C# Basics

Course Code: CSC 3115 Course Title: Object Oriented Programming 2

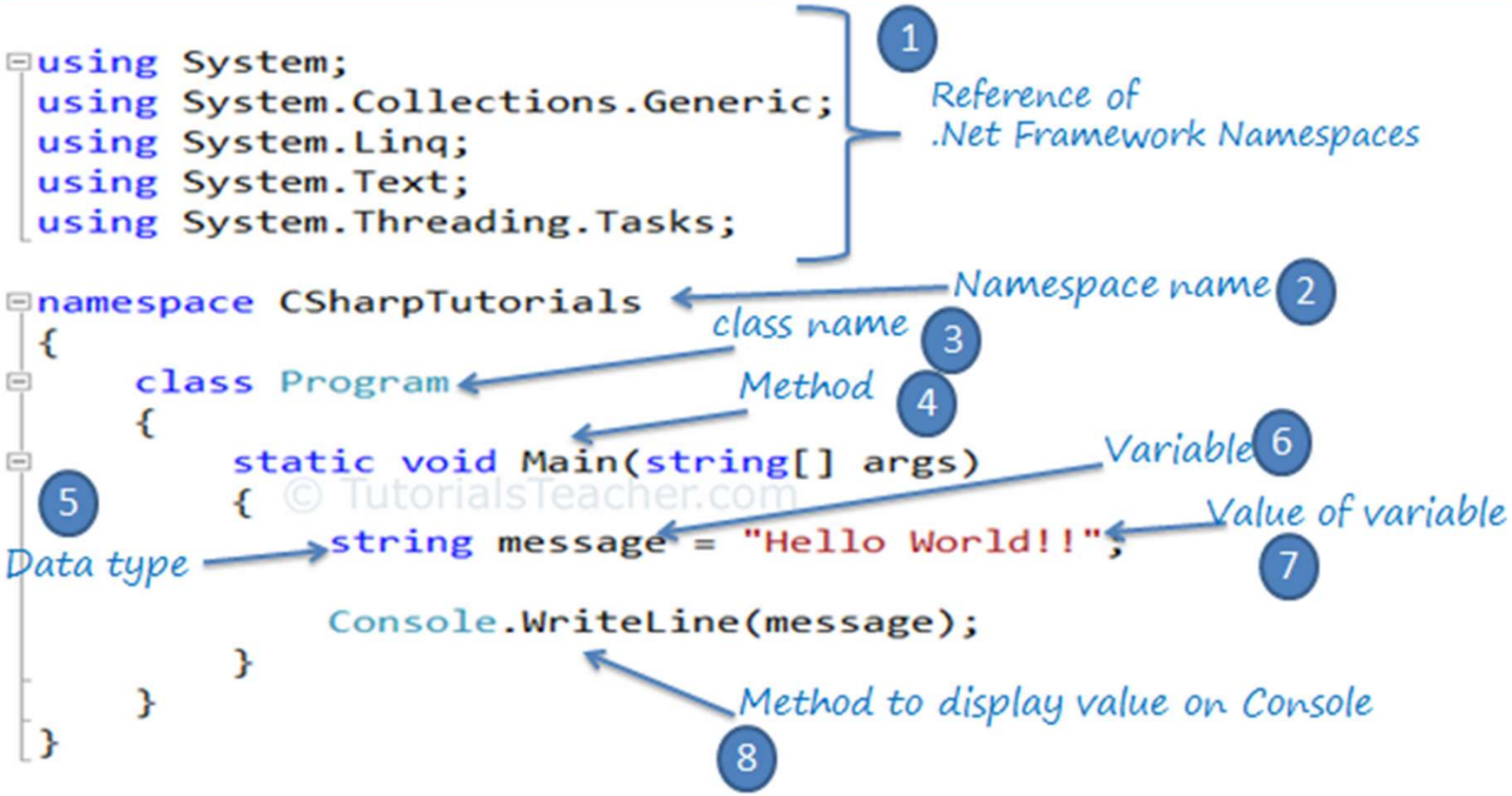
Dept. of Computer Science
Faculty of Science and Technology

Lecture No:	**02**	**Week No:**	**02**	**Semester:**	
Lecturer:					

Topics

- C# Data Types
- Type Casting
- The keywords readonly and const
- Structures
- Enumeration
- String
- Use of params, ref, out
- Understand the concept of Class and Object
- Learn about constructors
- Variable Types

Simple Console Project with C#



THE C# BASICS

- C# is a strongly-typed language. Every variable and constant has a type, as does every expression that evaluates to a value.

- Every method signature specifies a type for each input parameter and for the return value.

TYPES, VARIABLES, AND VALUES

- The .NET Framework class library defines a set of built-in types as well as more complex types that represent a wide variety of logical constructs, such as the file system, network connections, collections and arrays of objects, and dates.
- A typical C# program uses types from the class library as well as user-defined types that model the concepts that are specific to the program's problem domain.

TYPES, VARIABLES, AND VALUES

The information stored in a type can include the following:

- The storage space that a variable of the type requires.
- The maximum and minimum values that it can represent.
- The members (methods, fields, events, and so on) that it contains.
- The base type it inherits from.
- The kinds of operations that are permitted.

TYPES, VARIABLES, AND VALUES

- The compiler uses type information to make sure that all operations that are performed in your code are *type safe*.

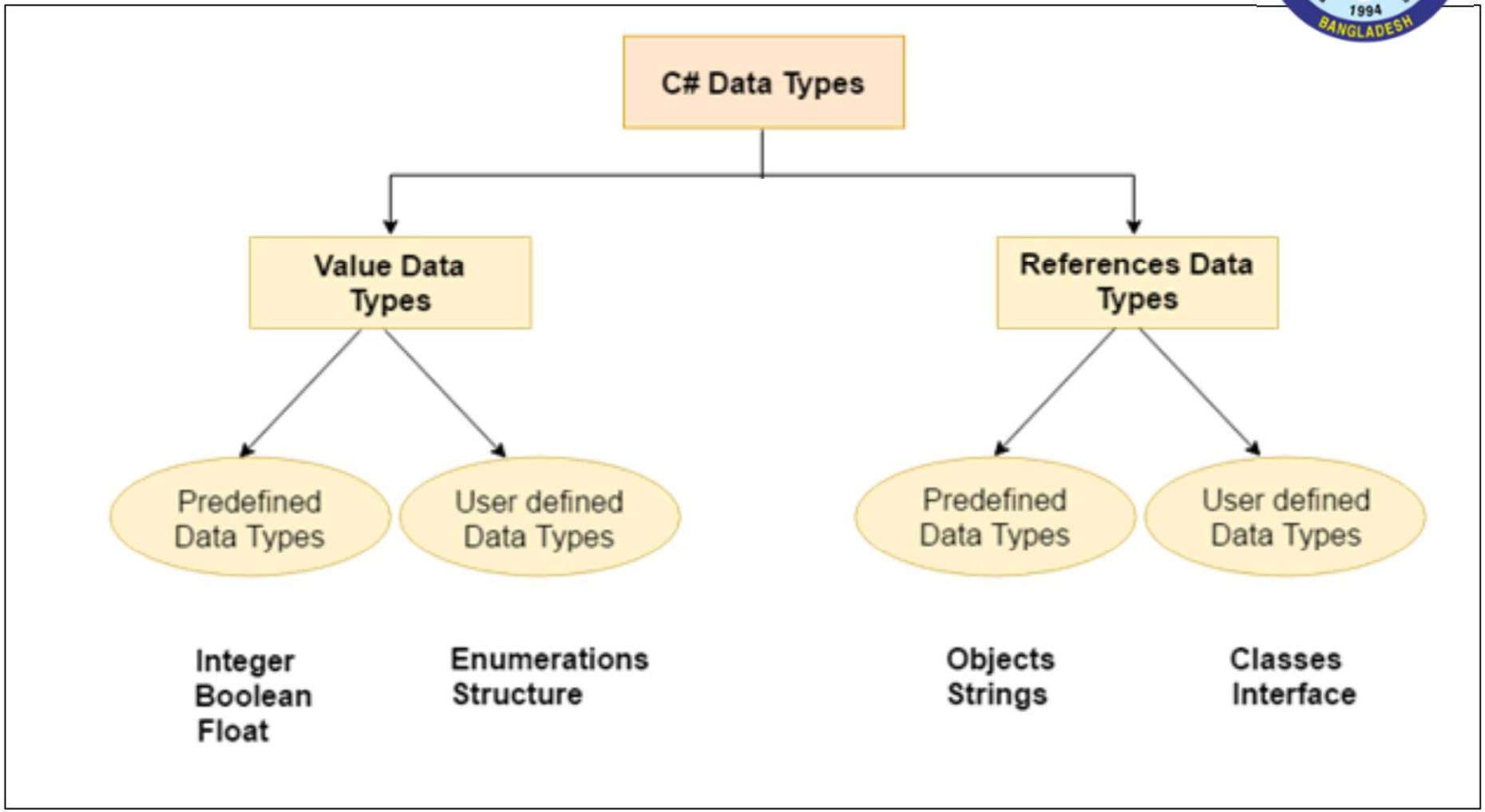
```
int a = 5;
int b = a + 2; //OK
bool test = true;
int c = a + test; // Error. Operator '+' cannot be
                     applied to operands of type 'int'
                     and 'bool'.
```

Note that in C#, bool is not convertible to int.

SPECIFYING TYPES IN VARIABLE DECLARATIONS

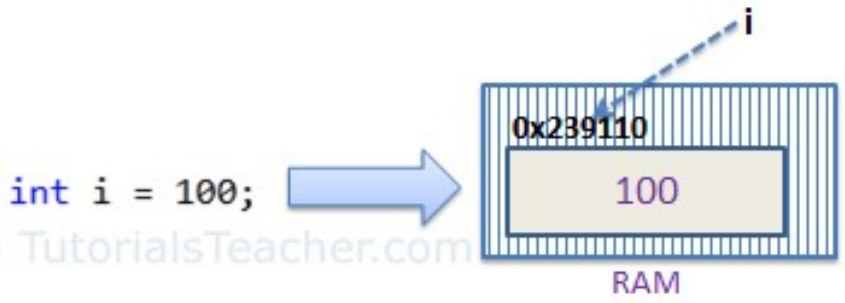
- When you declare a variable or constant in a program, you must either specify its type or use the var keyword to let the compiler infer the type.

```
// Declaration only:
float temperature;
string name;
MyClass myClass;
// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source where item <= limit select item;
```


C# Data Types
Value Data Types
References Data Types
Predefined Data Types
User defined Data Types
Predefined Data Types
User defined Data Types
Integer
Boolean
Float
Enumerations
Structure
Objects
Strings
Classes
Interface

VALUE TYPES

- A data type is a value type if it holds a data value within its own memory space.
- It means variables of these data types directly contain their values.
- For example,



N.B. All the value types derive from *System.ValueType*, which in-turn, derives from *System.Object*.

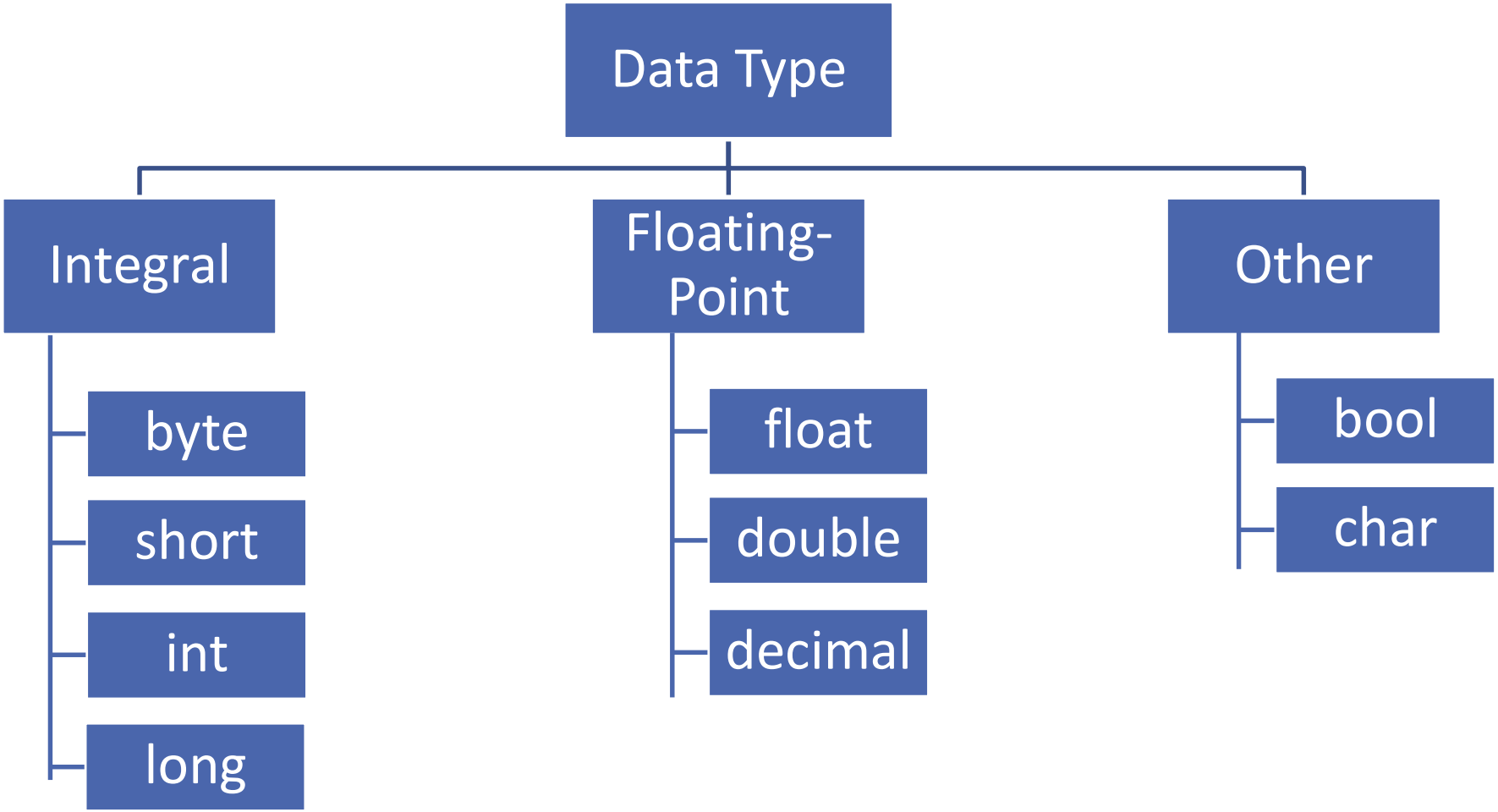
VALUE TYPES

- The value data types are integer-based and floating-point based. C# language supports both signed and unsigned literals.
- There are 2 types of value data type in C# language.

    1) Predefined Data Types - such as Integer, Boolean, Float, etc.

    2) User defined Data Types - such as Structure, Enumerations, etc.

N.B. All the value types derive from *System.ValueType*, which in-turn, derives from *System.Object*.

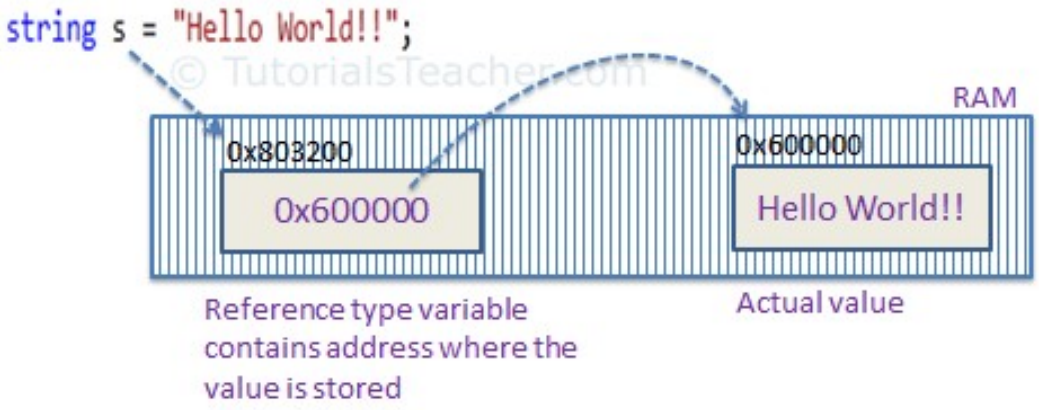

Data Type
Integral
Floating-Point
Other
byte
short
int
long
float
double
decimal
bool
char

VALUE TYPES

DATA TYPES	SIZE	VALUES
sbyte	8 bit	-128 to 127
byte	8 bit	0 to 255
short	16 bit	-32,768 to 32,767
ushort	16 bit	0 to 65,535
int	32 bit	-2,147,483,648 to 2,147,483,647
uint	32 bit	0 to 4,294,967,295
long	64 bit	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	64 bit	0 to 18,446,744,073,709,551,615
char	16 bit	0 to 65535
float	32 bit	-1.5 x 1045 to 3.4 x 1038
double	64 bit	-5 x 10324 to 1.7 x 10308
decimal	128 bit	-1028 to 7.9 x 1028
bool	—	True or false

REFERENCE TYPES

- Unlike value types, a reference type doesn't store its value directly.
- Instead, it stores the address where the value is being stored.
- In other words, a reference type contains a pointer to another memory location that holds the data.



REFERENCE TYPES

- The reference data types do not contain the actual data stored in a variable, but they contain a reference to the variables.
- If the data is changed by one of the variables, the other variable automatically reflects this change in value.
- There are 2 types of reference data type in C# language.

  1) Predefined Types - such as Objects, String.

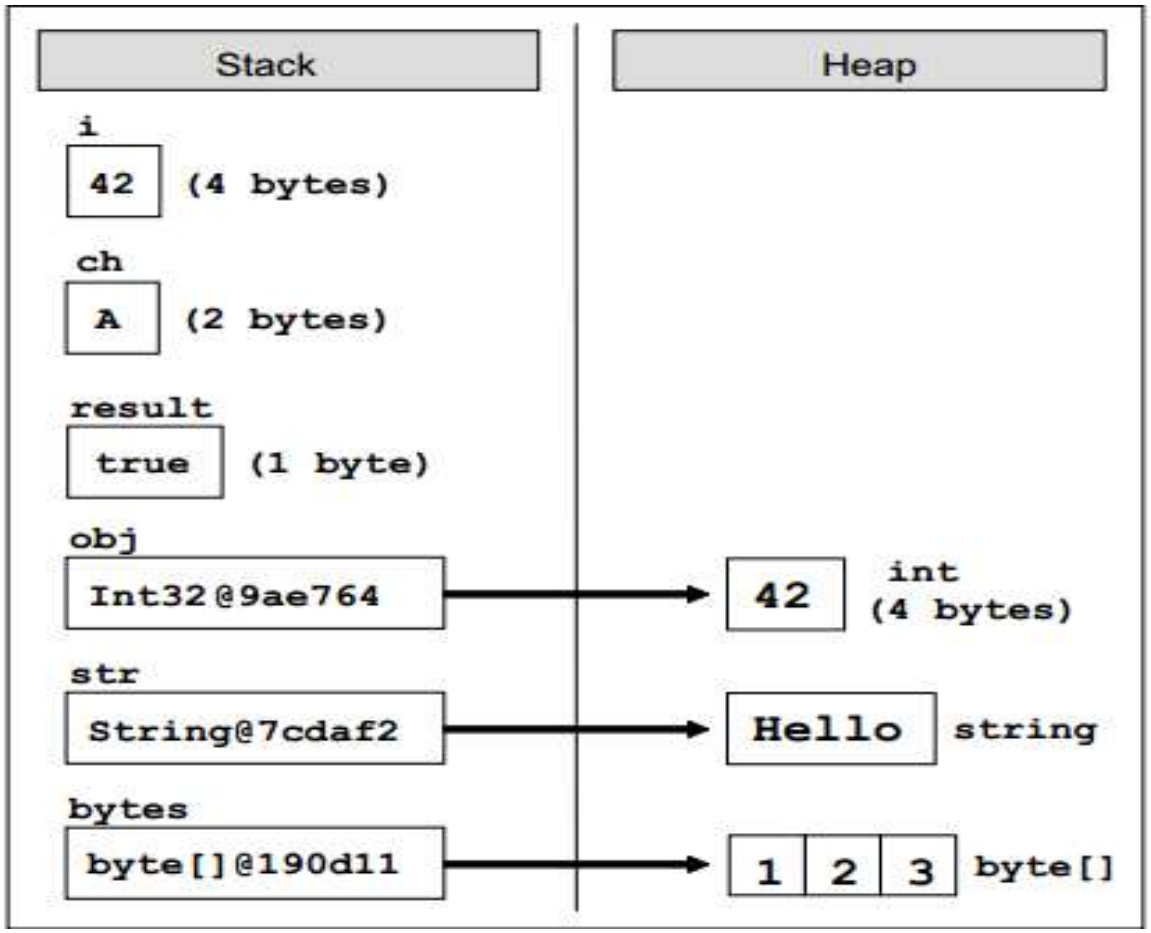
  2) User defined Types - such as Classes, Interface.

Value and Reference Types and the Memory

In this example we will illustrate how value and reference types are **represented in memory**. Consider the execution of the following programming code:

```
int i = 42;
char ch = 'A';
bool result = true;
object obj = 42;
string str = "Hello";
byte[] bytes ={1,2,3};
```

At this point the variables are located in the memory as follows:

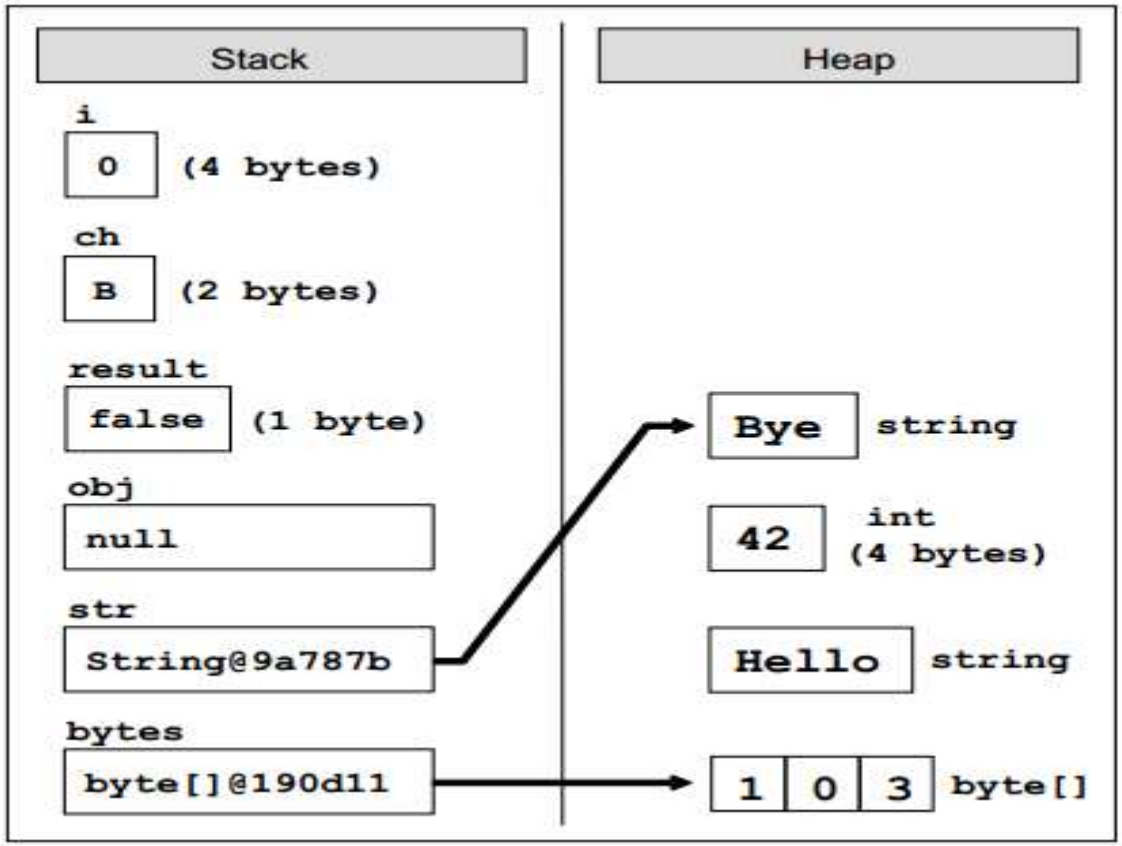


Value and Reference Types and the Memory

If we now execute the following code, which changes the values of the variables, we will see **what happens to the memory** when changing the value and reference types:

```
i = 0;
ch = 'B';
result = false;
obj = null;
str = "Bye";
bytes[1] = 0;
```

After these changes the variables and their values are **located in the memory** as follows:



THE COMMON TYPE SYSTEM

- It is important to understand two fundamental points about the type system in .NET

1→

- It supports the principle of inheritance. Types can derive from other types, called base types. The derived type inherits (with some restrictions) the methods, properties, and other members of the base type.
- The base type can in turn derive from some other type, in which case the derived type inherits the members of both base types in its inheritance hierarchy.
- All types, including built-in numeric types such as System.Int32 (C# keyword: int), derive ultimately from a single base type, which is System.Object (C# keyword: object).
- This unified type hierarchy is called the Common Type System (CTS).

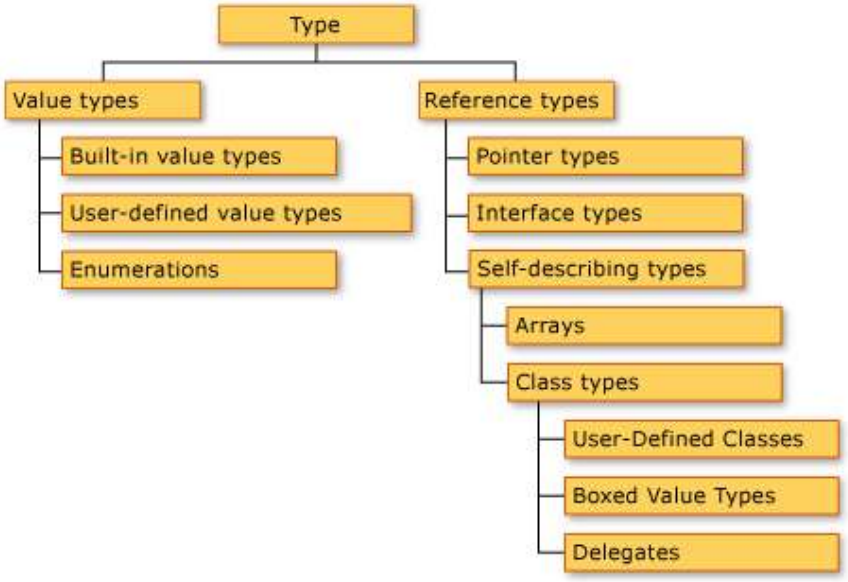
THE COMMON TYPE SYSTEM

2→

- Each type in the CTS is defined as either a value type or a reference type. This includes all custom types in the .NET class library and also your own user-defined types.
- Reference types and value types have different compile-time rules, and different run-time behavior.

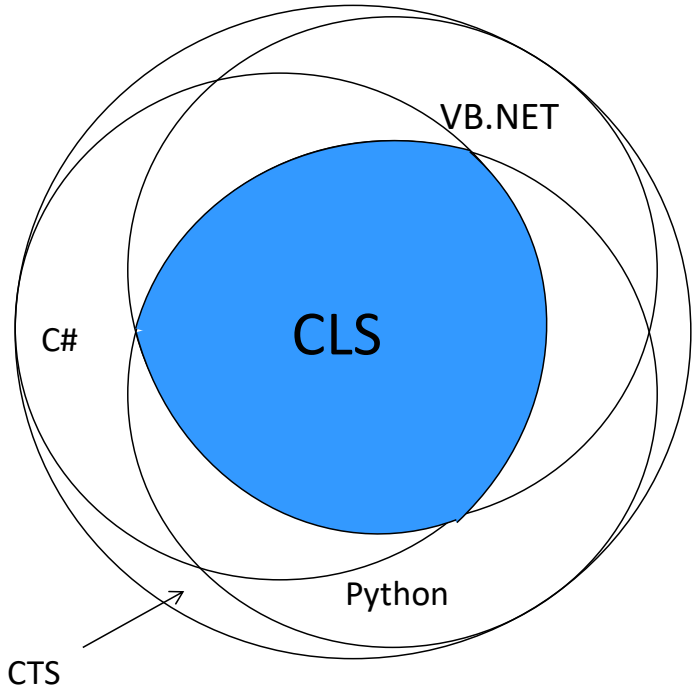
The Common Type System

To support multiple programming languages on a single CLR and have the ability to reuse the FCL, the types of each programming language must be compatible. This binary compatibility between language types is called the Common Type System (CTS).



- Defines what is a “type”
  - How types are declared, used, managed by the CLR
  - In-memory model
    - Value types VS Reference types
  - Meta-Model
- Enables cross language interoperability

Common Language Specification (CLS)



- Subset of CTS
- Defines set of CTS features that all languages must implement.
- Important for cross-language interoperability

Common Language Specification (CLS)

- For programming languages to communicate effectively, there must be a common set of standards to which every .NET language must support. This common set of language features is called the **Common Language Specification (CLS)**.

- Most .NET compilers can produce both CLS-compliant and non-CLS-compliant code, and it is up to the developer to choose which language features to use.

- For example, C# supports unsigned types, which are non-CLS compliant. For CLS compliance, you can still use unsigned types within your code so long as you don't expose them in the public interface of your code, where code written in other languages can see.

Common Language Specification (CLS)

- Example:
  - unsigned int available in C# but not available in VB.NET

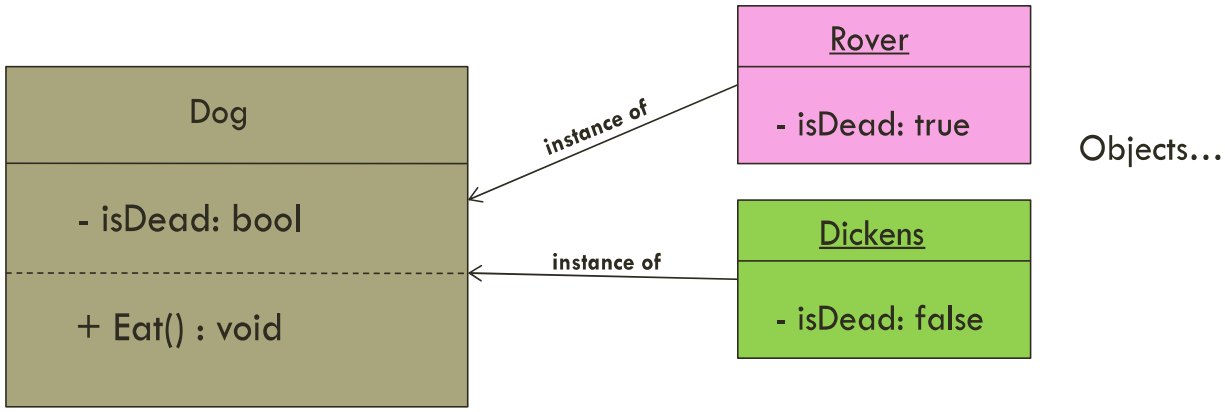
```
public class NotCLSCompliant {
  public uint SomeProperty
  { … }
}

public class CLSCompliant {
  private uint SomeProperty
  { … }
}
```

TYPES

Types are the basis for objects (instances of types)

Objects have “state” + “behaviour”

Objects have “identity”



Types

- Types have "names"
- Types are organized in "namespaces"

```
namespace Zoo {
    namespace Mammal {
        class Dog {…}
        class Cat {…}
    }
}
```

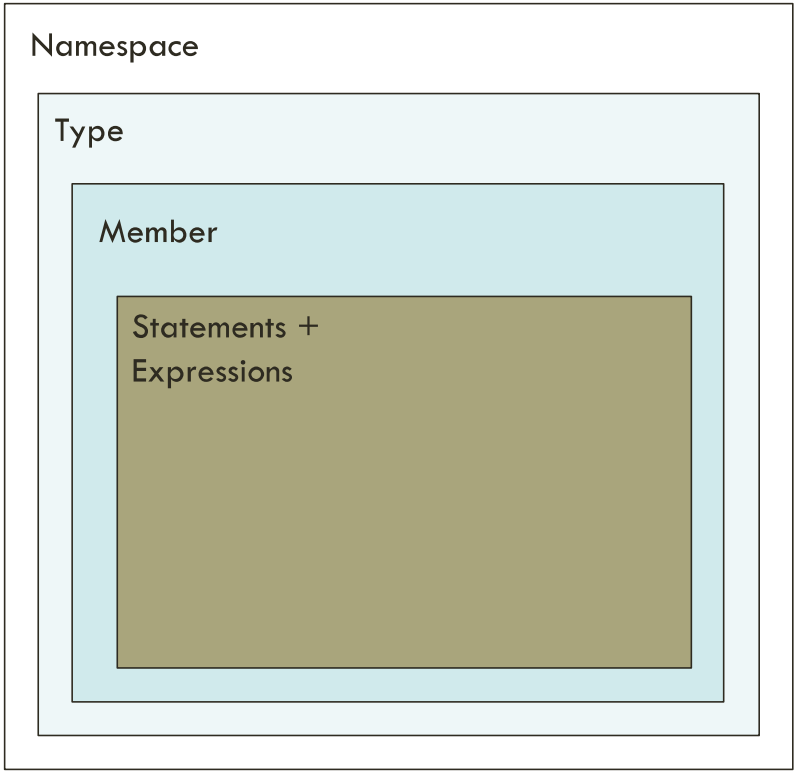
Q: Fully qualified name of Dog?

A: Zoo.Mammal.Dog

Types

- Types have “members”
- There are different types of members
  - Fields, Methods, Constructors, Properties, Events, Indexers, other Types... More…
- Members can be *static* or *instance*
  - Static members are associated with the Type
  - Instance members are associated with the instance of a given Type (e.g. object)
- Members always execute within some “context”
  - static or instance
  - This is a very important concept…

TYPES



a well-defined hierarchy

e.g. no members at Namespace level, no statements at the Type level)…

CASTING AND TYPE CONVERSIONS

- Because C# is statically-typed at compile time, after a variable is declared, it cannot be declared again or used to store values of another type unless that type is convertible to the variable's type.
- For example,

```
int i;
i = "Hello"; // Error: "Cannot implicitly
                convert type 'string' to 'int'"
```

CASTING AND TYPE CONVERSIONS

- However, we might sometimes need to copy a value into a variable or method parameter of another type.
- For example, we might have an integer variable that you need to pass to a method whose parameter is typed as double.
- Or we might need to assign a class variable to a variable of an derived type.
- These kinds of operations are called type conversions.

CASTING AND TYPE CONVERSIONS

In C#, we can perform the following kinds of conversions:

- **Implicit conversions**: No special syntax is required because the conversion is type safe and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.

- **Explicit conversions (casts)**: Explicit conversions require a cast operator. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range, and conversion of a base-class instance to a derived class.

- **Conversions with helper classes:** To convert between non-compatible types, such as integers and strings and byte arrays, we can use **Parse methods** of the built-in numeric types, such as Int32.Parse.

IMPLICIT CONVERSIONS

- For built-in numeric types, an implicit conversion can be made when the value to be stored can fit into the variable without being truncated or rounded off.
- For example,

```
// Implicit conversion. num long can
// hold any value an int can hold, and more!
int num = 21474836;
long bigNum = num;
```

- For reference types, an implicit conversion always exists from a class to any one of its direct or indirect base classes. No special syntax is necessary because a derived class always contains all the members of a base class.

```
Derived d = new Derived();
Base b = d; // Always OK.
```

EXPLICIT CONVERSIONS

- However, if a conversion cannot be made without a risk of losing information, the compiler requires that you perform an explicit conversion, which is called a *cast*.
- A cast is a way of explicitly informing the compiler that you intend to make the conversion and that you are aware that data loss might occur.
- To perform a cast, specify the type that you are casting to in parentheses in front of the value or variable to be converted.

```
class Test
{   static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

EXPLICIT CONVERSIONS

- For reference types, an explicit cast is required if you need to convert from a base type to a derived type:

```
// Create a new derived type.
Giraffe g = new Giraffe();
// Implicit conversion to base type is safe.
Animal a = g;
// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe) a;
```

TYPE CONVERSION EXCEPTIONS AT RUN TIME

- In some reference type conversions, the compiler cannot determine whether a cast will be valid. It is possible for a cast operation that compiles correctly to fail at run time.

CAST BY USING **AS** AND **IS** OPERATORS

Look at the example given below:

```
Circle c = new Circle(32);
object o = c;

int i = (int)o;      // it compiles okay but throws an exception at runtime
```

CAST BY USING AS AND IS OPERATORS

- Because objects are polymorphic.
- However, to attempt a simple cast in these cases creates the risk of throwing an InvalidCastException. That is why C# provides the is and as operators.
- You can use these operators to test whether a cast will succeed without causing an exception to be thrown.
- 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 returns only a Boolean value. It can therefore be used when you just want to determine an object's type but do not have to actually cast it.

CAST BY USING **AS** AND **IS** OPERATORS

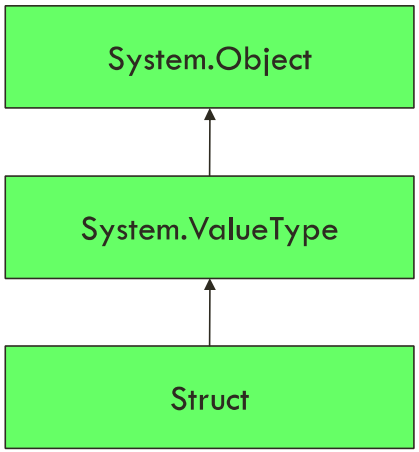
- **is Operator**
- The "is" operator is used to check whether the run-time type of an object is compatible with a given type or not. In other words, we use the "is" operator to verify that the type of an object is what we expect it to be.

- **as Operator**
- The "as" operator is used to perform conversions between compatible types. Actually, the "as" operator returns the cast value if the cast can be made successfully.

STRUCTS

```
struct Point {
 public int X
 {get; set;}
 public int Y
 {get; set;}
 public double DistanceFrom(Point p)
 {... }
}
```

STRUCTS



Although Structs do not support inheritance, they implicitly derive from **System.ValueType**, which in turn derives from **System.Object**

STRUCTS

Structs can have the following members

- Constructors (can't have Destructors)
- Fields
- Methods
- Indexers
- Properties
- Events
- Other types (structs, enums, classes, interfaces...)

Structs can implement interfaces

Structs can't inherit from other classes

CONSTANTS

A ***constant*** is an object whose value can't be changed. The are used to prevent reassignment

- Attempt to reassign a constant results in compile error

Constants come in three flavors:

- *Literals*
- *Symbolic constants*
- *Enumerations*

Literals *example:*

- *e.g. 33 is a literal you cannot change its value*

Symbolic constants assign a name to a constant value. You declare a symbolic constant using the **const** keyword and the following syntax:

const *type identifier = value;*

CONSTANTS (CONTINUE)

You must initialize a constant when you declare it, and once initialized, it can't be altered

- E.g.

```
const int FreezingPoint = 32;
```

In this declaration, 32 is a literal constant, and FreezingPoint is a symbolic constant of type int

As a matter of style, constant names are typically written in Pascal notation or all caps.

CONSTANTS - ENUMERATIONS

An **enumeration** is a distinct value type, consisting of a set of named constants (called the ***enumerator list***)

Every enumeration has an underlying type, which can be any integral type (integer, short, long, etc.) except for char

An enumeration begins with the keyword **enum**, which is followed by an identifier

Syntax

[*attributes*] [*modifiers*] **enum** *identifier*[:*base-type*]

{*enumerator-list***};**

Default *base-type is* **int**

Example:

```
enum ServingSizes :units
{
Small = 1,
Regular = 2,
Large = 3
}
```

STRINGS

A string object holds a series of characters.

You declare a string variable using the **string** keyword much as you would create an instance of any object:

```
string myString;
```

You create a **string literal** by placing double quotes around a string of letters:

```
"Hello World"
```

It is common to initialize a string variable with a string literal:

```
string myString = "Hello World";
```

IDENTIFIERS

An **identifier** is just the name the programmer chooses for the types, methods, variables, constants, objects, and so on in the program

An identifier must begin with a letter or an underscore

Identifiers are case-sensitive, so C# treats someName and SomeName as two different identifiers

- The Microsoft naming conventions suggest using **camel** notation (initial lowercase, such as someVariable) **for variable names**, and **Pascal** notation (initial uppercase, such as SomeMethodOrProperty) **for method names and most other identifiers**

Class & Object

- **What is the class**? – blueprint from which an object can be built
- A class can be thought of as a "type", with the objects being a "variable" of that type
- **Advantage of classes** in OO programming is that they encapsulate the characteristics and capabilities of an entity within a single, self-contained unit of code
- Complete **syntax** for a class definition:

```
[attributes] [access-modifiers] class identifier [:base-class [, interface(s)]]
{
class-body
}
```

Note: only highlighted in bold are mandatory

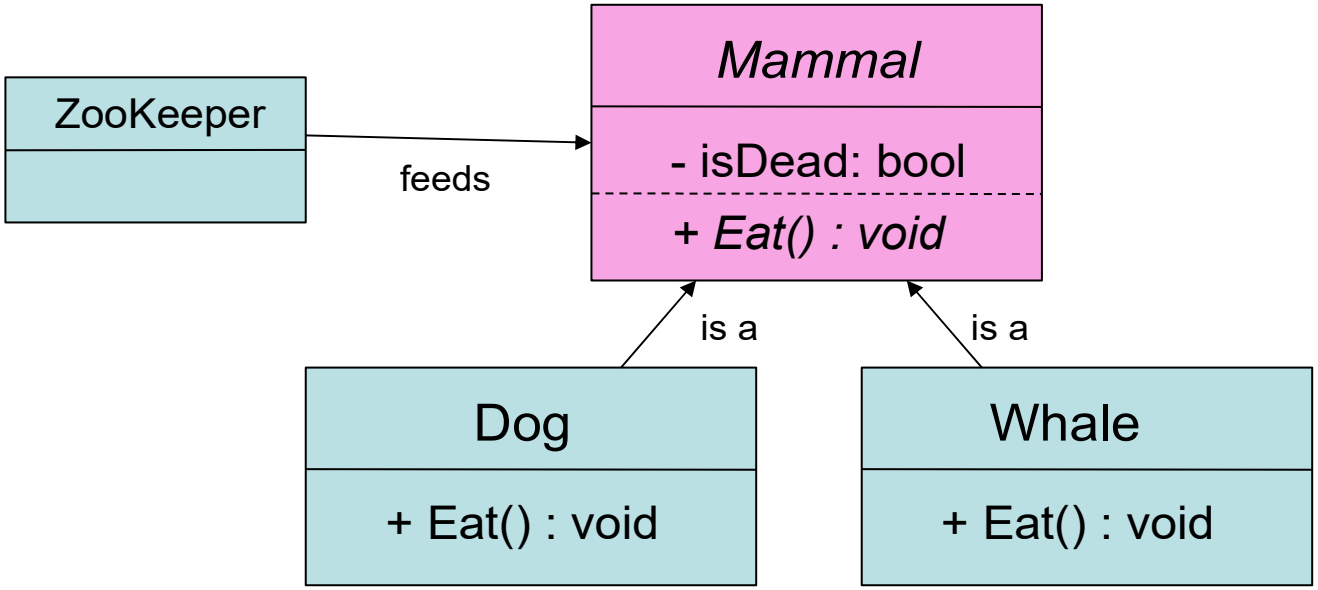
- Syntax to create an instance of a class (i.e. object)

```
Time t = new Time()
```

Consider: Day3of7 / TimeClass
Day3of7 / MyTime
Day3of7 / Time

Classes

- Classes are reference types
- Two fundamental types of classes
  - **concrete** → you can create instances of them
  - **abstract** → you CAN'T create instances of them
- Abstract classes are meant to be inherited from...

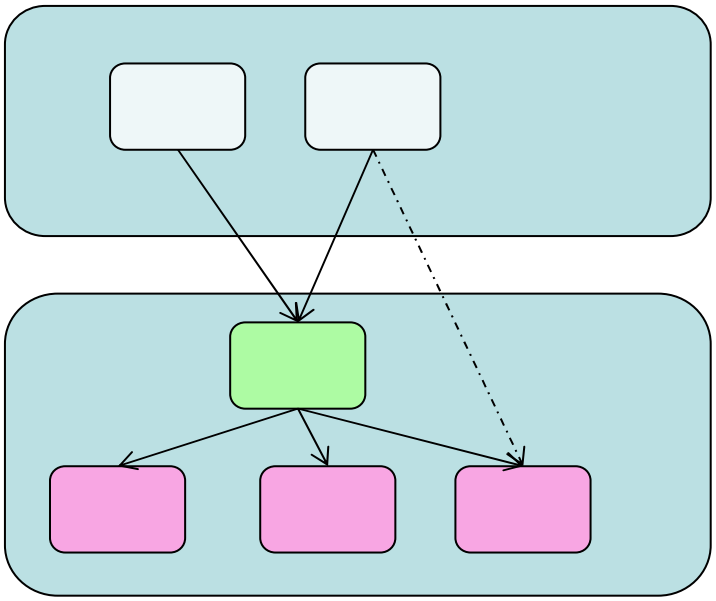


Classes

- Classes can have the following members
  - Constructors/Destructors
  - Fields
  - Methods
  - Indexers
  - Properties
  - Delegates and Events
  - Other types (e.g. structs, enums, interfaces, classes...)
- Classes can implement interfaces and inherit from other classes

Access Modifiers

- Access modifiers control the "visibility" of a type or member to other types and members
- They are key to encapsulation...
- There are defaults if you do not specify an access modifier; providing more restrictive access



- pink is not visible to light blue but visible to green
- green is visible to light blue

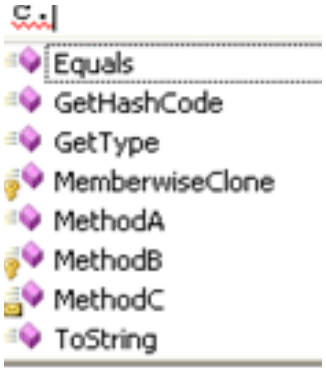
Access Modifiers

Access Modifier	Restrictions
public	• **Applicable** to types and members • No restrictions. **Accessible** to all...
private	• **Applicable** only to members • A private member is **accessible** to only other members in the same class • A **private** is the **default** access for members
protected	• **Applicable** only to members (and types that are members – defined in another type) • **Accessible** to other members in the same class and sub-class
internal	• **Applicable** to types and members • **Accessible** to other types and members in the same assembly • **internal** is **default** access to class
protected internal	• is a **union** of **protected** and **internal**

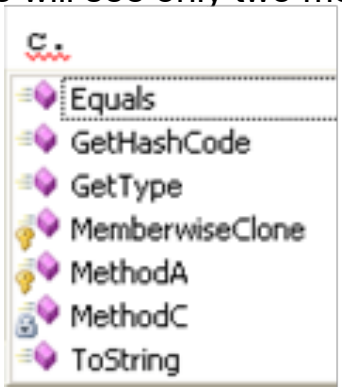
Access Modifiers

Understanding protected

```
class A
  {
     protected void MethodA()
     {
     }
  }
 class B : A
  {
     protected void MethodB()
     {
     }
  }
class C:B
 {
    private void MethodC()
    {
       C c = new C();
    }
  }
```



If we will change access modifier of MethodB() from **protected** to **private** then we will see only two methods available:



Access Modifiers

- **Understanding internal**
- **Accessible** to other types and members in the same assembly i.e. only within the same project in Visual Studio
- By default class has internal access (to see it create class diagram and look at property window)
- It cannot be accessed from any other assembly (or corresponding project) even though it is referenced from this assembly (project)

Consider: Day3of7/AccessModifiersTest
Day3of7/AccessModifiersInternal

Members

- **Constructors**
  - puts an instance of some type into a "well-known" state
- **Destructors**
  - releases non-memory resources (e.g. file handles, network sockets, database connections, etc)
- **Fields**
  - the state of objects
- **Properties**
  - provides read/write access to an object's state
- **Indexers**
  - like properties but are "parameterful"
- **Events**
  - enables other objects to "subscribe" to another object and be notified where something interesting happens...
- **Types**
  - types defined within another type...
  - makes sense when the "inner" type only makes sense within the context of the "outer" type

Members - Constructors

- Constructors are **methods** invoked whenever we instantiate an object
- Explicitly specified constructor must have **the same name as class name**
- If we not specify, then default constructor is called
- After constructor runs memory holds a valid instance of that class type
- As any other method constructor can be overloaded

CONSTRUCTORS IN C#

- A special method of the class that will be automatically invoked when an instance of the class is created is called a constructor.
- The main use of constructors is to initialize fields of the class while creating an instance for the class.
- When you have not created a constructor in the class, the compiler will automatically create a default constructor in the class.
- The default constructor initializes all numeric fields in the class to zero and all string and object fields to null.

KEY POINTS REGARDING THE CONSTRUCTOR ARE:

- A class can have any number of constructors.
- A constructor doesn't have any return type, not even void.
- A static constructor can not be a parametrized constructor.
- Within a class you can create only one static constructor.

Types of Constructors in C#

1. Default or Parameter less Constructor.
2. Parameterized Constructor.
3. Copy Constructor.
4. Static Constructor.

Default Constructor

- A constructor without any parameters is called a default constructor; in other words this type of constructor does not take parameters.

- The drawback of a default constructor is that every instance of the class will be initialized to the same values and it is not possible to initialize each instance of the class to different values.

- The default constructor initializes:
  - ✓ All numeric fields in the class to zero.
  - ✓ All string and object fields to null.

Parameterized Constructor

- A constructor with at least one parameter is called a parametrized constructor.

- The advantage of a parametrized constructor is that you can initialize each instance of the class to different values.

Parameterized Constructor

- A constructor with at least one parameter is called a parametrized constructor.

- The advantage of a parametrized constructor is that you can initialize each instance of the class to different values.

Copy Constructor

- The constructor which creates an object by copying variables from another object is called a copy constructor.
- The purpose of a copy constructor is to initialize a new instance to the values of an existing instance.

Static Constructor

- When a constructor is created as static, it will be invoked only once for all of instances of the class .
- A static constructor is used to initialize static fields of the class and to write the code that needs to be executed only once.

Some key points of a static constructor are:

- A static constructor does not take access modifiers or have any parameters.
- A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.
- A static constructor cannot be called directly.
- The user has no control on when the static constructor is executed in the program.

Members - Initializers

- Member variables can be initialized by initializer instead of constructor
- We can initialize an object without constructor by specifying public (!) member variables in object initializer (NOTE: curly braces in this case)
- ***this* keyword** refers to the current instance of an object
- ***this*** keyword is used:
  - To distinguish parameters from instance members (e.g. this.hour = hour;)
  - To pass the current object as a parameter

Members - Overloading Methods and Constructors

- Overloading – is an ability to have more than one method with the same name, but with different **signatures** (and, optionally, return type).
- NOTE: changing only return type does NOT overload the method.
- Method **signature** – is a method name and parameter list (including parameters' types)

Destructors

- Are methods called by the Garbage Collector thread when the object is to be "finalized"
- looks like C++ destructors but are completely different
- C# destructors are "non-deterministic" i.e. finalization (removing objects from memory) cannot be determined when that will happen
- whereas C++ destructors are invoked in a "deterministic" manner (see garbage collector)

Destructors

In C#, most of the time, you do not need to code destructors (or finalizers) because you can trust GC to clean up for you with destructors.

These methods are **called implicitly** by the C# runtime system's garbage collector.

```
class TypeName
{
    // implicit destructor ~TypeName()
    // created for you automatically by C# even if you did not put it in the code
}
```

NOTE: explicit or implicit destructor

```
~TypeName()
{
    body...
}
```

actually gets converted by compiler to

```
protected override void Finalize()
{
    try
    {
        body...
    }
    finally
    {
        base.Finalize();
    }
}
```

Destructors

- C# is a "managed" language (e.g. CLR manages memory)
- So what should be released in destructors?
- Not memory (GC will handle that...)
- You should release other limited resources
  - File handles
  - Network sockets
  - Database connections
  - etc...
- But destructors come at a cost... (see garbage collection)

Members: Fields

- Variables declared **outside any method**/constructor but **inside the class** block, is a **Field** (or **field variable)**
- Variables that are declared within a method or a specific block of statements are **local variables**
- **Local Variables** are kept alive as long as the execution is within the block they're defined in. Once the block is exited, the local variables can no more be used.
- **Field Variables** have longer life than local variables in that they can live as long as the instance they belong to is active
- Field variable is the one which can have different access level whereas local variable access is restricted to method only
- **Field Variable** is a member of a class whereas **Local Variable** is a member of a method

Members: Readonly Fields

- The keyword **readonly** will prevent **field variable** from being reassigned (from outside the class)
- A value can be assigned to readonly fields in 2 cases:
    1. In a field declaration
    2. In a constructor of the same class
- The **readonly** keyword is different from the **const** keyword. A **const** field can only be initialized at the declaration of the field. A **readonly** field can be initialized either at the declaration or in a constructor. Therefore, **readonly** fields can have different values depending on the constructor used.
- Also, while a **const** field is a **compile-time constant**, the **readonly** field can be used for **runtime constants**
- The **readonly** keyword cannot be applied to local variables but to fields only

Members: Properties

Properties allow accessing the state of the object as though they were accessing member directly, while actually implementing that access through a class method.

Syntax:

```
public int Hour // this is a property name
{
      get
      {
        return hour;
      }

      set
      {
        hour = value;
      }
}
```

NOTE: in this example " hour" is so called "backing field" – private member variable to store the property value

NOTE: To access the property we use "dot" (access) operator on instance.

Members: Automatic Properties

- **Automatic Properties** is just shorthand notation for properties declaration

Syntax:

public int Hour { **get; set;** }

- NOTE: In auto properties you cannot code implementations of accessor, the compiler automatically generates the underlying code and backing fields
- In this case, to "shut down" either getter or setter just declare it as private
- e.g. **public int** Hour { **get**; **private set**; } – **this is read only property because only getter is accessible**
- e.g. **public int** Hour {**private get**; **set**; } – **this is write only property because only setter is accessible**

Static vs. Instance

- All members are associated with some context
  - static or instance
- When you invoke a method it executes within a context

```
DoSomething(); //what is the context?

this.DoSomething(); //is it clearer?


SomethingElse(); //what is the context?

//is it clearer?

static void Main() { SomethingElse(); }
```

Structs

Structs are:

- lightweight alternative to class
- CAN implement interfaces
- Is a value type
- Implicitly SEALED
- NO DEFAULT CONSTRUCTOR
- A struct cannot inherit from another struct or class, and it cannot be the base of a class.
- **Structs, however, inherit from the base class object (System.Object)**

Consider: Day3of7/UsingStructs project

- [attributes] [access-modifiers] **struct** identifier [: interface-list]
- {
- struct-members
- }
- When you create a struct object using the **new** operator, it gets created and the appropriate constructor is called.
- (E.g. CoOrds coords2 = new CoOrds(10, 10);)

Structs

- Unlike classes, structs can be instantiated without using the **new** operator. In such a case, there is no constructor call, which makes the allocation more efficient. However, the fields will remain unassigned and the object cannot be used until all of the fields are initialized
- The example below demonstrates a feature that is unique to structs. It creates a CoOrds object without using the **new** operator. If you replace the word **struct** with the word **class**, the program will not compile

  Conclusion:
  - Structs are simple to use and can prove to be useful at times. Just keep in mind that they're created on the stack and that you're not dealing with references to them but dealing directly with them.
  - **Whenever you have a need for a type that will be used often and is mostly just a piece of data, structs might be a good option.**
  - **Complex numbers, points in a co-ordinate systems etc are good examples for struct types.**

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