg green

# Window Status Color Palette
setw -g window-status-bg black
setw -g window-status-current-fg green
setw -g window-status-bell-attr default
setw -g window-status-bell-fg red
setw -g window-status-content-attr default
setw -g window-status-content-fg yellow
setw -g window-status-activity-attr default
setw -g window-status-activity-fg yellow


### UI
###########################################################################

# Notification
setw -g monitor-activity on
set -g visual-activity on
set-option -g bell-action any
set-option -g visual-bell off

# Automatically set window titles
set-option -g set-titles on
set-option -g set-titles-string '#H:#S.#I.#P #W #T' # window number,program name,active (or not)

# Statusbar Adjustments
set -g status-left "#[fg=red] #H#[fg=green]:#[fg=white]#S#[fg=green] |#[default]"

# Show performance counters in statusbar
# Requires https://github.com/thewtex/tmux-mem-cpu-load/
set -g status-interval 4
set -g status-right "#[fg=green] | #[fg=white]#(tmux-mem-cpu-load)#[fg=green] | #[fg=cyan]%H:%M #[default]
```

References

Tmux | Home

Tmux Manual page

Gentoo Wiki

Archlinux Wiki

Display CPU/MEM % in statusbar

tmuxinator - Manage complex tmux sessions

Typescript

TypeScript is a language that aims at easing development of large scale applications written in JavaScript. TypeScript adds common concepts such as classes, modules, interfaces, generics and (optional) static typing to JavaScript. It is a superset of JavaScript: all JavaScript code is valid TypeScript code so it can be added seamlessly to any project. The TypeScript compiler emits JavaScript.

This article will focus only on TypeScript extra syntax, as opposed to JavaScript (../javascript/).

To test TypeScript's compiler, head to the [Playground] (http://www.typescriptlang.org/Playground) where you will be able to type code, have auto completion and directly see the emitted JavaScript.

```
// There are 3 basic types in TypeScript
var isDone: boolean = false;
var lines: number = 42;
var name: string = "Anders";

// When it's impossible to know, there is the "Any" type
var notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean

// For collections, there are typed arrays and generic arrays
var list: number[] = [1, 2, 3];
// Alternatively, using the generic array type
var list: Array<number> = [1, 2, 3];

// For enumerations:
enum Color {Red, Green, Blue};
var c: Color = Color.Green;

// Lastly, "void" is used in the special case of a function returning nothing
function bigHorribleAlert(): void {
  alert("I'm a little annoying box!");
}

// Functions are first class citizens, support the lambda "fat arrow" syntax and
// use type inference

// The following are equivalent, the same signature will be infered by the
// compiler, and same JavaScript will be emitted
var f1 = function(i: number): number { return i * i; }
// Return type inferred
```

```
var f2 = function(i: number) { return i * i; }
var f3 = (i: number): number => { return i * i; }
// Return type inferred
var f4 = (i: number) => { return i * i; }
// Return type inferred, one-liner means no return keyword needed
var f5 = (i: number) =>  i * i;

// Interfaces are structural, anything that has the properties is compliant with
// the interface
interface Person {
  name: string;
  // Optional properties, marked with a "?"
  age?: number;
  // And of course functions
  move(): void;
}

// Object that implements the "Person" interface
// Can be treated as a Person since it has the name and move properties
var p: Person = { name: "Bobby", move: () => {} };
// Objects that have the optional property:
var validPerson: Person = { name: "Bobby", age: 42, move: () => {} };
// Is not a person because age is not a number
var invalidPerson: Person = { name: "Bobby", age: true };

// Interfaces can also describe a function type
interface SearchFunc {
  (source: string, subString: string): boolean;
}
// Only the parameters' types are important, names are not important.
var mySearch: SearchFunc;
mySearch = function(src: string, sub: string) {
  return src.search(sub) != -1;
}

// Classes - members are public by default
class Point {
  // Properties
  x: number;

  // Constructor - the public/private keywords in this context will generate
  // the boiler plate code for the property and the initialization in the
  // constructor.
  // In this example, "y" will be defined just like "x" is, but with less code
  // Default values are also supported

  constructor(x: number, public y: number = 0) {
    this.x = x;
  }

  // Functions
  dist() { return Math.sqrt(this.x * this.x + this.y * this.y); }

  // Static members
  static origin = new Point(0, 0);
}

var p1 = new Point(10 ,20);
var p2 = new Point(25); //y will be 0

// Inheritance
class Point3D extends Point {
  constructor(x: number, y: number, public z: number = 0) {
    super(x, y); // Explicit call to the super class constructor is mandatory
  }

  // Overwrite
  dist() {
    var d = super.dist();
    return Math.sqrt(d * d + this.z * this.z);
  }
}

// Modules, "." can be used as separator for sub modules
module Geometry {
  export class Square {
    constructor(public sideLength: number = 0) {
    }
    area() {
      return Math.pow(this.sideLength, 2);
    }
  }
}

var s1 = new Geometry.Square(5);

// Local alias for referencing a module
import G = Geometry;

var s2 = new G.Square(10);

// Generics
// Classes
class Tuple<T1, T2> {
  constructor(public item1: T1, public item2: T2) {
  }
}

// Interfaces
interface Pair<T> {
  item1: T;
  item2: T;
}

// And functions
var pairToTuple = function<T>(p: Pair<T>) {
  return new Tuple(p.item1, p.item2);
};
```

```
var tuple = pairToTuple({ item1:"hello", item2:"world"});

// Including references to a definition file:
/// <reference path="jquery.d.ts" />

// Template Strings (strings that use backticks)
// String Interpolation with Template Strings
var name = 'Tyrone';
var greeting = `Hi ${name}, how are you?`
// Multiline Strings with Template Strings
var multiline = `This is an example
of a multiline string`;
```

Further Reading

- [TypeScript Official website] (http://www.typescriptlang.org/)
- [TypeScript language specifications (pdf)] (http://go.microsoft.com/fwlink/?LinkId=267238)
- [Anders Hejlsberg - Introducing TypeScript on Channel 9] (http://channel9.msdn.com/posts/Anders-Hejlsberg-Introducing-TypeScript)
- [Source Code on GitHub] (https://github.com/Microsoft/TypeScript)
- [Definitely Typed - repository for type definitions] (http://definitelytyped.org/)

Visualbasic

```
Module Module1

    Sub Main()
        'A Quick Overview of Visual Basic Console Applications before we dive
        'in to the deep end.
        'Apostrophe starts comments.
        'To Navigate this tutorial within the Visual Basic Complier, I've put
        'together a navigation system.
        'This navigation system is explained however as we go deeper into this
        'tutorial, you'll understand what it all means.
        Console.Title = ("Learn X in Y Minutes")
        Console.WriteLine("NAVIGATION") 'Display
        Console.WriteLine("")
        Console.ForegroundColor = ConsoleColor.Green
        Console.WriteLine("1. Hello World Output")
        Console.WriteLine("2. Hello World Input")
        Console.WriteLine("3. Calculating Whole Numbers")
        Console.WriteLine("4. Calculating Decimal Numbers")
        Console.WriteLine("5. Working Calculator")
        Console.WriteLine("6. Using Do While Loops")
        Console.WriteLine("7. Using For While Loops")
        Console.WriteLine("8. Conditional Statements")
        Console.WriteLine("9. Select A Drink")
        Console.WriteLine("50. About")
        Console.WriteLine("Please Choose A Number From The Above List")
        Dim selection As String = Console.ReadLine
        'The "Case" in the Select statement is optional.
        'For example, "Select selection" instead of "Select Case selection"
        'will also work.
        Select Case selection
            Case "1" 'HelloWorld Output
                Console.Clear() 'Clears the application and opens the private sub
                HelloWorldOutput() 'Name Private Sub, Opens Private Sub
            Case "2" 'Hello Input
                Console.Clear()
                HelloWorldInput()
            Case "3" 'Calculating Whole Numbers
                Console.Clear()
                CalculatingWholeNumbers()
            Case "4" 'Calculating Decimal Numbers
                Console.Clear()
                CalculatingDecimalNumbers()
            Case "5" 'Working Calculator
                Console.Clear()
                WorkingCalculator()
            Case "6" 'Using Do While Loops
                Console.Clear()
                UsingDoWhileLoops()
            Case "7" 'Using For While Loops
                Console.Clear()
                UsingForLoops()
            Case "8" 'Conditional Statements
                Console.Clear()
                ConditionalStatement()
            Case "9" 'If/Else Statement
                Console.Clear()
                IfElseStatement() 'Select a drink
            Case "50" 'About msg box
                Console.Clear()
                Console.Title = ("Learn X in Y Minutes :: About")
                MsgBox("This tutorial is by Brian Martin (@BrianMartinn")
                Console.Clear()
                Main()
                Console.ReadLine()

        End Select
    End Sub

    'One - I'm using numbers to help with the above navigation when I come back
    'later to build it.

    'We use private subs to separate different sections of the program.
    Private Sub HelloWorldOutput()
        'Title of Console Application
        Console.Title = "Hello World Output | Learn X in Y Minutes"
        'Use Console.Write("") or Console.WriteLine("") to print outputs.
        'Followed by Console.Read() alternatively Console.Readline()
        'Console.ReadLine() prints the output to the console.
        Console.WriteLine("Hello World")
        Console.ReadLine()
    End Sub
```

```
'Two
Private Sub HelloWorldInput()
    Console.Title = "Hello World YourName | Learn X in Y Minutes"
    'Variables
    'Data entered by a user needs to be stored.
    'Variables also start with a Dim and end with an As VariableType.

    'In this tutorial, we want to know what your name, and make the program
    'respond to what is said.
    Dim username As String
    'We use string as string is a text based variable.
    Console.WriteLine("Hello, What is your name? ") 'Ask the user their name.
    username = Console.ReadLine() 'Stores the users name.
    Console.WriteLine("Hello " + username) 'Output is Hello 'Their name'
    Console.ReadLine() 'Outputs the above.
    'The above will ask you a question followed by printing your answer.
    'Other variables include Integer and we use Integer for whole numbers.
End Sub

'Three
Private Sub CalculatingWholeNumbers()
    Console.Title = "Calculating Whole Numbers | Learn X in Y Minutes"
    Console.Write("First number: ") 'Enter a whole number, 1, 2, 50, 104, etc
    Dim a As Integer = Console.ReadLine()
    Console.Write("Second number: ") 'Enter second whole number.
    Dim b As Integer = Console.ReadLine()
    Dim c As Integer = a + b
    Console.WriteLine(c)
    Console.ReadLine()
    'The above is a simple calculator
End Sub

'Four
Private Sub CalculatingDecimalNumbers()
    Console.Title = "Calculating with Double | Learn X in Y Minutes"
    'Of course we would like to be able to add up decimals.
    'Therefore we could change the above from Integer to Double.

    'Enter a floating-point number, 1.2, 2.4, 50.1, 104.9, etc
    Console.Write("First number: ")
    Dim a As Double = Console.ReadLine
    Console.Write("Second number: ") 'Enter second floating-point number.
    Dim b As Double = Console.ReadLine
    Dim c As Double = a + b
    Console.WriteLine(c)
    Console.ReadLine()
    'Therefore the above program can add up 1.1 - 2.2
End Sub

'Five
Private Sub WorkingCalculator()
    Console.Title = "The Working Calculator| Learn X in Y Minutes"
    'However if you'd like the calculator to subtract, divide, multiple and
    'add up.
    'Copy and paste the above again.
    Console.Write("First number: ")
    Dim a As Double = Console.ReadLine
    Console.Write("Second number: ") 'Enter second floating-point number.
    Dim b As Double = Console.ReadLine
    Dim c As Double = a + b
    Dim d As Double = a * b
    Dim e As Double = a - b
    Dim f As Double = a / b

    'By adding the below lines we are able to calculate the subtract,
    'multiply as well as divide the a and b values
    Console.Write(a.ToString() + " + " + b.ToString())
    'We want to pad the answers to the left by 3 spaces.
    Console.WriteLine(" = " + c.ToString.PadLeft(3))
    Console.Write(a.ToString() + " * " + b.ToString())
    Console.WriteLine(" = " + d.ToString.PadLeft(3))
    Console.Write(a.ToString() + " - " + b.ToString())
    Console.WriteLine(" = " + e.ToString.PadLeft(3))
    Console.Write(a.ToString() + " / " + b.ToString())
    Console.WriteLine(" = " + f.ToString.PadLeft(3))
    Console.ReadLine()

End Sub

'Six
Private Sub UsingDoWhileLoops()
    'Just as the previous private sub
    'This Time We Ask If The User Wishes To Continue (Yes or No?)
    'We're using Do While Loop as we're unsure if the user wants to use the
    'program more than once.
    Console.Title = "UsingDoWhileLoops | Learn X in Y Minutes"
    Dim answer As String 'We use the variable "String" as the answer is text
    Do 'We start the program with
        Console.Write("First number: ")
        Dim a As Double = Console.ReadLine
        Console.Write("Second number: ")
        Dim b As Double = Console.ReadLine
        Dim c As Double = a + b
        Dim d As Double = a * b
        Dim e As Double = a - b
        Dim f As Double = a / b

        Console.Write(a.ToString() + " + " + b.ToString())
        Console.WriteLine(" = " + c.ToString.PadLeft(3))
        Console.Write(a.ToString() + " * " + b.ToString())
        Console.WriteLine(" = " + d.ToString.PadLeft(3))
        Console.Write(a.ToString() + " - " + b.ToString())
        Console.WriteLine(" = " + e.ToString.PadLeft(3))
        Console.Write(a.ToString() + " / " + b.ToString())
        Console.WriteLine(" = " + f.ToString.PadLeft(3))
        Console.ReadLine()
        'Ask the question, does the user wish to continue? Unfortunately it
        'is case sensitive.
        Console.Write("Would you like to continue? (yes / no) ")
        'The program grabs the variable and prints and starts again.
        answer = Console.ReadLine
        'The command for the variable to work would be in this case "yes"
    Loop While answer = "yes"

End Sub

'Seven
Private Sub UsingForLoops()
    'Sometimes the program only needs to run once.
    'In this program we'll be counting down from 10.

    Console.Title = "Using For Loops | Learn X in Y Minutes"
    'Declare Variable and what number it should count down in Step -1,
    'Step -2, Step -3, etc.
    For i As Integer = 10 To 0 Step -1
        Console.WriteLine(i.ToString) 'Print the value of the counter
    Next i 'Calculate new value
    Console.WriteLine("Start") 'Lets start the program baby!!
    Console.ReadLine() 'POW!! - Perhaps I got a little excited then :)
End Sub

'Eight
Private Sub ConditionalStatement()
    Console.Title = "Conditional Statements | Learn X in Y Minutes"
    Dim userName As String
    Console.WriteLine("Hello, What is your name? ") 'Ask the user their name.
    userName = Console.ReadLine() 'Stores the users name.
    If userName = "Adam" Then
        Console.WriteLine("Hello Adam")
        Console.WriteLine("Thanks for creating this useful site")
        Console.ReadLine()
    Else
        Console.WriteLine("Hello " + userName)
        Console.WriteLine("Have you checked out www.learnxinyminutes.com")
        Console.ReadLine() 'Ends and prints the above statement.
    End If
End Sub

'Nine
Private Sub IfElseStatement()
    Console.Title = "If / Else Statement | Learn X in Y Minutes"
    'Sometimes it is important to consider more than two alternatives.
    'Sometimes there are a good few others.
    'When this is the case, more than one if statement would be required.
    'An if statement is great for vending machines. Where the user enters a code.
    'A1, A2, A3, etc to select an item.
    'All choices can be combined into a single if block.

    Dim selection As String 'Declare a variable for selection
    Console.WriteLine("Please select a product form our lovely vending machine.")
    Console.WriteLine("A1. for 7Up")
```

```
        Console.WriteLine("A2. for Fanta")
        Console.WriteLine("A3. for Dr. Pepper")
        Console.WriteLine("A4. for Diet Coke")

        selection = Console.ReadLine() 'Store a selection from the user
        If selection = "A1" Then
            Console.WriteLine("7up")
        ElseIf selection = "A2" Then
            Console.WriteLine("fanta")
        ElseIf selection = "A3" Then
            Console.WriteLine("dr. pepper")
        ElseIf selection = "A4" Then
            Console.WriteLine("diet coke")
        Else
            Console.WriteLine("Sorry, I don't have any " + selection)
        End If
        Console.ReadLine()

    End Sub

End Module
```

References

I learnt Visual Basic in the console application. It allowed me to understand the principles of computer programming to go on to learn other programming languages easily.

I created a more indepth Visual Basic tutorial for those who would like to learn more.

The entire syntax is valid. Copy the and paste in to the Visual Basic compiler and run (F5) the program.

Whip

Whip is a LISP-dialect made for scripting and simplified concepts. It has also borrowed a lot of functions and syntax from Haskell (a non-related language).

These docs were written by the creator of the language himself. So is this line.

```
; Comments are like LISP. Semi-colons...

; Majority of first-level statements are inside "forms"
; which are just things inside parens separated by whitespace
not_in_form
(in_form)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; 1. Numbers, Strings, and Operators

; Whip has one number type (which is a 64-bit IEEE 754 double, from JavaScript).
3 ; => 3
1.5 ; => 1.5

; Functions are called if they are the first element in a form
(called_function args)
```

```
; Majority of operations are done with functions
; All the basic arithmetic is pretty straight forward
(+ 1 1) ; => 2
(- 2 1) ; => 1
(* 1 2) ; => 2
(/ 2 1) ; => 2
; even modulo
(% 9 4) ; => 1
; JavaScript-style uneven division.
(/ 5 2) ; => 2.5

; Nesting forms works as you expect.
(* 2 (+ 1 3)) ; => 8

; There's a boolean type.
true
false

; Strings are created with ".
"Hello, world"

; Single chars are created with '.
'a'

; Negation uses the 'not' function.
(not true) ; => false
(not false) ; => true

; But the majority of non-haskell functions have shortcuts
; not's shortcut is a '!'.
(! (! true)) ; => true

; Equality is `equal` or `=`.
(= 1 1) ; => true
(equal 2 1) ; => false

; For example, inequality would be combining the not and equal functions.
(! (= 2 1)) ; => true

; More comparisons
(< 1 10) ; => true
(> 1 10) ; => false
; and their word counterpart.
(lesser 1 10) ; => true
(greater 1 10) ; => false

; Strings can be concatenated with +.
(+ "Hello " "world!") ; => "Hello world!"

; You can use JavaScript's comparative abilities.
(< 'a' 'b') ; => true
; ...and type coercion
(= '5' 5)
```

```
; The `at` or @ function will access characters in strings, starting at 0.
(at 0 'a') ; => 'a'
(@ 3 "foobar") ; => 'b'

; There is also the `null` and `undefined` variables.
null ; used to indicate a deliberate non-value
undefined ; user to indicate a value that hasn't been set

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; 2. Variables, Lists, and Dicts

; Variables are declared with the `def` or `let` functions.
; Variables that haven't been set will be `undefined`.
(def some_var 5)
; `def` will keep the variable in the global context.
; `let` will only have the variable inside its context, and has a weirder syntax.
(let ((a_var 5)) (+ a_var 5)) ; => 10
(+ a_var 5) ; = undefined + 5 => undefined

; Lists are arrays of values of any type.
; They basically are just forms without functions at the beginning.
(1 2 3) ; => [1, 2, 3] (JavaScript syntax)

; Dictionaries are Whip's equivalent to JavaScript 'objects' or Python 'dicts'
; or Ruby 'hashes': an unordered collection of key-value pairs.
{"key1" "value1" "key2" 2 3 3}

; Keys are just values, either identifier, number, or string.
(def my_dict {my_key "my_value" "my other key" 4})
; But in Whip, dictionaries get parsed like: value, whitespace, value;
; with more whitespace between each. So that means
{"key" "value"
"another key"
1234
}
; is evaluated to the same as
{"key" "value" "another key" 1234}

; Dictionary definitions can be accessed used the `at` function
; (like strings and lists.)
(@ "my other key" my_dict) ; => 4

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; 3. Logic and Control sequences

; The `if` function is pretty simple, though different than most imperative langs.
(if true "returned if first arg is true" "returned if first arg is false")
; => "returned if first arg is true"

; And for the sake of ternary operator legacy
; `?` is if's unused shortcut.
(? false true false) ; => false
```

```
; `both` is a logical 'and' statement, and `either` is a logical 'or'.
(both true true) ; => true
(both true false) ; => false
(either true false) ; => true
(either false false) ; => false
; And their shortcuts are
; & => both
; ^ => either
(& true true) ; => true
(^ false true) ; => true

;;;;;;;;;
; Lambdas

; Lambdas in Whip are declared with the `lambda` or `->` function.
; And functions are really just lambdas with names.
(def my_function (-> (x y) (+ (+ x y) 10)))
;         |       |    |          |
;         |       |    |    returned value(with scope containing argument vars)
;         |       | arguments
;         | lambda declaration function
;         |
;   name of the to-be-declared lambda

(my_function 10 10) ; = (+ (+ 10 10) 10) => 30

; Obviously, all lambdas by definition are anonymous and
; technically always used anonymously. Redundancy.
((lambda (x) x) 10) ; => 10

;;;;;;;;;;;;;;;;
; Comprehensions

; `range` or `..` generates a list of numbers for
; each number between its two args.
(range 1 5) ; => (1 2 3 4 5)
(.. 0 2)    ; => (0 1 2)

; `map` applies its first arg (which should be a lambda/function)
; to each item in the following arg (which should be a list)
(map (-> (x) (+ x 1)) (1 2 3)) ; => (2 3 4)

; Reduce
(reduce + (.. 1 5))
; equivalent to
((+ (+ (+ 1 2) 3) 4) 5)

; Note: map and reduce don't have shortcuts

; `slice` or `\` is just like JavaScript's .slice()
; But do note, it takes the list as the first argument, not the last.
(slice (.. 1 5) 2) ; => (3 4 5)
(\ (.. 0 100) -5) ; => (96 97 98 99 100)
```

```
; `append` or `<<` is self explanatory
(append 4 (1 2 3)) ; => (1 2 3 4)
(<< "bar" ("foo")) ; => ("foo" "bar")

; Length is self explanatory.
(length (1 2 3)) ; => 3
(_ "foobar") ; => 6

;;;;;;;;;;;;;;;
; Haskell fluff

; First item in list
(head (1 2 3)) ; => 1
; List from second to last elements in list
(tail (1 2 3)) ; => (2 3)
; Last item in list
(last (1 2 3)) ; => 3
; Reverse of `tail`
(init (1 2 3)) ; => (1 2)
; List from first to specified elements in list
(take 1 (1 2 3 4)) ; (1 2)
; Reverse of `take`
(drop 1 (1 2 3 4)) ; (3 4)
; Lowest value in list
(min (1 2 3 4)) ; 1
; Highest value in list
(max (1 2 3 4)) ; 4
; If value is in list or object
(elem 1 (1 2 3)) ; true
(elem "foo" {"foo" "bar"}) ; true
(elem "bar" {"foo" "bar"}) ; false
; Reverse list order
(reverse (1 2 3 4)) ; => (4 3 2 1)
; If value is even or odd
(even 1) ; => false
(odd 1) ; => true
; Split string into list of strings by whitespace
(words "foobar nachos cheese") ; => ("foobar" "nachos" "cheese")
; Join list of strings together.
(unwords ("foo" "bar")) ; => "foobar"
(pred 21) ; => 20
(succ 20) ; => 21
```

For more info, check out the repo

Wolfram

The Wolfram Language is the underlying language originally used in Mathematica, but now available for use in multiple contexts.

Wolfram Language has several interfaces: * The command line kernel interface on Raspberry Pi (just called *The Wolfram Language*) which runs interactively and can't produce graphical input. * *Mathematica* which is a rich text/maths editor with interactive Wolfram built in: pressing shift+Return on a "code cell" creates an output cell with the result, which is not dynamic * *Wolfram Workbench* which is Eclipse interfaced to the Wolfram Language backend

The code in this example can be typed in to any interface and edited with Wolfram Workbench. Loading directly into Mathematica may be awkward because the file contains no cell formatting information (which would make the file a huge mess to read as text) - it can be viewed/edited but may require some setting up.

```
(* This is a comment *)

(* In Mathematica instead of using these comments you can create a text cell
   and annotate your code with nicely typeset text and images *)

(* Typing an expression returns the result *)
2*2              (* 4 *)
5+8              (* 13 *)

(* Function Call *)
(* Note, function names (and everything else) are case sensitive *)
Sin[Pi/2]        (* 1 *)

(* Alternate Syntaxes for Function Call with one parameter *)
Sin@(Pi/2)       (* 1 *)
(Pi/2) // Sin    (* 1 *)

(* Every syntax in WL has some equivalent as a function call *)
Times[2, 2]      (* 4 *)
Plus[5, 8]       (* 13 *)

(* Using a variable for the first time defines it and makes it global *)
x = 5            (* 5 *)
x == 5           (* True, C-style assignment and equality testing *)
x                (* 5 *)
x = x + 5        (* 10 *)
x                (* 10 *)
Set[x, 20]       (* I wasn't kidding when I said EVERYTHING has a function equivalent *)
x                (* 20 *)

(* Because WL is based on a computer algebra system, *)
(* using undefined variables is fine, they just obstruct evaluation *)
cow + 5          (* 5 + cow, cow is undefined so can't evaluate further *)
cow + 5 + 10     (* 15 + cow, it'll evaluate what it can *)
%                (* 15 + cow, % fetches the last return *)
% - cow          (* 15, undefined variable cow cancelled out *)
moo = cow + 5    (* Beware, moo now holds an expression, not a number! *)

(* Defining a function *)
Double[x_] := x * 2    (* Note := to prevent immediate evaluation of the RHS
                          And _ after x to indicate no pattern matching constraints *)
Double[10]             (* 20 *)
Double[Sin[Pi/2]]      (* 2 *)
Double @ Sin @ (Pi/2)  (* 2, @-syntax avoids queues of close brackets *)
(Pi/2) // Sin // Double(* 2, //-syntax lists functions in execution order *)

(* For imperative-style programming use ; to separate statements *)
(* Discards any output from LHS and runs RHS *)
```

```
MyFirst[] := (Print@"Hello"; Print@"World")  (* Note outer parens are critical
                                                ;'s precedence is lower than := *)
MyFirst[]                                    (* Hello World *)

(* C-Style For Loop *)
PrintTo[x_] := For[y=0, y<x, y++, (Print[y])]  (* Start, test, incr, body *)
PrintTo[5]                                     (* 0 1 2 3 4 *)

(* While Loop *)
x = 0; While[x < 2, (Print@x; x++)]     (* While loop with test and body *)

(* If and conditionals *)
x = 8; If[x==8, Print@"Yes", Print@"No"]   (* Condition, true case, else case *)
Switch[x, 2, Print@"Two", 8, Print@"Yes"]  (* Value match style switch *)
Which[x==2, Print@"No", x==8, Print@"Yes"] (* Elif style switch *)

(* Variables other than parameters are global by default, even inside functions *)
y = 10             (* 10, global variable y *)
PrintTo[5]         (* 0 1 2 3 4 *)
y                  (* 5, global y clobbered by loop counter inside PrintTo *)
x = 20             (* 20, global variable x *)
PrintTo[5]         (* 0 1 2 3 4 *)
x                  (* 20, x in PrintTo is a parameter and automatically local *)

(* Local variables are declared using the Module metafunction *)
(* Version with local variable *)
BetterPrintTo[x_] := Module[{y}, (For[y=0, y<x, y++, (Print@y)])]
y = 20             (* Global variable y *)
BetterPrintTo[5]   (* 0 1 2 3 4 *)
y                  (* 20, that's better *)

(* Module actually lets us declare any scope we like *)
Module[{count}, count=0;        (* Declare scope of this variable count *)
  (IncCount[] := ++count);      (* These functions are inside that scope *)
  (DecCount[] := --count)]
count              (* count - global variable count is not defined *)
IncCount[]         (* 1, using the count variable inside the scope *)
IncCount[]         (* 2, incCount updates it *)
DecCount[]         (* 1, so does decCount *)
count              (* count - still no global variable by that name *)

(* Lists *)
myList = {1, 2, 3, 4}     (* {1, 2, 3, 4} *)
myList[[1]]               (* 1 - note list indexes start at 1, not 0 *)
Map[Double, myList]       (* {2, 4, 6, 8} - functional style list map function *)
Double /@ myList          (* {2, 4, 6, 8} - Abbreviated syntax for above *)
Scan[Print, myList]       (* 1 2 3 4 - imperative style loop over list *)
Fold[Plus, 0, myList]     (* 10 (0+1+2+3+4) *)
FoldList[Plus, 0, myList] (* {0, 1, 3, 6, 10} - fold storing intermediate results *)
Append[myList, 5]         (* {1, 2, 3, 4, 5} - note myList is not updated *)
Prepend[myList, 5]        (* {5, 1, 2, 3, 4} - add "myList = " if you want it to be *)
Join[myList, {3, 4}]      (* {1, 2, 3, 4, 3, 4} *)
myList[[2]] = 5          (* {1, 5, 3, 4} - this does update myList *)
```

```
(* Associations, aka Dictionaries/Hashes *)
myHash = <|"Green" -> 2, "Red" -> 1|>   (* Create an association *)
myHash[["Green"]]                       (* 2, use it *)
myHash[["Green"]] := 5                  (* 5, update it *)
myHash[["Puce"]] := 3.5                 (* 3.5, extend it *)
KeyDropFrom[myHash, "Green"]            (* Wipes out key Green *)
Keys[myHash]                            (* {Red} *)
Values[myHash]                          (* {1} *)

(* And you can't do any demo of Wolfram without showing this off *)
Manipulate[y^2, {y, 0, 20}] (* Return a reactive user interface that displays y^2
                               and allows y to be adjusted between 0-20 with a slider.
                               Only works on graphical frontends *)
```

Ready For More?

- Wolfram Language Documentation Center

Xml

XML is a markup language designed to store and transport data. It is supposed to be both human readable and machine readable.

Unlike HTML, XML does not specify how to display or to format data, it just carries it.

Distinctions are made between the **content** and the **markup**. In short, content could be anything, markup is defined.

Some definitions and introductions

XML Documents are basically made up of *elements* which can have *attributes* describing them and may contain some textual content or more elements as its children. All XML documents must have a root element, which is the ancestor of all the other elements in the document.

XML Parsers are designed to be very strict, and will stop parsing malformed documents. Therefore it must be ensured that all XML documents follow the XML Syntax Rules.

```
<!-- This is a comment. It must not contain two consecutive hyphens (-). -->
<!-- Comments can span
  multiple lines -->

<!-- Elements -->
<!-- An element is a basic XML component. There are two types, empty: -->
<element1 attribute="value" /> <!-- Empty elements do not hold any content -->
<!-- and non-empty: -->
<element2 attribute="value">Content</element2>
<!-- Element names may only contain alphabets and numbers. -->

<empty /> <!-- An element either consists an empty element tag… -->
<!-- …which does not hold any content and is pure markup. -->

<notempty> <!-- Or, it consists of a start tag… -->
  <!-- …some content… -->
```

```xml
</notempty> <!-- and an end tag. -->

<!-- Element names are case sensitive. -->
<element />
<!-- is not the same as -->
<eLEMENT />

<!-- Attributes -->
<!-- An attribute is a key-value pair and exists within an element. -->
<element attribute="value" another="anotherValue" many="space-separated list" />
<!-- An attribute may appear only once in an element. It holds just one value.
  Common workarounds to this involve the use of space-separated lists. -->

<!-- Nesting elements -->
<!-- An element's content may include other elements: -->
<parent>
  <child>Text</child>
  <emptysibling />
</parent>
<!-- Standard tree nomenclature is followed. Each element being called a node.
  An ancestor a level up is the parent, descendants a level down are children.
  Elements within the same parent element are siblings. -->

<!-- XML preserves whitespace. -->
<child>
  Text
</child>
<!-- is not the same as -->
<child>Text</child>
```

An XML document

This is what makes XML versatile. It is human readable too. The following document tells us that it defines a bookstore which sells three books, one of which is Learning XML by Erik T. Ray. All this without having used an XML Parser yet.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is called an XML prolog. Optional, but recommended. -->
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="CHILDREN">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="WEB">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

Well-formedness and Validation

A XML document is *well-formed* if it is syntactically correct. However, it is possible to add more constraints to the document, using Document Type Definitions (DTDs). A document whose elements are attributes are declared in a DTD and which follows the grammar specified in that DTD is called *valid* with respect to that DTD, in addition to being well-formed.

```
<!-- Declaring a DTD externally: -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookstore SYSTEM "Bookstore.dtd">
<!-- Declares that bookstore is our root element and 'Bookstore.dtd' is the path
  to our DTD file. -->
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
</bookstore>

<!-- The DTD file: -->
<!ELEMENT bookstore (book+)>
<!-- The bookstore element may contain one or more child book elements. -->
<!ELEMENT book (title, price)>
<!-- Each book must have a title and a price as its children. -->
<!ATTLIST book category CDATA "Literature">
<!-- A book should have a category attribute. If it doesn't, its default value
  will be 'Literature'. -->
<!ELEMENT title (#PCDATA)>
<!-- The element title must only contain parsed character data. That is, it may
  only contain text which is read by the parser and must not contain children.
  Compare with CDATA, or character data. -->
<!ELEMENT price (#PCDATA)>
]>

<!-- The DTD could be declared inside the XML file itself.-->

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE bookstore [
<!ELEMENT bookstore (book+)>
<!ELEMENT book (title, price)>
<!ATTLIST book category CDATA "Literature">
<!ELEMENT title (#PCDATA)>
<!ELEMENT price (#PCDATA)>
]>

<bookstore>
```

```
  <book category="COOKING">
    <title>Everyday Italian</title>
    <price>30.00</price>
  </book>
</bookstore>
```

DTD Compatibility and XML Schema Definitions

Support for DTDs is ubiquitous because they are so old. Unfortunately, modern XML features like namespaces are not supported by DTDs. XML Schema Definitions (XSDs) are meant to replace DTDs for defining XML document grammar.

Resources

- Validate your XML

Further Reading

- XML Schema Definitions Tutorial
- DTD Tutorial
- XML Tutorial
- Using XPath queries to parse XML

Yaml

YAML is a data serialisation language designed to be directly writable and readable by humans.

It's a strict superset of JSON, with the addition of syntactically significant newlines and indentation, like Python. Unlike Python, however, YAML doesn't allow literal tab characters at all.

```
# Comments in YAML look like this.

################
# SCALAR TYPES #
################

# Our root object (which continues for the entire document) will be a map,
# which is equivalent to a dictionary, hash or object in other languages.
key: value
another_key: Another value goes here.
a_number_value: 100
# If you want to use number 1 as a value, you have to enclose it in quotes,
# otherwise, YAML parser will assume that it is a boolean value of true.
scientific_notation: 1e+12
boolean: true
null_value: null
key with spaces: value
# Notice that strings don't need to be quoted. However, they can be.
however: "A string, enclosed in quotes."
"Keys can be quoted too.": "Useful if you want to put a ':' in your key."

# Multiple-line strings can be written either as a 'literal block' (using |),
```

```yaml
# or a 'folded block' (using '>').
literal_block: |
    This entire block of text will be the value of the 'literal_block' key,
    with line breaks being preserved.

    The literal continues until de-dented, and the leading indentation is
    stripped.

        Any lines that are 'more-indented' keep the rest of their indentation -
        these lines will be indented by 4 spaces.
folded_style: >
    This entire block of text will be the value of 'folded_style', but this
    time, all newlines will be replaced with a single space.

    Blank lines, like above, are converted to a newline character.

        'More-indented' lines keep their newlines, too -
        this text will appear over two lines.

####################
# COLLECTION TYPES #
####################

# Nesting is achieved by indentation.
a_nested_map:
    key: value
    another_key: Another Value
    another_nested_map:
        hello: hello

# Maps don't have to have string keys.
0.25: a float key

# Keys can also be complex, like multi-line objects
# We use ? followed by a space to indicate the start of a complex key.
? |
    This is a key
    that has multiple lines
: and this is its value

# YAML also allows mapping between sequences with the complex key syntax
# Some language parsers might complain
# An example
? - Manchester United
  - Real Madrid
: [ 2001-01-01, 2002-02-02 ]

# Sequences (equivalent to lists or arrays) look like this:
a_sequence:
    - Item 1
    - Item 2
    - 0.5 # sequences can contain disparate types.
    - Item 4
    - key: value
```

```yaml
      another_key: another_value
    -
        - This is a sequence
        - inside another sequence

# Since YAML is a superset of JSON, you can also write JSON-style maps and
# sequences:
json_map: {"key": "value"}
json_seq: [3, 2, 1, "takeoff"]

#######################
# EXTRA YAML FEATURES #
#######################

# YAML also has a handy feature called 'anchors', which let you easily duplicate
# content across your document. Both of these keys will have the same value:
anchored_content: &anchor_name This string will appear as the value of two keys.
other_anchor: *anchor_name

# Anchors can be used to duplicate/inherit properties
base: &base
    name: Everyone has same name

foo: &foo
    <<: *base
    age: 10

bar: &bar
    <<: *base
    age: 20

# foo and bar would also have name: Everyone has same name

# YAML also has tags, which you can use to explicitly declare types.
explicit_string: !!str 0.5
# Some parsers implement language specific tags, like this one for Python's
# complex number type.
python_complex_number: !!python/complex 1+2j

# We can also use yaml complex keys with language specific tags
? !!python/tuple [5, 7]
: Fifty Seven
# Would be {(5, 7): 'Fifty Seven'} in python

####################
# EXTRA YAML TYPES #
####################

# Strings and numbers aren't the only scalars that YAML can understand.
# ISO-formatted date and datetime literals are also parsed.
datetime: 2001-12-15T02:59:43.1Z
datetime_with_spaces: 2001-12-14 21:59:43.10 -5
date: 2002-12-14
```

```
# The !!binary tag indicates that a string is actually a base64-encoded
# representation of a binary blob.
gif_file: !!binary |
    R0lGODlhDAAMAIQAAP//9/X17unp5WZmZgAAAOfn515eXvPz7Y6OjuDg4J+fn5
    OTk6enp56enmlpaWNjY6Ojo4SEhP/++f/++f/++f/++f/++f/++f/++f/++f/+
    +f/++f/++f/++f/++SH+Dk1hZGUgd2l0aCBHSU1QACwAAAAADAAMAAAFLC
    AgjoEwnuNAFOhpEMTRiggcz4BNJHrv/zCFcLiwMWYNG84BwwEeECcgggoBADs=

# YAML also has a set type, which looks like this:
set:
    ? item1
    ? item2
    ? item3

# Like Python, sets are just maps with null values; the above is equivalent to:
set2:
    item1: null
    item2: null
    item3: null
```

Zfs

ZFS is a rethinking of the storage stack, combining traditional file systems as well as volume managers into one cohesive tool. ZFS has some specific teminology that sets it appart from more traditional storage systems, however it has a great set of features with a focus on usability for systems administrators.

ZFS Concepts

Virtual Devices

A VDEV is similar to a raid device presented by a RAID card, there are several different types of VDEV's that offer various advantages, including redundancy and speed. In general VDEV's offer better reliability and safety than a RAID card. It is discouraged to use a RAID setup with ZFS, as ZFS expects to directly manage the underlying disks.

Types of VDEV's * stripe (a single disk, no redundancy) * mirror (n-way mirrors supported) * raidz * raidz1 (1-disk parity, similar to RAID 5) * raidz2 (2-disk parity, similar to RAID 6) * raidz3 (3-disk parity, no RAID analog) * disk * file (not recommended for production due to another filesystem adding unnecessary layering)

Your data is striped across all the VDEV's present in your Storage Pool, so more VDEV's will increase your IOPS.

Storage Pools

ZFS uses Storage Pools as an abstraction over the lower level storage provider (VDEV), allow you to separate the user visable file system from the physcal layout.

ZFS Dataset

ZFS datasets are analagous to traditional filesystems but with many more features. They provide many of ZFS's advantages. Datasets support Copy on Write snapshots, quota's, compression and deduplication.

Limits

One directory may contain up to 2^48 files, up to 16 exabytes each. A single storage pool can contain up to 256 zettabytes (2^78) of space, and can be striped across 2^64 devices. A single host can have 2^64 storage pools. The limits are huge.

Commands

Storage Pools

Actions: * List * Status * Destroy * Get/Set properties

List zpools

```
## Create a raidz zpool
$ zpool create bucket raidz1 gpt/zfs0 gpt/zfs1 gpt/zfs2

## List ZPools
$ zpool list
NAME    SIZE  ALLOC   FREE  EXPANDSZ   FRAG    CAP  DEDUP  HEALTH  ALTROOT
zroot   141G   106G  35.2G         -    43%    75%  1.00x  ONLINE  -

## List detailed information about a specific zpool
$ zpool list -v zroot
NAME                                     SIZE  ALLOC   FREE  EXPANDSZ   FRAG    CAP  DEDUP HEALTH  ALTRO
zroot                                    141G   106G  35.2G         -    43%    75%  1.00x ONLINE  -
  gptid/c92a5ccf-a5bb-11e4-a77d-001b2172c655   141G   106G  35.2G         -    43%    75%
```

Status of zpools

```
## Get status information about zpools
$ zpool status
  pool: zroot
 state: ONLINE
  scan: scrub repaired 0 in 2h51m with 0 errors on Thu Oct  1 07:08:31 2015
config:

        NAME                                          STATE     READ WRITE CKSUM
        zroot                                         ONLINE       0     0     0
          gptid/c92a5ccf-a5bb-11e4-a77d-001b2172c655  ONLINE       0     0     0

errors: No known data errors

## Scrubbing a zpool to correct any errors
$ zpool scrub zroot
$ zpool status -v zroot
  pool: zroot
 state: ONLINE
  scan: scrub in progress since Thu Oct 15 16:59:14 2015
        39.1M scanned out of 106G at 1.45M/s, 20h47m to go
        0 repaired, 0.04% done
config:

        NAME                                          STATE     READ WRITE CKSUM
        zroot                                         ONLINE       0     0     0
          gptid/c92a5ccf-a5bb-11e4-a77d-001b2172c655  ONLINE       0     0     0
```

```
errors: No known data errors
```

Properties of zpools

```
## Getting properties from the pool properties can be user set or system provided.
$ zpool get all zroot
NAME   PROPERTY                       VALUE                          SOURCE
zroot  size                           141G                           -
zroot  capacity                       75%                            -
zroot  altroot                        -                              default
zroot  health                         ONLINE                         -
...

## Setting a zpool property
$ zpool set comment="Storage of mah stuff" zroot
$ zpool get comment
NAME   PROPERTY  VALUE                 SOURCE
tank   comment   -                     default
zroot  comment   Storage of mah stuff  local
```

Remove zpool

```
$ zpool destroy test
```

Datasets

Actions: * Create * List * Rename * Delete * Get/Set properties

Create datasets

```
## Create dataset
$ zfs create tank/root/data
$ mount | grep data
tank/root/data on /data (zfs, local, nfsv4acls)

## Create child dataset
$ zfs create tank/root/data/stuff
$ mount | grep data
tank/root/data on /data (zfs, local, nfsv4acls)
tank/root/data/stuff on /data/stuff (zfs, local, nfsv4acls)


## Create Volume
$ zfs create -V zroot/win_vm
$ zfs list zroot/win_vm
NAME                 USED  AVAIL  REFER  MOUNTPOINT
tank/win_vm         4.13G  17.9G    64K  -
```

List datasets

```
## List all datasets
$ zfs list
NAME                                                                       USED  AVAIL  REFER  MOUNTPOI
zroot                                                                      106G  30.8G   144K  none
zroot/ROOT                                                                18.5G  30.8G   144K  none
zroot/ROOT/10.1                                                              8K  30.8G  9.63G  /
```

```
zroot/ROOT/default                                                        18.5G  30.8G  11.2G  /
zroot/backup                                                              5.23G  30.8G   144K  none
zroot/home                                                                 288K  30.8G   144K  none
...

## List a specific dataset
$ zfs list zroot/home
NAME         USED  AVAIL  REFER  MOUNTPOINT
zroot/home   288K  30.8G   144K  none

## List snapshots
$ zfs list -t snapshot
zroot@daily-2015-10-15                                                                  0      -   144K
zroot/ROOT@daily-2015-10-15                                                             0      -   144K
zroot/ROOT/default@daily-2015-10-15                                                     0      -  24.2G
zroot/tmp@daily-2015-10-15                                                           124K      -   708M
zroot/usr@daily-2015-10-15                                                              0      -   144K
zroot/home@daily-2015-10-15                                                             0      -  11.9G
zroot/var@daily-2015-10-15                                                           704K      -  1.42G
zroot/var/log@daily-2015-10-15                                                       192K      -   828K
zroot/var/tmp@daily-2015-10-15                                                          0      -   152K
```

Rename datasets

```
$ zfs rename tank/root/home tank/root/old_home
$ zfs rename tank/root/new_home tank/root/home
```

Delete dataset

```
## Datasets cannot be deleted if they have any snapshots
zfs destroy tank/root/home
```

Get / set properties of a dataset

```
## Get all properties
$ zfs get all  zroot/usr/home
NAME            PROPERTY              VALUE                  SOURCE
zroot/home      type                  filesystem             -
zroot/home      creation              Mon Oct 20 14:44 2014  -
zroot/home      used                  11.9G                  -
zroot/home      available             94.1G                  -
zroot/home      referenced            11.9G                  -
zroot/home      mounted               yes                    -
...

## Get property from dataset
$ zfs get compression zroot/usr/home
NAME            PROPERTY     VALUE     SOURCE
zroot/home      compression  off       default

## Set property on dataset
$ zfs set compression=gzip-9 mypool/lamb

## Get a set of properties from all datasets
$ zfs list -o name,quota,reservation
NAME                                                               QUOTA  RESERV
zroot                                                               none    none
```

```
zroot/ROOT                                                    none    none
zroot/ROOT/default                                            none    none
zroot/tmp                                                     none    none
zroot/usr                                                     none    none
zroot/home                                                    none    none
zroot/var                                                     none    none
...
```

Snapshots

ZFS snapshots are one of the things about zfs that are a really big deal

- The space they take up is equal to the difference in data between the filesystem and its snapshot
- Creation time is only seconds
- Recovery is as fast as you can write data.
- They are easy to automate.

Actions: * Create * Delete * Rename * Access snapshots * Send / Receive * Clone

Create snapshots

“‘bash ## Create a snapshot of a single dataset zfs snapshot tank/home/sarlalian@now

Create a snapshot of a dataset and its children

$ zfs snapshot -r tank/home@now $ zfs list -t snapshot NAME USED AVAIL REFER MOUNTPOINT tank/home@now 0 - 26K - tank/home/sarlalian@now 0 - 259M - tank/home/alice@now 0 - 156M - tank/home/bob@now 0 - 156M - ...

Destroy snapshots

```
## How to destroy a snapshot
$ zfs destroy tank/home/sarlalian@now

## Delete a snapshot on a parent dataset and its children
$ zfs destroy -r tank/home/sarlalian@now
```

Renaming Snapshots

```
## Rename a snapshot
$ zfs rename tank/home/sarlalian@now tank/home/sarlalian@today
$ zfs rename tank/home/sarlalian@now today

## zfs rename -r tank/home@now @yesterday
```

Accessing snapshots

```
## CD Into a snapshot directory
$ cd /home/.zfs/snapshot/
```

Sending and Receiving

```
## Backup a snapshot to a file
$ zfs send tank/home/sarlalian@now | gzip > backup_file.gz

## Send a snapshot to another dataset
$ zfs send tank/home/sarlalian@now | zfs recv backups/home/sarlalian

## Send a snapshot to a remote host
$ zfs send tank/home/sarlalian@now | ssh root@backup_server 'zfs recv tank/home/sarlalian'
```

```
## Send full dataset with snapshos to new host
$ zfs send -v -R tank/home@now | ssh root@backup_server 'zfs recv tank/home'
```

Cloneing Snapshots

```
## Clone a snapshot
$ zfs clone tank/home/sarlalian@now tank/home/sarlalian_new

## Promoting the clone so it is no longer dependent on the snapshot
$ zfs promote tank/home/sarlalian_new
```

Putting it all together

This following a script utilizing FreeBSD, jails and ZFS to automate provisioning a clean copy of a mysql staging database from a live replication slave.

```
#!/bin/sh

echo "==== Stopping the staging database server ===="
jail -r staging

echo "==== Cleaning up existing staging server and snapshot ===="
zfs destroy -r zroot/jails/staging
zfs destroy zroot/jails/slave@staging

echo "==== Quiescing the slave database ===="
echo "FLUSH TABLES WITH READ LOCK;" | /usr/local/bin/mysql -u root -pmyrootpassword -h slave

echo "==== Snapshotting the slave db filesystem as zroot/jails/slave@staging ===="
zfs snapshot zroot/jails/slave@staging

echo "==== Starting the slave database server ===="
jail -c slave

echo "==== Cloning the slave snapshot to the staging server ===="
zfs clone zroot/jails/slave@staging zroot/jails/staging

echo "==== Installing the staging mysql config ===="
mv /jails/staging/usr/local/etc/my.cnf /jails/staging/usr/local/etc/my.cnf.slave
cp /jails/staging/usr/local/etc/my.cnf.staging /jails/staging/usr/local/etc/my.cnf

echo "==== Setting up the staging rc.conf file ===="
mv /jails/staging/etc/rc.conf.local /jails/staging/etc/rc.conf.slave
mv /jails/staging/etc/rc.conf.staging /jails/staging/etc/rc.conf.local

echo "==== Starting the staging db server ===="
jail -c staging

echo "==== Make sthe staging database not pull from the master ===="
echo "STOP SLAVE;" | /usr/local/bin/mysql -u root -pmyrootpassword -h staging
echo "RESET SLAVE;" | /usr/local/bin/mysql -u root -pmyrootpassword -h staging
```

Additional Reading

- BSDNow's Crash Course on ZFS
- FreeBSD Handbook on ZFS
- BSDNow's Crash Course on ZFS
- Oracle's Tuning Guide
- OpenZFS Tuning Guide
- FreeBSD ZFS Tuning Guide