F – If-Else Structures

Code Samples – Documentation

# Examples Overview

The following examples are used to illustrate this topic.

1. **StockItem** - This class represents an item that is part of an inventory. The item has an item name, a cost and a profit margin (which can be positive or negative). By using the profit margin, it can derive the price of the item. The class can also report if the item is priced at or below cost.
2. **Account** - This class illustrates simple if structure in handling withdrawals; withdrawals are only made when the amount does not exceed the balance and the overdraft. It also identifies if the account is overdrawn.
3. **Person** - 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*.
4. **Fraction** - This class now ensures that any negative denominators have their negative sign "moved" to the numerator. It also recognizes whether a fraction is proper (numerator less than denominator) or not.
5. **Angle** - This version of the Angle class includes an attribute to identify the type of the angle as either *acute*, *right*, *obtuse*, *straight*, *reflex*, *full rotation*, or *undefined*.
6. **ParkingCounter** - This class represents a simple counter to monitor whether a parking lot is full or not; it tracks vehicles entering and leaving the parking lot and allows the counter to be reset when the lot is full or empty. This class illustrates increment and decrement operators and/or the assignment increment or assignment decrement operators.

# StockItem

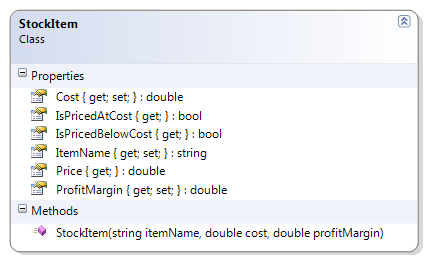
This class represents an item that is part of an inventory. The item has an item name, a cost and a profit margin (which can be positive or negative). By using the profit margin, it can derive the price of the item. The class can also report if the item is priced at or below cost.

## Problem Statement

Write the code for 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)***

Use the following class diagram when creating your solution.



## Code Solution

public bool IsPricedAtCost

{

get

{

bool atCost = false;

if (ProfitMargin == 0)

atCost = true;

return atCost;

}

}

public bool IsPricedBelowCost

{

get

{

bool belowCost;

if (ProfitMargin < 0)

belowCost = true;

else

belowCost = false;

return belowCost;

}

}

# Account

This class illustrates simple if structure in handling withdrawals; withdrawals are only made when the amount does not exceed the balance and the overdraft. It also identifies if the account is overdrawn.

## Problem Statement

Write the code that will represent a simple bank account.

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***
* ***Should identify if the account is overdrawn***

Use the following class diagram when creating your solution.



## Code Solution

public double Withdraw(double amount)

{

if (amount <= Balance + OverdraftLimit)

Balance = Balance - amount;

else

amount = 0;

return amount;

}

public bool IsOverdrawn

{

get

{

bool overdrawn;

if (Balance < 0)

overdrawn = true;

else

overdrawn = false;

return overdrawn;

}

}

# Person

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

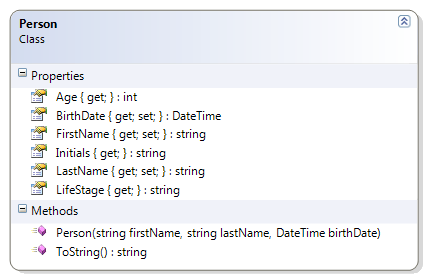
Write the code that will represent a person with a first and last name and a date of birth.

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 |

Use the following class diagram when creating your solution.



## Code Solution

public string LifeStage

{

get{

string stage;

if (Age == 0)

stage = "infant";

else if (Age < 3)

stage = "toddler";

else if (Age < 5)

stage = "preschooler";

else if (Age < 18)

stage = "school age";

else

stage = "adult";

return stage;

}

}

# Fraction

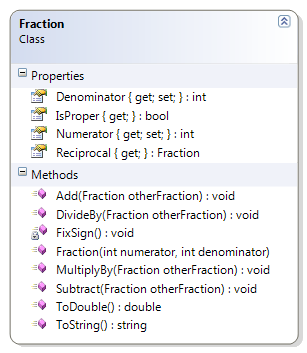
This class now ensures that any negative denominators have their negative sign "moved" to the numerator. It also recognizes whether a fraction is proper (numerator less than denominator) or not.

## 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***

Use the following class diagram when creating your solution.



## Code Solution

public Fraction(int numerator, int denominator)

{

Numerator = numerator;

Denominator = denominator;

FixSign();

}

private void FixSign()

{

if (Denominator < 0)

{

Denominator \*= -1;

Numerator \*= -1;

}

}

public bool IsProper

{

get

{

bool proper;

if (Numerator < Denominator)

proper = true;

else

proper = false;

return proper;

}

}

# Angle

This version of the Angle class includes an attribute to identify the type of the angle as either acute, right, obtuse, straight, reflex, full rotation, or undefined.

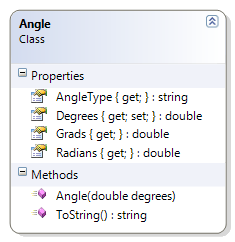
## Problem Statement

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

* Should get and set the angle’s value (in degrees)
* Should calculate the equivalent angle in Radians and Grads, using the following formulas:
  + Radians = Degrees \* (π / 180)
  + Grads = Radians \* (200 / π)
* Should override the toString() method to return the angle in degrees, in the following format:
  + degrees°
  + The Unicode character for the degrees symbol (°) is '\u00B0'
* ***Should get the type of angle, based on the following table***

|  |  |
| --- | --- |
| **Angle Range** | **Angle Type** |
| < = 0 or > 360 | Undefined |
| > 0 and < 90 | Acute |
| = 90 | Right |
| > 90 and < 180 | Obtuse |
| = 180 | Straight |
| > 180 and < 360 | Reflex |
| = 360 | Full Rotation |

Use the following class diagram when creating your solution.



## Code Solution

public string AngleType

{

get

{

string angleType;

if (Degrees <= 0)

angleType = "undefined";

else if (Degrees < 90)

angleType = "acute";

else if (Degrees == 90)

angleType = "right";

else if (Degrees < 180)

angleType = "obtuse";

else if (Degrees == 180)

angleType = "straight";

else if (Degrees < 360)

angleType = "reflex";

else if (Degrees == 360)

angleType = "full rotation";

else

angleType = "undefined";

return angleType;

}

}

# ParkingCounter

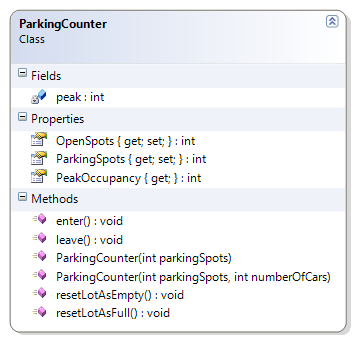
This class represents a simple counter to monitor whether a parking lot is full or not; it tracks vehicles entering and leaving the parking lot and allows the counter to be reset when the lot is full or empty. This class illustrates increment and decrement operators and/or the assignment increment or assignment decrement operators.

## 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)

Use the following class diagram when creating your solution.



## Code Solution

public int PeakOccupancy { get; private set; }

public ParkingCounter(int parkingSpots)

{

this.ParkingSpots = parkingSpots;

this.OpenSpots = parkingSpots;

this.PeakOccupancy = 0;

}

public ParkingCounter(int parkingSpots, int numberOfCars)

{

this.ParkingSpots = parkingSpots;

this.OpenSpots = this.ParkingSpots - numberOfCars;

this.PeakOccupancy = numberOfCars;

}

public void enter()

{

OpenSpots--;

int numberOfCars = ParkingSpots - OpenSpots;

if (numberOfCars > PeakOccupancy)

PeakOccupancy = numberOfCars;

}