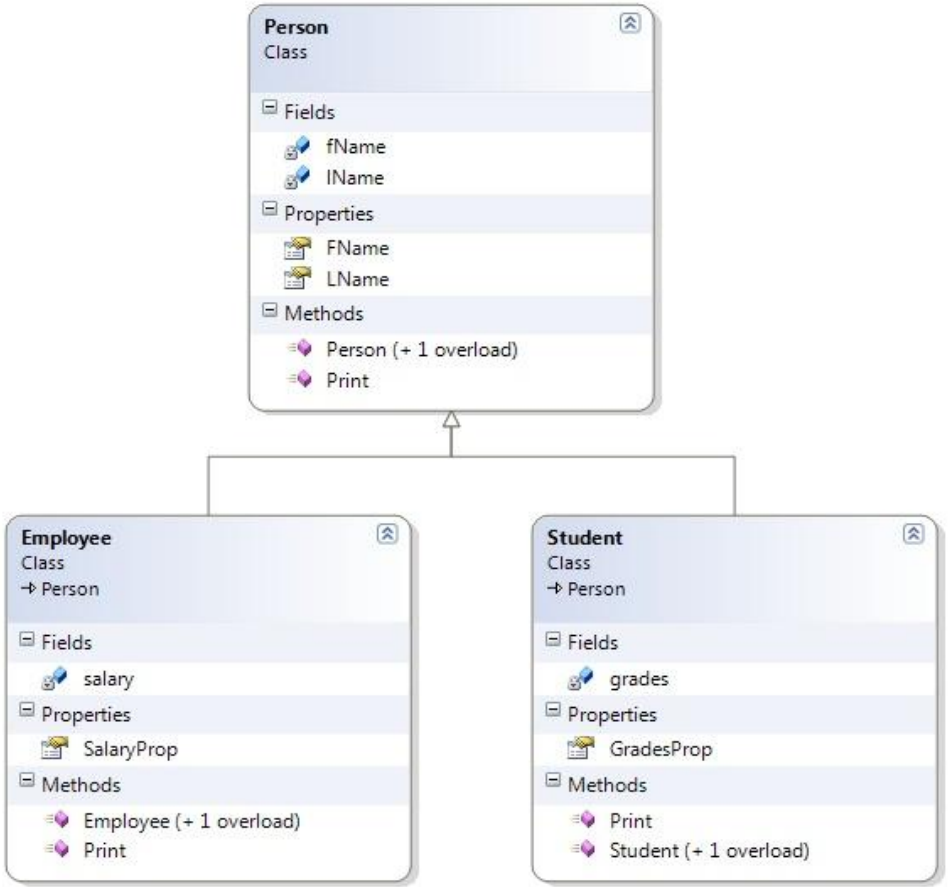
Advanced Programming 2
Recitation 2 – Introduction to C#

Roi Yehoshua
2017

Inheritance and Polymorphism

Class Hierarchy Example



Class Hierarchy Example

```
public abstract class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }

    public Person(string firstName, string lastName, int age = 0)
    {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }

    public Person() : this("", "", 0) { }

    public override string ToString()
    {
        return $"Name: {FirstName} {LastName}, Age: {Age}";
    }
}
```

Class Hierarchy Example

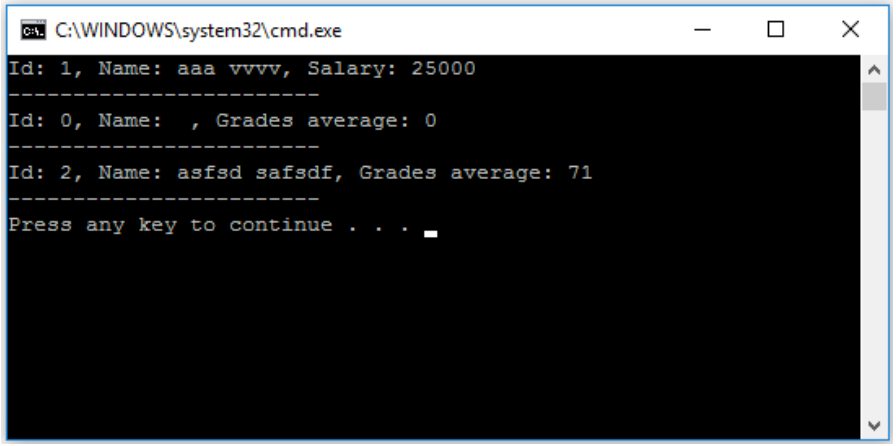
```
public class Employee : Person
{
    private double salary;
    public double Salary
    {
        get { return salary; }
        set {
            if (salary < 0)
                throw new ArgumentOutOfRangeException("Salary cannot be negative");
            salary = value;
        }
    }

    public Employee(int id, string firstName, string lastName, double salary) : base(id, firstName,
lastName)
    {
        Salary = salary;
    }

    public Employee(int id, string firstName, string lastName) : this(id, firstName, lastName, 0) { }

    public override string ToString()
    {
        return base.ToString() + $", Salary: {Salary}";
    }
}
```

Class Hierarchy Example

```
public class Student : Person
{
    public int[] Grades { get; set; }

    public double GradesAverage
    {
        get { return Grades.Sum(g => g) / Grades.Count(); }
    }

    public Student(int id, string firstName, string lastName, int[] grades) : base(id,
firstName, lastName)
    {
        Grades = grades;
    }

    public Student()
    {
        Grades = new int[10];
    }

    public override string ToString()
    {
        return base.ToString() + $", Grades average: {GradesAverage}";
    }
}
```

Class Hierarchy Example

```
static void Main(string[] args)
{
    Employee e1 = new Employee(1, "aaa", "vvvv", 25000);
    Console.WriteLine(e1);
    Console.WriteLine("------------------------");

    int[] grades = new int[] { 60, 80, 75 };
    Student s1 = new Student();
    Student s2 = new Student(2, "asfsd", "safsdf", grades);
    Console.WriteLine(s1);
    Console.WriteLine("------------------------");
    Console.WriteLine(s2);
    Console.WriteLine("------------------------");
}
```



Access Modifiers

- In C#, class members can have one of the following 5 access modifiers:

	public	protected	protected internal		internal		private
description	ניתן לגשת מכל מקום	ניתן לגשת מתוך האובייקט או מתוך אובייקט יורש	מחוץ ל Assembly	בתוך ה Assembly	מחוץ ל Assembly	בתוך ה Assembly	ניתן לגשת אך ורק מתוך האובייקט
			protected	public	private	public	
variable / const / readonly / property / method / event	V	V	V		V		V
enum / struct / class / interface / delegate	V				V		
Inner enum / struct / class / interface / delegate	V	V	V		V		V

Casting Operators

- The **cast** operator attempts to cast an object to a specific type, and throws an exception if it fails
- The **is** operator checks whether an object is compatible with a given type, returns boolean
- The **as** operator attempts to cast an object to a specific type, and returns null if it fails
- In general, the as operator is more efficient because it actually returns the cast value if the cast can be made successfully
- The is operator is typically used when you just want to determine an object's type but do not have to actually cast it

```
MyClass b = (MyClass)someObject;
```

```
if (someObject is MyClass) ...
```

```
MyClass obj = someObject as MyClass;
if (obj != null) ...
```

Interfaces

Interfaces in C#

- **C++** have abstract classes that can be used as “interfaces”
  - All methods are pure virtual
  - No data members or CTORs
  - Multiple inheritance allows to implement many “interfaces”
- **Java** interfaces – contain only signatures of methods
- a **C#** interface allows:
  - Signatures of methods
  - Properties
  - Events
- In C# it is common that interfaces start with the letter I

Interfaces Example

```
interface IPlayable
{
    void Play();
}

interface IRecordable
{
    void Record();
}
```

```
class Movie : IPlayable, IRecordable
{
    public void Play()
    {
        Console.WriteLine("Playing movie");
    }

    public void Record()
    {
        Console.WriteLine("Recording
movie");
    }
}
```

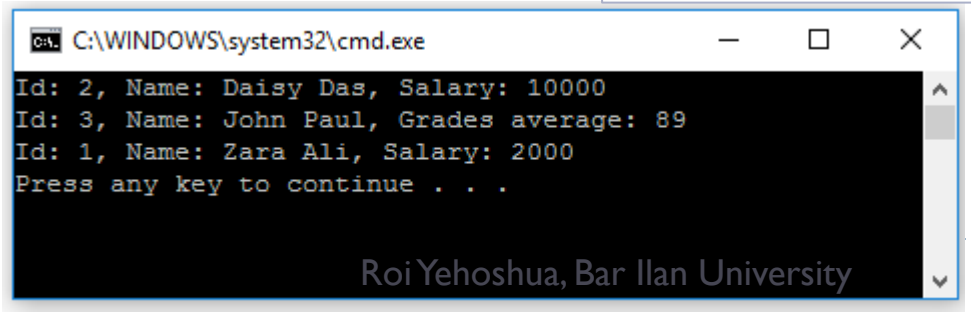
```
static void Main(string[] args)
{
    Movie movie = new Movie();

    IPlayable playable = movie;
    playable.Play();

    IRecordable recordable = movie;
    recordable.Record();
}
```

IComparable<T>

- IComparable<T> defines a generalized comparison method that a type implements to create a type-specific comparison method for ordering or sorting its instances

```
public abstract class Person : IComparable<Person>
{
    public int Id { get; private set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    …
    public int CompareTo(Person other)
    {
        return FirstName.CompareTo(other.FirstName);
    }
}
```

```
static void Main(string[] args)
{
    Person[] pArr = new Person[3];
    pArr[0] = new Employee(1, "Zara", "Ali", 2000);
    pArr[1] = new Employee(2, "Daisy", "Das", 10000);
    pArr[2] = new Student(3, "John", "Paul", new int[]
{ 95, 83 });

    Array.Sort(pArr);
    foreach (Person p in pArr)
    {
        Console.WriteLine(p);
    }
}
```



Generics and Collections

The Need for Generics

- Performance
  - Reduce boxing / unboxing
  - Fewer downcasts
- Type safety
  - Compile time vs. run time
- Generic algorithms
  - Sorting, searching, etc.
- Increased code re-use
  - Type independent code, such as collections

Life With and Without Generics

```
using System;
using System.Collections;

public class NoGenerics {
   public static void Main() {
      ArrayList list = new ArrayList();
      list.Add("Hello");
      list.Add(5);   // what happens here? (1)

      string s = (string)list[0];   // (2)
      int n = (int)list[1];      // (3)

      int x = (int)list[0];      // (4)
   }
}
```

```
using System;
using System.Collections.Generic;

public class WithGenerics {
   public static void Main() {
      List<DateTime> list = new List<DateTime>();

      list.Add(DateTime.Now);   // no boxing
      list.Add(5);              // compilation error
      list.Add("Hello");        // ditto

      DateTime now = list[0];   // no unboxing
   }
}
```

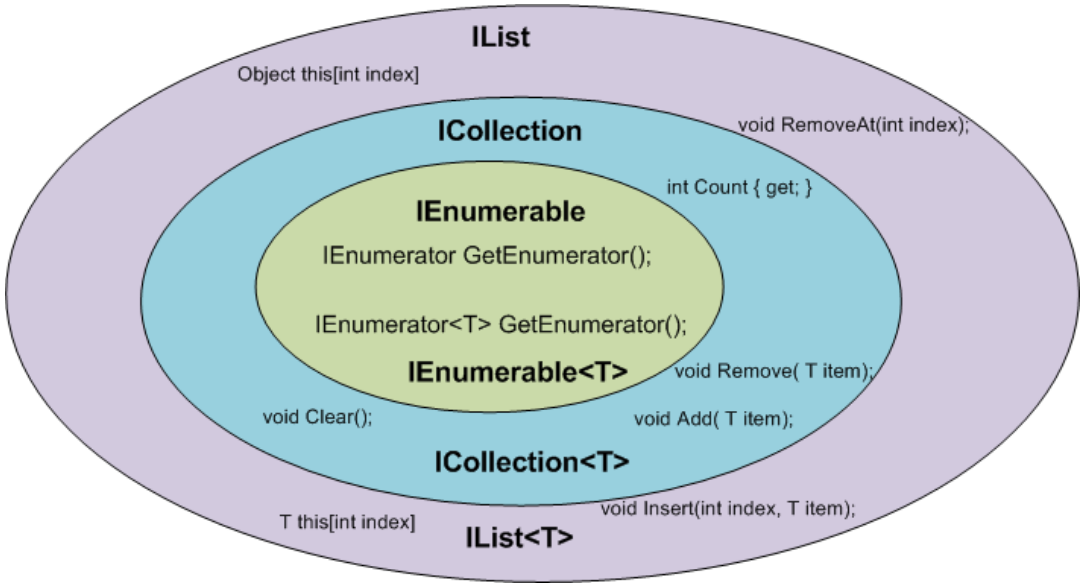
Constraints

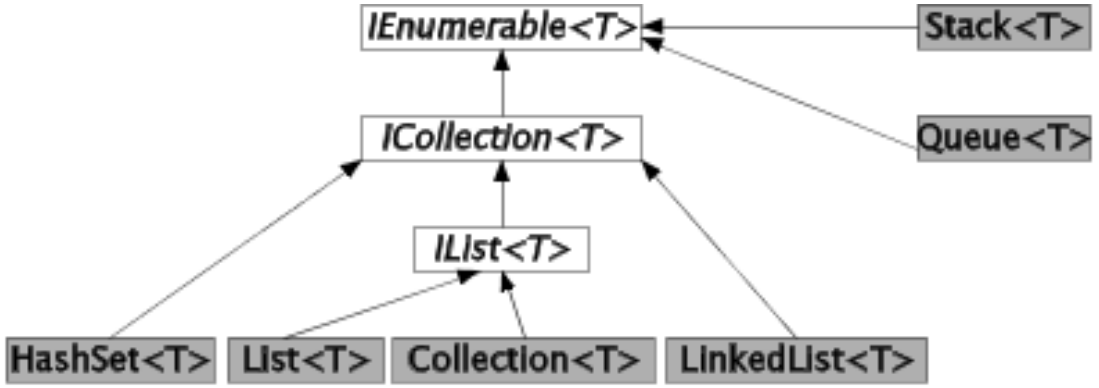
- A way to limit the type replacements for generic arguments
  - Base type
  - Supported interfaces
  - Default constructor
- Use the **`where`** keyword for base type and interfaces, the **`new`** keyword for default constructor

```
static T Min<T>(T[] arr) where T : IComparable<T>
{
    T min = arr[0];
    for (int i = 1; i < arr.Length; i++)
    {
        if (arr[i].CompareTo(min) < 0)
            min = arr[i];
    }
    return min;
}
```

```
static void Main(string[] args)
{
    int[] arr = { 12, 3, 8, 15, 7 };
    Console.WriteLine("Min: {0}", Min(arr));
}
```

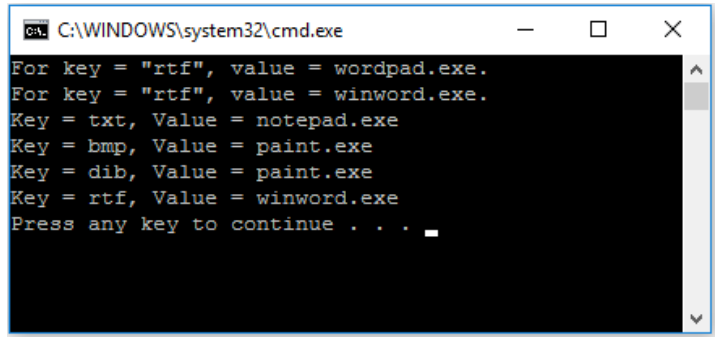
C# Collections Hierarchy





Example for Dictionary

```
static void Main(string[] args)
{
    // Create a new dictionary of strings, with string keys.
    Dictionary<string, string> openWith = new Dictionary<string, string>();

    // Add some elements to the dictionary. There are no
    // duplicate keys, but some of the values are duplicates.
    openWith.Add("txt", "notepad.exe");
    openWith.Add("bmp", "paint.exe");
    openWith.Add("dib", "paint.exe");
    openWith.Add("rtf", "wordpad.exe");

    // You can use the indexer to access elements
    Console.WriteLine("For key = \"rtf\", value = {0}.", openWith["rtf"]);

    // The indexer can be used to change the value associated with a key.
    openWith["rtf"] = "winword.exe";
    Console.WriteLine("For key = \"rtf\", value = {0}.", openWith["rtf"]);

    // Print all key-value pairs
    foreach (KeyValuePair<string, string> kvp in openWith)
    {
        Console.WriteLine("Key = {0}, Value = {1}", kvp.Key, kvp.Value);
    }
}
```



Delegates and Events

Delegates

- A delegate is a safe pointer to a function
- Delegates allow us to write functions and classes, that their implementation is only partially known at build time
- For example, when we want to build a function that sorts an array, but we don't know how to compare the elements
  - In such case, we can pass to the sorting function a delegate that will point to a compare function

Delegate Definition

- Delegate is defined using the keyword delegate followed by the signature of the functions that it can point to
- CalcDelegate can point to any function that gets two ints and return an int
- You can create instances of this delegate as if it were a class
- In its constructor we need to pass the function that the delegate will point to
- You can invoke a delegate in the same way you invoke a function

```
delegate int CalcDelegate(int x, int y);
```

```
static int Plus(int x, int y)
{
    return x + y;
}

static void Main(string[] args)
{
    CalcDelegate del = new CalcDelegate(Plus);
    int result = del(3, 5);
    Console.WriteLine($"3 + 5 = {result}");
}
```

MultiCast Delegates

- A delegate can point to more than one function
  - Actually, a delegate holds an array of pointers to functions
- You can use the operators += and -= to add/remove pointers to functions
- If the delegate returns a result, only the result of the last invoked function will be returned

```
delegate int CalcDelegate(int x, int y);

static int Plus(int x, int y)
{
    return x + y;
}

static int Minus(int x, int y)
{
    return x - y;
}

static void Main(string[] args)
{
    CalcDelegate del = new CalcDelegate(Plus);
    del += new CalcDelegate(Minus);
    int result = del(3, 5);
    Console.WriteLine($"result = {result}");
}
```

Delegates as function parameters

- You can use delegates to pass functions as parameters to other functions

```
delegate bool FilterDelegate(int num);

static int[] FilterArray(int[] arr, FilterDelegate filterMethod)
{
    List<int> list = new List<int>();
    for (int i = 0; i < arr.Length; i++)
    {
        if (filterMethod(arr[i]))
            list.Add(arr[i]);
    }
    return list.ToArray();
}
```

```
static bool IsEven(int num)
{
    return (num % 2 == 0);
}

int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int[] newArr = FilterArray(arr, IsEven);

for (int i = 0; i < newArr.Length; i++)
    Console.WriteLine(newArr[i]);
```

Anonymous Functions

- Allow to pass a block of code as a parameter to another function without explicitly defining the function
- Can access the local parameters of the invoking function

```
// Anonymous method:
int[] newArr = FilterArray(arr, delegate(int num)
{
    return (num % 2 == 0);
});
```

Lambda Expressions

- Lambda expressions allows a neater way to create delegates
- The syntax of a lambda expression is:

  Function parameters => Return value
- Example:

```
int[] newArr = FilterArray(arr, x => x % 2 == 0);
```

- If the function is parameterless, then you write an empty parenthesis:

  () => “test”
- If the function gets more than one parameter, you need to use brackets:

  (x, y) => x + y

Generic Delegates

- You can also define generic delegates

```
delegate bool FilterDelegate<T>(T item);

static T[] FilterArray<T>(T[] arr, FilterDelegate<T> filterMethod)
{
    List<T> list = new List<T>();

    for (int i = 0; i < arr.Length; i++)
    {
        if (filterMethod(arr[i]))
            list.Add(arr[i]);
    }

    return list.ToArray();
}
```

Events

- Delegates are not usually exposed directly as public members
  - Anyone can manipulate and even nullify
- Events allow controlled access to a delegate chain
  - Subscribing and unsubscribing
- An event consists of
  - A private delegate member
  - Add / remove methods to add / remove subscribers

Declaring an Event

- Create the appropriate delegate type
- By convention, use the **`EventHandler`** or the **`EventHandler<T>`** delegate types
  - Where T is a type deriving from `System.EventArgs`
    - In itself, an empty class
- Use the `EventArgs.Empty` static field to convey no information

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

Event Implementation Example

- For example, let's raise an event each time the player moves on the board
- First define the EventArgs type:

```
public class PlayerMovedEventArgs : EventArgs
{
    public Direction Direction { get; private set; }
    public PlayerMovedEventArgs(Direction direction)
    {
        Direction = direction;
    }
}
```

Event Implementation Example

- Publisher:

Declare an event based on the generic delegate EventHandler<T>

This is the new null conditional operator introduced in C# 6.0

```
public partial class MazeBoard : UserControl
{
    public event EventHandler<PlayerMovedEventArgs> PlayerMoved;
    private void Window_KeyDown(object sender, KeyEventArgs e)
    {
        switch (e.Key)
        {
            case Key.Left:
                direction = Direction.Left;
                break;
            case Key.Right:
            // …
        }
    }

    PlayerMoved?.Invoke(this, new PlayerMovedEventArgs(direction));
}
```

Raise the event

Event Implementation Example

- Subscriber:

Subscribe to the event

Event handler – defines what will happen when the event occurs

```
class MazeBoardListener
{
    private MazeBoard mazeBoard;

    public MazeBoardListener()
    {
        mazeBoard = new MazeBoard();
        mazeBoard.PlayerMoved += MazeBoard_PlayerMoved;
    }

    private void MazeBoard_PlayerMoved(object sender, PlayerMovedEventArgs e)
    {
        Console.WriteLine($"Player moved in direction: {e.Direction}");
    }
}
```