Agenda

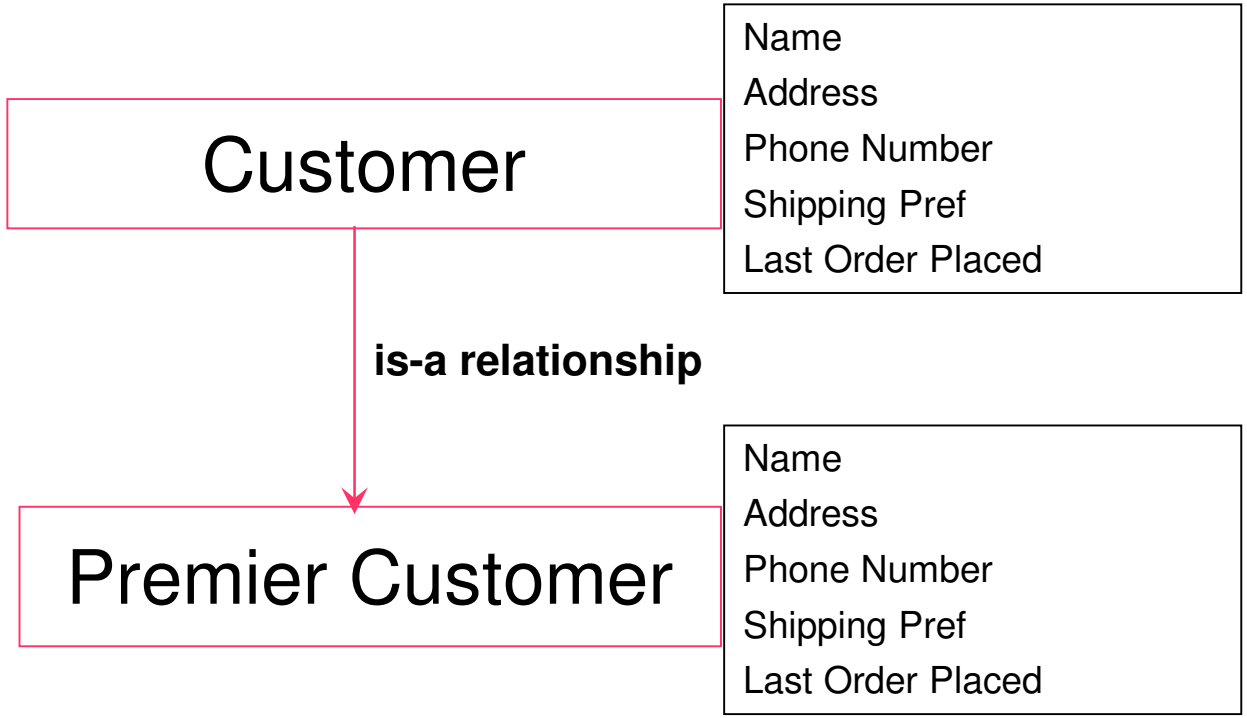
- Defining Inheritance
- Relationships of Inheritance
- Types of Inheritance
- Rules of Inheritance
- this and base References
- Constructor Chaining

Defining Inheritance

- A class that is derived from another class (a base class) automatically contains all the public, protected, and internal members of the base class.
- Inheritance enables you to create a new class that can reuse, extend, and modify the behavior that is defined in the other class.
- The class whose members are inherited is called the base class and the class that inherits those members is called the derived class.
- A derived class can have only one direct base class.
- Inheritance is transitive.
- Conceptually a derived class is a specialization of the base class.

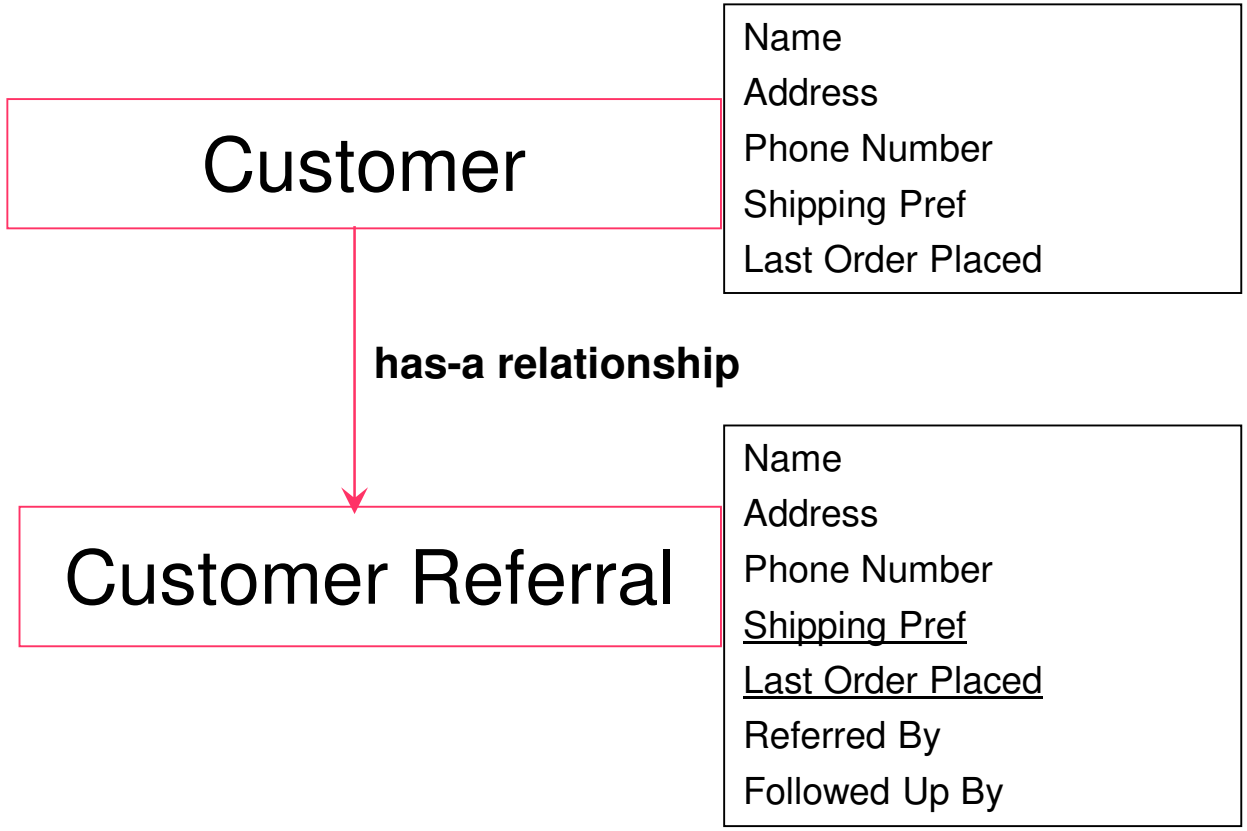
Relationships of Inheritance

- "is-a" relationship in inheritance:
  - Derived class can be used wherever a base class can be used.
  - Is implemented in C# by extending a class.

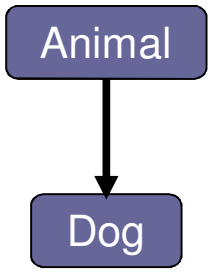


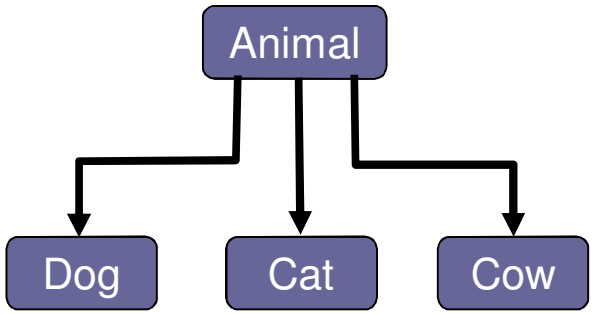
Relationships of Inheritance (cont.)

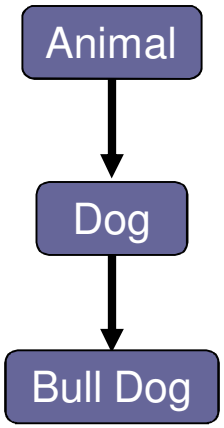
- "has-a" relationship in inheritance:
  - Whole-class relationship between a class and its parts.
  - Implemented in C# by instantiating an object inside a class.



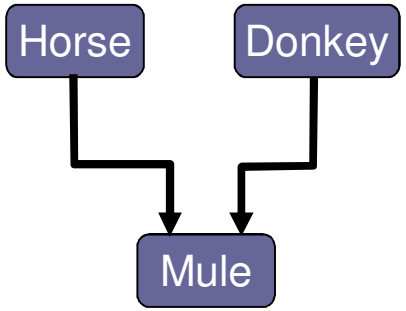
Types of Inheritance







Single Inheritance



Multiple Inheritance

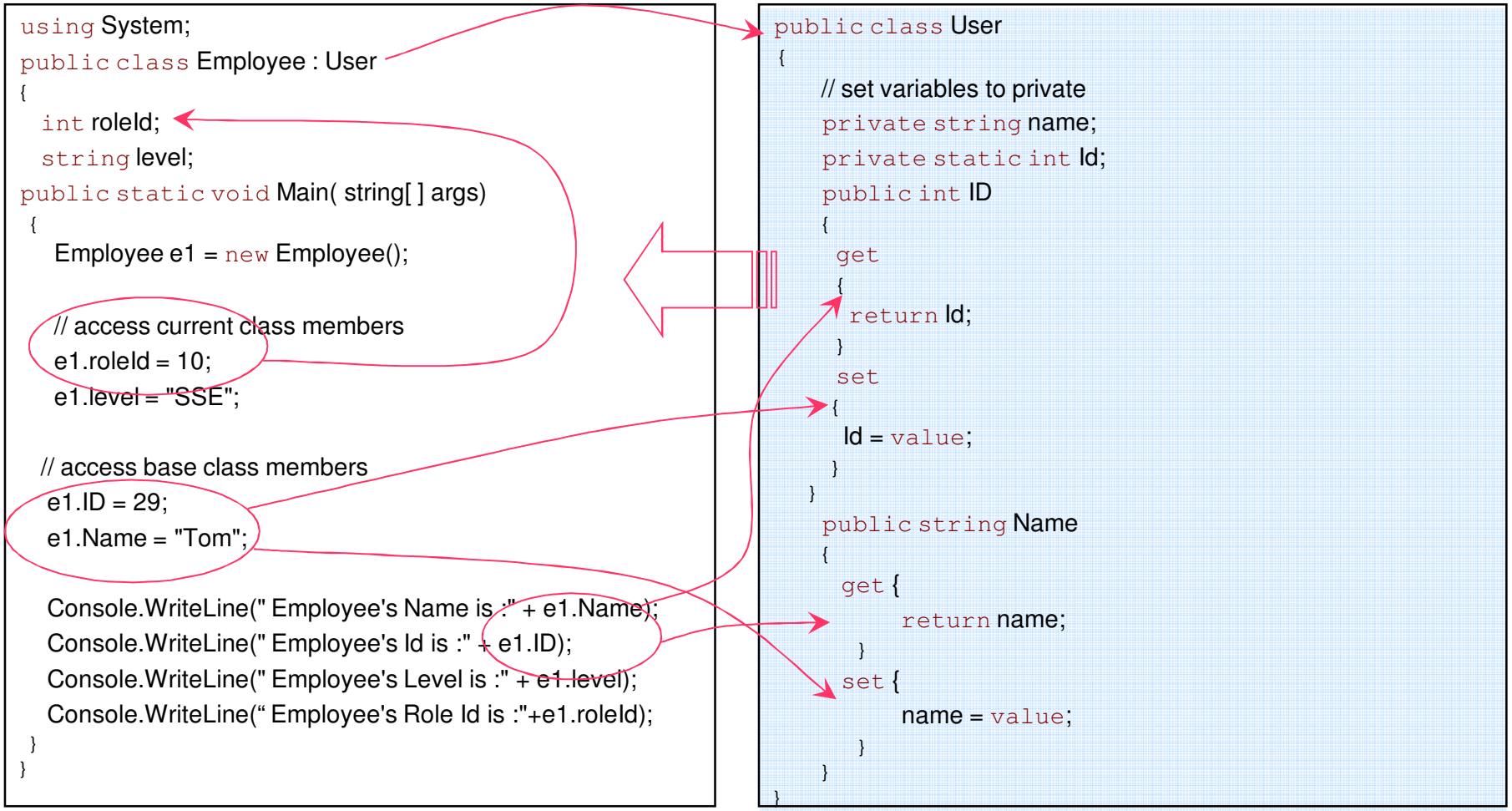
Note: C# does not directly support multiple inheritance. This concept is implemented using interfaces.

Rules of Inheritance

- A class can only inherit from one class (known as single inheritance).
- A subclass is guaranteed to do everything the base class can do.
- A subclass inherits members from its base class and can modify or add to its behavior and properties.
- A subclass can define members of the same name in the base class, thus hiding the base class members.
- Inheritance is transitive (i.e., class A inherits from class B, including what B inherited from class C).
- All classes inherit from the highest Object class in the inheritance hierarchy.
- private members and constructors are not inherited by subclasses.

Implementing Inheritance

- Inheritance is implemented in C# by the **:** operator



```
using System;
public class Employee : User
{
  int roleId;
  string level;
public static void Main( string[ ] args)
 {
    Employee e1 = new Employee();

   // access current class members
    e1.roleId = 10;
    e1.level = "SSE";

  // access base class members
   e1.ID = 29;
   e1.Name = "Tom";

   Console.WriteLine(" Employee's Name is :" + e1.Name);
   Console.WriteLine(" Employee's Id is :" + e1.ID);
   Console.WriteLine(" Employee's Level is :" + e1.level);
   Console.WriteLine(" Employee's Role Id is :"+e1.roleId);
 }
}
```

```
public class User
{
    // set variables to private
    private string name;
    private static int Id;
    public int ID
    {
      get
      {
       return Id;
      }
      set
      {
        Id = value;
      }
    }
    public string Name
    {
      get {
            return name;
        }
      set {
            name = value;
        }
    }
}
```

this and *base*

- `this` is a reference to the current class.
  - It qualifies the members hidden by similar names.
  - It is used to pass as parameter to the other methods.
  - It is an error to refer to “this” keyword in a static method.

- `base` is a reference to the base class.
  - It is used to access members of the base class from within the derived class.
  - It can call a method on the base class that has been overridden by another method.
  - It is an error to use the base keyword from within a static method.

Using *this* and *base*

```
using System;
class Employee: User
{
    private string username;
    private double grade;
    // Setting Accessors
    public string UserName
     {
     get{return username;}
      set{this.username = value;}
     }
    public double Grade
      {
       get{return grade;}
       set{this.grade = value;}
      }
    // access base class members
    public void printInfo()
     {
     Console.WriteLine(base.Name);
     Console.WriteLine(base.Id);
     Console.WriteLine(this.username);
     Console.WriteLine(this.grade);
    }
}
```

```
class User {
    // set variables to private
    private string name;
    private int id;

    /*
     * setters & getters, set to public
     */
    public int Id
     {
      get
      {
        return id;
      }
     set
      {
       id = value;
      }
     }
    public string Name
     {
      get
      {
      return name;
     }
     set
      {
        name = value;
     }
    }
}
```

Constructor Chaining

- *Constructor chaining* is invoking all the constructors in the inheritance hierarchy.
- Constructor chaining guarantees that all base class constructors are called.
- Constructors are invoked starting with the current class up to the `Object` class, then they are executed in reverse order.

Implementing Constructor Chaining

```
class User
 {
   private string name;
   public User( string name) //overloaded base class constructor (non-default)
   {
     this.name = name;
    }
 }
 class Employee : User
 {
   string username;

    //sub class constructor
   public Employee (string name, string username) : base (name)
    //Calling base constructor
    {
     this.username = username;
    }
   public static void Main(string[] args)
   {
    Employee e1 = new Employee("Andrew","ACA.NET");
   }
  }
```