Matrix class

```
//http://calcsharp.codeplex.com/license
//Microsoft Public License (Ms-PL)
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using System.IO;
using System.Drawing;

namespace CalcSharp.Core.Containers
{
[Serializable]
public abstract class AMatrix : IFormattable, ICloneable, IEquatable
{
private int nRows, nCols;

protected AMatrix(int rows, int columns)
{
if (rows < 1) throw new ArgumentException("must be greater than 0", "rows"); if (columns < 1) throw new ArgumentException("must be greater than 0", "columns"); nRows = rows; nCols = columns; } #region Properties public virtual double this[int row, int column] { get { RangeCheck(row, column); return ValueAt(row, column); } set { RangeCheck(row, column); ValueAt(row, column, value); } } public int Columns { get { return nCols; } protected set { nCols = value; } } public int Rows { get { return nRows; } protected set { nRows = value; } } #endregion #region Interface Implementations public override bool Equals(object obj) { return Equals(obj as AMatrix); } public bool Equals(AMatrix other) { // Reject equality when the argument is null or has a different shape. if (other == null) { return false; } if (Columns !=
```

```
other.Columns || Rows != other.Rows) { return false; } // Accept if the argument is the same object as this. if (ReferenceEquals(this, other)) { return true; } // If all else fails, perform elementwise comparison. for (int i = 0; i < Rows; i++) { for (int j = 0; j < Columns; j++) { if (ValueAt(i, j) != other.ValueAt(i, j)) { return false; } } } return true; } public override int GetHashCode() { int hashNum = System.Math.Max(Rows * Columns, 25); double[] hashBase = new double[hashNum]; for (int i = 0; i < hashNum; i++) { int col = i % Columns; int row = (i - col) / Rows; hashBase[i] = this[row, col]; } return hashBase.GetHashCode(); } object ICloneable.Clone() { return Clone(); } public virtual AMatrix Clone() { AMatrix result = CreateMatrix(Rows, Columns); CopyTo(result); return result; } public virtual string ToString(string format, IFormatProvider formatProvider) { StringBuilder sb = new StringBuilder(); for (int i = 0; i < Rows; i++) { for (int j = 0; j < Columns; j++) { sb.Append(ValueAt(i, j).ToString(format, formatProvider)); if (j != Columns - 1) { sb.Append(", "); } } if (i != Rows - 1) { sb.Append(Environment.NewLine); } } return sb.ToString(); } #endregion private void RangeCheck(int row, int column) { if (row < 0 || row >= nRows) throw new ArgumentOutOfRangeException(“row”);
if (column < 0 || column >= nCols) throw new ArgumentOutOfRangeException(“column”);
}

public virtual void CopyTo(AMatrix target)
{
if (target == null) throw new ArgumentNullException(“target”);
if (Rows != target.Rows || Columns != target.Columns) throw new Exception(“Target & Source rows/column count mismatch”);

for (int i = 0; i < this.nRows; i++) { for (int j = 0; j < this.nCols; j++) { target.ValueAt(i, j, this.ValueAt(i, j)); } } } public void Negate() { for (int i = 0; i < Rows; i++) { for (int j = 0; j < Columns; j++) { this.ValueAt(i, j,
```

-1*this.ValueAt(i, j)); } } } #region Abstract parts protected internal abstract void ValueAt(int row, int column, double value); protected internal abstract double ValueAt(int row, int column); protected internal abstract AMatrix CreateMatrix(int numberOfRows, int numberOfColumns); public abstract double Determinant(); public abstract AMatrix Transpose(); public abstract AMatrix Inverse(); public abstract void Clear(); public abstract double[] GetRow(int index); public abstract double[] GetColumn(int index); public abstract void SetColumn(int index, double[] source); public abstract void SetRow(int index, double[] source); public abstract Bitmap GetPreview(); #endregion } } [/csharp]