```
/*-------------------------------------------------------------------------
---
* linq.js - LINQ for JavaScript
* ver 2.2.0.2 (Jan. 21th, 2011)
*
* created and maintained by neuecc <ils@neue.cc>
* licensed under Microsoft Public License(Ms-PL)
* http://neue.cc/
* http://linqjs.codeplex.com/
*-------------------------------------------------------------------------
--*/

Enumerable = (function ()
{
    var Enumerable = function (getEnumerator)
    {
        this.GetEnumerator = getEnumerator;
    }

    // Generator

    Enumerable.Choice = function () // variable argument
    {
        var args = (arguments[0] instanceof Array) ? arguments[0] :
arguments;

        return new Enumerable(function ()
        {
            return new IEnumerator(
                Functions.Blank,
                function ()
                {
                    return this.Yield(args[Math.floor(Math.random() *
args.length)]);
                },
                Functions.Blank);
        });
    }

    Enumerable.Cycle = function () // variable argument
    {
        var args = (arguments[0] instanceof Array) ? arguments[0] :
arguments;

        return new Enumerable(function ()
        {
            var index = 0;
            return new IEnumerator(
                Functions.Blank,
                function ()
                {
                    if (index >= args.length) index = 0;
                    return this.Yield(args[index++]);
                },
```

```
                Functions.Blank);
        });
    }

    Enumerable.Empty = function ()
    {
        return new Enumerable(function ()
        {
            return new IEnumerator(
                Functions.Blank,
                function () { return false; },
                Functions.Blank);
        });
    }

    Enumerable.From = function (obj)
    {
        if (obj == null)
        {
            return Enumerable.Empty();
        }
        if (obj instanceof Enumerable)
        {
            return obj;
        }
        if (typeof obj == Types.Number || typeof obj == Types.Boolean)
        {
            return Enumerable.Repeat(obj, 1);
        }
        if (typeof obj == Types.String)
        {
            return new Enumerable(function ()
            {
                var index = 0;
                return new IEnumerator(
                    Functions.Blank,
                    function ()
                    {
                        return (index < obj.length) ?
this.Yield(obj.charAt(index++)) : false;
                    },
                    Functions.Blank);
            });
        }
        if (typeof obj != Types.Function)
        {
            // array or array like object
            if (typeof obj.length == Types.Number)
            {
                return new ArrayEnumerable(obj);
            }

            // JScript's IEnumerable
            if (!(obj instanceof Object) && Utils.IsIEnumerable(obj))
```

```
            {
                return new Enumerable(function ()
                {
                    var isFirst = true;
                    var enumerator;
                    return new IEnumerator(
                        function () { enumerator = new Enumerator(obj);
},
                        function ()
                        {
                            if (isFirst) isFirst = false;
                            else enumerator.moveNext();

                            return (enumerator.atEnd()) ? false :
this.Yield(enumerator.item());
                        },
                        Functions.Blank);
                });
            }
        }

        // case function/object : Create KeyValuePair[]
        return new Enumerable(function ()
        {
            var array = [];
            var index = 0;

            return new IEnumerator(
                function ()
                {
                    for (var key in obj)
                    {
                        if (!(obj[key] instanceof Function))
                        {
                            array.push({ Key: key, Value: obj[key] });
                        }
                    }
                },
                function ()
                {
                    return (index < array.length)
                        ? this.Yield(array[index++])
                        : false;
                },
                Functions.Blank);
        });
    },

    Enumerable.Return = function (element)
    {
        return Enumerable.Repeat(element, 1);
    }

    // Overload:function(input, pattern)
```

```
    // Overload:function(input, pattern, flags)
    Enumerable.Matches = function (input, pattern, flags)
    {
        if (flags == null) flags = "";
        if (pattern instanceof RegExp)
        {
            flags += (pattern.ignoreCase) ? "i" : "";
            flags += (pattern.multiline) ? "m" : "";
            pattern = pattern.source;
        }
        if (flags.indexOf("g") === -1) flags += "g";

        return new Enumerable(function ()
        {
            var regex;
            return new IEnumerator(
                function () { regex = new RegExp(pattern, flags) },
                function ()
                {
                    var match = regex.exec(input);
                    return (match) ? this.Yield(match) : false;
                },
                Functions.Blank);
        });
    }

    // Overload:function(start, count)
    // Overload:function(start, count, step)
    Enumerable.Range = function (start, count, step)
    {
        if (step == null) step = 1;
        return Enumerable.ToInfinity(start, step).Take(count);
    }

    // Overload:function(start, count)
    // Overload:function(start, count, step)
    Enumerable.RangeDown = function (start, count, step)
    {
        if (step == null) step = 1;
        return Enumerable.ToNegativeInfinity(start, step).Take(count);
    }

    // Overload:function(start, to)
    // Overload:function(start, to, step)
    Enumerable.RangeTo = function (start, to, step)
    {
        if (step == null) step = 1;
        return (start < to)
            ? Enumerable.ToInfinity(start, step).TakeWhile(function (i) {
return i <= to; })
            : Enumerable.ToNegativeInfinity(start,
step).TakeWhile(function (i) { return i >= to; })
    }
```

```
// Overload:function(obj)
// Overload:function(obj, num)
Enumerable.Repeat = function (obj, num)
{
    if (num != null) return Enumerable.Repeat(obj).Take(num);

    return new Enumerable(function ()
    {
        return new IEnumerator(
            Functions.Blank,
            function () { return this.Yield(obj); },
            Functions.Blank);
    });
}

Enumerable.RepeatWithFinalize = function (initializer, finalizer)
{
    initializer = Utils.CreateLambda(initializer);
    finalizer = Utils.CreateLambda(finalizer);

    return new Enumerable(function ()
    {
        var element;
        return new IEnumerator(
            function () { element = initializer(); },
            function () { return this.Yield(element); },
            function ()
            {
                if (element != null)
                {
                    finalizer(element);
                    element = null;
                }
            });
    });
}

// Overload:function(func)
// Overload:function(func, count)
Enumerable.Generate = function (func, count)
{
    if (count != null) return Enumerable.Generate(func).Take(count);
    func = Utils.CreateLambda(func);

    return new Enumerable(function ()
    {
        return new IEnumerator(
            Functions.Blank,
            function () { return this.Yield(func()); },
            Functions.Blank);
    });
}

// Overload:function()
```

```
// Overload:function(start)
// Overload:function(start, step)
Enumerable.ToInfinity = function (start, step)
{
    if (start == null) start = 0;
    if (step == null) step = 1;

    return new Enumerable(function ()
    {
        var value;
        return new IEnumerator(
            function () { value = start - step },
            function () { return this.Yield(value += step); },
            Functions.Blank);
    });
}

// Overload:function()
// Overload:function(start)
// Overload:function(start, step)
Enumerable.ToNegativeInfinity = function (start, step)
{
    if (start == null) start = 0;
    if (step == null) step = 1;

    return new Enumerable(function ()
    {
        var value;
        return new IEnumerator(
            function () { value = start + step },
            function () { return this.Yield(value -= step); },
            Functions.Blank);
    });
}

Enumerable.Unfold = function (seed, func)
{
    func = Utils.CreateLambda(func);

    return new Enumerable(function ()
    {
        var isFirst = true;
        var value;
        return new IEnumerator(
            Functions.Blank,
            function ()
            {
                if (isFirst)
                {
                    isFirst = false;
                    value = seed;
                    return this.Yield(value);
                }
                value = func(value);
```

```
                    return this.Yield(value);
                },
                Functions.Blank);
        });
    }

    // Extension Methods

    Enumerable.prototype =
    {
        /* Projection and Filtering Methods */

        // Overload:function(func)
        // Overload:function(func, resultSelector<element>)
        // Overload:function(func, resultSelector<element, nestLevel>)
        CascadeBreadthFirst: function (func, resultSelector)
        {
            var source = this;
            func = Utils.CreateLambda(func);
            resultSelector = Utils.CreateLambda(resultSelector);

            return new Enumerable(function ()
            {
                var enumerator;
                var nestLevel = 0;
                var buffer = [];

                return new IEnumerator(
                    function () { enumerator = source.GetEnumerator(); },
                    function ()
                    {
                        while (true)
                        {
                            if (enumerator.MoveNext())
                            {
                                buffer.push(enumerator.Current());
                                return
this.Yield(resultSelector(enumerator.Current(), nestLevel));
                            }

                            var next =
Enumerable.From(buffer).SelectMany(function (x) { return func(x); });
                            if (!next.Any())
                            {
                                return false;
                            }
                            else
                            {
                                nestLevel++;
                                buffer = [];
                                Utils.Dispose(enumerator);
                                enumerator = next.GetEnumerator();
                            }
                        }
```

```
                },
                function () { Utils.Dispose(enumerator); });
        });
    },

    // Overload:function(func)
    // Overload:function(func, resultSelector<element>)
    // Overload:function(func, resultSelector<element, nestLevel>)
    CascadeDepthFirst: function (func, resultSelector)
    {
        var source = this;
        func = Utils.CreateLambda(func);
        resultSelector = Utils.CreateLambda(resultSelector);

        return new Enumerable(function ()
        {
            var enumeratorStack = [];
            var enumerator;

            return new IEnumerator(
                function () { enumerator = source.GetEnumerator(); },
                function ()
                {
                    while (true)
                    {
                        if (enumerator.MoveNext())
                        {
                            var value =
resultSelector(enumerator.Current(), enumeratorStack.length);
                            enumeratorStack.push(enumerator);
                            enumerator =
Enumerable.From(func(enumerator.Current())).GetEnumerator();
                            return this.Yield(value);
                        }

                        if (enumeratorStack.length <= 0) return
false;
                        Utils.Dispose(enumerator);
                        enumerator = enumeratorStack.pop();
                    }
                },
                function ()
                {
                    try { Utils.Dispose(enumerator); }
                    finally {
Enumerable.From(enumeratorStack).ForEach(function (s) { s.Dispose(); }) }
                });
        });
    },

    ByHierarchy: function(firstLevel, connectBy, orderBy, ascending,
parent) {
        ascending = ascending == undefined ? true : ascending;
```

```
            var orderMethod = ascending == true ? 'OrderBy' :
'OrderByDescending';
            var source = this;
            firstLevel = Utils.CreateLambda(firstLevel);
            connectBy = Utils.CreateLambda(connectBy);
            orderBy = Utils.CreateLambda(orderBy);

            //Initiate or increase level
            var level = parent === undefined ? 1 : parent.level + 1;

            return new Enumerable(function() {
                var enumerator;
                var index = 0;

                var createLevel = function() {
                        var obj = {
                            item: enumerator.Current(),
                            level : level
                        };
                        obj.children =
Enumerable.From(source).ByHierarchy(firstLevel, connectBy, orderBy,
ascending, obj);
                        if (orderBy !== undefined) {
                            obj.children =
obj.children[orderMethod](function(d) {
                                return orderBy(d.item); //unwrap the
actual item for sort to work
                            });
                        }
                        obj.children = obj.children.ToArray();

Enumerable.From(obj.children).ForEach(function(child) {
                            child.getParent = function() {
                                return obj;
                            };
                        });
                        return obj;
                    };

                return new IEnumerator(

                function() {
                    enumerator = source.GetEnumerator();
                }, function() {
                    while (enumerator.MoveNext()) {
                        var returnArr;
                        if (!parent) {
                            if (firstLevel(enumerator.Current(),
index++)) {
                                return this.Yield(createLevel());
                            }

                        } else {
```

```
                            if (connectBy(parent.item,
enumerator.Current(), index++)) {
                                return this.Yield(createLevel());
                            }
                        }
                    }
                    return false;
                }, function() {
                    Utils.Dispose(enumerator);
                })
            });
        },

        Flatten: function ()
        {
            var source = this;

            return new Enumerable(function ()
            {
                var enumerator;
                var middleEnumerator = null;

                return new IEnumerator(
                    function () { enumerator = source.GetEnumerator(); },
                    function ()
                    {
                        while (true)
                        {
                            if (middleEnumerator != null)
                            {
                                if (middleEnumerator.MoveNext())
                                {
                                    return
this.Yield(middleEnumerator.Current());
                                }
                                else
                                {
                                    middleEnumerator = null;
                                }
                            }

                            if (enumerator.MoveNext())
                            {
                                if (enumerator.Current() instanceof
Array)
                                {
                                    Utils.Dispose(middleEnumerator);
                                    middleEnumerator =
Enumerable.From(enumerator.Current())
                                        .SelectMany(Functions.Identity)
                                        .Flatten()
                                        .GetEnumerator();
                                    continue;
                                }
```

```
                                else
                                {
                                    return
this.Yield(enumerator.Current());
                                }
                            }

                            return false;
                        }
                    },
                    function ()
                    {
                        try { Utils.Dispose(enumerator); }
                        finally { Utils.Dispose(middleEnumerator); }
                    });
            });
        },

        Pairwise: function (selector)
        {
            var source = this;
            selector = Utils.CreateLambda(selector);

            return new Enumerable(function ()
            {
                var enumerator;

                return new IEnumerator(
                    function ()
                    {
                        enumerator = source.GetEnumerator();
                        enumerator.MoveNext();
                    },
                    function ()
                    {
                        var prev = enumerator.Current();
                        return (enumerator.MoveNext())
                            ? this.Yield(selector(prev,
enumerator.Current()))
                            : false;
                    },
                    function () { Utils.Dispose(enumerator); });
            });
        },

        // Overload:function(func)
        // Overload:function(seed,func<value,element>)
        // Overload:function(seed,func<value,element>,resultSelector)
        Scan: function (seed, func, resultSelector)
        {
            if (resultSelector != null) return this.Scan(seed,
func).Select(resultSelector);

            var isUseSeed;
```

```
            if (func == null)
            {
                func = Utils.CreateLambda(seed); // arguments[0]
                isUseSeed = false;
            }
            else
            {
                func = Utils.CreateLambda(func);
                isUseSeed = true;
            }
            var source = this;

            return new Enumerable(function ()
            {
                var enumerator;
                var value;
                var isFirst = true;

                return new IEnumerator(
                    function () { enumerator = source.GetEnumerator(); },
                    function ()
                    {
                        if (isFirst)
                        {
                            isFirst = false;
                            if (!isUseSeed)
                            {
                                if (enumerator.MoveNext())
                                {
                                    return this.Yield(value =
enumerator.Current());
                                }
                            }
                            else
                            {
                                return this.Yield(value = seed);
                            }
                        }

                        return (enumerator.MoveNext())
                            ? this.Yield(value = func(value,
enumerator.Current()))
                            : false;
                    },
                    function () { Utils.Dispose(enumerator); });
            });
        },

        // Overload:function(selector<element>)
        // Overload:function(selector<element,index>)
        Select: function (selector)
        {
            var source = this;
            selector = Utils.CreateLambda(selector);
```

```
            return new Enumerable(function ()
            {
                var enumerator;
                var index = 0;

                return new IEnumerator(
                    function () { enumerator = source.GetEnumerator(); },
                    function ()
                    {
                        return (enumerator.MoveNext())
                            ? this.Yield(selector(enumerator.Current(),
index++))
                            : false;
                    },
                    function () { Utils.Dispose(enumerator); })
            });
        },

        // Overload:function(collectionSelector<element>)
        // Overload:function(collectionSelector<element,index>)
        // Overload:function(collectionSelector<element>,resultSelector)
        //
Overload:function(collectionSelector<element,index>,resultSelector)
        SelectMany: function (collectionSelector, resultSelector)
        {
            var source = this;
            collectionSelector = Utils.CreateLambda(collectionSelector);
            if (resultSelector == null) resultSelector = function (a, b)
{ return b; }
            resultSelector = Utils.CreateLambda(resultSelector);

            return new Enumerable(function ()
            {
                var enumerator;
                var middleEnumerator = undefined;
                var index = 0;

                return new IEnumerator(
                    function () { enumerator = source.GetEnumerator(); },
                    function ()
                    {
                        if (middleEnumerator === undefined)
                        {
                            if (!enumerator.MoveNext()) return false;
                        }
                        do
                        {
                            if (middleEnumerator == null)
                            {
                                var middleSeq =
collectionSelector(enumerator.Current(), index++);
                                middleEnumerator =
Enumerable.From(middleSeq).GetEnumerator();
```

```
                            }
                            if (middleEnumerator.MoveNext())
                            {
                                return
this.Yield(resultSelector(enumerator.Current(),
middleEnumerator.Current()));
                            }
                            Utils.Dispose(middleEnumerator);
                            middleEnumerator = null;
                        } while (enumerator.MoveNext())
                        return false;
                    },
                    function ()
                    {
                        try { Utils.Dispose(enumerator); }
                        finally { Utils.Dispose(middleEnumerator); }
                    })
            });
        },

        // Overload:function(predicate<element>)
        // Overload:function(predicate<element,index>)
        Where: function (predicate)
        {
            predicate = Utils.CreateLambda(predicate);
            var source = this;

            return new Enumerable(function ()
            {
                var enumerator;
                var index = 0;

                return new IEnumerator(
                    function () { enumerator = source.GetEnumerator(); },
                    function ()
                    {
                        while (enumerator.MoveNext())
                        {
                            if (predicate(enumerator.Current(), index++))
                            {
                                return this.Yield(enumerator.Current());
                            }
                        }
                        return false;
                    },
                    function () { Utils.Dispose(enumerator); })
            });
        },

        OfType: function (type)
        {
            var typeName;
            switch (type)
            {
```

```
                case Number: typeName = Types.Number; break;
                case String: typeName = Types.String; break;
                case Boolean: typeName = Types.Boolean; break;
                case Function: typeName = Types.Function; break;
                default: typeName = null; break;
            }
            return (typeName === null)
                ? this.Where(function (x) { return x instanceof type })
                : this.Where(function (x) { return typeof x === typeName
});
        },

        // Overload:function(second,selector<outer,inner>)
        // Overload:function(second,selector<outer,inner,index>)
        Zip: function (second, selector)
        {
            selector = Utils.CreateLambda(selector);
            var source = this;

            return new Enumerable(function ()
            {
                var firstEnumerator;
                var secondEnumerator;
                var index = 0;

                return new IEnumerator(
                    function ()
                    {
                        firstEnumerator = source.GetEnumerator();
                        secondEnumerator =
Enumerable.From(second).GetEnumerator();
                    },
                    function ()
                    {
                        if (firstEnumerator.MoveNext() &&
secondEnumerator.MoveNext())
                        {
                            return
this.Yield(selector(firstEnumerator.Current(),
secondEnumerator.Current(), index++));
                        }
                        return false;
                    },
                    function ()
                    {
                        try { Utils.Dispose(firstEnumerator); }
                        finally { Utils.Dispose(secondEnumerator); }
                    })
            });
        },

        /* Join Methods */
```

```
        // Overload:function (inner, outerKeySelector, innerKeySelector,
resultSelector)
        // Overload:function (inner, outerKeySelector, innerKeySelector,
resultSelector, compareSelector)
        Join: function (inner, outerKeySelector, innerKeySelector,
resultSelector, compareSelector)
        {
            outerKeySelector = Utils.CreateLambda(outerKeySelector);
            innerKeySelector = Utils.CreateLambda(innerKeySelector);
            resultSelector = Utils.CreateLambda(resultSelector);
            compareSelector = Utils.CreateLambda(compareSelector);
            var source = this;

            return new Enumerable(function ()
            {
                var outerEnumerator;
                var lookup;
                var innerElements = null;
                var innerCount = 0;

                return new IEnumerator(
                    function ()
                    {
                        outerEnumerator = source.GetEnumerator();
                        lookup =
Enumerable.From(inner).ToLookup(innerKeySelector, Functions.Identity,
compareSelector);
                    },
                    function ()
                    {
                        while (true)
                        {
                            if (innerElements != null)
                            {
                                var innerElement =
innerElements[innerCount++];
                                if (innerElement !== undefined)
                                {
                                    return
this.Yield(resultSelector(outerEnumerator.Current(), innerElement));
                                }

                                innerElement = null;
                                innerCount = 0;
                            }

                            if (outerEnumerator.MoveNext())
                            {
                                var key =
outerKeySelector(outerEnumerator.Current());
                                innerElements =
lookup.Get(key).ToArray();
                            }
                            else
```

```
                        {
                            return false;
                        }
                    }
                },
                function () { Utils.Dispose(outerEnumerator); })
            });
        },

        // Overload:function (inner, outerKeySelector, innerKeySelector,
resultSelector)
        // Overload:function (inner, outerKeySelector, innerKeySelector,
resultSelector, compareSelector)
        GroupJoin: function (inner, outerKeySelector, innerKeySelector,
resultSelector, compareSelector)
        {
            outerKeySelector = Utils.CreateLambda(outerKeySelector);
            innerKeySelector = Utils.CreateLambda(innerKeySelector);
            resultSelector = Utils.CreateLambda(resultSelector);
            compareSelector = Utils.CreateLambda(compareSelector);
            var source = this;

            return new Enumerable(function ()
            {
                var enumerator = source.GetEnumerator();
                var lookup = null;

                return new IEnumerator(
                    function ()
                    {
                        enumerator = source.GetEnumerator();
                        lookup =
Enumerable.From(inner).ToLookup(innerKeySelector, Functions.Identity,
compareSelector);
                    },
                    function ()
                    {
                        if (enumerator.MoveNext())
                        {
                            var innerElement =
lookup.Get(outerKeySelector(enumerator.Current()));
                            return
this.Yield(resultSelector(enumerator.Current(), innerElement));
                        }
                        return false;
                    },
                    function () { Utils.Dispose(enumerator); })
            });
        },

        /* Set Methods */

        All: function (predicate)
        {
```

```
            predicate = Utils.CreateLambda(predicate);

            var result = true;
            this.ForEach(function (x)
            {
                if (!predicate(x))
                {
                    result = false;
                    return false; // break
                }
            });
            return result;
        },

        // Overload:function()
        // Overload:function(predicate)
        Any: function (predicate)
        {
            predicate = Utils.CreateLambda(predicate);

            var enumerator = this.GetEnumerator();
            try
            {
                if (arguments.length == 0) return enumerator.MoveNext();
// case:function()

                while (enumerator.MoveNext()) // case:function(predicate)
                {
                    if (predicate(enumerator.Current())) return true;
                }
                return false;
            }
            finally { Utils.Dispose(enumerator); }
        },

        Concat: function (second)
        {
            var source = this;

            return new Enumerable(function ()
            {
                var firstEnumerator;
                var secondEnumerator;

                return new IEnumerator(
                    function () { firstEnumerator =
source.GetEnumerator(); },
                    function ()
                    {
                        if (secondEnumerator == null)
                        {
                            if (firstEnumerator.MoveNext()) return
this.Yield(firstEnumerator.Current());
```

```
                            secondEnumerator =
Enumerable.From(second).GetEnumerator();
                        }
                        if (secondEnumerator.MoveNext()) return
this.Yield(secondEnumerator.Current());
                        return false;
                    },
                    function ()
                    {
                        try { Utils.Dispose(firstEnumerator); }
                        finally { Utils.Dispose(secondEnumerator); }
                    })
            });
        },

        Insert: function (index, second)
        {
            var source = this;

            return new Enumerable(function ()
            {
                var firstEnumerator;
                var secondEnumerator;
                var count = 0;
                var isEnumerated = false;

                return new IEnumerator(
                    function ()
                    {
                        firstEnumerator = source.GetEnumerator();
                        secondEnumerator =
Enumerable.From(second).GetEnumerator();
                    },
                    function ()
                    {
                        if (count == index &&
secondEnumerator.MoveNext())
                        {
                            isEnumerated = true;
                            return
this.Yield(secondEnumerator.Current());
                        }
                        if (firstEnumerator.MoveNext())
                        {
                            count++;
                            return this.Yield(firstEnumerator.Current());
                        }
                        if (!isEnumerated && secondEnumerator.MoveNext())
                        {
                            return
this.Yield(secondEnumerator.Current());
                        }
                        return false;
                    },
```

```
                function ()
                {
                    try { Utils.Dispose(firstEnumerator); }
                    finally { Utils.Dispose(secondEnumerator); }
                })
            });
        },

        Alternate: function (value)
        {
            value = Enumerable.Return(value);
            return this.SelectMany(function (elem)
            {
                return Enumerable.Return(elem).Concat(value);
            }).TakeExceptLast();
        },

        // Overload:function(value)
        // Overload:function(value, compareSelector)
        Contains: function (value, compareSelector)
        {
            compareSelector = Utils.CreateLambda(compareSelector);
            var enumerator = this.GetEnumerator();
            try
            {
                while (enumerator.MoveNext())
                {
                    if (compareSelector(enumerator.Current()) === value)
return true;
                }
                return false;
            }
            finally { Utils.Dispose(enumerator) }
        },

        DefaultIfEmpty: function (defaultValue)
        {
            var source = this;

            return new Enumerable(function ()
            {
                var enumerator;
                var isFirst = true;

                return new IEnumerator(
                    function () { enumerator = source.GetEnumerator(); },
                    function ()
                    {
                        if (enumerator.MoveNext())
                        {
                            isFirst = false;
                            return this.Yield(enumerator.Current());
                        }
                        else if (isFirst)
```

```
                    {
                        isFirst = false;
                        return this.Yield(defaultValue);
                    }
                    return false;
                },
                function () { Utils.Dispose(enumerator); })
        });
    },

    // Overload:function()
    // Overload:function(compareSelector)
    Distinct: function (compareSelector)
    {
        return this.Except(Enumerable.Empty(), compareSelector);
    },

    // Overload:function(second)
    // Overload:function(second, compareSelector)
    Except: function (second, compareSelector)
    {
        compareSelector = Utils.CreateLambda(compareSelector);
        var source = this;

        return new Enumerable(function ()
        {
            var enumerator;
            var keys;

            return new IEnumerator(
                function ()
                {
                    enumerator = source.GetEnumerator();
                    keys = new Dictionary(compareSelector);
                    Enumerable.From(second).ForEach(function (key) {
keys.Add(key); });
                },
                function ()
                {
                    while (enumerator.MoveNext())
                    {
                        var current = enumerator.Current();
                        if (!keys.Contains(current))
                        {
                            keys.Add(current);
                            return this.Yield(current);
                        }
                    }
                    return false;
                },
                function () { Utils.Dispose(enumerator); })
        });
    },
```

```
        // Overload:function(second)
        // Overload:function(second, compareSelector)
        Intersect: function (second, compareSelector)
        {
            compareSelector = Utils.CreateLambda(compareSelector);
            var source = this;

            return new Enumerable(function ()
            {
                var enumerator;
                var keys;
                var outs;

                return new IEnumerator(
                    function ()
                    {
                        enumerator = source.GetEnumerator();

                        keys = new Dictionary(compareSelector);
                        Enumerable.From(second).ForEach(function (key) {
keys.Add(key); });
                        outs = new Dictionary(compareSelector);
                    },
                    function ()
                    {
                        while (enumerator.MoveNext())
                        {
                            var current = enumerator.Current();
                            if (!outs.Contains(current) &&
keys.Contains(current))
                            {
                                outs.Add(current);
                                return this.Yield(current);
                            }
                        }
                        return false;
                    },
                    function () { Utils.Dispose(enumerator); })
            });
        },

        // Overload:function(second)
        // Overload:function(second, compareSelector)
        SequenceEqual: function (second, compareSelector)
        {
            compareSelector = Utils.CreateLambda(compareSelector);

            var firstEnumerator = this.GetEnumerator();
            try
            {
                var secondEnumerator =
Enumerable.From(second).GetEnumerator();
                try
                {
```

```
                    while (firstEnumerator.MoveNext())
                    {
                        if (!secondEnumerator.MoveNext()
                            || compareSelector(firstEnumerator.Current())
!== compareSelector(secondEnumerator.Current()))
                        {
                            return false;
                        }
                    }

                    if (secondEnumerator.MoveNext()) return false;
                    return true;
                }
                finally { Utils.Dispose(secondEnumerator); }
            }
            finally { Utils.Dispose(firstEnumerator); }
        },

        Union: function (second, compareSelector)
        {
            compareSelector = Utils.CreateLambda(compareSelector);
            var source = this;

            return new Enumerable(function ()
            {
                var firstEnumerator;
                var secondEnumerator;
                var keys;

                return new IEnumerator(
                    function ()
                    {
                        firstEnumerator = source.GetEnumerator();
                        keys = new Dictionary(compareSelector);
                    },
                    function ()
                    {
                        var current;
                        if (secondEnumerator === undefined)
                        {
                            while (firstEnumerator.MoveNext())
                            {
                                current = firstEnumerator.Current();
                                if (!keys.Contains(current))
                                {
                                    keys.Add(current);
                                    return this.Yield(current);
                                }
                            }
                            secondEnumerator =
Enumerable.From(second).GetEnumerator();
                        }
                        while (secondEnumerator.MoveNext())
                        {
```

```
                    current = secondEnumerator.Current();
                    if (!keys.Contains(current))
                    {
                        keys.Add(current);
                        return this.Yield(current);
                    }
                }
                return false;
            },
            function ()
            {
                try { Utils.Dispose(firstEnumerator); }
                finally { Utils.Dispose(secondEnumerator); }
            })
    });
},

/* Ordering Methods */

OrderBy: function (keySelector)
{
    return new OrderedEnumerable(this, keySelector, false);
},

OrderByDescending: function (keySelector)
{
    return new OrderedEnumerable(this, keySelector, true);
},

Reverse: function ()
{
    var source = this;

    return new Enumerable(function ()
    {
        var buffer;
        var index;

        return new IEnumerator(
            function ()
            {
                buffer = source.ToArray();
                index = buffer.length;
            },
            function ()
            {
                return (index > 0)
                    ? this.Yield(buffer[--index])
                    : false;
            },
            Functions.Blank)
    });
},
```

```
        Shuffle: function ()
        {
            var source = this;

            return new Enumerable(function ()
            {
                var buffer;

                return new IEnumerator(
                    function () { buffer = source.ToArray(); },
                    function ()
                    {
                        if (buffer.length > 0)
                        {
                            var i = Math.floor(Math.random() *
buffer.length);
                            return this.Yield(buffer.splice(i, 1)[0]);
                        }
                        return false;
                    },
                    Functions.Blank)
            });
        },

        /* Grouping Methods */

        // Overload:function(keySelector)
        // Overload:function(keySelector,elementSelector)
        // Overload:function(keySelector,elementSelector,resultSelector)
        //
Overload:function(keySelector,elementSelector,resultSelector,compareSelec
tor)
        GroupBy: function (keySelector, elementSelector, resultSelector,
compareSelector)
        {
            var source = this;
            keySelector = Utils.CreateLambda(keySelector);
            elementSelector = Utils.CreateLambda(elementSelector);
            if (resultSelector != null) resultSelector =
Utils.CreateLambda(resultSelector);
            compareSelector = Utils.CreateLambda(compareSelector);

            return new Enumerable(function ()
            {
                var enumerator;

                return new IEnumerator(
                    function ()
                    {
                        enumerator = source.ToLookup(keySelector,
elementSelector, compareSelector)
                            .ToEnumerable()
                            .GetEnumerator();
                    },
```

```
                    function ()
                    {
                        while (enumerator.MoveNext())
                        {
                            return (resultSelector == null)
                                ? this.Yield(enumerator.Current())
                                :
this.Yield(resultSelector(enumerator.Current().Key(),
enumerator.Current()));
                        }
                        return false;
                    },
                    function () { Utils.Dispose(enumerator); })
            });
        },

        // Overload:function(keySelector)
        // Overload:function(keySelector,elementSelector)
        // Overload:function(keySelector,elementSelector,resultSelector)
        //
Overload:function(keySelector,elementSelector,resultSelector,compareSelec
tor)
        PartitionBy: function (keySelector, elementSelector,
resultSelector, compareSelector)
        {

            var source = this;
            keySelector = Utils.CreateLambda(keySelector);
            elementSelector = Utils.CreateLambda(elementSelector);
            compareSelector = Utils.CreateLambda(compareSelector);
            var hasResultSelector;
            if (resultSelector == null)
            {
                hasResultSelector = false;
                resultSelector = function (key, group) { return new
Grouping(key, group) }
            }
            else
            {
                hasResultSelector = true;
                resultSelector = Utils.CreateLambda(resultSelector);
            }

            return new Enumerable(function ()
            {
                var enumerator;
                var key;
                var compareKey;
                var group = [];

                return new IEnumerator(
                    function ()
                    {
                        enumerator = source.GetEnumerator();
```

```
                        if (enumerator.MoveNext())
                        {
                            key = keySelector(enumerator.Current());
                            compareKey = compareSelector(key);

group.push(elementSelector(enumerator.Current()));
                        }
                    },
                    function ()
                    {
                        var hasNext;
                        while ((hasNext = enumerator.MoveNext()) == true)
                        {
                            if (compareKey ===
compareSelector(keySelector(enumerator.Current())))
                            {

group.push(elementSelector(enumerator.Current()));
                            }
                            else break;
                        }

                        if (group.length > 0)
                        {
                            var result = (hasResultSelector)
                                ? resultSelector(key,
Enumerable.From(group))
                                : resultSelector(key, group);
                            if (hasNext)
                            {
                                key = keySelector(enumerator.Current());
                                compareKey = compareSelector(key);
                                group =
[elementSelector(enumerator.Current())];
                            }
                            else group = [];

                            return this.Yield(result);
                        }

                        return false;
                    },
                    function () { Utils.Dispose(enumerator); })
            });
        },

        BufferWithCount: function (count)
        {
            var source = this;

            return new Enumerable(function ()
            {
                var enumerator;
```

```
                return new IEnumerator(
                function () { enumerator = source.GetEnumerator(); },
                function ()
                {
                    var array = [];
                    var index = 0;
                    while (enumerator.MoveNext())
                    {
                        array.push(enumerator.Current());
                        if (++index >= count) return this.Yield(array);
                    }
                    if (array.length > 0) return this.Yield(array);
                    return false;
                },
                function () { Utils.Dispose(enumerator); })
            });
        },

        /* Aggregate Methods */

        // Overload:function(func)
        // Overload:function(seed,func)
        // Overload:function(seed,func,resultSelector)
        Aggregate: function (seed, func, resultSelector)
        {
            return this.Scan(seed, func, resultSelector).Last();
        },

        // Overload:function()
        // Overload:function(selector)
        Average: function (selector)
        {
            selector = Utils.CreateLambda(selector);

            var sum = 0;
            var count = 0;
            this.ForEach(function (x)
            {
                sum += selector(x);
                ++count;
            });

            return sum / count;
        },

        // Overload:function()
        // Overload:function(predicate)
        Count: function (predicate)
        {
            predicate = (predicate == null) ? Functions.True :
Utils.CreateLambda(predicate);

            var count = 0;
            this.ForEach(function (x, i)
```

```
            {
                if (predicate(x, i)) ++count;
            });
            return count;
        },

        // Overload:function()
        // Overload:function(selector)
        Max: function (selector)
        {
            if (selector == null) selector = Functions.Identity;
            return this.Select(selector).Aggregate(function (a, b) {
return (a > b) ? a : b; });
        },

        // Overload:function()
        // Overload:function(selector)
        Min: function (selector)
        {
            if (selector == null) selector = Functions.Identity;
            return this.Select(selector).Aggregate(function (a, b) {
return (a < b) ? a : b; });
        },

        MaxBy: function (keySelector)
        {
            keySelector = Utils.CreateLambda(keySelector);
            return this.Aggregate(function (a, b) { return
(keySelector(a) > keySelector(b)) ? a : b });
        },

        MinBy: function (keySelector)
        {
            keySelector = Utils.CreateLambda(keySelector);
            return this.Aggregate(function (a, b) { return
(keySelector(a) < keySelector(b)) ? a : b });
        },

        // Overload:function()
        // Overload:function(selector)
        Sum: function (selector)
        {
            if (selector == null) selector = Functions.Identity;
            return this.Select(selector).Aggregate(0, function (a, b) {
return a + b; });
        },

        /* Paging Methods */

        ElementAt: function (index)
        {
            var value;
            var found = false;
            this.ForEach(function (x, i)
```

```
            {
                if (i == index)
                {
                    value = x;
                    found = true;
                    return false;
                }
            });

            if (!found) throw new Error("index is less than 0 or greater
than or equal to the number of elements in source.");
            return value;
        },

        ElementAtOrDefault: function (index, defaultValue)
        {
            var value;
            var found = false;
            this.ForEach(function (x, i)
            {
                if (i == index)
                {
                    value = x;
                    found = true;
                    return false;
                }
            });

            return (!found) ? defaultValue : value;
        },

        // Overload:function()
        // Overload:function(predicate)
        First: function (predicate)
        {
            if (predicate != null) return this.Where(predicate).First();

            var value;
            var found = false;
            this.ForEach(function (x)
            {
                value = x;
                found = true;
                return false;
            });

            if (!found) throw new Error("First:No element satisfies the
condition.");
            return value;
        },

        // Overload:function(defaultValue)
        // Overload:function(defaultValue,predicate)
        FirstOrDefault: function (defaultValue, predicate)
```

```
        {
            if (predicate != null) return
this.Where(predicate).FirstOrDefault(defaultValue);

            var value;
            var found = false;
            this.ForEach(function (x)
            {
                value = x;
                found = true;
                return false;
            });
            return (!found) ? defaultValue : value;
        },

        // Overload:function()
        // Overload:function(predicate)
        Last: function (predicate)
        {
            if (predicate != null) return this.Where(predicate).Last();

            var value;
            var found = false;
            this.ForEach(function (x)
            {
                found = true;
                value = x;
            });

            if (!found) throw new Error("Last:No element satisfies the
condition.");
            return value;
        },

        // Overload:function(defaultValue)
        // Overload:function(defaultValue,predicate)
        LastOrDefault: function (defaultValue, predicate)
        {
            if (predicate != null) return
this.Where(predicate).LastOrDefault(defaultValue);

            var value;
            var found = false;
            this.ForEach(function (x)
            {
                found = true;
                value = x;
            });
            return (!found) ? defaultValue : value;
        },

        // Overload:function()
        // Overload:function(predicate)
        Single: function (predicate)
```

```
        {
            if (predicate != null) return this.Where(predicate).Single();

            var value;
            var found = false;
            this.ForEach(function (x)
            {
                if (!found)
                {
                    found = true;
                    value = x;
                }
                else throw new Error("Single:sequence contains more than
one element.");
            });

            if (!found) throw new Error("Single:No element satisfies the
condition.");
            return value;
        },

        // Overload:function(defaultValue)
        // Overload:function(defaultValue,predicate)
        SingleOrDefault: function (defaultValue, predicate)
        {
            if (predicate != null) return
this.Where(predicate).SingleOrDefault(defaultValue);

            var value;
            var found = false;
            this.ForEach(function (x)
            {
                if (!found)
                {
                    found = true;
                    value = x;
                }
                else throw new Error("Single:sequence contains more than
one element.");
            });

            return (!found) ? defaultValue : value;
        },

        Skip: function (count)
        {
            var source = this;

            return new Enumerable(function ()
            {
                var enumerator;
                var index = 0;

                return new IEnumerator(
```

```
                    function ()
                    {
                        enumerator = source.GetEnumerator();
                        while (index++ < count && enumerator.MoveNext())
{ };
                    },
                    function ()
                    {
                        return (enumerator.MoveNext())
                            ? this.Yield(enumerator.Current())
                            : false;
                    },
                    function () { Utils.Dispose(enumerator); })
            });
        },

        // Overload:function(predicate<element>)
        // Overload:function(predicate<element,index>)
        SkipWhile: function (predicate)
        {
            predicate = Utils.CreateLambda(predicate);
            var source = this;

            return new Enumerable(function ()
            {
                var enumerator;
                var index = 0;
                var isSkipEnd = false;

                return new IEnumerator(
                    function () { enumerator = source.GetEnumerator(); },
                    function ()
                    {
                        while (!isSkipEnd)
                        {
                            if (enumerator.MoveNext())
                            {
                                if (!predicate(enumerator.Current(),
index++))
                                {
                                    isSkipEnd = true;
                                    return
this.Yield(enumerator.Current());
                                }
                                continue;
                            }
                            else return false;
                        }

                        return (enumerator.MoveNext())
                            ? this.Yield(enumerator.Current())
                            : false;

                    },
```

```
                    function () { Utils.Dispose(enumerator); });
            });
        },

        Take: function (count)
        {
            var source = this;

            return new Enumerable(function ()
            {
                var enumerator;
                var index = 0;

                return new IEnumerator(
                    function () { enumerator = source.GetEnumerator(); },
                    function ()
                    {
                        return (index++ < count && enumerator.MoveNext())
                            ? this.Yield(enumerator.Current())
                            : false;
                    },
                    function () { Utils.Dispose(enumerator); }
                )
            });
        },

        // Overload:function(predicate<element>)
        // Overload:function(predicate<element,index>)
        TakeWhile: function (predicate)
        {
            predicate = Utils.CreateLambda(predicate);
            var source = this;

            return new Enumerable(function ()
            {
                var enumerator;
                var index = 0;

                return new IEnumerator(
                    function () { enumerator = source.GetEnumerator(); },
                    function ()
                    {
                        return (enumerator.MoveNext() &&
predicate(enumerator.Current(), index++))
                            ? this.Yield(enumerator.Current())
                            : false;
                    },
                    function () { Utils.Dispose(enumerator); });
            });
        },

        // Overload:function()
        // Overload:function(count)
        TakeExceptLast: function (count)
```

```
        {
            if (count == null) count = 1;
            var source = this;

            return new Enumerable(function ()
            {
                if (count <= 0) return source.GetEnumerator(); // do
nothing

                var enumerator;
                var q = [];

                return new IEnumerator(
                    function () { enumerator = source.GetEnumerator(); },
                    function ()
                    {
                        while (enumerator.MoveNext())
                        {
                            if (q.length == count)
                            {
                                q.push(enumerator.Current());
                                return this.Yield(q.shift());
                            }
                            q.push(enumerator.Current());
                        }
                        return false;
                    },
                    function () { Utils.Dispose(enumerator); });
            });
        },

        TakeFromLast: function (count)
        {
            if (count <= 0 || count == null) return Enumerable.Empty();
            var source = this;

            return new Enumerable(function ()
            {
                var sourceEnumerator;
                var enumerator;
                var q = [];

                return new IEnumerator(
                    function () { sourceEnumerator =
source.GetEnumerator(); },
                    function ()
                    {
                        while (sourceEnumerator.MoveNext())
                        {
                            if (q.length == count) q.shift()
                            q.push(sourceEnumerator.Current());
                        }
                        if (enumerator == null)
                        {
```

```
                            enumerator =
Enumerable.From(q).GetEnumerator();
                        }
                        return (enumerator.MoveNext())
                            ? this.Yield(enumerator.Current())
                            : false;
                    },
                    function () { Utils.Dispose(enumerator); });
            });
        },

        IndexOf: function (item)
        {
            var found = null;
            this.ForEach(function (x, i)
            {
                if (x === item)
                {
                    found = i;
                    return true;
                }
            });

            return (found !== null) ? found : -1;
        },

        LastIndexOf: function (item)
        {
            var result = -1;
            this.ForEach(function (x, i)
            {
                if (x === item) result = i;
            });

            return result;
        },

        /* Convert Methods */

        ToArray: function ()
        {
            var array = [];
            this.ForEach(function (x) { array.push(x) });
            return array;
        },

        // Overload:function(keySelector)
        // Overload:function(keySelector, elementSelector)
        // Overload:function(keySelector, elementSelector,
compareSelector)
        ToLookup: function (keySelector, elementSelector,
compareSelector)
        {
            keySelector = Utils.CreateLambda(keySelector);
```

```
            elementSelector = Utils.CreateLambda(elementSelector);
            compareSelector = Utils.CreateLambda(compareSelector);

            var dict = new Dictionary(compareSelector);
            this.ForEach(function (x)
            {
                var key = keySelector(x);
                var element = elementSelector(x);

                var array = dict.Get(key);
                if (array !== undefined) array.push(element);
                else dict.Add(key, [element]);
            });
            return new Lookup(dict);
        },

        ToObject: function (keySelector, elementSelector)
        {
            keySelector = Utils.CreateLambda(keySelector);
            elementSelector = Utils.CreateLambda(elementSelector);

            var obj = {};
            this.ForEach(function (x)
            {
                obj[keySelector(x)] = elementSelector(x);
            });
            return obj;
        },

        // Overload:function(keySelector, elementSelector)
        // Overload:function(keySelector, elementSelector,
compareSelector)
        ToDictionary: function (keySelector, elementSelector,
compareSelector)
        {
            keySelector = Utils.CreateLambda(keySelector);
            elementSelector = Utils.CreateLambda(elementSelector);
            compareSelector = Utils.CreateLambda(compareSelector);

            var dict = new Dictionary(compareSelector);
            this.ForEach(function (x)
            {
                dict.Add(keySelector(x), elementSelector(x));
            });
            return dict;
        },

        // Overload:function()
        // Overload:function(replacer)
        // Overload:function(replacer, space)
        ToJSON: function (replacer, space)
        {
            return JSON.stringify(this.ToArray(), replacer, space);
        },
```

```
// Overload:function()
// Overload:function(separator)
// Overload:function(separator,selector)
ToString: function (separator, selector)
{
    if (separator == null) separator = "";
    if (selector == null) selector = Functions.Identity;

    return this.Select(selector).ToArray().join(separator);
},


/* Action Methods */

// Overload:function(action<element>)
// Overload:function(action<element,index>)
Do: function (action)
{
    var source = this;
    action = Utils.CreateLambda(action);

    return new Enumerable(function ()
    {
        var enumerator;
        var index = 0;

        return new IEnumerator(
            function () { enumerator = source.GetEnumerator(); },
            function ()
            {
                if (enumerator.MoveNext())
                {
                    action(enumerator.Current(), index++);
                    return this.Yield(enumerator.Current());
                }
                return false;
            },
            function () { Utils.Dispose(enumerator); });
    });
},

// Overload:function(action<element>)
// Overload:function(action<element,index>)
// Overload:function(func<element,bool>)
// Overload:function(func<element,index,bool>)
ForEach: function (action)
{
    action = Utils.CreateLambda(action);

    var index = 0;
    var enumerator = this.GetEnumerator();
    try
    {
```

```
                while (enumerator.MoveNext())
                {
                    if (action(enumerator.Current(), index++) === false)
break;
                }
            }
            finally { Utils.Dispose(enumerator); }
        },

        // Overload:function()
        // Overload:function(separator)
        // Overload:function(separator,selector)
        Write: function (separator, selector)
        {
            if (separator == null) separator = "";
            selector = Utils.CreateLambda(selector);

            var isFirst = true;
            this.ForEach(function (item)
            {
                if (isFirst) isFirst = false;
                else document.write(separator);
                document.write(selector(item));
            });
        },

        // Overload:function()
        // Overload:function(selector)
        WriteLine: function (selector)
        {
            selector = Utils.CreateLambda(selector);

            this.ForEach(function (item)
            {
                document.write(selector(item));
                document.write("<br />");
            });
        },

        Force: function ()
        {
            var enumerator = this.GetEnumerator();

            try { while (enumerator.MoveNext()) { } }
            finally { Utils.Dispose(enumerator); }
        },

        /* Functional Methods */

        Let: function (func)
        {
            func = Utils.CreateLambda(func);
            var source = this;
```

```
            return new Enumerable(function ()
            {
                var enumerator;

                return new IEnumerator(
                    function ()
                    {
                        enumerator =
Enumerable.From(func(source)).GetEnumerator();
                    },
                    function ()
                    {
                        return (enumerator.MoveNext())
                            ? this.Yield(enumerator.Current())
                            : false;
                    },
                    function () { Utils.Dispose(enumerator); })
            });
        },

        Share: function ()
        {
            var source = this;
            var sharedEnumerator;

            return new Enumerable(function ()
            {
                return new IEnumerator(
                    function ()
                    {
                        if (sharedEnumerator == null)
                        {
                            sharedEnumerator = source.GetEnumerator();
                        }
                    },
                    function ()
                    {
                        return (sharedEnumerator.MoveNext())
                            ? this.Yield(sharedEnumerator.Current())
                            : false;
                    },
                    Functions.Blank
                )
            });
        },

        MemoizeAll: function ()
        {
            var source = this;
            var cache;
            var enumerator;

            return new Enumerable(function ()
            {
```

```
                var index = -1;

                return new IEnumerator(
                    function ()
                    {
                        if (enumerator == null)
                        {
                            enumerator = source.GetEnumerator();
                            cache = [];
                        }
                    },
                    function ()
                    {
                        index++;
                        if (cache.length <= index)
                        {
                            return (enumerator.MoveNext())
                                ? this.Yield(cache[index] =
enumerator.Current())
                                : false;
                        }

                        return this.Yield(cache[index]);
                    },
                    Functions.Blank
                )
            });
        },

        /* Error Handling Methods */

        Catch: function (handler)
        {
            handler = Utils.CreateLambda(handler);
            var source = this;

            return new Enumerable(function ()
            {
                var enumerator;

                return new IEnumerator(
                    function () { enumerator = source.GetEnumerator(); },
                    function ()
                    {
                        try
                        {
                            return (enumerator.MoveNext())
                               ? this.Yield(enumerator.Current())
                               : false;
                        }
                        catch (e)
                        {
                            handler(e);
                            return false;
```

```
                    }
                },
                function () { Utils.Dispose(enumerator); });
        });
    },

    Finally: function (finallyAction)
    {
        finallyAction = Utils.CreateLambda(finallyAction);
        var source = this;

        return new Enumerable(function ()
        {
            var enumerator;

            return new IEnumerator(
                function () { enumerator = source.GetEnumerator(); },
                function ()
                {
                    return (enumerator.MoveNext())
                       ? this.Yield(enumerator.Current())
                       : false;
                },
                function ()
                {
                    try { Utils.Dispose(enumerator); }
                    finally { finallyAction(); }
                });
        });
    },

    /* For Debug Methods */

    // Overload:function()
    // Overload:function(message)
    // Overload:function(message,selector)
    Trace: function (message, selector)
    {
        if (message == null) message = "Trace";
        selector = Utils.CreateLambda(selector);

        return this.Do(function (item)
        {
            console.log(message, ":", selector(item));
        });
    }
}

// private

// static functions
var Functions =
{
    Identity: function (x) { return x; },
```

```
        True: function () { return true; },
        Blank: function () { }
    }

    // static const
    var Types =
    {
        Boolean: typeof true,
        Number: typeof 0,
        String: typeof "",
        Object: typeof {},
        Undefined: typeof undefined,
        Function: typeof function () { }
    }

    // static utility methods
    var Utils =
    {
        // Create anonymous function from lambda expression string
        CreateLambda: function (expression)
        {
            if (expression == null) return Functions.Identity;
            if (typeof expression == Types.String)
            {
                if (expression == "")
                {
                    return Functions.Identity;
                }
                else if (expression.indexOf("=>") == -1)
                {
                    return new Function("$,$$,$$$,$$$$", "return " +
expression);
                }
                else
                {
                    var expr =
expression.match(/^[(\s]*([^()]*?)[)\s]*=>(.*)/);
                    return new Function(expr[1], "return " + expr[2]);
                }
            }
            return expression;
        },

        IsIEnumerable: function (obj)
        {
            if (typeof Enumerator != Types.Undefined)
            {
                try
                {
                    new Enumerator(obj);
                    return true;
                }
                catch (e) { }
            }
```

```
        return false;
    },

    Compare: function (a, b)
    {
        return (a === b) ? 0
            : (a > b) ? 1
            : -1;
    },

    Dispose: function (obj)
    {
        if (obj != null) obj.Dispose();
    }
}

// IEnumerator State
var State = { Before: 0, Running: 1, After: 2 }

// name "Enumerator" is conflict JScript's "Enumerator"
var IEnumerator = function (initialize, tryGetNext, dispose)
{
    var yielder = new Yielder();
    var state = State.Before;

    this.Current = yielder.Current;
    this.MoveNext = function ()
    {
        try
        {
            switch (state)
            {
                case State.Before:
                    state = State.Running;
                    initialize(); // fall through
                case State.Running:
                    if (tryGetNext.apply(yielder))
                    {
                        return true;
                    }
                    else
                    {
                        this.Dispose();
                        return false;
                    }
                case State.After:
                    return false;
            }
        }
        catch (e)
        {
            this.Dispose();
            throw e;
        }
```

```
        }
        this.Dispose = function ()
        {
            if (state != State.Running) return;

            try { dispose(); }
            finally { state = State.After; }
        }
    }

    // for tryGetNext
    var Yielder = function ()
    {
        var current = null;
        this.Current = function () { return current; }
        this.Yield = function (value)
        {
            current = value;
            return true;
        }
    }

    // for OrderBy/ThenBy

    var OrderedEnumerable = function (source, keySelector, descending,
parent)
    {
        this.source = source;
        this.keySelector = Utils.CreateLambda(keySelector);
        this.descending = descending;
        this.parent = parent;
    }
    OrderedEnumerable.prototype = new Enumerable();

    OrderedEnumerable.prototype.CreateOrderedEnumerable = function
(keySelector, descending)
    {
        return new OrderedEnumerable(this.source, keySelector,
descending, this);
    }

    OrderedEnumerable.prototype.ThenBy = function (keySelector)
    {
        return this.CreateOrderedEnumerable(keySelector, false);
    }

    OrderedEnumerable.prototype.ThenByDescending = function (keySelector)
    {
        return this.CreateOrderedEnumerable(keySelector, true);
    }

    OrderedEnumerable.prototype.GetEnumerator = function ()
    {
        var self = this;
```

```
        var buffer;
        var indexes;
        var index = 0;

        return new IEnumerator(
            function ()
            {
                buffer = [];
                indexes = [];
                self.source.ForEach(function (item, index)
                {
                    buffer.push(item);
                    indexes.push(index);
                });
                var sortContext = SortContext.Create(self, null);
                sortContext.GenerateKeys(buffer);

                indexes.sort(function (a, b) { return
sortContext.Compare(a, b); });
            },
            function ()
            {
                return (index < indexes.length)
                    ? this.Yield(buffer[indexes[index++]])
                    : false;
            },
            Functions.Blank
        )
    }

    var SortContext = function (keySelector, descending, child)
    {
        this.keySelector = keySelector;
        this.descending = descending;
        this.child = child;
        this.keys = null;
    }

    SortContext.Create = function (orderedEnumerable, currentContext)
    {
        var context = new SortContext(orderedEnumerable.keySelector,
orderedEnumerable.descending, currentContext);
        if (orderedEnumerable.parent != null) return
SortContext.Create(orderedEnumerable.parent, context);
        return context;
    }

    SortContext.prototype.GenerateKeys = function (source)
    {
        var len = source.length;
        var keySelector = this.keySelector;
        var keys = new Array(len);
        for (var i = 0; i < len; i++) keys[i] = keySelector(source[i]);
        this.keys = keys;
```

```
        if (this.child != null) this.child.GenerateKeys(source);
    }

    SortContext.prototype.Compare = function (index1, index2)
    {
        var comparison = Utils.Compare(this.keys[index1],
this.keys[index2]);

        if (comparison == 0)
        {
            if (this.child != null) return this.child.Compare(index1,
index2)
            comparison = Utils.Compare(index1, index2);
        }

        return (this.descending) ? -comparison : comparison;
    }

    // optimize array or arraylike object

    var ArrayEnumerable = function (source)
    {
        this.source = source;
    }
    ArrayEnumerable.prototype = new Enumerable();

    ArrayEnumerable.prototype.Any = function (predicate)
    {
        return (predicate == null)
            ? (this.source.length > 0)
            : Enumerable.prototype.Any.apply(this, arguments);
    }

    ArrayEnumerable.prototype.Count = function (predicate)
    {
        return (predicate == null)
            ? this.source.length
            : Enumerable.prototype.Count.apply(this, arguments);
    }

    ArrayEnumerable.prototype.ElementAt = function (index)
    {
        return (0 <= index && index < this.source.length)
            ? this.source[index]
            : Enumerable.prototype.ElementAt.apply(this, arguments);
    }

    ArrayEnumerable.prototype.ElementAtOrDefault = function (index,
defaultValue)
    {
        return (0 <= index && index < this.source.length)
            ? this.source[index]
            : defaultValue;
```

```
    }

    ArrayEnumerable.prototype.First = function (predicate)
    {
        return (predicate == null && this.source.length > 0)
            ? this.source[0]
            : Enumerable.prototype.First.apply(this, arguments);
    }

    ArrayEnumerable.prototype.FirstOrDefault = function (defaultValue,
predicate)
    {
        if (predicate != null)
        {
            return Enumerable.prototype.FirstOrDefault.apply(this,
arguments);
        }

        return this.source.length > 0 ? this.source[0] : defaultValue;
    }

    ArrayEnumerable.prototype.Last = function (predicate)
    {
        return (predicate == null && this.source.length > 0)
            ? this.source[this.source.length - 1]
            : Enumerable.prototype.Last.apply(this, arguments);
    }

    ArrayEnumerable.prototype.LastOrDefault = function (defaultValue,
predicate)
    {
        if (predicate != null)
        {
            return Enumerable.prototype.LastOrDefault.apply(this,
arguments);
        }

        return this.source.length > 0 ? this.source[this.source.length -
1] : defaultValue;
    }

    ArrayEnumerable.prototype.Skip = function (count)
    {
        var source = this.source;

        return new Enumerable(function ()
        {
            var index;

            return new IEnumerator(
                function () { index = (count < 0) ? 0 : count },
                function ()
                {
                    return (index < source.length)
```

```
                    ? this.Yield(source[index++])
                    : false;
            },
            Functions.Blank);
    });
};

ArrayEnumerable.prototype.TakeExceptLast = function (count)
{
    if (count == null) count = 1;
    return this.Take(this.source.length - count);
}

ArrayEnumerable.prototype.TakeFromLast = function (count)
{
    return this.Skip(this.source.length - count);
}

ArrayEnumerable.prototype.Reverse = function ()
{
    var source = this.source;

    return new Enumerable(function ()
    {
        var index;

        return new IEnumerator(
            function ()
            {
                index = source.length;
            },
            function ()
            {
                return (index > 0)
                    ? this.Yield(source[--index])
                    : false;
            },
            Functions.Blank)
    });
}

ArrayEnumerable.prototype.SequenceEqual = function (second,
compareSelector)
{
    if ((second instanceof ArrayEnumerable || second instanceof
Array)
        && compareSelector == null
        && Enumerable.From(second).Count() != this.Count())
    {
        return false;
    }

    return Enumerable.prototype.SequenceEqual.apply(this, arguments);
}
```

```
ArrayEnumerable.prototype.ToString = function (separator, selector)
{
    if (selector != null || !(this.source instanceof Array))
    {
        return Enumerable.prototype.ToString.apply(this, arguments);
    }

    if (separator == null) separator = "";
    return this.source.join(separator);
}

ArrayEnumerable.prototype.GetEnumerator = function ()
{
    var source = this.source;
    var index = 0;

    return new IEnumerator(
        Functions.Blank,
        function ()
        {
            return (index < source.length)
                ? this.Yield(source[index++])
                : false;
        },
        Functions.Blank);
}

// Collections

var Dictionary = (function ()
{
    // static utility methods
    var HasOwnProperty = function (target, key)
    {
        return Object.prototype.hasOwnProperty.call(target, key);
    }

    var ComputeHashCode = function (obj)
    {
        if (obj === null) return "null";
        if (obj === undefined) return "undefined";

        return (typeof obj.toString === Types.Function)
            ? obj.toString()
            : Object.prototype.toString.call(obj);
    }

    // LinkedList for Dictionary
    var HashEntry = function (key, value)
    {
        this.Key = key;
        this.Value = value;
        this.Prev = null;
```

```
    this.Next = null;
}

var EntryList = function ()
{
    this.First = null;
    this.Last = null;
}
EntryList.prototype =
{
    AddLast: function (entry)
    {
        if (this.Last != null)
        {
            this.Last.Next = entry;
            entry.Prev = this.Last;
            this.Last = entry;
        }
        else this.First = this.Last = entry;
    },

    Replace: function (entry, newEntry)
    {
        if (entry.Prev != null)
        {
            entry.Prev.Next = newEntry;
            newEntry.Prev = entry.Prev;
        }
        else this.First = newEntry;

        if (entry.Next != null)
        {
            entry.Next.Prev = newEntry;
            newEntry.Next = entry.Next;
        }
        else this.Last = newEntry;

    },

    Remove: function (entry)
    {
        if (entry.Prev != null) entry.Prev.Next = entry.Next;
        else this.First = entry.Next;

        if (entry.Next != null) entry.Next.Prev = entry.Prev;
        else this.Last = entry.Prev;
    }
}

// Overload:function()
// Overload:function(compareSelector)
var Dictionary = function (compareSelector)
{
    this.count = 0;
```

```
            this.entryList = new EntryList();
            this.buckets = {}; // as Dictionary<string,List<object>>
            this.compareSelector = (compareSelector == null) ?
Functions.Identity : compareSelector;
        }

        Dictionary.prototype =
        {
            Add: function (key, value)
            {
                var compareKey = this.compareSelector(key);
                var hash = ComputeHashCode(compareKey);
                var entry = new HashEntry(key, value);
                if (HasOwnProperty(this.buckets, hash))
                {
                    var array = this.buckets[hash];
                    for (var i = 0; i < array.length; i++)
                    {
                        if (this.compareSelector(array[i].Key) ===
compareKey)
                        {
                            this.entryList.Replace(array[i], entry);
                            array[i] = entry;
                            return;
                        }
                    }
                    array.push(entry);
                }
                else
                {
                    this.buckets[hash] = [entry];
                }
                this.count++;
                this.entryList.AddLast(entry);
            },

            Get: function (key)
            {
                var compareKey = this.compareSelector(key);
                var hash = ComputeHashCode(compareKey);
                if (!HasOwnProperty(this.buckets, hash)) return
undefined;

                var array = this.buckets[hash];
                for (var i = 0; i < array.length; i++)
                {
                    var entry = array[i];
                    if (this.compareSelector(entry.Key) === compareKey)
return entry.Value;
                }
                return undefined;
            },

            Set: function (key, value)
```

```
            {
                var compareKey = this.compareSelector(key);
                var hash = ComputeHashCode(compareKey);
                if (HasOwnProperty(this.buckets, hash))
                {
                    var array = this.buckets[hash];
                    for (var i = 0; i < array.length; i++)
                    {
                        if (this.compareSelector(array[i].Key) ===
compareKey)
                        {
                            var newEntry = new HashEntry(key, value);
                            this.entryList.Replace(array[i], newEntry);
                            array[i] = newEntry;
                            return true;
                        }
                    }
                }
                return false;
            },

            Contains: function (key)
            {
                var compareKey = this.compareSelector(key);
                var hash = ComputeHashCode(compareKey);
                if (!HasOwnProperty(this.buckets, hash)) return false;

                var array = this.buckets[hash];
                for (var i = 0; i < array.length; i++)
                {
                    if (this.compareSelector(array[i].Key) ===
compareKey) return true;
                }
                return false;
            },

            Clear: function ()
            {
                this.count = 0;
                this.buckets = {};
                this.entryList = new EntryList();
            },

            Remove: function (key)
            {
                var compareKey = this.compareSelector(key);
                var hash = ComputeHashCode(compareKey);
                if (!HasOwnProperty(this.buckets, hash)) return;

                var array = this.buckets[hash];
                for (var i = 0; i < array.length; i++)
                {
                    if (this.compareSelector(array[i].Key) ===
compareKey)
```

```
                    {
                        this.entryList.Remove(array[i]);
                        array.splice(i, 1);
                        if (array.length == 0) delete this.buckets[hash];
                        this.count--;
                        return;
                    }
                }
            },

            Count: function ()
            {
                return this.count;
            },

            ToEnumerable: function ()
            {
                var self = this;
                return new Enumerable(function ()
                {
                    var currentEntry;

                    return new IEnumerator(
                        function () { currentEntry = self.entryList.First
},
                        function ()
                        {
                            if (currentEntry != null)
                            {
                                var result = { Key: currentEntry.Key,
Value: currentEntry.Value };
                                currentEntry = currentEntry.Next;
                                return this.Yield(result);
                            }
                            return false;
                        },
                        Functions.Blank);
                });
            }
        }

        return Dictionary;
    })();

    // dictionary = Dictionary<TKey, TValue[]>
    var Lookup = function (dictionary)
    {
        this.Count = function ()
        {
            return dictionary.Count();
        }

        this.Get = function (key)
        {
```

```
            return Enumerable.From(dictionary.Get(key));
        }

        this.Contains = function (key)
        {
            return dictionary.Contains(key);
        }

        this.ToEnumerable = function ()
        {
            return dictionary.ToEnumerable().Select(function (kvp)
            {
                return new Grouping(kvp.Key, kvp.Value);
            });
        }
    }

    var Grouping = function (key, elements)
    {
        this.Key = function ()
        {
            return key;
        }

        ArrayEnumerable.call(this, elements);
    }
    Grouping.prototype = new ArrayEnumerable();

    // out to global
    return Enumerable;
})()
```