CS 2530

Midterm1 Prep

C# Basics

- Strongly-typed
- Case-sensitive
- Object-oriented
- No global methods / functions
- No implicit variable declarations
- Extensible (examples)
- Allows to overload operators like +, -, *, etc. in your own classes
- Developed by MS, later approved as standard

.NET

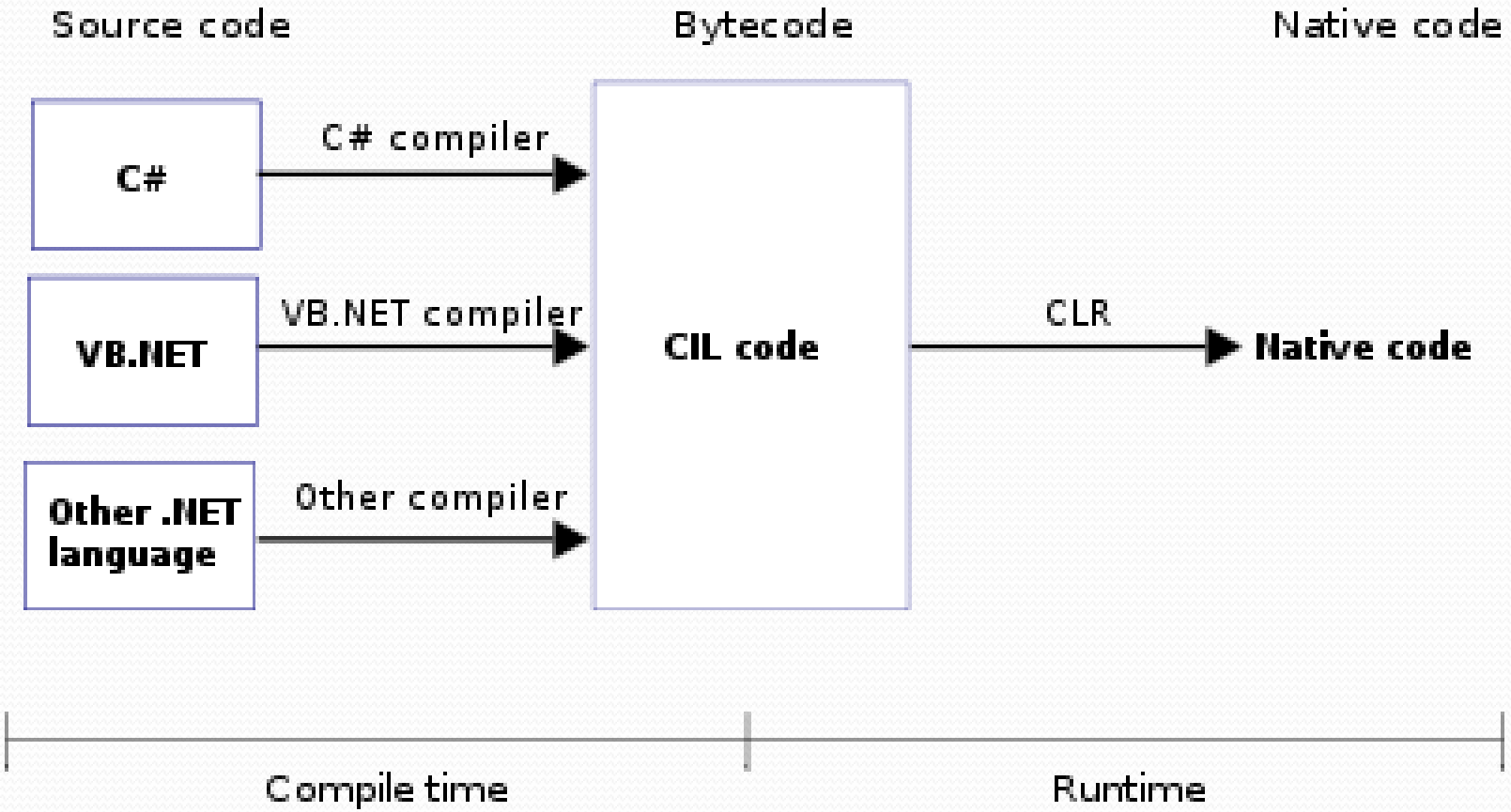
- .NET Framework:

  A programming infrastructure created by MS for building , deploying, and running applications
- .NET Framework Class Library

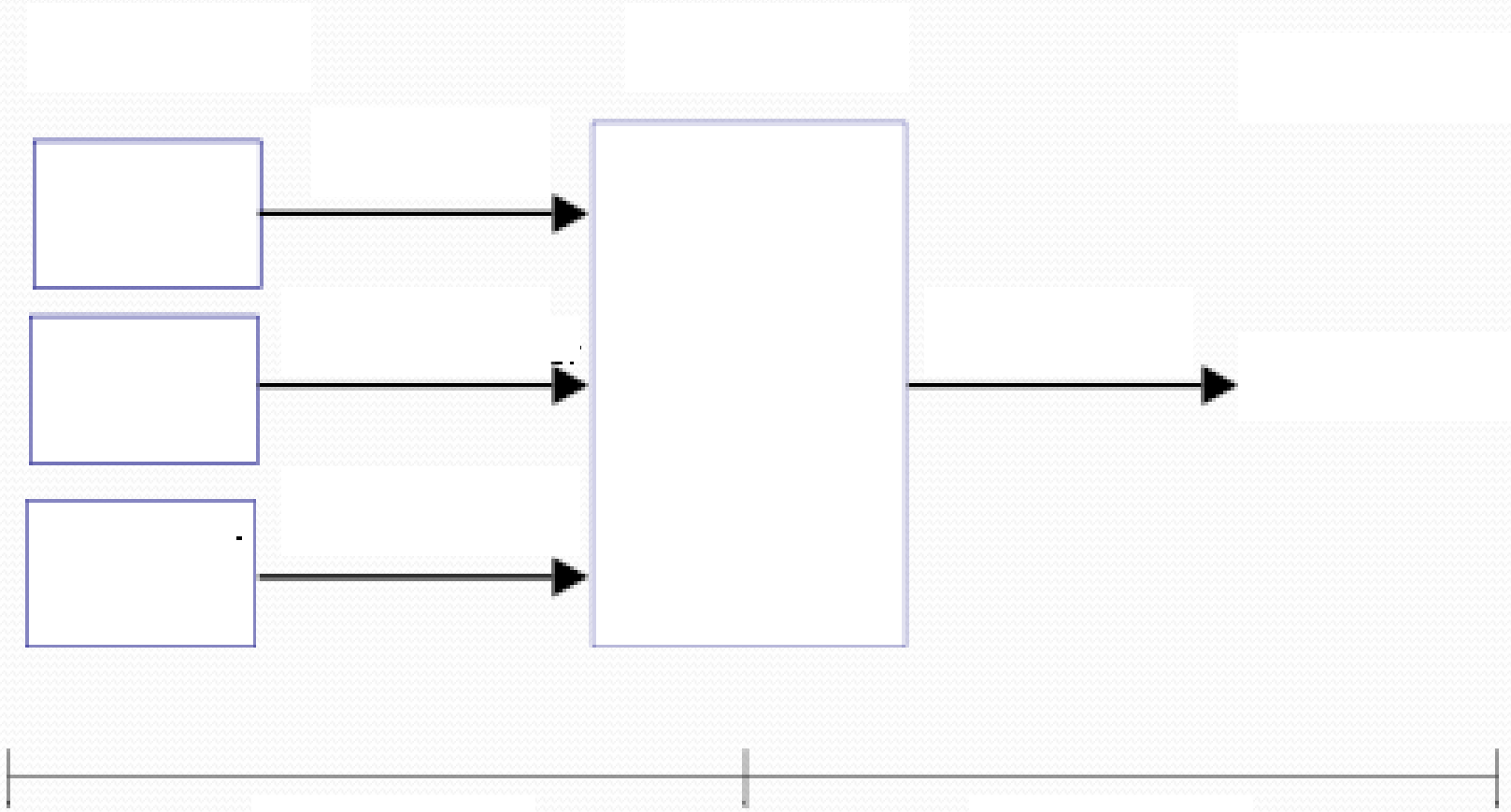
  Classes, interfaces, and value types that provides access to system functionality
- CLR (Common Language Runtime)

  Run time environment that provides the underlying infrastructure for .NET Framework
- CIL .. Common Intermediate Language

a) .NET Framework - CLR



a) .NET Framework - CLR



Visual Studio

- Microsoft's Integrated Development Environment (IDE)
- Includes code editor, compiler, debugger, built-in tools for designing, . . .
- IntelliSense
- Supports different languages (C#, Visual Basic, Visual C++, ..)
- Provides templates for a variety of applications (WPF, Console, Windows Fomrs, …)

TODO:

- What is the advantage of compiling to CIL?
- List two major advantages of using methods from the .NET library over writing the same functionality yourself
- What do the following icons indicate?

Value Type vs Reference Type

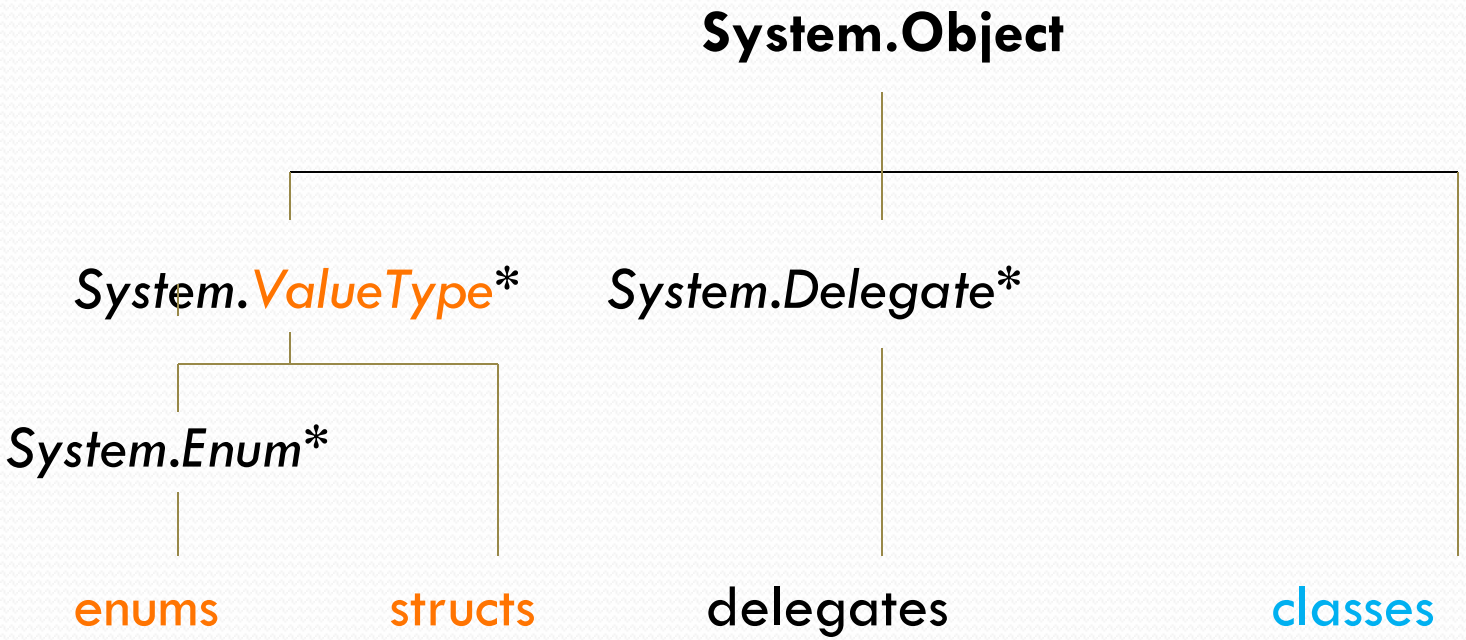
int	**ulong**	**double**	**string**
struct Point	**dynamic**	**bool**	**enum Suit**
char	**byte**	**object**	**float**
class Deck	**short**	**decimal**	

Value Types

- Copied by value
- C# has 2 kinds of value types:
  1. Enumerations
  2. Structs
     - Numeric types
       - Integral types (`byte`, `short`, `int`, `long`, `char`, ..)
       - Floating-point types (`float`, `double`)
       - `decimal`
     - `bool`
     - User-defined structs

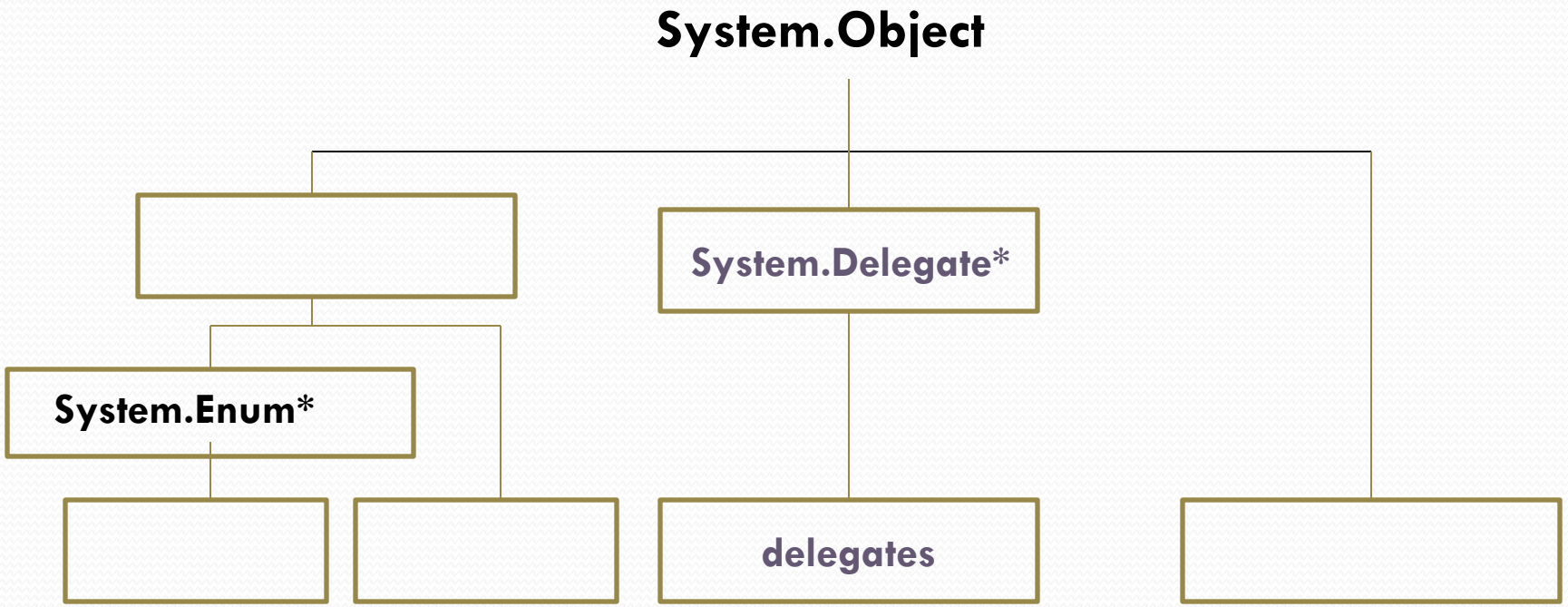
- All build-in types are value types except for
  - string,
  - object
  - dynamic

System.Object



*.. Abstract classes you cannot explicitly derive from

System.Object



Exercise: .NET Framework / Object

Value Type vs Reference Type

```
public class CircleClass
{
      public double Radius { get; set; }
}

public struct CircleStruct
{
      public double Radius { get; set; }
}
```

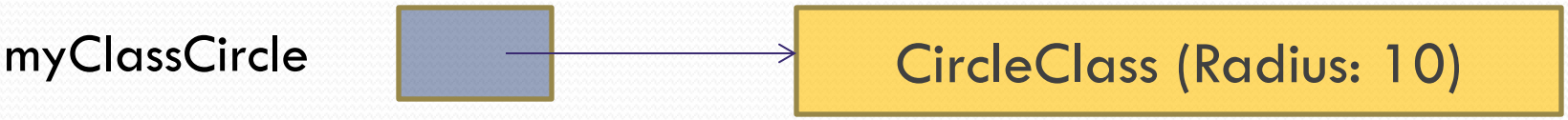
Value Type vs Reference Type

```
 static void DoubleRadius(CircleClass circle)
{
   circle.Radius *= 2;
}

static void DoubleRadius(CircleStruct circle)
{
   circle.Radius *= 2;
}
```

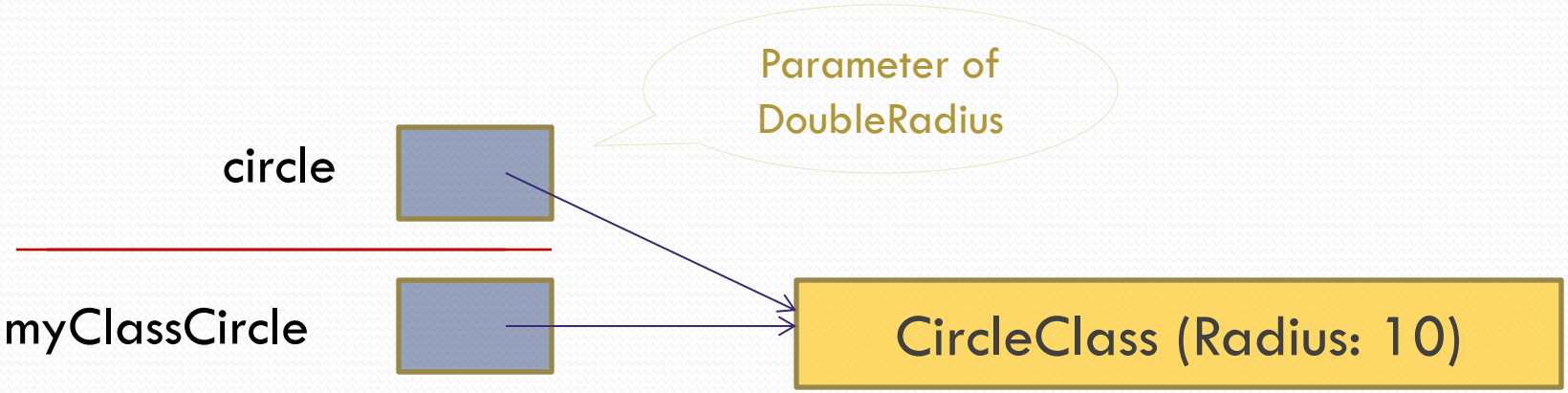
Class CircleClass

```
CircleClass myClassCircle = new CircleClass { Radius = 10 };
```



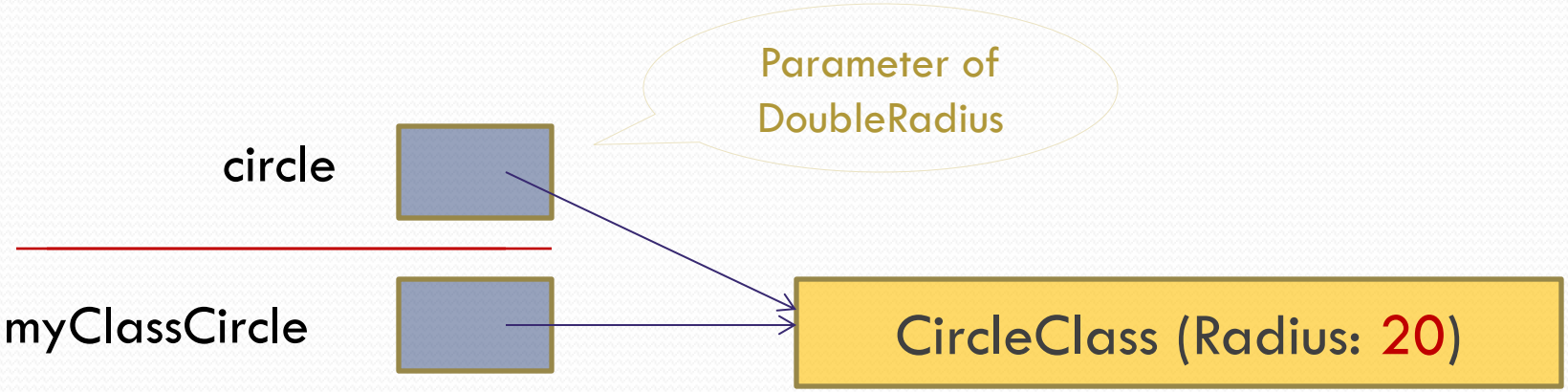
Class CircleClass

DoubleRadius(myClassCircle);



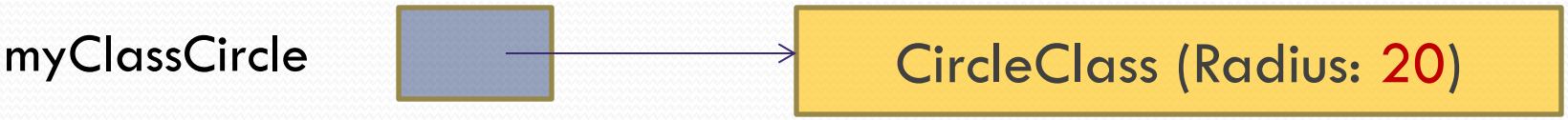
Class CircleClass

DoubleRadius(myClassCircle); // method changes radius



Class CircleClass

. . . after method terminated . . .



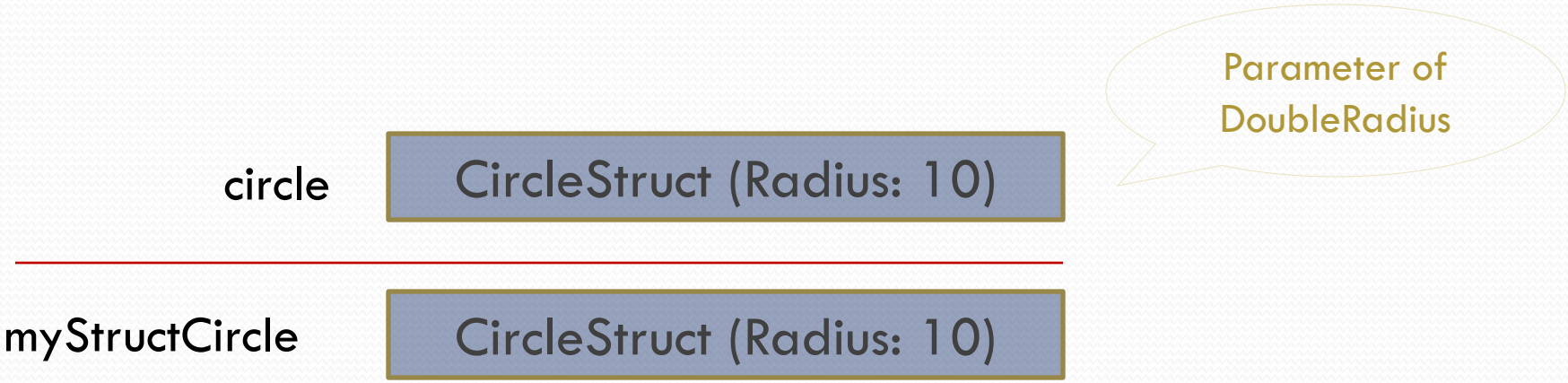
Struct CircleStruct

```
CircleStruct myStructCircle = new CircleStruct { Radius = 10 };
```

myStructCircle

Struct CircleStruct

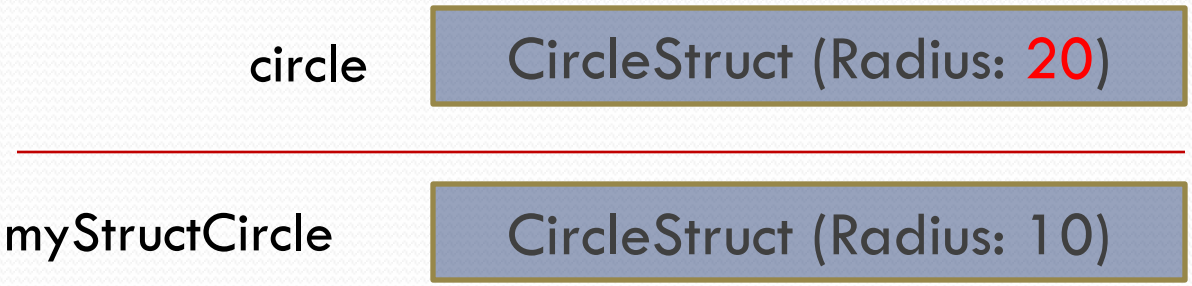
DoubleRadius(myStructCircle);



Struct CircleStruct

DoubleRadius(myStructCircle); // method changes radius

Parameter of DoubleRadius



Struct CircleStruct

. . . after method terminated . . .

myStructCircle

CircleStruct (Radius: 10)

Conclusion:

Avoid subtle errors by declaring properties

in Structs with a private setter or

read-only (no setter)

- Exercise class / struct

Naming Convention

- Fields . . . . ***camelCase*** ✓
  - Instance variables and static variables
- Properties . ***PascalCase*** !
- Methods . . ***PascalCase*** !
- Constructor ***same spelling as class (PascalCase)*** ✓
  - Initializes class instance
  - Compiler provides default constructor if no constructor is declared
- Local variables and parameters: ***camelCase*** ✓

TODO:

- Following the C# naming conventions which of the following items should be spelled in PascalCase?

a) field

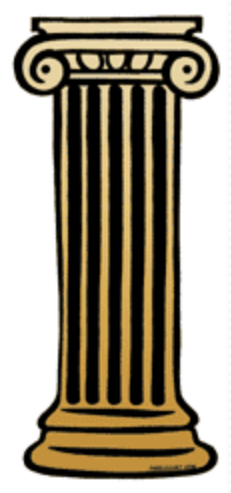
b) local variable

c) parameter

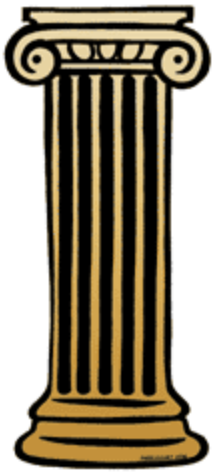
d) property

Some important concepts

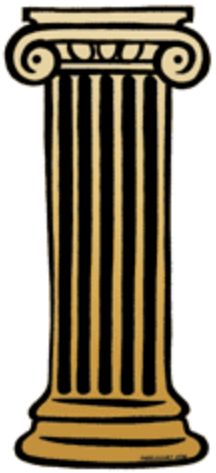
3 Pillars of Object-Oriented Programming:



Encapsulation



Inheritance

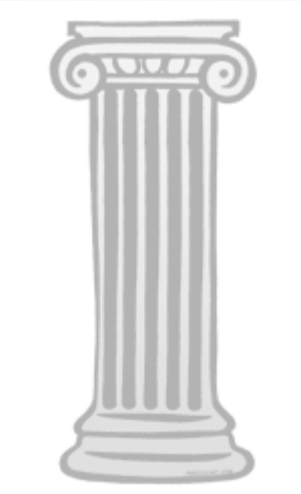


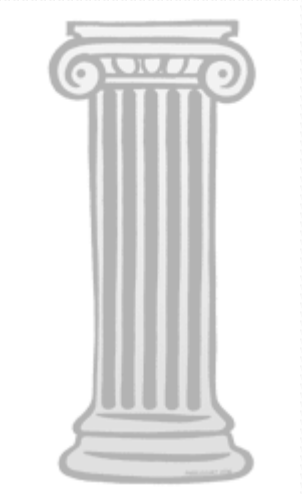
Polymorphism

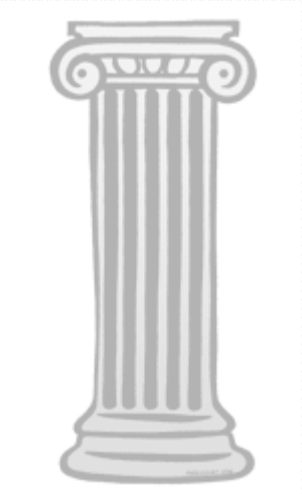
- Encapsulation is the process of hiding internal details of an object – like a “black box”
- Inheritance allows to create a new object based on an existing object (a form of code-reuse)
- Polymorphism means "many-shaped“

TODO:

What are the three pillars of object oriented programming?







_____ _____ _____

Exercise Inheritance

Extension Method

- Extends class without affecting existing code
- `this` keyword precedes the first parameter
- Defined as static method in any static class
- Specifies type that is extended as first parameter following the keyword `this`

TODO:

```
static class Helper
{
    // extension method
    public static double Mean(this IEnumerable<double> numbers)
    {
        return Averages.Mean(numbers);
    }
}
```

TODO:

```
static class Helper
{
    // extension method
    public static double Mean(this IEnumerable<double> numbers)
    {
        return Averages.Mean(numbers);
    }
}
```

Write a code statement that calculates the mean of myDataList and assigns it to a variable called result

TODO DEMO:

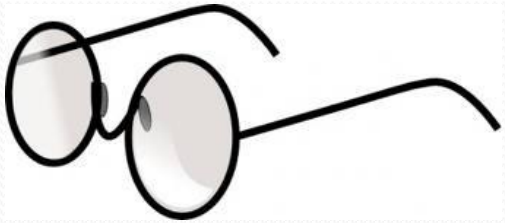
Mark the feature with **F** if properties share it with fields

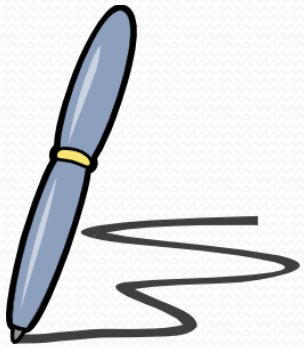
Mark the feature with **M** if properties shared it with accessor methods

Using Properties		
YES		Provide encapsulation
YES		Allow read only access
YES		Can provide input validation
YES		Can add additional logic
YES		Provide field-like syntax
YES		Concise

Properties

- Read-Only Properties vs Write-Only Properties





TODO:

- Create a public auto-implemented property of type int named Altitude.

  Users should have full read and write access.

TODO:

- Create 2 auto-implemented properties called AreaCode and PhoneNumber for a struct named Phone.
  Both properties are of type int.

_____

_____

Nullable Types

- Represent value-types that can also have value null
  - E.g. data from database that could be null
- Normal range of underlying value type + null value
- You cannot assign a nullable value to an ordinary value type variable

Null-coalescing Operator ??

- **??** is called the null-coalescing operator
- Defines default value for nullable value types and reference types.
- Returns left-hand operand if it is not null otherwise the right operand.

TODO:

e.g.:

```
int number = n1 ?? 5;

string str = s1 ?? “none”;
```

- What is the type of n1, s1 ?

TODO:

The range of `char?` is _____

TODO:

The range of `char?` is _____

```
char? c1 = 'x';
char? c2 = null;
char charA = c1 ?? 'y';
char charB = c2 ?? 'z';
```

What is the value of charA and charB?

- Nullable Exercise

Generic Collection

- Collections of specific types
- Major advantages:
  - Compile-time safety
  - Efficiency (eliminates need for boxing / unboxing)
- Also: less error-prone, code reuse

Generic Collections

- List<T>
- Queue<T>
- Stack<T>
- Dictionary<TKey, TValue>

TODO:

How can you tell whether a collection is generic or not?

_____.

What is the type of the elements contained in a **non**-generic collection class?

_____

TODO:

(List<T>, Stack<T>, Queue<T>)

Elements of _____ can be accessed like an array

(DictionaryEntry<Tkey, TValue>, KeyValuePair<Tkey, Tvalue>)

Each element of Dictionary<TKey, TValue> is an object of type

_____ .

TODO: Match each data collection on the left to a problem on the right

Dictionary <>	a) You need a variable representing people waiting in a ticket line
List<>	b) You need to store the name and height of the 10 highest mountains
Queue<>	c) You need to represent the towers of Hanoii
Stack<>	d) At any given time you need to know all the table numbers of those tables, that are currently free.

IEnumerable<T>	ICollection<T>	IList<T>	IDictionary<TKey, TValue>
	Count IsReadOnly	Count IsReadOnly Item	Count IsReadOnly Item Keys Values
GetEnumerator	GetEnumerator Add Clear Contains CopyTo Remove	GetEnumerator Add Clear Contains CopyTo Remove IndexOf Insert RemoveAt	GetEnumerator Add(T) Clear Contains CopyTo Remove(T) Add(TKey, TValue) ContainsKey Remove(TKey) TryGetValue

Used by collection initializer

List<T>

- Can be accessed like an array
- Resizable
- Compile-type safe

Abstract, Virtual, Override

- Abstract methods
  - Have no method body (no implementation)
  - Must be implemented in derived class
- Virtual methods
  - Provide default implementation
  - Can be overridden in derived class
- Override methods
  - Declare that this is an implementation of a class that was already declared in a base class (any ancestor)

TODO: abstract, virtual, override

Where are you more likely to see these keywords used – in classes or in structs?

_____

Why?

_____

Static methods

- Static methods
  - Method declaration is preceded by keyword static
  - A static member belongs to the type itself not an instance
  - Static methods have no direct access to instance members (fields, methods) only to static members

Abstract and static classes

- **TODO: What do you know about Abstract Classes**

Abstract and static classes

- Abstract Classes
  - Cannot be instantiated
  - Intended to be used as base class
  - May include members that are **not** abstract

Abstract and static classes

- Abstract Classes
  - Cannot be instantiated
  - Intended to be used as base class
  - May include members that are **not** abstract

- Static Classes
  - Cannot be instantiated
  - Implicitly sealed (i.e. cannot be used as base class)
  - All members have to be static

TODO: 4 methods: abstract, override, static, virtual

- Which of these 4 methods can not be overridden?
- Which two methods are always declared with an intend to be overridden
- Which one is always used to provide a default implementation?
- What is an implication of declaring a method abstract?

TODO: 4 methods: abstract, override, static, virtual

- Which of these 4 methods can not be overridden?
- Which two methods are always declared with an intend to be overridden
- Which one is always used to provide a default implementation?
- What is an implication of declaring a method abstract?
- How is an abstract class similar to a static class?
- How is an abstract class different from a static class?

Interfaces

- Like a contract
- Contain **no** implementation and **no** data
- When a class or struct implements an interface it commits to implement every member defined by the interface.
- A class can implement multiple interfaces
- Interface name should start with upper-case `I`

TODO:

You have a collection of type List<T> and you would like to sort it by calling the Sort method of List<T>.

This will work fine, if T implements

a) IComparable<T>

b) IDisposable<T>

c) IEnumerable<T>

d) ISortable<T>

TODO:

IComparable<T> has one single method.

- What is the name of the method?
- What is its return type?
- What happens if you use this method to compare an object to null?

Arrays

- Sequence of elements with same type
- Live in contiguous block of memory on heap
- Accessed by integer index in rectangular brackets
- Elements initialized with default values (0,false, null)
- Cannot be resized
- Array are zero indexed

TODO:

Circle the statement that declares a 2-dimensional integer array of 2 rows and 3 colums and initialize it with zeros.

a)
```
int[,] intArray = new int[3,2];
```

b)
```
int intArray[2,3] = new int[,];
```

c)
```
int[3,2] intArray = new int[,]);
```

d)
```
int[,] intArray = {{ 0, 0, 0 },
                   { 0, 0, 0 }};
```

Iterating with foreach

- Pro:
  - Clear and simple syntax
  - Iterates through all array elements
- Con:
  - No index
  - Read-only

Extension Method

- Extends class without affecting existing code

  The source code of a class is not needed in order to extend it.

- Defined as static method in a static class

- Specifies type that is extended as first parameter following the keyword `this`

TODO:

```
static class Helper
{
    // extension method
    public static double Fourth(this double number)
    {
        return number / 4;
    }
}
```

How many arguments need to be provided when this extension method is called?

TODO:

- Which of the following is a correct method call of the extension method Fourth?

a) `double.Fourth(number);`

b) `number.Fourth(number);`

c) `number.Fourth(4);`

d) `number.Fourth();`

THE END