# CIS*3260 F22 – [Individual] Assignment 1

**Update:** All references to the Throw class changed to the Results class (this year's name for the class)
RandomizerCollection.count(rc) changed to RandomizerCollection.count()

## Purpose

- Implement classes and methods in Ruby from a given (simple) type/interface design provided
- The system being designed models a player with a cup full of dice and or coins;
  the cup is thrown, and the results recorded

## Main idea behind the system to be built

- Each "player" has a bag and a cup
- Dice and coins are created using a class method, and placed in a player's bag
- Dice and coins can be removed from a bag (using a "selection description" described below)
  and placed in a cup
- When moving dice and coins they are placed in a "hand" object, which must be iterable
  (e.g. a stack, queue, or list); the implementation is your choice
- Each of the dice and coins are randomized when the cup is "thrown"
    - This returns a Results object which holds the results rolls/flips of all the dice/coins in the cup
    - Note: the items remain in the cup
        - only the results are recorded (however you want to record them)
        - a cup can be thrown multiple times, each time produces a different "result"
        - the new result should not change the old result
- The items in the cup can be emptied back into the player's bag
- A different selection of coins and dice can be placed into the cup, which again can be thrown
- Each player stores the result of each throw, and the player can be asked for the results as well as
  simple statistical summaries of the throws

## How to understand the requirements below

**You are coding "support" classes, not a program**

- The classes described below is not a program with a "mainline"
- Rather they comprise a "class library" that can be used by a script or irb session to support a
  "game" with players that use coins and dice in order to play

**How to read the class descriptions provided below**

- You are not being given complete descriptions of the classes and methods to be implemented
- For example, no instance variables are specified
- In effect, you are being supplied with the class interface (the API),
  not the complete implementational details
- This way you have complete flexibility to design the internals of the class as you see fit

# The classes and methods to be implemented in Ruby

**Randomizer Classes**

`Randomizer` *an "abstract class", stores common methods*

- `randomize()`
  *both randomizes as well as returns self (for method chaining)*
- `result()`
  *returns the result of the randomization, or nil if never randomized*
- `calls()`
  *returns the number of randomizations performed*
- `reset()`
  *sets the result to nil and number of randomizations performed to 0*
  *returns self (for method chaining)*

`Coin` *<inherits from Randomizer>*

- `initialize(denomination:Enum)`
  *constructor (i.e. Coin.new(denomination))*
- `denomination()`
  *returns the denomination of the coin (does not set it)*
- `flip()`
  *flips the coin*
  *returns self (for method chaining)*
  *is a synonym for randomize()*
- `sideup()`
  *returns :H or :T (the result of the last flip) or nil (if no flips yet done)*
  *is a synonym for result()*

`Die` *<inherits from Randomizer>*

- `initialize(sides:Int, colour:Enum)`
  *constructor (i.e. Die.new(sides, colour))*
- `colour()`
  *returns the colour of the die (does not set it)*
- `sides()`
  *returns the number of sides ( does not set it)*
- `roll()`
  *randomizes and returns self (for method chaining)*
  *is a synonym for randomize()*
- `sideup()`
  *returns 1..sides or nil*
  *is a synonym for result()*

*Note: Int and Enum are only indications of the arg types expected by the method.*

- *UML style type-notation has been used,*
  - *e.g.,* `method foo(bar:int)` *not* `method foo(int bar)`
- *However, the types (of any style) will not be present in your code as Ruby is "duck typed", so no type declarations are used in Ruby code*
- *Enum does not need to be implemented as a class, but rather a protocol … see the description of how enumerations should be handled on pg 5.*

**RandomizerContainer Classes**

`RandomizerContainer` *an "abstract class", stores common methods*

- `store(r:Randomizer)`
  *stores a randomizer in the container*
  *returns self (for method chaining)*
- `store_all(l:List)`
  *stores all randomizers from a list*
  *returns self (for method chaining)*
- `count()`
  *returns the count of all randomizer stored in itself*
- `move_all(rc:RandomizerContainer)`
  *remove each randomizer in rc & store it in self*
  *returns self (for method chaining)*
- `empty()`
  *abstract method that should remove all members from the container*
- `select(description:Hash, amt=:all)`
  *selects items based on the description provided (see the section on descriptions on pg. 5)*
  *remove the selected items from self;*
  *returns a Hand object that holds the selected items up to the number entered into amount*
  *(if you want all items, supply the symbol :all instead of a number)*

`Bag` *<Inherits from RandomizerContainer>*
*when store() or move_all() invoked, Bag makes sure that all randomizers added to the bag are reset*

- `empty()`
  *empties all items from the Bag into a Hand, which is returned*

`Hand` *<Inherits from RandomizerContainer>*

- `next()`
  *removes and returns the last objected added to the hand*
  *if no objects are in the hand, return nil*
- `empty()`
  *returns nil (items are "dropped on the ground") i.e. the pointers to the contained objects are lost (and the objects will be garbage collected by the system)*

`Cup` *<Inherits from RandomizerContainer>*

- `throw()`
  *each randomizer in the cup is rolled or flipped*
  *the results of the thrown randomizers are stored in a newly created Results object, which is returned*
- `load(hand:Hand)`
  *enters each randomizer from a Hand* (*synonym of* `move_all()`)
- `empty()`
  *returns a Hand object to be returned to the bag, and leaves the cup empty*

*Note: All RandomizerContainer objects are created empty.*
*Therefore, they are just called with 'new', e.g. Cup.new*

**High Level Classes**

Results

- `initialize(cup: Cup)`
  *constructor (i.e. Results.new(cup))*
- `description(description:Hash)`
  *Stores a description in Results from which the "randomizer" objects stored in Results (i.e. in self) can be selected when computing the results, tally or sum*
- `results()`
  *Returns an array containing the "side-up" values of the randomizers recorded in the Results (i.e. in self). Only include the values from randomizers that match the description stored in the Results. If description() has not yet been called, return the results from all randomizers*
- `tally()`
  *Counts the items in the that match the description and returns the value If description() has not yet been called, count all randomizers*
- `sum()`
  *totals the value of the randomizer items in the Results that match the description, where the value equals the number that is "up" (for coins, :H = 1 and :T = 0), and returns the value*
  *If description() has not yet been called, total the values across all randomizers*

Player

- `initialize(name:String)`
  *constructor (i.e. Player.new("Gandalf the Grey"))*
- `name()`
  *returns the name of the player (does not set it)*
- `store(item:Randomizer)`
  *stores the item in the player's bag*
  *returns self (for method chaining)*
- `move_all(rc:RandomizerContainer)`
  *gets each item in rc and stores it in the player's bag*
  *returns self (for method chaining)*
- `load(description:Hash = { })`
  *loads items from the player's bag to the player's cup based on the description*
  *returns self (for method chaining)*
- `throw()`
  *throws the (loaded) cup*
  *store and return the Results of the "thrown items" (which are still stored in the cup)*
- `replace(description:Hash = { })`
  *replaces the items selected by the description from the cup into the bag*
  *returns self (for method chaining)*
- `clear()`
  *clears all stored results*
  *returns self (for method chaining)*
- `tally(description:Hash = { })`
  *sets the description, and calls tally() on each of the stored results and returns each of the values within a single array*
- `sum(description:Hash = { })`
  *sets the description, and calls sum() on each of the stored results and returns the combined values as an array*
- `results(description:Hash = { }, throw:Int = 0)`
  *sets the description and returns the result values as an array, where the last Results is "throw=0", the throw before is "throw=1", etc. If a throw is requested that doesn't exist (too far back in time and never occurred), return nil Here a "throw" is short for "the result of a given throw"*

## Enumerations

Enum is not actually a class; it is a duck-type style interface, i.e., a protocol.
E.g. colour is consistently represented by symbols such as `:red` and `:blue`

There is no class holding or enforcing these values
(although they can be error checked against by the methods that accept them if you wish).

Here are the enumeration protocols used for the Randomizers:

- enum used for Randomizer items = `:coin, :die`
- enum used for coin denominations = `0.05, 0.10, 0.25, 1, 2`
- enum used for die colours = `:red, :green, :blue, :yellow, :black, :white`
- enum used for coin sides = `:H, :T`

## Descriptions

Descriptions are lists of pairs (i.e., Hash class, a.k.a. "dictionaries" in other programming languages). Each pair contains an 'attribute' with its associated 'value'.

Every Randomizer object has instance variables defined by the class, which stores a value.
These are what the attribute/value pair is describing.
E.g., a 'die' object has an instance variable called 'colour', which can hold a value such as :red or :green.

All Randomizers have the instance variables: 'sides' and 'up'.
The 'coin' has a 'denomination' attribute
The 'die' randomizer has a 'colour' attribute.

All Randomizers also have the 'item' attribute. This is a special attribute/value pair as it doesn't refer to an instance variable within the object, but rather the object's class name, which in our case can either be either :coin or :die.

A description is used to select items from a RandomizerCollection. Each attribute/value pair in the Hash must be satisfied for the object in the collection to be selected. I.e., each pair can be thought of as an 'AND'.

If an attributed is not mentioned in the description, any value it holds is acceptable and the object can be selected provided it matches the other attributes mentioned in the description.

If an attribute from the description does not exist in the Randomizer object, then the description value of the attribute cannot be matched against. Consequently, the object will not be selected.

Some description examples:

- `{ item: :die, sides: 4, colour: :red, up: 4 }` ← *selects all 4-sided red dice showing the number 4*
- `{ item: :die, colour: :blue }` ← *selects all blue dice*
- `{ colour: :blue }` ← *selects same as above*[1]
- `{ item: :die, sides: 6 }` ← *selects all 6-sided dice*
- `{ item: :die }` ← *selects all dice*
- `{ item: :coin, denomination: 0.25 }` ← *selects all quarters*
- `{ item: :coin, up: :H }` ← *selects all coins that are heads up*
- `{ item: :coin }` ← *selects all coins*
- `{ }` ← *selects everything*

Note: an "impossible" description just means that you will not select any of the objects
e.g., each of these descriptions will not select any item

- `{ item: :coin, sides: 6 }` ← *a coin can only have 2 sides*
- `{ item: :die, sides: 4 , sides: 6}` ← *a die cannot both have 4 and 6 sides*
- `{ item: nil }` ← *nil is not a :coin or :die*

[1] Coins don't have a `colour` instance variable, so it can't be `:blue` (it is assumed to be `nil`). Consequently, it won't be selected.

## Creating Use Cases

As part of the "reverse engineering" process, I want you to create at least 4 use cases to document the classes as they would be typically used. You may come up with any scenario that you want.

As described in class, make sure that the use cases are written at the right level. Not so narrow so that a use case only runs a single method, or only involves a single class; not so broad that it involves the majority of the library all at once.

## Testing the class behaviour

To prove that the classes you have coded work, you must create a script that utilize the classes and methods in the library you are creating

This script is to be called '`testgames.rb`'

At least some of your tests should be based on the use cases as described above.

Your tests should exercise each of the methods described in the assignment at least once each. This does not mean you have to have one test per method. A single test could utilize many of the methods all at once.

## Grading

- [6 pts] The ruby code i.e. how much functionality does your written code try to encapsulate
- [6 pts] Use Cases
- [12 pts] Execute your test cases and report the results
    - i.e. for each test
        - state purpose of test,
        - state expected output
        - report actual output
        - report pass/fail
- [6 pts] Passes our tests
- [5 pts] OO Style
- [1 pt] Ruby Style

## Submit in Courselink

- Each class is kept in its own file (ClassName.rb)
- Provide a script A1_Classes.rb that loads all your classes by successively calling
  ```
  require ClassName.rb
  ```
  on the successive class files; one 'require' per line
- The script demonstrating that your classes are working properly, should be called testgames.rb
  You should also require A1_Classes.rb at the beginning of the script
- Provide a .pdf or .txt file describing your use cases for the above testing script
- Provide a file that has the results of your tests (see grading),
  called mytest.pdf or mytest.txt
- Archive all files together (e.g. as a .zip file)