What is Ruby?

- Everything is an object.
- Everything can be changed: method can be added to classes at runtime, for example.
- There’s no “compile-time”: everything happens at runtime.
- Variables have no type, they can contain different kinds of objects at different times.
- Classes are not “types” the way they are in Java — A class is identified by the messages (method calls) it responds to.

Ducks!

- Create a class and a constructor (it’s called initialize).
- Instance variables start with @.
- Duck.new is a standard class (static) method that creates a new object.
- The class definition is actually executable: it’s executed at runtime and creates the class.
- The statements after the class definition are also executed, as the file is loaded.
Defining a class

class Duck
def initialize(name,type)
    @name = name
    @type = type
end
end
d1 = Duck.new("larry","rubber")
puts d1

Running Ruby

- Run like this:
  > ruby ducks.rb
- Or like this:
  > irb --prompt simple
  >> d1 = Duck.new("larry","rubber")
  >> puts d1
- irb is the interactive Ruby shell.

Running Ruby

- You can also (if you're on a Unix system) put your script in a file like this:
  #!/usr/local/bin/ruby
  puts "hello ducks!"
- Make the file executable, and then you can execute it like any other Unix program:
  > chmod a+rx hello.rb
  > hello.rb
  hello ducks!

Syntax

- No semi-colons, as long as you keep one statement per line.
- Comments start with a # and go to the end of the line.
  # This is a comment.
- You can leave out parentheses around method arguments (but don’t). These are the same:
  # This is a comment.
  puts("arg1","arg2")
  puts "arg1","arg2"
Adding a method

- Let’s add a new method, `quack!`
- Method names can end in `!` (typically for methods that change some data), `?` (methods that return true/false), and `=` (setter methods).
- Notice that we’re not actually editing the class definition, but simply adding another method at runtime!

```ruby
class Duck
  def quack!
    puts "quack!"
  end
end
d1.quack!
```

Method parameters

- Class names should start with an upper case letter, method names with a lower case.
- Add a parameter to `quack!`. The new definition replaces the old one. There’s no overloading (methods with different types/number of parameters are different) like in Java.

```ruby
class Duck
  def quack!(times)
    puts "quack! " * times
  end
end
d1.quack!(5)
```

Types

- Variables don’t have type, but objects have. You can ask an object’s type using `.class`.

```ruby
>> 5**57
=> 6938893903907228377647697925567626953125
>> 5.class
=> Fixnum
>> (5**57).class
=> Bignum
>> "duck!".class
=> String
>> r1 = Duck.new("larry","rubber")
>> r1.class
=> Duck
```

Arrays

- Arrays can contain any type of object.
- Arrays are indexed by integers, starting from 0.

```ruby
flock = [d1,Duck.new("ruby","rubber"),
         "roast duck"]
puts flock
puts flock[0]
puts flock[1..2]
flock[0] = "kao ya"
puts flock
```
Hashes

Hashtables are indexed by, well, anything. You can map one object to any other kind of object.

```ruby
flock = {
  d1 => "hot",
  Duck.new("ruby","rubber") => "cute",
  "roast duck" => "tasty"
}
puts flock
```

Class methods and variables

Class variables start with `@@`. Class methods start with the class name followed by a . (period).

```ruby
class Duck
  @@count = 0
  def initialize(name,type)
    @name = name
    @type = type
    @@count += 1
  end
  def Duck.howMany
    return @@count
  end
end
d1 = Duck.new("larry","rubber")
```

Many classes define the `<<`-operator. For strings, it appends a value onto the end of the string. For arrays, it adds an element to the end of the array.

```ruby
s = "yo"
s << ",dude"
a = [1,2,3]
a << "ducks are cute as can be!"
```

...you’re the one...
Freezing Objects

Freezing objects

You can freeze an object to prevent someone from modifying it.

class Bird
  def initialize(name, age)
    @name = name; @age = age
  end
  attr_writer :age
end

h = Bird.new("huey", 8)
h.age = 9
h.freeze
h.age = 10
puts h

Freezing classes

As we’ve seen, class definitions are executable code, they essentially build the class at runtime, as they’re encountered.

So, since classes are objects, too, it makes sense that we can freeze them:

Bird.freeze

class Bird
  def new
  end
end

The three of us are twins!
The Ruby Typesystem

Compile-time type-checking

- Some call it *static checking, type safety, strict type-checking, strong typing*…
- It does have some advantages:
  1. You catch certain errors at compile time which you now can be sure won’t occur at run-time: arithmetic between the wrong types, wrong number of arguments to functions, etc.
  2. Simple errors that appear during code refactoring are easily caught and fixed.
  3. The more the compiler knows about your code, the better optimized code it can produce.
  4. Types serve as comments to the programmer, reminding him/her of what types of arguments a method was designed to take.

But:

1. Even Java has many errors which cannot be caught until run-time, such as `ClassCastException` and `ArrayBoundsException`.
2. Sometimes you need more flexibility, and it can be hard to work around a strict typechecker.

Run-time type-checking

- On the one hand, on the other hand:
  1. Less static type-checking may make programs faster to write, but it may also make them harder to maintain.
  2. A program is written once, but read and re-written many times — types can help someone unfamiliar with the code to understand it quicker.
The type of an object is defined by what it can do.

If an object walks like a duck, and talks like a duck, let's treat it like it's a duck!

We call this Duck Typing.

Here's a simple class that logs data by appending it to a file:

```ruby
class Logger
  def initialize()
    @f = File.open("logfile", "w")
  end

  def log(message)
    @f << message
  end
end

l = Logger.new
l.log("Ducks ahoy!
")
```

Or a string, which also knows the `<<` message.
Notice that the only change we had to make was to the statement that creates the `f`-object.

```ruby
class Logger
  def initialize()
    @f = ""
  end

  def log(message)
    @f << message
  end
end
```

Or an array, which also responds to the `<<` message:

```ruby
class Logger
  def initialize()
    @f = []
  end

  def log(message)
    @f << message
  end
end
```

l = Logger.new
l.log("Ducks ahoy!
")
Ruby Type “Checking”

If you absolutely want to check types, you should really check whether an object responds to a particular message or not:

class Logger
  def initialize()
    @f = {}
  end

  def log(message)
    unless @f.respond_to?(:<<)
      fail TypeError.new("log needs <<")
    end
    @f << message
  end
end

Of course, all we’re checking here is that there’s a method by the name of <<, we know nothing about what arguments it takes, what it does to those arguments, etc, so this is pretty weak checking.

Well-Travelled Ducks

From http://www.dailymail.co.uk/pages/live/articles/news/news.html?id=464768