* 1. ***TP N°8: Introducción a Ruby***

|  |
| --- |
| La siguiente guía cubre los contenidos vistos en la clase teórica: **9. Introducción a Ruby** |

|  |
| --- |
| Para verificar que se cuenta con Ruby instalado ejecutar desde la consola el comando  ruby --version  Para utilizar la consola de Ruby ejecutar desde la consola el comando irb. Para salir de la consola de Ruby invocar a exit. |

***Ejercicio 1***

Retomando el ***Ejercicio 1*** del ***TP Nº1*** se cuenta con el siguiente script de Ruby:

|  |
| --- |
| chainring = 52  cog = 11  ratio = chainring / cog.to\_f  puts ratio # => 4.7272727272727275  chainring = 30  cog = 27  ratio = chainring / cog.to\_f  puts ratio # => 1.1111111111111112 |

Se desea crear una clase Gear que ofrezca un comportamiento similar al del script. Diseñar el diagrama de clases correspondiente. Luego, en un archivo **gear.rb**, implementar la clase Gear y debajo de la definición escribir un programa de prueba para obtener los mismos valores que los del script pero utilizando la nueva clase Gear.

1. Luego de haber ejecutado el programa, separar la definición de la clase Gear y el programa de prueba en dos archivos fuentes distintos gear.rb y gear\_tester.rb. Ejecutar gear\_tester.rb. ¿Cómo se indica en el archivo del programa de prueba donde se encuentra la definición de la clase Gear?
2. Los ciclistas utilizan el término *gear inches* para comparar bicicletas de distinto *gearing* y tamaño de rueda. La fórmula utilizada es la siguiente:

*gear inches = gear ratio \* wheel diameter*

donde

*wheel diameter = rim diameter + twice tire diameter*

Actualizar el diagrama de clases y modificar la clase Gear para que funcione con el siguiente programa de prueba:

|  |
| --- |
| require\_relative 'gear'  rim = 26  tire = 1.5  puts Gear.new(52, 11, rim, tire).gear\_inches # => 137.0909090909091  rim = 24  tire = 1.25  puts Gear.new(52, 11, rim, tire).gear\_inches # => 125.27272727272728 |

1. Con el siguiente programa de prueba y la implementación anterior se obtiene el siguiente error:

|  |
| --- |
| require\_relative 'gear'  puts Gear.new(52, 11).ratio # Esto funcionaba  ...  wrong number of arguments (given 2, expected 4) |

Como todos los parámetros son obligatorios, no se puede instanciar a la clase Gear sin especificar todos sus parámetros. Se busca poder seguir utilizando la clase Gear como en el apartado B, con sólo los primeros dos parámetros y así poder invocar el método ratio.

Para ello, crear una nueva clase Wheel que encapsule a las variables rim y tire. ¿Qué métodos deberían estar en esta clase? ¿Cómo cambia entonces la clase Gear? Realice una composición para que la clase Gear cuente con una instancia de la clase Wheel. Actualizar el diagrama de clases. Realizar la implementación de forma de poder ejecutar el siguiente programa de prueba:

|  |
| --- |
| require\_relative 'gear'  require\_relative 'wheel'  puts Gear.new(52, 11, Wheel.new(26, 1.5)).gear\_inches # => 137.0909090909091  puts Gear.new(52, 11).ratio # => 4.7272727272727275 |

1. Realizar los cambios necesarios para que la clase Gear sea responsable de instanciar a la clase Wheel. Actualizar el diagrama de clases. De esta forma no se invoca al método constructor de la clase Wheel desde el programa de prueba:

|  |
| --- |
| require\_relative 'gear'  puts Gear.new(52, 11, 26, 1.5).gear\_inches # => 137.0909090909091  puts Gear.new(52, 11).ratio # => 4.7272727272727275 |

* + 1. ***Ejercicio 2***

Retomando el ***Ejercicio 2*** del ***TP Nº1***, implementar la clase ComplexNumber que modela un número complejo (parte real y parte imaginaria).

La clase debe permitir que dos instancias de números complejos puedan sumarse retornado una nueva instancia con el resultado de la operación.

Diseñar el diagrama de clases correspondiente e implementar todo lo necesario para que con el siguiente programa de prueba se obtenga la salida indicada en el comentario:

|  |
| --- |
| first\_complex = ComplexNumber.new(2, -1)  second\_complex = ComplexNumber.new(3, 0)  res\_complex = first\_complex + second\_complex  puts res\_complex.real.to\_s + ' + ' + res\_complex.imaginary.to\_s + 'i'  # Imprime 5 + -1i |

Luego de haber implementado la clase, respecto a la última línea del ejemplo:

1. ¿Qué se obtiene si se quitan las invocaciones a to\_s de real y imaginary?
2. ¿Dónde corresponde que esté implementado la impresión de un número complejo?
3. ¿Cómo se puede simplificar la última línea del ejemplo de invocación?

***Ejercicio 3***

Retomando el ***Ejercicio 3*** del ***TP Nº1***, implementar la clase Point que modela un punto en el plano 2D y ofrece un método que calcula la distancia entre dos puntos. El programa de prueba es el siguiente:

|  |
| --- |
| my\_point = Point.new(1.5, 2.5)  puts my\_point # Imprime {1.5, 2.5}  puts my\_point.distance(Point.new(1.5, 3.0)) # Imprime 0.5 |

1. ***Ejercicio 4***

Implementar la familia de clases de figuras geométricas del ***Ejercicio 8*** del ***TP Nº1*** de tal forma que para el siguiente programa de prueba:

|  |
| --- |
| require\_relative 'rectangle'  require\_relative 'triangle'  require\_relative 'ellipse'  require\_relative 'circle'  start\_point = Point.new(0, 0)  my\_rectangle = Rectangle.new(start\_point, Point.new(2, 4))  puts my\_rectangle  puts my\_rectangle.perimeter  puts my\_rectangle.area  my\_triangle = Triangle.new(start\_point, Point.new(1, 0), Point.new(0, 1))  puts my\_triangle  puts my\_triangle.perimeter  puts my\_triangle.area  my\_ellipse = Ellipse.new(start\_point, 2, 1)  puts my\_ellipse  puts my\_ellipse.perimeter  puts my\_ellipse.area  my\_circle = Circle.new(start\_point, 3)  puts my\_circle  puts my\_circle.perimeter  puts my\_circle.area |

se obtenga la siguiente salida

|  |
| --- |
| Rectángulo [ {0, 0} , {2, 4} ]  12  8  Triángulo [ {0, 0} , {1, 0} , {0, 1} ]  3.414213562373095  0.49999999999999983  Elipse [Centro: {0, 0}, DMayor: 2, DMenor: 1]  4.71238898038469  1.5707963267948966  Círculo [Centro: {0, 0} , Radio: 3]  18.84955592153876  28.274333882308138 |

1. ***Ejercicio 5***

Implementar la familia de clases de cuentas bancarias del ***Ejercicio 10*** del ***TP Nº1*** de tal forma que para el siguiente programa de prueba:

|  |
| --- |
| require\_relative 'checking\_account'  require\_relative 'savings\_account'  my\_checking\_account = CheckingAccount.new(1001, 50)  my\_checking\_account.deposit(100.0)  puts my\_checking\_account  my\_checking\_account.extract(150.0)  puts my\_checking\_account  my\_savings\_account = SavingsAccount.new(1002)  my\_savings\_account.deposit(100.0)  puts my\_savings\_account  my\_savings\_account.extract(150.0) |

se obtenga la siguiente salida

|  |
| --- |
| Cuenta 1001 con saldo 100.0  Cuenta 1001 con saldo -50.0  Cuenta 1002 con saldo 100.0  ...  No cuenta con los fondos necesarios (RuntimeError) |

***Ejercicio 6***

Retomando el ***Ejercicio 6*** del ***Taller 1***, implementar la familia de clases del **sistema de tarjetas de viaje en subte** de tal forma que con el siguiente programa de prueba se obtenga la salida indicada en los comentarios:

|  |
| --- |
| require\_relative 'subway\_central'  require\_relative 'rechargeable\_subway\_card'  require\_relative 'fixed\_subway\_card'  require\_relative 'discount\_rechargeable\_subway\_card'  subway\_central = SubwayCentral.new(7.5)  rechargeable\_card = RechargeableSubwayCard.new(subway\_central)  rechargeable\_card.recharge(15)  rechargeable\_card.ride  puts rechargeable\_card # Rechargeable Subway Card [Central: Subway Central [$7.5]] [Balance: $7.5]  subway\_central.ride\_cost = 8.00  puts rechargeable\_card # Rechargeable Subway Card [Central: Subway Central [$8.0]] [Balance: $7.5]  # rechargeable\_card.ride # Not enough funds (RuntimeError)  fixed\_card = FixedSubwayCard.new(subway\_central, 3)  fixed\_card.ride  fixed\_card.ride  puts fixed\_card # Fixed Subway Card [Central: Subway Central [$8.0]] [Rides: 1]  fixed\_card.ride  #fixed\_card.ride # No rides available (RuntimeError)  discount\_card = DiscountRechargeableSubwayCard.new(subway\_central, 0.5)  discount\_card.recharge(10)  discount\_card.ride  puts discount\_card # Rechargeable Subway Card [Central: Subway Central [$8.0]] [Balance: $6.0] |

Para las líneas de código comentadas, en caso de descomentarlas, el programa aborta con el mensaje indicado en el comentario.

Agregar luego las validaciones necesarias para los parámetros numéricos. Por ejemplo, para el siguiente fragmento de código:

|  |
| --- |
| subway\_central = SubwayCentral.new('Hello World!') |

el programa debería abortar con el siguiente mensaje:

|  |
| --- |
| Invalid Ride Cost (RuntimeError) |

***Ejercicio 7***

Consultar la documentación de la clase Object e indicar 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 |

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

***Ejercicio 8***

Indicar en papel la salida del siguiente programa de prueba que utiliza la clase Point implementada en el ***Ejercicio 3***. Luego verificarlo en el entorno Ruby.

|  |
| --- |
| 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' |

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

***Ejercicio 9***

Agregar a la familia de clases de figuras geométricas del ***Ejercicio 4*** todo lo necesario para que las clases puedan responder correctamente a la **equivalencia**. Actualice el diagrama de clases correspondiente. Con el siguiente programa de prueba:

|  |
| --- |
| my\_rectangle = Rectangle.new(Point.new(0, 0), Point.new(2, 4))  puts my\_rectangle == Rectangle.new(Point.new(0, 0), Point.new(2, 4))  my\_circle = Circle.new(Point.new(0, 0), 3)  puts my\_circle == Circle.new(Point.new(0, 0), 3)  puts my\_rectangle == my\_circle |

se debe obtener la siguiente salida:

|  |
| --- |
| true  true  false |

***Ejercicio 10***

Dada la siguiente jerarquía de clases, con los métodos de instancia indicados para cada una, se cuenta con cuatro instancias homónimas a la clase a la cual pertenecen.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **class A**  def **m\_1**  m\_3  end  def **m\_2**  10  end  def **m\_3**  5  end  def **m\_4**  m\_4  end  **end** | **class B < A**  def **m\_1**  8  end  def **m\_2**  super.m\_1  end  def **m\_4**  20  end  def **m\_5**  m\_3  end  def **m\_7**  super.m\_4  end  **end** | **class C < B**  def **m\_1**  super  end  def **m\_2**  m\_6  end  def **m\_6**  3  end  **end**  **class D < B**  def **m\_1**  super.m\_3  end  def **m\_3**  2  end  def **m\_5**  m\_4  end  **end** |

Completar el siguiente cuadro indicando qué se obtiene al enviar los mensajes de la primera fila (por separado, no en cascada) a la instancia de la primera columna. Primero hacerlo en papel para verificar que se entendió la invocación a métodos y el uso de la palabra reservada super. Luego verificarlo en el entorno Ruby.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **m\_1** | **m\_2** | **m\_3** | **m\_4** | **m\_5** | **m\_6** | **m\_7** |
| **A** |  |  |  |  |  |  |  |
| **B** |  |  |  |  |  |  |  |
| **C** |  |  |  |  |  |  |  |
| **D** |  |  |  |  |  |  |  |