# Advanced C# Features

# Agenda

- Delegate & Event
- Object Instance & Inheritance
- Type Casting
- Generic
- Memory Management in C#

Section 1

# DELEGATE

# Delegate

- A delegate is a reference type that defines a method signature
- A delegate instance holds one or more methods
  - ✓ Essentially an “object-oriented function pointer”
  - ✓ Methods can be static or non-static
  - ✓ Methods can return a value
- Provides polymorphism for individual functions
- Foundation for event handling

# Delegate Example

```
//// Declare a delegate
delegate double MyDelegate(double x);

static void DemoDelegates()
{
    //// Instantiate
    MyDelegate delegateInstance = new MyDelegate(Math.Sin);
    //// Invoke
    double x = delegateInstance(1.0);
}
```

# Multicast Delegate

- A delegate can hold and invoke multiple methods
  - ✓ Multicast delegates must contain only methods that return `void`, else there is a run-time exception
- Each delegate has an invocation list
  - ✓ Methods are invoked sequentially, in the order added
- The += and −= operators are used to add and remove delegates, respectively
- += and −= operators are thread-safe

# Delegate Example

```
delegate void SomeEvent(int x, int y);
 static void Foo1(int x, int y)
 {
    Console.WriteLine("Foo1");
 }
 static void Foo2(int x, int y)
 {
    Console.WriteLine("Foo2");
 }
 public static void Main()
 {
    SomeEvent func = new SomeEvent(Foo1);
    func += new SomeEvent(Foo2);
    //// Foo1 and Foo2 are called
    func(1, 2);
```

Section 2

# EVENTS

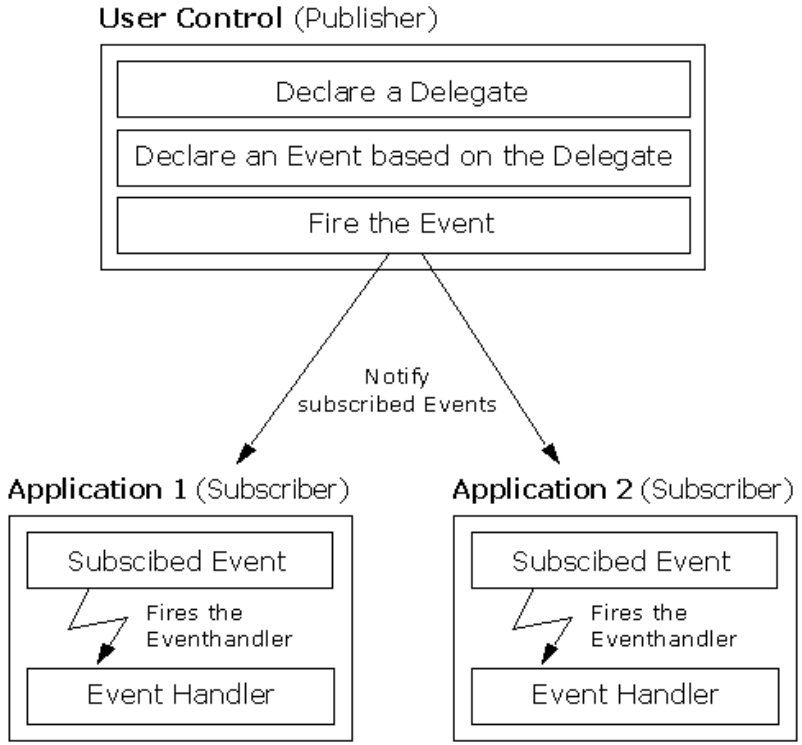
# Events

- Event handling is a style of programming where one object notifies another that something of interest has occurred
  - ✓ A publish-subscribe programming model
- Events allow you to tie your own code into the functioning of an independently created component
- Events are a type of “callback” mechanism

# Events

- Events are well suited for user-interfaces
  - ✓ The user does something (clicks a button, moves a mouse, changes a value, etc.) and the program reacts in response
- Many other uses, e.g.
  - ✓ Time-based events
  - ✓ Asynchronous operation completed
  - ✓ Email message has arrived
  - ✓ A web session has begun

# Events


User Control (Publisher)
Declare a Delegate
Declare an Event based on the Delegate
Fire the Event
Notify
subscribed Events
Application 1 (Subscriber)
Subscibed Event
Fires the
Eventhandler
Event Handler
Application 2 (Subscriber)
Subscibed Event
Fires the
Eventhandler
Event Handler

Section 3

# OBJECT INSTANCE & INHERITANCE

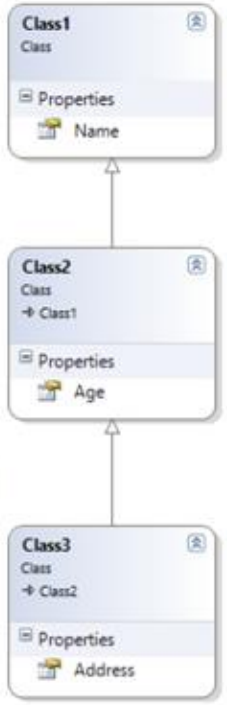
# Object Instance & Inheritance

- Classes can optionally be declared as:
  - ✓ static — can never be instantiated
  - ✓ abstract — can never be instantiated, it is an incomplete class
  - ✓ sealed — all classes can be inherited unless marked as sealed
- Virtual Methods
  - ✓ Virtual methods have implementations
  - ✓ They can be overridden in derived class.

# Object Instance & Inheritance

- Class1 defines one property: Name
- Class2 inherits Name from Class2 and defines Age.
- Class3 inherits Name from Class2 (via Class2) and Age from Class2, as well as defines Address.
- So an instance of Class3 has 3 properties:
  - ✓ Name
  - ✓ Age
  - ✓ Address

```
public class Class1
{
    public string Name { get; set; }
}
public class Class2 : Class1
{
    public int Age { get; set; }
}
public class Class3 : Class2
{
    public string Address { get; set; }
}
```



# Object Instance & Inheritance

- Demo Virtual
- Demo overrides
- Demo New

# Object Instance & Inheritance

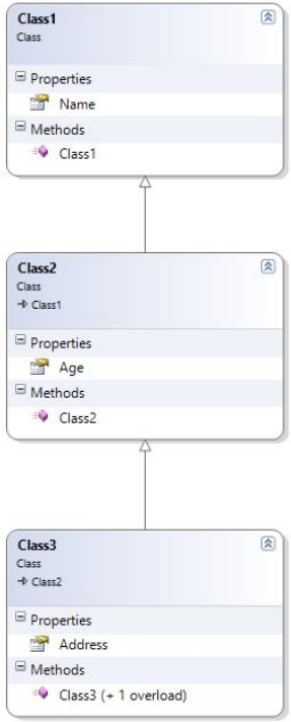
- Creating Object Instances
  - When a class or struct is created, a constructor is called.
  - Unless a class is static, the compiler generates a default constructor if not supplied in code.
  - Constructors are invoked by using the new keyword.
  - Constructors may require parameters.
  - More than one constructor can be defined.
  - Constructors can be chained together
  - Base class constructors are always called (first)

# Object Instance & Inheritance

```
public class Class1
{
    public string Name { get; set; }
    public Class1(string name)
    {
        Name = name;
    }
}

public class Class2 : Class1
{
    public int Age { get; set; }
    public Class2(string name, int age) : base(name)
    {
        Age = age;
    }
}

public class Class3 : Class2
{
    public string Address { get; set; }
    public Class3()
        : this("Fred", 1, "1 Microsoft Way") { }
    public Class3(string name, int age, string address) : base(name, age)
    {
        Address = address;
    }
}
```



Section 4

# CASTING TYPE

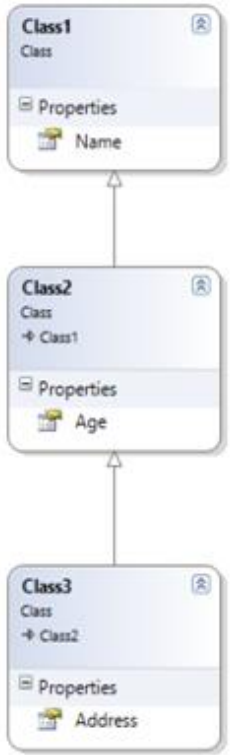
# Casting Type

- Casting Types
  - ✓ Casting allows us to work with types in a general sense — as their base object or as an instance of an interface implementation.
- We can explicitly attempt to cast an object to another type
  - ✓ An advantage of strong typing is that the compiler often knows when a cast is possible. It doesn't always know.
  - ✓ Compilation will fail if the compiler detects an invalid cast.
- But, what about scenarios that the compiler can't detect?

# Casting Type

- In this scenario, instances of Class2 and Class3 can be cast an instance of Class1.
- However, an instance of Class2 can never be cast to Class3.
- If Class2 is cast to Class1, can it then be cast to Class3?

```
public class Class1
{
    public string Name { get; set; }
}
public class Class2 : Class1
{
    public int Age { get; set; }
}
public class Class3 : Class2
{
    public string Address { get; set; }
}
```



# Casting Type is Operator

- The `is` operator is used to dynamically test if the run-time type of an object is compatible with a given type

```
static void DoSomething(object o) {
  if (o is Car)
    ((Car)o).Drive();
}
```

- Don’t abuse the is operator: it is preferable to design an appropriate type hierarchy with polymorphism methods

# Casting Type as Operator

- The **`as`** operator tries to convert a variable to a specified type; if no such conversion is possible the result is **`null`**

```
static void DoSomething(object o) {
  Car c = o as Car;
  if (c != null) c.Drive();
}
```

- More efficient than using `is` operator: test and convert in one operation
- Same design warning as with the `is` operator

# typeof Operator

- The **`typeof`** operator returns the `System.Type` object for a specified type
- Can then use reflection to dynamically obtain information about the type

```
Console.WriteLine(typeof(int).FullName);
Console.WriteLine(typeof(System.Int32).Name);
Console.WriteLine(typeof(float).Module);
Console.WriteLine(typeof(double).IsPublic);
Console.WriteLine(typeof(Car).MemberType);
```

Section 5

# GENERIC

# Generic

- Generics introduce to the .NET Framework the concept of type parameters
- Generics make it possible to design classes and methods that defer type specification until the class or method is declared
- One of the most common situations for using generics is when a strongly-typed collection is required.
  - ✓ lists, hash tables, queues, etc…

# Boxing & Unboxing

- Boxing is the act of converting a value type to a reference type.
- Unboxing is the reverse
  - ✓ Unboxing requires a cast.
- Boxing/Unboxing copies the value.
- Boxing is computationally expensive
  - ✓ avoid repetition
- Generics help avoid these scenarios

```
int count = 1;

// the value of count is copied and boxed
object countObject = count;

count += 1; // countObject is still 1

// the value of countObject (1) is unboxed
// and copied to count
count = (int)countObject;
```

# Generic

- You can create your own custom generic classes.
- When creating a generic class, consider:
  - ✓ Which types to generalize
  - ✓ What constraints to apply
  - ✓ Whether to create generic base classes
  - ✓ Whether to create generic interfaces
- Generic methods may also be created within non-generic classes

# Generic

```
void Breakfast()
{
    var bird = new Animal<Egg>();
    var pig = new Animal<Piglet>();
}

public class Animal<T> where T : Offspring
{
    public T Offspring { get; set; }
}

public abstract class Offspring { }
public class Egg : Offspring { }
public class Piglet : Offspring { }
```

Section 6

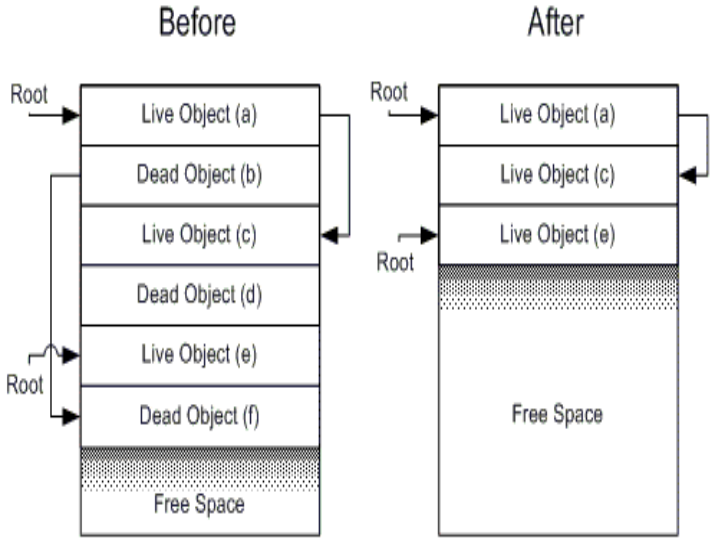
# MEMORY MANAGEMENT

# Garbage Collection

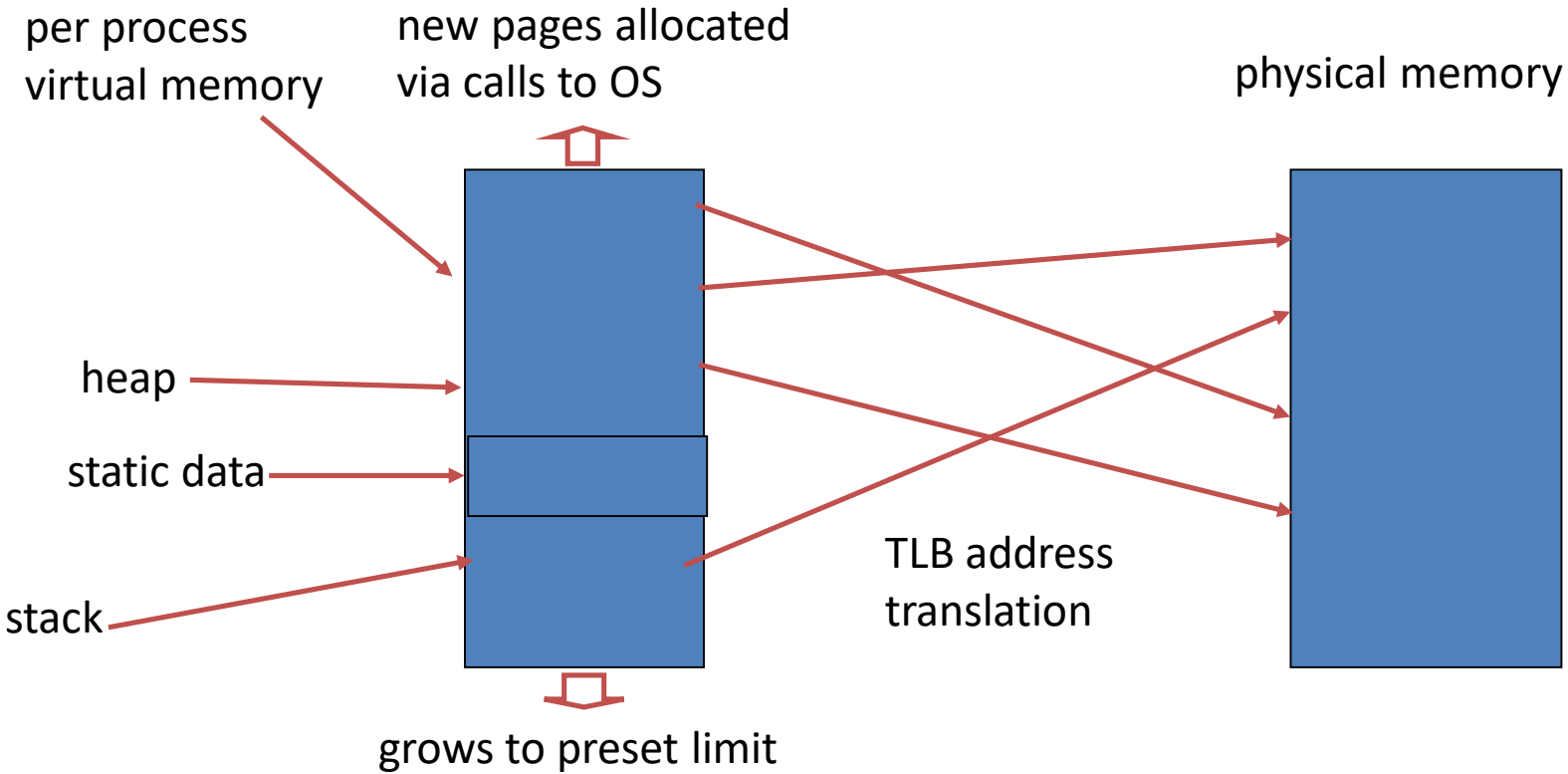
- C# have a managed runtime
  - ✓ all pointer types are known at runtime
  - ✓ can do reference counting in CLR
- Garbage Collection is program analysis
  - ✓ figure out properties of code automatically
  - ✓ two type of analysis: dynamic and static

# How GC works

- Mark all managed memory as garbage
- Look for used memory blocks, and mark them as valid
- Discard all unused memory blocks
- Compact the heap
- Can be expensive
  - ✓ everything stops and collection happens
  - ✓ this is a general problem for garbage collection



# Memory layout



# Heap VS Stack

- In a stack, the allocation and deallocation is automatically done by whereas, in heap, it needs to be done by the programmer manually.
- Handling of Heap frame is costlier than handling of stack frame.
- Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.
- Stack frame access is easier than the heap frame as the stack have small region of memory and is cache friendly, but in case of heap frames which are dispersed throughout the memory so it cause more cache misses.
- Stack is not flexible, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be altered.
- Accessing time of heap takes is more than a stack.

# Stack and Heap

## Value type

- bool
- byte
- char
- decimal
- double
- enum
- float
- int
- long
- sbyte
- short
- struct
- uint
- ulong
- ushort

## Reference type

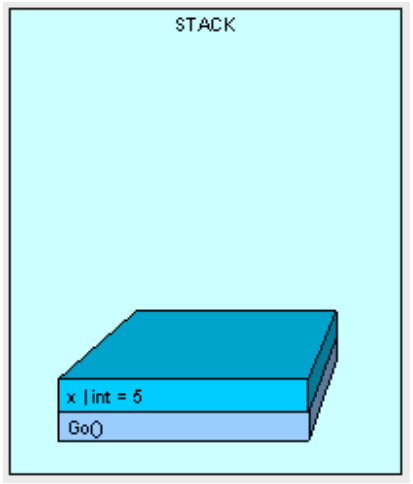
- class
- interface
- delegate
- object
- string

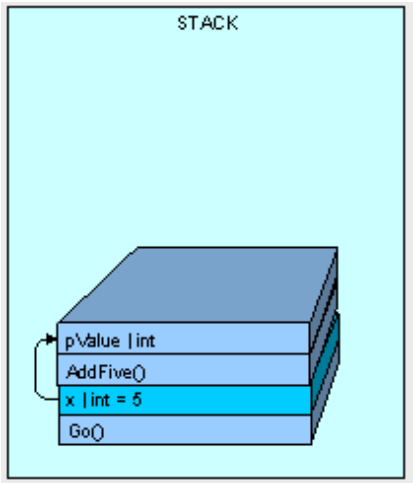
# Important Rules

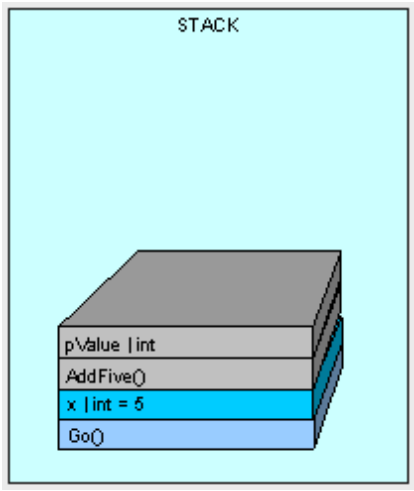
- A Reference Type always goes on the Heap
- Value Types and Pointers always go where they were declared.

# Passing Value Types

1. As the method executes, space for "x" is placed on the stack with a value of 5.
2. AddFive() is placed on the stack with space for it's parameters and the value is copied, bit by bit from x.
3. When AddFive() has finished execution, the thread is passed back to Go() and because AddFive() has completed, pValue is essentially "removed":



# Passing Reference Types

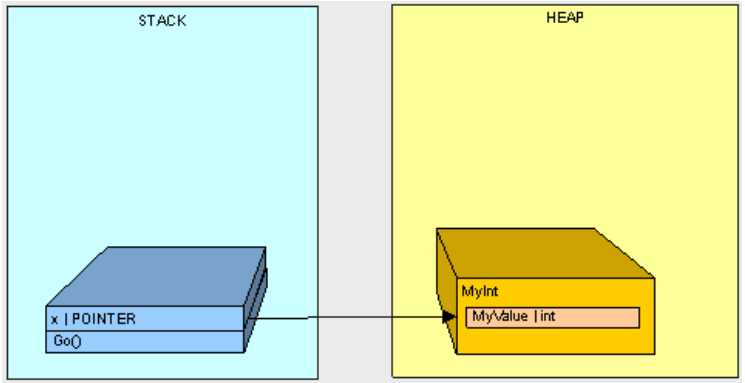
- Passing parameters that are reference types is similar to passing value types by reference

If we are using the value type

```
public class MyInt
        {
            public int MyValue;
        }
```

And call the Go() method, the MyInt ends up on the heap because it is a reference type:

```
public void Go()
        {
          MyInt x = new MyInt();
        }
```




Questions
Answers

# Thank you