# Abstraction

Course Code: CSC 2210 Course Title: Object Oriented Programming 2


**Dept. of Computer Science**
**Faculty of Science and Technology**

| | | | | | |
|---|---|---|---|---|---|
| **Lecturer No:** | **06** | **Week No:** | **06** | **Semester:** | |
| **Lecturer:** | | | | | |

# Topics


1. Abstract Types
2. Abstract Method
3. Nullable Type

# Abstraction


**Abstraction** is a principle of object-oriented programming and it is used to hide the implementation details and display only essential features of the object.


# WHAT IS AN ABSTRACT CLASS?

- An abstract class is a special kind of class that cannot be instantiated.
- An abstract class is only to be sub-classed (inherited from).
- The advantage is that it enforces certain hierarchies for all the subclasses.


# Nullable Type

- As you know, a value type cannot be assigned a null value.
- For example, **int i = null** will give you a compile time error.
- C# 2.0 introduced nullable types that allow you to assign null to value type variables.
- You can declare nullable types using **Nullable<T>** where **T is a type**.
- Syntax of Nullable type:

```
Nullable<int> i = null;
```


# Nullable Type

- A nullable type can represent the correct range of values for its underlying value type, plus an additional *null* value.
- For example, **Nullable<int>** can be assigned any value from

  -2147483648 to 2147483647, or a null value.

# Nullable Type

## Example: HasValue

```
static void Main(string[] args)
{
    Nullable<int> i = null;

    if (i.HasValue)
        Console.WriteLine(i.Value); // or Console.WriteLine(i)
    else
        Console.WriteLine("Null");
}
```

Output:

```
Null
```

# Nullable Type

- The **HasValue** returns **true** if the object has been assigned a value; if it has not been assigned any value or has been assigned a null value, it will return **false**.
- Accessing the value using **NullableType.value** will throw a runtime exception if nullable type is null or not assigned any value.
- For example, i.Value will throw an exception if i is null:

```
static void Main(string[] args)
{
    Nullable<int> i = null;

    Console.WriteLine(i.Value);

}
```

InvalidOperationException – run time

# Nullable Type

Example: GetValueOrDefault()

```
static void Main(string[] args)
{
    Nullable<int> i = null;

    Console.WriteLine(i.GetValueOrDefault());
}
```


# Shorthand Syntax for Nullable Types

- You can use the '?' operator to shorthand the syntax e.g. int?, long? instead of using Nullable<T>.

Example: Shorthand syntax for Nullable types

```
int? i = null;
double? D = null;
```


# ?? Operator in Nullable Types

- Use the '??' operator to assign a nullable type to a non-nullable type.

Example: ?? operator with Nullable Type

```
int? i = null;

int j = i ?? 0;

Console.WriteLine(j);
```

- In the above example, i is a nullable int and if you assign it to the non-nullable **int j** then it will throw a runtime exception if i is null. So to mitigate the risk of an exception, we have used the '**??**' operator to specify that if i is null then assign 0 to j.


# Characteristics of Nullable Types

- Nullable types can only be used with value types.
- The Value property will throw an InvalidOperationException if value is null; otherwise it will return the value.
- The HasValue property returns true if the variable contains a value, or false if it is null.
- You can only use == and != operators with a nullable type. For other comparison use the Nullable static class.
- Nested nullable types are not allowed. Nullable<Nullable<int>> i; will give a compile time error.


# WHAT DOES "VAR" MEAN IN C#?

- Variables that are declared at method scope can have an implicit type "var".
- An implicitly typed local variable is strongly typed just as if you had declared the type yourself, but the compiler determines the type.


# "VAR" IN C#

- It declares a type based on what is assigned to it in the initialization.

A simple example is that the code:

```
var i = 53;
```

- Will examine the type of 53, and essentially rewrite this as:

```
int i = 53;
```


# "VAR" IN C#

- It must be used when storing a reference to an object of an anonymous type, because the type name cannot be known in advance.
- When a variable is initialized with an anonymous type you must declare the variable as var if you need to access the properties of the object at a later point.

N.B. We cannot use var as the type of anything but locals. So we can't use the keyword var to declare field/ property/ parameter/ return types.


# WHAT DOES "VAR" MEAN IN C#?

- var can only be used when a local variable is declared and initialized in the same statement;
- The variable cannot be initialized to null, or to a method group or an anonymous function.
- You may find that **var** can also be useful with query expressions in which the exact constructed type of the query variable is difficult to determine.
- But remember, However, the use of **var** does have at least the potential to make your code more difficult to understand for other developers.
- For that reason, the C# documentation generally uses **var** only when it is required.


# Dynamic Type

- Programming languages can normally be considered to be either statically typed or dynamically typed.
- A static (not to be confused with the static keyword, used for classes) typed language validates the syntax or checks for any errors during the compilation of the code.
- On the other hand, dynamically typed languages validate the syntax or checks for errors only at run time.
- For example, C# and Java are a static type and JavaScript is a dynamically typed language.


# Dynamic Type

- C# was previously considered to be a statically typed language, since all the code written was validated at the compile time itself.
- C# 4.0 (.NET 4.5) introduced a new type that avoids compile time type checking.
- We have learned about the implicitly typed variable- var where the compiler assigns a specific type based on the value of the expression.
- A dynamic type escapes type checking at compile time; instead, it resolves type at run time.


# Dynamic Type

- A dynamic type can be defined using the ***dynamic*** keyword.

```
dynamic dynamicVariable = 1;
```

- The compiler compiles dynamic types into object types in most cases. The above statement would be compiled as:

```
object dynamicVariable = 1;
```


# Dynamic Type

- The actual type of dynamic would resolve at runtime.

- A dynamic type changes its type at runtime based on the value of the expression to the right of the "=" operator.


## Methods and properties of dynamic type:

- C# was until now a statically bound language. This means that if the compiler couldn't find the method for an object to bind to then it will throw compilation error.
- We declare at compile-time the type to be dynamic, but at run-time we get a strongly typed object.
- Dynamic objects expose members such as properties and methods at run time, instead of compile time.

| var | dynamic |
| --- | --- |
| Introduced in **C# 3.0** | Introduced in **C# 4.0** |
| **Statically typed** – This means the type of variable declared is decided by the compiler at compile time. | **Dynamically typed** - This means the type of variable declared is decided by the compiler at runtime time. |
| **Need** to initialize at the time of declaration.<br>e.g., var str=”I am a string”;<br>Looking at the value assigned to the variable str, the compiler will treat the variable str as string. | **No need** to initialize at the time of declaration.<br>e.g., dynamic str;<br>str=”I am a string”; //Works fine and compiles<br>str=2; //Works fine and compiles |


| var | dynamic |
|---|---|
| **Errors are caught at compile time.**<br>Since the compiler knows about the type and the methods and properties of the type at the compile time itself | **Errors are caught at runtime**<br>Since the compiler comes to about the type and the methods and properties of the type at the run time. |
| **Visual Studio shows intelligence** since the type of variable assigned is known to compiler. | **Intelligence is not available** since the type and its related methods and properties can be known at run time only |
| e.g., var obj1;<br>will **throw a compile error** since the variable is not initialized. The compiler needs that this variable should be initialized so that it can infer a type from the value. | e.g., dynamic obj1;<br>**will compile;** |

| var | dynamic |
| --- | --- |
| e.g. var obj1=1;<br>will compile<br>var obj1=" I am a string";<br>**will throw error** since the compiler has already decided that the type of obj1 is System.Int32 when the value 1 was assigned to it. Now assigning a string value to it violates the type safety. | e.g. dynamic obj1=1;<br>will compile and run<br>dynamic obj1=" I am a string";<br>**will compile and run** since the compiler creates the type for obj1 as System.Int32 and then recreates the type as string when the value "I am a string" was assigned to it.<br>This code will work fine. |
| **Restrictions on the usage**<br><br>var variables **cannot be used for property or return values from a function**. They can only be used as local variable in a function. | dynamic variables can be used **to create properties and return values** from a function. |


# Why Dynamic Type?

- The dynamic type enables the operations in which it occurs to bypass compile-time type checking. Instead, these operations are resolved at run time.
- The dynamic type simplifies access to COM APIs such as the **Office Automation APIs**, and also to dynamic APIs such as **IronPython** libraries, and to the **HTML Document Object Model (DOM**).


# Dynamic Type

- To conclude Dynamic Type is a nice feature when it comes to interoperability and .NET usage with other Dynamic languages.
- This capability was added to the CLR in order to support dynamic languages like Ruby and Python.

# *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