COM 1001

Introduction to Software Engineering

Professor Phil McMinn

Interacting with a Database Using Ruby

Object-Relational Mapping

Interacting with a Database from a Program

As we learnt last time, we can “talk” to a relational databases using a language called **SQL**.

One way to interact with a database from a program, is to construct strings of SQL, pass them to the database, and collect the results.

A better way of doing it is to use a pre-existing library of program code, better known as an **Application Programming Interface (API)**, to construct those SQL strings for us. Then we can just use the methods of the API as part of our regular program code.

Sequel 

Sequel is one such API for Ruby.

It provides a **DSL** for constructing SQL queries and database schemas.

Sequel works for many DBMSs, of which SQLite is one.

It will let us use the same code and we can switch the underlying DBMS. Any differences in SQL will be handled for us.

Plus, it makes interacting with a database from Ruby really easy.

```ruby
require "logger"
require "sequel"

DB = Sequel.sqlite("football_players.sqlite3",
                   logger: Logger.new("db.log"))

puts "Please enter a club name:"
supplied_club = gets.chomp

dataset = DB[:players].where(club: supplied_club)
num_rows = dataset.count

if num_rows.zero?
  puts "Sorry there are no players for that club."
else
  dataset.each do |record|
    puts "* #{record[:first_name]} #{record[:surname]}"
  end
end
```

orm/sequel_select.rb

```ruby
require "logger"
require "sequel"

DB = Sequel.sqlite("football_players.sqlite3",
                   logger: Logger.new("db.log"))

puts "Please enter a club name:"
supplied_club = gets.chomp

dataset = DB[:players].where(club: supplied_club)
num_rows = dataset.count

if num_rows.zero?
  puts "Sorry there are no players for that club."
else
  dataset.each do |record|
    puts "* #{record[:first_name]} #{record[:surname]}"
  end
end
```

orm/sequel_select.rb

This is how we connect to the SQLite database. We need to supply the SQLite database filename.

This will log the SQL statements sent to the database, which helps with debugging. We need the `logger` gem to do this.

This is equivalent to the SQL statement `SELECT * FROM players WHERE club = #{supplied_club}`

The table name is provided as the symbol `:players`

The column name (`club`) is also provided as a symbol (`club:`) to the `where` method, but it looks slightly different – the colon is after the symbol rather than before. It's actually just short hand for writing a key-value pair in the form `:club => supplied_club`
More on key-value pairs later.

This is equivalent to the SQL statement `SELECT COUNT(*) FROM players WHERE club = #{supplied_club}`

```ruby
require "logger"
require "sequel"

DB = Sequel.sqlite("football_players.sqlite3",
                   logger: Logger.new("db.log"))

puts "Please enter a club name:"
supplied_club = gets.chomp

dataset = DB[:players].where(club: supplied_club)
num_rows = dataset.count

if num_rows.zero?
  puts "Sorry there are no players for that club."
else
  dataset.each do |record|
    puts "* #{record[:first_name]} #{record[:surname]}"
  end
end
```

The query returns an instance of an `DataSet` object, which we can call the each method on to get each record for the query.

We get hold of the respective fields like this – using the column names, but provided as symbols

```
codio@north-mister:~/workspace/com1001-code/databases$ ruby sequel_select.rb
Please enter a club name:
Everton
* Dominic Calvert-Lewin
* Hayley Raso
* Michael Keane
codio@north-mister:~/workspace/com1001-code/databases$
```

You can see the SQL statements Sequel constructed and sent to the database by running the program and loading the log file *db.log* into a text editor

```
INFO -- : (0.000167s) SELECT count(*) AS 'count' FROM `players` WHERE (`club` = 'Everton') LIMIT 1
INFO -- : (0.000140s) SELECT * FROM `players` WHERE (`club` = 'Everton')
```

Records as Hashes

```ruby
dataset.each do |record|
  puts record
end
```

If we output the whole record, we can see that it's actually a Ruby **hash**

A **hash** is a collection of **key-value pairs**. We provide it with a key to look up some value. The key can be a string, or a symbol, or any Ruby object.

Sequel records are hashes where the keys are symbols where the symbol names are the same as the field names (columns) of the record.

```
codio@north-mister:~/workspace/com1001-code/databases$ ruby sequel_select.rb
Please enter a club name:
Everton
{:id=>1, :first_name=>"Dominic", :surname=>"Calvert-Lewin", :gender=>"M", :date_of_birth=>"1997-03-
16", :country=>"England", :position=>"Forward", :club=>"Everton"}
{:id=>8, :first_name=>"Hayley", :surname=>"Raso", :gender=>"F", :date_of_birth=>"1994-09-05", :coun
try=>"Australia", :position=>"Midfielder", :club=>"Everton"}
{:id=>11, :first_name=>"Michael", :surname=>"Keane", :gender=>"M", :date_of_birth=>"1993-01-11", :c
ountry=>"England", :position=>"Defender", :club=>"Everton"}
codio@north-mister:~/workspace/com1001-code/databases$
```

More on Hashes

Hashes will crop up again in this module – they are very common in Ruby, especially the variant using symbols as keys.

Hashes are a *bit* like arrays, except:

- They do not have to use integers as keys
- The keys to not have to be sequential or in any order

Other languages have similar constructs, e.g.:

- **Dictionaries** in Python
- **Maps** in Java

```ruby
# This is the preferred way to initialise
# an empty hash:
hash1 = {}

# You can also initialise with some key-value pairs.
# The syntax is:
#   key => value
hash2 = { "key1" => "value1", "key2" => "value2" }

# Here, we're using symbols.
# The syntax:
#   key: value
# is equivalent to writing:
#   :key => value
# But the former is the preferred style
hash3 = { key1: "value1", key2: "value2" }

# Adding or updating a key-value pair is done as follows:
hash1[1000] = 99
hash2["key3"] = "value3"
# or
hash3[:key3] = "value3"
hash3[:key3] = "value4"

# How we get a value for a key:
puts hash2["key1"]

# This is how we return a default value in the case
# that the key is not in the hash
puts hash1.fetch("non-existent-key", "hi!")

# This is how we remove a key
hash3.delete(:key1)
```

ruby/hashes.rb

Object-Relational Mapping

You should now have encountered **objects** and **classes** in COM1003.

Now a big reveal: every variable in Ruby is an object! Including types considered "primitive" in Java (i.e. "not worthy" of being an object), like integers, doubles etc.

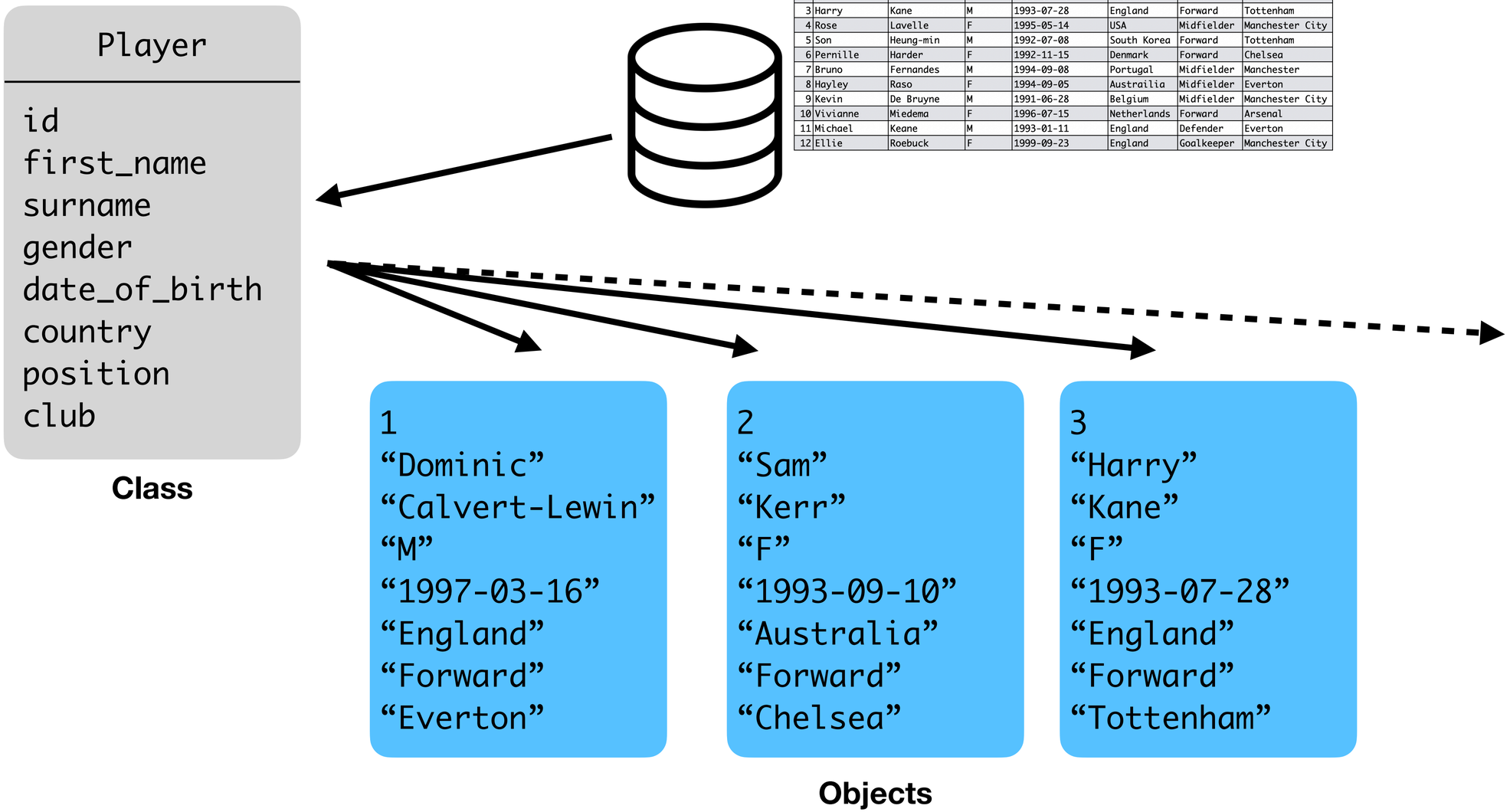
Classes define what data goes in an object and what methods can be executed on that data. Objects are "instances" of classes. That means they have their own individual data separate from other objects of the same class.

In Sequel, we can automatically create classes that correspond to a row of a database table, that can be instantiated to produce records as objects.

We can use data from the database as if it were an object like any other object in our program. The classes of those objects are called **Models**.

This is a step up from using plain hashes for records, and is known as **Object-Relational Mapping (ORM)**.

Object-Relational Mapping



id	first_name	surname	gender	date_of_birth	country	position	club
1	Dominic	Calvert-Lewin	M	1997-03-16	England	Forward	Everton
2	Sam	Kerr	F	1993-09-10	Australia	Forward	Chelsea
3	Harry	Kane	M	1993-07-28	England	Forward	Tottenham
4	Rose	Lavelle	F	1995-05-14	USA	Midfielder	Manchester City
5	Son	Heung-min	M	1992-07-08	South Korea	Forward	Tottenham
6	Pernille	Harder	F	1992-11-15	Denmark	Forward	Chelsea
7	Bruno	Fernandes	M	1994-09-08	Portugal	Midfielder	Manchester
8	Hayley	Raso	F	1994-09-05	Austrailia	Midfielder	Everton
9	Kevin	De Bruyne	M	1991-06-28	Belgium	Midfielder	Manchester City
10	Vivianne	Miedema	F	1996-07-15	Netherlands	Forward	Arsenal
11	Michael	Keane	M	1993-01-11	England	Defender	Everton
12	Ellie	Roebuck	F	1999-09-23	England	Goalkeeper	Manchester City

Our First Model

```ruby
require "logger"
require "sequel"

DB = Sequel.sqlite("../databases/football_players.sqlite3",
                   logger: Logger.new("db.log"))

class Player < Sequel::Model
end

players = Player.all
players.each do |player|
  puts "#{player.first_name} #{player.surname}"
end
```

orm/first_model.rb

Nothing new so far…

This is how we declare a class in Ruby. The "`<`" symbols means "extends". Our `Player` class extends the class `Sequel::Model`. There are no methods yet, but thanks to Sequel, we've just inherited a lot of functionality.

Sequel figures out from the class name that this class needs to map to the players table in the database.

The `Player` class has class-level methods (equivalent to `static` methods in Java) that let us get records from the players table. The method "`all`" returns an array of objects where each object corresponds to a records in the players table.

We can now access the record fields through method calls on the object.

Why Is This Useful?

```ruby
class Player < Sequel::Model
  # Get a string of the player's name in one method
  def name
    "#{first_name} #{surname}"
  end

  # Get the player's age, based on their date_of_birth
  def age(at_date = Date.today)
    dob = Date.strptime(date_of_birth, "%Y-%m-%d")
    TimeDifference.between(dob, at_date).in_years.floor
  end
end
```

orm/player.rb

In true object-oriented style, we can **package up functionality that belongs with our data**.

This makes our codebase more **cohesive**, reducing **coupling** between files.

In a real application, we can add business logic to our models to process the data as it comes in and out of the database.

Models are Easy to Unit Test!

```ruby
RSpec.describe Player do
  describe "#name" do
    it "returns the player's full name" do
      player = described_class.new(first_name: "A", surname: "B")
      expect(player.name).to eq("A B")
    end
  end

  describe "#age" do
    it "returns the age of the player" do
      player = described_class.new(date_of_birth: "2000-1-1")
      expect(player.age("2020-1-1")).to eq(20)
    end
  end
end
```

orm/player_spec.rb

Note that we put the class name between the "`RSpec.describe`" and the "`do`"

The method name as a string, preceded by a hash, goes in between the "`describe`" and the "`do`" for each describe block.

(This is the same as with the test for the `string_comparison` method from the testing lecture.)

RSpec prefers to refer to the class under test as "`described_class`" rather than its real name (which is `Player` in this instance)

Getting Many Specific Records

The form of the code here is not too dissimilar to the example using `DataSets` and hashes given earlier in the lecture.

We could supply several field-value pairs here, if we wanted, in a comma-separated list

```ruby
require "logger"
require "sequel"

DB = Sequel.sqlite("../databases/football_players.sqlite3",
                   logger: Logger.new("db.log"))
require_relative "player"

puts "Please enter a player's club:"
supplied_club = gets.chomp

players = Player.where(club: supplied_club)
num_players = players.count

if num_players.zero?
  puts "Sorry there are no players for that club."
else
  players.each do |player|
    puts "* #{player.name}, Age: #{player.age}"
  end
end
```

orm/obtain_specific_players.rb

Getting One Specific Record

```ruby
require "logger"
require "sequel"

DB = Sequel.sqlite("../databases/football_players.sqlite3",
                   logger: Logger.new("db.log"))
require_relative "player"

puts "Please enter a player's ID:"
supplied_id = gets.chomp

player = Player.first(id: supplied_id)
if player.nil?
  puts "No player exists with that ID"
else
  puts "#{player.name}, Age: #{player.age}"
end
```

orm/obtain_first_player.rb

This is the key part of this example. The `first` method is called in the same way as the `where` method in the previous example, except of course it returns the first record the database retrieves rather than all of them.

This is useful when there **should only be one record**, for example when we're **looking up a record by its primary key**, as we are doing here.

Create, Update, Delete

```ruby
# Create a new player instance
player = Player.new
player.first_name = "Marcus"
player.surname = "Rashford"
player.club = "Manchester United"

# Save to the database
player.save_changes

# Update his club and save again
player.club = "Manchester City"
player.save_changes

# Now delete
player.delete
```

orm/create_update_delete.rb

Creates a new player instance in memory only (i.e., not in the database)

This triggers Sequel to generate an SQL `INSERT` statement and send it to the database

Since the record already exists in the database, Sequel now generates an SQL `UPDATE` statement to update the corresponding record in the database

This triggers Sequel to generate an SQL `DELETE` statement to remove the corresponding record.

Create, Update, Delete

```ruby
# Create a new player instance
player = Player.new
player.first_name = "Marcus"
player.surname = "Rashford"
player.club = "Manchester United"

# Save to the database
player.save_changes

# Update his club and save again
player.club = "Manchester City"
player.save_changes

# Now delete
player.delete
```

We can see the effect of these SQL statements in the log.

(Run the program and load *db.log* into a text editor, and scan the last few lines)

```
INFO -- : (0.000232s) INSERT INTO `players` (`first_name`, `surname`,
`club`) VALUES ('Marcus', 'Rashford', 'Manchester United')
...
INFO -- : (0.000163s) UPDATE `players` SET `first_name` = 'Marcus',
`surname` = 'Rashford', ... `club` = 'Manchester City' WHERE (`id` = 13)
...
INFO -- : (0.000551s) DELETE FROM `players` WHERE `id` = 13
```

("…" indicate parts of the log removed for brevity)

Documentation

If there's something you need to do that I don't have an example to cover, it's worth checking out Sequel's extensive documentation.

See https://sequel.jeremyevans.net/documentation.html

The README.md of the GitHub page also contains some useful examples: https://github.com/jeremyevans/sequel