A Tour of Ruby

... for Java programmers

Everything has a value

Everything has a **value**, which I'll show in a comment as follows (copying Matsumoto):

```
1234                          # => 1234

2+2                           # => 4

"Hello" + "World"             # => "HelloWorld"
```

We'll see more unusual examples of this later on.

Everything is a object; everything has a class

Everything is an object, it really is:

```
my_greeter.class        # => Greeter

1.class                 # => Fixnum

0.0.class               # => Float

“Hello”.class           # => String
```

and some particular cases:

```
true.class              # => TrueClass

false.class             # => FalseClass

nil.class               # => NilClass
```

Strings

Single-quoted strings are interpreted exactly ‘as is’

```
‘Hello’                    # => “Hello”

‘Hello\tJohn’              # => “Hello\tJohn”
```

Double-quoted strings are *interpolated* thus:

```
“Hello”                    # => “Hello”

“Hello\tJohn”              # => “Hello  John”: uses tab
character

name = ‘Mary’
“Hello #{name}”            # => “Hello Mary”
                           #{name} is interpreted
```

Strings are **Mutable**

What does this Java fragment output?

```
String s = "HELLO";
s = s.toLowerCase();          // convert to lowercase
System.out.println(s);        // what does this output?
```

In Java, Strings are **immutable**, i.e., they cannot be changed.

In Ruby, Strings are **mutable**:

```
s = "HELLO"                   # => "HELLO"
s.downcase!                   # convert to lowercase
puts s                        # what does this output?
```

If you want to use a String don't need it to be mutable, consider using a Ruby **symbol** instead.

Symbols

Ruby Symbols tend to mystify Java programmers...

They are essentially **immutable strings**. Consider this example using regular strings:

```
def time_of_day hours
    if hours <12
        "morning"
    else
        "afternoon"
    end
end

t1 = time_of_day 10        # t1 is “morning”
t2 = time_of_day 11        # t2 is “morning”
```

Methods don’t need a return statement! Last expression evaluated is returned.

Now there are two copies of the same string in memory. Call `time_of_day` 1000 times and you have 1000 copies of the same string. Waste of memory.

... example using Symbols

Redoing that example with symbols:

```
def time_of_day hours
    if hours <12
        :MORNING
    else
        :AFTERNOON
    end
end

t1 = time_of_day 10          # t1 is :morning
t2 = time_of_day 11          # t2 is :morning
```

There is only **one** copy of the symbol **:morning** in memory.

Use symbols where you might have used a **enumerated type** in Java.

Variables

Variables don't have a type; you simply start using them. They start with a small letter or underscore.

```
name = "John"          # name contains the string "John"
_num = 10              # => _num contains the integer 10
```

Some examples:

```
fileName               # Avoid camel case! Prefer file_name
Filename               # wrong, don't start with a capital
file_name              # Good Ruby style
no_of_windows          # good ruby style
___                    # technically ok, but not sensible!
```

Constants

Like variables, constants don't have a type. They must start with a **capital** letter.

```
PI = 3.14          # PI is a constant
...
PI = 3             # Ruby will issue a warning
```

By convention, constants use capital letters and underscores.

Classes are also constant, so they start with a capital letter. Using CamelCase for classnames is conventional.

```
GameOfNim          # typical Ruby class name
WindowDecorator    # typical Ruby class name
NUM_OF_TRIES       # typical Ruby constant name
```

Boolean etc.

true and false are built in keywords.

true and false are special objects. Not related to integer values as is the case in Java/C++.

nil is another special object that indicates absence of a value, like null in Java.

In expressions, nil and false mean are interpreted as false; **everything else evaluates to true**.

So Java/C++ heads please note:

0 is true!

Ranges

An integer range can be defined:

```
(1..10)
```

A range is an object (of course) and provides some very useful methods, e.g.,

```
if !(0..150).include? age
   puts “not a valid age!”
end
```

Ranges need not just contain integers, and provide many more facilities. Read more when required.

Classes

Classes are introduced with the keyword class.

```
class Employee
end
```

Later in course we'll look at inheritance and polymorphism, as well as interesting features like **mixins**.

Methods

Like methods in Java. By convention, the name of a method should contain lowercase letters, digits, operators and underscores. For now, we'll only consider **public** methods.

Methods are introduced using the keyword **def**:

```
class Employee
    def print
    end
end
```

Methods must have unique names in a class, so method overloading is not possible. We'll see later how to get around this using a variable argument list.

Methods can also be private or protected, with a somewhat different meaning to the Java equivalents. More later in the course.

We'll also look at **closures** later in the course.

The initialize Method

Like a constructor in Java.

```
class Employee
      def initialize
      end
end
```

A class can only have one initialize method.

The **initialize** method is automatically invoked when an object is created. You cannot invoke it directly, except from inside the class (it's a private method, as we'll see later).

A destructor, called **finalize**, is possible but seldom required.

Instance Variables

Like fields in Java. The name of an instance variable must start with @. All instance variables are private to the class in which they are defined:

```
class Employee
    def initialize name, salary
        @name = name
        @salary = salary
    end
    def print
        @silly = 1234
        puts "#{@name} earns #{@salary}."
    end
end
```

(Class variables names start with @@ -- these are like static fields in Java.)

Method Names

If a method returns a boolean, then by convention its name should finish with ?:

```
class Employee
    ...
    def highly_paid?
        @salary > 495000
    end
end
```

A method that changes an object's state is usually terminated with a !. Many methods have a bang and non-bang version, e.g.,

```
s =  "HELLO"
s.downcase          # => "hello": s is not changed
s.downcase!         # => "hello": s is changed
```

Getters and Setters...

To enable an instance variable to be read and set, you might do as follows:

```
class Employee
    ...
    def name                        # getter for name
        @name
    end
    def name= new_name              # setter for name; note syntax!
        @name=new_name
    end
end
```

So for an Employee referred to by emp, @name can be set and read this way:

```
emp.name = "John"       # invokes name= method
puts emp.name           # invokes name method
```

Think about this!

... and a shorthand

Rather than writing all that code for every instance variable you need to access, Ruby will generate it for you as follows:

```
class Employee
    attr_accessor :name
    ...
end
```

If you only want to read the instance variable, use **attr_reader**; if you only want to write it (seems odd), use **attr_writer**.

Operator Overloading is a cinch!

```
class Point
    attr_reader :x, :y

    def initialize x, y
        @x = x
        @y = y
    end

    def + point                    # defines the + operator for Points
        Point.new(@x+point.x, @y+point.y)
    end
end

p = Point.new 1, 1
q = Point.new 10, 10
r=p+q                    # invokes the method called +, r is (11, 11)
```

if

Simplest case:

if ***expression*** **then**
    code
end

The "then" is optional if followed by a newline.

code is executed if and only if *expression* evaluates to true.
Another form is:

if ***expression***
    code1
else
    code2
end

elsif

The example explains the meaning:

```
if x == 1
    name = “one”
elsif x == 2
    name = “two”
elsif x == 3
    name = “three”
else
    name = “many”
```

this can also be achieved with a case statement. See next slide.

case

Similar to Java. This example explains the meaning:

```
name = case x
    when 1 then “one”
    when 2 then “two”
    when 3 then “three”
    else “many”
end
```

Note how the case statement itself has a value.

loops

Ruby has several loop types. Where possible, use **iteration** (each, inject, map, ...) rather than a loop.

Here's the format of a regular while loop:

```
while expression do
    code
end
```

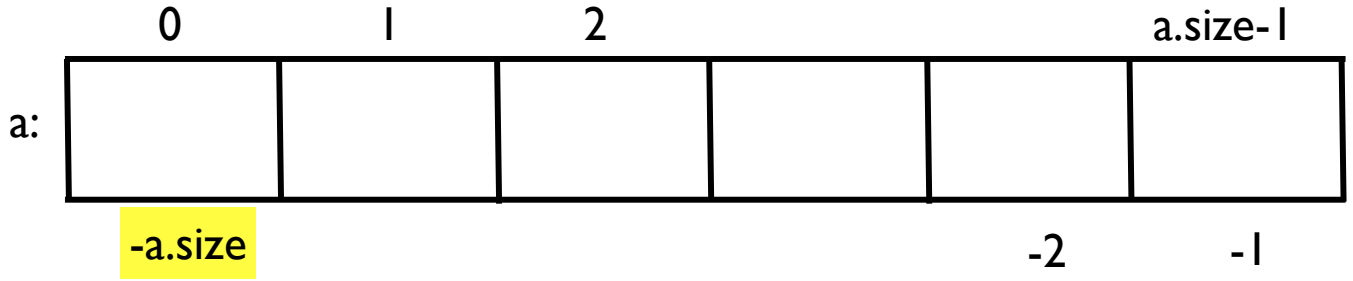
There are several other loop types as well. **break** and **next** have the same meaning in loops as in Java:

break: exit the loop immediately

next: start the next iteration of the loop immediately

Arrays

Arrays map integers to values. All arrays can be accessed with positive or negative indexes:



Some examples:

```
x = []                  # an empty array
y = Array.new           # an empty array
z = [12, 45, 764]       # a 3-element array
x[1] = 'hello'          # => [nil, 'hello']
z.pop                   # => [12, 45]
```

Hash Tables in Ruby

Arrays map integers to values

a[10] = “hello”

a[0] = 12

So arrays are indexed on integers

What if we want to index on something else?

e.g., to map names to phone numbers

or to map songs to artists...

For that, use a **hash table**. Like array, this is a built-in class in Ruby

Hash Tables

```
songs = {}                                  # an empty hash table

songs["Penny Lane"] = "Beatles"   # => {"Penny Lane" => "Beatles"}


songs["Dynamite"] = "Taylo Cruz"
# => {"Penny Lane" => "Beatles", "Dynamite" => "Taylo Cruz"}
```

We can also use the **{}** syntax in initialising a hash:

```
songs = {"Penny Lane" => "Beatles", "Dynamite" => "Taylo Cruz"}

songs["Penny Lane"]              # => "Beatles"

songs["Dynamite"]                # => "Taylo Cruz"
```

Code Blocks

There are two ways to denote a code block in Ruby:

Using braces:

```
{
    line 1
    line 2
}
```

Prefer braces for 1-line blocks, or if you **use** the return value of the block.

Using **do .. end**:

```
do
    line 1
    line 2
end
```

Prefer **do .. end** for multi-line blocks.

{} bind more tightly than **do .. end**, but this usually doesn't matter.

Iterators

Iterators are methods that execute a given block of code as many times as there are iterations. Simple example:

```
10.times {|i| puts i}
```

Explanation: We ask the integer object `10` to execute its `times` method using the block `{|i| puts i}`. The `times` method uses a counter that runs from 0..9. It passes this value to the block through the **block parameter** i.

Other similar iterators:

```
1.upto(10) {|i| puts i}
```

```
1.step(10, 3) { |i| puts i }
```

The each Iterator

The **each** iterator is ideal for processing arrays and hashes
Example:

```
[2, 3, 5, 7, 11, 13].each {|val| puts val}
```

Note that 1-line blocks are written using {}. Longer blocks are written using do..end:

```
my_array = []
...
my_array.each do |val|
    # process the current element, val
end
```

The each Iterator on hashes

When using the each iterator to traverse a hash, the block has to pick up both the value and key of the hash:

```
songs.each do |key, value|
    puts  "#{key} is performed by #{value}"
end
```

There are many iterators in Ruby. Other examples are each_value, each_index, map, inject, collect, select...

We'll see how to write your own iterators later in the course

Example: Iterating over hashes

```
class Encryptor
    def initialize
        @key={'a'=>'b', 'b'=>'c', 'c'=>'d', 'd'=>'a', ' '=> ' '}
    end
    def encrypt text
        cipher_text = ''
        text.each_char do |letter|
            cipher_text << @key[letter]
        end
        cipher_text
    end
end
plaintext = 'bad cab'
my_encryptor = Encryptor.new
cipher_text = my_encryptor.encrypt plaintext
puts cipher_text
```

Hash table to store encryption code

Look up iterators when you need them!

Summary

This has been a whistle-stop tour of the main features of Ruby.

Much of what we didn't cover is similar to Java.

More advanced topics we'll come to later in the course.

These slides are anything but exhaustive, so supplement them with extra reading.

Don't just read the slides passively. Run the code and play around with it.