Generics, Collection and Events

Course Code: CSC 2210 Course Title: Object Oriented Programming 2

Dept. of Computer Science
Faculty of Science and Technology

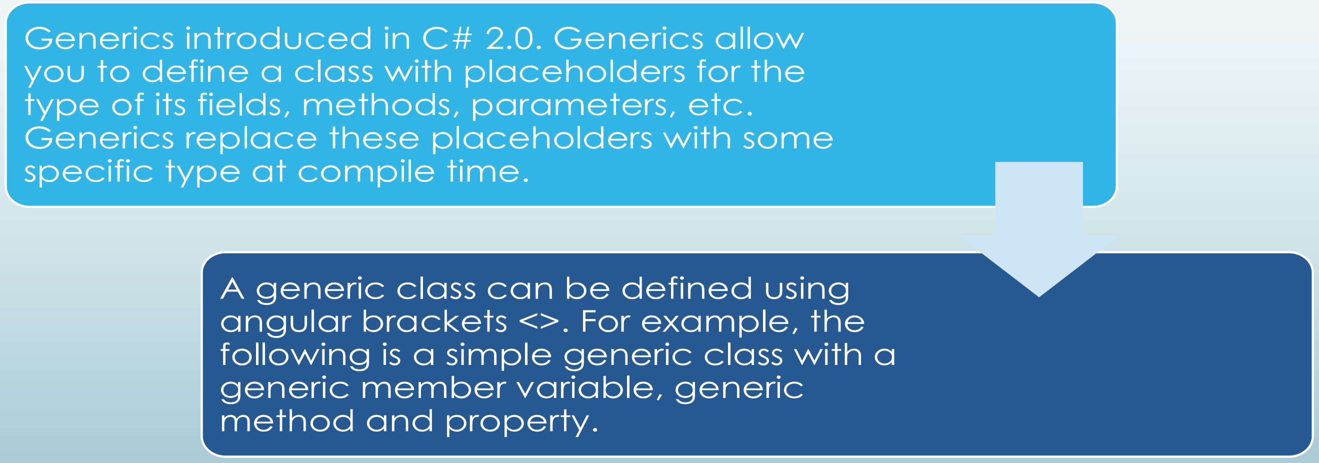
Lecturer No:	09	Week No:	10	Semester:	
Lecturer:					

Topics

1. Generics
2. Collection
3. Events

Generics

Introduction to Generics



Generics introduced in C# 2.0. Generics allow you to define a class with placeholders for the type of its fields, methods, parameters, etc. Generics replace these placeholders with some specific type at compile time.

A generic class can be defined using angular brackets <>. For example, the following is a simple generic class with a generic member variable, generic method and property.

Benefits

- Increases the reusability of the code.
- Generic are type safe. You get compile time errors if you try to use a different type of data than the one specified in the definition.
- Generic has a performance advantage because it removes the possibilities of boxing and unboxing.

```
LinkedList<string> llist = new LinkedList<string>();
```

Generic Class

```
class MyGenericClass<T>
{
private T genericMemberVariable;
public MyGenericClass(T value)
{
genericMemberVariable = value;
}
public T genericMethod(T genericParameter)
 {
Console.WriteLine("Parameter type: {0}, value: {1}",
typeof(T).ToString(),genericParameter);
Console.WriteLine("Return type: {0}, value: {1}", typeof(T).ToString(),
genericMemberVariable);
return genericMemberVariable;
}
public T genericProperty
{
get; set;
}
}
```

Generic Interfaces

- The preference for generic classes is to use generic interfaces, such as IComparable<T> rather than IComparable, in order to avoid boxing and unboxing operations on value types.

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T> { }
interface IDictionary<K, V> { }
interface IMonth<T> { }
interface IJanuary : IMonth<int> { } //No error
interface IFebruary<T> : IMonth<int> { } //No error
interface IMarch<T> : IMonth<T> { } //No error
//interface IApril<T> : IMonth<T, U> {} //Error
```

Generic Delegates

- The delegate defines the signature of the method which it can invoke. A generic delegate can be defined the same way as delegate but with generic type.
- A generic delegate can point to methods with different parameter types. However, the number of parameters should be the same.

```
class Program
{
     public delegate T add<T>(T param1, T param2);
     static void Main(string[] args)
     {
         add<int> sum = AddNumber;
         Console.WriteLine(sum(10, 20));
         add<string> conct = Concate;
         Console.WriteLine(conct("Hello","World!!"));
    }
    public static int AddNumber(int val1, int val2)
    {
          return val1 + val2;
     }
     public static string Concate(string str1, string str2)
     {
          return str1 + str2;
     }
}
```

Points to Remember

1. Generics denotes with angular bracket <>.
2. Compiler applies specified type for generics at compile time.
3. Generics can be applied to interface, abstract class, method, static method, property, event, delegate and operator.
4. Generics performs faster by not doing boxing & unboxing.

Collections

Array

System.Collections

Hashtables

Stack, Queue

SortedList

Collection Interfaces

System.Collections.Generic

List<T>

Array

- Array is a data structure that contains several variables of the same type.

```
        type [ ]   arrayName;
```

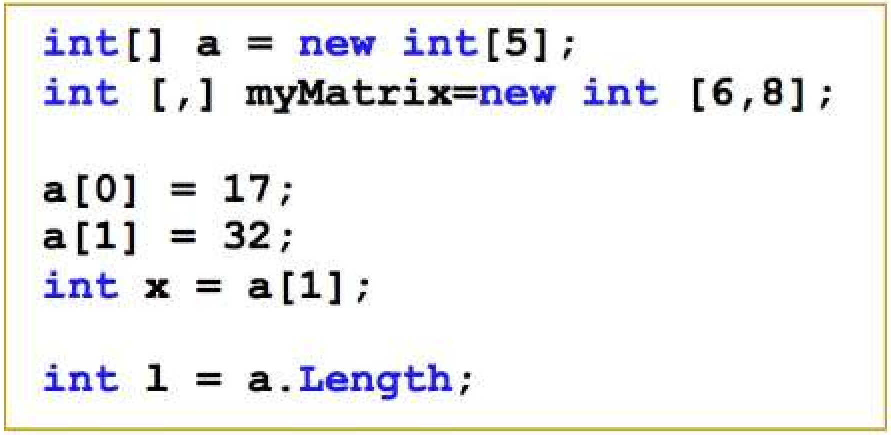
- Array has the following properties:
  - array can be **Single-dimensional**, **Multidimensional** or **Jagged**.
  - The default value of **numeric** array elements are set to **zero**, and **reference** elements are set to **null**.
  - Arrays are **zero indexed**: an array with **n** elements is indexed from **0** to **n-1**.
  - Array elements can be of **any type**, including an array type.
- Array types are reference types derived from the abstract base type **Array**. It implements **IEnumerable** and **IEnumerable<(Of <(T>)>)**, for using in **foreach**

Array. Examples

create

element access

number of elements

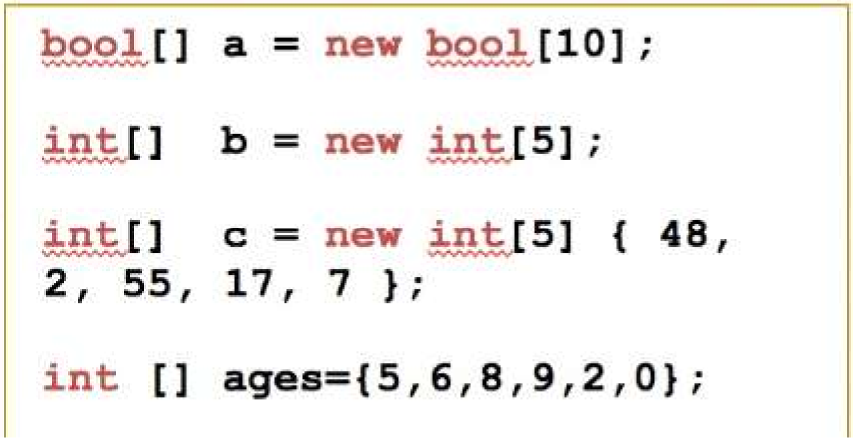


```
int[] a = new int[5];
int [,] myMatrix=new int [6,8];

a[0] = 17;
a[1] = 32;
int x = a[1];

int l = a.Length;
```

default to `false`

default to `0`

set to given values



```
bool[] a = new bool[10];

int[]  b = new int[5];

int[]  c = new int[5] { 48,
2, 55, 17, 7 };

int [] ages={5,6,8,9,2,0};
```

Array. Examples

- Multidimensional arrays:

```
string [ , ] names = new string[5,4];
```

- Array-of-arrays (jagged):

```
  byte [ ][ ] scores = new byte[ 5 ][ ];
  for ( int i = 0; i < scores.Length; i++)
  {
     scores[i] = new byte[4];
}
```

- Three-dimensional rectangular array:

```
int [  ,  ,  ] buttons = new int [ 4, 5, 3];
```

Array. Benefits. Limitations

- **Benefits of Arrays**:
  - **Easy** to use: arrays are used in almost every programming language
  - **Fast** to change **elements.**
  - **Fast** to **move** through elements: Because an array is stored continuously in memory, it's **quick** and easy to cycle through the elements one-by-one from start to finish in a loop.
  - You can specify the type of the elements: When you create an array, you can **define** the **datatype**.
- **Limitations of Arrays**:
  - **Fixed size**: Once you have created an array, it will not automatically items onto the end.
  - **Inserting** elements mid-way into a filled array is difficult.

System.Collections. ArrayList

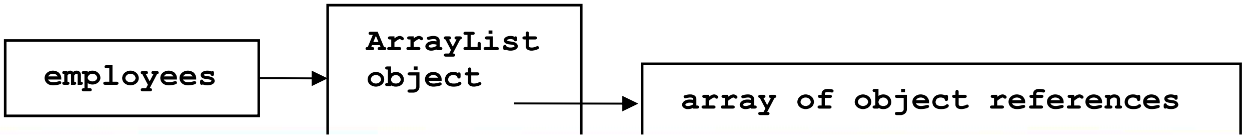
- System.Collections namespace
- ArrayList, HashTable, SortedList, Queue, Stack:
  - A collection can contain an **unspecified** number of members.
  - Elements of a collection do not have to share the same **datatype**.
  - An object's **position** in a collection can **change** whenever a change occurs in the whole, herefore, the position of a specific object in the collection can vary.

ArrayList

- ArrayList is a **special array** that provides us with some functionality over and above that of the standard Array.
- We can dynamically resize it by simply adding and removing elements.

create ArrayList to store Employees →

```
using System.Collections;

class Department
{
  ArrayList employees = new ArrayList();
  ...
}
```



employees → ArrayList object → array of object references

ArrayList services

add new elements →

remove →

containment testing →

read/write existing element →

control of memory in underlying array →

```
public class ArrayList : IList, ICloneable
{
  int  Add    (object value)  // at the end
  void Insert(int index, object value) ...

  void Remove   (object value) ...
  void RemoveAt(int     index) ...
  void Clear    () ...

  bool Contains(object value) ...
  int  IndexOf (object value) ...

  object this[int index] { get... set.. }

  int  Capacity { get... set... }
  void TrimToSize() //minimize memory
  ...
}
```

ArrayList. Benefits and Limitation

- **Benefits** of `ArrayList:`
  - Supports automatic **resizing.**
  - **Inserts** elements: An ArrayList starts with a collection containing no elements.
  - Flexibility when **removing elements**.
  - **Easy** to use.
- **Limitation** of `ArrayLists:`
  - There is **one major limitation** to an `ArrayList:` **speed**.
  - The flexibility of an `ArrayList` comes at a cost, and since memory allocation is a very expensive business the fixed structure of the simple array makes it a lot faster to work with.

Stack

- `Stack`: last-in-first-out

create `Stack` to store sequence of method calls →

```
using System.Collections;

class Trace
{
   Stack callChain = new Stack();
   ...
}
```

add →

examine →

remove →

```
Stack s = new Stack();

s.Push("aaa");
s.Push("bbb");

string t = (string)s.Peek();

string u = (string)s.Pop();
```

Queue

- Queue: first-in-first-out

create Queue to store events

```
using System.Collections;

class Watcher
{
   Queue events = new Queue();
   ...
}
```

add

examine

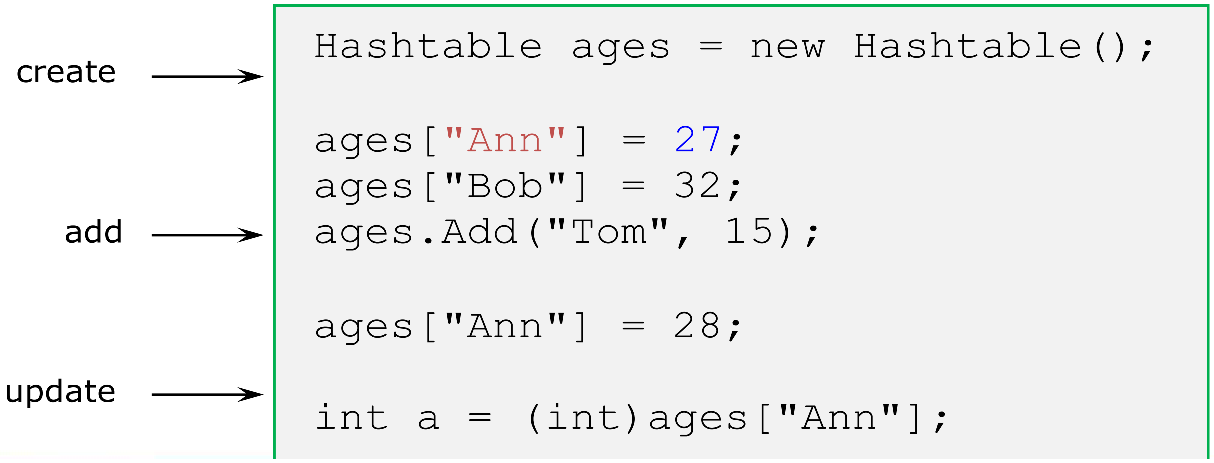
remove

```
Queue q = new Queue();

q.Enqueue("aaa");
q.Enqueue("bbb");
q.Enqueue("ccc");

string s = (string)q.Peek();

string t = (string)q.Dequeue();
```

Hashtable

- Represents a collection of **key/value pairs** that are organized based on the hash code of the key.
- The objects used as keys must override the **GetHashCode** method and the **Equals** method.

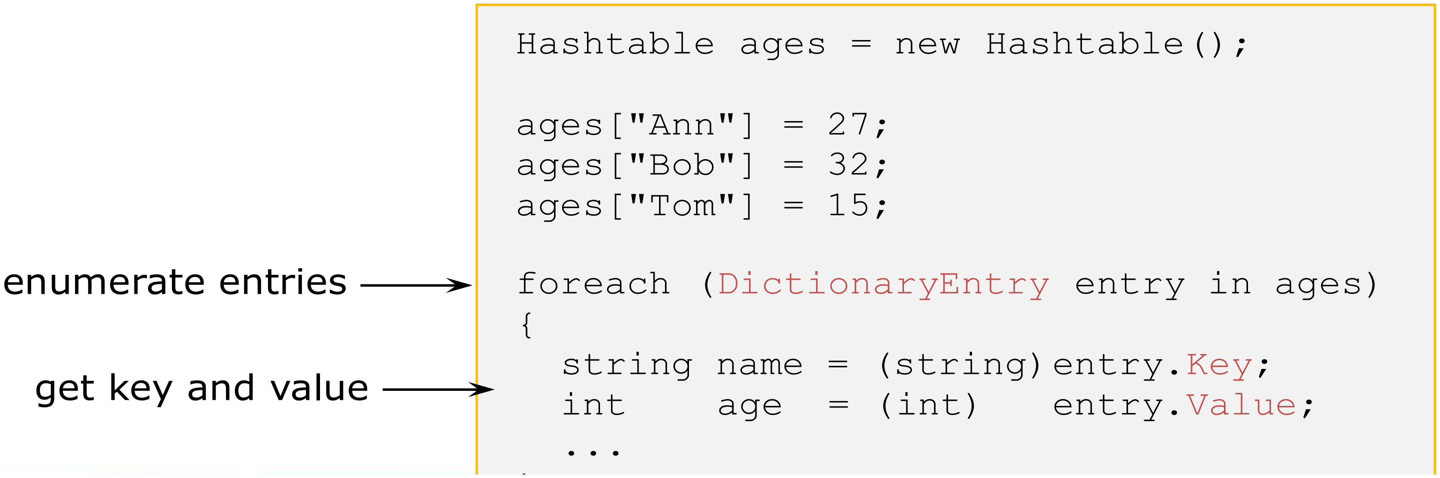
- **Benefits** of Hashtable:
  - **Non-numeric indexes** allowed. **Key** can be numeric, textual, or even in form of a date. But can't be null reference.
  - Easy **inserting** elements.
  - Easy **removing** elements.
  - Fast **lookup**.



```
Hashtable ages = new Hashtable();

ages["Ann"] = 27;
ages["Bob"] = 32;
ages.Add("Tom", 15);

ages["Ann"] = 28;

int a = (int)ages["Ann"];
```

Hashtable

- **Limitations** of Hashtable:
  - **Performance** and **speed**: `Hashtable` objects are **slower to update** but **faster to use** in a look-up than `ArrayList` objects.
  - **Keys** must be **unique**: An array automatically keeps the index values unique. In a `Hastable` we must monitor the key uniqueness.
  - No useful **sorting**: The items in a `Hashtable` are **sorted internally** to make it easy to find objects very quickly. It's not done by keys or values, the items may as well not be sorted at all.



```
Hashtable ages = new Hashtable();

ages["Ann"] = 27;
ages["Bob"] = 32;
ages["Tom"] = 15;

foreach (DictionaryEntry entry in ages)
{
  string name = (string)entry.Key;
  int    age  = (int)   entry.Value;
  ...
```

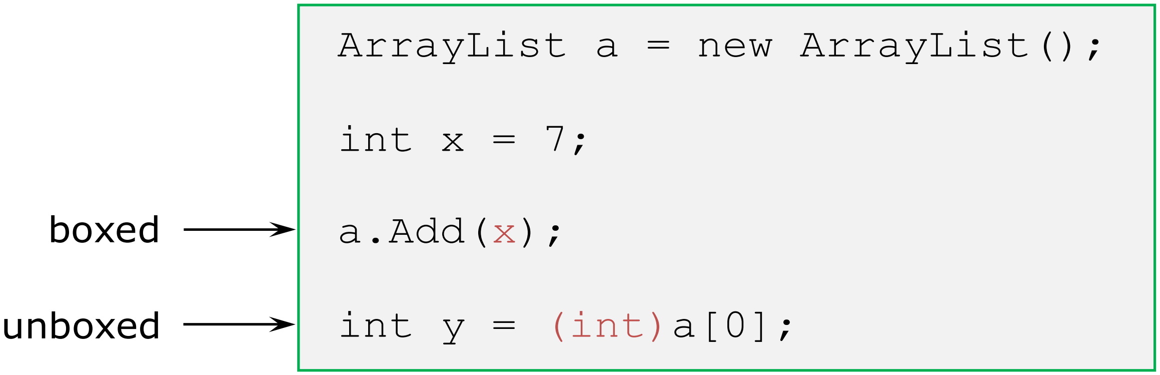
SortedList

- Represents a collection of **key/value pairs** that are **sorted** by the keys
- Are accessible by **key** and by **index**.
- A SortedList object internally maintains two arrays to store the elements of the list
- Use the new keyword when creating the object. Each adding item is automatically inserted in the correct position in the list, according to a specific IComparer implementation .

```
SortedList stlShippers = new SortedList();

stlShippers["cp"]="Canada Post";

stlShippers["fe"]="Federal Express";

stlShippers["us"]="United State Postal Service";

foreach (DictionaryEntry de in stlShippers)

{

    Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);

}
```

Collections. Drawbacks

- No type checking enforcement at compile time
  - Doesn’t prevent adding unwanted types
  - Can lead to difficult to troubleshoot issues
  - Runtime errors!
- All items are stored as objects
  - Must be cast going in and coming out
  - Performance overhead of boxing and unboxing specific types



```
ArrayList a = new ArrayList();

int x = 7;

a.Add(x);

int y = (int)a[0];
```

System.Collections.Generic

- **Open constructed types**
  - Classes defined without a specific type
  - Type is specified when instantiated
- **Provides type safety at compile time**

System.Collections.Generic	System.Collections
List<T>	ArrayList
Dictionary<K,T>	HashTable
SortedList<K,T>, SortedDictionary<K,T>	SortedList
Stack<T>	Stack
Queue<T>	Queue
LinkedList<T> O(1)	-
IList<T>	IList
IDictionary<K,T>	IDictionary
ICollection<T>	ICollection
IEnumerator<T>	IEnumerator
IEnumerable<T>	IEnumerable
IComparer<T>	IComparer
IComparable<T>	IComparable

List<T>

- **List generic** class:

```
[SerializableAttribute]
public class List<T> : IList<T>, ICollection<T>,
          IEnumerable<T>, IList, ICollection, Ienumerable
```

- The **List class** is the generic **equivalent** of the **ArrayList** class. It implements the `IList` generic interface using an array whose size is dynamically increased as required.
- The List class **uses** both an equality comparer and an ordering comparer.
- Methods such as **Contains**, **IndexOf**, **LastIndexOf**, and **Remove** use an equality comparer for the list elements.
- If type **T** implements the `IEquatable` generic interface, then the equality comparer **is the Equals method** of that interface; otherwise, the default equality comparer is `Object.Equals(Object).`

List<T>

- Methods such as **BinarySearch** and **Sort** use an ordering comparer for the list elements.
- The List is not guaranteed to be sorted. You must sort the List before performing operations (such as BinarySearch) that require the List to be sorted.
- Elements in this collection can be accessed using an integer index. Indexes in this collection are zero-based.
- List accepts a **null** reference **as a valid** value for reference types and allows duplicate elements.

Delegates and Events

Motivation – Function Pointers

- Treat functions as first-class objects
  - eg. Scheme: `(map myfunc somelist)`
    - works because functions are lists
  - eg. Mathematica:
    `{#, #^2} & /@ Range[1,10]`
  - eg. C/C++:

    ```
    typedef int (*fptr)(int*);
    int my_method(int* var) { if (var != NULL) return
    *var; }
    fptr some_method = my_method;
    int take_f_param(fptr g) { int x = 10; int y = &x;
    return g(y); }
    printf(“%d\n”, take_f_param(some_method));
    ```

    - Works because functions are pointers to code

Motivation – Function Pointers

- Java
  - no equivalent way to get function pointers
  - use inner classes that contain methods
  - or simply use interfaces
- Why not?
  - functions are not objects
  - same problem as integers

Comparators

- Sort method on many containers
  - provides efficient sorting
  - needs to be able to compare to objects
- Solution: IComparer

```
public class ArrivalComparer: IComparer {
    public ArrivalComparer() {}
    public int Compare(object x, object y) {
        return ((Process)x).Arrival.CompareTo(((Process)y).Arrival);
    }
}
```

- Can then call
  - `sortedList.Sort(new ArrivalComparer());`

Delegates

- An objectified function
  - inherits from System.Delegate
  - sealed implicitly
  - looks much like C/C++ style function pointer
- eg. `delegate int Func(ref int x)`
  - defines a new type: Func: takes int, returns int
  - declared like a function with an extra keyword
  - stores a list of methods to call

Delegates – Example

```
delegate int Func(ref int x);
int Increment(ref int x) { return x++; }
int Decrement(ref int x) { return x--; }
Func F1 = new Func(Increment);
F1 += Decrement;
x = 10;
Console.WriteLine(F1(ref x));
Console.WriteLine(x);
```

- Delegate calls methods in order
  - ref values updated between calls
  - return value is the value of the last call

Delegates – Usage Patterns

- Declared like a function
- Instantiated like a reference type
  - takes a method parameter in the constructor
- Modified with +, -, +=, -=
  - can add more than one instance of a method
  - - removes the last instance of the method in the list
- Called like a function: functional programming

Delegates – Usage Examples

```
delegate int Func(int x);
List<int> Map(Func d, List<int> l) {
      List<int> newL = new List<int>();
     foreach(int i in l) {
         newL.Add(d(l));
     }
     return newL;
}
```

- Allows code written in a more functional style

Notes on Delegates

- `this` pointer captured by instance delegates
  - thus can capture temporary objects from method calls or elsewhere in delegates
  - eg.

    ```
    delegate int Func(int x);
    Func f = new Func();
    …
    { TempObject o = new TempObject();
    f += o.m; } // o is now out of scope
    ```

Covariance & Contravariance

- If the type of the return value is subclass
  - then the delegate is still acceptable
  - called **covariance**
- If the type of the args are subclasses
  - then the delegate is likewise acceptable
  - called **contravariance**

Anonymous Methods

```
Func f = new Func();
int y = 10;
f += delegate(int x) { return x + y; }
```

- Creates a method and adds it to delegate
  - treated the same as other methods
  - good for one-time, short delegates
- Variables captured by anonymous method
  - **outer variables**
  - like `y` in the above example

Anonymous Methods

- `using System;`
- `delegate void D();`

-
  ```
  class Test
  {
      static D[] F() {
            D[] result = new D[3];
            for (int i = 0; i < 3; i++) {
                   int x = i * 2 + 1;
                   result[i] = delegate
  { Console.WriteLine(x); };
            }
            return result;
      }
  ```

-
  ```
      static void Main() {
            foreach (D d in F()) d();
      }
  }
  ```

Anonymous Methods

-
  ```
  static D[] F() {
      D[] result = new D[3];
      int x;
      for (int i = 0; i < 3; i++) {
             x = i * 2 + 1;
             result[i] = delegate { Console.WriteLine(x); };
      }
      return result;
  }
  ```
- First returns 1,3,5. Second returns 5,5,5
  - Outer variables are captured locations
  - Not given values at delegate creation time
- Can communicate through outer variables

Defining and using Delegates

- three steps:
  1. Declaration
  2. Instantiation
  3. Invocation

Delegate Declaration

```
namespace some_namespace
{
   delegate void MyDelegate(int x, int y);


   class MyClass
  {
      public delegate int AnotherDelegate(object o, int y);
   }
}
```

Delegate Type Name

Delegate Instantiation

```
delegate void MyDelegate(int x, int y);

class MyClass
{
   private MyDelegate myDelegate = new MyDelegate( SomeFun );

   public static void SomeFun(int dx, int dy)
   {
   }
}
```

Invocation Method

Invocation Method name (no params or perens)

More Realistically

```
class MyClass
{
   private MyDelegate myDelegate;

   public MyDelegate MyDelegate
   {
      set { this.myDelegate = value; }
   }
}

class MainClass
{
   [STAThread]
   static void Main()
   {
      MyClass mc = new MyClass();
      mc.MyDelegate = new MyDelegate( MainClassMethod );
   }

   private static void MainClassMethod(int x, int y) {   }
}
```

Delegate Variable (null)

Passed in and assigned in the constructor

Instance Methods

```
class MyClass
{
   private MyDelegate myDelegate;

   public MyDelegate MyDelegate
   {
      set { this.MyDelegate = value; }
   }
}

class MainClass
{
   [STAThread]
   static void Main()
   {
      MainClass mainClass = new MainClass();
      MyClass mc = new MyClass();
      mc.MyDelegate = new MyDelegate( mainClass.MainClassMethod );
   }

   private void MainClassMethod(int x, int y) {   }
}
```

Method attached to an instance

Delegate-Method Compatibility

- A Method is compatible with a Delegate if
  - They have the same parameters
  - They have the same return type

Delegate Invocation

```
class MyClass
{
   private MyDelegate myDelegate;

   public MyClass(MyDelegate myDelegate)
   {
      this.MyDelegate = myDelegate;
   }

   private void WorkerMethod()
   {
      int x = 500, y = 1450;

      if(myDelegate != null)
         myDelegate(x, y);
   }
}
```

Attempting to invoke a delegate instance whose value is null results in an exception of type *System.NullReferenceException*.

Delegate’s “Multicast” Nature

- Delegate is really an array of function pointers

```
mc.MyDelegate += new MyDelegate( mc.Method1 );
mc.MyDelegate += new MyDelegate( mc.Method2 );
mc.MyDelegate = mc.MyDelegate + new MyDelegate( mc.Method3 );
```

- Now when Invoked, mc.MyDelegate will execute all three Methods

- Notice that you don’t have to instantiate the delegate before using +=
  - The compiler does it for you when calling +=

The Invocation List

- Methods are executed in the order they are added
- Add methods with + and +=
- Remove methods with - and -=
  - Attempting to remove a method that does not exist is not an error
- Return value is whatever the last method returns
- A delegate may be present in the invocation list more than once
  - The delegate is executed as many times as it appears (in the appropriate order)
  - Removing a delegate that is present more than once removes only the last occurrence

Multicast example

```
mc.MyDelegate = new MyDelegate( mc.Method1 );
mc.MyDelegate += new MyDelegate( mc.Method2 );
mc.MyDelegate = mc.MyDelegate + new MyDelegate( mc.Method3 );


// The call to:
mc.MyDelegate(0, 0);
// executes:

// mc.Method1
// mc.Method2
// mc.Method3
```

System.Delegate

- abstract
- Special
  - Only the system and compiler can inherit from *System.Delegate*
- Members:
  - Method
    - Returns A MethodInfo object
  - Target
    - The ‘this’ pointer belonging to the method
    - null if the method was static
  - operator == and operator !=

System.MulticastDelegate

- Target
- Method
- operator== and operator !=
- _prev
  - Linked list of *System.Delegate* objects
- GetInvocationList()
  - Returns the Invocation List as a Delegate[]

Events

- Events enable a class or object to notify other classes or objects when something of interest occurs.

- The class that sends (or raises) the event is called the publisher and the classes that receive (or handle) the event are called subscribers.

- The publisher determines when an event is raised.

- The subscribers determine what action is taken in response to the event.

  - Creating a Event:
    - public event EventHandler<ClassName> Event-Name

      ```
      {
         add{//………}
         remove{//…………}
      }
      ```

Events

- Events are “safe” delegates
  - But they are delegates
- Restricts use of the delegate (event) to the target of a += or -= operation
  - No assignment
  - No invocation
  - No access of delegate members (like GetInvocation List)
- Allow for their own Exposure
  - Event Accessors

Events

- Special class of delegates
  - given the `event` keyword
  -
    ```
    class Room {
        public event EventHandler Enter;
        public void RegisterGuest(object source,
    EventArgs e) { … }
        public static void Main(string[] args) {
            Enter += new EventHandler(RegisterGuest);
            if (Enter != null) {
                Enter(this, new EventArgs());
            }
        }
    }
    ```

Events

- `Enter` is an object of type delegate
  - when event is “raised” each delegate called
  - C# allows any delegate to be attached to an event
  - .NET requires only EventHandlers
    - needed for CLS compatibility
- Adds restrictions to the delegate
  - can only raise an event in its defining class
  - outside, can only do += and -= : return void

Events

- Modifiers on events
  - public/private: define accessibility of += and -=
- Delegates cannot be defined in interfaces
  - events can be defined in interfaces
- Since can only do += and -= outside, how do we raise events?
  - normally with methods: eg. Button.OnClick
  - sole purpose is to raise the event

Events – Accessors

- `add` and `remove` accessors
  - can be explicitly defined for events
  - use anonymous methods
  - normally generated by compiler
- For example
  - when want to control the space used for storage
  - access the custom data structure in OnClick()
  - or use to control accessibility

Event Accessors

```
public delegate void FireThisEvent();
class MyEventWrapper
{
   private event FireThisEvent fireThisEvent;

   public void OnSomethingHappens()
   {
      if(fireThisEvent != null)
         fireThisEvent();
   }

   public event FireThisEvent FireThisEvent
   {
      add { fireThisEvent += value; }
      remove { fireThisEvent -= value; }
   }
}
```

add and remove keywords

Events - Uses

- Registering callbacks
  - common programming paradigm
  - examples in OS and threaded code
  - any asynchronous code does this
- Windowing code
  - eg. Button.OnClick
  - basis of Windows.Forms
  - Handles the asynchrony of the user

Event-Based Programming

- Style of concurrent programming
  - contrasts with thread based
  - concurrency based on the number of events
  - not on the number of processors
    - although limited by number of processors
- events in C# could be backed by an event-based system
  - full support for events already in language

Generics and Delegates

- Delegates can use generic parameters:

  ```
  delegate int Func<T>(T t)
  ```

  - allows interaction with generic classes

    ```
    class Test<T> {
        public Test(Func<T> f, T val) { … }
    ```

  - Is the type Func<T> open or closed?
- Methods can use delegates similarly
- Both can add where constraints like classes

Generics and Delegates

- Delegates can use generic parameters:

  ```
  delegate int Func<T>(T t)
  ```

  - allows interaction with generic classes

    ```
    class Test<T> {
        public Test(Func<T> f, T val) { … }
    ```

  - Is the type Func<T> open or closed?
- Methods can use delegates similarly
- Both can add where constraints like classes

Generics and Delegates

- Does the following work?

```
delegate int F<T>(int x);
class Test2<T,U>
    where T : class
    where U : F<T> { … }
```

  - no: type constraints cannot be sealed classes
  - can, however, change F<T> to System.Delegate
    - then we lose some ability to statically typecheck

Thank You

Books

- C# 4.0 The Complete Reference; Herbert Schildt; McGraw-Hill Osborne Media; 2010
- Head First C# by Andrew Stellman
- Fundamentals of Computer Programming with CSharp – Nakov v2013

References

MSDN Library; URL: http://msdn.microsoft.com/library

C# Language Specification; URL: http://download.microsoft.com/download/0/B/D/0BDA894F-2CCD-4C2C-B5A7-4EB1171962E5/CSharp%20Language%20Specixfication.doc

C# 4.0 The Complete Reference; Herbert Schildt; McGraw-Hill Osborne Media; 2010