# Generics

**Windows Programming Course**

# Agenda

1. Generics Overview
2. Using Generics

# 1.
# Generics Overview

# Motivation

- Generics allow you to define type-safe data structures, without committing to actual data types.

- This results in a significant performance boost and higher quality code, because you get to reuse data processing algorithms without duplicating type-specific code.

- Using generic types to maximize Code Reused, Type Safety, Performance and Code Bloat.

# Generic usage example

**Generic List<T>**

```
List<string> list = new List<string>();
list.Add("Bob");
list.Add(22); // Rejected by the compiler
string str = list.Get(0);
```

**Non-generic ArrayList**

```
ArrayList list = new ArrayList();
list.Add("Bob");
list.Add(22); // Objects of any type can be added to the list
string str = (string) list.Get(0); // Type cast necessary
```

# Syntax

```
public class LibraryManager<T> where T : IDocument
{
  public void AddBook(T book) { … }
  public T FindBook(int id) { … }
}
```

# 2.
# Using Generics

# Naming Guidelines

Prefix generic type names with the letter T

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
public delegate TOutput Converter<TInput, TOutput>(TInput from);
public class SortedList<TKey, TValue> { }
```

If the generic type can be replaced by any class because there's no special requirement, and only one generic type is used => Use character T directly

```
public class List<T> { }
public class LinkedList<T> { }
```

# Default Values

Sometimes you need to get a default value for a generic type parameter

- e.g.: `T FindBook(int id)`
- If there is no element with the specified id we want to return the default

Default values

- Null: References types like String, etc.
- 0: int, etc.
- False: bool

Syntax:

- `default(T)`

E.g.: `T FindBook(int id) { … return default(T); }`

# Constrains

When you define a generic class, you can apply restrictions to the kinds of types that client code can use for type arguments when it instantiates your class:

```
public class LibraryManager<T> where T : IDocument
```

# Constrains (cont.)

| CONSTRAINT | DESCRIPTION |
| --- | --- |
| `where T: struct` | With a struct constraint, type T must be a value type. |
| `where T: class` | The class constraint indicates that type T must be a reference type. |
| `where T: IFoo` | Specifies that type T is required to implement interface IFoo. |
| `where T: Foo` | Specifies that type T is required to derive from base class Foo. |
| `where T: new()` | A constructor constraint; specifies that type T must have a default constructor. |
| `where T1: T2` | With constraints it is also possible to specify that type T1 derives from a generic type T2. |

# Constructor constrain

Special constraint using the new keyword:

```
public class Stack<T> where T : new() {
    public T PopEmpty() {
        return new T();
    }
}
```

Parameter-less **constructor constraint**

- Type T must provide a public parameter-less constructor
- No support for other constructors or other method syntaxes.
- The new() constraint must be the last constraint.

# Secondary Constraints

A generic type parameter, like a regular type, can have zero or more interface constraints:

```
public class GraphNode<T> where T : ICloneable, Icomparable
{


}
```

# Generic Methods

C# also allow you to parameterize a method with generic types:

```
public static void Swap<T>(ref T a, ref T b)
{
   T temp = a;
   a = b;
   b = temp;
}
```

# Hands-on Exercise

**Using Generic to avoid code duplication:**

- Get the source code on GitHub
- Implement `BaseRepository` which derive `IRepository`
- Replace all usages of 2 repositories: `StudentRepository` and `ClassRepository` by `BaseRepository`
- Complemetely remove `StudentRepository` and `ClassRepository` by

# Thanks!

## Any questions?

You can find me at:

tranminhphuoc@gmail.com