Intercepting calls to undefined methods

Whenever a call to an undefined method is made on an object, Ruby provides an option to intercept the call.

This is done by implementing the method `method_missing` within the class definition. Ruby passes as parameters the name of the method called and the arguments passed to it.

In the online SaaS edX lectures, you saw a use of `method_missing` to implement currency conversion.

Another interesting use of `method_missing` can be found on the Web by looking for “Roman method_missing”.

Define a module (or class) `Roman`. This class contains a `method_missing` method that intercepts calls to “class methods” that are undefined. It then tries to interpret the method name as a Roman numeral.

For example,

- evaluating `Roman.xix` calls the `xix` method of module `Roman`.
- `Roman` has no `xix` method, so `method_missing` is invoked with `:xix` as the argument.
- The `id2name` method of class `Symbol` is invoked on `:xix`, returning "xix".
- The "xix" is then parsed according to the rules for evaluating Roman numerals, and evaluates to 19.

Here is the code:

```ruby
# Define a module (or class)Roman
DIGITS = {
  'I' => 1, 'V' => 5, 'X' => 10, 'L' => 50,
  'C' => 100, 'D' => 500, 'M' => 1000,
}

def roman_to_integer(roman_string)
  # ...
prev = nil
roman_string.to_s.upcase.split(/\//).reverse.inject(0) do |
  running_sum, digit|
    if digit_value = DIGITS[digit]
      if prev && prev > digit_value
        running_sum -= digit_value
      else
        running_sum += digit_value
      end
    end
    prev = digit_value
  end
end
end

def method_missing(method)
  str = method.id2name
  roman_to_integer(str)
end
end

**Exercise:** Explain how the code works. Submit your explanations [here](#).

### Testing in Ruby

To illustrate how testing is performed in Ruby, let’s consider this class to convert Arabic numbers to Roman numerals.

```ruby
# This code has bugs
class Roman
  MAX_ROMAN = 4999

  def initialize(value)
    if value <= 0 || value > MAX_ROMAN
      fail "Roman values must be > 0 and <= #{MAX_ROMAN}"
    end
    @value = value
  end

  FACTORS = [['m', 1000], ['cm', 900], ['d', 500], ['cd', 400], ['c', 100], ['xc', 90],
```
Let's study how this code works.

- The `initialize` method is pretty clear; it checks to be sure that the value is in range for a Roman number and initializes the instance variable.

- It sets up an array of the values of each Roman digit, treating the “minuses” as part of the following character.

- After initializing the result to a null string, it gets in a loop, repeatedly dividing the value of the number by the value of the current Roman digit. Current Roman digits start out with the largest (m) and proceed to the smallest (i).

- The Ruby function `value.divmod(factor)` returns an array containing the quotient and modulus obtained by dividing `value` by `factor`.

- If there is a non-zero quotient, the “code” for the digit is appended to the Roman number.

Let's start out by testing this code on various numbers, e.g., 1, 9, 164.

```ruby
require_relative 'roman'

r = Roman.new(1)
fail "'I' expected" unless r.to_s == "I"
```
require_relative refers to code you’ve written; it gives a pathname relative to the current directory.

By contrast, require is used for system files.

What is a disadvantage of running tests in this way?

Testing using Test::Unit

Test::Unit is a testing framework that comes with Ruby. Test::Unit is similar to JUnit for Java. Other Ruby test frameworks, like RSpec and Cucumber, are built on top of it.

You normally put all your tests in a special subdirectory of the project, called (e.g.) test. In RubyMine, you right-click on the directory icon and select “Mark directory as > Test source root.”

This enables special options for the folder, such as “Run all tests”.

It predefines two stub methods that you can extend:

- **setup** This allows you to, e.g., open a database connection, or set up objects that you will test with.

- **teardown** This is run after the tests and can, e.g., close a database connection.

How shall we test to_s? What are some conditions that we should test for?

Eventually, we find the bug and correct the code (see above, in blue).
Adding Roman numbers
It’s also possible to add Roman numbers by overloading the + operator.

In its simplest form, the code for this is

```ruby
def +(other)
    Roman.new(@value + other.value)
end
```

However, this causes a protection violation. What is it? How can we correct it?

Then we can enhance the code to allow adding an integer to a Roman.

Our revised code is as follows:

```ruby
require_relative 'roman'

class Roman

    attr_reader :value
    protected :value

    def +(other)
        if Roman === other
            other = other.value
        end

        Roman.new(@value + other)
    end
end
```

Ruby on Rails
Ruby on Rails is a Web application framework for Ruby. It was first released to the public in July 2004.
Within months, it was a widely used development environment. Many multinational corporations are using it to create Web applications.

It is the standard Web-development framework for Ruby.

**Model/View/Controller**

All Rails applications are implemented using the Model/View/Controller (MVC) architecture.

*Models* are objects that represent the components of an application that perform information processing in the problem domain.

Models should represent “real world” entities:
- physical entities like a valve in a control system, or
- conceptual entities like a department in an office, or a contract between two businesses.

*Views* are objects that display some aspect of the model. They are the output mechanism for the models.

You could have a view that represents:
- the position of the valve or the temperature of a chemical vat (graphical)
- cells in a spreadsheet (tabular)
- the terms and conditions of a contract (textual)
- product installation instructions (video or audio)

*Controllers* are objects that control how user actions are interpreted. They are the input mechanism for the views and models.

For example, the interpretation of a double-click on a temperature gauge would be handled by the controller notifying the model in a way it agrees to respond to.

MVCs come in a triad, with communication between the components occurring as follows:
Whenever the state of a model changed, the view needs to display itself differently.

For instance, consider a View that represents a car’s engine temperature.

If the engine temperature goes up (say, because a fan belt breaks) the gauge showing the temperature will need to redisplay its new value in response to that change.

In Rails, when a project is created, it is given folders for model, view, and controller classes.

A Rails application accepts an incoming request from a Web page, then gives it to a router. The router parses the URL and eventually directs the request to a particular controller.

For example, the URL might be something like http://expertiza.ncsu.edu/users/show/108.

This means to show the various fields in the entry for User 108 (name, e-mail address, role). In this case,

- the controller is users,
- the action is show, and
- the ID of the user is 108.

Or, we might have something like http://www.etailer.com/store/add_to_cart/353.

What do you think this would represent?
The Rails 4.0 Cookbook Application

After we start the server, we see this series of messages:

When we visit the application in our browser, we see this initial screen:
Let’s create a category:

And a couple of beverages …
Now let's go and look at what's in the database:

- Go to View → Tools Window → Database.
- clicking on Rails:cookbook3 development will show the schemas of all the tables.
- Right-clicking on a table and clicking Table Editor will show the records in that table.
Notice the created_at and updated_at fields. These are automatically updated with the timestamp of the time that a row was created or updated.

Also notice the category_id. **What do you think this is?**

### The controllers

Let’s take a look at the code for categories_controller.rb.

```ruby
class CategoriesController < ApplicationController
  # GET /categories
  # GET /categories.json

  def index
    @categories = Category.all
    respond_to do |format|
      format.html # index.html.erb
      format.json { render json: @categories }
    end
  end
end
```

The statement inside the method does a database lookup of all categories, and assigns the resulting array to the `@categories` instance variable.

Without web-service support, that would be the whole method. The code that follows determines whether to respond with HTML (as when we are interacting with a user) or JSON (if we are returning an object). What that says is, "if the client wants HTML, just send back the categories in HTML, but if the client wants JSON, return the list of categories in JSON format." The response format is determined by HTTP Accept header sent by the client.

Immediately after executing a method, e.g., `index`, the controller will render a template of the same name. This template is in the corresponding directory in the `views` folder. In this case, the template is
views/categories/index.html.erb. In a few minutes, we will talk about .erb files.

There is very little difference between the index method and the show method.

```ruby
# GET /categories/1
# GET /categories/1.json
def show
  @category = Category.find(params[:id])

  respond_to do |format|
    format.html # show.html.erb
    format.json { render json: @category }
  end
end
```

The show method looks for a category with a particular key. Its output is very basic (above). The formatting is different (the text “Listing categories” doesn’t appear, etc.) because
the corresponding views are different (as we will see).

Look again at the difference between the assignment statements involving categories.

- **index** has `@categories = Category.all`
- **show** has `@category = Category.find(params[:id])`

Both of these methods do retrievals from the **categories** table in the db.

- The **all** method retrieves all records from the **categories** table and assigns the collection to a variable called `@categories`.
- The **find(...)** method retrieves a particular record, the record with the specified **id**.

This is the first use we have seen of **Active Record**.

**Active Record**

Our Ruby on Rails programs deal with **objects**, but they are mapped into relational databases.

There’s a mismatch here—how are the database tables translated into objects, and how are objects created in the program saved to the db?

Ruby on Rails’ solution is Active Record. In Active Record,

- Database tables correspond to Rails (model) classes.
- Database records (rows) correspond to Rails objects.

We can perform operations on tables by invoking class methods, as is done in both the RecipesController and the CategoriesController:

```ruby
@recipes = Recipe.all
@categories = Category.all
```

Let’s take a closer look at the **find** in the **show** method:
Category.find(params[:id])

This illustrates two common features of Active Record calls.

- **params** is an object (a hash) that holds all of the parameters passed in a browser request.
- **params[:id]** holds the id, or primary key, of the object.

When you click a link for a specific category, the id of that category is passed in the **params** object, so that **show** can display that category.

**Method pairs**

Next we have the **new** and **create** methods. Sounds like they might do the same thing …

What’s the difference between the two? Well, which one is called first?

The _____ method prepares the form for display; the _______ method processes the data that was entered and attempts to save it to the db.

```ruby
# GET /categories/new
def new
  @category = Category.new
  respond_to do |format|
    format.html # new.html.erb
    format.json { render json: @category }
  end
end

# POST /categories
# POST /categories.json
def create
  @category = Category.new(category_params)
  respond_to do |format|
    if @category.save
      format.html { redirect_to @category, notice: 'Category was successfully created.' }
      format.json { render json: @category, status: :created, :location: @category }
    else
      format.html { render action: "new" }
      format.json { render json: @category.errors, :status: :unprocessable_entity }
    end
  end
end
```
The code that follows the call to `Category.new` determines whether to respond with HTML (as when we are interacting with a user) or JSON (if we are returning an object).

What that says is, "if the client wants HTML, just send back the categories in HTML, but if the client wants JSON, return the list of categories in JSON format." The response format is determined by HTTP Accept header sent by the client.

There is a similar distinction between `edit` and `update`. Edit retrieves a table entry and displays it in a window. When the changes are submitted, `update` is invoked.

```ruby
# GET /categories/1/edit
def edit
  @category = Category.find(params[:id])
end

# PUT /categories/1
# PUT /categories/1.xml
def update
  respond_to do |format|
    if @category.update_attributes(category_params)
      format.html { redirect_to @category, notice: 'Category was successfully updated.' }
      format.json { head :no_content }
    else
      format.html { render action: 'edit' }
      format.xml { render json: @category.errors, status: :unprocessable_entity }
    end
  end
end
end

# DELETE /categories/1
# DELETE /categories/1.xml
def destroy
  @category.destroy

  respond_to do |format|
    format.html { redirect_to categories_url }
    format.xml { head :no_content }
  end
end
```
The categories_controller is extremely similar to the recipes_controller. Let’s take a look …

These controllers were generated by the Rails scaffold mechanism. We can use the scaffold to create a recipes table, with title, description, and instructions fields.

We can then proceed to exercise the application as we did before. But now we can change its functionality too.

Let’s make a trivial change: Change “Show all recipes” and “Show all categories” to “List all recipes and “List all categories”.

Where shall we make this change?

Well, if we click on “Show all categories”, we get this screen.

What do you notice about the bottom line?
And if we click on “Create new recipe”, we get this screen:

![Image of a webpage and directory listing](image)

Again, the bottom line is the same. Let’s take a look at the directory listing. Which file do you think contains that code?

This is an .html.erb file, the first time we’ve seen this type. What do you think it stands for?

Let’s look at the code in this file.

```html
<!DOCTYPE html>
<html>
<head>
  <title>Online Cookbook</title>
</head>
</html>
```
This raises several questions.

- What gets invoked when we click on the “Show all recipes” link?

- How do we make it say, “List all recipes”?

- What is the <%= yield %> for?

- What’s that strange notation surrounding yield?

- What do “recipes_url”, “categories_url”, etc., mean?

Older versions of Rails had separate routes for index, show, new, edit, create, update, and destroy operations. In Rails 3 and 4, you just declare

resources :categories
and this generates routes for each of the methods for the categories_controller.
For a description, see this tutorial.

And if we poke around and look at all the views, we will find that all have the same “navigation bar” at the bottom.

Now, to make sure you are following, here are some review questions.

1. What URL do we type in to find the homepage of our cookbook application?
2. When we click on “Show all categories”, what URL will be taken to?
3. What are the filenames that contain views associated with recipes?
4. What is Embedded Ruby, and how have we seen it used?

The models
There are only two files in the model, one each for the tables in the application.

Let’s take a look at them.

category.rb

class Category < ActiveRecord::Base
  has_many :recipes
  # validates :name, :presence => true
end

recipe.rb

class Recipe < ActiveRecord::Base
  belongs_to :category
  # validates :title, :presence => true
  # validates :description, :presence => true
  # validates :instructions, :presence => true
  # validates :category, :presence => true
end
The `validates` of Rails 3 is replaced with the `strong-parameter` mechanism in Rails 4.

Relationships in models
A relationship may be

- one-to-one (e.g., a course has a syllabus and a syllabus belongs to one course).
  
  o `class Syllabus < ActiveRecord::Base
      belongs_to :course
      # ...
     end`

  o `class Course < ActiveRecord::Base
      has_one :syllabus
      # ...
     end`

- one-to-many (e.g., a course has many assignments)
  
  o `class Assignment < ActiveRecord::Base
      belongs_to :course
      # ...
     end`

  o `class Course < ActiveRecord::Base
      has_many :assignments
      # ...
     end`

- many-to-many (e.g., a course has many students; students have many courses)
  
  o `class Student < ActiveRecord::Base
      has_and_belongs_to_many :courses`
You should be able to answer the following questions about these files.

1. What kind of relationship is there between recipes and categories?

2. Where is this relationship represented?

The views

Now, let’s take a look at the View code for the categories. We’ll look at it line by line, which may obscure the flow, but if you have trouble, just look in your Cookbook folder for the uncommented code.

`edit.html.erb`

<h1>Editing category</h1>

<%= render 'form' %>

<%= link_to 'Show', @category %> |
<%= link_to 'Back', categories_path %>

The form refers to a partial named `_form.html.erb`. Answer these questions about partials.

`_form.html.erb`

<%= form_for(@category) do |f| %>
  <%= if @category.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@category.errors.count, "error") prohibited this category from being saved:"%>
    </div>
  <%end%>
</f>
Compare this with _form.html.erb for recipes:

```erb
<% form_for(@recipe) do |f| %>
  <% if @recipe.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@recipe.errors.count, "error") %> prohibited this recipe from being saved:</h2>
      <ul>
        <% @recipe.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :description %><br>
    <%= f.text_field :description %>
  </div>
  <div class="field">
    <%= f.label :category %><br>
    <%= select("recipe", "category_id", Category.all.collect{ |c| [c.name, c.id] }) %>
  </div>

  <%= f.submit %>
</%>
```

Explain the differences.

**show.html.erb**

This file has very basic functionality.

<p id="notice"><%= notice %></p>

<p></p>

<b>Name:</b> <%= @category.name %>

<p><%= link_to 'Edit', edit_category_path(@category) %> | <%= link_to 'Back', categories_path %></p>

Compare with **show.html.erb** for recipes.

<p id="notice"><%= notice %></p>

<p></p>

<b>Title:</b> <%= @recipe.title %>

<p></p>

<b>Description:</b> <%= @recipe.description %>

<p></p>

<b>Instructions:</b> <%= @recipe.instructions %>

<p><%= link_to 'Edit', edit_recipe_path(@recipe) %> | <%= link_to 'Back', recipes_path %></p>
new.html.erb

The only remaining view is new.html.erb. It doesn’t illustrate much that we haven’t seen before, so I’ll ask you the questions (below).

<h1>New category</h1>
<% render 'form' %>
<% link_to 'Back', categories_path %>

1. **Which controller** is invoked when the form is submitted? Where is the code for this controller?

2. What says to print out a blank for the name of the category?