G – Raising Exceptions

Code Samples – Documentation

# Examples

The following examples are used to illustrate this topic.

1. **Fraction** – The fraction class avoid the division by zero error by ensuring that the supplied denominator is not zero.
2. **Square** – Only accepts positive, non-zero lengths for the side.
3. **Circle** – Only accepts positive, non-zero diameters.
4. **Die** – Only accepts from 4 to 20 sides for a die.
5. **Person** – First and last names cannot be empty and the birth date cannot be in the future. This illustrates putting the validation on the setters and calling the setters from the constructor (to reduce the duplication of code).
6. **Student** – Gender must be ‘M’ or ‘F’ (and will be converted to upper-case). The student name and program cannot be empty. The student ID must be 9 digits. The GPA must be between 0.0 and 9.0 (inclusive).
7. **ParkingCounter** – Only accepts positive, non-zero counts for available parking spaces and number of cars. Also, the number of cars must not exceed the number of parking spaces. The rules for the number of cars must also apply for cars entering and leaving the parking lot.
8. **StockItem** – Represents an inventory item that is kept in stock. The item’s description, cost and profit margin are all part of the class design. Empty descriptions and zero or negative costs, as well as profit margins less than -100, are not allowed.
9. **Account** – The following account information is now verified when the class is created:
   1. Bank name and account type cannot be empty
   2. The opening balance must be greater than zero
   3. The overdraft limit cannot be negative
   4. The institution number must be 3 digits
   5. The branch number must be 6 digits
   6. Attempts to withdraw amounts beyond the overdraft limit should throw an “Insufficient Funds” exception

# Fraction

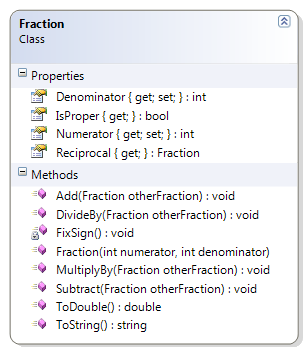
The fraction class avoids the division by zero error by ensuring that the supplied denominator is not zero.

## Problem Statement

Write the code for the Fraction class. The solution must meet the following requirements (new requirements are in ***green, bold italic*** font):

* Should get the string representation of the fraction, as “numerator/denominator”
* Should get the numeric value of the fraction (as a real number)
* Should get the reciprocal of the fraction
* Should get the numerator and denominator
* Should add another fraction to its existing value
* Should subtract another fraction from its existing value
* Should multiply its existing value by another fraction
* Should divide its existing value by another fraction
* Should affix the sign for negative fractions onto the numerator only
* Should identify if the fraction is a proper fraction
* ***Should reject zero denominators***

Use the following class diagram when creating your solution.



## Code Solution

public Fraction(int numerator, int denominator)

{

if (denominator == 0)

throw new System.Exception("A fraction cannot have a denominator of zero (0)");

Numerator = numerator;

Denominator = denominator;

FixSign();

}

# Square

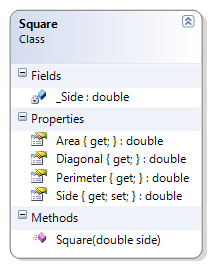
Only accepts positive, non-zero lengths for the side.

## Problem Statement

Write the code needed to add the ability for a Square to determine the length of its diagonal. The solution must meet the following requirements (new requirements are in ***green, bold italic*** font):

* Should get and set the length of the side of the square
* Should calculate the area, perimeter, and diagonal of the square
* ***Should only accept positive, non-zero lengths for the side***

Use the following class diagram when creating your solution.



## Code Solution

private double \_Side;

public double Side

{

get

{

return \_Side;

}

set

{

if (value <= 0)

throw new System.Exception("A square must have a positive non-zero length for its side");

\_Side = value;

}

}

# Circle

Only accepts positive, non-zero diameters.

## Problem Statement

Write the code for the Circle class. The solution must meet the following requirements (new requirements are in ***green, bold italic*** font):

* Should get and set the diameter
* Should calculate the area, radius, and circumference
* ***Should only accept positive, non-zero lengths for the diameter***

Use the following class diagram when creating your solution.



## Code Solution

private double \_Diameter;

public double Diameter

{

get

{

return \_Diameter;

}

set

{

if (value <= 0)

throw new System.Exception("Diameter must be a positive non-zero value");

\_Diameter = value;

}

}

# Die

Only accepts from 4 to 20 sides for a die.

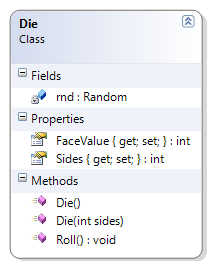
This class represents a single six-sided die. This example is used to illustrate random number generation and casting.

## Problem Statement

Write the code for the Die class. The solution must meet the following requirements (new requirements are in ***green, bold italic*** font):

* Should generate a random value from 1 to the number of sides on the die, when initially created and when re-rolled
* Should get the face value of the die
* Should get the number of sides of the die
* Should randomly generate each side (if rolled enough); for example, if the die has ten sides, it should eventually roll a 1, 2, 3, 4, 5 6, 7, 8, 9, and 10
* ***Should only accept 4 to 20 sides for the die***

Use the following class diagram when creating your solution.



## Code Solution

public Die(int sides)

{

if (sides < 4 || sides > 20)

throw new System.Exception("A die can only have from 4 to 20 sides");

this.Sides = sides;

Roll();

}

# Person

First and last names cannot be empty and the birth date cannot be in the future. This illustrates putting the validation on the setters and calling the setters from the constructor (to reduce the duplication of code).

This adaptation of the person class checks the age of the person to see if the person's life stage is infant, toddler, preschooler, school age, or adult.

## Problem Statement

Write the code that will represent a person with a first and last name and a date of birth.

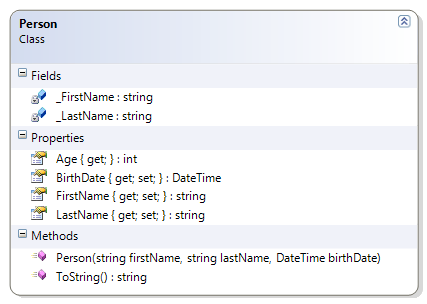
The solution must meet the following requirements (new requirements are in ***green, bold italic*** font):

* Should get and set the first and last name
* Should get the birth date
* Should get the person’s approximate age (which is the age that the person will turn to in the current year)
* Should override toString() to get the person’s full name (as first name then last name)
* Should get the life stage, based on the following table

|  |  |
| --- | --- |
| **Age Range (Years)** | **Life Stage** |
| 0 | Infant |
| < 3 | Toddler |
| < 5 | Preschooler |
| < 18 | School age |
| >= 18 | Adult |

* ***Should ensure the first and last names are not empty (or null)***
* ***Should trim leading and trailing spaces from the first and last names***
* ***Should reject birthdates that are in the future***

Use the following class diagram when creating your solution.



## Code Solution

private string \_FirstName;

public string FirstName

{

get

{

return \_FirstName;

}

set

{

if (string.IsNullOrEmpty(value) || string.IsNullOrEmpty(value.Trim()))

throw new Exception("FirstName cannot be empty");

\_FirstName = value.Trim();

}

}

private string \_LastName;

public string LastName

{

get

{

return \_LastName;

}

set

{

if (string.IsNullOrEmpty(value) || string.IsNullOrEmpty(value.Trim()))

throw new Exception("LastName cannot be empty");

\_LastName = value.Trim();

}

}

public Person(string firstName, string lastName, DateTime birthDate)

{

if (birthDate.CompareTo(DateTime.Today) > 0)

throw new System.Exception("Birthdates in the future are not allowed");

FirstName = firstName;

LastName = lastName;

BirthDate = birthDate;

}

# Student

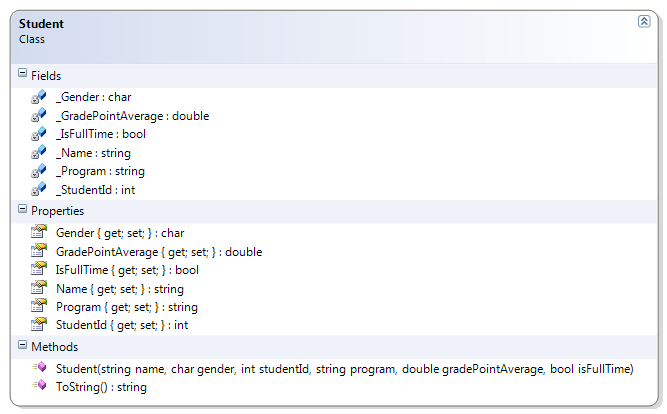
Gender must be ‘M’ or ‘F’ (and will be converted to upper-case). The student name and program cannot be empty. The student ID must be 9 digits. The GPA must be between 0.0 and 9.0 (inclusive).

## Problem Statement

Write the code for the Student class. The class must now ensure that the supplied information is valid. The solution must meet the following requirements (new requirements are in ***green, bold italic*** font):

* 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***
* ***Should require a nine-digit student ID***

This class reinforces the idea of encapsulation and constructors. It also demonstrates the idea of overloading the default ToString() method that every class inherits from the Object class.



## Code Solution

public class Student

{

#region Fields

private string \_Name;

private char \_Gender;

private int \_StudentId;

private string \_Program;

private double \_GradePointAverage;

private bool \_IsFullTime;

#endregion

#region Properties

public string Name

{

get { return \_Name; }

set

{

if (string.IsNullOrEmpty(value) || string.IsNullOrEmpty(value.Trim()))

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

\_Name = value.Trim();

}

}

public char Gender

{

get { return \_Gender; }

set

{

value = char.ToUpper(value);

if (value != 'M' && value != 'F')

throw new Exception("Gender must be M or F");

\_Gender = value;

}

}

public int StudentId

{

get { return \_StudentId; }

set

{

if (value < 100000000 || value > 999999999)

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

\_StudentId = value;

}

}

public string Program

{

get { return \_Program; }

set

{

if (string.IsNullOrEmpty(value) || string.IsNullOrEmpty(value.Trim()))

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

\_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");

\_GradePointAverage = value;

}

}

public bool IsFullTime

{

get { return \_IsFullTime; }

set { \_IsFullTime = value; }

}

#endregion

#region Constructors

public Student(string name, char gender, int studentId, string program, double gradePointAverage, bool isFullTime)

{

if (studentId < 100000000 || studentId > 999999999)

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

this.Name = name;

this.Gender = gender;

this.StudentId = studentId;

this.Program = program;

this.GradePointAverage = gradePointAverage;

this.IsFullTime = isFullTime;

}

#endregion

#region Methods

public override string ToString()

{

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

}

#endregion

}

# ParkingCounter

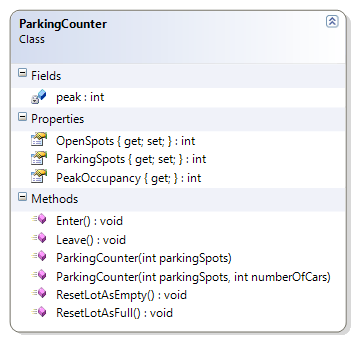
Only accepts positive, non-zero counts for available parking spaces and number of cars. Also, the number of cars must not exceed the number of parking spaces. The rules for the number of cars must also apply for cars entering and leaving the parking lot.

## Problem Statement

Write the code that will monitor vehicles entering and leaving a parking lot. The solution must meet the following requirements (new requirements are in ***green, bold italic*** font):

* Should track vehicles entering
* Should track vehicles leaving
* Should track the peak occupancy of the parking lot
  + The peak occupancy represents the highest number of cars in the parking lot at any one time
* Should get total parking spots
* Should get open (empty) spots
* Should reset lot as full (that is, fill the parking lot)
* Should reset lot as empty (that is, clear all the parking spots of vehicles)
* ***Should only allow a positive (non-zero) number of parking spots***
* ***Should not allow a negative number of cars (when using the overloaded constructor), and should not allow more cars than parking spots***
* ***Should not allow available (open) parking spots to go negative or to exceed the actual number of parking spots***
  + ***Should raise an error when trying to enter a full parking lot***
  + ***Should raise an error if trying to leave a parking lot that is already empty***

Use the following class diagram when creating your solution.



## Code Solution

public ParkingCounter(int parkingSpots)

{

if (parkingSpots <= 0)

throw new System.Exception("Negative or zero parkingSpots not allowed");

this.ParkingSpots = parkingSpots;

this.OpenSpots = parkingSpots;

this.peak = 0;

}

public ParkingCounter(int parkingSpots, int numberOfCars)

{

if (parkingSpots <= 0)

throw new System.Exception("Negative or zero parkingSpots not allowed");

if (numberOfCars < 0)

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

if (numberOfCars > parkingSpots)

throw new System.Exception("The number of cars cannot exceed the number of parking spots");

this.ParkingSpots = parkingSpots;

this.OpenSpots = this.ParkingSpots - numberOfCars;

this.peak = numberOfCars;

}

public void leave()

{

if (OpenSpots == ParkingSpots)

throw new System.Exception("Parking lot is empty");

OpenSpots++;

}

public void enter()

{

if (OpenSpots == 0)

throw new System.Exception("Parking lot is full");

OpenSpots--;

int numberOfCars = ParkingSpots - OpenSpots;

if (numberOfCars > peak)

peak = numberOfCars;

}

# StockItem

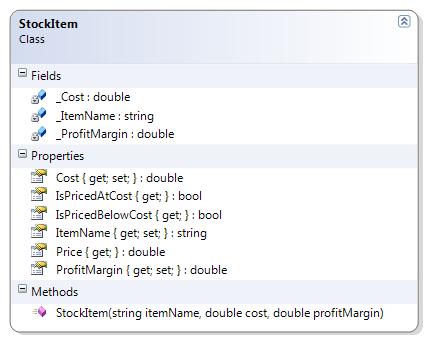
The StockItem class represents an inventory item that is kept in stock. The item’s description, cost and profit margin are all part of the class design. Empty descriptions and zero or negative costs, as well as profit margins less than -100, are not allowed.

## Problem Statement

Write the code for adding validation to the StockItem class. The solution must meet the following requirements (new requirements are in ***green, bold italic*** font):

* Should get and set the name, cost and profit margin of the stock item
* Should represent the profit margin as a percent; a value of 45 means 45%
* Should calculate the price of the item, to the nearest cent
  + Use the rounding where values under a half-cent are rounded down and values greater than or equal to a half-cent are rounded up
* Should recognize when the stock item is priced at cost (that is, the profit margin is zero)
* Should recognize when the stock item is priced below cost (that is, the profit margin is negative)
* ***Should reject an empty (or null) item name***
* ***Should trim excess spaces from the ends of the item name***
* ***Should require cost to be greater than zero***
* ***Should only allow negative profit margins up to 100% (which is a full mark-down)***

Use the following class diagram when creating your solution.



## Code Solution

private double \_Cost;

public double Cost

{

get

{

return \_Cost;

}

set

{

if (value <= 0)

throw new Exception("Cost must be positive");

\_Cost = value;

}

}

private double \_ProfitMargin;

public double ProfitMargin

{

get

{

return \_ProfitMargin;

}

set

{

if (value < -100)

throw new Exception("A profit margin below 100% (more than the cost) is not allowed");

\_ProfitMargin = value;

}

}

private string \_ItemName;

public string ItemName

{

get

{

return \_ItemName;

}

set

{

if (string.IsNullOrEmpty(value) || string.IsNullOrEmpty(value.Trim()))

throw new Exception("ItemName cannot be empty");

\_ItemName = value.Trim();

}

}

# Account

The following account information is now verified when the class is created:

1. Bank name and account type cannot be empty
2. The opening balance must be greater than zero
3. The overdraft limit cannot be negative
4. The institution number must be 3 digits
5. The branch number must be 6 digits
6. Attempts to withdraw amounts beyond the overdraft limit should throw an “Insufficient Funds” exception

## Problem Statement

Write the code that will add validation to 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 and account type***
* ***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 double Balance

{

get { return \_Balance; }

set {

if (value < -OverdraftLimit)

throw new Exception("Negative balances cannot exceed the Overdraft Limit");

\_Balance = value;

}

}

public double OverdraftLimit

{

get { return \_OverdraftLimit; }

set

{

if (value < 0)

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

\_OverdraftLimit = value;

}

}

public int BranchNumber

{

get { return \_BranchNumber; }

set {

if (value < 10000 || value > 99999)

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

\_BranchNumber = value;

}

}

public int InstitutionNumber

{

get { return \_InstitutionNumber; }

set

{

if (value < 100 || value > 999)

throw new Exception("InstitutionNumber must be a three-digit value");

\_InstitutionNumber = value;

}

}

public string AccountType

{

get { return \_AccountType; }

set {

if (string.IsNullOrEmpty(value) || string.IsNullOrEmpty(value.Trim()))

throw new Exception("Account type cannot be empty");

\_AccountType = value.Trim();

}

}

public string BankName

{

get { return \_BankName; }

set

{

if (string.IsNullOrEmpty(value) || string.IsNullOrEmpty(value.Trim()))

throw new Exception("BankName is required");

\_BankName = value.Trim();

}

}

public Account(string bankName, int branchNumber, int institutionNumber, int accountNumber, double balance, double overdraftLimit, string accountType)

{

if (balance <= 0)

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

if (balance != Math.Round(balance, 2))

throw new Exception("Opening balances cannot include a fraction of a penny");

if (overdraftLimit != Math.Round(overdraftLimit, 2))

throw new Exception("Overdraft limit amounts cannot include a fraction of a penny");

AccountNumber = accountNumber;

Balance = balance;

OverdraftLimit = overdraftLimit;

BankName = bankName;

BranchNumber = branchNumber;

InstitutionNumber = institutionNumber;

AccountType = accountType;

}

public double Withdraw(double amount)

{

if (amount != Math.Round(amount, 2))

throw new Exception("Withdrawal amounts cannot include fractions of a penny");

if (amount > Balance + OverdraftLimit)

throw new Exception("Insufficient Funds");

if (amount <= Balance + OverdraftLimit)

Balance = Math.Round(Balance - amount, 2);

else

amount = 0;

return amount;

}

public void Deposit(double amount)

{

if (amount != Math.Round(amount, 2))

throw new Exception("Deposit amounts cannot include fractions of a penny");

Balance = Math.Round(Balance + amount, 2);

}