A Tour of Ruby

... for Java programmers

Everything has a value

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

```
1234                            # => 1234

2 + 2                           # => 4

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

We'll see other examples of 'expressions having values' later.

Ruby is said to follow the
Principle of Least Astonishment.

Everything is a object; all objects have a class

(Almost) everything is an object, it really is.

Trying asking things what class they belong to by sending them the message **`class`**:

```
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
```

So there's no built-in Boolean class as you might have expected

Strings, Variables, Constants, Boolean and Ranges

Strings

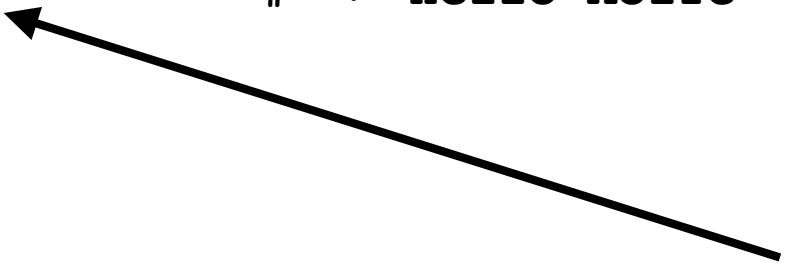
Single-quoted strings are interpreted exactly 'as is'

```
'Hello'                 # => Hello

'Hello\tSean'           # => Hello\tSean
```

Double-quoted strings are **interpolated** thus:

```
"Hello"                 # => Hello

"Hello\tSean"           # => Hello   Sean
```

```
name = 'Aoife'
"Hello #{name}"         # => Hello Aoife
```



Within a double-quoted string, code inside `#{...}` is executed.

Variables

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

```
name = 'John'          # name contains the string 'John'

_num = 10              # _num contains the integer 10
```

Not really 'contains'.
We'll return to this later.

Any variable can store anything, e.g

```
my_var = 'John'        # my_var contains the string 'John'

my_var = 99            # now my_var now contains the integer 99

my_var = nil           # now my_var now contains nil


my_var = Greeter.new # now my_var now contains a Greeter object
```

This may seem odd coming from Java, where all variables have a type that restricts what you can do with them.

Variable Names

Some examples of variable names:

```
fileName ✗          # Avoid camel case! Prefer file_name
Filename ✗          # Can't start with a capital letter
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 all capital letters and underscores (i.e., SCREAMING_SNAKE_CASE).

Class names also 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
```

Follow these convention in your own programs

Boolean etc.

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

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

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

So C/C++ programmers please note:

0 is interpreted as true!

Ranges

An integer range can be defined, e.g.

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

Parentheses not essential here

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 can contain more than just integers, e.g.,

```
('a'..'e').to_a    #=> ['a', 'b', 'c', 'd', 'e']
```

Ranges provide much other useful functionality. Read more when required.

More on **arrays** later

Input/Output

Input/Output

We've seen **`puts`** for performing output.

To get a line of text from the keyboard, use **`gets`** e.g.,

```
name = gets
```

`gets` will return the final newline character as well, so if you type **`'John'`**, **`name`** will contain **`'John\n'`**.

The string method **`chomp`** gets rid of the newline:

```
name = name.chomp
```

Now **`name`** will contain just **`'John'`**.

File Input

There are many ways to get input from a file in Ruby. We'll look at one of the easier and more flexible ways.

Say this is the file (**`products.txt`**) that is to be read:

```
Bag       100.00
Hat       10.00
Scarf      10.50
Tie       20.75
```

So it's a list of item names (strings) and their associated prices (floating-point values).

File Input

This code fragment reads the input file on previous slide:

```
1.  IO.foreach('products.txt') do |line|
2.    data = line.split
3.    name = data[0]
4.    price = data[1].to_f
      ...   # do something with name and price
    end
```

This code will make more sense later in the module.

1. Opens 'products.txt', and executes the block of code between the **`do .. end`** once for each line in the file. **`line`** represents the line being processed.

2. The **`split`** method splits the line into an array of strings.

3, 4. **`data[0]`** accesses the first element of the array; **`to_f`** converts the string to a floating point number (see also **`to_i`**).

Classes and Methods

Classes

Classes are introduced with the keyword **`class`**.

```
class Employee
   ...
   ...
end
```

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

Methods

Like methods in Java. Method names follow similar conventions to variable names.

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. You can get around this using a **variable argument list**.

Methods can be public, private or protected, with a somewhat different meaning than in Java (more later in the course).

For now, we'll only consider **public** methods.

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

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.

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 can also invoke it directly from inside the class (it's a private method, as we'll see later).

A destructor, called `finalize`, also exists but is 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 to_s
    return "#{@name} earns #{@salary}."
  end
end
```

(Class variable 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
```

The name of a 'dangerous' method should end with a ! if there exists a safe version of the method., e.g.,

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

Java-style Getters and Setters...

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

```
class Employee
  ...
  def get_name # Getter for name
    @name
  end
  def set_name(new_name) # Setter for name
    @name = new_name
  end
end
```

Don't do this!

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

```
emp.set_name('John')
puts emp.get_name
```

More Rubyesque Getters and Setters

To enable an instance variable to be read and set, it is more Rubyesque to 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
```

Don't do this either!

So for an **Employee** object referred to by **emp**, **@name** can now 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 want to grant access to, Ruby will generate it for you as follows:

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

Use this only if you need to access the instance variables from **outside** the class

So adding e.g. **`attr_accessor :count`** to a class is the same as adding methods called **`count`** and **`count=`**, as defined on previous slide.

If you only want to read the instance variable, use **`attr_reader`**; if you only want to write it, use **`attr_writer`**.

Do it this way!

Operator Overloading is easy!

```
class Point
  attr_reader :x, :y

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

  def +(other)
    Point.new(@x+other.x, @y+other.y)
  end
end
```

defines the **+** operator for **Point** objects

```
p = Point.new(1, 2)
q = Point.new(10, 10)
...
r = p + q
```

invokes the method called +; r now contains (11, 12)

self

Within a method, it may be necessary to refer to the **receiving object**, i.e. the object that is executing the method.

In Java, **`this`** refers to the receiving object.

In Ruby, **`self`** refers to the receiving object.

```
class Employee
  ...
  def ask_for_promotion?
    @manager.promote_employee?(self)
  end
  ...
end
```

Passes reference to this Employee object to promote_employee method in Manager

Class Variables and Methods

Class variables in Ruby are like static fields in Java.

A class variable is shared by all instances of the class.

In Ruby, their names start with '@@'.

Class methods (like static methods in Java) can access class variables.

Q: Can a static method access instance variables?

A: No, there is no instance to access!

Class methods are defined by prefixing the method name with `self.` Example on next slide.

Note that **class instance variables** are something else; we won't cover them

Example: counting instances

Say we want to count the number of instances of the Employee class that are created as the program executes.

```
class Employee
  @@no_of_instances = 0

  def self.how_many_instances
    @@no_of_instances
  end

  def initialize(name)
    @name = name
    @@no_of_instances += 1
  end
end
```

`@@no_of_instances` starts at zero…

Class method to return class variable, `@@no_of_instances`

incremented by 1 whenever new object created

Class methods can be invoked on the class, e.g.:

```
puts Employee.how_many_instances
```

Variables, Objects and Equality

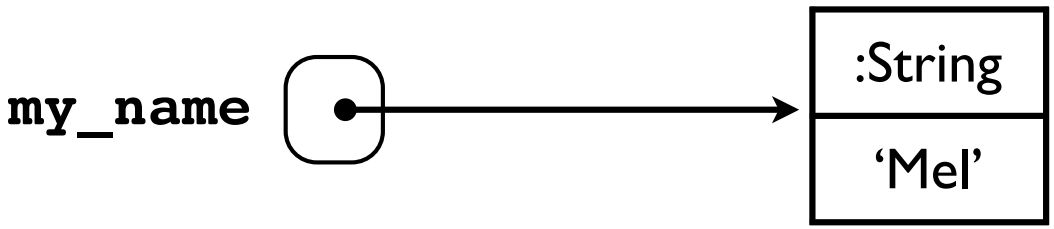
Variables and Objects in Ruby

What does this Ruby statement do?

```
my_name = String.new('Mel')
```

1. It introduces a new variable called **`my_name`**
2. It creates a new **`String`** object initialised to **`'Mel'`**
3. It sets **`my_name`** to refer to this new **`String`** object.

This can be visualised thus:



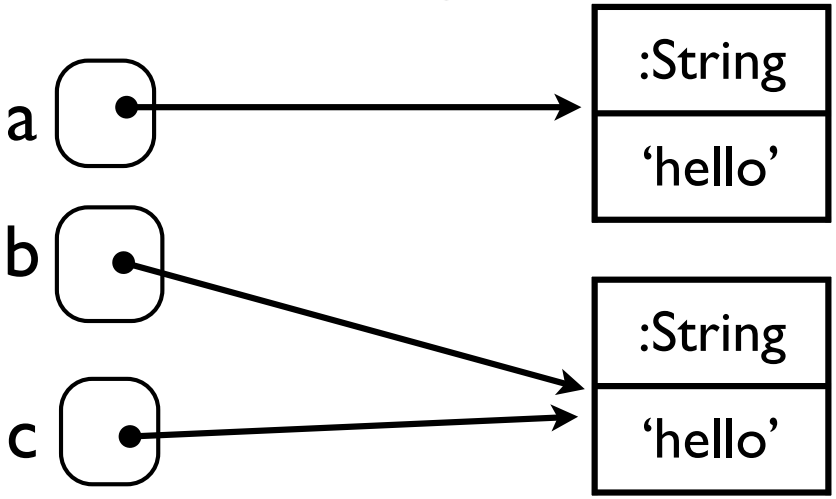
Keep this model in mind whenever confused about objects and variables in Ruby.

Object Equality: equal?

Understanding object equality is vital when learning a new language. The Ruby syntax is more-or-less the opposite of the Java syntax, so beware!

All objects can respond to the **equal?** method that checks if two variables refer to the **same object**, e.g.,

```
a = 'hello'
b = 'hello'
c = b
```



Are a and b equal?

a.equal?(b) # => false: a and b refer to different objects

Are b and c equal?

b.equal?(c) # => true: b and c refer to the same object

Object Equality: ==

The == operator is defined to have the same meaning as **`equal?`** by default. However…

Normally the == operator is **redefined** in a class to mean a **by-value comparison**, e.g., it has already been defined this way in the **`String`** class, so:

```
a = 'hello'
b = 'hello'
a.equal?(b) # => false: a and b are different objects
a == b      # => true:  b and c have equal values
```

So when you create a new class, consider implementing the == operator for comparison purposes.

If you don't, == will test if the receiver and argument refer to the same object.

Strings and Symbols

Java Strings are **Immutable**

What does this Java fragment output?

```
String s = 'HELLO';
s.toLowerCase();            // make lowercase
System.out.println(s);      // what does this print?
```

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

This is probably what the programmer intended:

```
String s = 'HELLO';
s = s.toLowerCase();          // make lowercase
System.out.println(s);        // prints 'hello'
```

Immutable strings are faster to operate on, and consume less memory, which is why Java provides them.

Ruby Strings are **Mutable**

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 and don't need it to be mutable, you should use a Ruby **symbol** instead….

Symbols

Ruby symbols tend to mystify Java programmers...

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 will have 1000 copies of the same string. Waste of memory.

This wastage would not happen in Java, due in **interning**

Symbols

Symbols are generally written with a colon followed by a non-quoted string, e.g.

```
:john

:morning

:action
```

Symbols are immutable, **interned** strings.

Interned means that no matter how many instances exist, only one copy of the object is stored in memory.

So what would the previous solution look like with symbols?

... 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.

When to use Symbols?

If you're using a string as an **identifier**, rather than **actual text**, you should probably use a symbol instead.

In a traffic simulator, would you use
`'red', 'amber', 'green'` or `:red, :amber, :green` ?

For people's names, would you use
`'aoife', 'sean', 'alice'` or `:aoife, :sean, :alice` ?

For months, would you use
`'january'`, `'february'`, etc. or `:january`, `:february`, etc.?

Also, where you might use a enumerated type in Java, prefer symbols, e.g. for compass directions you would use: `:north, :south, :east, :west`

Good blog on symbols: https://www.culttt.com/2015/04/22/what-are-symbols-in-ruby/

`if`, `case` and loops

The `if` Statement

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
```

Using `elsif`

The example explains the meaning:

```
if age <= 2
  person = :infant
elsif age <= 12
  person = :child
elsif age <= 70
  person = :adult
else
  person = :elder
end
```

This can also be achieved more succinctly with a **`case`** statement. See next slide.

case

Similar to Java. This example explains the meaning:

```
person =
   case age
   when 0..2 then :infant
   when 3..12 then :child
   when 13..70 then :adult
   else :elder
   end
```

The **when** clause can have a value or a range

Note how the case statement itself has a value.

(Yes, this is the recommended indentation for a case statement.)

loops

Ruby has several loop types. Where possible, use an **iterator** (each, inject, map, see later) rather than a loop.

Here's the format of a regular while loop:

```
while expression
  <loop body>
end
```

Ruby also has an **until** statement with same syntax.

There are several other loop types as well.

break has the same meaning in loops as in Java; **next** is the equivalent of continue in Java:

break: exit the loop immediately

next: start the next iteration of the loop immediately

Argument Passing

How Arguments are passed to a Method

What does this Ruby fragment output?

```
class FooBar
  def change_name(name)
    name = 'John'
  end
end

my_name = 'Mel'
my_foobar = FooBar.new

my_foobar.change_name(my_name)

puts my_name
```

How Arguments are passed to a Method

What does this Ruby fragment output?

```
class FooBar
    def change_name(name)
      name = name.upcase
    end
end

my_name = 'Mel'
my_foobar = FooBar.new

my_foobar.change_name(my_name)

puts my_name
```

How Arguments are passed to a Method

What does this Ruby fragment output?

```
class FooBar
    def change_name(name)
      name.upcase!
    end
end

my_name = 'Mel'
my_foobar = FooBar.new

my_foobar.change_name(my_name)

puts my_name
```

Pass-by-reference and pass-by-value

In general (not Ruby-specific), the two main ways to pass arguments to a method are:

- **pass-by-reference** and

- **pass-by-value**.

In **pass-by-reference** a reference (pointer) to the argument is passed into the method, so any changes to the argument within the method changes the object outside the method as well.

In **pass-by-value** a copy of the argument is passed into the method, so changes to the argument within the method have no effect on the object outside the method.

Which does Ruby use?

Pass-by-value is the cleaner approach and this is what C, Java, Ruby, and most languages, provide.

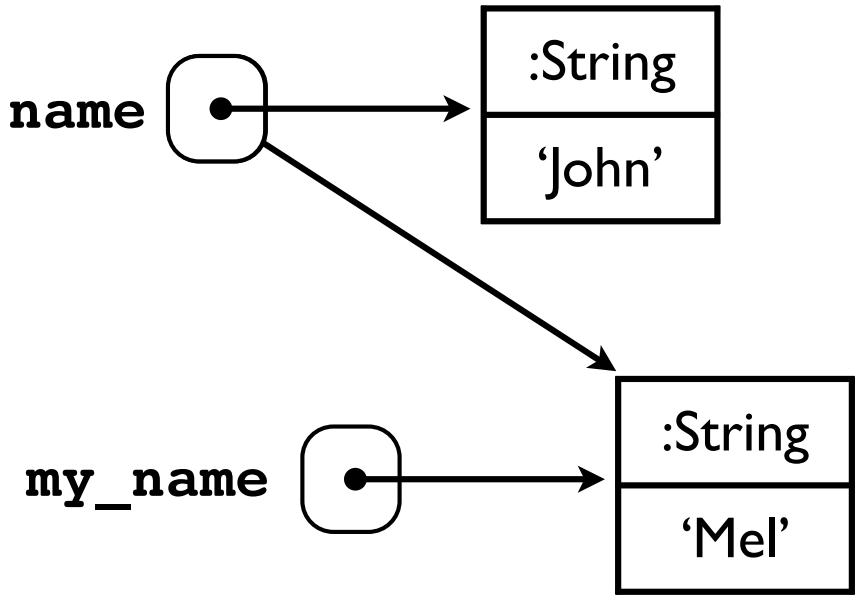
However, Ruby arguments are actually object references (pointers), so the effects of pass-by-value can be surprising!

This is explained further in the next slide.

Method Arguments Explained

So how does this Ruby fragment operate?

```
class FooBar
    def change_name(name)
        name = 'John'
    end
end

my_name = 'Mel'
my_foobar = FooBar.new

my_foobar.change_name(my_name)

puts my_name
```

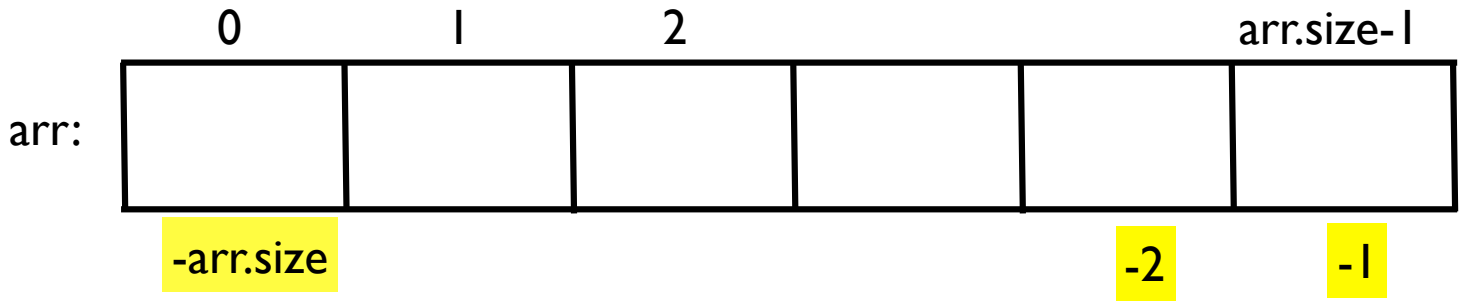


Using the animation on the right should make it clear what happens in each case.

Arrays, Hash Tables and Iterators

Arrays

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



Some examples:

```
x = []                  # an empty array

y = Array.new           # an empty array (avoid this syntax)

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*,

or we map from integers, but they are not contiguous starting at zero?

In such cases, use a **hash table**.

Read up on **hash tables** if you haven't seen them previously

Creating a Hash Table

Create an empty hash table:

```
songs = {}
```

'Penny Lane' is a Beatles song:

```
songs['Penny Lane'] = 'Beatles'
```

'I Will Follow' is a U2 song:

```
songs['I Will Follow'] = 'U2'
```

Now **songs** looks like this:

```
{'Penny Lane' => 'Beatles', 'I Will Follow' => 'U2'}
```

The above two assignments could be replaced with:

```
songs = {'Penny Lane' => 'Beatles', 'I Will Follow' => 'U2'}
```

Iterators

An **iterator** is a method that executes a given block of code, usually a number of times. Simple example:

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

Simplified Explanation: We ask the integer object `10` to execute its **`times`** method using the block **`{|i| puts i}`**.

The **`times`** method then 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 }
```

(Recall that 1-line blocks are written using **{}**)

Longer blocks are written using **do..end**, e.g.

```
my_array = [2, 3, 5, 7, 11, 13]
...
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, including:

```
each_value
each_index
map
inject
collect
select
...
```

We'll look at these further later in the module.

We'll also see in detail how to write your own iterators.

Example: Iterating over hashes

```
class Encryptor
  def initialize
    @key = {'a'=>'x', 'b'=>'s', 'c'=>'z', 'd'=>'t'}
  end

  def encrypt(text)
    cipher_text = ''
    text.each_char do |letter|
      cipher_text << @key[letter]
    end
    cipher_text
  end
end

plaintext = 'badcab'
my_encryptor = Encryptor.new
cipher_text = my_encryptor.encrypt(plaintext)
puts cipher_text
```

Hash table to store encryption code

Multiple Source Files

Handling Multiple Ruby Files

A small Ruby program can be written in a single file. This won't work for larger systems of course.

Normal practice is to put each class into its own file. E.g. for a class called **Employee**, name the file **employee.rb**.

Use **require_relative** when a class in one file uses a class defined in another file (like a Java **import**), e.g.

```
class Point
  …
end
```

point.rb

```
require_relative 'point.rb'

class Kangaroo
  def initialize
    start = Point.new(0, 0)
 end
  …
end
```

kangaroo.rb

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 not exhaustive, so supplement them with extra reading.

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