Basic Interfaces and Polymorphisms

greenwich.edu.vn

UNIVERSITY of GREENWICH

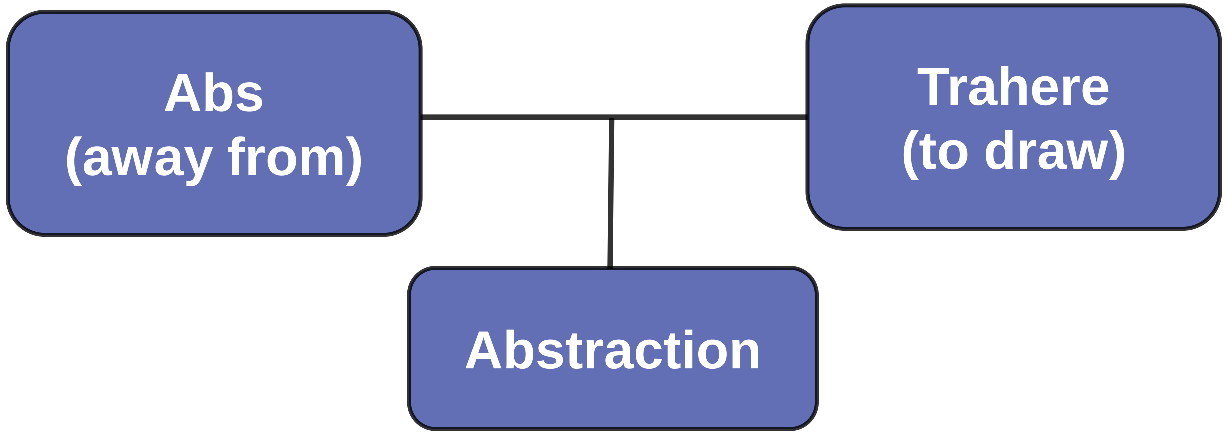
Alliance with FPT Education

Table of Contents

ACHIEVING ABSTRACTION

What is Abstraction?

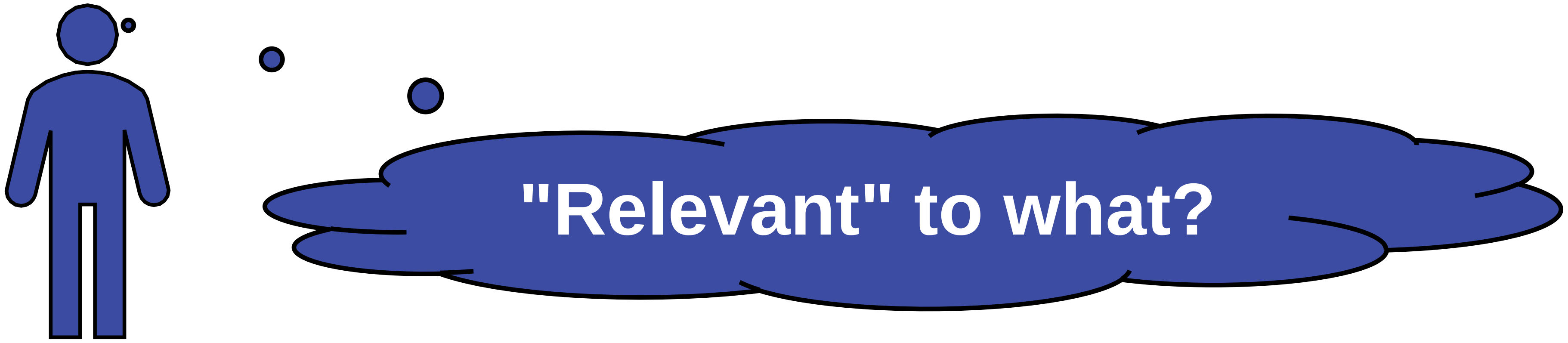
- From the Latin



- Preserving information, relevant in a given context, and forgetting information that is irrelevant in that context

Abstraction in OOP

- Abstraction means ignoring irrelevant features, properties, or functions and emphasizing the ones …



- ... relevant to the context of the project we develop
- Abstraction helps managing complexity
- Abstraction lets you focus on what the object does instead of how it does it

How Do We Achieve Abstraction?

- There are two ways to achieve abstraction
  - – Interfaces
  - – Abstract class

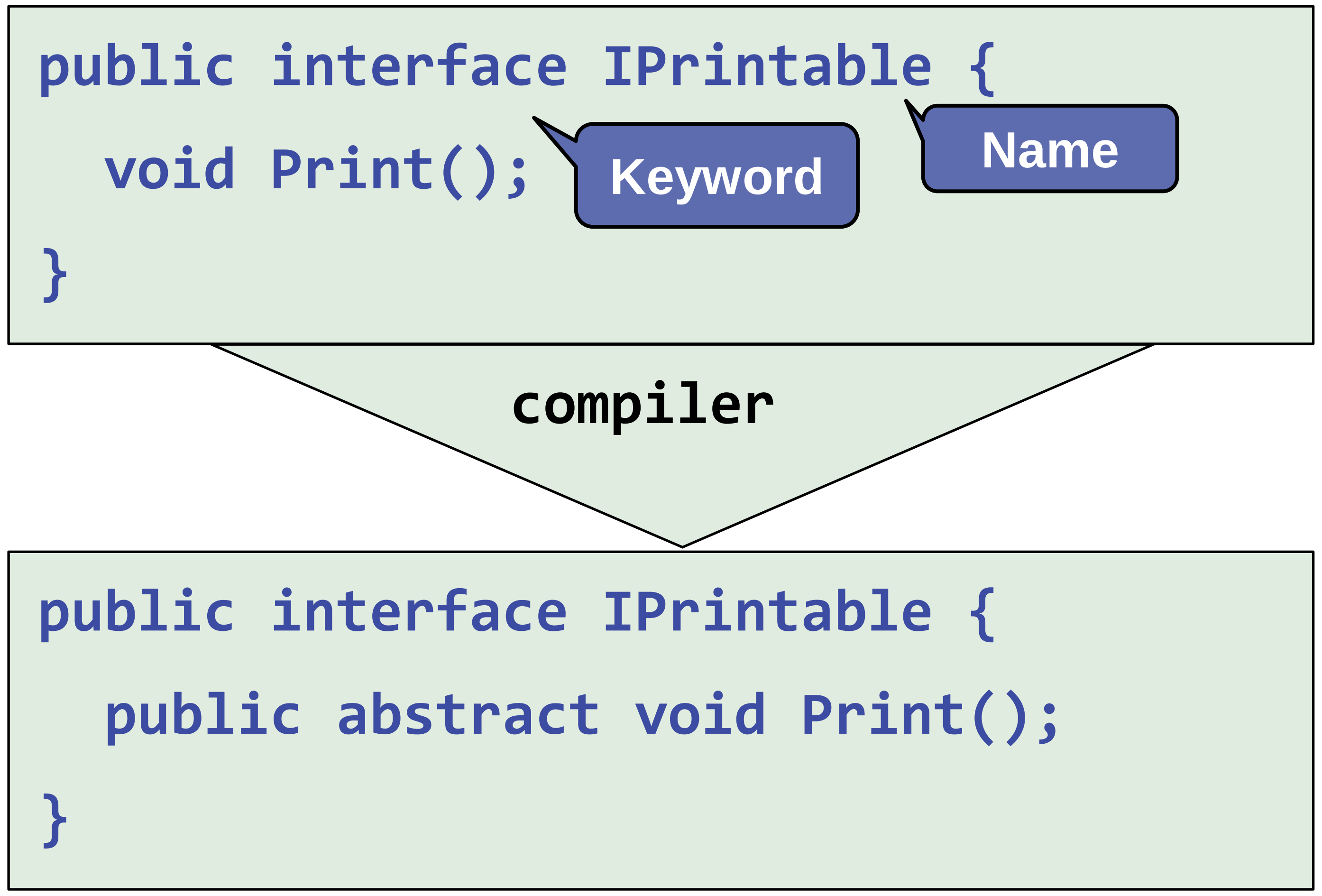
```
public interface IAnimal {}
public abstract class Mammal {}
public class Person : Mammal, IAnimal {}
```

WORKING WITH INTERFACES

Interfaces

Interface

- Internal addition by compiler



```
public interface IPrintable {
  void Print();
}
```

```
public interface IPrintable {
  public abstract void Print();
}
```

Interface Example

- The implementation of Print() is provided in class Document

```
public interface IPrintable {
  void Print();
}
```

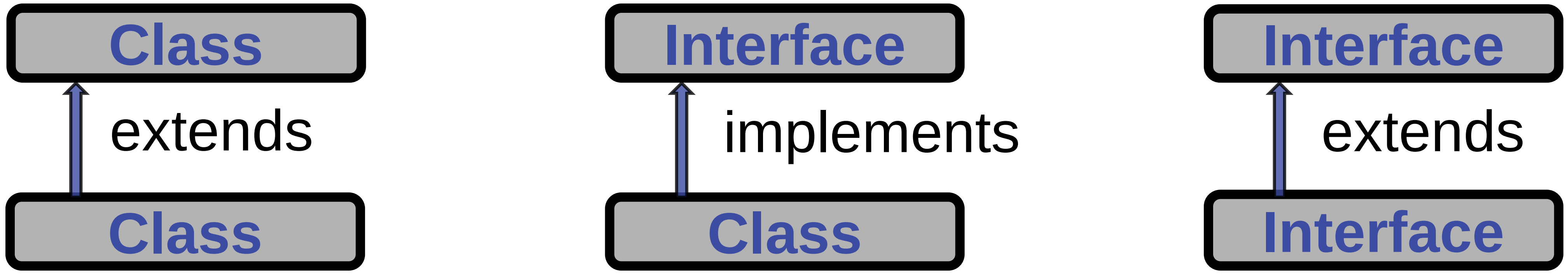
```
class Document : IPrintable {
  public void Print()
  { Console.WriteLine("Hello"); }
```

Interface (2)

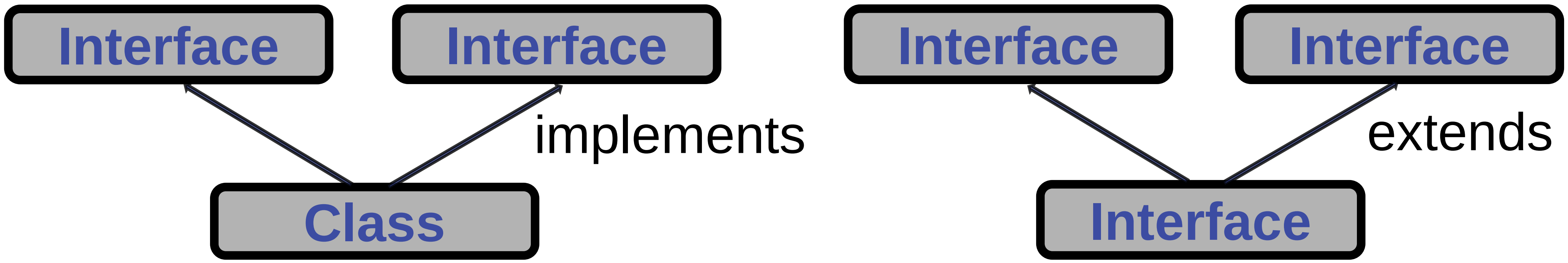
- Contains only the signatures of methods, properties, events or indexers
- Can inherit one or more base interfaces
- When a base type list contains a base class and interfaces, the base class must come first in the list
- A class that implements an interface can explicitly implement members of that interface
  - An explicitly implemented member cannot be accessed through a class instance, but only through an instance of the interface

Multiple Inheritance

- Relationship between classes and interfaces

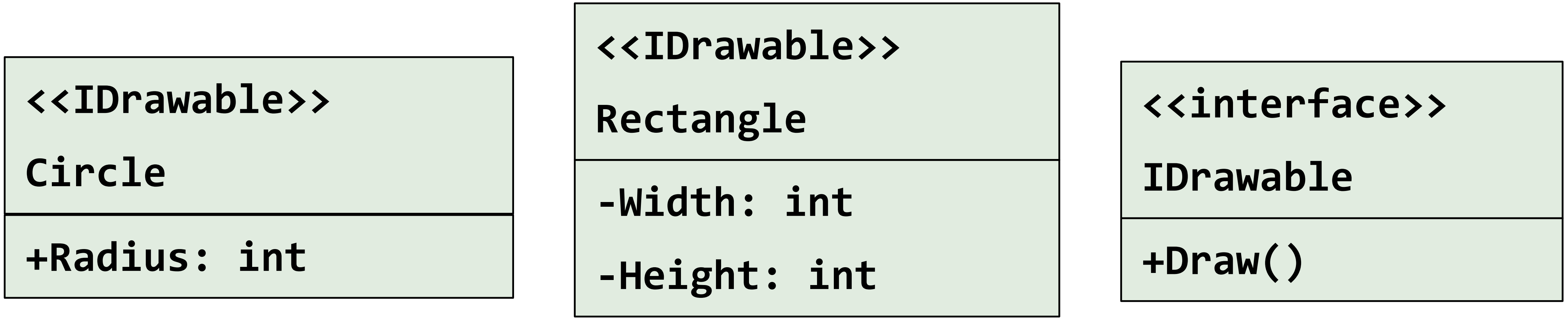


- Multiple inheritance



Problem: Shapes

- Build a project that contains an interface for drawable objects
- Implements two type of shapes: Circle and Rectangle
- Both classes have to print on the console their shape with "*"



Solution: Shapes

```
public interface IDrawable {
  void Draw();
}
```

```
public class Rectangle : IDrawable {
  // TODO: Add fields and a constructor
  public void Draw() { // TODO: implement } }
```

```
public class Circle : IDrawable {
  // TODO: Add fields and a constructor
  public void Draw() { // TODO: implement } }
```

Solution: Shapes – Rectangle Draw

```
public void Draw() {
  DrawLine(this.width, '*', '*');
  for (int i = 1; i < this.height - 1; ++i)
    DrawLine(this.width, '*', ' ');
  DrawLine(this.width, '*', '*'); }
private void DrawLine(int width, char end, char mid) {
  Console.Write(end);
  for (int i = 1; i < width - 1; ++i)
    Console.Write(mid);
  Console.WriteLine(end); }
```

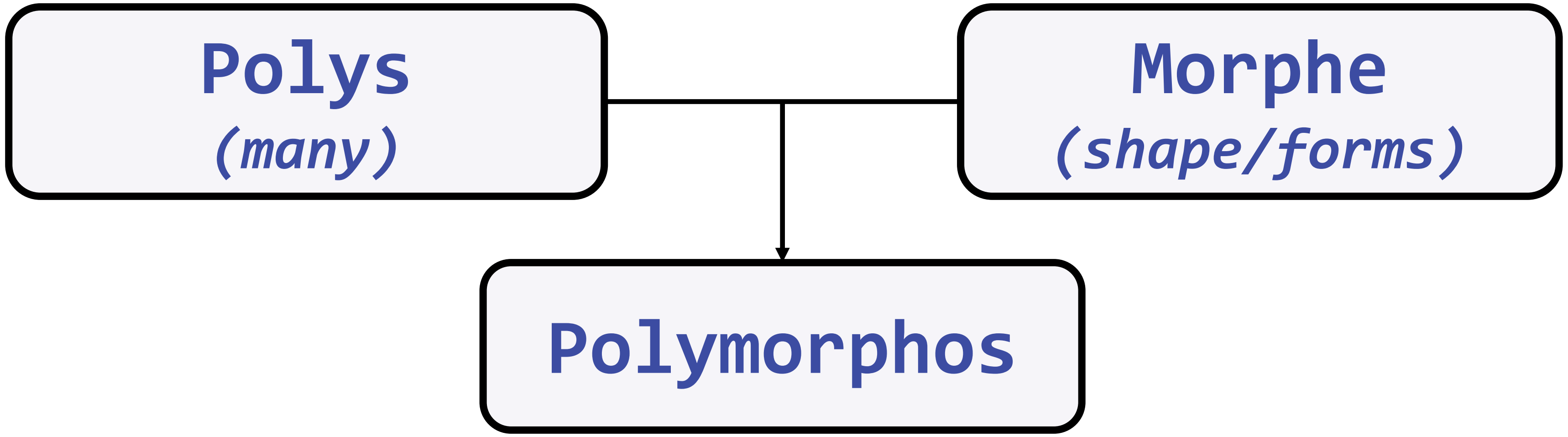
Solution: Shapes – Circle Draw

```
double rIn = this.radius - 0.4;
double rOut = this.radius + 0.4;
for (double y = this.radius; y >= -this.radius; --y) {
  for (double x = -this.Radius; x < rOut; x += 0.5) {
    double value = x * x + y * y;
    if (value >= rIn * rIn && value <= rOut * rOut)
      Console.Write("*");
    else
      Console.Write(" "); }
  Console.WriteLine(); }
```

POLYMORPHISM

What is Polimorphism?

- From the Greek



- This is something similar to a word having several different meanings depending on the context
- Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance

Types of Polymorphism

- Runtime

```
public class Shape {}
public class Circle : Shape {}
public static void Main()
{
  Shape shape = new Circle()
}
```

- Compile time

```
public static void Main()
{
  int Sum(int a, int b, int c)
  double Sum(Double a, Double b)
}
```

Compile Time Polymorphism

- Also known as

```
public static void Main()
{
  static int MyMethod(int a, int b) {}
  static double MyMethod(double a, double b) { … }
}
```

Method overloading

- Argument lists could differ in:
  - Number of parameters
  - Data type of parameters
  - Order of parameters

Problem: MathOperation

```
MathOperation
```

```
+Add(int, int): int
+Add(double, double, double): double
+Add(decimal, decimal, decimal): decimal
```

```
MathOperations mo = new MathOperations();
Console.WriteLine(mo.Add(2, 3));
Console.WriteLine(mo.Add(2.2, 3.3, 5.5));
Console.WriteLine(mo.Add(2.2m, 3.3m, 4.4m));
```

Solution: MathOperation

```
public int Add(int a, int b)
{
  return a + b;
}
public double Add(double a, double b, double c)
{
  return a + b + c;
}
public decimal Add(decimal a, decimal b, decimal c)
{
  return a + b + c;
}
```

Rules for Overloading a Method

- Signature should be different
  - – Number of arguments
  - – Type of arguments
  - – Order of arguments
- Return type is not a part of its signature
- Overloading can take place in the same class or in its sub-classes
- Constructors can be overloaded

Runtime Polymorphism

- Has two distinct aspects:
- At run time, objects of a derived class may be treated as objects of a base class in places, such as method parameters
  and collections or arrays
  - When this occurs, the object's declared type is no longer identical to its run-time type

Runtime Polymorphism(2)

- Base classes may define and implement
  - Derived classes can
  - They provide
- At run-time, the CLR looks up the run-time type of the object
  and invokes that override of the virtual method

Runtime Polymorphism

- Also known as

```
public class Rectangle {
  public virtual double Area() {
    return this.a * this.b;
  }
}
public class Square : Rectangle {
  public override double Area() {
    return this.a * this.a;
  }
}
```

Method overriding