# Generics

# Generics

- Generics considered as *template* with *placeholder* for type

```
static void Swap (ref int x, ref int y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```

```
static void Swap (ref char x, ref char y)
{
        char temp;
        temp = x;
        x = y;
        y = temp;
}
```

# Generic Method

- Definition

```
static void Swap <T> (ref T x, ref T y)
{
        T temp;
        temp = x;
        x = y;
        y = temp;
}
```

- Calling

```
Swap <char> (ref x, ref y);
```

```
Swap (ref x, ref y);
```

```
Swap <int> (ref x, ref y);
```

# Default generic value

- *default(T)*
  - Ex : return of pop Method

```
public T pop()
{
  if (tos > 0)
  {
    tos--;
    return stk[tos];
  }
  else
    return default(T);
}
```

```
public int pop()
{
  if (tos > 0)
  {
   tos--;
   return stk[tos];
  }
  else
    return -1;
}
```

# Generic Class

- Definition

Generic type

```
public class Demo <T>
{
        public T v;
        public Demo(T x )
        { v=x;}
}
```

```
public class Pair <T,U>
{
        public T v1;
        public U v2;
        public Pair(T x,U y )
        { v1=x; v2=y; }
}
```

- Declare Reference and Instantiating an Object

Constructed type

```
Demo<int> D=new Demo<int>(10);
```

```
Pair<int,string> D2=new Demo <int,string>(10,"Hi");
```

# Generic Interface

- Definition

```
public interface IGenInteface <T>
{
  T Prperty { get; set; }
}
```

# Generic Constraint

- Arithmetic operation Constraint

```
class Complex<T>
{
    public T real;
    public T img;
    public Complex()
    {
        real = img = default;
    }
    public static Complex<T> operator +(Complex<T> c1, Complex<T> c2)
    {
        Complex<T> c = new Complex<T>();
        c.real = c1.real + c2.real; // Error cant apply operator + for T and T
    }
}
```

# Generic Type Constraint

- Constraint on T could be achieve using *where* statement

```
GenericTypeName<T> where T   : contraint1, constraint2
```

```
class GenericClass<T, U> where T : class1, Interface1
                         where U : new()
    {...}
```

# Generic Type Constraint

| | |
|---|---|
| class | The type argument must be any class, interface, delegate, or array type. |
| class? | The type argument must be a nullable or non-nullable class, interface, delegate, or array type. |
| struct | The type argument must be non-nullable value types such as primitive data types int, char, bool, float, etc. |
| new() | The type argument must be a reference type which has a public parameterless constructor. It cannot be combined with struct and unmanaged constraints. |
| notnull | Available C# 8.0 onwards. The type argument can be non-nullable reference types or value types. If not, then the compiler generates a warning instead of an error. |
| unmanaged | The type argument must be non-nullable unmanged types. |

# Generic Type Constraint

| | |
|---|---|
| base class name | The type argument must be or derive from the specified base class. The Object, Array, ValueType classes are disallowed as a base class constraint. The Enum, Delegate, MulticastDelegate are disallowed as base class constraint before C# 7.3. |
| <base  class name>? | The type argument must be or derive from the specified nullable or non-nullable base class |
| <interface name> | The type argument must be or implement the specified interface. |
| <interface name>? | The type argument must be or implement the specified interface. It may be a nullable reference type, a non-nullable reference type, or a value type |
| where T: U | The type argument supplied for T must be or derive from the argument supplied for U. |

# Generic Type Constraint

| | |
|---|---|
| INumber<T> | The type argument must be numeric type |
| IBinaryInteger<T> | The type argument must be integer |

# Generic and Inheritance

- Inheriting generic types

```
public class GenStack <T>
{
        public T  [ ] stk;
        public int  size;
}
```

```
class specialStack <T>:Genstack<T>
{
   …
}
```

```
class specialStack:Genstack<int>
{
   …
}
```

# Generic and Inheritance

- Implementing Generic Interface

```
public interface IGenInteface <T>
{
  T Prperty { get; set; }
}
```

```
public class GenClass2<T>:
IGenInteface<T>
    {
        T t1;
        public T Prperty
        {
            get
            {
                return t1;
            }
            set
            {
                t1 = value;
            }
        }
    }
```

```
 public class Class3 :
 IGenInteface<int>
    {
        int t2;
        public int Prperty
        {
            get
            {
                return t2;
            }
            set
            {
                t2 = value;
            }
        }
    }
```

HasValue

# Nullable value Type

- A nullable value type allows a variable to contain either a value or null
- Declaration and assignment

```
string s=null;
int z=null; // error
Nullable<int> c=null;
int? k=null;
```

- HasValue property
- Value Property

```
int? b = 10;
if (b.HasValue)
{
  Console.WriteLine($"b is {b.Value}");
}
else
{
  Console.WriteLine("b does not have a value");
}
// Output: "b is 10"
```

# Nullable Reference Type

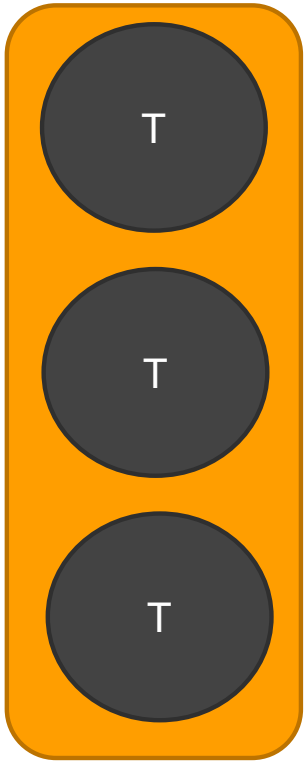
- Eliminate  warning generated to help developer to find potential null reference error (Exception)

```
string s=null; // warning
String? z=null;
```

# Generic Collection

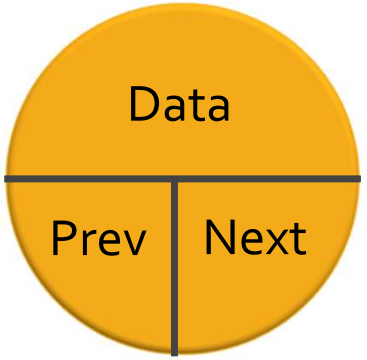
- It ensure Type safe (strongly typed)
- *System.Collections.Generic* namespace
- List<T>
- Stack<T>
- Queue<T>
- SortedList<TKey,Tvalue >
- Dictionary<TKey,TValue>

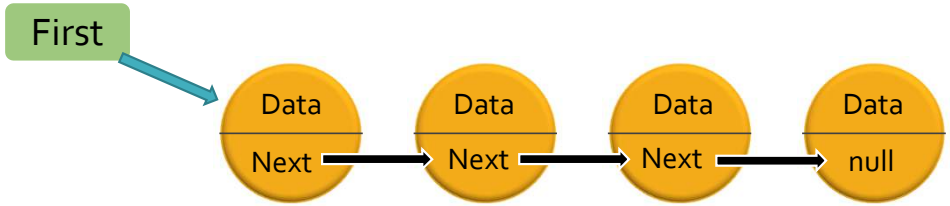
```
List<int> l = new List<int>();
List<employee> empl = new List<employee>();
```



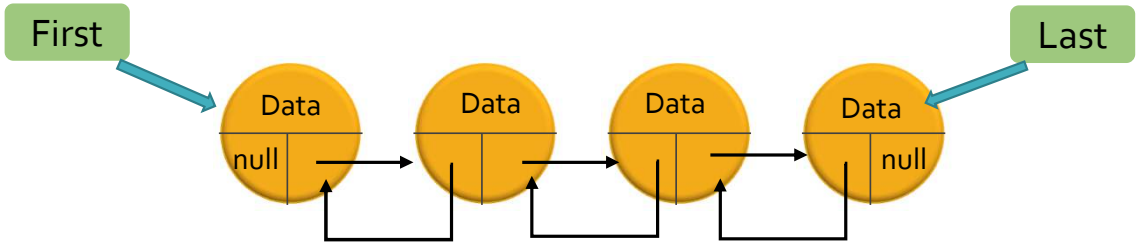
# Generic Collection

- LinkedList



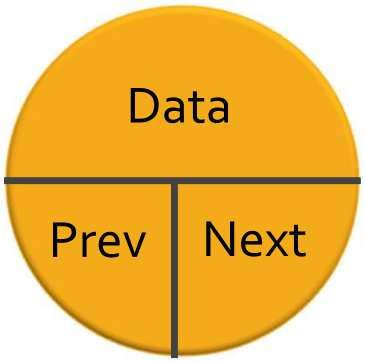

Single Linked List




Double Linked List

# LinkedList

- **LinkedListNode<T>** Class
  - Properties
    - List
    - Next
    - Previous
    - Value
    - ValueRef
  - Methods
    - LinkedListNode (T value)  => constructor



# LinkedList

- LinkedList<T> class
  - Double Linked List
- Properties
  - Count
  - First
  - Last
- Methods
  - AddAfter
  - AddBefore
  - AddFirst
  - AddLast
  - Find
  - FindLast
  - Remove
  - RemoveFirst
  - RemoveLast

# Collection Initializers

```
List<string> l;
l = new List<string> { "Ahmed", "Aly", "Mohamed" };
```

- Dictionary

```
var Numbers2 = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

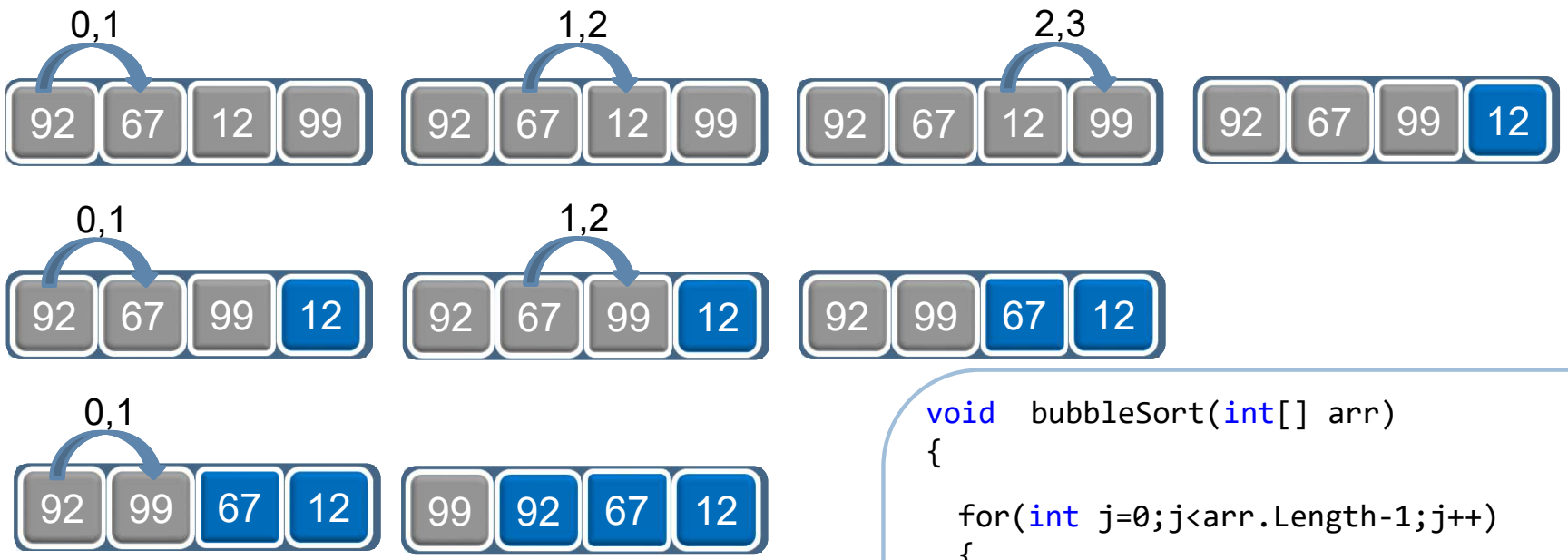
```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

# Assignment

- Write a generic stack class that implmenet Generic interface contain only one Method => T GetByIndex(int index);
- Change *ArrayList* To *List<T>* in menu program
  - Change sort (by name , by ID , by Salary) using Icomparer<T> interface

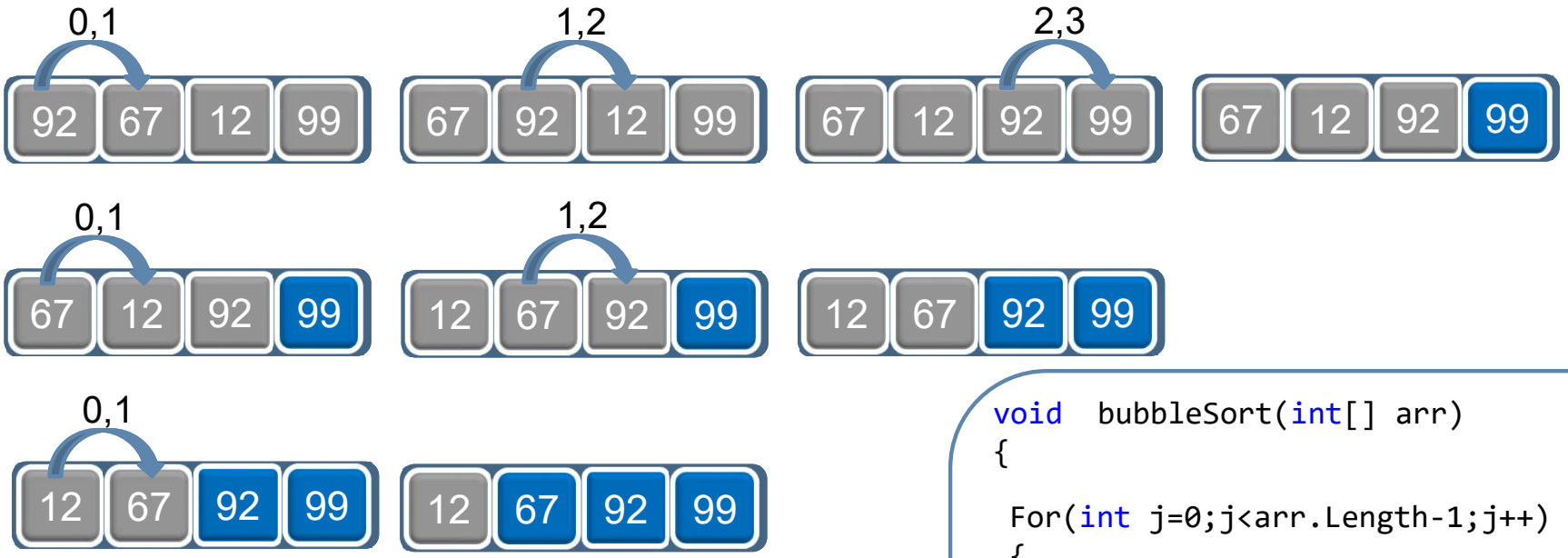
# Delegate and Event

# Bubble Sort Example (descending)




```
void  bubbleSort(int[] arr)
{

  for(int j=0;j<arr.Length-1;j++)
  {
    for(int i=0;i<arr.Length-1 -j ;i++)
    {
      if(arr[i]<arr[i + 1])
      {
       swap(ref arr[i],ref arr[i+1]);
      }
    }
  }
}
```

# Bubble Sort Example (Ascending)



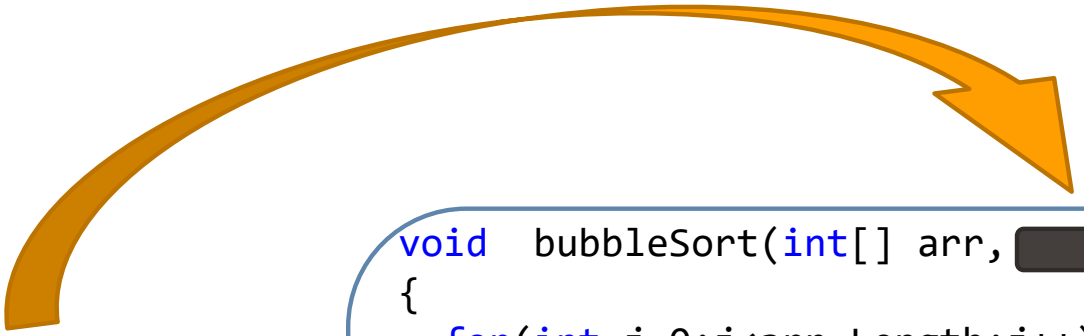

```
void  bubbleSort(int[] arr)
{

 For(int j=0;j<arr.Length-1;j++)
 {
  For(int i=0;i<arr.Length-1  -j  ;i++)
  {
    if(arr[i]>arr[i + 1])
    {
      swap(ref arr[i],ref arr[i+1]);
    }
  }
 }
}
```

# Assignment

- Implement and Trace Bubble Sort

# Delegate

- Why Delegate



```
static bool sortAscending(int L, int M)
{
        return (L > M);
}
```

```
static bool sortDescending(int L, int M)
{
        return (L < M);
}
```

```
void  bubbleSort(int[] arr, [redacted] )
{
  for(int j=0;j<arr.Length;j++)
  {
    for(int i=0;i< arr.Length-1-j;i++)
     {
       if ( [redacted] )
       {
          Swap(ref arr[i],ref arr[i+1]);

       }
     }
  }
}
```

# Delegate

- Delegate is a special data type used as a **reference to method** enables passing method around like any data
- Declare Delegate Data type

```
public delegate bool Mydelegate (int L, int M);
```

delegate keyword
To identify this data type
As a Delegate

Return type of a method
That could be referenced
By a variable of this delegate

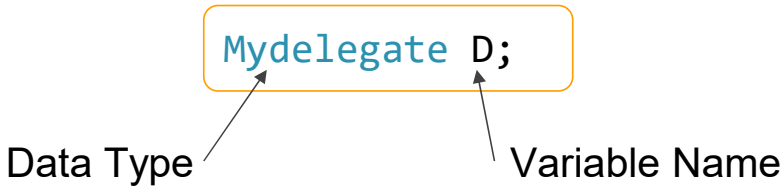
Name of delegate
Data type

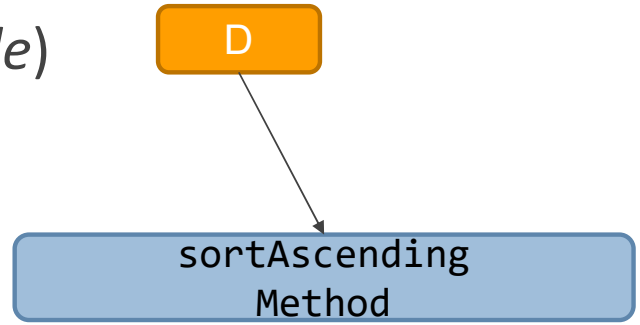
Parameter list of a method
That could be referenced
By a variable of this delegate

Variable of Mydelegate could refer to any method that return bool and takes 2 integers

# Delegate

- Declare delegate variable (*reference variable*)




- Instantiate an object of delegate (initializing delegate variable with value)

```
D = sortAscending;
…
D = sortDescending;
```

```
D = new Mydelegate(sortAscending);
…
D = new Mydelegate(sortDescending);
```

# Delegate

- Passing delegate variable to method

```
bubbleSort(arr, D);
           //or
bubbleSort(arr, sortAscending);
```

Delegate variable

Value
(method name)

```
void bubbleSort(int[] arr, Mydelegate del)
{
…
  if ( del ( arr[i] , arr[i + 1] ) )
  {
   Swap(ref arr[i],ref arr[i+1]);
  }
…
}
```

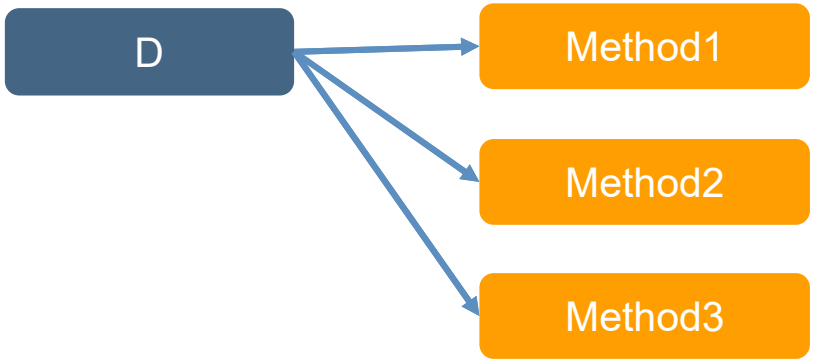
- Invoking delegate

```
D(x,y); //As if it is a method
```

# Delegate

- Multicast Delegate

```
D = Method1;
D += Method2;
D += Method3;
```



# Generic Delegate

```
public static void
swap(ref int l,ref int m)
{
  int temp;
  temp = l;
  l = m;
  m = temp;
}
```

- Normal Delegate

```
public delegate void sDelegate(ref int x, ref int y);
```

- Generic Delegate
  - Return void

```
public delegate void swapDelegate<T>(ref T x, ref T y);
```

```
public delegate void swapDelegate2<T1,T2>(ref T1 x, ref T2 y);
```

# Generic Delegate

```
public static int sum(int l,int int m)
{
  int result;
  result =l+m;
  return result;
}
```

- Generic Delegate
  - Return data type

```
public delegate TResult SumDelegate<T1,T2,TRseult>(ref T1 x, ref T2 y);
```

```
SumDelegate<int,int , int> sumd = sum;
int result = sumd(10, 15);
```

# General Purpose Delegates ( C# 3.0 )

- System.Action (void as a return)
  - public delegate void Action ();
  - public delegate void Action< in T>(T arg)
  - public delegate void Action<in T1, in T2>( in T1 arg1, in T2 arg2) ..... T16
- System.Func
  - public delegate TResult Func< out TResult>();
  - public delegate TResult Func<in T, out TResult>(T arg)
  - public delegate TResult Func<in T1, in T2, out TResult>( in T1 arg1, in T2 arg2)

specifies that the type parameter is ***Contravariant***
(Same type or super class type)

specifies that a type parameter is ***Covariant***.
(Same type or derived type)

```
public delegate bool Mydelegate(int L, int M);
….
Mydelegate d;
```

```
Gụŋçîŋʯîŋʯčộộľđ
```

```
void bubbleSort(int[] arr,Func<int, int, bool> d)
{
}
```

# Assignment

- Modify Menu Program
  - Using **List<T>.Sort(Comparison<T>)**
    - Write anonymous method that implement Comparison< T> delegate for sort employee (by salary ,by Name) ascending or descending