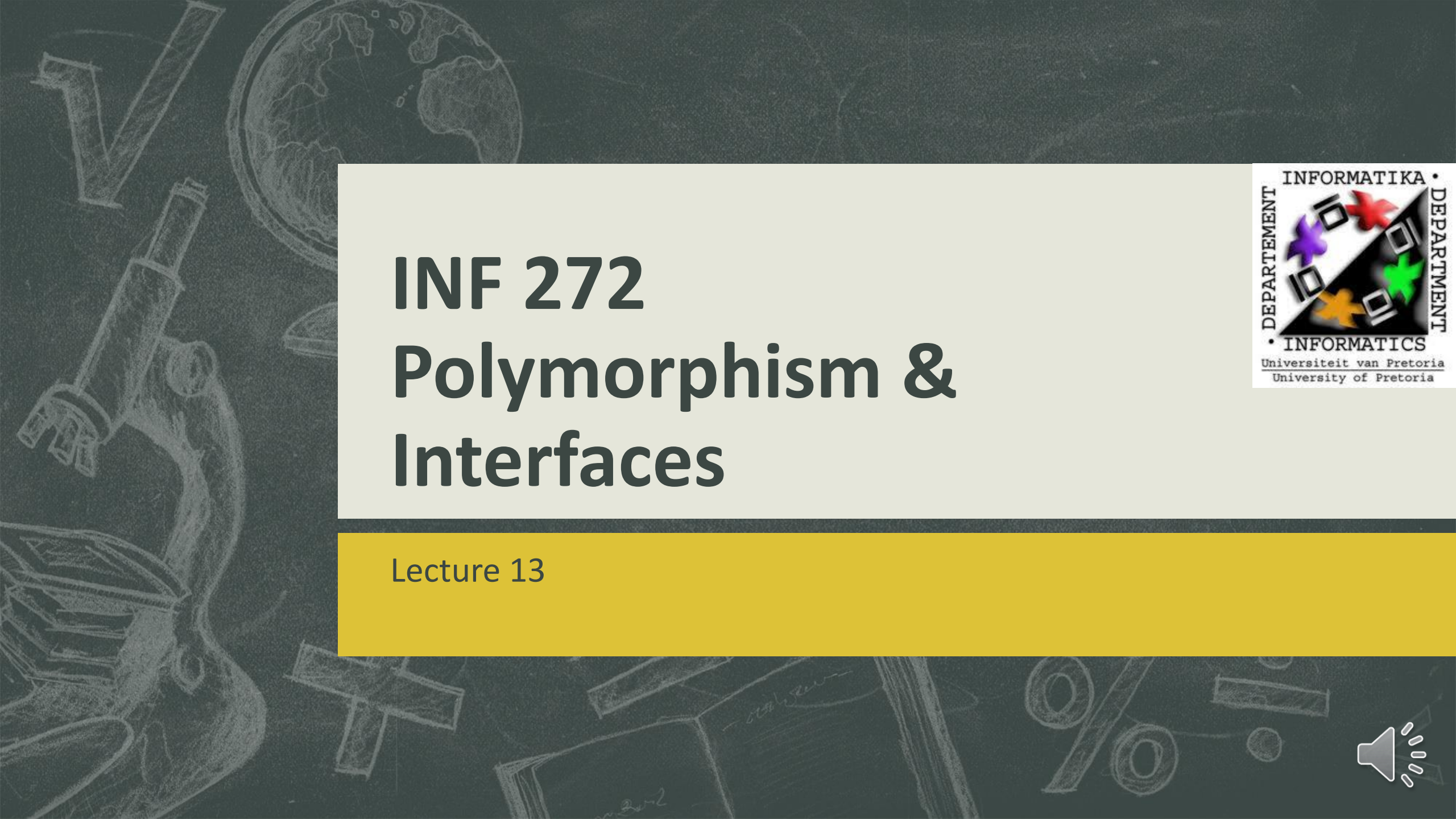
INF 272
Polymorphism & Interfaces

Lecture 13



Outcome:

- Understand and use ***this*** and ***base*** keywords.
- Understand and use ***virtual*** and ***abstract*** in base classes.
- Understand and use ***override*** and ***override sealed*** methods in derived classes.
- Understand and use ***Polymorphism***.
- Understand the benefits of polymorphism.
- Understand **casting** and how to apply it.
- Understand and use ***Interfaces***.

this keyword

- The this keyword refers to the current instance of the class. It can be used to access members from within constructors, instance methods, and instance accessors.
- Static constructors and member methods do not have a this pointer because they are not instantiated.
- When you are writing code in method or property of a class, using this will allow you to make use of the intellisense.

Example: without *this*

Class:

```
public class Demo
{
        int age;
        string name;

        public Demo(int age, string name)
        {
           age = age;
           name = name;
        }

        public string Show()
        {
            return "Your age is : " + age.ToString() + ", and your name is : " + name;
        }
}
```

Instantiation:

```
Demo object = new Demo(21, "Gabriel");

string str = object.Show();
```

Value of str:

Your age is 0, and your name is _

The above example displays that by using the same variable name the demo is unable to assign the values correctly because of bad variable naming conventions. In other words, using the same name for the variable. E.g. age and age (age = age).

Example: *this*

Class:

```
public class Demo
{
        int age;
        string name;

        public Demo(int age, string name)
        {
           //Add this keyword
           this.age = age;
           this.name = name;
        }

        public string Show()
        {
            return "Your age is : " + age.ToString() + ", and your name is : " + name;
        }
}
```

Instantiation:

```
Demo object = new Demo(21, "Gabriel");

string str = object.Show();
```

Value of str:

Your age is 21, and your name is Gabriel

The above example demonstrates that while we are still using bad variable naming conventions, the Demo can display the assigned variables. Because we used the 'this' keyword.

base keyword

- The base keyword is used to access members of the base class from within a derived class.
- A base class access is permitted only in a constructor, an instance method, or an instance property accessor.

```
public class Fruit
{
        protected string _name;
        protected int _calories;

        public Fruit(string name, int calories)
        {
            _name = name;
            _calories = name;
        }

        public string GetInfo()
        {
            return _name + " " + _calories.ToString();
        }
}
```

```
public class Apple : Fruit
{
        private string _type;

        public Apple(string name, int calories, string type) : base(name, calories)
        {
            _type = type;
        }

        public bool shouldEat()
        {
            return (this._type=="Golden Delicious") && (base._calories<1000);
        }
}
```

Key concepts with inherited methods

- virtual – The virtual keyword is assigned to a method, property, indexer, or event declared in the base (parent) class. The virtual keyword allows the method, property, indexer, or event to be overridden in the derived (child) class.
- override - The override keyword is used in the derived class to extend or modify a virtual/abstract method, property, indexer, or event of the base (parent) class.
- override sealed – Same as override, except that further derived classes cannot override the method, property, indexer, or event. It stops at the derived (child) class implementing the 'override sealed' keyword(s).

Examples: virtual, override, & override sealed

Base class

```
public class Person
{
        protected string _name;
        protected int _age;

        public Person(string name, int age)
        {
            _name = name;
            _age = age;
        }

        public virtual string GetInfo()
        {
            return _name + " (" + _age.ToString() + "yo)";
        }
}
```

Derived class

```
public class Student : Person
{
        protected string _studentNo;
        protected string _degree;

        public Student(string name, int age, string sNo, string degree) : base(name, age)
        {
            _studentNo = sNo;
            _degree = degree;
        }

        public override string GetInfo()
        {
            return base.GetInfo() + " is a student (u"+this._studentNo + ") studying " +        this._degree;
        }
}
```

Derived class

```
public class Pensioner : Person
{
        protected int _grandKids;

        public Person(string name, int age, int gKids) : base(name, age)
        {
            _ grandKids = gKids;
        }

        public override sealed string GetInfo()
        {
            return base.GetInfo() + "is a pensioner (grandchildren: "
              +this._grandKids + ")";
        }
}
```

abstract classes and methods

- The abstract modifier indicates that the thing being modified has a missing or incomplete implementation.
- The abstract modifier can be used with classes, methods, properties, indexers, and events.
- Use the abstract modifier in a class declaration to indicate that a class is intended only to be a base (parent) class of other classes. In other words, think of a abstract class as a base (parent) class that is a ‘contract’ for derived (child) classes to implement and further extend (build upon). The ‘contract’ specifies what you can use within your derived (child) class of the base (parent) class.
- ***NB: You cannot create an object (instance) of an abstract class. Further, and abstract method in an abstract class does not contain a body. I.e. a definition. The body of the method is created in the derived (child) class.***

Example of **abstract** class and method

Base class

```
public abstract class Person
{
        protected string _name;
        protected int _age;

        public Person(string name, int age)
        {
            _name = name;
            _age = age;
        }

        public abstract string DescribeWalk();

}
```

Derived class

```
public class Student : Person
{
        protected string _studentNo;
        protected string _degree;

        public Student(string name, int age, string sNo, string degree) : base(name, age)
        {
            _studentNo = sNo;
            _degree = degree;
        }

        public override string  DescribeWalk()
        {
            return this._degree == "Drama" ? "Theatrical prance" : "Amble";
        }
}
```

Derived class

```
public class Pensioner : Person
{
        protected int _grandKids;

        public Person(string name, int age, int gKids) : base(name, age)
        {
            _ grandKids = gKids;
        }

        public override string  DescribeWalk()
        {
            return "Lumber";
        }
}
```

What is polymorphism?

- The state of being polymorphus.
- Poly = multiple or more than one.
- Morph = form
- Polymorphus means having or assuming many or various forms.
- In OOP, polymorphism means providing multiple implementations for the same behaviour.
- ***NB: Could you spot the polymorphic behaviour in the previous slide? Hint: check the methods.***
- For example, **Language**
  - Language (the ability to communicate) can be viewed as a ability (behaviour) of people (or Person entities)
  - However, different people uses different languages (***polymorphic***) to communicate.

Why is polymorphism so great?

- Simplicity
  - Having many implementations, but one interface means you can handle standard behaviours of different objects in a standard way.
  - It allows you to invoke methods of derived class through base class reference during runtime.
- Extensibility – overriding behaviours with unique implementations
- Reusability – define an interface/signature once and interact with it in the same way despite multiple implementations.

Example: ***polymorphism***

Surprise!

- We have already seen polymorphism.
- When you override virtual or abstract methods, you are effecting polymorphism.

```
public abstract class Pet
{
        public abstract string Talk();
}
```

```
public class Cat : Pet
{
        public override string Talk()
        {
            return "Meow";
        }
}
```

```
public class Dog : Pet
{
        public override string Talk()
        {
            return "Woof";
        }
}
```

one interface

multiple implementations

Casting

- Casting makes a variable temporarily behave as if it is of a different type.

Casting basic types

```
int Value1 = 0;
double Value2 = 100;

Value1 = (int)Value2;
```

Adding different subclasses to list of base class

```
Cat  kitty = new Cat();
Dog  puppy = new Dog();
List<Pet> pets = new List<Pet>();
pets.Add(kitty);
pets.Add(puppy);
```

Casting objects

```
foreach(Pet pet in pets)
{
    if(pet.GetType() == typeof(Cat))
    {
        Cat myCat = (Cat)pet;
    }
    else
    {
        Dog myDog = (Dog)pet;
    }
}
```

Interfaces

- Interfaces are another way to achieve '***abstraction***'.
- An interface is much the same like an '***abstract class***' like you have learned thus far. However, an interface can only have abstract methods and properties.
- In other words, the methods and properties do not have any definitions to them. They have empty '***bodies***'.

```
// interface
interface Animal
{
  void animalSound(); // interface method (does not have a body)
  void run(); // interface method (does not have a body)
}
```

Source: [1]

Interfaces (continued…)

- Interfaces cannot be used to create objects (instances).
- Further, you can '***inherit***' from only '***one base (parent) class***', however, you can '***implement***' '***multiple interfaces***'.
- You MUST '***implement***' all the '***members (properties and methods)***' of a interface, or you will get errors.
- So, you can say '***overriding***' the interface '***members***' is a must when '***implementing***' interfaces.

```
// Interface
interface IAnimal
{
  void animalSound(); // interface method (does not have a body)
}

// Pig "implements" the IAnimal interface
class Pig : IAnimal
{
  public void animalSound()
  {
    // The body of animalSound() is provided here
    Console.WriteLine("The pig says: wee wee");
  }
}

class Program
{
  static void Main(string[] args)
  {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
  }
}
```

Source: [1]

Interfaces (continued…)

- An example of '***implementing***' multiple interfaces.

```
interface IFirstInterface
{
  void myMethod(); // interface method
}

interface ISecondInterface
{
  void myOtherMethod(); // interface method
}

// Implement multiple interfaces
class DemoClass : IFirstInterface, ISecondInterface
{
  public void myMethod()
  {
    Console.WriteLine("Some text..");
  }
  public void myOtherMethod()
  {
    Console.WriteLine("Some other text...");
  }
}

class Program
{
  static void Main(string[] args)
  {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
}
```

Source: [1]

Video

- This week's video is available at: https://youtu.be/w3A2AuUmCLE
- This week's video is available at: https://youtu.be/IcsBC_wDMlI
- This week's video is available at: https://www.youtube.com/watch?v=aQ8YkJrAbzE

Look at these tutorials and documentation for additional details and help:

- [1] - https://www.w3schools.com/cs/cs_interface.php
- http://www.dotnettricks.com/learn/csharp/understanding-virtual-override-and-new-keyword-in-csharp
- https://www.dotnetperls.com/abstract
- http://csharp-station.com/Tutorial/CSharp/Lesson09
- https://www.codeproject.com/Articles/447634/A-Beginners-Tutorial-Type-Casting-and-Type-Convers
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface