Games Technologies

Inheritance

Vassilis Markos, Mediterranean College

Week 03

Contents

Inheritance

On Animals

- Last time, we explored some simple Animal functionality, all wrapped in an `Animal` class.

On Animals

- Last time, we explored some simple Animal functionality, all wrapped in an `Animal` class.
    - As always, any code mentioned here lives in the `source` directory, in particular, in `Animal.cs`.

On Animals

- Last time, we explored some simple Animal functionality, all wrapped in an `Animal` class.
    - As always, any code mentioned here lives in the `source` directory, in particular, in `Animal.cs`.
- But, “animal” is a quite generic term, since various animals have different properties that distinguish them from others.

On Animals

- Last time, we explored some simple Animal functionality, all wrapped in an `Animal` class.
  - As always, any code mentioned here lives in the `source` directory, in particular, in `Animal.cs`.
- But, “animal” is a quite generic term, since various animals have different properties that distinguish them from others.
- For instance, many animals do “talk” in some sense, but, well not in the same way.

On Animals

- Last time, we explored some simple Animal functionality, all wrapped in an `Animal` class.
  - As always, any code mentioned here lives in the `source` directory, in particular, in `Animal.cs`.
- But, “animal” is a quite generic term, since various animals have different properties that distinguish them from others.
- For instance, many animals do “talk” in some sense, but, well not in the same way.
  - Cats “meow”, dogs “woof” and so on.

On Animals

- Last time, we explored some simple Animal functionality, all wrapped in an `Animal` class.
    - As always, any code mentioned here lives in the `source` directory, in particular, in `Animal.cs`.
- But, “animal” is a quite generic term, since various animals have different properties that distinguish them from others.
- For instance, many animals do “talk” in some sense, but, well not in the same way.
    - Cats “meow”, dogs “woof” and so on.
    - And some don’t talk.

On Animals

- Last time, we explored some simple Animal functionality, all wrapped in an `Animal` class.
    - As always, any code mentioned here lives in the `source` directory, in particular, in `Animal.cs`.
- But, “animal” is a quite generic term, since various animals have different properties that distinguish them from others.
- For instance, many animals do “talk” in some sense, but, well not in the same way.
    - Cats “meow”, dogs “woof” and so on.
    - And some don’t talk.
- And, not to be forgotten, all animals share some characteristics too!

On Classes

- So, we need a way to compactly express all of these things in the OOP paradigm.

On Classes

- So, we need a way to compactly express all of these things in the OOP paradigm.
- Evidently, a simple approach would be to define a single class for each animal we are interested in.

On Classes

- So, we need a way to compactly express all of these things in the OOP paradigm.
- Evidently, a simple approach would be to define a single class for each animal we are interested in.
- Can you see any problems here?

On Classes

- So, we need a way to compactly express all of these things in the OOP paradigm.
- Evidently, a simple approach would be to define a single class for each animal we are interested in.
- Can you see any problems here?
    - *Redundancy*, to begin with.

On Classes

- So, we need a way to compactly express all of these things in the OOP paradigm.
- Evidently, a simple approach would be to define a single class for each animal we are interested in.
- Can you see any problems here?
    - *Redundancy*, to begin with.
    - Lack of semantic mapping of those concepts in our design is another.

On Classes

- So, we need a way to compactly express all of these things in the OOP paradigm.
- Evidently, a simple approach would be to define a single class for each animal we are interested in.
- Can you see any problems here?
    - *Redundancy*, to begin with.
    - Lack of semantic mapping of those concepts in our design is another.
- In order to facilitate this sort of sharing some attributes / functionality but differentiating with respect to some other, OOP offers **Inheritance.**

Examples Of Inheritance

```
        /*
         * This is a brand new instance method that defines what it means
         * for an animal to "talk", by returning a generic string.
         * The virtual keyword, in practice, allows us to override this
         * method in child classes, if needed.
         */
        public virtual string Talk() {
            return "Talk!";
        }
```

Examples Of Inheritance

```csharp
using System;

namespace Animals {
    public class Cat : Animal {
        // All `Animal` fields are here, inherited from the `Animal` class.
        // NOTE: In C# we often prefer "derive" over "inherit" for inheritance.
```

Examples Of Inheritance

```
    // We define a `Cat` constructor by just calling the corresponding `Animal` one,
    // using the keyword `base`. In general, `base` is a pointer
    // to the parent class, while `this` is a pointer to the child class.
    public Cat(string name, int age) : base(name, age) {}

    // Same for the other Animal constructor.
    public Cat(string name, int age, double x, double y) : base(name, age, x, y) {}
```

Examples Of Inheritance

```
        // An example of overriding a parent method.
        public override string Talk() {
            return "Meow!";
        }
    }
}
```

- When it comes to overriding, we just need to define a child method with the same signature as the parent method.

Examples Of Inheritance

```
            // An example of overriding a parent method.
            public override string Talk() {
                return "Meow!";
            }
        }
}
```

- When it comes to overriding, we just need to define a child method with the same signature as the parent method.
- The parent method has to be declared as `virtual`.

Examples Of Inheritance

```
        // An example of overriding a parent method.
        public override string Talk() {
            return "Meow!";
        }
    }
}
```

- When it comes to overriding, we just need to define a child method with the same signature as the parent method.
- The parent method has to be declared as `virtual`.
- The child method has to be declared as `override`.

A Simple Test Class

```
using System;

namespace Animals {
    class Test {
        public static void Main(string[] args) {
            Animal alice = new Animal("Alice", 8, 0.0, 1.0);
            Cat bob = new Cat("Bob", 7);
            // Animal charlie = new Animal();
            Console.WriteLine("{0}\n{1}", alice, bob);
            Console.WriteLine("Alice says: {0}\nBob says: {1}"
    , alice.Talk(), bob.Talk());
        }
    }
}
```

Stranger Things

```
using System;

namespace Animals {
    class StrangeTest {
        public static void Main(string[] args) {
            Cat alice = new Cat("Alice", 8);
            Animal bob = new Cat("Bob", 7);
            // What will this print?
            Console.WriteLine("{0}\n{1}", alice.Talk(), bob.
    Talk());
        }
    }
}
```

Types And Inheritance

- We can construct an object using a constructor of a child class and then assign it to a variable of a parent class.

Types And Inheritance

- We can construct an object using a constructor of a child class and then assign it to a variable of a parent class.
- This is perfectly okay, and in many cases useful (examples to come soon).

Types And Inheritance

- We can construct an object using a constructor of a child class and then assign it to a variable of a parent class.
- This is perfectly okay, and in many cases useful (examples to come soon).
- However, since the object has been constructed with some child constructor, it is considered an object of type `child` and not `parent`.

Types And Inheritance

- We can construct an object using a constructor of a child class and then assign it to a variable of a parent class.
- This is perfectly okay, and in many cases useful (examples to come soon).
- However, since the object has been constructed with some child constructor, it is considered an object of type `child` and not `parent`.
- Thus, above, in both cases, the two instances use the overriden version of the `Talk()` method.

Dogs

```
using System;

namespace Animals {
    public class Dog : Animal {
        public Dog(string name, int age) : base(name, age) {}
        public Dog(string name, int age, double x, double y) :
    base(name, age, x, y) {}

        public override string Talk() {
            return "Woof!";
        }
    }
}
```

Animal Structures

```csharp
using System;
using System.Collections.Generic;

namespace Animals {
    public class AnimalStructures {
        public static void Main(string[] args) {
            Animal alice = new Animal("Alice", 8);
            Cat bob = new Cat("Bob", 7);
            Dog charlie = new Dog("Charlie", 6);
            List<Animal> animals = new List<Animal> {alice, bob, charlie};
            animals.ForEach(Console.WriteLine);
        }
    }
}
```

Types And Inheritance (Again)

- Using inheritance one can simulate a higher level behaviour found, e.g., in Python, where a data structure can contain elements of different type.

Types And Inheritance (Again)

- Using inheritance one can simulate a higher level behaviour found, e.g., in Python, where a data structure can contain elements of different type.
- This is naturally restricted to elements of some common ancestor type, in our case `Animal`.

Types And Inheritance (Again)

- Using inheritance one can simulate a higher level behaviour found, e.g., in Python, where a data structure can contain elements of different type.
- This is naturally restricted to elements of some common ancestor type, in our case `Animal`.
- However, this is enough in most cases to allow for significant code simplifications, while still maintaining the advantages of strong typing.

Types And Inheritance (Again)

- Using inheritance one can simulate a higher level behaviour found, e.g., in Python, where a data structure can contain elements of different type.
- This is naturally restricted to elements of some common ancestor type, in our case `Animal`.
- However, this is enough in most cases to allow for significant code simplifications, while still maintaining the advantages of strong typing.
- This is also useful in function / method signatures, in case one needs to handle multiple child types in a uniform way.

Fun Time!

In–class Exercise #001

Follow Lab instructions in

```
lab/Game_Lab_01.pdf
```

Use any online resources you might find useful.

In–class Exercise #002

Follow Lab instructions in

```
lab/Game_Lab_02.pdf
```

Use any online resources you might find useful.

Homework

Complete any incomplete lab exercises and then proceed to complete any missing parts of the game lab discussed in class today.

Any Questions?

Do not forget to fill in the questionnaire shown right!

https://forms.gle/dKSrmE1VRVWqxBGZA