CSC109

Inheritance and Polymorphism

Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Objectives

1. Use inheritance to eliminate repeated code
2. Use virtual methods and polymorphism to write generic code

Use inheritance to eliminate repeated code

Tasks

1. Apply generalization to classes
2. Create one class that derives from another

Model objects

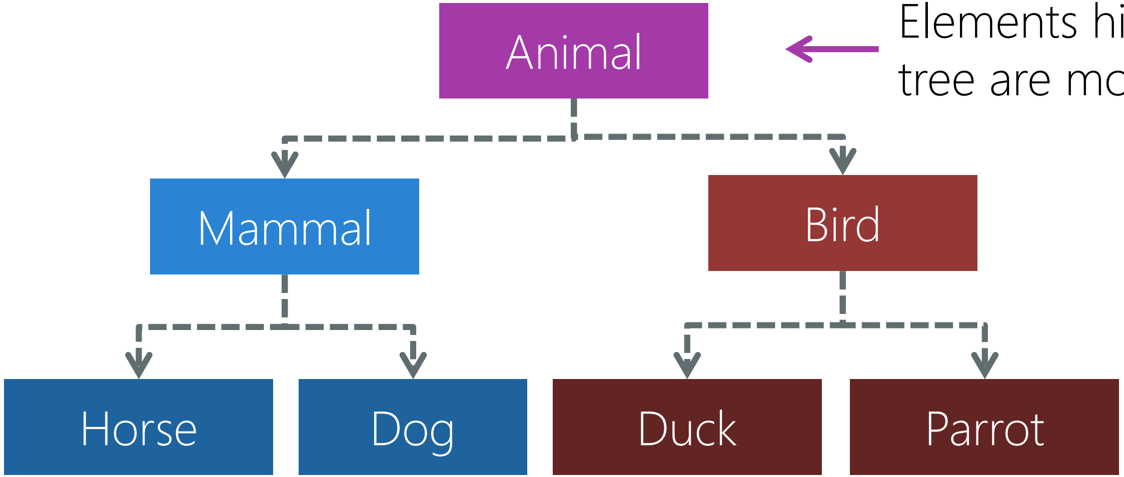
- We often write classes to model objects and organize hierarchies for objects in the real world

= Animal

= Computer

Generalization and specialization

- Humans create to hierarchies to describe relationships between similar objects and help them deal with complexity

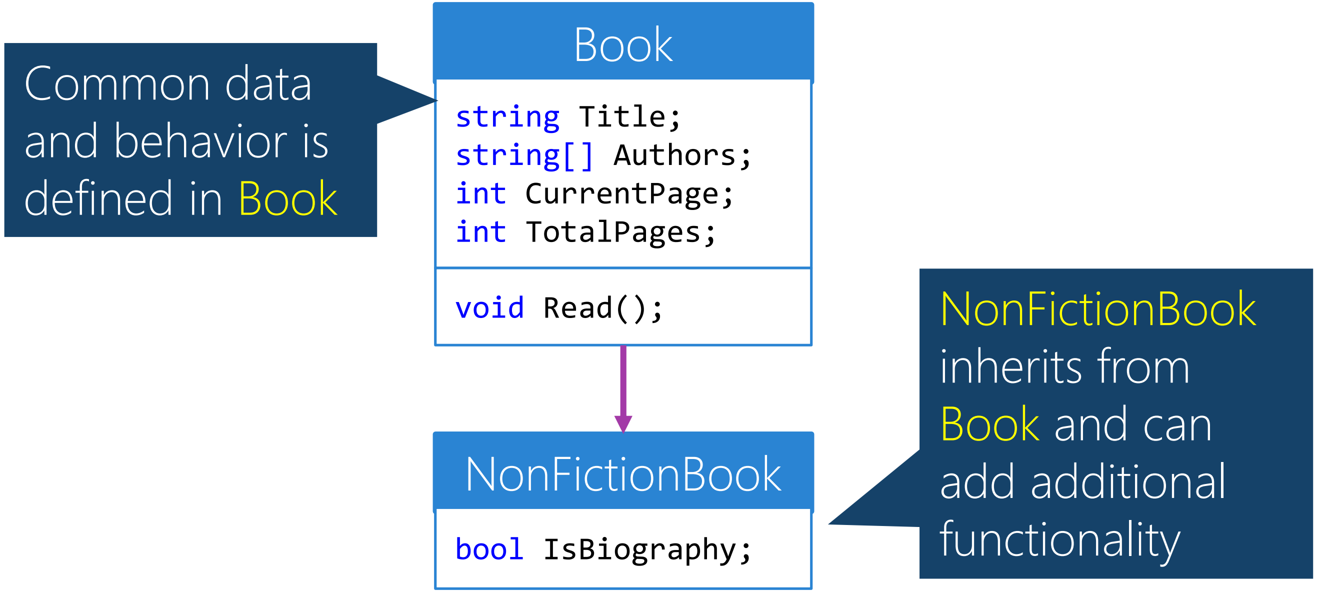


Elements higher on the tree are more general

Elements lower on the tree are more specialized

What is inheritance?

- Inheritance is a C# language feature that allows you to define a new class that extends an existing class



Common data and behavior is defined in Book

Book

```
string Title;
string[] Authors;
int CurrentPage;
int TotalPages;
```

```
void Read();
```

NonFictionBook

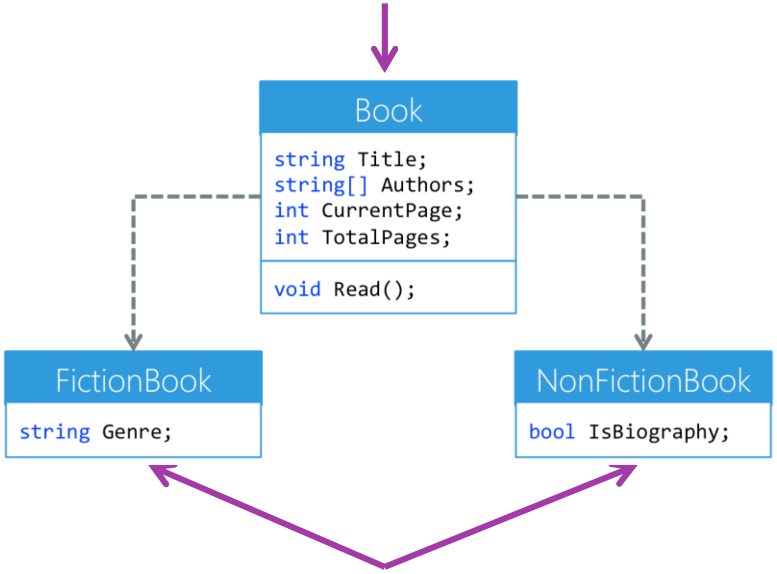
```
bool IsBiography;
```

NonFictionBook inherits from Book and can add additional functionality

Definitions

- Parent class is often called the *base class*, or sometimes the *super class*
- Child class that inherits from the parent is called the *derived class*, or sometimes the *subclass*

Parent or base class



```
string Title;
string[] Authors;
int CurrentPage;
int TotalPages;

void Read();
```

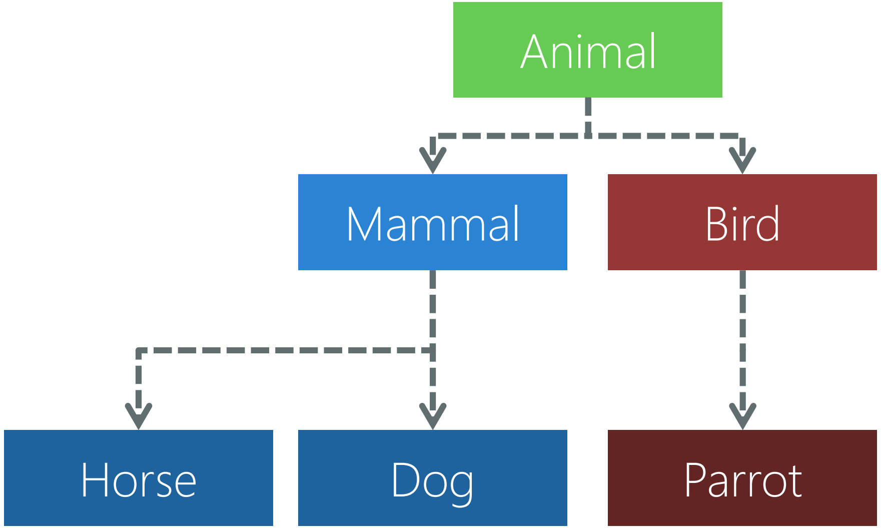
```
string Genre;
```

```
bool IsBiography;
```

Derived or subclass

Why use inheritance?

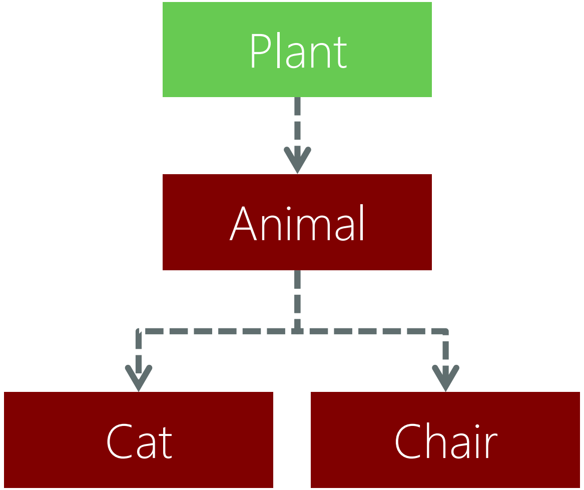
- Inheritance *avoids duplication* of common logic and data and *promotes* reuse of tested and working code



When *not* to use inheritance

- Inheritance should be used to describe an "is-a" relationship (e.g. **Horse** is-a **Mammal**)

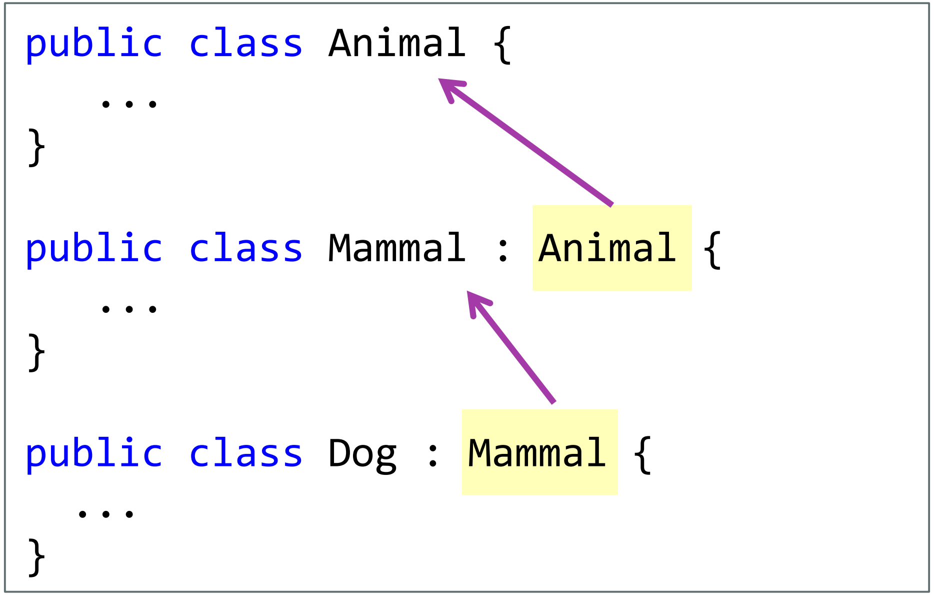
- Do <u>not</u> use inheritance for unrelated classes – this can produce unneeded properties in the derived classes



These objects may have things in common, but they cannot be used *interchangeably*

Using inheritance in C#

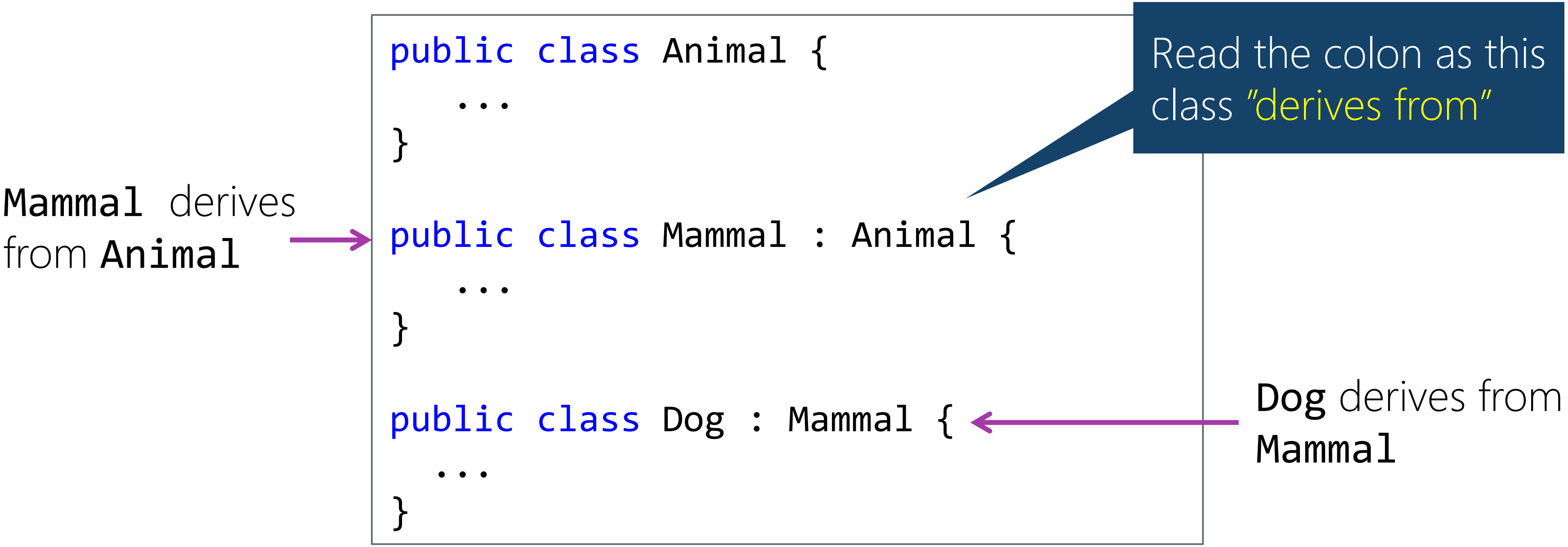
- In C#, we can indicate a class *derives* from another when we define the class



```
public class Animal {
    ...
}

public class Mammal : Animal {
    ...
}

public class Dog : Mammal {
   ...
}
```

Using inheritance in C#

- In C#, we can indicate a class *derives* from another when we define the class



```
public class Animal {
    ...
}

public class Mammal : Animal {
    ...
}

public class Dog : Mammal {
   ...
}
```

Read the colon as this class "derives from"

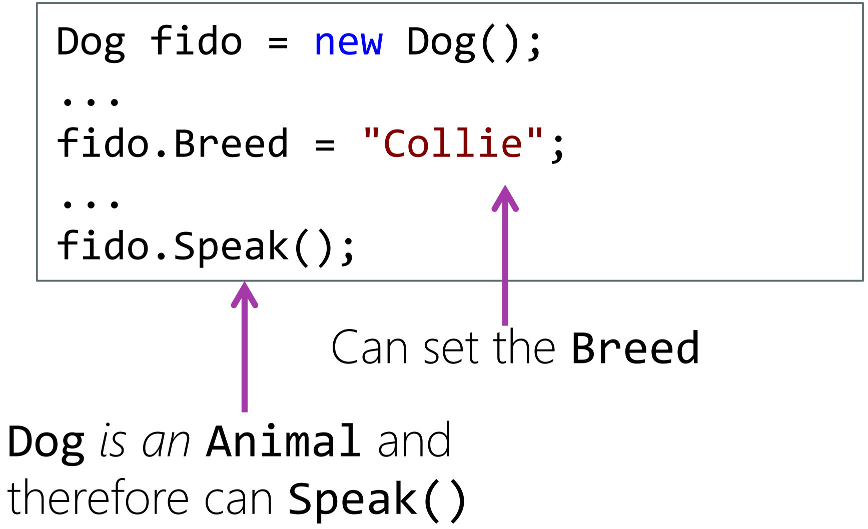
Mammal derives from **Animal**

Dog derives from **Mammal**

Treating classes generically

❖ Inheritance allows our code to treat a derived class like the base class, we can access all the common public fields, methods and properties

```
public class Animal {
  public void Speak() {...}
}

public class Dog : Animal {
  public string Breed { get; set; }
  ...
}
```



```
Dog fido = new Dog();
...
fido.Breed = "Collie";
...
fido.Speak();
```

Can set the **Breed**

Dog *is an* **Animal** and therefore can **Speak()**

Treating classes generically

- Inheritance allows our code to treat a derived class like the base class, we can access all the common public fields, methods and properties

```
public class Animal {
  public void Speak() {...}
}

publ
  p
  .
}
```

We create a new **Dog** instance, but *assign it* to an **Animal** variable, this works but limits our access to things defined only on the **Animal** class

```
Animal animal = new Dog();
...
animal.Breed = "Collie"; X
...
animal.Speak();
```

Can be useful to create a collection of animals that includes Dogs and Cats and Horses

Going back to a derived type

- Use the **as** C# keyword to try to cast a reference to a more derived type, this returns **null** if it is unsuccessful

Takes the generic **Animal** object and returns it as a **Dog** so we can access dog-specific properties and methods

```
Animal animal = new Dog();
...

Dog fido = animal as Dog;

if (fido != null)
{
    fido.Breed = "Collie"; ✓
}
```

Testing for a derived type

- Can use the **is** C# keyword to test an instance to see if it is a specific type – returns **true**/**false** result

```
Animal animal = new Dog();
...

if (animal is Dog) {
   Dog dog = (Dog) animal;
}
```

Common to do an explicit cast after the test – this is not as efficient as using **as**

Working with constructors

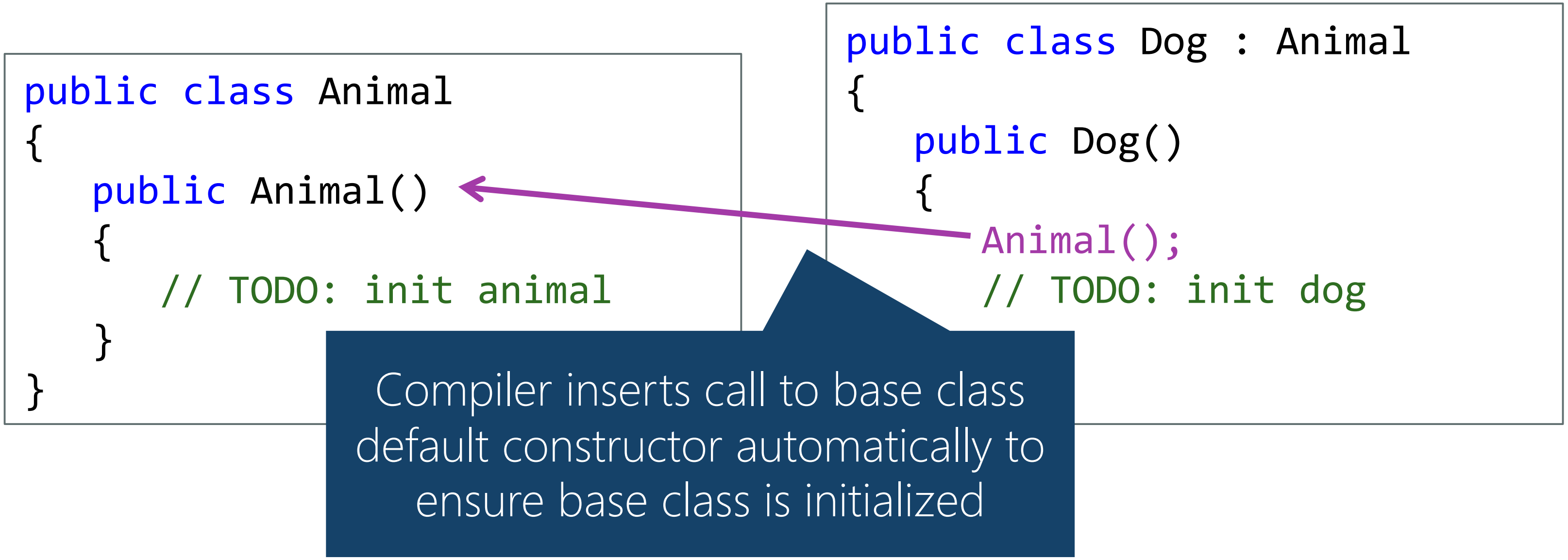
- Constructors for each class work together to initialize the object

```
public class Animal
{
    public Animal()
    {
        // TODO: init animal
    }
}
```

```
public class Dog : Animal
{
    public Dog()
    {

        // TODO: init dog
    }
}
```

Working with constructors

- Constructors for each class work together to initialize the object



```
public class Animal
{
    public Animal()
    {
        // TODO: init animal
    }
}
```

```
public class Dog : Animal
{
    public Dog()
    {
        Animal();
        // TODO: init dog
```

Compiler inserts call to base class default constructor automatically to ensure base class is initialized

Working with custom constructors

- If the base class does not have a default constructor, then the derived class must call the constructor itself

```
public class Animal
{
   public Animal(string name)
   {
      // TODO: init animal
   }
}
```

```
public class Dog : Animal
{
   public Dog()
   {
      // TODO: init dog
   }
}
```

X The type 'Animal' does not contain a constructor that takes `0' arguments

Working with custom constructors

- If the base class does not have a default constructor, then the derived class must call the constructor itself

```
public class Animal
{
   public Animal(string name)
   {
      // TODO: init animal
   }
}
```

```
public class Dog : Animal
{
   public Dog()
     : base("Fido")
   {
      // TODO: init dog
```

"base" keyword indicates base class and is used here to call the base constructor and pass a string so we can create the Dog

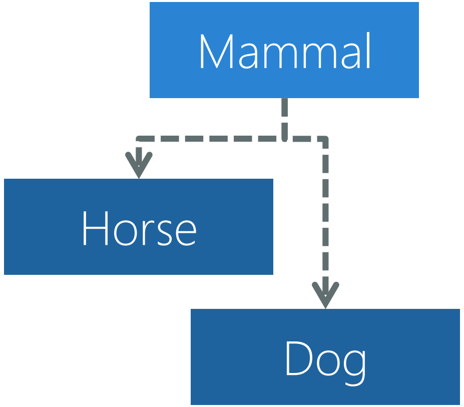
Flash Quiz

Flash Quiz

① Is the following statement correct?

```
Mammal fido = new Dog();
```

a) No, that will not compile
b) Yes, it works without any restrictions
c) Yes, but you only get the **`Mammal`** properties and methods



Flash Quiz

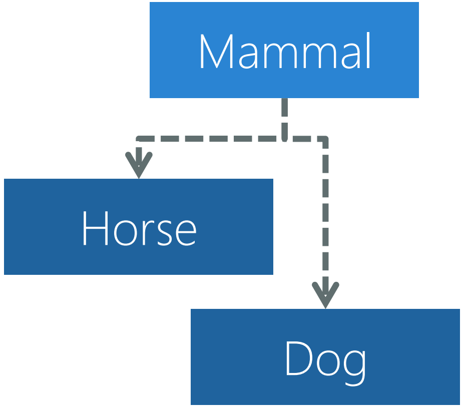
① Is the following statement correct?

```
Mammal fido = new Dog();
```

a) No, that will not compile

b) Yes, it works without any restrictions

c) Yes, but you only get the **`Mammal`** properties and methods



Xamarin University

Flash Quiz

② Inheritance allows you to treat a derived class like the base class?

a) True

b) False

Flash Quiz

② Inheritance allows you to treat a derived class like the base class?

a) True

b) False

Flash Quiz

③ We use inheritance to express _________ relationships

a) "is-a"

b) "has-a"

c) "wants-a"

d) monogamous

Xamarin University

Flash Quiz

③ We use inheritance to express ________ relationships

a) "is-a"

b) "has-a"

c) "wants-a"

d) monogamous

Individual Exercise

Inheritance: create a derived class

Summary

1. Apply generalization to classes
2. Create one class that derives from another

Use virtual methods and polymorphism to write generic code

Xamarin
University

Tasks

1. Create a virtual method
2. Override methods in derived class
3. Model abstract concepts using abstract classes

What is Polymorphism?

- Polymorphism means *many-shaped* and it has two aspects:

Derived classes can be treated as objects of a base class at runtime

Derived classes can *replace* or *extend* the behavior of the base class

Treating derived classes like base classes

- Derived classes should always have an "is-a" relationship to the base class, therefore they can be treated just like the base class in code

```
public class Fruit
public class Apple : Fruit
public class Banana : Fruit
public class Grape : Fruit
...
```

```
Fruit badFruit = new Apple();

Fruit[] fruits = {
   new Apple(),
   new Banana(),
   new Grape()
}
```

Treating derived classes like base classes

- Derived classes should always have an "is-a" relationship to the base class, therefore they can be treated just like the base class in code

```
public cla
public cla
public cla
public class Grape : Fruit
...
```

Can assign a derived type to a field declared as a base type

```
Fruit badFruit = new Apple();

Fruit[] fruits = {
   new Apple(),
   new Banana(),
   new Grape()
}
```

Treating derived classes like base classes

- Derived classes should always have an "is-a" relationship to the base class, therefore they can be treated just like the base class in code

```
public class Fruit
public class Apple : Fruit
public class Banana : Fruit
public class
...
```

Can create an array which holds derived types

```
Fruit badFruit = new Apple();

Fruit[] fruits = {
   new Apple(),
   new Banana(),
   new Grape()
}
```

Treating base types like derived types

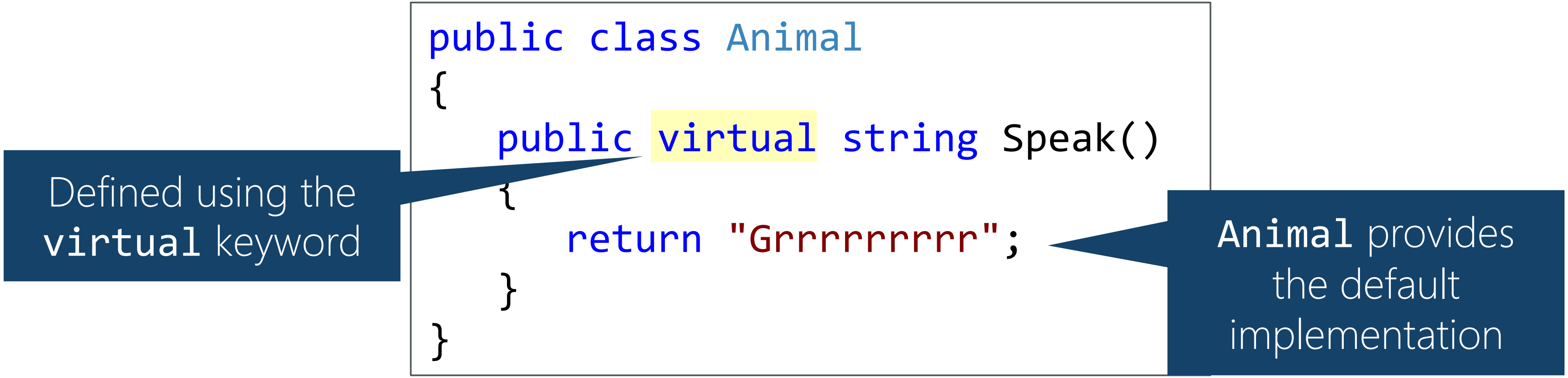
- Base types can **never** be used as a derived type, this is a compile time error

```
public class Fruit
public class Apple  : Fruit
public class Banana : Fruit
public class Grape  : Fruit
...
```

```
Apple badFruit = new Fruit(); X

Apple[] fruits = {
   new Apple(),    // ok
   new Banana(),   // error
   new Grape()     // error
}
```

What are virtual methods?

- Virtual methods are special methods declared on the base class which can be *replaced* with a different implementation by a derived class



```
public class Animal
{
   public virtual string Speak()
   {
      return "Grrrrrrrr";
   }
}
```

Defined using the **`virtual`** keyword

`Animal` provides the default implementation

Overriding a virtual method

- Derived classes can *override* a virtual method and provide a more specific or appropriate implementation

To override the method, you add an identical method to the derived class and use the `override` keyword

```
public class Dog : Animal
{
    public override string Speak()
    {
        return "WOOF!";
    }
}
```

Calling virtual methods

- When a virtual method is called, the most derived version of the method is called – even if the variable is assigned to a base class type

```
Animal fido = new Dog();
fido.Speak();         // WOOF!

Animal spot = new Pig();
spot.Speak();         // OINK!

Animal cuddles = new Bear();
cuddles.Speak();      // Grrrrrrrr
```

Bear didn't override **Speak**, base implementation called

Calling virtual methods

- Virtual methods are useful when collections of the base type hold derived objects

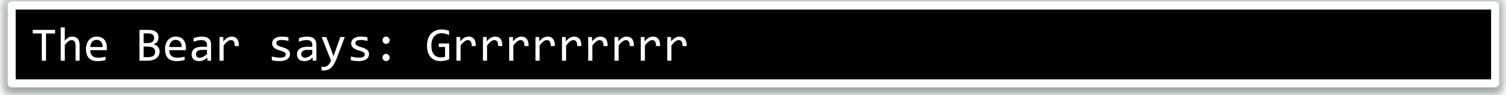
```
List<Animal> animals = new List<Animal> ( );
animals.Add( new Dog ( ) );
animals.Add( new Pig ( ) );
animals.Add( new Dog ( ) );
animals.Add( new Dog ( ) );

foreach ( Animal animal in animals ) {
   Console.WriteLine( animal.Speak( ) );
   // WOOF! Oink! WOOF! WOOF!
}
```

What if the method is *not* overridden?

- If a virtual method is called on a derived class, and that class *does not* override the method, then the **base class method is called**

```
public class Bear : Animal
{
    public bool IsHibernating;
}
```

```
Bear bear = new Bear();
Console.WriteLine("The Bear says: {0}", bear.Speak());
```



```
The Bear says: Grrrrrrrr
```

The base keyword

- Use the **base** keyword to access any method or property in the base class – this provides access to overridden virtual types

The **base** keyword calls the **Eat()** method in the base class that the Dog inherited from

```
public class Dog : Animal
{
   public override bool Eat(string food)
   {
      Console.WriteLine($"{food} eaten");
      return base.Eat(food);
   }
}
```

Flash Quiz

Xamarin University

Flash Quiz

① If Dog derives from Mammal, can a Dog object be in an array of Mammals?

a) Yes

b) No

Flash Quiz

① If Dog derives from Mammal, can a Dog object be in an array of Mammals?

a) Yes

b) No

Flash Quiz

② If Dog derives from Mammal, can a Mammal be in an array of Dogs?

a) Yes

b) No

Flash Quiz

② If Dog derives from Mammal, can a Mammal be in an array of Dogs?

a) Yes

b) [✓ No]

Flash Quiz

③ Polymorphism allows derived classes to override or replace the behavior of the base class

a) True

b) False

Flash Quiz

③ Polymorphism allows derived classes to override or replace the behavior of the base class

a) True

b) False

Forcing a method to be overridden

- Sometimes the base class might not have a reasonable implementation and *require* that the derived class implement it

```
public class Animal
{
    public virtual string Speak()
    {
        return //what should we return?
    }
}
```

Abstract methods

- In these cases, we can make the method *abstract* – in this case we provide **no implementation**

```
public class Animal
{
   public abstract string Speak();
}
```

Abstract methods are *always* virtual – they **must** be overridden by the derived class

Abstract classes

- When any method is abstract (not provided), then the class itself *must also* be marked as abstract – this indicates that the class is not complete

```
public abstract class Animal
{
    public abstract string Speak();
}
```

Implementing abstract classes

- Derived class must provide implementation of each abstract method – compiler will generate an error if any abstract methods are not supplied

```
public class Dog : Animal
{
    public override string Speak()     ✓
    {
        return "Woof!";
    }
}
```

Abstract class rules

- You cannot create an instance of an abstract class

```
~~Animal a1 = new Animal();~~    X
Animal a2 = new Dog();           ✓
Dog a3 = new Dog();              ✓
```

Individual Exercise

Create and override virtual methods

Summary

1. Create a virtual method
2. Override methods in derived class
3. Model abstract concepts using abstract classes

Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile

Microsoft