I – Enumerations and Composition

Code Samples - Documentation

# Examples

The following examples are used to illustrate this topic.

#### Enumerations:

1. **Coin + CoinFace** – The CoinFace enumeration helps to describe the two sides of a Coin, which can then be used in a tossing game.
2. **Account + AccountType** – The Account’s account type is no longer being represented as a string, but is its own enumeration: AccountType.
3. **LetterGrade + QuebecLetterGrade** – The QuebecLetterGrade uses the simple LetterGrade enumeration and assigns specific ranges of percentage marks for the possible LetterGrade values.

#### Composition:

1. **Address + Student + GenderType** – This revised version of the Student class now has an Address field. The address field is new; although similar to the CanadianAddress class, the Address class is simpler and more “generic” (having “State” instead of “Province” and “ZipCode” instead of “PostalCode”).
2. **ImproperFraction + MixedNumber + ProperFraction** – In this sample, the idea of a Fraction class is made more specific by replacing it with three more specific types of numbers: MixedNumber, ProperFraction, and ImproperFraction. A MixedNumber is made up of a whole number and a ProperFraction. A MixedNumber can also be expressed as or derived from an ImproperFraction. The reciprocal of a ProperFraction is an ImproperFraction and the reciprocal of an ImproperFraction is a ProperFraction.

# Coin + CoinFace

The CoinFace enumeration helps to describe the two sides of a Coin, which can then be used in a tossing game.

## Problem Statement

Write the code needed to represent a coin that could be used in a coin-toss game. The solution must meet the following requirements.

* Should randomly generate the coin face that is showing when creating the coin
* Should get the side of the coin that is face showing
* Should allow the coin to be tossed to randomly generate a new coin face
* Should only support two sides for a coin’s face: Heads and Tails

Use the following diagram when creating your solution.

## Code Solution

public class Coin

{

public enum CoinFace { HEADS, TAILS };

public CoinFace FaceShowing { get; private set; }

public Coin()

{

Toss();

}

public void Toss()

{

if (Rnd.Next(2) == 0)

FaceShowing = CoinFace.HEADS;

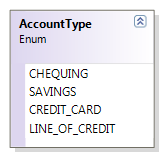
else

FaceShowing = CoinFace.TAILS;

}

}

# Account + AccountType

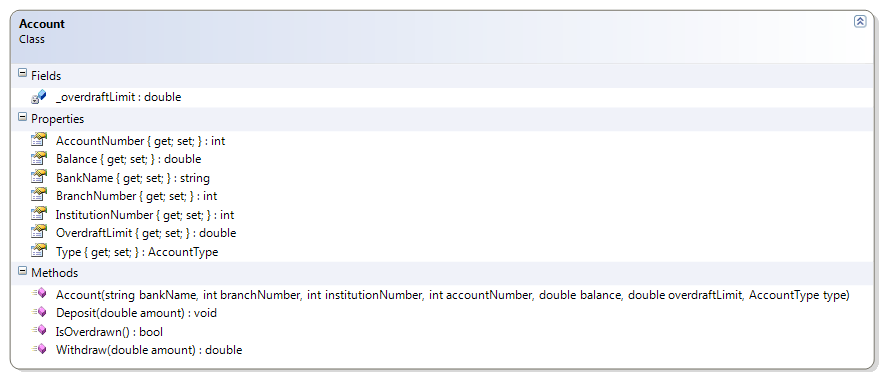
The Account’s account type is no longer being represented as a string, but is its own enumeration: AccountType. 

## Problem Statement

Write the code that will make the account type a type-safe value for the Account class. The solution must meet the following requirements (new requirements are in ***green, bold italic*** font):

* Should get the bank name, branch number, institution number, account number, balance, overdraft limit, and account type and allow the overdraft limit to be set
* Should support deposits
* Should only support withdrawals if the amount does not exceed the sum of the balance and the overdraft limit, otherwise an exception stating “Insufficient Funds” should occur
* Should identify if the account is overdrawn
* Should require bank name and account type (that is, they cannot be empty or null)
* Should trim the bank name
* ***Should ensure that the Account Type is type-safe and that it is supplied when creating the account***
* ***Should support the following types of accounts: Chequing, Saving, Credit Card, and Line of Credit***
* Should verify that the branch number is six digits and the institution number is three digits
* Should require an opening balance
* Should not allow a negative overdraft limit

Use the following class diagram when creating your solution.



## Code Solution

public class Account

{

public Account(string bankName, int branchNumber, int institutionNumber,

int accountNumber, double balance, double overdraftLimit,

AccountType type)

{

if (string.IsNullOrEmpty(bankName) ||

string.IsNullOrEmpty(bankName.Trim()))

throw new System.Exception("Bank name cannot be empty");

if (branchNumber < 10000 || branchNumber > 99999)

throw new System.Exception("Branch number must be 5 digits");

if (institutionNumber < 100 || institutionNumber > 999)

throw new System.Exception("Institution number must be 3 digits");

if (balance <= 0)

throw new System.Exception("Opening balance must be greater than zero");

OverdraftLimit =overdraftLimit;

this.BankName = bankName;

this.BranchNumber = branchNumber;

this.InstitutionNumber = institutionNumber;

this.AccountNumber = accountNumber;

this.Balance = balance;

this.Type = type;

}

private double \_overdraftLimit;

public double Balance { get; private set; }

public string BankName { get; private set; }

public int BranchNumber { get; private set; }

public int InstitutionNumber { get; private set; }

public int AccountNumber { get; private set; }

public AccountType Type { get; private set; }

public double OverdraftLimit

{

get

{ return \_overdraftLimit; }

set

{

if (value < 0)

throw new System.Exception("Negative overdraft limits not allowed");

this.\_overdraftLimit = value;

}

}

public bool IsOverdrawn()

{

return Balance < 0.0;

}

public void Deposit(double amount)

{

Balance += amount;

}

public double Withdraw(double amount)

{

if (amount > Balance + \_overdraftLimit)

throw new System.Exception("Insufficient Funds");

Balance -= amount;

return amount;

}

}

# LetterGrade + QuebecLetterGrade

The QuebecLetterGrade uses the simple LetterGrade enumeration and assigns specific ranges of percentage marks for the possible LetterGrade values.

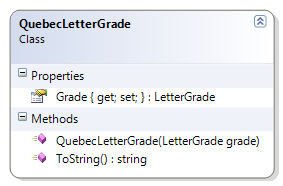
## Problem Statement

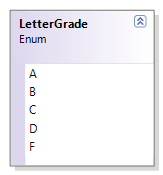
Write the code for the LetterGrade class that represents a letter grade as assigned in Quebec universities (Source: <http://en.wikipedia.org/wiki/Letter_grade>). The solution must meet the following requirements (new requirements are in ***green, bold italic*** font):

* ***Should get and set the grade as a type-safe value***
* Should get the appropriate descriptions for the grade, based on the following table:

|  |  |
| --- | --- |
| Grade | Description |
| A | A - 80-100% - Greatly Above Standards |
| B | B - 70-79% - Above Standards |
| C | C - 60-69% - At Government Standards |
| D | D - 50-60% - Lower Standards |
| F | F - 0-49% - Failure |

Use the following class diagram when creating your solution.





## Code Solution

public enum LetterGrade

{

A, B, C, D, F

}

public class QuebecLetterGrade

{

public LetterGrade Grade { get; set; }

public QuebecLetterGrade(LetterGrade grade)

{

this.Grade = grade;

}

public override string ToString()

{

string description;

switch (Grade)

{

case LetterGrade.A:

description = "A - 80-100% - Greatly Above Standards";

break;

case LetterGrade.B:

description = "B - 70-79% - Above Standards";

break;

case LetterGrade.C:

description = "C - 60-69% - At Government Standards";

break;

case LetterGrade.D:

description = "D - 50-60% - Lower Standards";

break;

case LetterGrade.F:

description = "F - 0-49% - Failure";

break;

default:

description = "Invalid Letter Grade";

break;

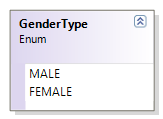
}

return description;

}

}

# Address + Student + GenderType

This revised version of the Student class now has an Address field. The address field is new; although similar to the CanadianAddress class, the Address class is simpler and more “generic” (having “State” instead of “Province” and “ZipCode” instead of “PostalCode”). 

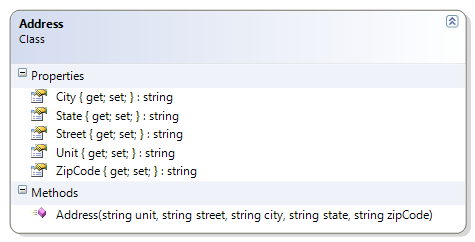
## Problem Statement

Extend the Student class to now include information about the student’s home address. Create an Address class to represent a simple, generic address. In addition, the Student class must represent the Gender using an enumeration.

The solution must meet the following requirements (new requirements are in ***green, bold italic*** font):

* ***The Address class must***
  + ***Get and set the city, state, street, unit and zip code.***
* The Student class must
  + ***Verify that an address object is supplied (is not null)***
  + ***Represent the Gender as a GenderType of Male and Female***
  + Should get and set the student’s name, gender, GPA, program of studies, and whether or not the student is full-time.
  + Should override the toString() method to get the student’s ID and name in this format:  
    (ID) Name
  + Should no longer allow the student ID to be set (it’s only set through the constructor)
  + Should reject empty text (and null values) for the student’s name and program of studies.
  + Should trim the student’s name and the program name
  + Should only accept ‘M’ and ‘F’ as valid genders
  + Should set the gender to upper-case
  + Should reject negative GPAs and GPAs over 9

Use the following class diagrams to guide your design.





## Code Solution

public enum GenderType

{

MALE, FEMALE

}

public class Address

{

public string Street { get; set; }

public string Unit { get; set; }

public string City { get; set; }

public string State { get; set; }

public string ZipCode { get; set; }

public Address(string unit, string street, string city, string state,

string zipCode)

{

this.Unit = unit;

this.Street = street;

this.City = city;

this.State = state;

this.ZipCode = zipCode;

}

}

public class Student

{

private string \_name; // The full name of the student

private string \_program; // The course program; e.g.: "CST"

private double \_gradePointAverage; // GPA is from 1.0 to 9.0

private Address \_homeAddress;

public Student(string name, Address homeAddress, GenderType gender,

int studentId, string program, double gradePointAverage,

bool isFullTime)

{

if (studentId < 100000000 || studentId > 999999999)

throw new System.Exception("Student Ids must be 9 digits");

Name = name;

HomeAddress = homeAddress;

Gender = gender;

StudentId = studentId;

Program = program;

GradePointAverage = gradePointAverage;

IsFullTime = isFullTime;

}

public GenderType Gender { get; set; }

public bool IsFullTime { get; set; }

public int StudentId { get; private set; }

public string Name

{

get

{ return \_name; }

set

{

if (string.IsNullOrEmpty(value) ||

string.IsNullOrEmpty(value.Trim()))

throw new System.Exception("name cannot be empty");

this.\_name = value.Trim();

}

}

public Address HomeAddress

{

get

{ return \_homeAddress; }

set

{

if (value == null)

throw new System.Exception("Address is required");

this.\_homeAddress = value;

}

}

public string Program

{

get

{ return \_program; }

set

{

if (string.IsNullOrEmpty(value) ||

string.IsNullOrEmpty(value.Trim()))

throw new System.Exception("Program cannot be empty");

this.\_program = value.Trim();

}

}

public double GradePointAverage

{

get

{ return \_gradePointAverage; }

set

{

if (value < 0 || value > 9)

throw new System.Exception("GPA must be between 0 and 9 inclusive");

this.\_gradePointAverage = value;

}

}

public override string ToString()

{

return "(" + StudentId + ") " + Name;

}

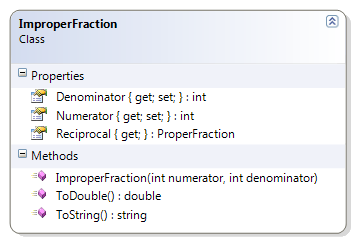
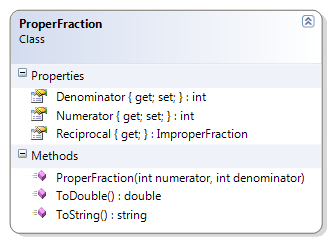
}

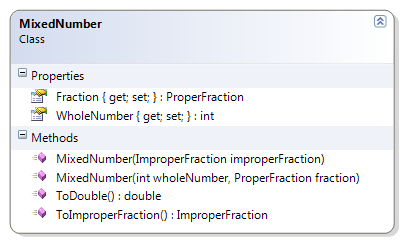
# ImproperFraction + MixedNumber + ProperFraction

In this sample, the idea of a Fraction class is made more specific by replacing it with three more specific types of numbers: MixedNumber, ProperFraction, and ImproperFraction. A MixedNumber is made up of a whole number and a ProperFraction. A MixedNumber can also be expressed as or derived from an ImproperFraction. The reciprocal of a ProperFraction is an ImproperFraction and the reciprocal of an ImproperFraction is a ProperFraction.

## Problem Statement

The generic Fraction class is now being replaced with a set of three more specific classes for numbers with fractional values: ProperFraction, ImproperFraction, and MixedNumber. Create these classes, using the following requirement and class diagrams.



* + ProperFraction should ensure that it is indeed a proper fraction
  + ImproperFraction should ensure that it is indeed an improper fraction
  + MixedNumber should construct either from a whole number and a proper fraction or from an improper fraction
  + ProperFraction and ImproperFraction should get their respective reciprocals as well as their values as real numbers
  + ProperFraction and ImproperFraction should ensure that the denominator is always positive
  + MixedNumber should get the whole number portion as well as the fractional portion
  + MixedNumber should get its value as a real number as well as get its value as an ImproperFraction

## Code Solution

public class ProperFraction

{

public ProperFraction(int numerator, int denominator)

{

if (denominator == 0)

throw new System.Exception("Zero denominator fractions are undefined");

if (Math.Abs(numerator) >= Math.Abs(denominator))

throw new System.Exception("Proper fractions must have a numerator that is less than the denominator");

if (denominator < 0)

{

numerator \*= -1;

denominator \*= -1;

}

this.Numerator = numerator;

this.Denominator = denominator;

}

public int Numerator { get; private set; }

public int Denominator { get; private set; }

public ImproperFraction Reciprocal

{

get

{ return new ImproperFraction(Denominator, Numerator); }

}

public double ToDouble()

{

return (double)(Numerator) / Denominator;

}

public override string ToString()

{

return Numerator + "/" + Denominator;

}

}

public class ImproperFraction

{

public ImproperFraction(int numerator, int denominator)

{

if (denominator == 0)

throw new System.Exception("Zero denominator fractions are undefined");

if (Math.Abs(numerator) < Math.Abs(denominator))

throw new System.Exception("Improper fractions must have a numerator that is greater than or equal to the denominator");

if (denominator < 0)

{

numerator \*= -1;

denominator \*= -1;

}

this.Numerator = numerator;

this.Denominator = denominator;

}

public int Numerator { get; private set; }

public int Denominator { get; private set; }

public ProperFraction Reciprocal

{

get { return new ProperFraction(Denominator, Numerator); }

}

public double ToDouble()

{

return (double)(Numerator) / Denominator;

}

public override string ToString()

{

return Numerator + "/" + Denominator;

}

}

public class MixedNumber

{

public int WholeNumber { get; private set; }

public ProperFraction Fraction { get; private set; }

public MixedNumber(int wholeNumber, ProperFraction fraction)

{

if (wholeNumber == 0)

throw new System.Exception("wholeNumber portion cannot be zero for a Mixed Number");

if (fraction == null)

throw new System.Exception("MixedNumbers must have a fractional portion");

if (fraction.Numerator < 0)

{

fraction = new ProperFraction(-fraction.Numerator, fraction.Denominator);

wholeNumber \*= -1;

}

this.WholeNumber = wholeNumber;

this.Fraction = fraction;

}

public MixedNumber(ImproperFraction improperFraction) :

this(improperFraction.Numerator / improperFraction.Denominator,

new ProperFraction(improperFraction.Numerator %

improperFraction.Denominator,

improperFraction.Denominator))

{

}

public ImproperFraction ToImproperFraction()

{

return new ImproperFraction(WholeNumber \*

Fraction.Denominator + Fraction.Numerator,

Fraction.Denominator);

}

public double ToDouble()

{

double realValue = Math.Abs(WholeNumber) + Fraction.ToDouble();

if (WholeNumber < 0)

realValue \*= -1;

return realValue;

}

}