TP N°2: Colecciones en Ruby

La siguiente guía cubre los contenidos vistos en las clases teóricas **2** y **3**

Ejercicio 1

Consulte la documentación de la clase `Array` e indique en papel la salida de las siguientes expresiones. Luego comprobarlo con la consola.

```
my_array = [9, 5, 1, 2, 3, 4, 0, -1]
p my_array.size
p my_array.first
p my_array.last
p my_array[1]
p my_array[9]
p my_array[2..4]
p my_array[2...4]
p my_array[2, 4]
p my_array[-3]
p my_array.first(4)
p my_array.drop(4)
p my_array.prepend(10, 11)
p my_array.append(20, 21)
p my_array.map { |e| e * 3 }
p my_array.map{ |e| e * 3 }.reduce { |sum, e| sum + e }
```

1. 8
2. 9
3. -1
4. 5
5. error
6. 1,2,3
7. 1,2
8. 1,3
9. 4
10. 9,5,1,2
11. 12.10, 11 +array
13. array + 20, 21
14.todo elemento *3
15.todo elemento * 3 sumado

1.
2.
3.
4.
5. nul
6.
7.
8. 1,2,3,4
9.
10.
11. 3,4,0,-1
12.
13.nuevo array + 20,21
14.
15.

Ejercicio 2

Implementar la clase `Stack` para poder operar con una pila de elementos. Con el siguiente programa de prueba, se debe obtener la salida indicada en los comentarios. Realice el diagrama de clases correspondiente.

```
stack = Stack.new
stack.push(2)
stack.push(3)
puts stack.peek # 3
puts stack.pop # 3
puts stack.empty? # false
puts stack.pop # 2
puts stack.empty? # true
puts stack.pop # Stack is empty (RuntimeError)
```

Ejercicio 3

Implementar la clase `AccessStack`. Debe contar con dos métodos para conocer el número de veces que se invocó a los métodos `push` y `pop`. Realizar el diagrama de clases correspondiente y un programa de prueba.

Ejercicio 4

Consulte la documentación de la clase `Object` e indique en papel la salida de las siguientes expresiones. Luego comprobarlo con la consola.

```
a = String.new('Hola Mundo')
b = String.new('Hola Mundo')
puts a == b
puts a === b
puts a.eql? b
puts a.equal? b
puts a <=> b
```

1. true
2.true
3.true
4.false
5.0

1.
2.
3.
4.
5.

Indique el propósito de cada uno de los métodos. ¿Cuáles recomienda la documentación sobreescribir en las subclases?

Ejercicio 5

Indicar en papel la salida del siguiente programa de prueba que utiliza la clase `Point` implementada en el ***TP Nº1***. Luego verificarlo en el entorno Ruby.

```
require_relative '../tp1/ej4'

my_point = Point.new(1, 2)
puts my_point == my_point
puts my_point == Point.new(1, 2)
puts my_point != Point.new(3, 4)
puts my_point == 'Hola Mundo'
```

true
false
true
false

Agregar a la clase `Point` todo lo necesario para que la clase pueda responder correctamente a la **equivalencia**.

Ejercicio 6

Agregar a la familia de clases de figuras geométricas del ***TP Nº1*** todo lo necesario para que las clases puedan responder correctamente a la **equivalencia**. Actualice el diagrama de clases correspondiente.

Ejercicio 7

Consultar la documentación de la clase `Set` (colección sin repetidos). Indicar en papel la salida del siguiente programa de prueba. Luego verificarlo en el entorno Ruby.

```
require 'set'

point_set = Set.new
point_set.add(Point.new(1,2))
point_set.add(Point.new(3,4))
point_set.add(Point.new(1,2))
puts point_set
```

Agregar a la clase `Point` del ***Ejercicio 5*** todo lo necesario para que la clase pueda funcionar correctamente en la colección `Set` de Ruby (de forma que no existan números complejos equivalentes en la colección).

~~*Ejercicio 8*~~

Agregar a la familia de clases de figuras geométricas del ***Ejercicio 6*** todo lo necesario para que la clase pueda funcionar correctamente en la colección `Set` de Ruby.

A modo de ejemplo, para el siguiente programa de prueba:

```
require_relative 'ej6'
require 'set'

my_figure_set = Set.new
my_figure_set.add(Circle.new(Point.new(10, 20), 15))
my_figure_set.add(Circle.new(Point.new(10, 20), 15))
my_figure_set.add(Rectangle.new(Point.new(10, 20), Point.new(20, 10)))
puts my_figure_set
```

se espera obtener la siguiente salida:

```
#<Set: {Círculo [Centro: {10,20} , Radio: 15], Rectángulo [ {10,20} , {20,10} ]}>
```

~~*Ejercicio 9*~~

Consultar la documentación de la clase `SortedSet` (colección con orden sin repetidos). Indicar en papel la salida del siguiente programa de prueba. Luego verificarlo en el entorno Ruby.

```
require_relative 'ej8'

my_sorted_figure_set = SortedSet.new
my_sorted_figure_set.add(Rectangle.new(Point.new(10, 20), Point.new(20, 10)))
my_sorted_figure_set.add(Circle.new(Point.new(10, 20), 1))
my_sorted_figure_set.add(Circle.new(Point.new(10, 20), 3))
puts my_sorted_figure_set
```

Agregar a la familia de clases de figuras geométricas del ***Ejercicio 8*** todo lo necesario para que la clase pueda funcionar correctamente en la colección `SortedSet` de Ruby.

A modo de ejemplo, para el programa de prueba anterior, se espera obtener la siguiente salida:

```
#<SortedSet: {Círculo [Centro: {10,20} , Radio: 1], Círculo [Centro: {10,20} , Radio:
3], Rectángulo [ {10,20} , {20,10} ]}>
```

donde el orden natural de las figuras es ascendente por el área de la figura.

¿Cómo cambiaría el programa de prueba para tener las figuras en el orden inverso sin cambiar el orden natural de las mismas?

Ejercicio 10

Una bolsa o *bag* es un conjunto de elementos sin orden donde cada elemento puede aparecer una o más veces.

Implementar la clase `Bag` de forma que el siguiente ejemplo

```
require_relative 'ej7'

my_bag = Bag.new
my_bag.add(Point.new(0, 0))
my_bag.add(Point.new(1, 2))
my_bag.add(Point.new(3, 4))
my_bag.add(Point.new(1, 2))
puts my_bag
puts my_bag.size
puts my_bag.count(Point.new(1, 2))
puts my_bag.delete(Point.new(1, 2))
puts my_bag
puts my_bag.delete(Point.new(1, 2))
puts my_bag
```

produzca la siguiente salida

```
{{0,0}=>1, {1,2}=>2, {3,4}=>1}
3
2
1
{{0,0}=>1, {1,2}=>1, {3,4}=>1}
0
{{0,0}=>1, {3,4}=>1}
```

donde

- el método `size` retorna la cantidad de elementos distintos presentes en la bolsa
- el método `count(element)` retorna cuántas veces aparece el elemento `element` en la bolsa
- el método `delete(element)` remueve una aparición del elemento `element`, y retorna la cantidad de veces que aparece el elemento en el conjunto luego de haberlo removido. Si el elemento no estaba retorna cero.