Using .NET to Query an AutoCAD® Drawing Database

James E. Johnson – Synergis Software

This class is for AutoCAD® software developers who want to learn how to use generic collections, LINQ (Language Integrated Query), delegates, anonymous methods, extension methods, and other .NET features in their AutoCAD® applications. The class will explore how to improve code for working with an AutoCAD® drawing database when searching for data and getting collections, such as layers, layouts, and entities. We will discuss using .NET dynamic types (DLR) with the AutoCAD® 2013 API. In this class, you will learn the basics of using some intermediate to advanced .NET features with the AutoCAD® API. LINQ is a .NET Framework component that allows querying data much like SQL is used to query databases. DLR (Dynamic Language Runtime) runs on top of the CLR (Common Language Runtime) and provides language services for different languages.

Learning Objectives
At the end of this class, you will be able to:

• Query AutoCAD® drawing databases using LINQ
• Create and use anonymous methods, delegates, and extension methods
• Use generic and enumerable collections
• Use .NET Dynamic Language Runtime with AutoCAD API

About the Speaker
James has worked with CAD products for more than 25 years, involved in many positions from being a CAD drafter to writing automation applications. In his current position, he is doing CAD integration for Adept document management system. In previous positions, he has used Autodesk® RealDWG® to write custom automation to create AutoCAD® drawings of industrial kitchen equipment, and has worked at Autodesk resellers in software development groups doing custom applications for Inventor® and AutoCAD®. He has taught AutoCAD® and VBA classes while working for resellers, and was a CAD instructor at two different community colleges.

Email: james.e.johnson@me.com
What is LINQ ("Language Integrated Query")?

LINQ is a .NET programming model that adds querying capabilities to .NET programming languages. The term “Language Integrated Query” designates that query capabilities are integrated into the programming language being used (e.g., C#, Visual Basic).

LINQ is available with .NET framework version 3.0 and newer and defines a set of proprietary query operators that can be used to query objects and filter data in arrays, enumerable classes, XML, relational databases and third party data sources. Using generics, delegates and other features introduced in .NET 2.0 and other functional programming techniques, LINQ provides a SQL-like syntax for making query expressions. Queries written using LINQ query operators provides a short descriptive syntax to manipulate data and are executed by either the LINQ query processing engine or an extension mechanism of a LINQ provider.

AutoCAD® Visual Studio application requirements

To get started with LINQ, your project needs to have its target framework set to .NET Framework 3.0 or newer.

Also, adding the proper using or Imports statements into the class is recommended...

C#

using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;

VB.Net

Imports System.Collections
Imports System.Collections.Generic
Imports System.Linq
Imports System.Linq.Expressions
Querying the AutoCAD® Database

AutoCAD® customization applications often require adding, modifying and removing drawing entities, this is typically done using looping functions to step through collections of entities. Using LINQ can simplify the code required for doing searches and queries.

The AutoCAD® drawing database contains several tables (e.g. blocktable, layertable etc) and available collections that can be retrieved as arrays or a generic List<T>. This allows applying a LINQ operators to query the drawing database entities and tables for values of their properties.

This example uses foreach and an ArrayList to collect and step through the current active drawings layer table...

```csharp
public void getLayerNames()
{
    ArrayList layernames = new ArrayList();
    Database db = Application.DocumentManager.MdiActiveDocument.Database;
    using (Transaction tr = db.TransactionManager.StartTransaction())
    {
        LayerTable lt = (LayerTable)tr.GetObject(db.LayerTableId, OpenMode.ForRead);
        foreach (ObjectId obID in lt)
        {
            LayerTableRecord ltr = (LayerTableRecord)tr.GetObject(obID, OpenMode.ForRead);
            layernames.Add(ltr.Name);
        }
    }
    foreach (string name in layernames)
    {
        ed.WriteMessage(name + "\n");
    }
}
```

Same as above using a LINQ query with an anonymous method statement...

```csharp
public void getLayerNamesLinq()
{
    List<string> layernames = new List<string>();
    Database db = Application.DocumentManager.MdiActiveDocument.Database;
    using (Transaction tr = db.TransactionManager.StartTransaction())
    {
        LayerTable lt = (LayerTable)tr.GetObject(db.LayerTableId, OpenMode.ForRead);
        layernames = lt.Cast<ObjectId>()
            .Select<ObjectId, string>(oid =>
            {
                return ((LayerTableRecord)tr.GetObject(oid, OpenMode.ForRead, false)).Name;
            }).ToList();
    }
    layernames.ForEach(name => ed.WriteMessage(name + "\n"));
}
Similar to above samples using dynamic...

```csharp
public void getLayerNamesDynamic()
{
    List<string> layernames = new List<string>();
    Database db = Application.DocumentManager.MdiActiveDocument.Database;

    dynamic lts = db.LayerTableId;
    foreach (dynamic lt in lts)
    {
        layernames.Add(lt.Name);
    }

    layernames.ForEach(name => ed.WriteMessage(name + "\n"));
}
```

Dynamic sample to get Records from several tables...

```csharp
[CommandMethod("dumpTables")]
public static void dumpTables()
{
    Database db = Application.DocumentManager.MdiActiveDocument.Database;

    ed.WriteMessage("\nBlock Definitions: ");
    dump(db.BlockTableId);
    ed.WriteMessage("\n");

    ed.WriteMessage("\nLayers: ");
    dump(db.LayerTableId);
    ed.WriteMessage("\n");

    ed.WriteMessage("\nViews: ");
    dump(db.ViewTableId);
}

static void dump(dynamic tableID)
{
    List<string> tableNames = new List<string>();

    foreach (dynamic id in tableID)
    {
        tableNames.Add(id.Name);

        tableNames.ForEach(name => ed.WriteMessage("\n " + name));
    }
}
LINQ Comprehension Query Syntax vs. Method Query Syntax

Queries found in LINQ samples and documentation often are written as query expressions similar to SQL statements in a declarative query syntax introduced in C# 3.0. The .NET common language runtime (CLR) does not use query syntax and at compile time, query expressions are translated to method calls. The System.Linq namespace has many of these standard query operator methods which have names such as Where, Select, First, FirstOrDefault, Any, All, and many more. These operators can be called using method syntax instead of query syntax in your code.

Comprehension Query Syntax:

```csharp
var layernames =
  from oid in lt.Cast<ObjectId>()
  select ((LayerTableRecord)tr.GetObject(oid, OpenMode.ForRead, false)).Name;
```

Method Query Syntax:

```csharp
var layernames =
  lt.Cast<ObjectId>()
  .Select<ObjectId, string>(oid =>
  {
    return ((LayerTableRecord)tr.GetObject(oid, OpenMode.ForRead, false)).Name;
  });
```

There is no semantic difference between method query syntax and comprehension query syntax. In addition, queries can be written with comprehension query syntax and contain method query syntax within the query. Some queries that match a specified condition, or that retrieve an element that has the maximum value in a source sequence can be expressed only as method calls.

What are Generics?

The term "generic" is a common term for describing something that is not a brand name. For example, we often have medicine prescriptions filled with the “generic” version of a medicine or buy a store brand for groceries and other items. In programming “generic” can refer to a class that is not forced to any specific Type, but still be used in a type-safe manner. An example of where Generics are needed is in working with collections of items in AutoCAD® (Layers, Points, Entities, ObjectId collections etc). Generic classes are created to allow definition of type-safe collections without specifying the actual types used.

```csharp
List<string> layernames = new List<string>();

Func<Point3d, bool> pointFilter = p => (p.X > 0.0);

List<Database> dbAll = Autodesk.AutoCAD.DatabaseServices.Database.GetAllDatabases();
```
Arrays, Generic List<T> and IEnumerable


The Array class provides methods for creating, manipulating, searching and sorting arrays, which serves as the base class for all arrays. The Array class implements the IEnumerable interface.

The List<T> class represents a strongly typed list of objects that can be accessed by index and is the generic equivalent of the ArrayList class. It provides methods to search, sort, and manipulate lists.

The IEnumerable generic interface exposes the enumerator, which supports a simple iteration over a collection of a specified type.

```csharp
ArrayList layenames = new ArrayList();
List<string> layenames = new List<string>();
```

The primary difference between ArrayList and List<T> is that ArrayList does boxing and unboxing when operations are performed on the collection. Boxing is a conversion of a value-type to the object type which can add overhead causing a performance hit when a lot of operations are required.

The Generic IEnumerable<T> and IQueryable<T> interfaces

The generic interface IEnumerable<T> is defined for offering generic types a mechanism to iterate the elements of a sequence by exposing the enumerator, which supports a simple iteration over a collection of a specific type. The importance of the IEnumerable<T> interface with LINQ is the fact that any data type that implements it can directly serve as a source for query expressions.

The IQueryable<T> interface is primarily used for query providers and inherits the IEnumerable<T> interface so that a query can be enumerated. Enumeration executes the expression tree associated with an IQueryable<T> this may involve translating the expression tree to a query language appropriate for an underlying data source.
**Standard Query operators**

The Standard Query Operators which is the base of LINQ uses extension methods that implement the `IEnumerable<T>` interface. There are many different types of operators which can be found using the object browser to look at the System.Linq classes. The following list shows some of the different types of operators.

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregation</td>
<td>Aggregate, Average, Count, LongCount, Max, Min, Sum</td>
</tr>
<tr>
<td>Conversion</td>
<td>Cast, ConvertAll, OfType, ToArray, ToDictionary, ToList, ToLookup, ToSequence</td>
</tr>
<tr>
<td>Element</td>
<td>DefaultEmpty, ElementAT, ElementOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault</td>
</tr>
<tr>
<td>Equality</td>
<td>EqualAll</td>
</tr>
<tr>
<td>Generation</td>
<td>Empty, Range, Repeat</td>
</tr>
<tr>
<td>Grouping</td>
<td>GroupBy</td>
</tr>
<tr>
<td>Joining</td>
<td>GroupJoin, Join</td>
</tr>
<tr>
<td>Ordering</td>
<td>OrderBy, ThenBy, OrderByDescending, ThenByDescending, Reverse</td>
</tr>
<tr>
<td>Partitioning</td>
<td>Skip, SkipWhile, Take, TakeWhile</td>
</tr>
<tr>
<td>Quantifiers</td>
<td>All, Any, Contains</td>
</tr>
<tr>
<td>Restriction</td>
<td>Where</td>
</tr>
<tr>
<td>Selection</td>
<td>Select, SelectMany</td>
</tr>
<tr>
<td>Set</td>
<td>Concat, Distinct, Except, Intersect, Union</td>
</tr>
</tbody>
</table>

- `.Cast<ObjectId>()` - Casts to `IEnumerable`  
- `.ConvertAll<string>(DBT => { return ...... })` - Converts type  
- `.ToList<ObjectId>()` - Converts from `IEnumerable` to `Generic List`  
- `.ToArray<ObjectId>()` - Converts from `IEnumerable` to an `Array`  
- `.Select<ObjectId, Entity>(oid => ......)` - Projects elements to new form  
- `.Where<Entity>(ent => ....)` - Applies a predicate  
- `.Min<Extents3d>(ext => ext.MinPoint.X)` - Returns lowest value  
- `.Max<Extents3d>(ext => ext.MaxPoint.X)` - Returns Highest value
# LINQ Query Syntax Keywords

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>Query syntax expressions must start with a ‘from’ clause. Specifies the data source and a range variable that represents each element in the source.</td>
</tr>
<tr>
<td>where</td>
<td>Used in an expression to specify which elements will be returned. Applies a predicate to each element in the source specified in the ‘from’ clause range variable.</td>
</tr>
<tr>
<td>select</td>
<td>Specifies the type of values in the returned sequence when the query is executed.</td>
</tr>
<tr>
<td>group</td>
<td>Returns a sequence that contains none or many items that match a specified key value.</td>
</tr>
<tr>
<td>into</td>
<td>Used to create an identifier that can serve as a reference to the results of a join, group or select clause.</td>
</tr>
<tr>
<td>orderby</td>
<td>Sorts query results in ascending or descending order based on the default comparer for the element type.</td>
</tr>
<tr>
<td>join</td>
<td>Joins two data sources based on an equality comparison between two specified matching criteria.</td>
</tr>
<tr>
<td>let</td>
<td>Introduces a range variable to store sub-expression results in a query expression.</td>
</tr>
<tr>
<td>in</td>
<td>Contextual keyword in a join clause.</td>
</tr>
<tr>
<td>on</td>
<td>Contextual keyword in a join clause.</td>
</tr>
<tr>
<td>equals</td>
<td>Contextual keyword in a join clause.</td>
</tr>
<tr>
<td>by</td>
<td>Contextual keyword in a group clause.</td>
</tr>
<tr>
<td>ascending</td>
<td>Contextual keyword in an orderby clause.</td>
</tr>
<tr>
<td>descending</td>
<td>Contextual keyword in an orderby clause.</td>
</tr>
</tbody>
</table>
LET Keyword...

The let keyword is used to create new temporary variables with their values set or calculated inside the query.

```csharp
[CommandMethod("getCircleObjects")]
static public void getCircleObjects()
{
    List<Circle> dwgCircle = new List<Circle>();
    Database db = Application.DocumentManager.MdiActiveDocument.Database;
    using (Transaction tr = db.TransactionManager.StartTransaction())
    {
        BlockTable bt = (BlockTable)tr.GetObject(db.BlockTableId, OpenMode.ForRead);
        BlockTableRecord btr =
            (BlockTableRecord)tr.GetObject(bt[BlockTableRecord.ModelSpace], OpenMode.ForRead);

        var dbObjCollection = from oid in btr.Cast<ObjectId>()
            select new
            {
                DBObj = (DBObject)tr.GetObject(oid, OpenMode.ForRead, false)
            };

        IEnumerable<Circle> dbCirc = from obj in dbObjCollection
            where (obj.DBObj.GetType() == typeof(Circle))
            select (Circle)obj.DBObj;

        // Use Let keyword to define min area for where evaluation
        IEnumerable<Circle> dbCirc2 = from circ in dbCirc
            let minArea = 7.0
            let layer = "Layer1"
            where ((circ.Area > minArea) &&
                    (circ.Layer == layer) &&
                    (circ.Color == color))
            select (Circle)circ;

        dwgCircle = dbCirc2.Cast<Circle>().ToList();
        dwgCircle.ForEach(dbc => ed.WriteMessage(dbc.Diameter + "\n"));
    }
}
```
Anonymous Types and “Var” ...

LINQ is a data driven programming style which uses static structures instead of objects, anonymous types are used to allow new structures to be defined “inline” with their initialization.

This example takes an ObjectId (oid) from a BlockTableRecord, the creating an “inline” anonymous type with a property named “DBObj” assigns the DBObject found with that ObjectId to the anonymous type collection dbObj.

```csharp
var dbObj = from oid in btr.Cast<ObjectId>()
             select new
             {
               DBObj = (DBObject)tr.GetObject(oid, OpenMode.ForRead, false)
             };
```

This example then takes the above anonymous collection and casts to an anonymous collection of DBText entities then it is cast to a generic List<T>.

```csharp
var dbTxt = from obj in dbObj
            where (obj.DBObj.GetType() == typeof(DBText))
            select obj.DBObj;

List<DBText> dwgDBText = dbTxt.Cast<DBText>().ToList();
```

The use of the “var” keyword in many LINQ examples may remind you of less strongly typed languages like VB. Most languages now use strongly typed objects, also is the ability to use Generics, which brings even more Type control. The “var” keyword tells the compiler to infer the type of the variable from the static type of the expression used to initialize the variable. LINQ determines the type at compile time and VB determined the type at runtime. Used carefully the “var” type keyword can actually help decrease coding time.

You can stick to hardcore type setting and specify the type:

```csharp
IEnumerable<DBText> dbTxt = from obj in dbObj
                             where (obj.DBObj.GetType() == typeof(DBText))
                             select (DBText)obj.DBObj;
```

Instead of:

```csharp
var dbTxt = from obj in dbObj
            where (obj.DBObj.GetType() == typeof(DBText))
            select obj.DBObj;
```

In the first example, we are selecting the result type of “DBText”, so we must declare the result of the query as IEnumerable<DBText> to match the query result. This is how to strongly type a query result.

If the query was changed to not return a “DBText” Type, instead maybe a string Type then the code would need to be changed from the query result type IEnumerable<DBText> to IEnumerable<string>.
Delegates and Anonymous methods...

Delegates are supported for invoking methods and provide methods for adding and removing target methods. Sometimes a class or a method is created for using a delegate. Anonymous methods is a feature that was introduced in C# 2.0 which allows defining anonymous (nameless) method to be called by a delegate.

The following example defines a delegate that is processed in the using statement. Once the delegate is defined, it can be called as often as required.

```csharp
delegate List<DBText> getTxtObjects();
static public void getTextDelegate()
{
    Database db = Application.DocumentManager.MdiActiveDocument.Database;
    getTxtObjects delText = delegate()
    {
        BlockTable bt = (BlockTable)db.TransactionManager
            .TopTransaction
            .GetObject(db.BlockTableId, OpenMode.ForRead);
        BlockTableRecord btr = (BlockTableRecord)db.TransactionManager
            .TopTransaction
            .GetObject(bt[BlockTableRecord.ModelSpace],
            OpenMode.ForRead);
        var dbObj = from oid in btr.Cast<ObjectId>()
                    select new
                    {
                        DBObj = (DBObject)db.TransactionManager
                            .TopTransaction
                            .GetObject(oid, OpenMode.ForRead, false)
                    };
        IEnumerable<DBText> dbTxt = from obj in dbObj
                                    where (obj.DBObj.GetType() == typeof(DBText))
                                    select (DBText)obj.DBObj;
        return dbTxt.Cast<DBText>().ToList();
    };
    using (Transaction tr = db.TransactionManager.StartTransaction())
    {
        List<DBText> dwgDBText = delText();
        dwgDBText.ForEach(dbt => ed.WriteLine(dbt.TextString + "\n"));
    }
}
```
Sample of an anonymous method and Lambda delegate:

delegate void sampDelegate(string s);
[CommandMethod("sampleDelegate")]
static public void sampleDelegate()
{

    //anonymous delegate
    sampDelegate annonyDel = delegate(string s) { ed.WriteMessage(s); };
    annonyDel("I am anonymous");

    //lambda delegate
    sampDelegate lambdaDel = (x) => { ed.WriteMessage(x); };
    lambdaDel("I am a Lambda expression");
}

Sample using an anonymous list to collect properties...

[CommandMethod("getLayoutsNamesDynamicAnon")]
public void getLayoutsNamesDynamicAnon()
{
    // Create Anonymous list with properties
    var annoLayouts = new[]
    { new { LayoutName = "Model", TabNumber = 0 } }.ToList();
    annoLayouts.Clear();

    Database db = Application.DocumentManager.MdiActiveDocument.Database;
    dynamic bts = db.BlockTableId;
    foreach (dynamic bt in bts)
    {
        if (bt.IsLayout)
        {
            dynamic LO = bt.LayoutId;
            string loName = LO.LayoutName;
            int loOrder = LO.TabOrder;
            annoLayouts.Add(new { LayoutName = loName, TabNumber = loOrder });
        }
    }

    annoLayouts.ForEach(name =>
        ed.WriteMessage(name.LayoutName + "," + name.TabNumber.ToString() + "\n");
}
The Generic Delegate System.Func and System.Action...

The generic delegate System.Func can be used to construct delegate types created when needed without the requiring defining explicit delegate type declarations.

\[
\text{Func}<T\text{Result}>(); \\
\text{Func}<T\text{Arg0}, T\text{Result}>(T\text{Arg0} \ arg0); \\
\text{Func}<T\text{Arg0}, T\text{Arg1}, T\text{Result}>(T\text{Arg0} \ arg0, T\text{Arg1} \ arg1); \\
\text{Func}<T\text{Arg0}, T\text{Arg1}, T\text{Arg2}, T\text{Result}>(T\text{Arg0} \ arg0, T\text{Arg1} \ arg1, T\text{Arg2} \ arg2); \\
\text{Func}<T\text{Arg0}, T\text{Arg1}, T\text{Arg2}, T\text{Arg3}, T\text{Result}>(T\text{Arg0} \ arg0, T\text{Arg1} \ arg1, T\text{Arg2} \ arg2, T\text{Arg3} \ arg3); \\
\]

In the above Func types, the \(T\text{Arg0}\), \(T\text{Arg1}\), \(T\text{Arg2}\), and \(T\text{Arg3}\) parameters are argument types and the \(T\text{Result}\) parameter represents the result type.

The following example uses Func to declare an X and Y filter, then applies them when collecting the text objects...

```csharp
static public void getTextFunc()
{
    List<DBText> dwgDBText = new List<DBText>();
    Func<Point3d, bool> pointFilterX = p => (p.X > 4.0);
    Func<Point3d, bool> pointFilterY = p => (p.Y > 4.0);

    Database db = Application.DocumentManager.MdiActiveDocument.Database;
    using (Transaction tr = db.TransactionManager.StartTransaction())
    {
        BlockTable bt = (BlockTable)tr.GetObject(db.BlockTableId, OpenMode.ForRead);
        BlockTableRecord btr = (BlockTableRecord)tr.GetObject(bt[BlockTableRecord.ModelSpace], OpenMode.ForRead);

        var dbObj = from oid in btr.Cast<ObjectId>()
        select new
        {
            DBObj = (DBObject)tr.GetObject(oid, OpenMode.ForRead, false)
        };

        IEnumerable<DBText> dbTxt = from obj in dbObj
        where (objDBObject.GetType() == typeof(DBText))
        select (DBText)objDBObject;

        dwgDBText = dbTxt.Cast<DBText>()
            .ToList()
            .Where<DBText>(dbt => ((pointFilterX(dbt.Position) &&
            (pointFilterY(dbt.Position)))))
            .ToList();
    }
}
```

System.Action is a set of predefined delegates that encapsulate up to 4 arguments with no return values...

```csharp
Point3d newpoint = new Point3d(4.0, 4.0, 0.0);
Action<Point3d, string> pointAction = (p, s) => ed.WriteMessage(s + p.X.ToString());
pointAction(newpoint, "value of X: ");
```
Predicates...

A predicate is a function that returns true or false and can be used to filter the results. Also, a predicate defines a set of criteria, and determines if a specified object meets those criteria. Predicates take advantage of the generic features introduced in the .NET Framework 2.0 the documentation defines the System.Predicate delegate as:

```
Public Delegate Function Predicate(Of T)(obj As T) As Boolean
```

Some methods that use an instance of the System.Predicate delegate to perform their tasks are Exists, Find, FindAll, FindIndex, FindLastIndex, FindLast, RemoveAll and TrueForAll.

Example searches for layers containing the string “Layer” in the delegate used as the predicate:

```csharp
List<string> layernames = new List<string>();
List<string> foundLayers = new List<string>();

LayerTable lt = (LayerTable)tr.GetObject(db.LayerTableId, OpenMode.ForRead);
layernames =
    lt.Cast<ObjectId>()
    .Select<ObjectId, string>(oid =>
    {
        return ((LayerTableRecord)tr.GetObject(oid, OpenMode.ForRead, false)).Name;
    }).ToList();

string searchLayer = "Layer";
foundLayers = layernames.FindAll(delegate(string t)
    {
        return t.Contains(searchLayer);
    });
```

Also, the ForEach method of the List and Array classes uses the System.Action delegate to define an action to be done on each element.

```
dwgDBText.ForEach(dbt => ed.WriteMessage(dbt.TextString + "\n"));
```

Using the sample from above where the System.Func delegate is discussed. The delegates are defined and then applied when collecting the text. The pointFilterX and pointFilterY are used as predicates.

```
Func<Point3d, bool> pointFilterX = p => (p.X > 4.0);
Func<Point3d, bool> pointFilterY = p => (p.Y > 4.0);
IEnumerable<DBText> dbTxt = from obj in dbObj
    where (obj.DBObj.GetType() == typeof(DBText))
    select (DBText)obj.DBObj;
dwgDBText = dbTxt.Cast<DBText>()
    .ToList()
    .Where<DBText>(dbt => ((pointFilterX(dbt.Position) &&
                          (pointFilterY(dbt.Position)))))
    .ToList();
```

The best way of thinking about what predicates are is to think about them as filters, that will evaluate to True or False, and will filter the IEnumerable<T> source that the Expression is being applied to only contain the elements that match the filter (predicate).
Lambda Expressions...

Query operators provide functionality to perform filtering, projection, or extraction. These queries build on the concept of lambda expressions, which is a more powerful structure that builds on anonymous methods that can be passed as arguments for evaluation. Lambda expressions are an evolution of anonymous methods introduced in .NET 2.0. The ‘=>' symbol is used in lambda statements stating that the defined object “goes into” the expression...

From above example these delegates used lambda expressions...

```
Func<Point3d, bool> pointFilterX = p => (p.X > 4.0);
Func<Point3d, bool> pointFilterY = p => (p.Y > 4.0);
```

Nested lambda example ('ent' of type entity can be used in second inline lambda statement):

```
.Where<Entity>(ent => layernames.Any<string>(lay => lay == ent.Layer))
```

Lambda expressions used in delegates:

```
string[] parts = { "Bolt", "Nut", "Flange", "Pipe", "Bar", "Cylinder"};
Func<string, bool> filt = s => s.Length > 5;
Func<string, string> val = s => s;
Func<string, string> sel = s => s.ToUpper();
IEnumerable<string> query = parts.Where(filt).OrderBy(val).Select(sel);
```

Above example done just with delegates:

```
string[] parts = { "Bolt", "Nut", "Flange", "Pipe", "Bar", "Cylinder"};
Func<string, bool> filt = delegate(string s)
{
    return s.Length > 5;
};
Func<string, string> val = delegate(string s)
{
    return s;
};
Func<string, string> sel = delegate(string s)
{
    return s.ToUpper();
};
IEnumerable<string> query = parts.Where(filt).OrderBy(val).Select(sel);
```

Expressions or statement blocks can be contained in Lambda expressions.

```
a => 3.1416 * a // Expression
a => {return 3.1416 * a;} // Statement
```
The following delegate and lambda expression are equivalent and the lambda expression here is automatically converted into a corresponding delegate.

```csharp
Func<int> funcDelegate = delegate(int a)
{
    return a * 3.1416;
};

Func<int> lambda = a => a * 3.1416;
```

Developers use named methods, anonymous methods, or lambda expressions with query operators. The advantage with Lambda expressions is that the syntax is more readable and usually less verbose. Also, lambda expressions can be processed at runtime by optimizers, translators, and evaluators because it can be compiled as either code or data.

---

**Expression Trees...**

LINQ provides a simple syntax for translating code into a data structure called an expression tree. Using the System.Linq.Expressions namespace defines a generic type, Expression<T>, where T is the type of the delegate that defines the expression's signature. Expression trees are also used for creating LINQ Providers to extend querying to multiple data structures.

```csharp
// defines expression tree
Expression<Func<double, double, double>> pythag=(a, b)=>
    Math.Sqrt((a*a)+(b*b));

var trigCalc = pythag.Compile();
double C = trigCalc(4.0, 6.0);

// returns the body... "Sqrt((a*a)+(b*b))"
string pythagBody = pythag.Body.ToString();

// Uses point filter sample from other example but as an expression
Expression<Func<Point3d, bool>> pointFilterExpX = p => (p.X > 4.0);
var pntExpX = pointFilterExpX.Compile();
bool testPnt = pntExpX.Invoke(new Point3d(6, 4, 0));
```
Object Initializers...

Lambda expressions and extension methods provide everything needed to make queries for filtering members of a collection, perform an enumeration of those members, and change the members of the original sequence into members whose value or type may differ from the original.

A feature available in C# 3.0 is the syntax for object and collection initializers. This feature allows you to initialize an object by both creating the object instance (i.e. a new expression) as well as assign values to one or more properties in a single sentence, or initialize a collection with a set of values to add to it in a single expression similar to how you already do with arrays.

The following example illustrates using object initializers to set the values to instances of the class and then adding those objects to a generic list.

```csharp
// Title block data class
public class titleData
{
    public string Description { get; set; }
    public string DwgNumber { get; set; }
    public string RevisionNo { get; set; }
    public string DrawnBY { get; set; }
    public string Date { get; set; }
}
static public void objInitialize()
{
    titleData newDwg1 = new titleData()
    {
        Description = "New Drawing",
        DrawnBY = "User",
        Date = "12/2/2009",
        DwgNumber = "123456",
        RevisionNo = "A"
    };
    titleData newDwg2 = new titleData()
    {
        Description = "New Drawing",
        DrawnBY = "User",
        Date = "12/1/2009",
        DwgNumber = "7890",
        RevisionNo = "B"
    };
    List<titleData> drawings = new List<titleData>() { newDwg1, newDwg2 };
}
```

Anonymous types can also be populated using object initializers.

```csharp
var dwgInfo = new {
    Description = "New Drawing",
    DrawnBY = "James",
    Date = "12/2/2009",
    DwgNumber = "123456",
    RevisionNo = "A"
};
```
Extension Methods...

Extension methods allow the creation of methods on an existing “Type” creating the illusion of new methods, even when a class is compiled outside of the current assembly and can not be changed. With extension methods developers can augment the “Type” with new methods to provide their own specialized implementation. Extension methods are defined in static classes as static methods. In C#, extension methods are indicated by the “this” modifier which must be applied to the first parameter of the extension method.

```csharp
public static class extendAcad
{
    public static BlockTableRecord GetBlockTableRecord(this Database _db, bool mspace)
    {
        string space = mspace ? BlockTableRecord.ModelSpace : BlockTableRecord.PaperSpace;
        BlockTable bt = (_db as TransactionManager).TopTransaction.GetObject(_db.BlockTableId, OpenMode.ForRead);
        BlockTableRecord btr = (_db as TransactionManager).TopTransaction.GetObject(bt[space], OpenMode.ForRead);
        return btr;
    }

    public static IEnumerable<DBText> GetDBTextObjects(this Database _db, bool mspace)
    {
        BlockTableRecord btr = _db.GetBlockTableRecord(mspace);
        IEnumerable<DBText> dbTxt = btr.Cast<ObjectId>().ToList<ObjectId>().ConvertAll<DBObject>(oid =>
        {
            return _db.TransactionManager.TopTransaction.GetObject(oid, OpenMode.ForRead, false);
        }).Where<DBObject>(obj => obj.GetType() == typeof(DBText)).Cast<DBObject>().ToList();
        return dbTxt;
    }
}
```

The type of the first parameter of an extension method indicates what type the extension applies to.

Extension methods are resolved at compile-time based on which extension methods are in scope. When a namespace is imported with C#’s using statement or VB’s Import statement, all extension methods that are defined by static classes for that namespace are brought into scope.

Many of the LINQ standard query operators are extension methods in the System.Linq namespace, these extension methods extend IEnumerable<T> and IQueryable<T>. All but a few (e.g. OfType) of the standard query operators extend from the IEnumerable<T> interface, which means that all IEnumerable<T> types get the standard query operators by adding this using statement in C#:

```csharp
using System.Linq;
```
Deferred Query Evaluation...

Most LINQ standard query operators return elements when enumerated not when declared. These operators do no work UNTIL the query requests an element then suspends until the next element is requested. Deferred evaluation allows queries to be kept as IEnumerable<T>-based values that can be evaluated multiple times, each time yielding potentially different results. If you need a query to be enumerated immediately, use ToList() or ToArray() which both will enumerate the entire sequence returning a result.

The following example uses one of the previous examples to get text entities then modifies the "Last" text entity found then runs the query again...

```csharp
static public void getTextObjectsDeferred()
{
    List<DBText> dwgDBText = new List<DBText>();
    Database db = Application.DocumentManager.MdiActiveDocument.Database;
    using (Transaction tr = db.TransactionManager.StartTransaction())
    {
        BlockTable bt = (BlockTable)tr.GetObject(db.BlockTableId, OpenMode.ForRead);
        BlockTableRecord btr = (BlockTableRecord)tr.GetObject(bt[BlockTableRecord.ModelSpace], OpenMode.ForRead);
        var dbObj = from oid in btr.Cast<ObjectId>()
            select new {
                DBObj = (DBObject)tr.GetObject(oid, OpenMode.ForRead, false)
            };
        IEnumerable<DBText> dbTxt = from obj in dbObj
            where (obj.DBObj.GetType() == typeof(DBText))
            select (DBText)obj.DBObj;
        dwgDBText = dbTxt.Cast<DBText>().ToList();

        //OutPut text values
        dwgDBText.ForEach(dbt => ed.WriteMessage(dbt.TextString + "\n"));

        // Modify the text value of last entity found
        DBText lastTextEnt = dwgDBText.Last<DBText>();
        lastTextEnt.UpgradeOpen();
        lastTextEnt.TextString = "MODIFIED";

        //OutPut Text values again ... Query of text objects is processed again
        dwgDBText.ForEach(dbt => ed.WriteMessage(dbt.TextString + "\n"));
    }
}
```

The query is evaluated each time the variable is iterated over. To indicate that a cached copy of the results is needed, as mentioned above append a ToList() or ToArray() operator to the query like this:

The same is true for the standard query operators that return singleton values (e.g., First, FirstOrDefault, ElementAt, ElementAtOrDefault, Sum, Average, All, Any).
Use .NET Dynamic Language Runtime with AutoCAD API...

The Dynamic Language Runtime (DLR) is used to implement dynamic languages like Python and Ruby on the .NET Framework.

In Visual Studio 2010 C# a new type of 'dynamic' was introduced. This is a static type that bypasses static type checking and functions like it has a type of object. A dynamic type at compile time assumes support for any operation, whether its value comes from a COM API, a dynamic language, an HTML Document Object Model (DOM), reflection, or somewhere in the program. Errors are caught at run time for invalid code.

The equivalent to the dynamic keyword is object in VB.NET but with 'Option Strict Off', with 'Option Strict On' there is no equivalent.

This example was used above to show using dynamic...

```csharp
public void getLayerNamesDynamic()
{
    List<string> layernames = new List<string>();
    Database db = Application.DocumentManager.MdiActiveDocument.Database;
    dynamic lts = db.LayerTableId;
    foreach (dynamic lt in lts)
    {
        layernames.Add(lt.Name);
    }
    layernames.ForEach(name => ed.WriteMessage(name + "\n"));
}
```

Note: the properties and methods are effectively being determined and then accessed/called at run-time, there is no Intellisense available when coding...

Similar to above layer sample to get Layouts...

```csharp
[CommandMethod("getLayoutsNamesDynamic")]
public void getLayoutsNamesDynamic()
{
    List<string> layoutNames = new List<string>();
    Database db = Application.DocumentManager.MdiActiveDocument.Database;
    dynamic bts = db.BlockTableId;
    foreach (dynamic bt in bts)
    {
        if (bt.IsLayout)
        {
            dynamic lid = bt.LayoutId;
            layoutNames.Add(lid.LayoutName);
        }
    }
    layoutNames.ForEach(name => ed.WriteMessage(name + "\n"));
}
```
LINQ Providers...

LINQ works between the programming language and the data source that queries are applied against. A provider works between the LINQ engine and the data source to extend the query functionality on the data source.

Sample walkthrough for building a LINQ provider:


Web Links...

The following web links are some of the ones used for research in creating this paper...

http://www.hookedonlinq.com
http://www.thinqling.com
http://linqinaction.net
http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b

For some sample LINQ code and additional documents defining Standard Queries, download files at:

http://code.msdn.microsoft.com/site/search?query=linq&f%5B0%5D.Value=linq&f%5B0%5D.Type=SearchText&ac=1