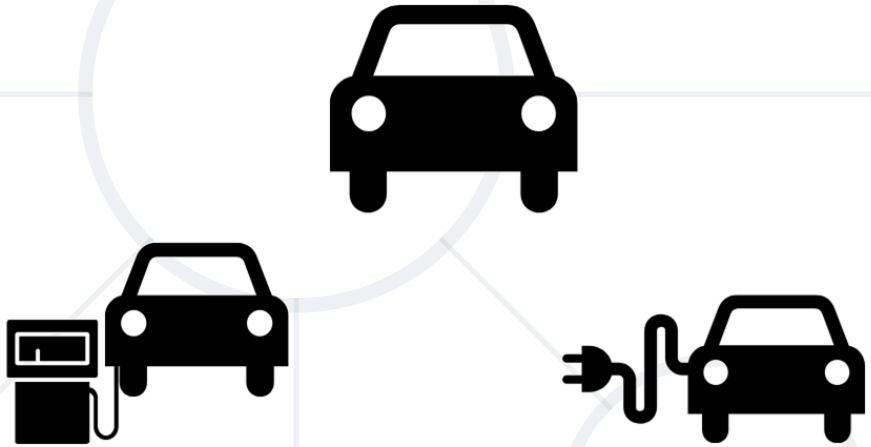
Polymorphism

Polymorphism, Override and Overload Methods



ONE NAME FOR MANY FORMS

SoftUni Team

Technical Trainers

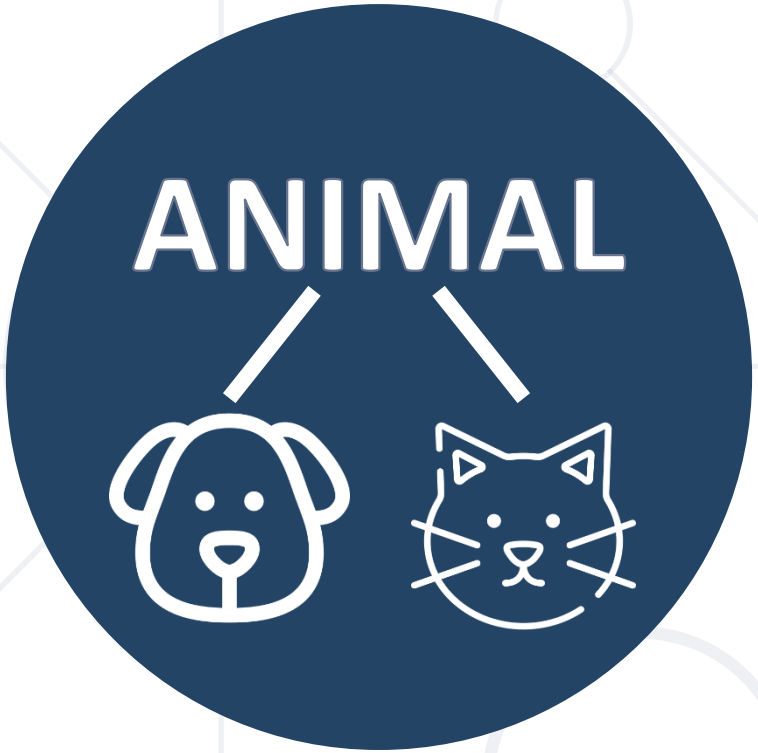
Software University

SoftUni

Software University

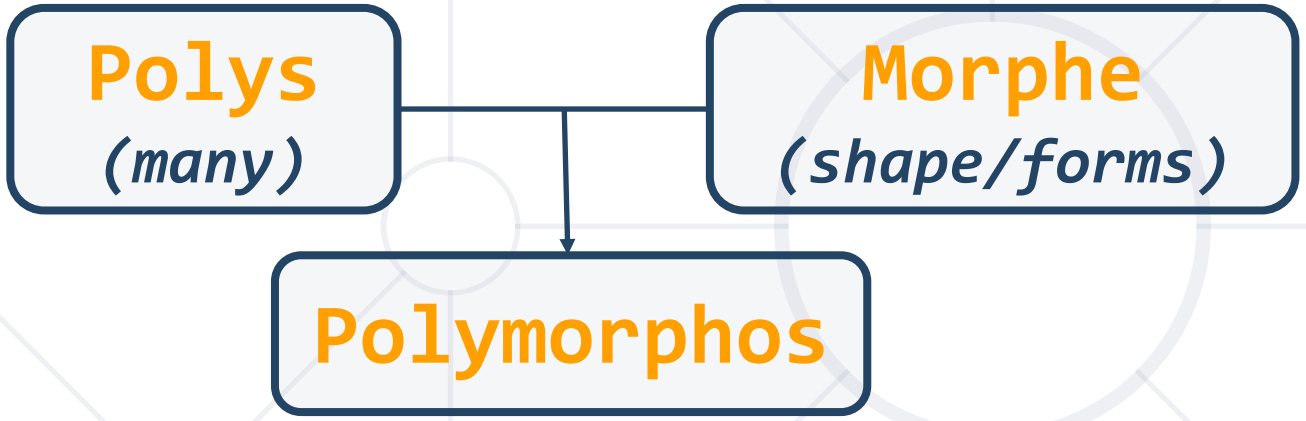
https://softuni.bg

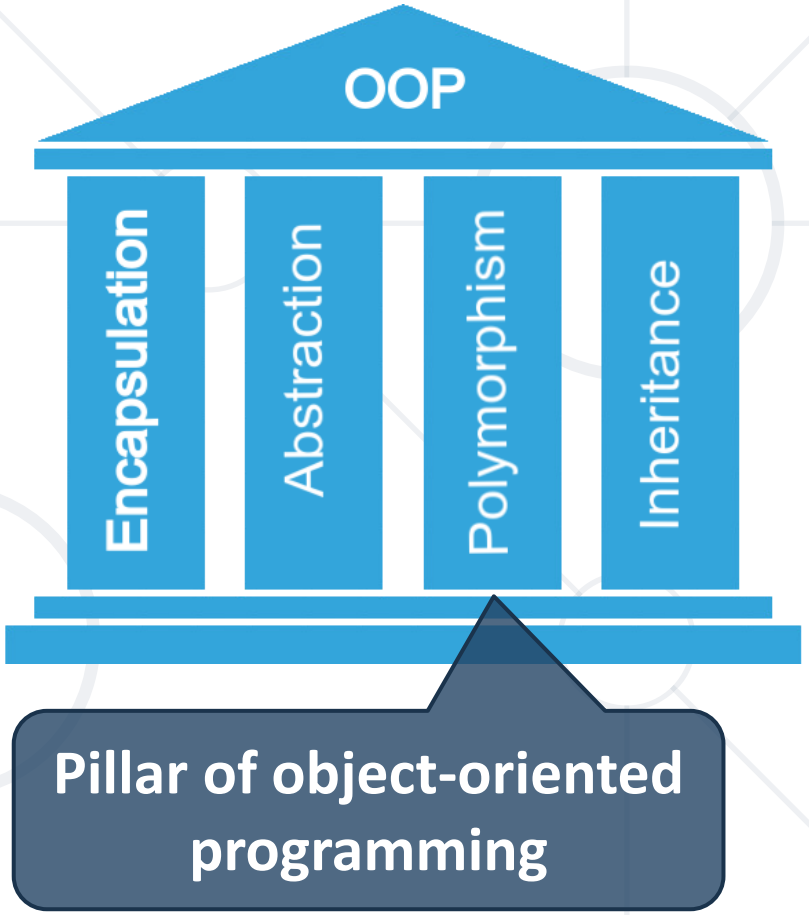
Table of Contents



Ability of an Object to Take On Many Forms

What is Polymorphism?





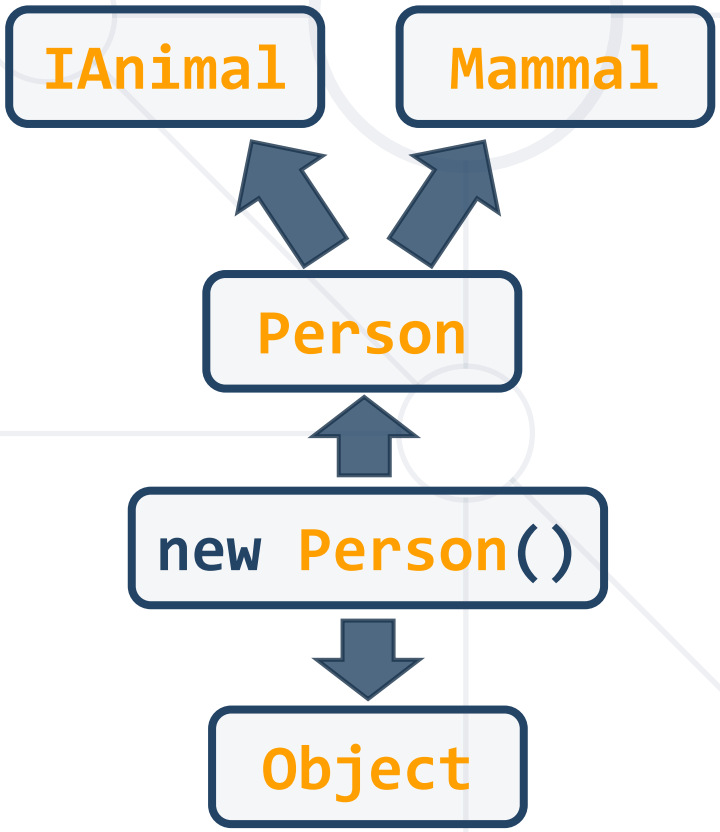
- Polymorphism is a Greek word, meaning "**one name many forms**"

Polymorphism in OOP

- Ability of an **object** to take on **many forms**
- Allows treating objects of a **derived class** as objects of its **base class**

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

`Person IS-AN Object`	`Person IS-A Mammal`
`Person IS-A Person`	`Person IS-AN Animal`



Reference Type and Object Type

- **Variables** are saved in a **reference** type
- You can use only **reference methods**
- If you need an **object method** you need to **cast it or override it**

```
public class Person : Mammal, IAnimal {}
IAnimal  person   = new Person();
Mammal personOne = new Person();
Person personTwo = new Person();
```

Reference Type

Object Type

Type-casting and Compatibility Checking

Keyword – is

- Check if an **object** is an **instance** of a specific **class**

```
public class Person : Mammal, IAnimal {}
IAnimal person = new Person();
Mammal personOne = new Person();
Person personTwo = new Person();
if (person is Person)
{
  ((Person) person).getSalary();
}
```

Check object type of person

Cast to object type and use its methods

IS Type Pattern

- **is type pattern** - tests whether an expression can be **converted** to a specified type **and casts** it to a variable of that type

```
public class Person : Mammal, IAnimal {}
Mammal personOne = new Person();
Person personTwo = new Person();
if (personTwo is Person person)
{
  person.GetSalary();
}
```

Checks if object is of type person and casts it

Uses its methods

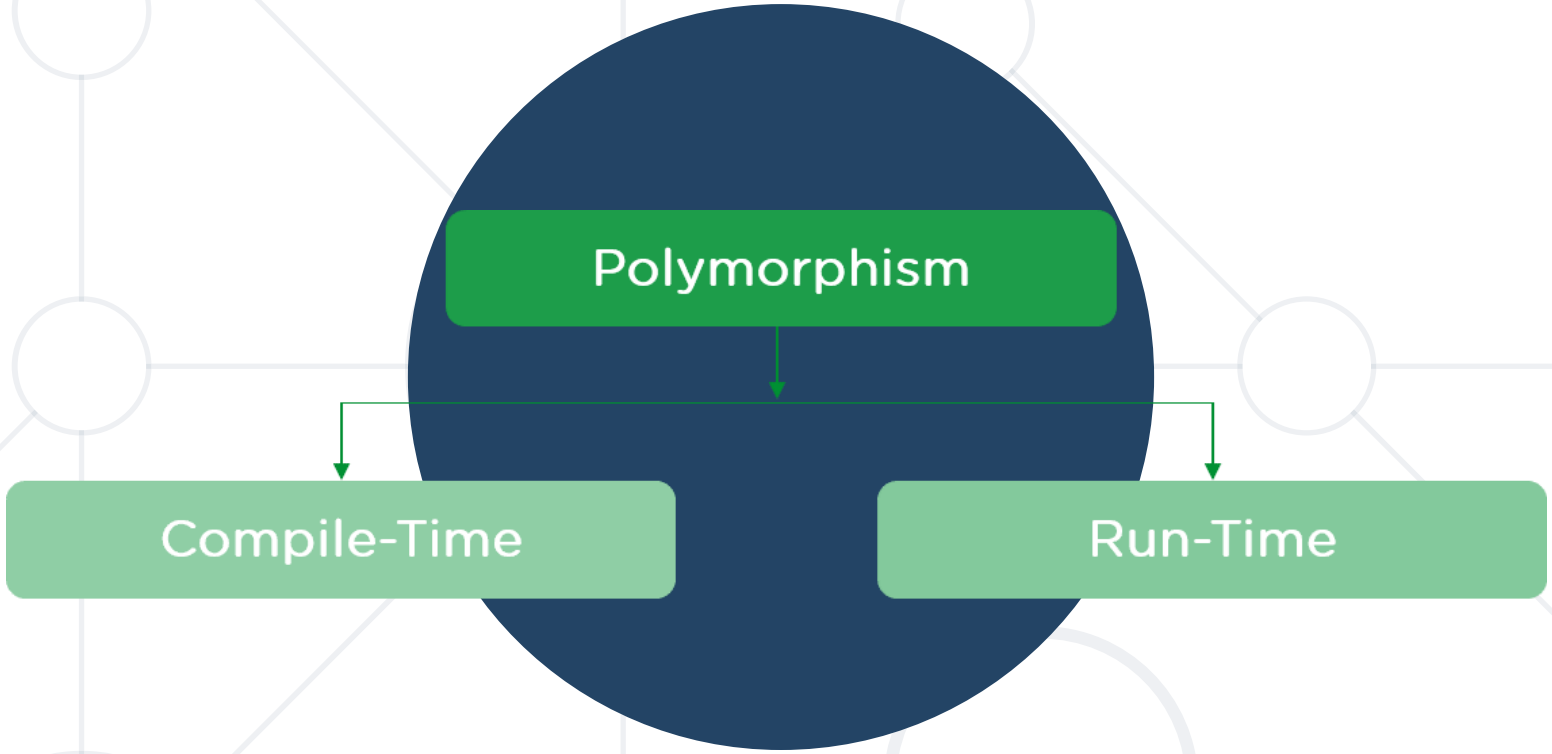
Keyword – As

- **as** operator is used for **conversions** between compatible reference types

```
public class Person : Mammal, Animal {}
Animal person = new Person();
Mammal personOne = new Person();
Person personTwo;
personTwo = personOne as Person;
if (personTwo != null)
{
  // Do something specific for Person
}
```

Convert Mammal to Person

Check if conversion is successful



Compile-Time and Run-Time Polymorphism

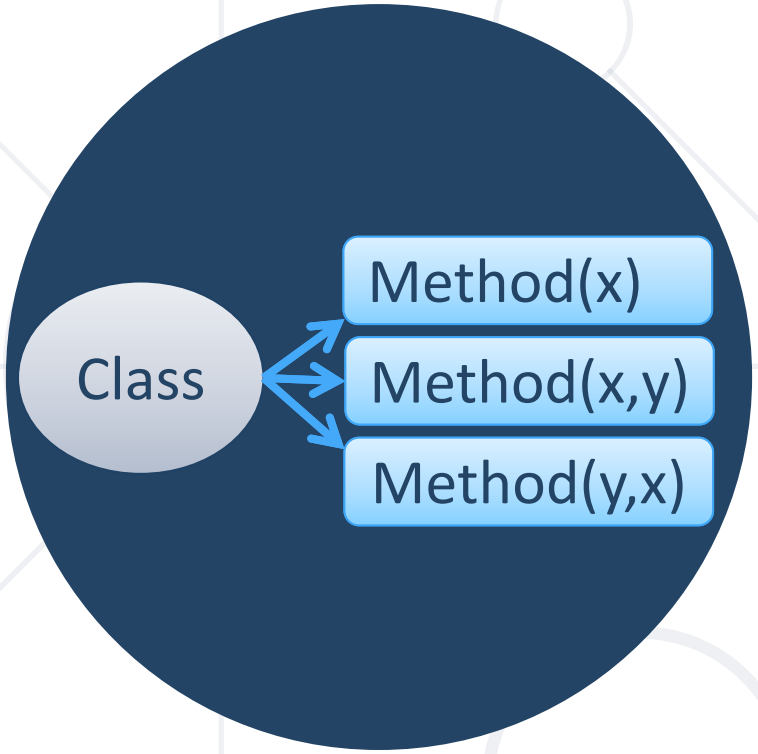
Types of Polymorphism

- 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)
}
```



Overloading

Compile Time Polymorphism

Software University

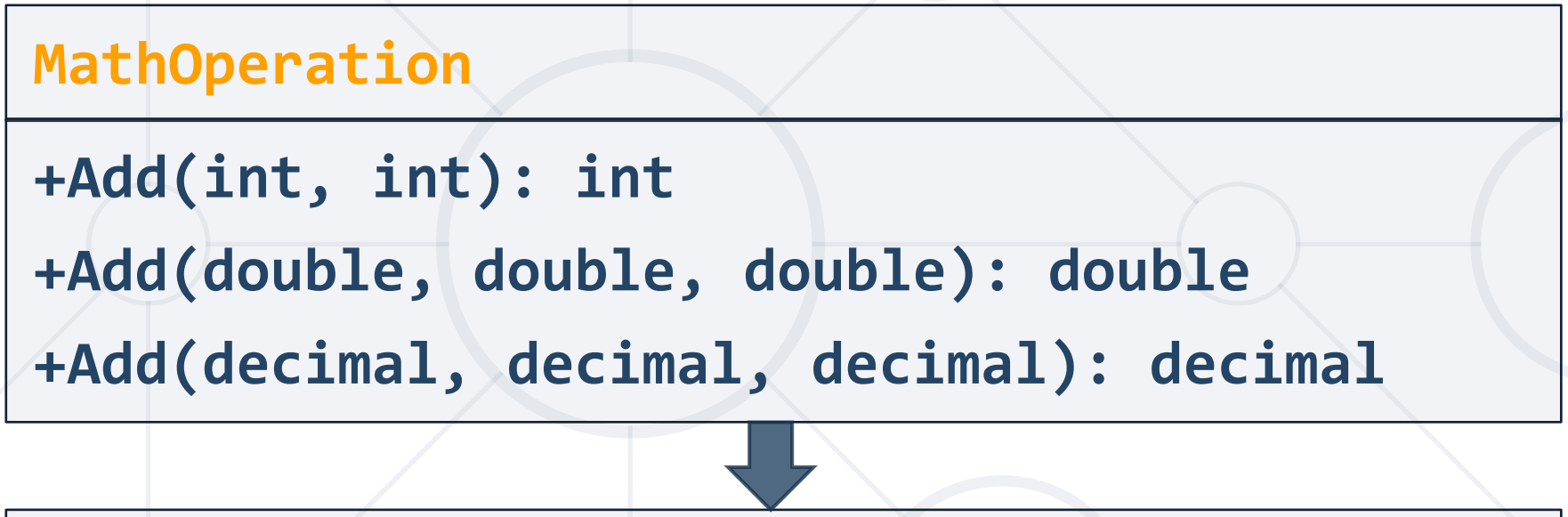
- Also known as **Static Polymorphism** – realized by **overloading**

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

Method overloading – same name but different implementation

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

Problem: MathOperation




```
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;
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3167#0

Rules for Overloading a Method (1)

- Signature **must be different**, either:
  - **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**

Rules for Overloading a Method (2)

- **Different number and type** of arguments

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

```
static void Main(){
  Calculator calc = new Calculator();
  int sum1 = calc.Add(1, 2);
  int sum2 = calc.Add(1, 2, 3);
  int sum3 = calc.Add(1.0, 2.3, 3.1);
}
```

Rules for Overloading a Method (3)

- **Different order** of arguments

```
class Guest {
string Identity(string name, int id)
  { return $"{name} + {id}"; }
void Identity(int id, string name)
  { return $"{name} + {id}"; }
}
```

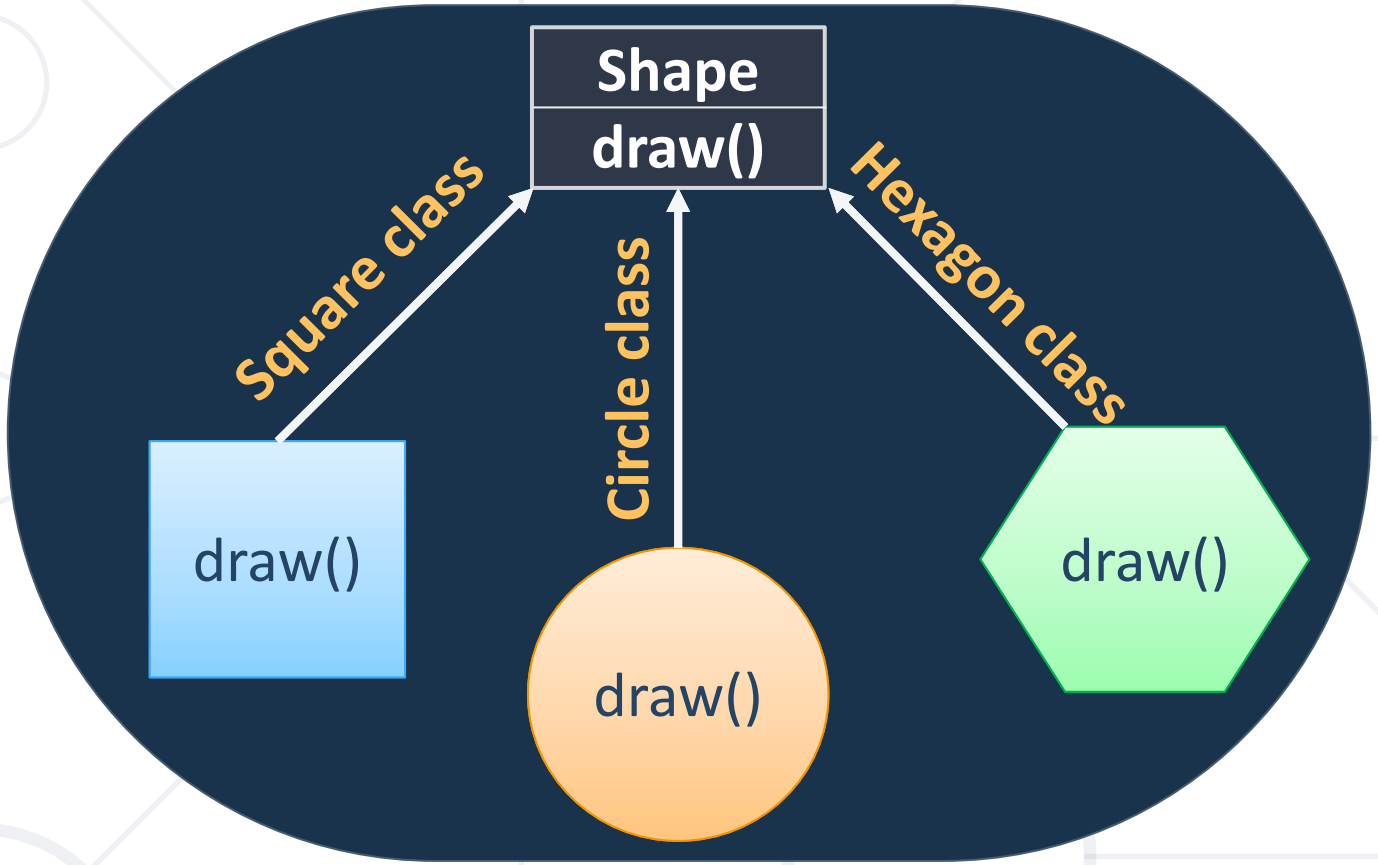
```
static void Main()
{
  Guest guest = new Guest();
  string guest1 = guest.Identity("Stephen", 15);
  string guest2 = guest.Identity(15, "Stephen");
}
```

Same Signatures Different Return Type

- You **cannot** declare two **methods** with the **same signature** and **different return type**

```
static void Print(string text)
{
  Console.WriteLine("Printing");
}

static string Print(string text)
{
  return "Printing";
}
```


Shape
draw()
Square class
Circle class
Hexagon class
draw()
draw()
draw()

Overriding

Runtime Polymorphism (1)

- Also known as **Dynamic Polymorphism** – realized by override a base class function using **`virtual`** or **`override`** keyword

```
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;
  }
}
```

Own definition and implementation

Runtime Polymorphism (2)

- Usage of **override** method

```
public static void Main()
{
  Rectangle rect = new Rectangle(3.0, 4.0);
  Rectangle square = new Square(4.0);

  Console.WriteLine(rect.Area());
  Console.WriteLine(square.Area());
}
```

Method overriding

Runtime Polymorphism (3)

- At run time, objects of a **derived class** may be treated as objects of a **base class**
- When this occurs, the **object's declared type** is no longer identical to **its run-time type**

```
public class Animal
{
…
}
```

```
public class Cat : Animal
{
…
}
```

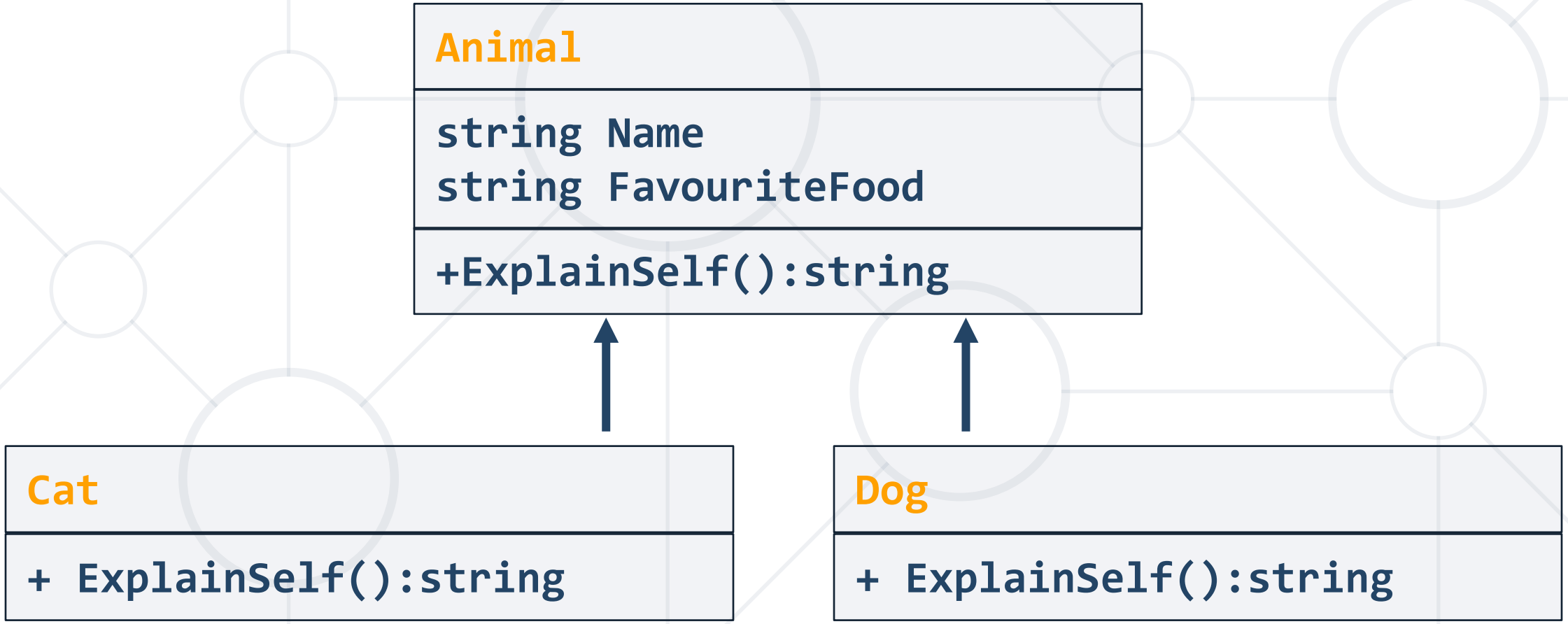
```
Animal cat = new Cat();
```

declared type

run-time type

Problem: Animals

- Implement the following class hierarchy:



Solution: Animals (1)

```
public abstract class Animal
{
  // Create Constructor
  public string Name { get; private set; }
  public string FavouriteFood { get; private set; }
  public virtual string ExplainSelf()
  {
    return string.Format(
      "I am {0} and my favourite food is {1}",
      this.Name,
      this.FavouriteFood);
  }
}
```

Solution: Animals (2)

```
public class Dog : Animal
{
  public Dog(string name, string favouriteFood)
    : base(name, favouriteFood) { }

  public override string ExplainSelf()
  {
    return base.ExplainSelf() +
    Environment.NewLine +
    "BARK";
  }
}
```

Solution: Animals (3)

```
public class Cat : Animal
{
  public Cat(string name, string favouriteFood)
    : base(name, favouriteFood) { }

  public override string ExplainSelf()
  {
    return base.ExplainSelf() +
    Environment.NewLine +
    "MEOW";
  }
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3167#1

Rules for Overriding Method (1)

```
public class Rectangle
{
  public virtual double Area()
  {
    return a * b;
  }
}
```

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

virtual in base method

Same return type and signature

override or abstract in sub-classes

- **Private and static** methods **cannot** be overridden

Rules for Overriding Method (2)

- **Virtual** members use **base keyword** to call the **base class**

```
class Bird
{
  public virtual void Fly()
  {
    Console.Write("Flying");
  }
}
```

```
class Swallow : Bird
{
  public override void Fly()
  {
    base.Fly();
    Console.WriteLine("Hunt");
  }
}
```

Extends the base class virtual method

Can add specific behavior

Rules for Overriding Method (3)

- A derived class can **stop virtual inheritance** by declaring an override as **sealed**

```
class Penguin : Bird
{
  public sealed override void Fly() {}
}
```

```
class NewTypePenguin : Penguin
{
  public new void Fly()
  {
    base.Fly();
  }
}
```

- Sealed methods can be replaced by derived classes by using the **new** keyword
- The **new** modifier hides an accessible base class method

Summary

- Polymorphism – ability of an object to take many forms
- Types of polymorphism:
  - Compile-time
    - Performed through overloading – same method name but different implementation
  - Run-time
    - Performed through overriding – using virtual + override keywords

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://softuni.org
- © Software University – https://softuni.bg