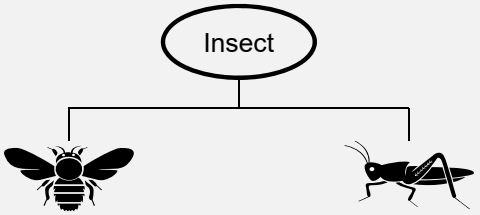
10.1 Inheritance

- **Inheritance** allows a new class to extend an existing class
  - The new class inherits the members of the class it extends
  - It helps to **specialize** a class from an existing class
    - *bumblebees* class and *grasshoppers* class are two specialized classes of the *insect* class
  - It also helps to **generalize** common members of many classes into a new class
    - *Sedan*, *pickup*, and *SUV* have common characteristics of a *car*

AUBURN UNIVERSITY

"Is a" Relationship

- When one object is a specialized version of another object, there is an **"is a" relationship** between them
  - A grasshopper **is an** insect
  - A bumblebee **is an** insect



- The logic is:
  - All insects have certain common characteristics and can be described by an **Insect** class
  - The grasshopper has its own unique characteristics to be described by a **Grasshopper** class
  - The bumblebee has its own unique characteristics to be described by a **Bumblebee** class

AUBURN UNIVERSITY

"Is a" Relationship (Cont'd)

- An "is a" relationship implies that the specialized object has all the characteristics of the generalized object
  - The specialized object has additional characteristics that make it special, which the generalized object does not have
- In object-oriented programming, inheritance creates an "is a" relationship among classes when you declare a class to be a specialized class of another

  specialized "is a" *generalized*

- This allows you to extend the capabilities of a class by creating another class that is a specialized version of it

Base and Derived Classes

- Inheritance involves **base** and **derived** classes
  - The base class is the generalized class and is sometimes called **superclass**
  - The derived class is the specialized class and is sometimes called **subclass**
  - You can think of the derived class as an extended version of the base class
  - The derived class inherits fields, properties, and methods from the base class without any of them having to be rewritten
  - New fields, properties, and methods may be added to the derived class to make it special

Inheritance Notation

- Assuming there exists an `Automobile` class:

```
class Automobile
{
  Members….
}
```

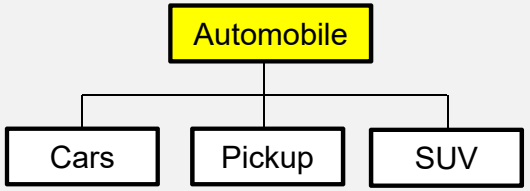
- In C# the generic format to declare inheritance in the class header

```
class Car : Automobile
{ }
```

  - where `Car` is the derived class and `Automobile` is the base class
  - the colon (`:`) indicates that this class is derived from another class

Examples

- A dealership's inventory includes three types of automobiles: cars, pickup trucks, and SUVs
- The dealership keeps the following data:
  - Make
  - Year model
  - Mileage
  - Price



- Each type of vehicle that is kept in inventory has the above general data
- Each type of vehicle also has its own specialized data as shown below:

Items	Cars	Pickups	SUVs
Specialized data	Number of doors	Drive type	Passenger capacity

AUBURN UNIVERSITY

Sample Code

```
// base

class Automobile
{
 // fields
 private string _make;
 private string _model;
 private int _mileage;
 private decimal _price;

 // parameterless constructor
 public Automobile() { … }

 // properties
 public string Make { … }
 public string Model { … }
 public int Mileage { … }
 public decimal Price { … }
}
```

```
// derived
class Car : Automobile
{
  // field
  private int _doors;

  // parameterless constructor
  public Car()
  {
    _door = 0;
  }
  public int Doors
  {
     get { return _doors; }
     set { _doors = value; }
  }
}
```

```
// instantiation

Car myCar = new Car();
myCar.Make = "Ford";
myCar.Model = "Echo";
myCar.Mileage = 56781;
myCar.Price = 7010m;
```

AUBURN UNIVERSITY

Base Class and Derived Class Constructors

- When you create an instance of a derived class,
  - the base class constructor is executed first
  - the derived class constructor next
  - by default the base class' parameterless constructor is automatically executed

```
// base
class Rectangle
{
  …
  // parameterless constructor
  public Rectangle() {  …  }

  // parameterized constructor
  public Rectangle(int length, int width)
  {  …  }
 …
}
```

inheritance

```
// derived
class Box : Rectangle
{
  private int _height;
  // parameter constructor
  public Box() { … }

  // parameterized constructor
  public Box(int length, int width, int height)
          : base(length, width)
  { … }
 …
}
```

Constructor Issues in Inheritance

- If you want a parameterized constructor in the base class to execute, or
- if the base class does not have a parameterless constructor,
  - you must explicitly call the base class' parameterized constructor using the **base** keyword

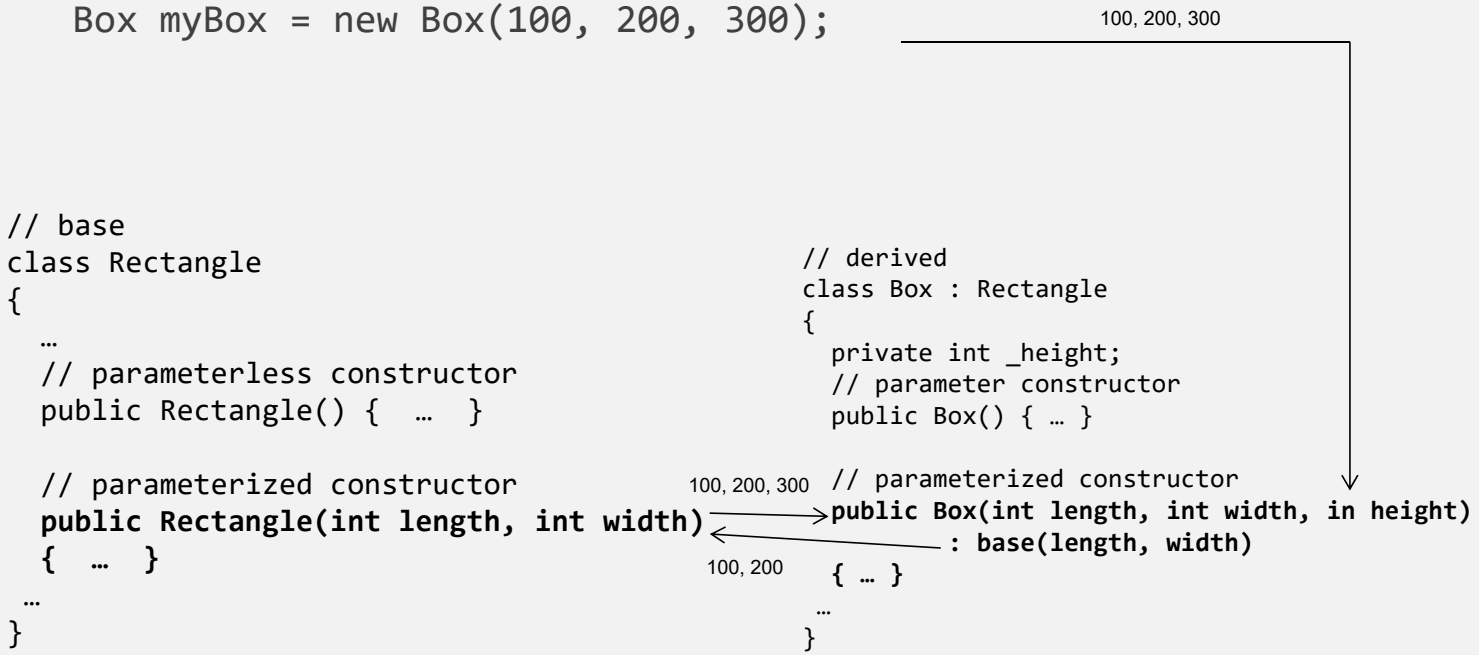
```
public Box(int length, int width, int height)
          : base(length, width)
{ … }
```

  - The above example calls the base class' parameterized constructor, passing *length* and *width* as arguments

Constructor Issues in Inheritance (Cont'd)

- Given the following statement,

```
Box myBox = new Box(100, 200, 300);
```




```
// base
class Rectangle
{
  …
  // parameterless constructor
  public Rectangle() {  …  }

  // parameterized constructor
  public Rectangle(int length, int width)
  {  …  }
 …
}
```

```
// derived
class Box : Rectangle
{
  private int _height;
  // parameter constructor
  public Box() { … }

  // parameterized constructor
  public Box(int length, int width, in height)
    : base(length, width)
  { … }
 …
}
```

10.2 Polymorphism

- The term **polymorphism** refers to an object's ability to take different forms
  - It allows derived classes to have methods with the same names as methods in their base classes
  - It gives the ability for a program to call the correct method, depending on the type of object that is used to call it
- When a derived class inherits from a base class, it gains all the methods, fields, properties and events of the base class
  - To change the data and behavior of a base class, you have an option to override a virtual base member

Essential Ingredients of Polymorphism

- The textbook identifies two essential ingredients of polymorphic behavior:
  - The ability to define a method in a base class and then define a method with the same name in a derived class
    - The derived class **overrides** the base class method
  - The ability to call the correct version of an overridden method, depending on the type of object that is used to call it
    - If a derived class object is used to call an overridden method, then the derived class's version is the one that executes
    - If a base class object is used to call an overridden method, then the base class' version is the one that executes

Polymorphism (Cont'd)

- The keyword **`virtual`** is used to declare that a derived class is allowed to override a method of a base class

```
class Animal // base class
{
 private string _species; // field
 public Animal(string species) { _species = species; } // constructor
 public string Species { … } // property
 public virtual void MakeSound() { … } // allow derived class to
override
}
```

- To create an `Animal` object, your only option is:

```
Animal myAnimal = new myAnimal("regular animal");
```

Polymorphism (Cont'd)

- The keyword **override** declares that this method overrides a method in the base class

```
class Dog : Animal
{
 private string _name; // field
 public Dog(string name) : base ("Dog") {  _name = name; } //
constructor
 public string Name { … } // property
 public override void MakeSound() { … }
}
```

- To create an `Animal` object, your options are:

```
Dog myDog = new Dog("Fido");
Animal myAnimal = new Dog("Fido"); // an Dog object is also an Animal object
```

Overriding Properties

- Properties in a base class can be overridden in the same way that methods can be overridden

```
public virtual double Weight
{
  get { return _weight; }
  set { _weight = value; }
}
```

- To override the property in the derived class you use the **`override`** keyword

```
public override double Weight
{
  get { return _weight * 0.165; }
  set { _weight = value; }
}
```

Passing Objects to Base Class Parameters

- Given the following method of a derived class,

```
private void ShowAnimalInfo(Animal animal)
{
  MessageBox.Show("Species: " + animal.Species);
  animal.MakeSound();
}
```

  - This method has an `Animal` variable as its parameter. You can pass an `Animal` object to the method

```
Animal myAnimal = new Animal("Regular animal");
ShowAnimalInfo(myAnimal);
```

  - The method can display the object's Species property and calls its `MakeSound` method

- Due to polymorphism, you can also pass a `Dog` object as argument to the `ShowAnimalInfo` method

```
Dog myDog = new Dog("Fido");
ShowAnimalInfo(myAnimal);
```

Base Class Reference

- A **base class reference variable** can reference an object of any class that is derived from the base class
- A base class reference variable knows only about the members that are declared in the base class

```
class Animal // base class
{
 private string _species;
 …
 public string Species { }
}
```

```
class Dog : Animal // derived class
{
  private string _name;
  ….
  public string Name { }
}
```

- If the derived class introduces additional methods, properties, or fields, a base class reference variable cannot access them

```
Animal myAnimal = new Dog("Fido");
MessageBox.Show("The species is " + myAnimal.Species);
MessageBox.Show("The animal's name is " + myAnimal.Name); // ERROR!
```

The "Is a" Relationship Does Not Work in Reverse

- It is important to understand that the "is a" relationship does not work in reverse
- A dog is an animal. Yet, "an animal is a dog" is not always true.

```
Dog myDog = new Animal("Dog"); // will not compile
```

- You cannot assign an `Animal` reference to a `Dog` variable because `Dog` is a derived class

```
class Animal // base class
{ … }

class Dog : Animal // derived class
{ … }
```

AUBURN UNIVERSITY

Summary of Polymorphism Issues

- In order for an instance of a derived class to completely take over a class member from a base class, the base class has to declare that member as **`virtual`**
  - Fields cannot be virtual
  - Only methods, properties, events and indexers can be `virtual`
- A derived class then has the option of using the **`override`** keyword to replace the base class implementation with its own

10.3 Abstract Classes

- In certain applications, some base classes are not intended to be instantiated to create objects
  - They are designed solely for the purpose of providing an outline for subclasses. For example,
    - a Student class that describes what is common to all students, but does not provide details for students majoring in Computer Science
    - The Student class is intended to be a base class that can be derived by the Computer Science class
- An **abstract class** serves as a base class but is not instantiated itself
  - It only provides some class members to its derived classes

Abstract Classes (Cont'd)

- To declare a class as abstract, use the **`abstract`** keyword in the class header. For example,

```
abstract class Person
{ … }
```

- The `abstract` keyword indicates that a class is intended only to be a base class of other classes
  - The primary differences between an abstract class and a regular class (aka **concrete class**) is:
- An abstract class cannot be instantiated
- A concrete class can be instantiated

Members of Abstract Classes

- An abstract class can have abstract and concrete members. For example,

```
abstract class Student
{
    private string _name; // concrete
    public Student(string name) { _name = name; } // concrete
    public string Name { get { return _name; } // concrete
    public abstract double Required Hours { get; } // abstract
}
```

Abstract Methods

- Abstract classes can contain **abstract methods**
  - An abstract method has only a header and no body
    - It must be overridden in a derived class
  - To create an abstract method, use the **`abstract`** keyword before the return type

```
abstract class Person
{
    public abstract void DoSomething(); // no method body
}
```

Abstract Properties

- Abstract classes can also contain **abstract properties**
  - An abstract property is a property that appears in a base class
  - Abstract properties are expected to be overridden in a derived class
- To create an abstract property, use the `abstract` keyword before the property type

```
abstract class Person
{
   public abstract string JobTitle // abstract property
   {
     get; // abstract get accessor
     set; // abstract set accessor
   }
}
```

- To create an abstract read-only property, leave out the `set` accessor