OBJECT-ORIENTED PROGRAMMING WITH C#

POLYMORPHISM

issntt@nus.edu.sg

A common interview question

What is polymorphism?

Have you ever used it in your coding?

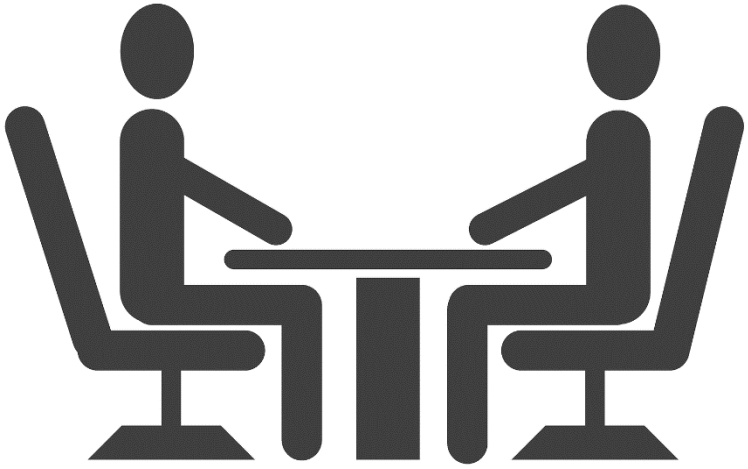


Image by Tumisu from Pixabay

Poll

Have you ever heard that Polymorphism is a **challenging** concept?

Image by Rappellingusa from Pixabay

Objectives

At the end of the course, students will be able to

- Describe the concept of polymorphism
- Describe how polymorphism can manipulate objects with similar interface subset
- Use polymorphic arrays/collections, polymorphic arguments and polymorphic return type to design and implement solutions
- Explain why polymorphism is a key feature of Object-Oriented Programming

Topics

- **Introduction**
- Object References and Inheritance
- Polymorphism
- Casting
- Method override vs Method hiding

Problem

- A company has 2 **different types** of employees, with **different ways of payment**
  - *Salaried Employees: fixed weekly salary*
  - *Hourly Employees: paid by hours*
- For each employee, the company also stores their name

Implement an app that performs its weekly **payroll calculation**

A Solution

1

Create classes for **each type of employees**

Each has **its own method** to **compute** the respective employee's **earnings**

2

Then, keeps **2 arrays**

- An array of *Salaried Employees*
- An array of *Hourly Employees*

3

When **computing** the **pay**

- **Loop** through all *Salaried Employees*
- **Loop** through all *Hourly Employees*
- **Add** the pay together

A Solution - Implementation

```
public class Employee {
    public string Name {
        set; get;
    }
    public Employee(string name) {
        Name = name;
    }
    public virtual double Earnings() {
        return 0;
    }
}
```

A Solution - Implementation

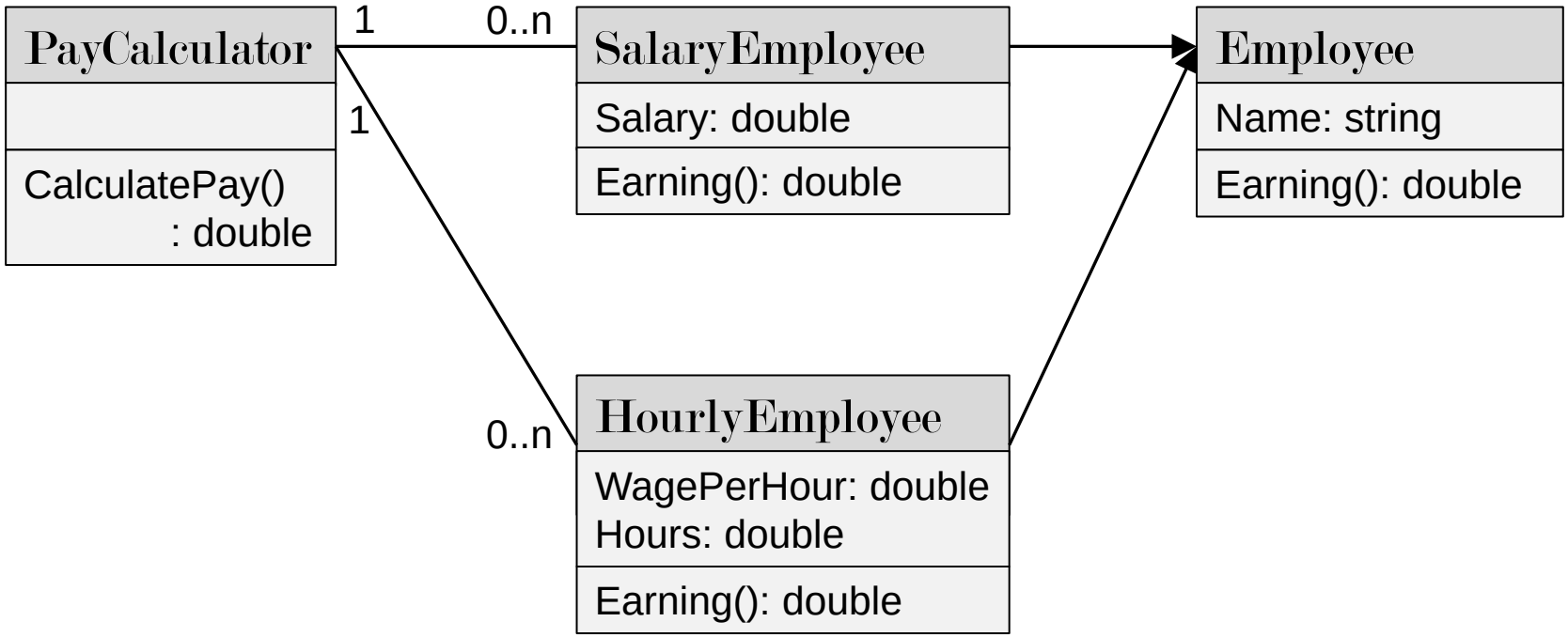
```
public class
  1 SalariedEmployee
                        : Employee {
  public double WeeklySalary
  {
    get; set;
  }
  public SalariedEmployee(
     string name,
     double weeklySalary)
                  : base(name)
  {
    WeeklySalary = weeklySalary;
  }
  public override
         double Earnings()
  {
    return WeeklySalary;
  }
}
```

```
public class
 1 HourlyEmployee
                  : Employee {
  public double WagePerHour {
    set; get;
  }
  private double Hours {
    set; get;
  }
  public HourlyEmployee(
    string name,
    double wagePerHour,
         double hours)
            : base (name)
  {
    WagePerHour = wagePerHour;
    Hours = hours;
  }
  public override
       double Earnings() {
      return WagePerHour *
                          Hours;
  }
}
```

A Solution - Implementation

```
public class PayCalculator {
    private SalariedEmployee[] salariedEmployees;
    private HourlyEmployee[] hourlyEmployees;

    public PayCalculator(SalariedEmployee[] salariedEmployees,
         HourlyEmployee[] hourlyEmployees) {
         this.salariedEmployees = salariedEmployees;
         this.hourlyEmployees = hourlyEmployees;
    }
    public double CalculatePay() {
         double totalPay = 0;
         foreach (SalariedEmployee se in salariedEmployees) {
              totalPay += se.Earnings();
         }
         foreach (HourlyEmployee he in hourlyEmployees)
         {
              totalPay += he.Earnings();
         }
         return totalPay;
    }
}
```

A Solution – Class Diagram



What issues may this design have?

Sorry! Requirements Change

- Now, the company adds **2 more different types** of employees:
  - *Commission Employees: a percentage of their sales*
  - *Salaried-Commission Employees: base salary + a percentage of their sales*
- So totally, the company has **4 different types** of employees, with **different ways of payment**

Implement an app that performs its weekly **payroll calculation**

Is there such change in real life?

A Solution

Just create **4 different classes** and let *PayCalculator* keeps **4 arrays** for different types of Employees

```
public class PayCalculator
{
    private SalariedEmployee[] salariedEmployees;
    private HourlyEmployee[] hourlyEmployees;
    private CommissionEmployee[] commissionEmployees;
    private SalaryCommissionEmployee[] salaryEmployees;

    //...
}
```

What issues may this design have?

Topics

- Introduction
- **Object References and Inheritance**
  - **References and Subtype Objects**
  - What does the Reference Type decide?
  - What does the Object Type decide?
- Polymorphism
- Casting
- Method override vs Method hiding

Review Question

How can we create a reference to an object?

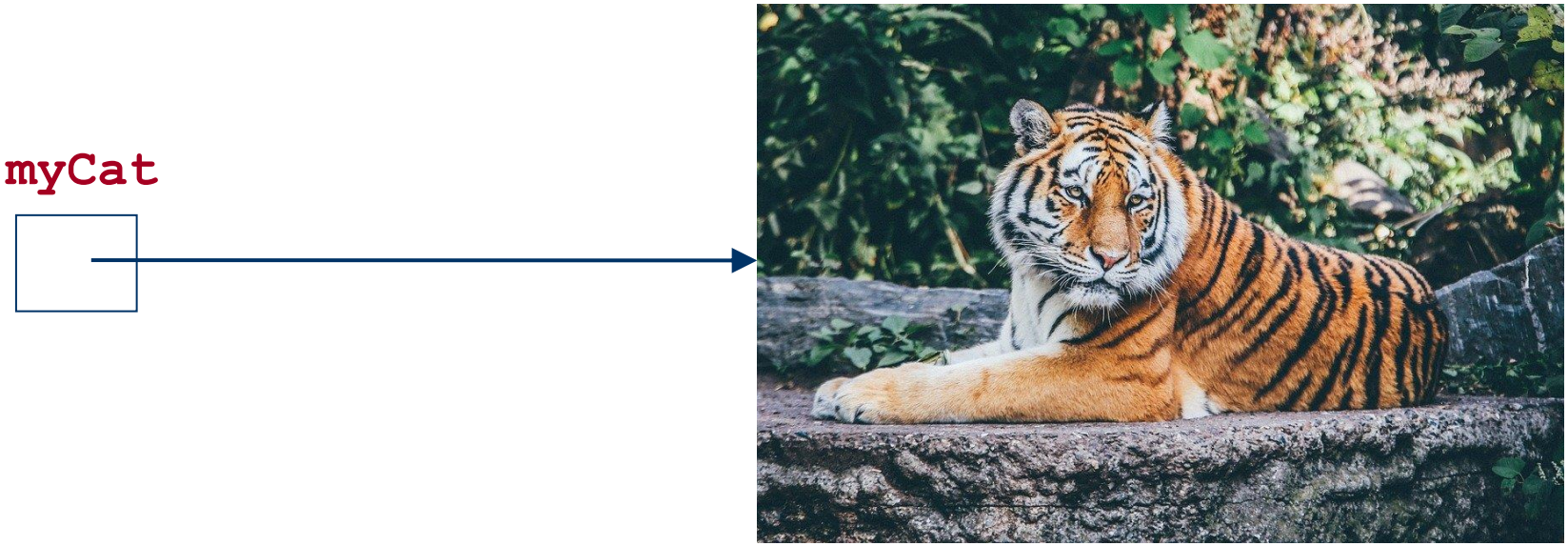
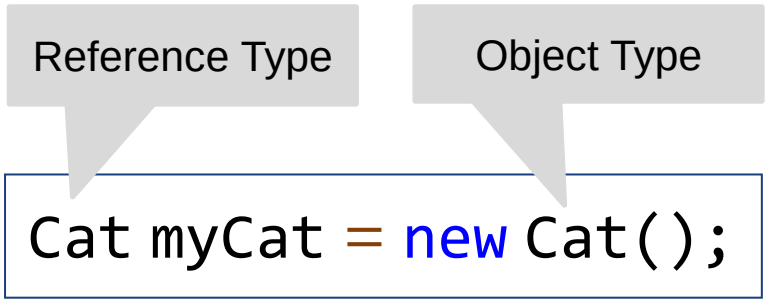


Image by Pexels from Pixabay

Object References Revisit

There are 3 steps of **object instantiation** and **assignment**

1. Create an **object** `new Cat()`
2. Declare a **reference variable** `Cat myCat`
3. Assign the variable to the object `=`



Object References

Variables of a **reference type** can be **always** assigned to **objects** of the **same type**

Reference Type

Object Type

```
Cat myCat = new Cat();
```

```
Dog myDog = new Dog();
```

```
Car myCar = new Car("Toyota Camry", "red");
```

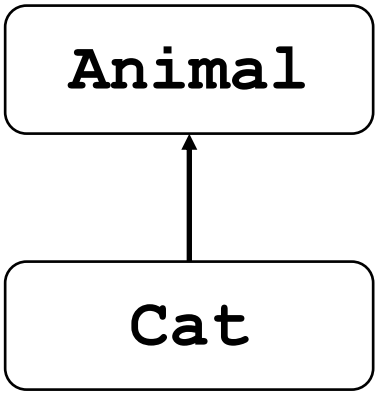
Object References and Inheritance

Can you guess?

Now, let's say class *Cat* is **derived** from class *Animal*

1. Can a variable typed *Cat* reference to an *Animal* object?

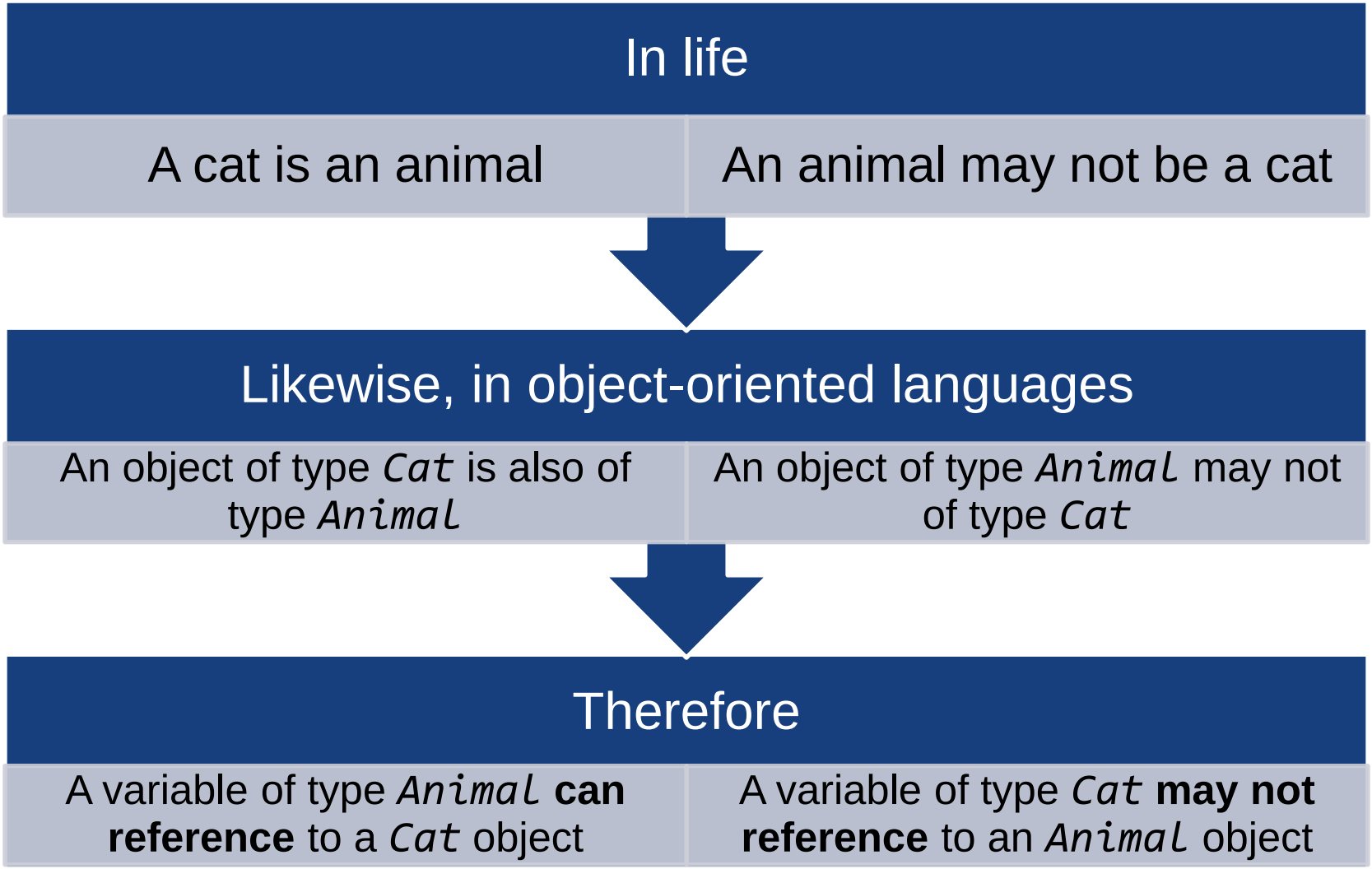
2. Can a variable typed *Animal* reference to a *Cat* object?




```
Cat myAnimal = new Animal();
Animal myCat = new Cat();
```

?

Object References and Inheritance



Object References and Inheritance

Variables of a reference type **can** be **also assigned** to **objects** of its **sub-types**

For example

- **Reference** variable **type** is declared as *Animal*, but
- The **object type** is *Cat*

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

Quiz

Which groups of statements are NOT allowed (1) ?

A

```
Cat myCat1 = new Cat();
Animal myCat2 = new Cat();
```

C

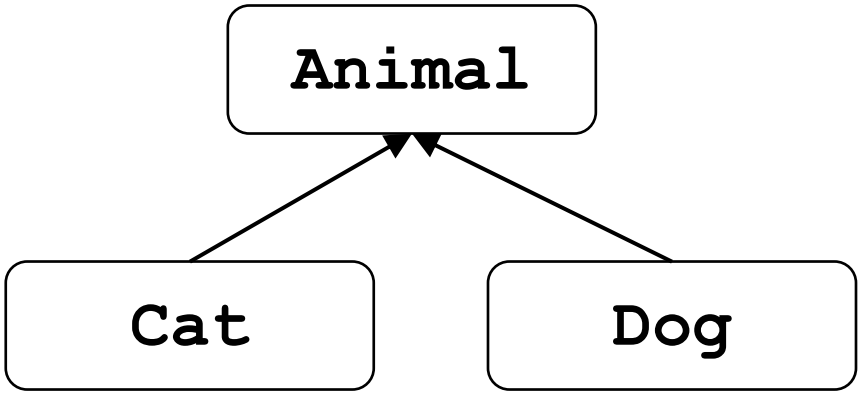
```
Cat myAni1 = new Animal();
Dog myAni2 = new Animal();
```

B

```
Dog myDog1 = new Dog();
Animal myDog2 = new Dog();
```

D

```
Cat myDog = new Dog();
Dog myCat = new Cat();
```



Quiz

Which groups of statements are NOT allowed (2)?

A

```
Cat myCat1 = new Cat();
Animal myCat2 = myCat1;
```

C

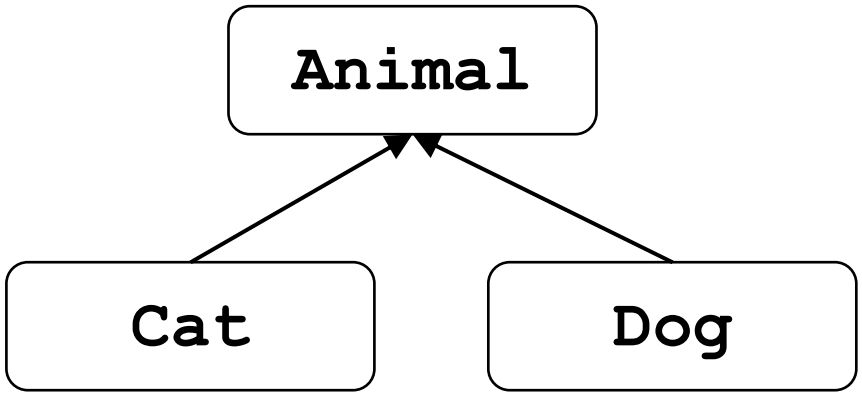
```
Animal myAni1 = new Animal();
Cat myAni2 = myAni1;
```

B

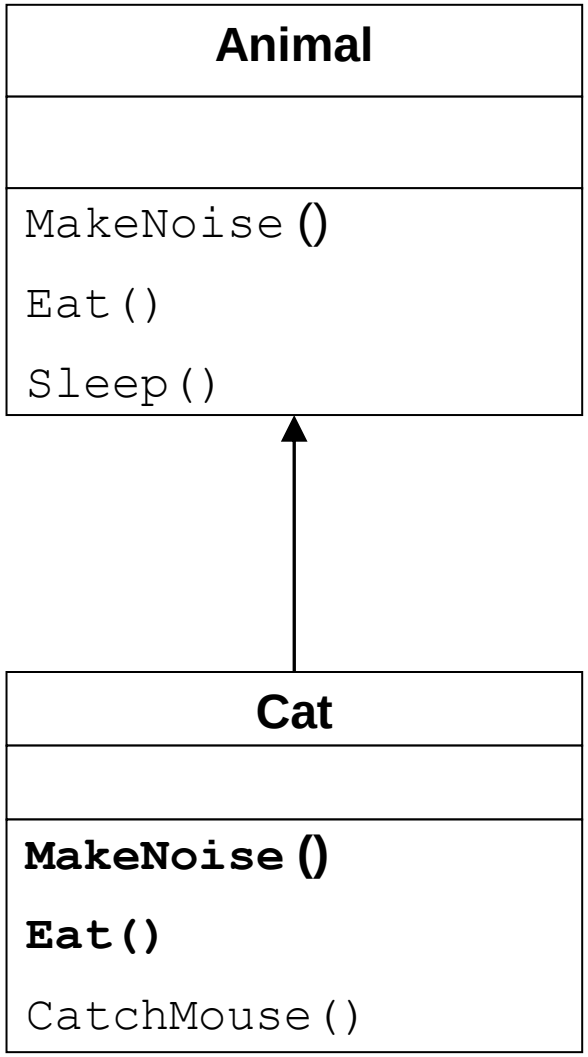
```
Dog myDog1 = new Dog();
Animal myDog2 = myDog1;
```

D

```
Cat myCat1 = new Cat();
Dog myCat2 = myCat1;
```



Next



- Class *Cat* extends class *Animal*, so it inherits all the methods
- It overrides methods *MakeNoise()* and *Eat()*
- It has its own method *CatchMouse()*

```
Cat myCat1 = new Cat();
Animal myCat2 = new Cat();
```

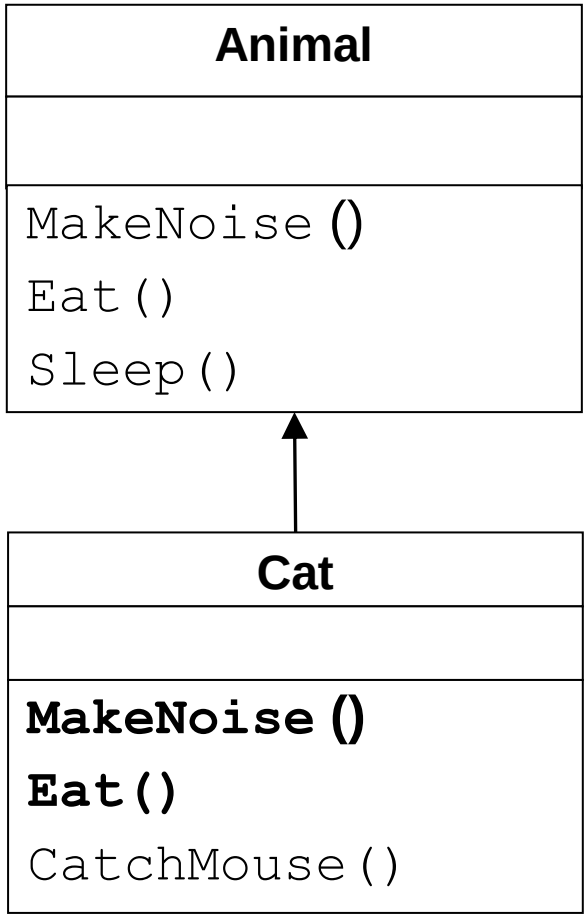
Although the two statements above are **allowed**, how are they **different**?

Topics

- Introduction
- **Object References and Inheritance**
  - References and Subtype Objects
  - **What does the Reference Type decide?**
  - **What does Object Type decide?**
- Polymorphism
- Casting
- Method override vs Method hiding

Quiz

Determine the Reference Type of each variable *myCat* and *myAni*, and the Object Type each references to

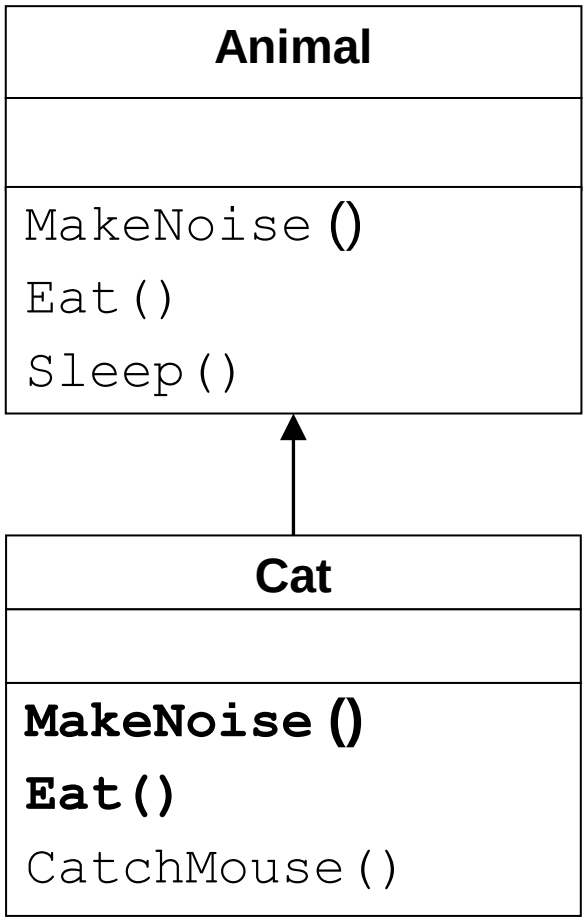



```
Cat myCat = new Cat();

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

Can a variable's Reference Type change? Can the Object Type it assigns to change?

Reference Type decides what can be **called**

Only the **methods** and **data members** in the **reference type** can be called or used




```
Cat myCat1 = new Cat();
Animal myCat2 = new Cat();
```

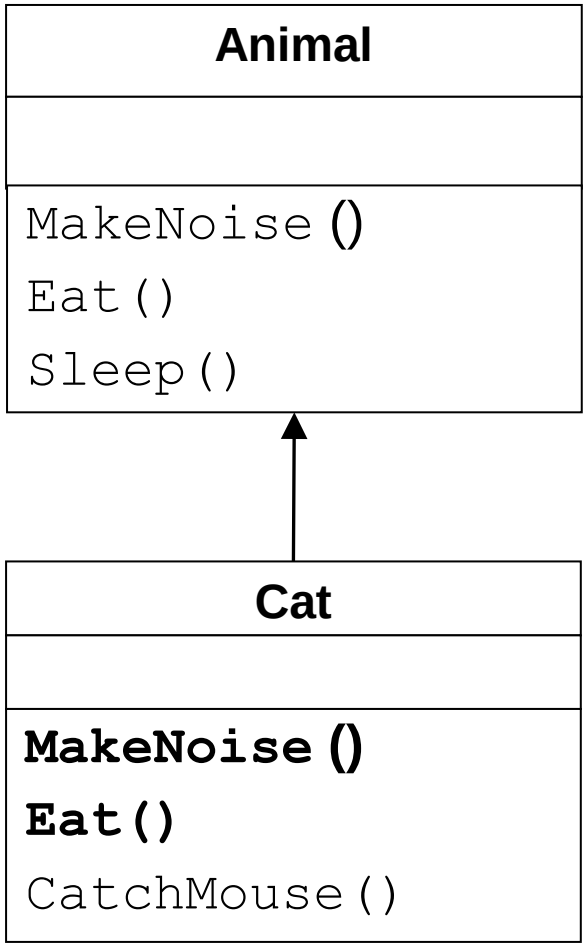
Let's say all the methods are `public`

What methods can *myCat1* call?

What methods can *myCat2* call?

But Object Type decides the overridden method executed

The **method version** that is executed when an **overridden method** is invoked depends on the **Object Type**




```
Animal myAni = new Animal();
myAni.MakeNoise();

myAni = new Cat();
myAni.MakeNoise();
```

For each case, which version of *MakeNoise()* is executed?

Polymorphic Reference

- Because reference *myAni* can reference to **different Object Types** at **different times**, it is called **polymorphic reference**
- Which **version** of the overridden method to be executed is **unknown,**
  - until the **moment** that method is **really executed** at run time
- The compiler uses a mechanism called **dynamic binding** to **execute** the **appropriate version** (aka implementation)

3 most important points so far

1. A reference type variable can reference to an object of the **same** type, or any of its **sub-type**
2. The **Reference Type**, not Object Type, decides what **methods** and **data members** can be **called**
3. The **Object Type**, not Reference Type, decides the **overridden methods to be executed**

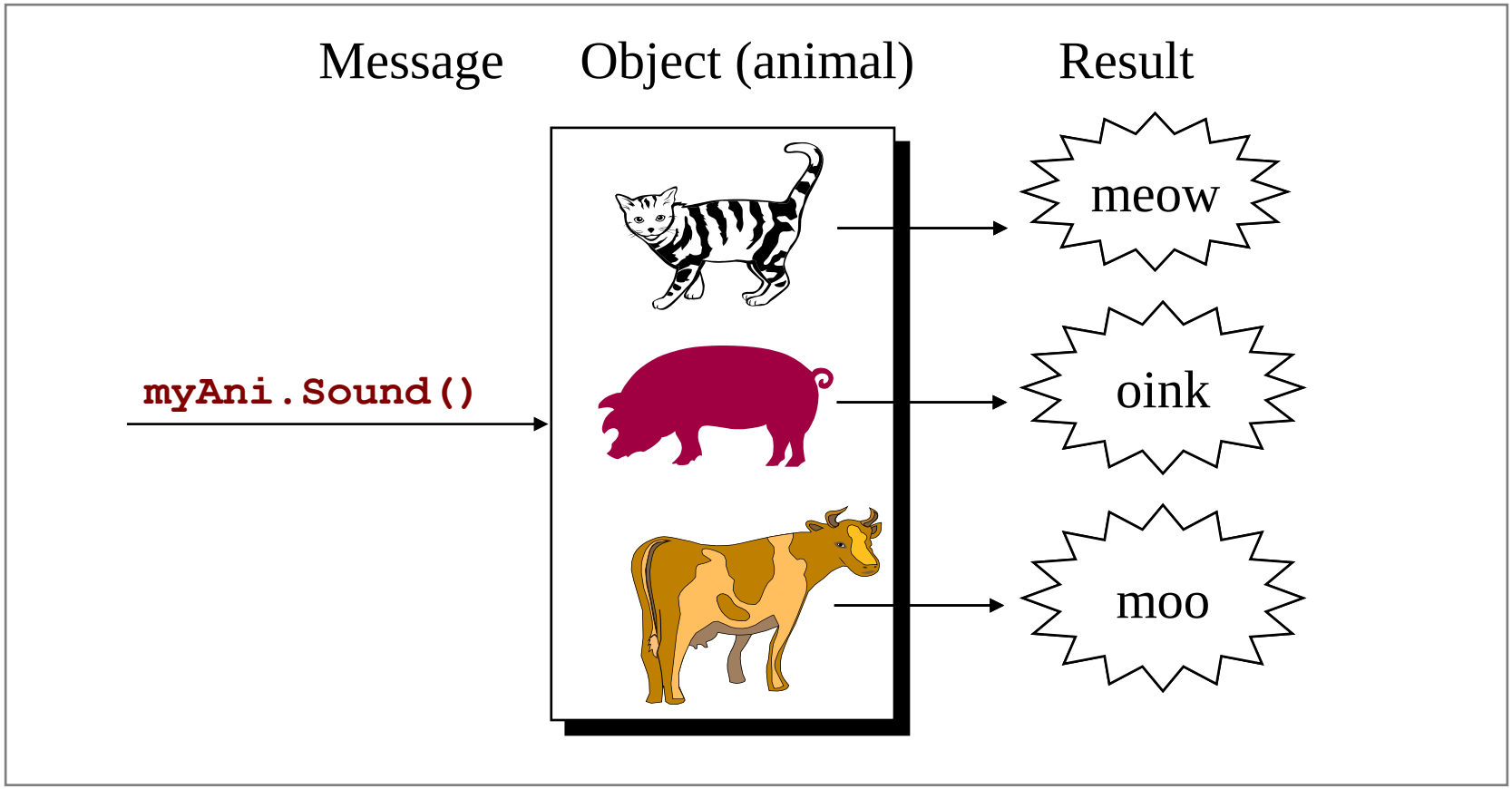
How does it work in code?

Topics

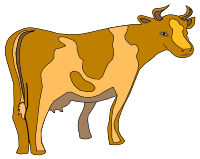
- Introduction
- Object References and Inheritance
- **Polymorphism**
  - **Manipulating objects with similar interface subset**
  - Polymorphic Arrays
  - Polymorphic Arguments
  - Polymorphic Return Type
  - Benefits of Polymorphism
- Casting
- Method override vs Method hiding

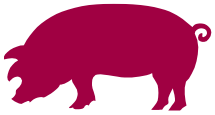
Manipulate objects with similar interface subset

- Let's say there are 3 types of different animals
- They all make sound (**same interface**), but their sounds are different (**different implementation**)



Polymorphism & Dynamic Binding





```
Class Animal
public virtual void Sound() {
   Console.WriteLine("");
}
```

```
Class Cow
public override void
                Sound() {
   Console
     .WriteLine("Moo");
}
```

```
Class Pig
public override void
                Sound() {
   Console
     .WriteLine("Oink");
}
```

```
Class Cat
public override void
                Sound() {
   Console
     .WriteLine("Meow");
}
```

```
Animal myAni;
…
myAni.Sound();
```

Given variable *myAni* type *Animal*. At some point, we'll call the *Sound()* method. During **compile-time**, do we know what will be the output?

Polymorphism & Dynamic Binding

No, we don't know! We only know when **program really runs**

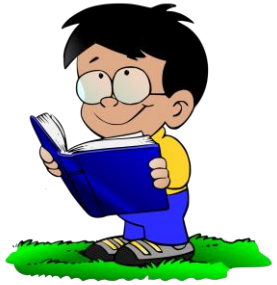
1. Variable *myAni* is a **polymorphic reference**, which can hold any *Animal*, *Cow*, *Pig* or *Cat* object
2. Through dynamic binding, the call to *Sound()* will invoke the corresponding implementation
   - depending on the **type of object** held by *myAni* at that moment

```
1 Animal myAni;
  …
2 myAni.Sound();
```

Quiz

Given the Animal class hierarchy in the last few slides, what is the output of the following program?

```
public static void Main() {
    Animal myAni;
    myAni = new Cat();
    myAni.Sound();

    myAni = new Cow();
    myAni.Sound();

    myAni = new Cat();
    myAni.Sound();
}
```



Topics

- Introduction
- Object References and Inheritance
- **Polymorphism**
  - Manipulating objects with similar interface subset
  - **Polymorphic Arrays**
  - **Polymorphic Arguments**
  - **Polymorphic Return Type**
  - Benefits of Polymorphism
- Casting
- Method override vs Method hiding

Polymorphic Arrays

We can **put** and **mix** many objects of **a type and its subtypes** into the **same** array

```
Animal[] myAnimals = new Animal[4];
myAnimals[0] = new Cat();
myAnimals[1] = new Cow();
myAnimals[2] = new Pig();
myAnimals[3] = new Cow();
```

We can **loop** through the array, call the super-type's methods, and the version of the **respective object** will be executed

Polymorphic Arrays

What is the output of the following program?

```
public static void Main() {
    Animal[] myAnimals = new Animal[4];
    myAnimals[0] = new Cat();
    myAnimals[1] = new Cow();
    myAnimals[2] = new Pig();
    myAnimals[3] = new Cow();

    foreach (Animal myAni in myAnimals)
    {
        myAni.Sound();
    }
}
```

One of the best parts of polymorphism!

Polymorphic Arguments

What is the output of the following program?

```
public static void MakeManySound(
    1 Animal myAni, int times)
{
   for (int i = 0; i < times; i++)
   {
    2 myAni.Sound();
   }
}
```

```
public static void Main() {
   Cat myCat = new Cat();
   MakeManySound(myCat, 3);

   MakeManySound(new Cow(), 2);
   MakeManySound(new Pig(), 2);
}
```

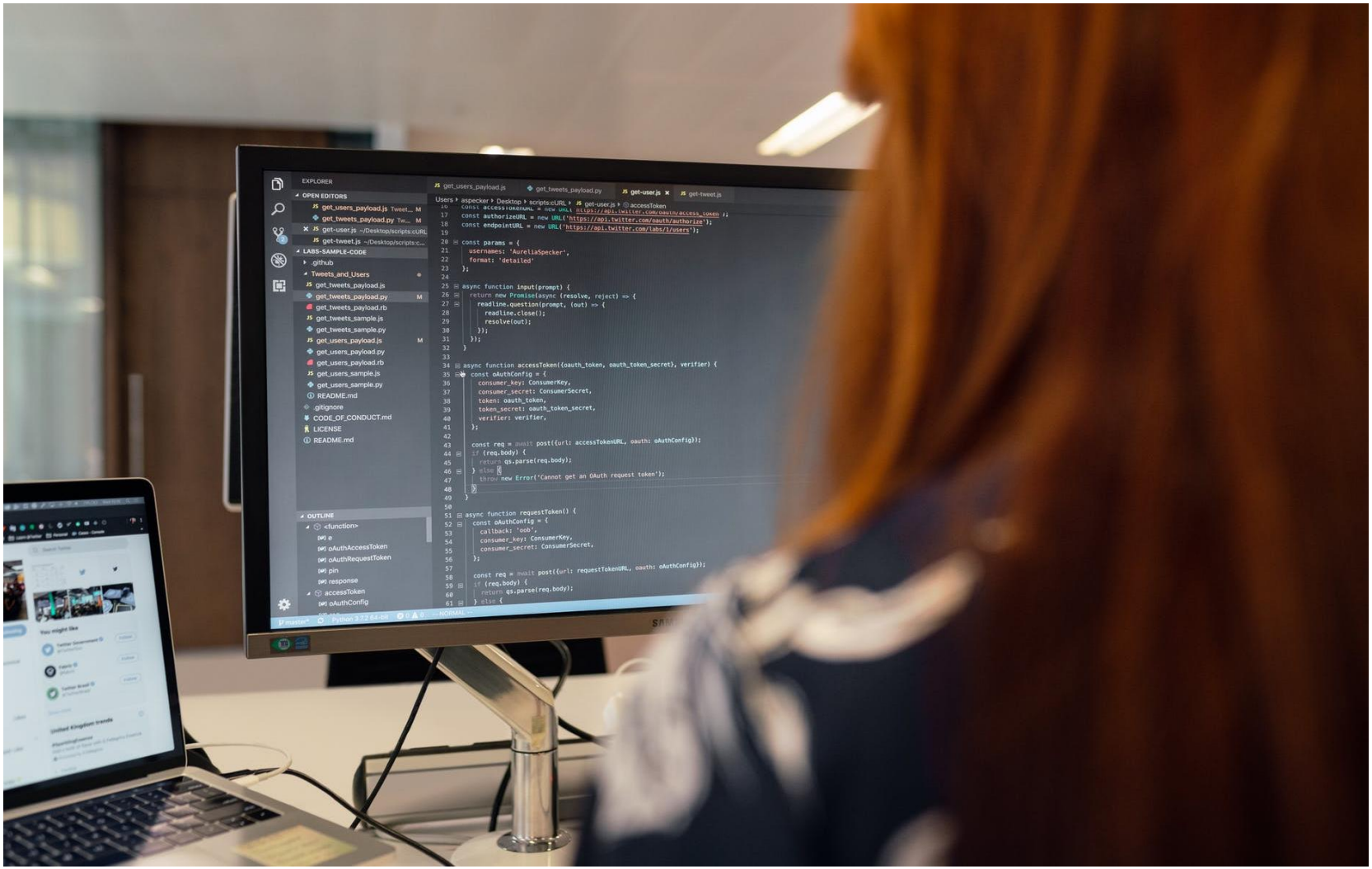
Polymorphic Return Type

What is the output of the following program?

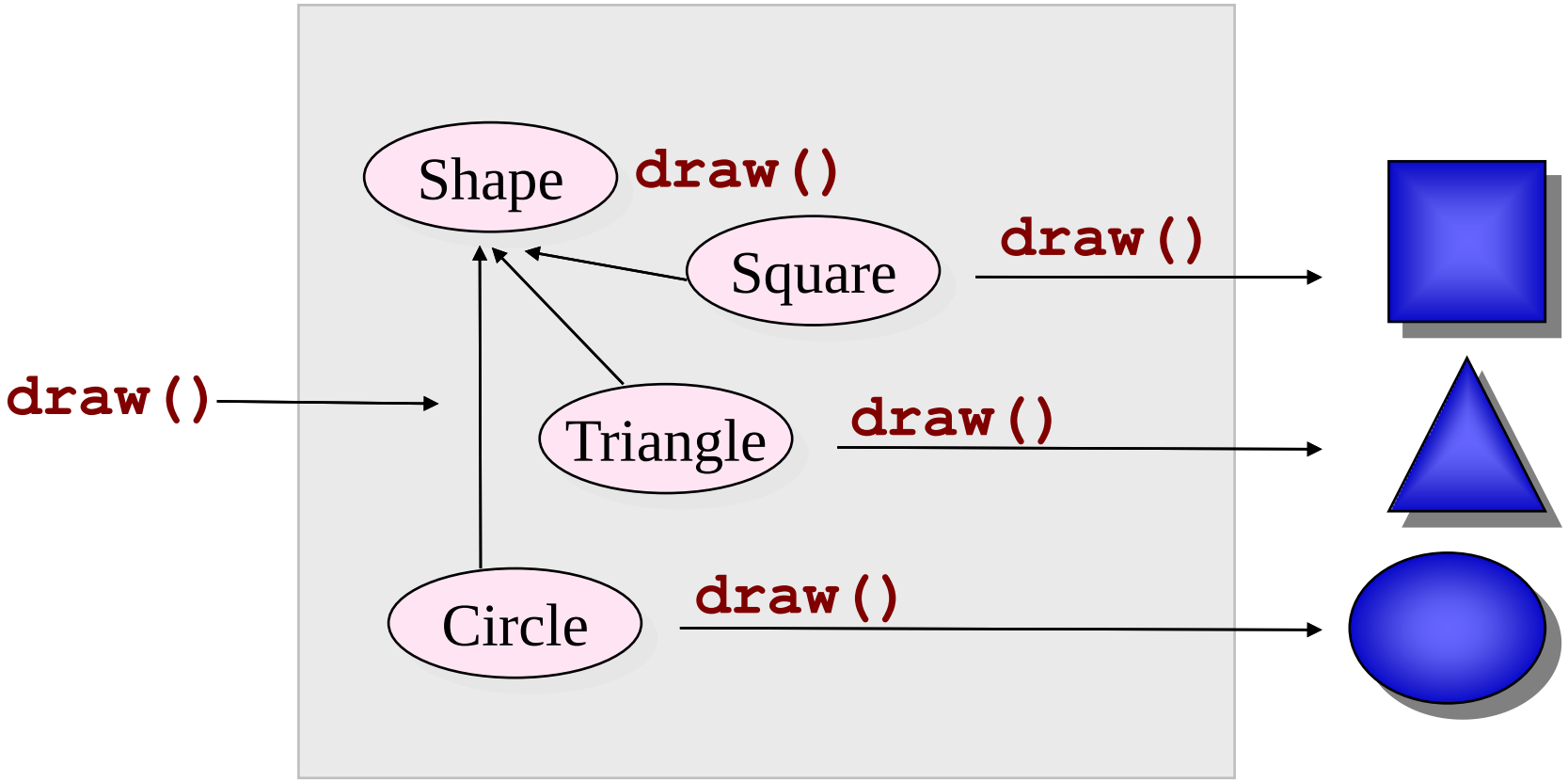
```
static 1 Animal CreateAnimal(
        string type)
{
    switch (type)
    {
        case "cat":
            return 2 new Cat();

        case "pig":
            return 2 new Pig();

        case "cow":
            return 2 new Cow();

        default:
            return null;
    }
}
```

```
public static void Main() {
    Animal myAni1 =
            CreateAnimal("cat");
    Animal myAni2 =
            CreateAnimal("cow");
    myAni1.Sound();
    myAni2.Sound();

    CreateAnimal("cat").Sound();
}
```

Code Demo



Another Example



Another Example - Code

```
class Shape
{
    public virtual void Draw() { }
}
```

```
class Rectangle : Shape
{
   double width;
   double height;
   public Rectangle(
       double width, double height)
   {
      this.width = width;
      this.height = height;
   }

   public override void Draw() {
      Console.WriteLine(
         "A rectangle with width {0},
            height {1}", width, height);
   }
}
```

```
class Circle : Shape
{
   double radius;
   public Circle(double radius) {
      this.radius = radius;
   }
   public override void Draw() {
      Console.WriteLine(
         "A circle with radius {0} ",
                                  radius);
   }
}
```

```
class Square : Shape
{
   double length;
   public Square(double length) {
      this.length = length;
   }
   public override void Draw()
   {
      Console.WriteLine(
         "A square with length {0}",
                                  length);
   }
}
```

Another Example - Code

```
class Drawing
{
   List<Shape> allshapes =
               new List<Shape>();
   public void Add(Shape s)
   {
      allshapes.Add(s);
   }
   public void Draw()
   {
      for (int i = 0;
         i < allshapes.Count; i++)
      {
         Shape s = allshapes[i];
         s.Draw();
      }
   }
}
```

```
public static void Main()
{
   Drawing drawing =
      new Drawing();
   drawing.Add(
      new Circle(1));
   drawing.Add(
      new Rectangle(2, 3));
   drawing.Add(
      new Square(4));
   drawing.Add(
      new Circle(4.5));
   drawing.Draw();
}
```

```
A circle with radius 1
A rectangle with width 2, height 3
A square with length 4
A circle with radius 4.5
```

Topics

- Introduction
- Object References and Inheritance
- **Polymorphism**
  - Manipulating objects with similar interface subset
  - Polymorphic Arrays
  - Polymorphic Arguments
  - Polymorphic Return Type
  - **Benefits of Polymorphism**
- Casting
- Method override vs Method hiding

Polymorphism Benefits

- Enable us to write code that **doesn't have to be changed** while **new subclass** types are introduced
- Therefore, many **frameworks** can be implemented **without knowing** exactly what sub-classes will be written

In our Shape example, which class **doesn't need to be changed** if we need another subclass of *Shape*, e.g., *Triangle*?

Topics

- Introduction
- Object References and Inheritance
- Polymorphism
- **Casting**
  - **Up-casting**
  - **Down-casting**
- Method override vs Method hiding

Up Casting

As we have known, variables of a reference type **can** be **assigned** to **objects** of its **sub-types**

Variable of a **superclass**

Instance of a **subclass**

```
Person p = new Student();
Animal myCat = new Cat();
```

This is **also called up casting**

Down Casting

And variables of a reference type **cannot** be **assigned** to **objects** of its **super-types**

```
Cat myCat = new Cat();
Animal myAnimal = myCat;
Cat myCat2 = myAnimal;
```

Variable of a **subclass**

Instance of a **superclass**

Down Casting

However, we can use casting to assign a **variable** of a reference type to object of its super-types

```
Cat myCat = new Cat();
Animal myAnimal = myCat;
Cat myCat2 = (Cat) myAnimal;
```

Variable of a **subclass**

Instance of a **superclass**

This is **called down casting** and must be done **explicitly**

Wrong down-casting will cause **runtime errors!**

Explicit casting means that we the developers declare we KNOW variable *myAnimal* is referencing to an object of Cat type

Casting – Down Casting

If we are **not sure** the **type** of an **object**, we should **check** the down-casting in either of the following ways

```
Animal myAni;
// more code here
if (1) (myAni is Cat)
{
  Cat myCat = (2) (Cat) myani;
  myCat.catchMouse();
}
```

```
Animal myAni;
// more code here
Cat myCat =
        (3) myAni as Cat;
if ((4) myCat != null) {
  // correct down-casting
  myCat.catchMouse();
}
else {
  // wrong down-casting
}
```

Option 1
1. Check with `is`, then
2. Down cast

Option 2
3. Downcast with `as`, then
4. Check against `null`

Topics

- Introduction
- Object References and Inheritance
- Polymorphism
- Casting
- **Method override vs Method hiding (Self-study)**

Method Override vs Method hiding

Self study

- Up to now, we know that a method override its base class while:
  - It has the **same name**, **return type** and **signature** as the inherited method
  - It contains a **different implementation** in the body
  - It is declared as `override` method
- Thanks to **overriding**, method of the **object type**, not the variable reference type is executed
  - and therefore, there are all polymorphic benefits

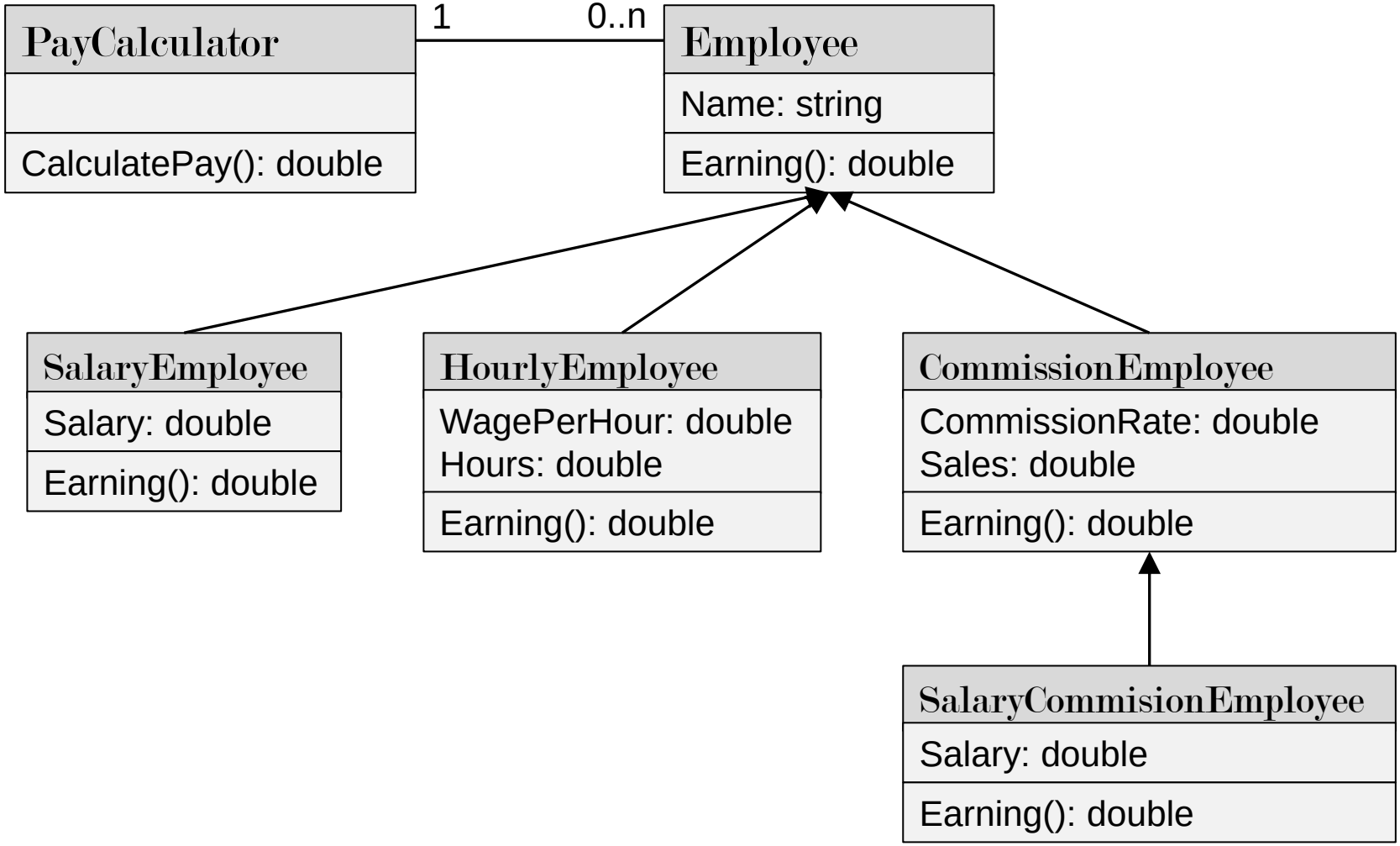
Method Hiding

Self study

- If you don't want the polymorphic benefits, you can **hide the base method** instead of overriding it
- To **hide**, or **redefine**, an inherited member in a derived class, method is declared with `new` keyword
- This is **not commonly** used

This topic is **non-examinable**. If interested, you can read this article for more information

Problem Solution



Problem Solution

```
public class Employee
{
    public string Name { get; }
    public Employee(string name) {
        Name = name;
    }
    public virtual double Earnings() {
        return 0;
    }
}
```

```
class SalaryEmployee : Employee
{
    public double Salary { get; }
    public SalaryEmployee(
        string name,
        double salary) : base(name)
    {
        Salary = salary;
    }
    public override double Earnings()
    {
        return Salary;
    }
}
```

```
class HourlyEmployee : Employee {
    public double WagePerHour
    { get; }
    public double Hours { get; }
    public HourlyEmployee(
        string name,
        double wagePerHour,
        double hours) : base(name) {
        WagePerHour = wagePerHour;
        Hours = hours;
    }
    public override double Earnings()
    {
        return WagePerHour * Hours;
    }
}
```

Problem Solution

```
class CommissionEmployee : Employee
{
   public double CommissionRate
   { get; }
   public double Sales { get; }
   public CommissionEmployee(
      string name,
      double commisionRate,
      double sales) : base(name)
   {
      CommissionRate = commisionRate;
      Sales = sales;
   }

   public override double Earnings()
   {
      return CommissionRate * Sales;
   }
}
```

```
class SalaryCommisionEmployee :
                CommissionEmployee
{
   public double Salary { get; }
   public SalaryCommisionEmployee(
      string name,
      double salary,
      double commisionRate,
      double sales) : base(
         name, commisionRate, sales)
   {
      Salary = salary;
   }

   public override double Earnings()
   {
      return Salary +
               base.Earnings();
   }
}
```

Problem Solution

```
public class PayCalculator
{
   private Employee[] employees;

   public PayCalculator(
         Employee[] employees)
   {
      this.employees = employees;
   }

   public double CalculatePay()
   {
      double totalPay = 0;
      foreach (
          Employee e in employees)
      {
         totalPay += e.Earnings();
      }

      return totalPay;
   }
}
```

```
public static void Main()
{
   Employee[] emps = new Employee[5];
   emps[0] =
        new SalaryEmployee("A", 5000);
   emps[1] =
        new SalaryEmployee("B", 4000);
   emps[2] =
        new HourlyEmployee("C", 20, 100);
   emps[3] =
        new CommissionEmployee(
                  "D", 0.1, 30000);
   emps[4] =
        new SalaryCommisionEmployee(
                  "E", 2000, 0.2, 10000);

   PayCalculator calculator =
        new PayCalculator(emps);
   Console.WriteLine(
        calculator.CalculatePay());
}
```

```
18000
```

Readings

- Visual C# 2012 How to Program, 5th edition – Chapter 12, *Paul Deitel and Harvey Deitel*
- Head First C#, 3rd edition – Chapter 6, *Andrew Stellman and Jennifer Greene*