J – Looping Structures

Code Samples - Documentation

# Examples

The following examples are used to illustrate this topic.

1. **Aggregator** – The Aggregator class supports static methods for doing various types of aggregation on Numbers. (Numbers is a supplied class that stores a series of real values that can be obtained by calling getNext().) The common aggregations of totalItems(), averageItems(), maxValue() are given as examples of simple looping.
2. **Math** – fibonnacciNumber() and isPerfect()
3. **Fraction** – The fraction example is now enhanced to simplify the fractional value by finding the greatest common denominator of the fraction’s numerator and denominator.
4. **GuessingGame** – This example uses a supplied class called SecretNumber, which picks a random whole number between a specific range. An instance of the SecretNumber is passed to the constructor of the GuessingGame, and the GuessingGame’s guess() method attempts to get the correct guess within a specified number of tries.

# Aggregator

The Aggregator class supports static methods for doing various types of aggregation on Numbers. (Numbers is a supplied class that stores a series of real values that can be obtained by calling GetNext().) The common aggregations of TotalItems(), AverageItems(), MaxValue() are given as examples of simple looping.

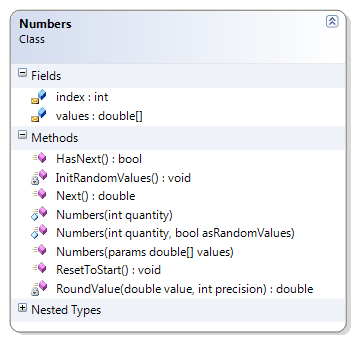
## Problem Statement

### The Numbers Class

The methods of the Aggregator class all take a single argument: a Numbers object. The Numbers object contains a bunch of real numbers and it has two methods that the Aggregator methods will use:

* **HasNext()** – This returns a Boolean indicating that the object has another number available.
* **GetNext()** – This returns a real number from the set of values in the Numbers object.

For example, if a Numbers object has a set of four values such as 1.0, 5.5, 3.2, and 5.3, then the GetNext() method can be called four times to retrieve each value. Any attempt to call GetNext() when the object’s HasNext() is false will cause an exception.

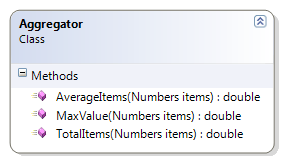


**🡸 This supporting class is already coded for you.**

### Aggregator Methods

The following Aggregator methods will demonstrate looping logic and syntax. All of these examples demonstrate the while statement.

* **TotalItems()** – This method loops through all the values inside of the Numbers object to calculate and return the total of the values.
* **AverageItems()** – This method gets all the values inside of the Numbers object so as to calculate the average value. If no values exist in the Numbers object, then the method returns an average of zero.
* **MaxValue()** – This method examines the Numbers object’s values to see which value is the largest. If there are no values inside of the Numbers object, then the method returns the smallest possible value that is supported by the programming language.



## Code Solution

public class Aggregator

{

public static double TotalItems(Numbers items)

{

double total = 0;

while (items.HasNext())

total += items.Next();

return total;

}

public static double AverageItems(Numbers items)

{

double average, total = 0;

int count = 0;

while (items.HasNext())

{

total += items.Next();

count++;

}

if (count > 0)

average = total / count;

else

average = 0;

return average;

}

public static double MaxValue(Numbers items)

{

double max = double.MinValue;

while (items.HasNext())

{

double value = items.Next();

if (max < value)

max = value;

}

return max;

}

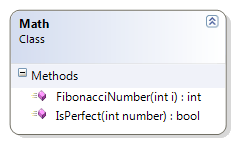
}

# Math

## Problem Statement

The Math class provides methods for the following mathematical algorithms not found in the default Math class.

* **FibonacciNumber()** – Returns a number from the Fibonacci sequence for a given position. If the given position is less than or equal to zero, then this method should throw an exception. This example demonstrates the for statement. An alternative version illustrates the do-while statement.
* **IsPerfect()** – Indicates whether or not a number is a “perfect” number. A perfect number is a number where the sum of all the factors equals the number. By definition, any number less than 1 (including all negative numbers) are not perfect numbers. This example uses a while statement. An alternative version illustrates a do-while statement



## Code Solution

public class Math

{

public static int FibonacciNumber(int i)

{

int current = 1, previous = 1, beforePrevious = 0;

if (i >= 1)

{

for (int counter = 3; counter <= i; counter++)

{

beforePrevious = previous;

previous = current;

current = beforePrevious + previous;

}

}

else

throw new Exception("Can only create a fibonnaci number based on a positive non-zero position");

return current;

}

public static bool IsPerfect(int number)

{

bool perfect = true;

if (number <= 1)

perfect = false;

else

{

// The following logic attempts to prove it is imperfect

int halfWay = number / 2;

int total = 0;

int count = 1;

while (total != number && total <= halfWay && total < number)

{

if (number % count == 0) // Then count is a factor of number

total += count;

count++;

}

if (total != number)

perfect = false;

}

return perfect;

}

}

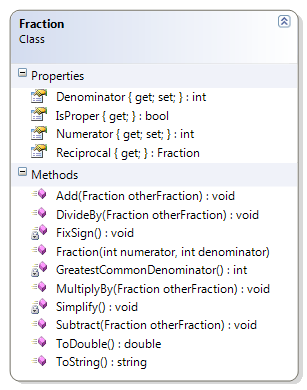
# Fraction

The fraction example is now enhanced to simplify the fractional value by finding the greatest common denominator of the fraction’s numerator and denominator.

## Problem Statement

The Fraction class represents a fraction as two integers: one for the numerator and one for the denominator. When a Fraction object is created, the class must now simplify the fractional values by calculating the greatest common denominator.

* **GreatestCommonDenominator()** – This private method is used by the Simplify() method to get the greatest common denominator for the numerator and denominator. Remember to allow for dealing with negative fractions (in which case, the negative sign will be on the numerator). This example uses a while statement (though it is possible to re-work the logic to use a do-while statement).



## Code Solution

private int GreatestCommonDenominator()

{

int commonDenominator = 1;

int count = 2, halfWay;

int absoluteNumerator = System.Math.Abs(Numerator);

if (absoluteNumerator > Denominator)

halfWay = absoluteNumerator / 2;

else

halfWay = Denominator / 2;

while (count <= halfWay)

{

if (absoluteNumerator % count == 0 &&

Denominator % count == 0)

commonDenominator = count;

count++;

}

return commonDenominator;

}

# GuessingGame

This example uses a supplied class called SecretNumber, which picks a random whole number between a specific range. An instance of the SecretNumber is passed to the constructor of the GuessingGame, and the GuessingGame’s Guess() method attempts to get the correct guess within a specified number of tries.

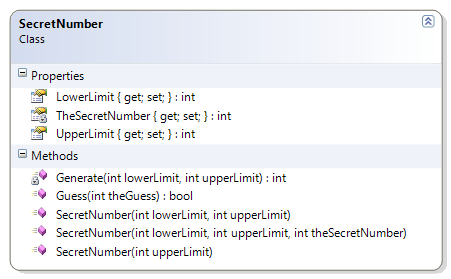
## Problem Statement

In a regular number guessing game between two people, one person asks another person to guess a whole number between a pair of values. In this example, two different classes take on the roles of the two people in the game: The SecretNumber class takes on the role of the person who has picked a value between some lower and upper limit, while the GuessingGame class takes on the role of the person who has to guess what that hidden number is.

### SecretNumber

Objects of this class will store a hidden value between some upper and lower limit (inclusive). The SecretNumber class supports three public methods:

* **GetLowerLimit()** – Returns a number representing the lower end (inclusive) of the range of possible values for the hidden value.
* **GetUpperLimit()** – Returns a number representing the upper end (inclusive) of the range of possible values for the hidden value.
* **Guess()** – Returns true if the supplied value matches the hidden value, otherwise it returns false.



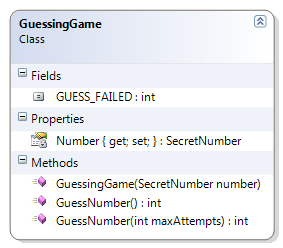
**🡸 This supporting class is already coded for you.**

### GuessingGame

This class simulates the action of guessing what hidden value is stored inside of a SecretNumber object. The SecretNumber is supplied to the constructor of the GuessingGame, and the following methods attempt to find out what that number is.

* **GuessNumber()** – This first method simply tries to guess the hidden value of the SecretNumber. It has “unlimited” guesses, and it will return the number of attempts it took to find out what that hidden number is.
* **GuessNumber(MaxAttempts : Integer)** – This method will also try to guess the SecretNumber’s hidden value, but it is limited to a maximum number of guesses. This method will return the actual number of attempts it took to find the hidden value, or it will return the GUESS\_FAILED constant if it was unable to guess the secret number.

Try creating alternate versions of this GuessingGame that will use a while statement and a for statement in the GuessNumber methods.



## Code Solution

public class GuessingGame

{

private SecretNumber Number { get; set; }

public const int GUESS\_FAILED = -1;

public GuessingGame(SecretNumber number)

{

if (number == null)

throw new System.Exception("SecretNumber cannot be null");

this.Number = number;

}

public int GuessNumber(int maxAttempts)

{

int numberOfAttempts = 0;

bool correct = false;

while (maxAttempts > 0 && !correct)

{

// Make a guess

int myBestGuess = Rnd.Next(Number.LowerLimit, Number.UpperLimit);

if (Number.Guess(myBestGuess))

correct = true;

numberOfAttempts++;

maxAttempts--;

}

if (!correct)

numberOfAttempts = GUESS\_FAILED; // a "flag" to say the guess was incorrect

return numberOfAttempts;

}

public int GuessNumber()

{

bool correct = false;

int numberOfAttempts = 0;

while (numberOfAttempts < int.MaxValue && !correct)

{

// Make a guess

int myBestGuess = Rnd.Next(Number.LowerLimit, Number.UpperLimit);

if (Number.Guess(myBestGuess))

correct = true;

numberOfAttempts++;

}

if (!correct)

numberOfAttempts = GUESS\_FAILED;

return numberOfAttempts;

}

}