Week 5 - Topic 1: Complex Data Types (Records, Enumerations)

Complex Data Types

This lecture we start by looking at **programmer** created Data Types:

1. Records/Classes
2. Enumerations

Versus Primitive Data Types

We have been using ‘**primitive**’ data types which are **basic** types that come with the language Ruby (and most other languages).

These primitive/basic data types are the essential ones you need to create more complex data types.

What are the primitive data types we have been using so far?

Programmer Created Data Types

- Also known as ‘**custom**’ data types.
- These are created to represent the entities you are modelling in your program.
- Each program is a model, or abstraction, of some aspects of the real world.
- As an abstraction, some details are left out, others are specifically included for the purpose of the model.

Complex Data Type Example

Consider a student.

- What attributes does a student have?

Complex Data Type Example II

- What student attributes would be included in a student administration system for a university?

Example Student Record (Data Dictionary)

Date complex data type

- Some complex data types are not ‘custom’ in the sense that the application programmer needs to create them.
- Some complex data types are in libraries provided with or for the language.
  Eg: Date (which exists already for Ruby)
- If you were to write this yourself you might start with a record that looks as follows:
- This is an improvement on one long integer.

Using Classes and structs to Represent Complex Data Types

In Ruby we use **classes** to represent records such as the Student and Date records above.

Run RUBY

```ruby
class Date
  attr_accessor :day, :month, :year
end
```

In C we use structs which would look as follows:

Run C

```c
typedef struct Date {
  int day;
  int month;
  int year;
} Date;

int main(){
  Date d;
  d.day = 31;
  d.month = 12;
  d.year = 2019;
}
```

Each of these has three attributes (day, month, year).

Ruby also offers a form of struct:

Run RUBY

```ruby
Student = Struct.new(:name, :id, :email, :course)

s = Student.new("Sam", "0320", "sam@uni.com.au")

puts "Name: " + s.name
puts "Id: " + s.id
puts "Email: " + s.email
```

Type/Token Distinction

- A class (or record or struct) represents a **type.**
- To allocate memory we need to create a **token**.
- We need to create an instance of the type.
  Eg for Ruby:

```
date = Date.new()
```

- And for C:

```
Date date;
```

Using complex data types

Once we create an instance (token) of a complex data type, we need to initialize its field/attribute values (which we should do with all variables):

For our Date class (not the standard Ruby one):

```ruby
class Date
  attr_accessor :day, :month, :year
end

date = Date.new()
date.day = 31
date.month = 12
date.year = 2019

puts "Date is #{date.day}-#{date.month}-#{date.year}"


```

And for C:

```c
#include <stdio.h>

typedef struct Date {
  int day;
  int month;
  int year;
} Date;

int main(){
  Date d;
  d.day = 31;
  d.month = 12;
  d.year = 2019;

```

Initialising Complex Data Types

In Ruby we can create a block of code to initialise the fields/attributes Eg for Ruby:

```ruby
class Date
  attr_accessor :day, :month, :year

  def initialize(day, month, year)
    @day = day
    @month = month
    @year = year
  end
end

date = Date.new(31, 12, 2019)


```

Nested Complex Data Types

A complex data type can have fields/members/attributes that are also complex data types:

Run RUBY

```ruby
class Date
  attr_accessor :day, :month, :year

  def initialize(day, month, year)
    @day = day
    @month = month
    @year = year
  end
end

class Booking
  attr_accessor :room, :date, :time

  def initialize(room, date, time)
```

Enumerations

- Enumerations are a custom data type that holds constant values.
- Enumerations simply assign meaningful names to a set of integers.
- Eg: to Represent different Genres of music we could use 0, 1, 2, 3 etc. Or we could create an enumeration as follows:

▶ Run RUBY

```ruby
module Genre
  Pop, Classic, Jazz, Rock = *0..3
end

```

Pop = 0, Classic = 1, Jazz = 2 and Rock = 3.

In C, this would be:

▶ Run C

```c
enum genre_names {Pop, Classic, Jazz, Rock};

int main(){
}

```

Example Enumeration

An enumeration you have already seen is one for the Z order of objects you draw on the screen in Gosu.

This substitutes 0 for BACKGROUND, 1 for PLAYER and 2 for UI.

Run RUBY

```ruby
module ZOrder
  BACKGROUND, PLAYER, UI = *0..2
end
```

Summary

Terminology:

- Primitive data types
- Complex Data Types
- Custom Data Types
- Enumerations
- Records
- Classes

Types Quiz

Which of the following statements are true:

- ☐ In Ruby we can represent complex types using classes
- ☐ In Ruby we can represent complex types using structs
- ☐ In Ruby we can represent custom types using modules
- ☐ In Ruby complex types can be components of other complex types
- ☐ Once you define a type you can assign values to the fields

Track File Handling Task

Lets look at how to solve the track reading task.

Types Supplement