Delegates

- A delegate references a method with specific signature and return type.
- Delegate is used to invoke the method that it references at runtime.
- Used to invoke the method that is not known until runtime.
- Similar to function pointers in C and C++ but delegates are object –oriented and type-safe.
- Delegate is created using keyword **delegate**.
- A delegate type is internally a class that is derived from **System**.**Delegate** class.

1. **Creating a delegate type**
Create a delegate type. Delegate type specifies signature of the function that delegate type can point.

```
public delegate return-type delegate-name(parameters);
```

Create a delegate type called **ArithmeticOperator**, which can point to a function that takes two integers and returns an integer, as follows:

```
public delegate int ArithmeticOperator (int a, int b);
```

2. Making delegate point to a method
The signature of the method and signature specified by delegate type must be same. To create a delegate, create an object of delegate type.

```
ArithmeticOperator a = new ArithmeticOperator (Test.Add);
```

3. Invoke method through delegate
Invoke method pointed by delegate by using either just delegate followed by () or by using **Invoke** method.

```
result = a.Invoke(10,20);
result = a(10,20)
```

The following is a complete program that demonstrates all three steps of a delegate.

```
namespace st{
   class DelegateDemo   {
     // declare delegate type
     public delegate void  PrintDelegate(string message);
     public static void Main()  {
       // create delegate and make it point to method
       PrintDelegate  d = new PrintDelegate (Print);
       // invoke method pointed by delegate
       d("Hello");
       d.Invoke("World");
     }
     public static void Print(string message) {
       Console.WriteLine(message);
     }
  }
}
```

Delegate Method group conversion

It is possible to assign a method directly to a delegate without the use of new or explicitly invoking constructor. This feature was added in C# 2.0.

```
// declare delegate type
public delegate void  PrintDelegate(string message);
public static void Main(){
   PrintDelegate d = Print; // method group conversion
   d("Hello");
}
```

```
public static void Print(string message){
   Console.WriteLine(message);
}
```

Using Delegate with Instance Methods

- It is possible to make a delegate point to an instance method.
- As instance methods are invoked with an object, associate the method with an object when it is assigned to delegate.

```
using System;
namespace st{
  class DelegateDemo  {
    public delegate void  PrintDelegate(string message);
    public static void Main() {
      PrintMessage obj = new PrintMessage(10);
      PrintDelegate pd = obj.Print;
      pd("Hello");
    }
  }
  class PrintMessage{
    private int count;
    public PrintMessage(int count){
       this.count = count;
    }
    public void Print(string messsage){
     for ( int i = 1; i <= count ; i ++)
            Console.WriteLine( messsage );
    }
 }
}
```

Multicasting

- Allows two or more methods to be called using a single delegate.
- The delegate must return **void**.
- Use **+** to add new methods and **–** to remove existing methods.

```
 class TestMulticasting  {
       delegate void PrintHandler();
       public static void Main () {
           PrintHandler ph1 = Print1;
           PrintHandler ph2 = Print2;

           PrintHandler both = ph1;
           both = both + ph2;  // add a delegate to another
           both();  // calls  print1 and print2
           both -= ph1;
           both();
       }
       public static void Print1() {
           Console.WriteLine("Print1");
       }
```

```
     public static void Print2() {
            Console.WriteLine("Print2");
        }
} // class
```

The output of the above program will be as follows:

```
Print1
Print2
Print2
```

Anonymous Method

- Starting from C# 2.0, anonymous methods can be used with delegate.
- Anonymous method is a block of code with **delegate** keyword that is passed to a delegate.

```
delegatetype delegate =  delegate [ (parameters) ] {
     // code
     [ return value; ]
};
```

The following example shows anonymous method associated with delegate.

```
namespace csharpdemo {
    class AnonymousMethod{
        public delegate int ArithmeticOperator(int a, int b);
        static void Main() {
            // anonymous method with delegate
            ArithmeticOperator addop =
                    delegate(int n1, int n2)
                    {
                         return n1 + n2;
                    };
            Console.WriteLine(addop(10, 20));
        }
    }
}
```

Using Delegates with Array.FindAll()

Array.**FindAll**() is used to select elements of the array based on the given condition. It expects an array as first parameter. Second parameter is of type **Predicate<T>**, which represents a function that takes a single parameter of type T and returns true if value is to be selected, otherwise false.

The following example shows how to use delegate with **FindAll**() method of **Array** class to select only positive numbers and then only negative numbers.

```
using System;
namespace csharpdemo {
    class ArrayAndDelegate {
        static void Main() {
            int[] a = { 10, -11, 0, 33, -14 };
            int[] sa;

            sa = Array.FindAll(a, Positive);
            foreach (int n in sa)
                Console.WriteLine(n);

            // use anonymous method
            sa = Array.FindAll(a, delegate (int v)
                                  {
                                      return v < 0;
                                  } );
           foreach (int n in sa)
                Console.WriteLine(n);
        }
        static bool Positive(int n){
            return n > 0;
        }
    }
}
```

Lambda Expressions

A lambda expression is written as a parameter list, followed by the **=>** token, and then an expression or statement

In an implicitly typed parameter list, the types of the parameters are inferred from the context in which the lambda expression is used. In addition, if a lambda expression has a single, implicitly typed parameter, the parentheses may be omitted from the parameter list.

```
x => x + 1       // implicit parameter list, expression
(x, y) => x * y; // implicit parameter list, expression
```

Lambda expressions simplify the code required. Notice that in this example the parameter list consisted of the variable '**c**' which was implicitly typed and therefore did not require the type to be explicitly stated.

```
using System;
namespace st {
    class LambdaDemo {
        public static void Main() {
           string [] names = { "Bill Gates","Larry Ellison",
                               "Steve Jobs", "Micheal Dell"};
           PrintArray(names);

           // normal delegate
           string[] selected1 = Array.FindAll(names,
                        new Predicate<string>(Check));
           PrintArray(selected1);

            // method group conversion
            string[] selected2 = Array.FindAll(names, Check);
            PrintArray(selected2);
```

```
            // anonymous method
            string[] selected3 = Array.FindAll(names,
                                    delegate(string s)
                                    { return s.Length > 10;}
                                   );
            PrintArray(selected3);

            // lambda expression
            string [] selected4 = Array.FindAll(names,
                                       s => s.Length > 10);
            PrintArray(selected4);
        }
        public static bool Check(String s)
        {
            return s.Length > 10;
        }
        public static void PrintArray(string[] names)
        {
            foreach (string s in names)
                Console.WriteLine(s);
        }
    }
}
```

Statement Lambdas

A statement lambda resembles an expression lambda except that the statement(s) is enclosed in braces:

```
(input-parameters) => { statement; }
```

```
namespace csharp
{
    delegate void PrintDelegate(string msg, int count);

    class LambdaStatementDemo
    {
        public static void Main()
        {
            PrintDelegate del = (msg, count) =>
            {
                for (int i = 1; i <= count; i++)
                {
                    Console.WriteLine(msg);
                }
            };

            del("Lambda Statement", 3);
        }
    }
}
```

Variable Scope in Lambda Expressions

- Lambdas can refer to outer variables that are in scope in the method that defines the lambda function, or in scope in the type that contains the lambda expression.
- Variables that are captured in this manner are stored for use in the lambda expression even if the variables would otherwise go out of scope and be garbage collected.
- An outer variable must be definitely assigned before it can be consumed in a lambda expression.

The following rules apply to variable scope in lambda expressions:

- A variable that is captured will not be garbage-collected until the delegate that references it becomes eligible for garbage collection.
- Variables introduced within a lambda expression are not visible in the outer method.
- A lambda expression cannot directly capture a ref or out parameter from an enclosing method.
- A return statement in a lambda expression does not cause the enclosing method to return.
- A lambda expression cannot contain a goto statement, break statement, or continuestatement that is inside the lambda function if the jump statement's target is outside the block. It is also an error to have a jump statement outside the lambda function block if the target is inside the block.

Action<T> Delegate

Encapsulates a method that has a single parameter and does not return a value.

```
public delegate void Action<in T>(
      T obj
)
```

Func<T,TResult> Delegate

Encapsulates a method that has one parameter and returns a value of the type specified by the TResult parameter.

```
public delegate TResult Func<in T, out TResult>(
      T arg
)
```

Type Parameters

T	The type of the parameter of the method that this delegate encapsulates.
TResult	The type of the return value of the method that this delegate encapsulates.

Events

- An event is a message sent by an object to signal the occurrence of an action.
- The object that raises the event is called as **event sender** or publisher.
- The object that receives the event is called as **event receiver** or consumer.
- The senders send event to consumers using delegate, which is specially defined for it.
- The method that handles the event is called as **event handler**.
- An event is a way for a class to notify objects that they need to perform an action of some kind.
- The most common use for events is in graphical user interfaces, although events can be useful at other times, such as signaling state changes.

```
event type member-name
```

- **event** is keyword to define event.
- **Type** is the delegate to which the event is to be associated with.
- **Member-name** is the name of the event.

```
using System;
namespace st {
        public delegate void MyEventHandler();  // 1
        class EventDemo    {
            public static event MyEventHandler myevent; // 2
            public static void Main() {
              myevent += Handler;  // add handler to event // 3
              // anonymous method
              myevent +=
               delegate  {
                 Console.WriteLine("Another Event Handler");
               };   // 4

              // Handler and anonymous methods are executed.
              if (myevent != null)
                        myevent();  // 5
            }

            public static void Handler() {
                Console.WriteLine("Event occurred");
            }
       }
}
```

Other points related to events:

- Events can be specified in interfaces. Then the implementing class must provide an event.
- Events can be specified abstract to make derived class implement an event.
- An event can be sealed.
- An event may be virtual to allow derived class to override it.
- To be compatible with .NET event guidelines, event must have two parameters – first is of **object** type and second is **EventArgs**.

Multithreading

- Allows multi-tasking within a single process.
- Threads are called as light-weight processes as they have fewer overheads compared with processes.
- Used when a single process has to perform multiple operations simultaneously.

Steps to create a new Thread

- Create an object of **Thread** class which is in **System**.**Threading** namespace.
- Create an object of **ThreadStart** delegate, which points to a method that takes nothing and returns nothing. This method defines the process related to thread.
- Pass object of **ThreadStart** to the constructor of **Thread** class.
- Call **Start** method of **Thread** object.

```
using System;
using System.Threading;

class Test{
   public static void CountUp()
   {
      for ( int i  = 1; i <= 1000; i ++)
        Console.WriteLine ("Increment --> {0}",i);
   }
   public static void CountDown()
   {
      for ( int i=1000; i >0; i --)
        Console.WriteLine ("Decrement --> {0}",i);
   }
   public static void Main()
   {
     Thread  t1 = new Thread (new ThreadStart(CountUp));
     Thread  t2 = new Thread (CountDown);
     t1.Start();
     t2.Start();
   }
} // end of class
```

Thread class

- A thread is an object of Thread class.
- Thread class creates and controls a thread, sets its priority, and gets its status.

Property	Meaning
static CurrentThread	Returns the currently running thread. It is a static property.
IsAlive	Returns true if thread is currently alive.
Name	Returns or sets name of the thread.
Priority	Returns or sets name of the thread. Available options : ❑ Highest ❑ AboveNormal ❑ Normal ❑ BelowNormal ❑ Lowest
ThreadState	Returns the state of the thread. See list below for different states.
IsBackground	Gets or sets a value indicating whether or not a thread is a background thread.

Members of **ThreadState** Enumeration :

State	Description
Running	The thread has been started, it is not blocked.
StopRequested	The thread is being requested to stop. This is for internal use only.
SuspendRequested	The thread is being requested to suspend.

Background	The thread is being executed as a background thread, as opposed to a foreground thread. This state is controlled by setting the IsBackground property.
Unstarted	The Start method has not been invoked on the thread.
Stopped	The thread has stopped.
WaitSleepJoin	The thread is blocked. This could be the result of calling Sleep or Join, of requesting a lock.
Suspended	The thread has been suspended.
AbortRequested	The Abort method has been invoked on the thread.
Aborted	The thread state includes AbortRequested and the thread is now dead, but its state has not yet changed to Stopped.

Method	**Meaning**
Abort	Terminates the thread. Also throws **ThreadAbortException**.
Join	Blocks calling thread until given thread terminates.
Start	Causes a thread to be scheduled for execution.
Interrupt	Interrupts a thread that is in **WaitSleepJoin** state.
Sleep	Makes the current thread sleep for the given number of milliseconds. It is a static method.
GetDomain	Returns domain in which thread is running.

Operator Overloading

- Allows standard operators to be used with user-defined types.
- Keyword **operator** and static method are used to overload an operator.
- Parameters of the method are operands for the operation.
- It is required to overload logical pairs such as == and !=, < and >, and >= and <= etc.

```
static  returntype operator<symbol>(type parameter)   // unary
{ }
static  returntype operator<symbol>(type p1, type p2) // binary
{ }
```

- **operator** is keyword.
- **symbol** is the operator to be overloaded.
- For unary operator, only one parameter is required.
- For binary operator, two parameters are required.

```
using System;
namespace st {
    class Point  {
        private int x, y;
        public Point(int x, int y) {
            this.x = x;
            this.y = y;
        }
        public static bool operator ==(Point p1, Point p2){
            return p1.x == p2.x && p1.y == p2.y;
        }
        public static bool operator !=(Point p1, Point p2){
            return p1.x != p2.x || p1.y != p2.y;
        }
        public static Point operator ++(Point p1){
            p1.x++;
            return p1;
        }
        public override string ToString(){
            return String.Format("{0},{1}", x, y);
        }
    }
```

```
    class OperatorOverloadingDemo {
      public static void Main() {
       Point p1 = new Point (10, 20);
       Point p2 = new Point (10, 20);

       Console.WriteLine(p1 == p2); // Point.operator==(p1,p2)
       p1++;                        // Point.operator++(p1)
       Console.WriteLine(p1);
      }
   }
}
```

Restrictions of operator overloading

The following are the restrictions regarding operator overloading.

- You cannot alter the precedence of any operator
- You cannot change the number of operands
- You cannot overload assignment operator (=) including compound assignments (+=, *= etc.)
- The operators that cannot be overloaded are :
  && || {} () new is sizeof typeof ? → . = as

Exercise: Provide operator functions for ==, !=, >, < and ++ for Time class.

Conversion Operators

- It is possible to convert user-defined type to standard type and vice-versa using conversion operators.
- Conversion operator is a special type of operator method.
- It is used to convert an object of class to another type.

```
public static implicit operator target-type (source-type)
{
  return   target-type-value;
}
public static explicit operator target-type (source-type)
{
  return   target-type-value;
}
```

- **operator** is the keyword.
- **implicit** keyword specifies the conversion is to be invoked automatically.
- **explicit** keyword specifies the conversion must be done only when cast is used.
- **Target-type** is the type to which you are converting.
- **Source-type** is the type from which you are converting.
- Return value must be of type **target-type.**

```
    class Point  {
       private int x, y;
       public Point(int x, int y){
           this.x = x;
           this.y = y;
       }
       public Point(int v) {
           this.x = this.y = v;
       }
       public void Print() {
           Console.WriteLine("{0}, {1}", x, y);
       }
```

```
        // conversion from int to Point
        public static implicit operator Point(int n) {
            return new Point(n);
        }
        // conversion from Point to int
        public static explicit operator int(Point obj) {
            return obj.x * obj.y;
        }
    } // end of Point
    class ConversionOperators  {
       public static void Main(){
        Point p = 10;   //Implicit conversion from int to Point
        p.Print();
        int v = (int)p; //Explicit conversion from Point to int
        Console.WriteLine("Value of v = {0}", v);
      }
    }
```

Restriction regarding conversion operators :

- Either source or target type of the conversion must be the class in which the conversion is declared.
- You cannot define a conversion to or from object.
- You cannot define both an implicit and an explicit conversion for the same source and target types.
- You cannot define a conversion from a base class to a derived class.
- You cannot define a conversion from or to an interface.
- Implicit conversion should be used only when conversion doesn't result in loss of data or exception, otherwise explicit conversion should be used.

Exercise: Add conversion operators to Time class to int (total seconds).

Generics

- Generics were added to version 2.0 of the C# language and the common language runtime (CLR).
- Generics introduce type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code.
- By using a generic type parameter T you can write a single class that other client code can use without incurring the cost or risk of runtime costs or boxing operations.
- Use generic types to maximize code reuse, type safety, and performance.
- The most common use of generics is to create collection classes.
- The .NET Framework class library contains several new generic collection classes in the System.Collections.Generic namespace.
- You can create your own generic interfaces, classes, methods, events and delegates.
- Generic classes may be constrained to enable access to methods on particular data types.
- Information on the types that are used in a generic data type may be obtained at run-time by using reflection.

```
class  class-name <type-parameter-list> {
}
```

The following program illustrates how to create Stack class that works with parameterized type.

```
using System;
namespace csharpdemo {
    class GenericStack<T> {
        T[] a = new T[10];
        int top = 0;
        public void Push(T v) {
            a[top] = v;
            top++;
        }
        public T Pop() {
            top--;
            return a[top];
        }
        public int Length {
            get
            {
                return top;
            }
        }
    }
class GenericStackDemo {
        public static void Main() {
            GenericStack<int> istack = new GenericStack<int>();
            istack.Push(10);
            istack.Push(20);

            Console.WriteLine(istack.Pop());
            Console.WriteLine(istack.Length);

            GenericStack<string> strstack=
                                    new GenericStack<string>();
            strstack.Push("Scott");
            strstack.Push("Joe");

            Console.WriteLine(strstack.Pop().ToUpper());
            Console.WriteLine(strstack.Length);
        }
    }
}
```

Constraints on Type Parameters

- When you define a generic class, you can apply restrictions to the kinds of types that client code can use for type arguments when it instantiates your class.
- If client code tries to instantiate your class by using a type that is not allowed by a constraint, the result is a compile-time error.
- These restrictions are called constraints.
- Constraints are specified by using the contextual keyword where.

Constraint	Description
where T: struct	The type argument must be a value type. Any value type except Nullable can be specified.
where T : class	The type argument must be a reference type; this applies also to any class, interface, delegate, or array type.
where T : new()	The type argument must have a public parameterless constructor. When used together with other constraints, the new() constraint must be specified last.
where T : <base class name>	The type argument must be or derived from the specified base class.
where T : <interface name>	The type argument must be or implements the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic.
where T : U	The type argument supplied for T must be or derived from the argument supplied for U. This is called a naked type constraint.

```
public class GenericList<T> where T : Employee
{
  // definition
}
```

```
class EmployeeList<T> where T : Employee, IEmployee, new()
{
    // ...
}
```

The following example demonstrates how to use constraints with generics.

```
using System;
namespace csharpdemo {
    class GenericConstraintsDemo {
        public static void Main()   {
          int[] a = { 10, 70, 40, 20, 60 };
          ArrayUtils<int> au = new ArrayUtils<int>(a);
          au.Sort();
          au.Print();
          Console.WriteLine(
               "Found at {0} position",au.Search(40));
        }
    }
    class ArrayUtils<T> where T : IComparable {
        T [] values;
        public ArrayUtils(T[] values) {
            this.values = values;
        }
        public void Print() {
            foreach (T v in values)
                Console.WriteLine(v);
        }
        public void Sort() {
            Array.Sort(values);
        }
```

```
     public int Search(T v) {
            for (int i = 0; i < values.Length; i++)
               if (values[i].CompareTo(v) == 0)
                     return i;
            return -1;
         }
     }
}
```

Generic Interfaces

An interface can define more than one type parameter.

```
interface ISeries<T> {
   void SetStart(T v);
   void Reset();
   T Value
   {
      get;
   }
}
class OneSeries<T>: ISeries<T> {
    T start, value;
    public void SetStart(T v) {
        start = value =  v;
    }
    public void Reset() {
        value = start;
    }
    public T Value {
        get
        {
           return value;
        }
    }
}
class GenericDemo {
   public static void Main() {
      OneSeries<int> s = new OneSeries<int>();
      s.SetStart(100);
      Console.WriteLine(s.Value);
   }
}
```

default keyword

- Given a variable t of a parameterized type T, the statement t = null is only valid if T is a reference type and t = 0 will only work for numeric value types but not for structs.
- The solution is to use the **default** keyword, which will return null for reference types and zero for numeric value types.
- For structs, it will return each member of the struct initialized to zero or null depending on whether they are value or reference types

```
class OneSeries<T> : ISeries<T> {
   T start, value;
   public void Reset()
   {
        value = default(T);
   }
}
```

Generic Methods

It is possible to create methods that use parameterized types in non-generic classes also.

```
using System;
namespace st {
    class ArrayUtil {
        public static void Fill<T>(T[] a) {
            for( int i = 0; i < a.Length ; i ++)
                a[i] = default(T);
        }
        public static void Print<T>(T [] a) {
            foreach (T v in a)
                Console.WriteLine(v);
        }
    }
    class GenericMethodsDemo {
        public static void Main(){
            int[] a = { 10, 20, 40, 22, 11 };
            ArrayUtil.Print(a);

            double [] d = new double[10];
            ArrayUtil.Print(d);
        }
    }
}
```

Extension methods

- Extension methods provide a way for developers to extend the functionality of existing types by defining new methods that are invoked using the normal instance method syntax. Extension methods are static methods that are declared by specifying the keyword **this** as a modifier on the first parameter of the methods.
- With **C# 3.0**, you can now define an extension method that can be invoked using instance method syntax. An extension method is declared by specifying the keyword **this** as a modifier on the first parameter of the method.
- Extension methods are only available if declared in a **static** class and are scoped by the associated namespace. They then appear as additional methods on the types that are given by their **first parameter**.

```
using System;
namespace csharpdemo {
    public class Customer  {
        public string Name { get; set; }
        public string City { get; set; }
    }
    class ExtensionMethods {
      public static void Main() {
       Customer c1 = new Customer  {Name="Srikanth",City="Vizag"};
       Customer c2 = new Customer  {Name="Praneeth",City="Vizag" };

       c1.Print();   // calls extension method
       if (c1.Compare(c2))   // calls extension method
           Console.WriteLine("Same");
       else
           Console.WriteLine("Not same");
      }
    }

    public static class Extensions {
     public static bool Compare(this Customer c1, Customer c2){
        if (c1.Name == c2.Name && c1.City == c2.City)
             return true;
        else
             return false;
    }

    // Extension method for Customer class
    public static void Print(this Customer c1) {
         Console.WriteLine("Name : {0}", c1.Name);
         Console.WriteLine("City : {0}", c1.City);
    }
  }
}
```

Implicitly Typed Local Variables and Arrays

- The implicit typing of local variables is a general language feature that relieves the programmer from having to specify the type of the variable.
- Instead, the type is inferred by the compiler from the type of the variable's initialization expression.
- With C# 3.0, arrays can be initialized implicitly and variables can be declared implicitly within a local scope.
- With implicitly typed arrays, this is achieved by initializing arrays using new[]{…} (notice the lack of type here).
- With implicitly typed local variables, this is done when declaring a local variable as type var (assuming no type named var is in scope).

```
static void VarTest() {
  var i = 43;
  var s = "...This is only a test...";
  var numbers = new[] { 4, 9, 16 };
  var complex = new SortedDictionary <string, DateTime>();
  complex.Add("Today", DateTime.Now);
}
```

An implicit array initialization expression must specify that an array is being created and must include **new[]**.

```
var x = new[] {1, 2, 3};
```

This feature affects only the declaration of variables at compile time; the compiler infers the type of the variable from the type of expression used to initialize it. From then on throughout the program, it is as if the variable was declared with that type; assigning a value of a different type into that variable will result in a compile time error.

Anonymous Types

To facilitate the creation of classes from data values, **C# 3.0** provides the ability to easily declare an anonymous type and return an instance of that type. To create an anonymous type, the **new** operator is used with an anonymous object initializer.

The C# compiler automatically creates a new type that has two properties: one named **Name** of type string, and another named **Age** with type int:

```
var person = new { Name = "Steve", Age = 33 };
```

Each member of the anonymous type is a property inferred from the object initializer. The name of the anonymous type is automatically generated by the compiler and cannot be referenced from the user code.

It is also possible to omit the names, in which case, the names of the generated members are the same as the members used to initialize them. This is called a *projection initializer*.

```
public static void Main()
{
   string[] names = { "Kaka", "Messi", "Ronaldo", "Gerrad" };

   foreach (var v in names)
   {
     var details = new { Name = v, Length = v.Length };
     Console.WriteLine(details.Name + "\t" + details.Length);
   }
}
```

Dynamic Lookup with dynamic keyword

- Dynamic lookup allows you a unified approach to invoking things dynamically.
- With dynamic lookup, when you have an object in your hand you do not need to worry about whether it comes from COM, IronPython, the HTML DOM or reflection.
- You can apply operations to it and leave it to the runtime to figure out what exactly those operations mean for that particular object.
- This affords you enormous flexibility, and can greatly simplify your code, but it does come with a significant drawback: Static typing is not maintained for these operations.
- A dynamic object is assumed at compile time to support any operation, and only at runtime will you get an error if it wasn't so.

The `dynamic` keyword

- C# 4.0 introduces a new static type called `dynamic`. When you have an object of type `dynamic` you can "do things to it" that are resolved only at runtime.
- The C# compiler allows you to call a method with any name and any arguments on `d` because it is of type `dynamic`.
- At runtime the actual object that `d` refers to will be examined to determine what it means to "call `M` with an `int`" on it.
- The type `dynamic` can be thought of as a special version of the type `object`, which signals that the object can be used dynamically.
- Not only method calls, but also field and property accesses, indexer and operator calls and even delegate invocations can be dispatched dynamically.

```
dynamic d = GetDynamicObject(…);
d.M(7);             // calling methods
d.f = d.P;          // getting and setting fields and properties
int i = d + 3;      // calling operators
string s = d(5,7); // invoking as a delegate
```

Dynamic Language Runtime (DLR)

- An important component in the underlying implementation of dynamic lookup is the Dynamic Language Runtime (DLR), which is a new API in .NET 4.0.
- The DLR provides most of the infrastructure behind not only C# dynamic lookup but also the implementation of several dynamic programming languages on .NET, such as IronPython and IronRuby.
- Through this common infrastructure a high degree of interoperability is ensured, but just as importantly the DLR provides excellent caching mechanisms which serve to greatly enhance the efficiency of runtime dispatch.

LINQ (Language Integrated Query)

- Language-Integrated Query (LINQ) is a groundbreaking innovation in Visual Studio 2008 and the .NET Framework version 3.5 that bridges the gap between the world of objects and the world of data.
- LINQ is a methodology that simplifies and unifies the implementation of any kind of data access.
- Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on.
- LINQ makes a query a first-class language construct in C# and Visual Basic. You write queries against strongly typed collections of objects by using language keywords and familiar operators
- At compile time, query expressions are converted to Standard Query Operator method calls according to the rules set forth in the C# specification. Any query that can be expressed by using query syntax can also be expressed by using method syntax. However, in most cases query syntax is more readable and concise.
- However, some query operations, such as Count or Max, have no equivalent query expression clause and must therefore be expressed as a method call.

Query Operators

- Query operators can be combined to perform complex operations and queries on enumerations.
- Several query operators are predefined and cover a wide range of operations.
- These operators are called the standard query operators.
- Query operators are static methods that allow the expression of queries.
- Query operators are mainly extension methods working with **IEnumerable**<T> objects,

Family	Query Operators
Filtering	OfType, Where
Projection	Select, SelectMany
Partitioning	Skip, SkipWhile, Take, TakeWhile
Join	GroupJoin, Join
Concatenation	Concat
Ordering	OrderBy, OrderByDescending, Reverse, ThenBy, ThenByDescending
Grouping	GroupBy, ToLookup
Set	Distinct, Except, Intersect, Union
Conversion	AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary,
Equality	SequenceEqual
Element	ElementAt, ElementAtOrDefault, First, FirstOrDefault,LastOrDefault, Single, SingleOrDefault
Generation	DefaultIfEmpty, Empty, Range, Repeat
Quantifiers	All, Any, Contains
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum

Book.cs

```
using System;
namespace linqdemo
{
  class Book
  {
   public string Title { get; set; }
   public string Author { get; set; }
   public int Price { get; set; }
   public static Book[] GetBooks()   {
    return new Book[] {
     new Book{Title="C# 4.0", Author="Anders", Price = 550},
     new Book{Title="Oracle 11g",Author="Jason Price",Price=650},
```

```
     new Book { Title="Asp.Net 4.0 Unleahsed",  Author ="Walther", Price = 799},
     new Book { Title="ASP.NET AJAX In Action", Author ="Craig Walls", Price = 500},
     new Book { Title="Introduction To Microsoft ASP.NET AJAX",
                Author="Dino",Price=550}
    };
   }
  }  // end of Book class
}
```

QueryOperatorsDemo.cs

```
using System;
using System.Linq;
namespace linqdemo
{
  class QueryOperatorsDemo
  {
   static void Main()
   {
    Book[] books = Book.GetBooks();

    // take books where price is more than 500
    var selectedbooks = books.Where( b => b.Price > 500);

    foreach (var book in selectedbooks)
     Console.WriteLine("Title:{0},Price:{1}",
                                book.Title, book.Price);

    // skip first 2 rows and then take 2 rows
    var first10 = books.Skip(2).Take(2); // 2 books from 3rd book

    //sort books on desc order of price then by asc order of title
    var sortedbooks = books.OrderByDescending(b => b.Price)
                     .OrderBy(b => b.Title);

    foreach (var book in sortedbooks)
       Console.WriteLine("Title:{0},Price:{1}",book.Title, book.Price);

    // get maximum price
    var maxprice = books.Max(b => b.Price);

    // get first book with maximum price
    var mp_book = books.Where(b => b.Price == maxprice).
                   Select(b => new { b.Title, b.Price }).First();

    Console.WriteLine("{0} with price {1} is the highest priced",
                  mp_book.Title, mp_book.Price);
   }
 }
}
```

Query Expressions

- A query expression is convenient declarative shorthand for code you could write manually.
- Query expressions allow us to use the power of query operators, but with a query-oriented syntax.
- A query expression operates on one or more information sources by applying one or more query operators
- LINQ queries built with query expressions resemble SQL queries

Clause	Meaning
from	Specifies a data source and a range variable (similar to an iteration variable).
where	Filters source elements based on one or more Boolean expressions separated by logical AND and OR operators (&& or ||).
select	Specifies the type and shape that the elements in the returned sequence will have when the

	query is executed.
group	Groups query results according to a specified key value.
into	Provides an identifier that can serve as a reference to the results of a join, group or select clause.
orderby	Sorts query results in ascending or descending order based on the default comparer for the element type.
join	Joins two data sources based on an equality comparison between two specified matching criteria.
let	Introduces a range variable to store sub-expression results in a query expression.

The following table shows query operators (method) and corresponding clause in query expressions.

Query operator	C# syntax
GroupBy	group … by
GroupJoin	join … in … on … equals … into…
Join	join … in … on … equals …
OrderBy	Orderby
OrderByDescending	orderby … descending
Select	select
SelectMany	Multiple from clauses
ThenBy	orderby …, …
ThenByDescending	orderby …, … descending
Where	Where

QueryExpressionsDemo.cs

```
using System;
using System.Linq;
namespace linqdemo
{
    class QueryExpressionsDemo
    {
     static void Main()
     {
       Book[] books = Book.GetBooks();

       var selectedbooks = from book in books
                           where book.Price > 500
                           select book;

       foreach (var book in selectedbooks)
         Console.WriteLine("Title : {0}  Price : {1}",
                             book.Title, book.Price);

       // query expression
       var sortedbooks = from book in books
                         orderby book.Price descending
                         orderby book.Title
                         select book;

       foreach (var book in sortedbooks)
         Console.WriteLine("Title : {0}  Price : {1}",
                            book.Title, book.Price);

       var maxprice = books.Max(b => b.Price);

       var mp_book = (from book in books
                      where book.Price == maxprice
                      select new { book.Title, book.Price }
                      ).First();

       Console.WriteLine("{0} book has the highest price {1}",
                         mp_book.Title, mp_book.Price);
```

```
        }
    }
}
```