Object Oriented programming Part II

Class Relationships

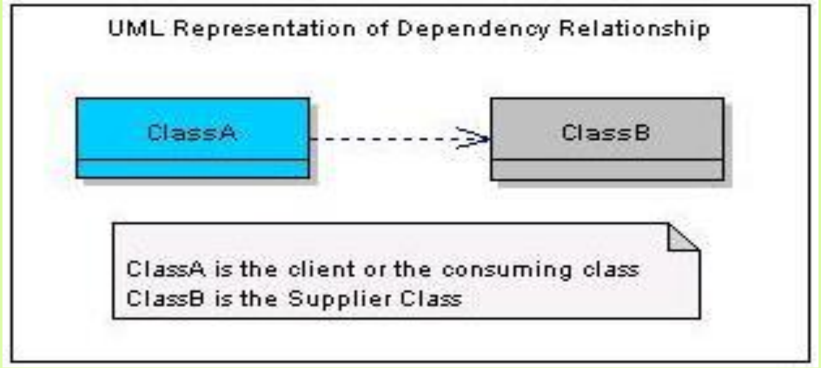
Outline

- Class Relationships
- Creating Subclasses
- Overriding Methods
- Class Hierarchies
- Abstract Classes
- Inheritance and Visibility
- Designing for Inheritance

Class Relationships

- Classes in a software system can have various types of relationships to each other
- Three of the most common relationships:
  - Dependency: A *uses* B
  - Aggregation and Composition: A *has-a* B
  - Inheritance: A *is-a* B
- Let's discuss dependency and aggregation further

Dependency


UML Representation of Dependency Relationship
ClassA
ClassB
ClassA is the client or the consuming class
ClassB is the Supplier Class

- A *dependency* exists when one class relies on another in some way, usually by invoking the methods of the other
- Sometimes the relationship between a two classes is very weak. They are not implemented with member variables at all. Rather they might be implemented as member function arguments.
- Dependency exists if a class is a parameter variable or local variable of a method of another class.
- This drives home the idea that the service is being requested from a particular object
- We've seen dependencies in many previous examples

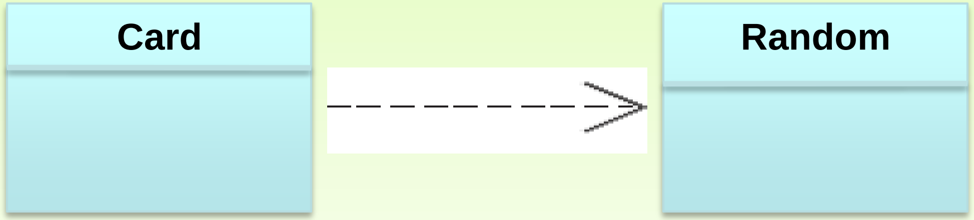
Dependency cont…

- Dependency relationships are inexpensive from resource consumption viewpoint and are transient i.e. their life is for a limited duration.
- There are two ways how a client can make a reference to a supplier class to form dependency.
  - As a Parameter: When the supplier class is used as a parameter in an operation or as a return type of an operation in the client or the consuming class.
  - As a Local Variable: When the supplier instance is created locally in the body of an operation of the consuming class.

Dependency cont…

```
Class Card
{               .
                .
                .

  public void GenerateRan(Random rnd)
   {

     int card = rnd.Next(52); // creates a number between 0 and 51    }
 }
```

When the supplier class is used as a parameter



```
Class Card
{               .
                .
                .

  public void GenerateRan()
   {
     Random rnd = new Random();
     int card = rnd.Next(52); // creates a number between 0 and 51
 }
 }
```

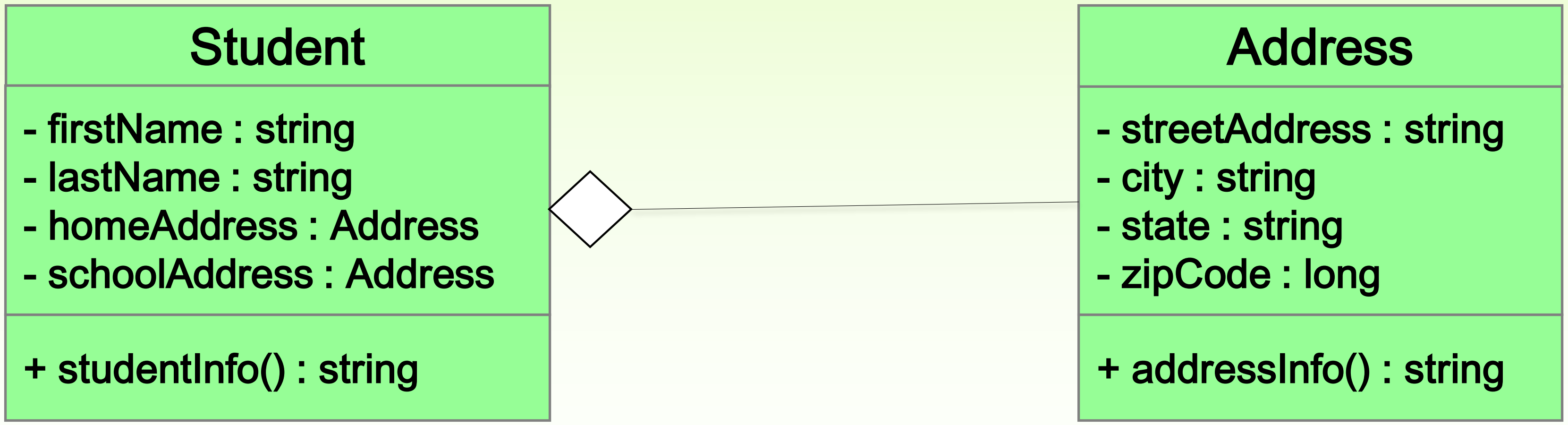
When the supplier class is used as Local reference

Aggregation

- An *aggregate* is an object that is made up of other objects
- Therefore aggregation is a *has-a* relationship
  - A car *has a* chassis
- In software, an aggregate object contains references to other objects as instance data
- This is a special kind of dependency; the aggregate relies on the objects that compose it

Aggregation in UML

- In the following example, a `Student` object is composed, in part, of `Address` objects
- A student has an address (in fact each student has two addresses)



```
//********************************************************************
//  StudentBody.cs       Author: Lewis/Loftus
//
//  Demonstrates the use of an aggregate class.
//********************************************************************

public class StudentBody
{
   //-----------------------------------------------------------------
   //  Creates some Address and Student objects and prints them.
   //-----------------------------------------------------------------
   public static void Main (string[] args)
   {
      Address school = new Address ("800 Lancaster Ave.", "Villanova",
                                    "PA", 19085);
      Address jHome = new Address ("21 Jump Street", "Lynchburg",
                                   "VA", 24551);
      Student john = new Student ("John", "Smith", jHome, school);

      Address mHome = new Address ("123 Main Street", "Euclid", "OH",
                                   44132);
      Student marsha = new Student ("Marsha", "Jones", mHome, school);

      Console.WriteLine (john.studentInfo());
      Console.WriteLine ();
      Console.WriteLine (marsha.studentInfo());
   }
}
```

```
//******************                          **********************
//  StudentBody.cs
//
//  Demonstrates the
//******************                          **********************

public class StudentB
{
   //----------------                          ----------------------
   //  Creates some A                          and prints them.
   //----------------                          ----------------------
   public static void
   {
      Address school                           er Ave.", "Villanova",
                                               ;
      Address jHome =                          et", "Lynchburg",

      Student john =                           ", jHome, school);

      Address mHome =                          eet", "Euclid", "OH",

      Student marsha = new Student ("Marsha", "Jones", mHome, school);

      Console.WriteLine (john.studentInfo());
      Console.WriteLine ();
      Console.WriteLine (marsha.studentInfo());
   }
}
```

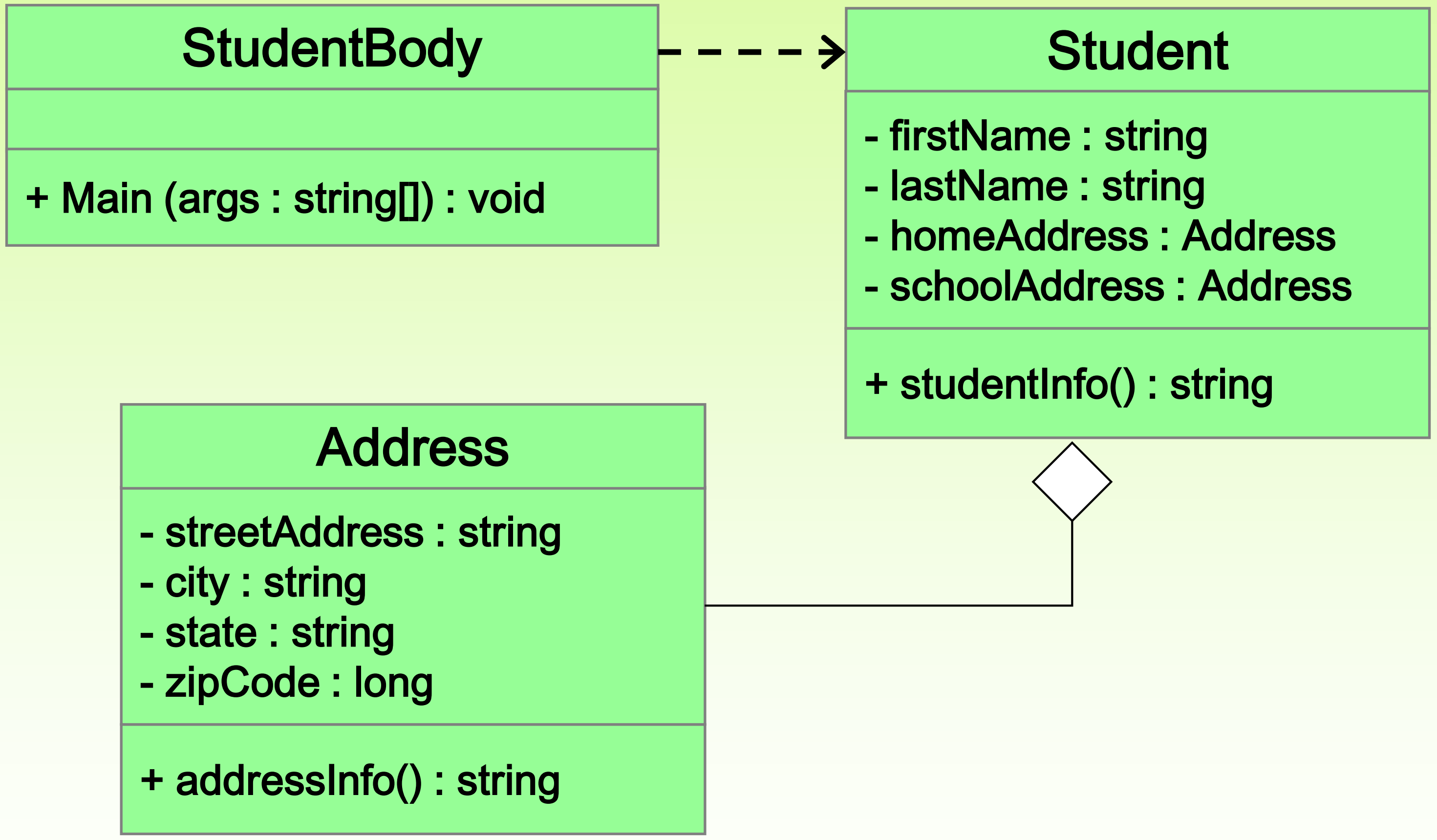
Output

```
John Smith
Home Address:
21 Jump Street
Lynchburg, VA  24551
School Address:
800 Lancaster Ave.
Villanova, PA  19085

Marsha Jones
Home Address:
123 Main Street
Euclid, OH  44132
School Address:
800 Lancaster Ave.
Villanova, PA  19085
```

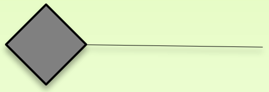
```
//*********************************************************************
//  Student.cs       Author: Lewis/Loftus
//
//  Represents a college student.
//*********************************************************************

public class Student
{
   private string firstName, lastName;
   private Address homeAddress, schoolAddress;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this student with the specified values.
   //-----------------------------------------------------------------
   public Student (string first, string last, Address home,
                   Address school)
   {
      firstName = first;
      lastName = last;
      homeAddress = home;
      schoolAddress = school;
   }

continue
continue

   //-------------------------------------------------------------------
   //  Returns a string description of this Student object.
   //-------------------------------------------------------------------
   public string studentInfo()
   {
      string result;

      result = firstName + "  " + lastName + "\n";
      result += "Home Address:\n" + homeAddress.addressInfo() + "\n";
      result += "School Address:\n" + schoolAddress.addressInfo();

      return result;
   }
}
```

```
//********************************************************************
//  Address.cs       Author: Lewis/Loftus
//
//  Represents a street address.
//********************************************************************

public class Address
{
   private string streetAddress, city, state;
   private long zipCode;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this address with the specified data.
   //-----------------------------------------------------------------
   public Address (string street, string town, string st, long zip)
   {
      streetAddress = street;
      city = town;
      state = st;
      zipCode = zip;
   }

continue
continue

   //-----------------------------------------------------------------
   //  Returns a description of this Address object.
   //-----------------------------------------------------------------
   public string AddressInfo()
   {
      string result;

      result = streetAddress + "\n";
      result += city + ", " + state + "  " + zipCode;

      return result;
   }
}
```

Aggregation in UML



Aggregation v.s. Composition

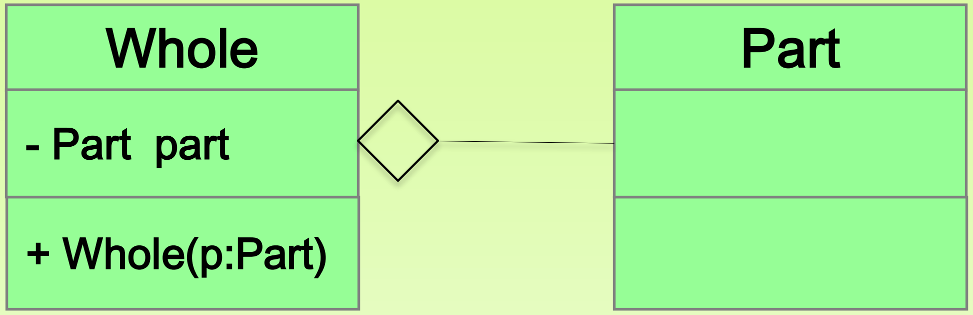
- A composition relationship, also known as a composite aggregation, is a stronger form of aggregation where the part is created and destroyed with the whole. A composition relationship is indicated... 
- An example of composition is a House object contains zero or more Room objects. The state of each Room object has an influence on the House object. If the House object is destroyed, the Room objects will also be destroyed.

Aggregation v.s. Composition

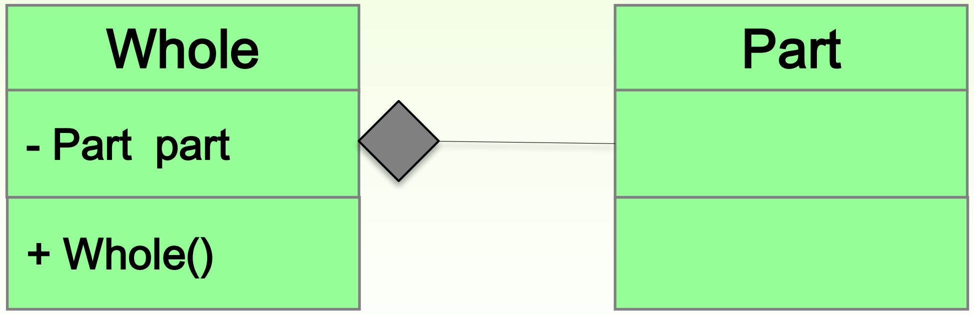
- In an aggregation relationship, the part may be independent of the whole but the whole requires the part hence the part can belong to more than one whole
- An example of aggregation is a History-Class object contains zero or more of Student objects. The state of each Student object has an influence on the state of the History-Class object. If the History-Class object is destroyed, the Student objects may continue to exist.
- An aggregation relationship is indicated in the UML with an unfilled diamond and a line

Aggregation v.s. Composition

```
Public class Whole {
Private Part  part
public Whole (Part p)
{
   part = p;
}}
```



```
Public class Whole {
Private Part  part
public Whole ()
{
   part = new Part();
}}
```



Outline

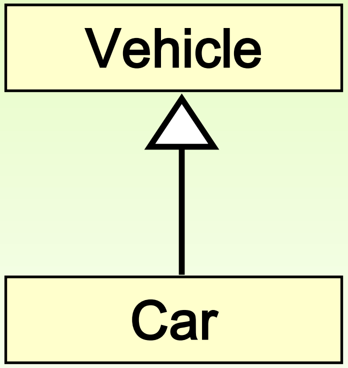
- Class Relationships
- ➡ Creating Subclasses
- Overriding Methods
- Class Hierarchies
- Abstract Classes
- Inheritance and Visibility
- Designing for Inheritance

Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined by the parent class

Inheritance

- Inheritance relationships are shown in a UML class diagram using a solid arrow with an unfilled triangular arrowhead pointing to the parent class



- **Proper inheritance creates an *is-a* relationship, meaning the child *is a* more specific version of the parent**

Inheritance

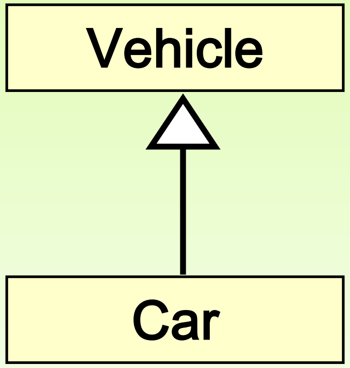
- A programmer can tailor a derived class as needed by adding new variables or methods, or by modifying the inherited ones
- One benefit of inheritance is *software reuse*
- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

Deriving Subclasses

- In C#, we use the reserved word `:` to establish an inheritance relationship

```
public class Car : Vehicle
{
   // class contents
}
```

UML diagram



- **See Words.cs**
- **See Book.cs**
- **See Dictionary.cs**

The protected Modifier

- Visibility modifiers affect the way that class members can be used in a child class
- Variables and methods declared with private visibility cannot be referenced in a child class
- They can be referenced in the child class if they are declared with public visibility -- but public variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

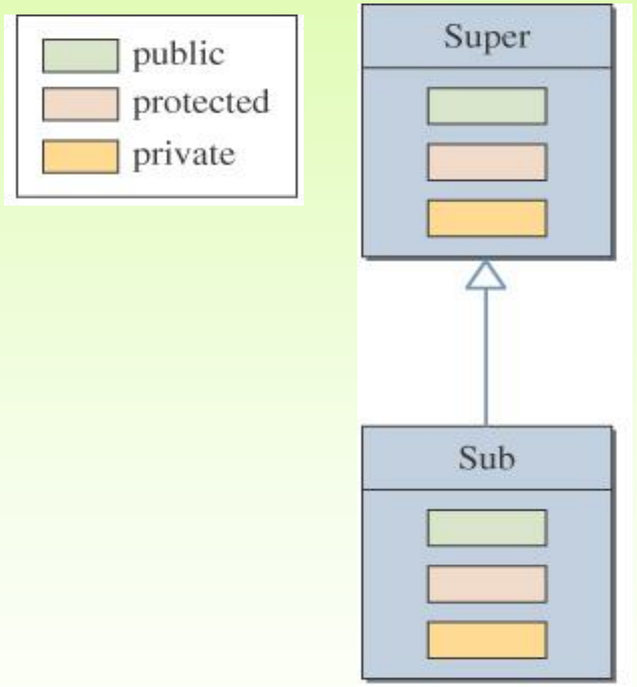
The protected Modifier

- The `protected` modifier allows a child class to reference a variable or method in the child class
- It provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility
- A protected variable is also visible to any class in the same Namespace as the parent class
- Protected variables and methods can be shown with a # symbol preceding them in UML diagrams

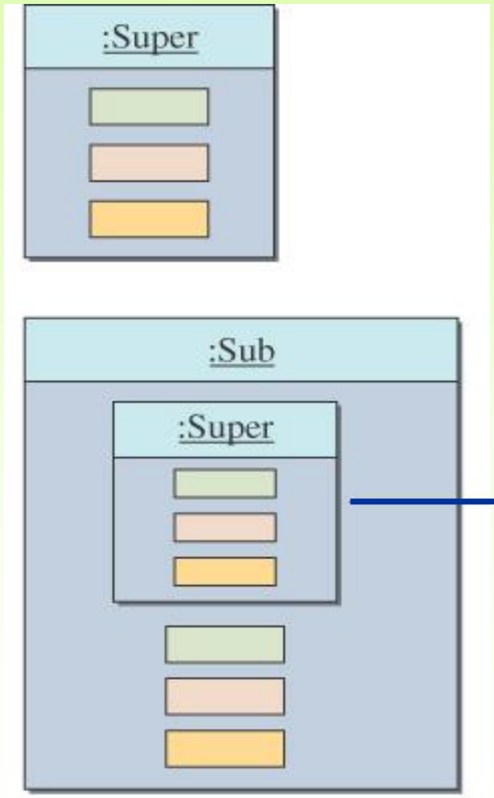
visibility / keyword	Containing Classes	Derived Classes	Containing Assembly	Anywhere outside the containing assembly
public	yes	yes	yes	yes
protected internal	yes	yes	yes	no
protected	yes	yes	no	no
private	yes	no	no	no
internal	yes	no	yes	no

Inheritance and Member Accessibility

- We use the following visual representation of inheritance to illustrate data member accessibility.



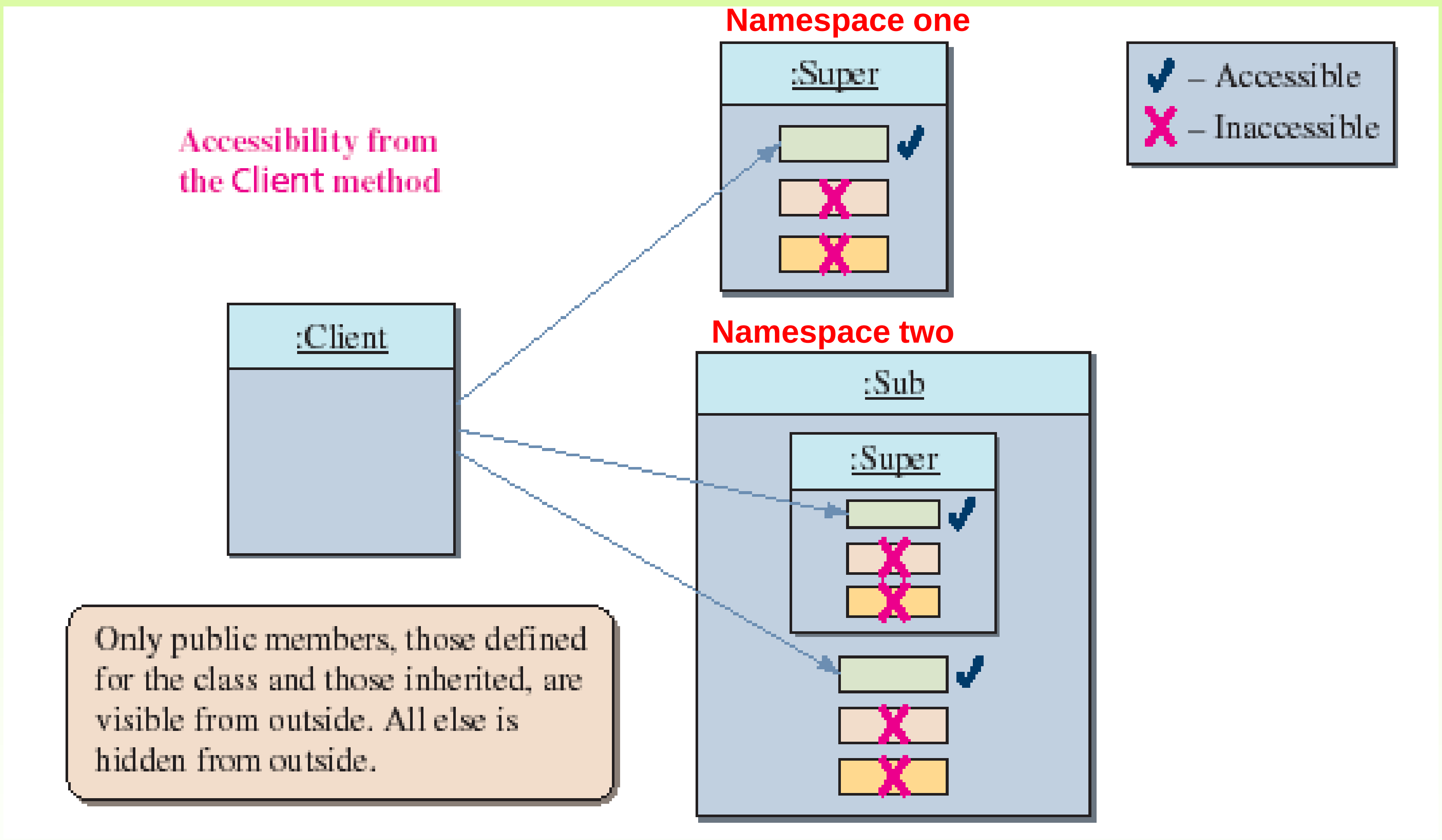
Class Hierarchy



Instances

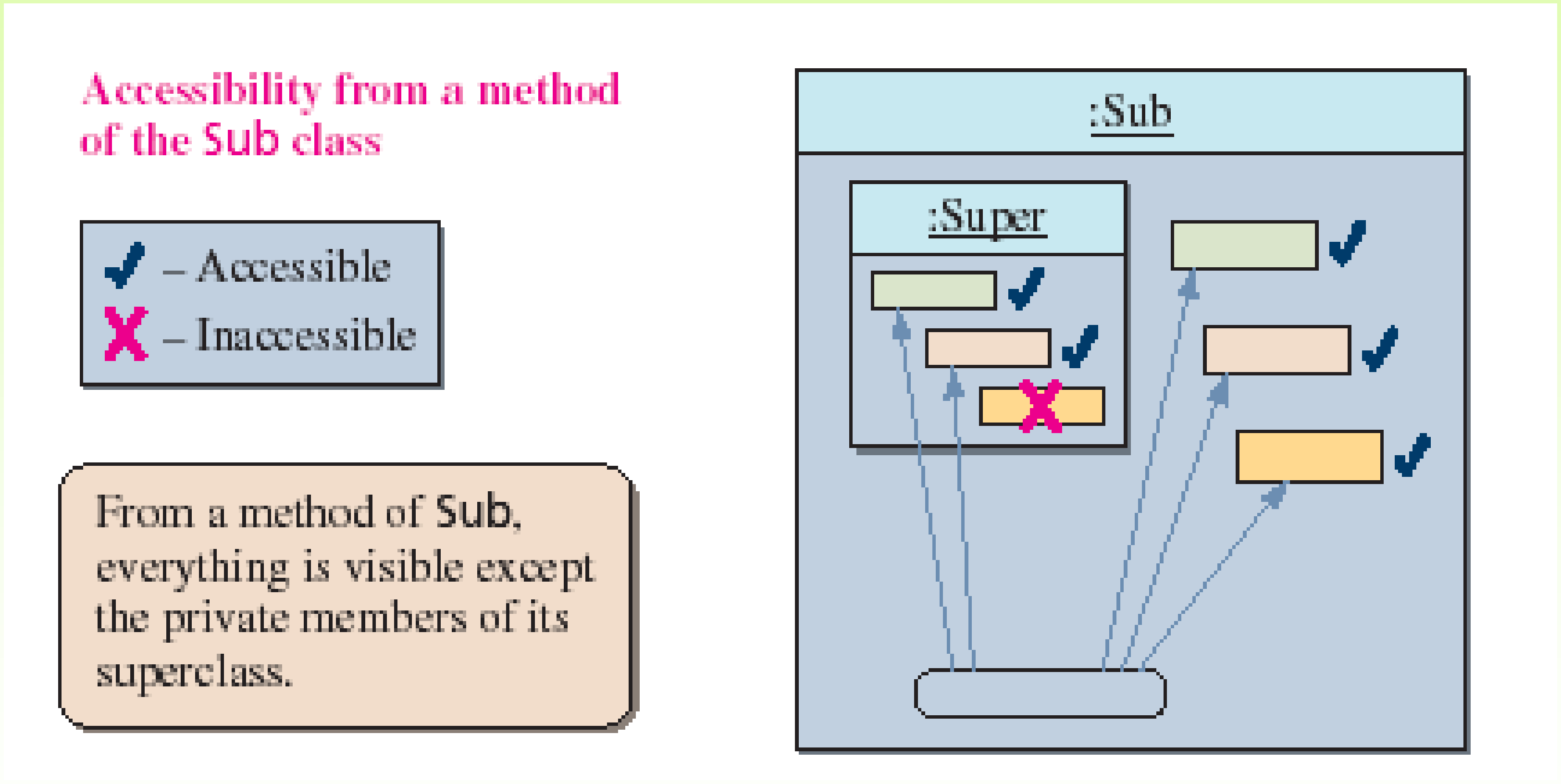
This shows the inherited components of the superclass are part of the subclass instance

The Effect of Three Visibility Modifiers

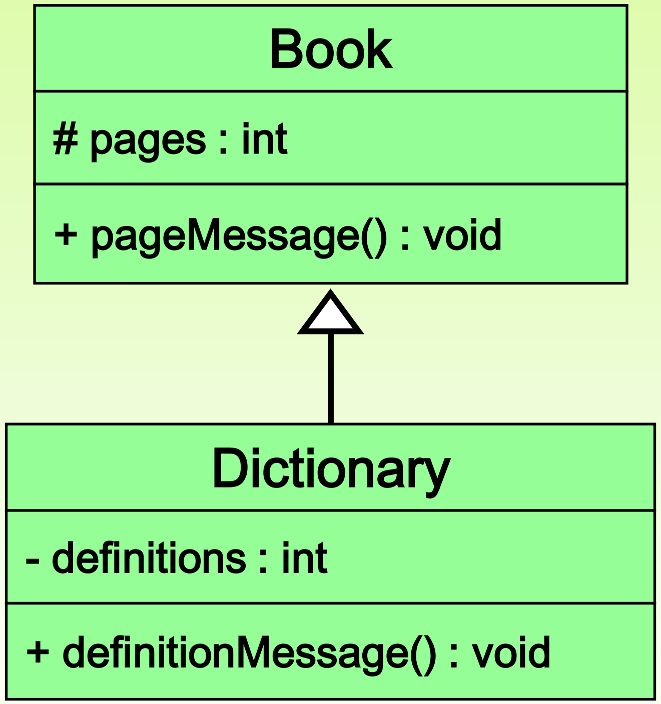

Namespace one
:Super
✓ – Accessible
✗ – Inaccessible
Accessibility from the Client method
:Client
Namespace two
:Sub
:Super
Only public members, those defined for the class and those inherited, are visible from outside. All else is hidden from outside.

Accessibility of Super from Sub

- Everything except the private members of the Super class is visible from a method of the Sub class.

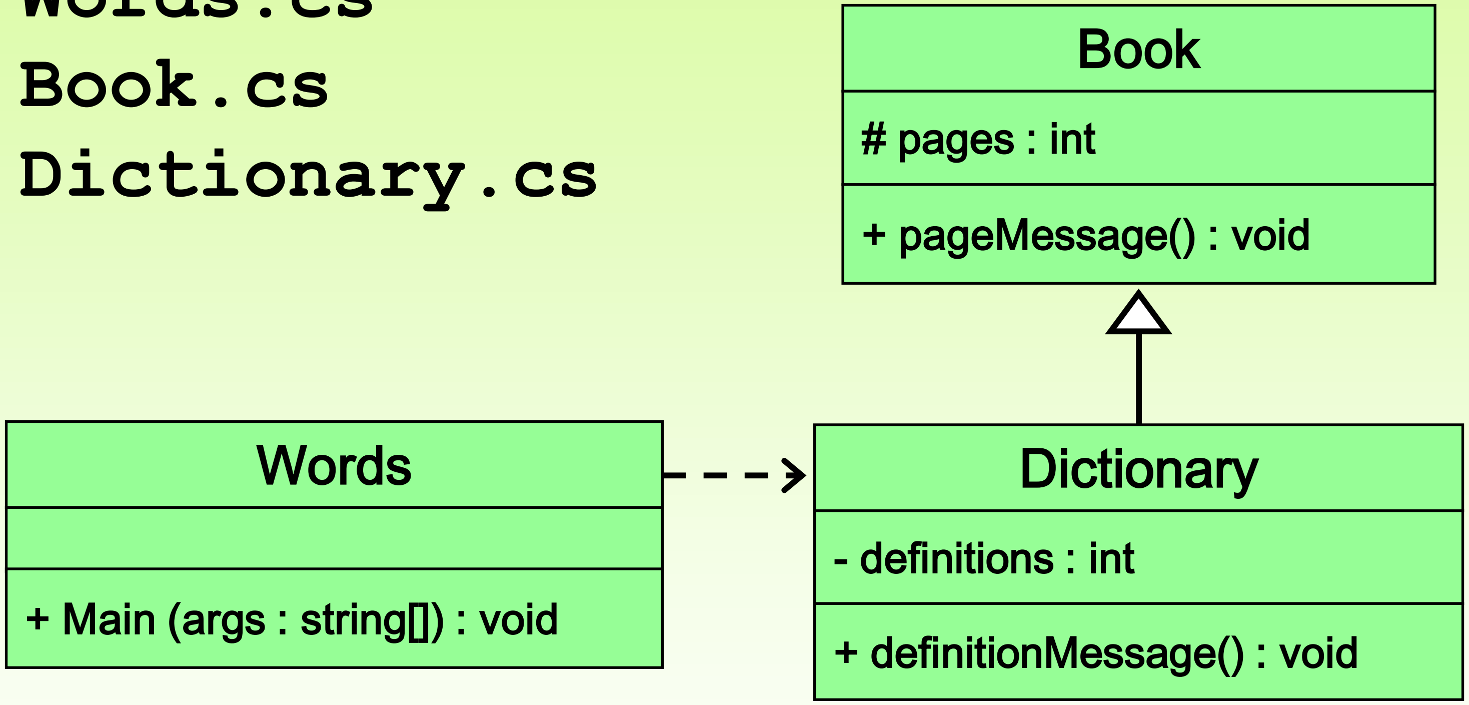

Accessibility from a method of the Sub class
✓ – Accessible
✗ – Inaccessible
From a method of Sub, everything is visible except the private members of its superclass.
:Sub
:Super

- See `Words.cs`
- See `Book.cs`
- See `Dictionary.cs`



Class Diagram for Words Example

- `See Words.cs`
- `See Book.cs`
- `See Dictionary.cs`



```
//********************************************************************
//  Words.cs       Author: Lewis/Loftus
//
//  Demonstrates the use of an inherited method.
//********************************************************************

public class Words
{
   //-----------------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------------
   public static void Main (string[] args)
   {
      Dictionary webster = new Dictionary();

      Console.WriteLine ("Number of pages: " + webster.getPages());

      Console.WriteLine ("Number of definitions: " +
                          webster.getDefinitions());

      Console.WriteLine ("Definitions per page: " +
                          webster.computeRatio());
   }
}
```

```
//**************                                   *****************
//  Words.cs
//
//  Demonstrates
//**************                                   *****************

public class Words
{
   //-----------------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------------
   public static void Main (string[] args)
   {
      Dictionary webster = new Dictionary();

      Console.WriteLine ("Number of pages: " + webster.getPages());

      Console.WriteLine ("Number of definitions: " +
                         webster.getDefinitions());

      Console.WriteLine ("Definitions per page: " +
                         webster.computeRatio());
   }
}
```

Output

```
Number of pages: 1500
Number of definitions: 52500
Definitions per page: 35.0
```

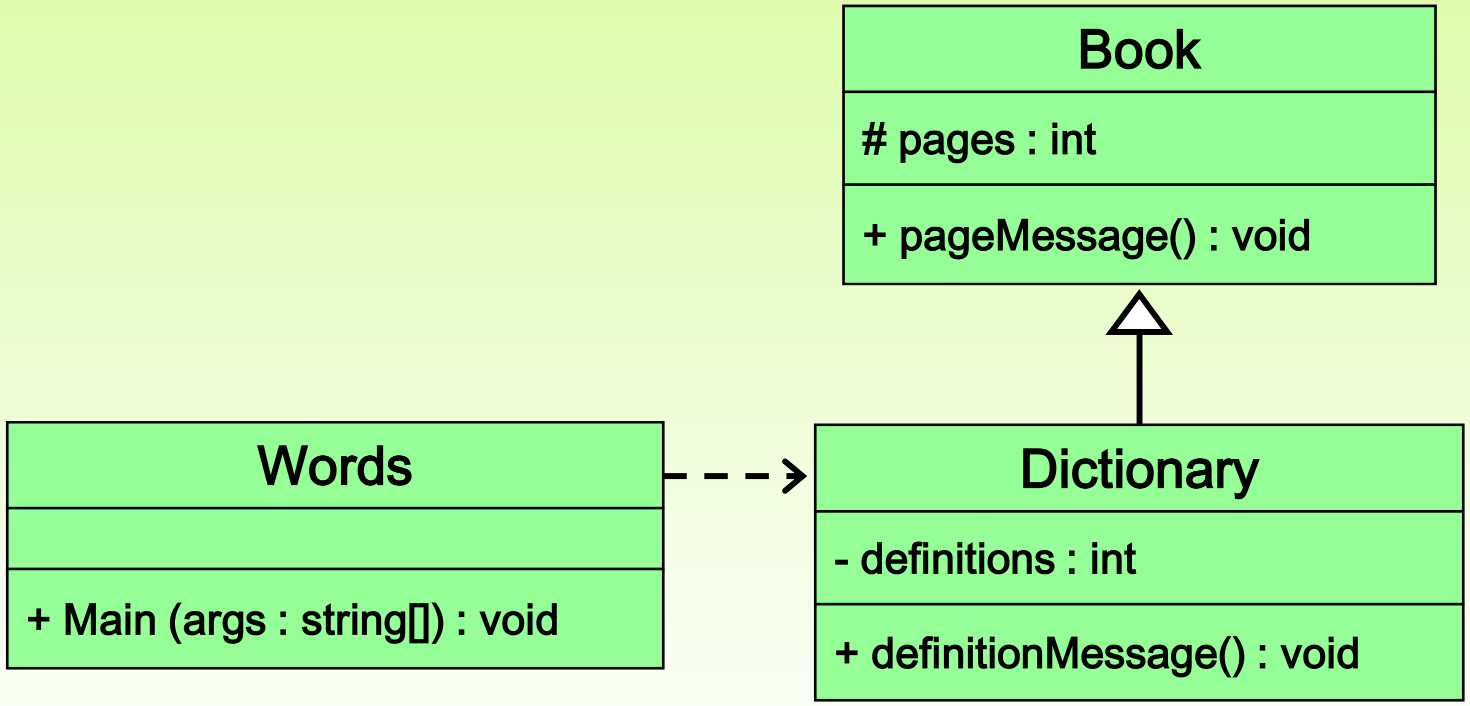
```
//**********************************************************************
//  Book.cs       Author: Lewis/Loftus
//
//  Represents a book. Used as the parent of a derived class to
//  demonstrate inheritance.
//**********************************************************************

public class Book
{
   protected int pages = 1500;

   //-----------------------------------------------------------------
   //  Pages mutator.
   //-----------------------------------------------------------------
   public void setPages (int numPages)
   {
      pages = numPages;
   }

   //-----------------------------------------------------------------
   //  Pages accessor.
   //-----------------------------------------------------------------
   public int getPages ()
   {
      return pages;
   }
}
```

```
//********************************************************************
//  Dictionary.cs       Author: Lewis/Loftus
//
//  Represents a dictionary, which is a book. Used to demonstrate
//  inheritance.
//********************************************************************

public class Dictionary : Book
{
   private int definitions = 52500;

   //-----------------------------------------------------------------
   //  Prints a message using both local and inherited values.
   //-----------------------------------------------------------------
   public double computeRatio ()
   {
      return (double) definitions/pages;
   }

continue
```

continue

```
   //-----------------------------------------------------------------
   //  Definitions mutator.
   //-----------------------------------------------------------------
   public void setDefinitions (int numDefinitions)
   {
      definitions = numDefinitions;
   }

   //-----------------------------------------------------------------
   //  Definitions accessor.
   //-----------------------------------------------------------------
   public int getDefinitions ()
   {
      return definitions;
   }
}
```

Class Diagram for Words



The base Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `base` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor
- A child’s constructor is responsible for calling the parent’s constructor

The base Reference

- A child's constructor is responsible for calling the parent's constructor

- The first line of a child's constructor should use the `base` reference to call the parent's constructor

- The `base` reference can also be used to reference other variables and methods defined in the parent's class

- See `Words2.cs`
- See `Book2.cs`
- See `Dictionary2.cs`

```
//********************************************************************
//  Words2.cs       Author: Lewis/Loftus
//
//  Demonstrates the use of the super reference.
//********************************************************************

public class Words2
{
   //-----------------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------------
   public static void Main (string[] args)
   {
      Dictionary2 webster = new Dictionary2 (1500, 52500);

      Console.WriteLine ("Number of pages: " + webster.getPages());

      Console.WriteLine ("Number of definitions: " +
                          webster.getDefinitions());

      Console.WriteLine ("Definitions per page: " +
                          webster.computeRatio());
   }
}
```

```
//****************                                    *****************
//  Words2.cs
//
//  Demonstrates t
//****************                                    *****************

public class Words2
{
   //-----------------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------------
   public static void Main (string[] args)
   {
      Dictionary2 webster = new Dictionary2 (1500, 52500);

      Console.WriteLine ("Number of pages: " + webster.getPages());

      Console.WriteLine ("Number of definitions: " +
                         webster.getDefinitions());

      Console.WriteLine ("Definitions per page: " +
                         webster.computeRatio());
   }
}
```

Output

```
Number of pages: 1500
Number of definitions: 52500
Definitions per page: 35.0
```

```
//********************************************************************
//  Book2.cs       Author: Lewis/Loftus
//
//  Represents a book. Used as the parent of a derived class to
//  demonstrate inheritance and the use of the super reference.
//********************************************************************

public class Book2
{
   protected int pages;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the book with the specified number of
   //  pages.
   //-----------------------------------------------------------------
   public Book2 (int numPages)
   {
      pages = numPages;
   }
```

continue

```
continue

   //------------------------------------------------------------------
   //  Pages mutator.
   //------------------------------------------------------------------
   public void setPages (int numPages)
   {
      pages = numPages;
   }

   //------------------------------------------------------------------
   //  Pages accessor.
   //------------------------------------------------------------------
   public int getPages ()
   {
      return pages;
   }
}
```

```
//*********************************************************************
//  Dictionary2.cs       Author: Lewis/Loftus
//
//  Represents a dictionary, which is a book. Used to demonstrate
//  the use of the super reference.
//*********************************************************************

public class Dictionary2 : Book2
{
   private int definitions;

   //------------------------------------------------------------------
   //  Constructor: Sets up the dictionary with the specified number
   //  of pages and definitions.
   //------------------------------------------------------------------
   public Dictionary2 (int numPages, int numDefinitions): base(numPages)
   {

      definitions = numDefinitions;
   }

continue
```

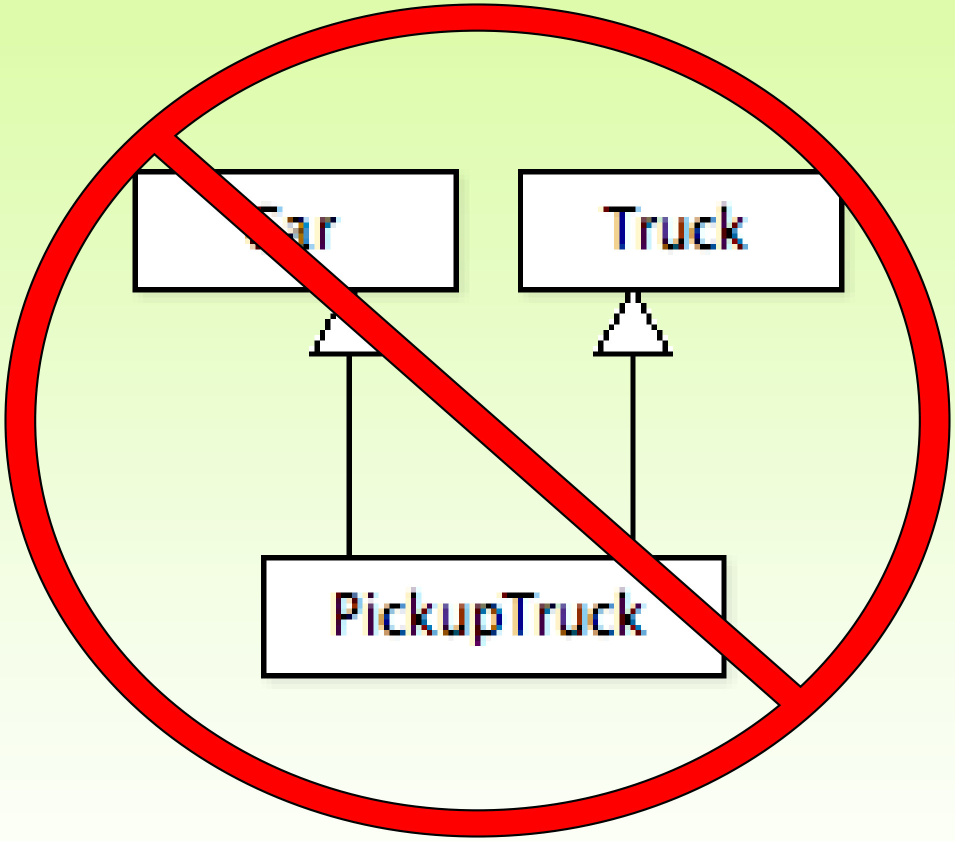
continue

```
   //-----------------------------------------------------------------
   //  Prints a message using both local and inherited values.
   //-----------------------------------------------------------------
   public double computeRatio ()
   {
      return (double) definitions/pages;
   }

   //----------------------------------------------------------------
   //  Definitions mutator.
   //----------------------------------------------------------------
   public void setDefinitions (int numDefinitions)
   {
      definitions = numDefinitions;
   }

   //----------------------------------------------------------------
   //  Definitions accessor.
   //----------------------------------------------------------------
   public int getDefinitions ()
   {
      return definitions;
   }
}
```

Multiple Inheritance

- C# supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Multiple inheritance is generally not needed, and C# does not support it

Multiple Inheritance



Outline

- Creating Subclasses
- ⇨ Overriding Methods
- Class Hierarchies
- Abstract Classes
- Inheritance and Visibility
- Designing for Inheritance

Overriding Methods

- A child class can *override* the definition of an inherited method in favor of its own
- The new method must have the same signature as the parent's method, but can have a different body
- The Virtual Modifier is used to mark that a method\property(ect) can be modified in a derived class by using the override modifier.
- The type of the object executing the method determines which version of the method is invoked
- See `Messages.cs`
- See `Thought.cs`
- See `Advice.cs`

```
//********************************************************************
//  Messages.cs       Author: Lewis/Loftus
//
//  Demonstrates the use of an overridden method.
//********************************************************************

public class Messages
{
   //-----------------------------------------------------------------
   //  Creates two objects and invokes the message method in each.
   //-----------------------------------------------------------------
   public static void Main (string[] args)
   {
      Thought parked = new Thought();
      Advice dates = new Advice();

      parked.message();

      dates.message();  // overridden
   }
}
```

Output

```
I feel like I'm diagonally parked in a parallel universe.

Warning: Dates in calendar are closer than they appear.

I feel like I'm diagonally parked in a parallel universe.
```

```
//
//
//
//
//

pu
{
   //-----------------------------------------------------------------
   //  Creates two objects and invokes the message method in each.
   //-----------------------------------------------------------------
   public static void Main (string[] args)
   {
      Thought parked = new Thought();
      Advice dates = new Advice();

      parked.message();

      dates.message();  // overridden
   }
}
```

```
//********************************************************************
//  Thought.cs       Author: Lewis/Loftus
//
//  Represents a stray thought. Used as the parent of a derived
//  class to demonstrate the use of an overridden method.
//********************************************************************

public class Thought
{
   //-----------------------------------------------------------------
   //  Prints a message.
   //-----------------------------------------------------------------
   public virtual void message()
   {
      Console.WriteLine ("I feel like I'm diagonally parked in a " +
                         "parallel universe.");

      Console.WriteLine();
   }
}
```

```
//********************************************************************
//  Advice.cs       Author: Lewis/Loftus
//
//  Represents some thoughtful advice. Used to demonstrate the use
//  of an overridden method.
//********************************************************************

public class Advice : Thought
{
   //-----------------------------------------------------------------
   //  Prints a message. This method overrides the parent's version.
   //-----------------------------------------------------------------
   public override void message()
   {
      Console.WriteLine ("Warning: Dates in calendar are closer " +
                         "than they appear.");

      Console.WriteLine();

      base.message();  // explicitly invokes the parent's version
   }
}
```

Overriding

- The override modifier is required to extend or modify the abstract or virtual implementation of an inherited method
- A method in the parent class can be invoked explicitly using the `base` reference
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Overloading vs. Overriding

- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different parameters
- Overriding lets you define a similar operation in different ways for different object types

Quick Check

`True or False?`

A child class may define a method with the same name as a method in the parent.

A child class can override the constructor of the parent class.

A child class cannot override a `sealed` not virtual method of the parent class.
It is considered poor design when a child class overrides a method from the parent.

A child class may define a variable with the same name as a variable in the parent.

Quick Check

True or False?

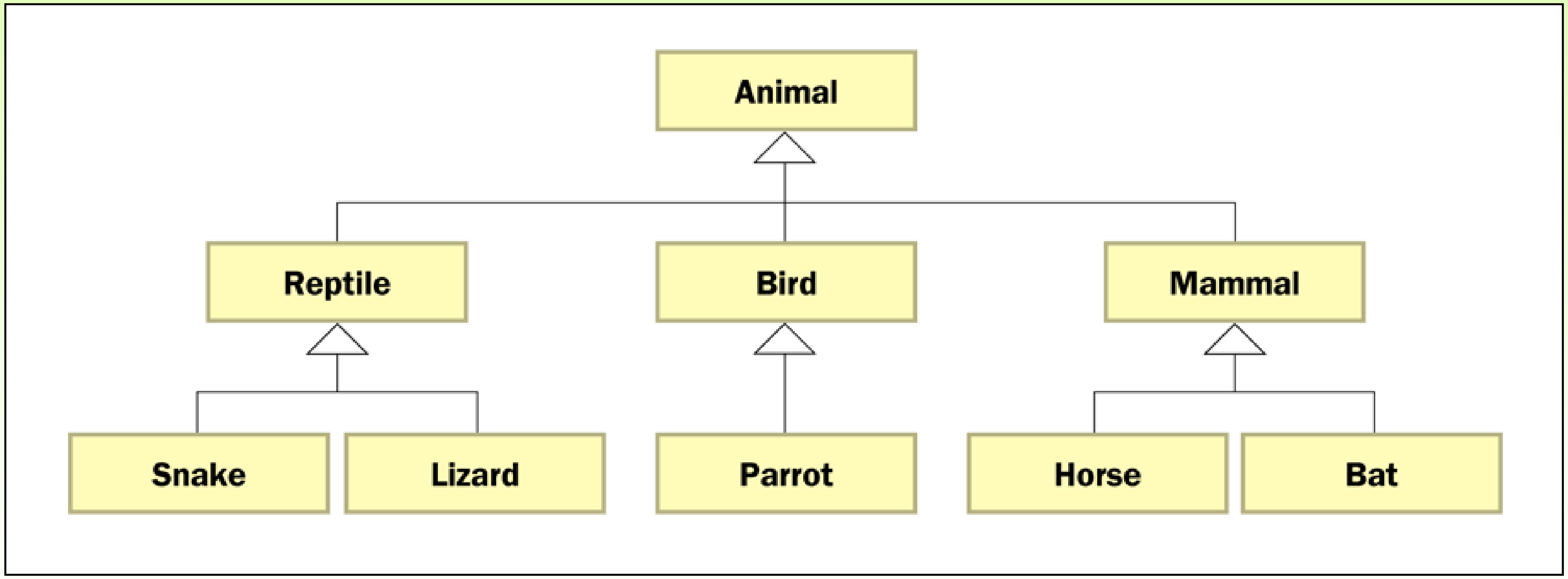
A child class may define a method with the same name as a method in the parent.	True
A child class can override the constructor of the parent class.	False
A child class cannot override a sealed not virtual method of the parent class.	True
It is considered poor design when a child class overrides a method from the parent.	False
A child class may define a variable with the same name as a variable in the parent.	True, but shouldn't

Outline

- Creating Subclasses
- Overriding Methods
- ⇨ Class Hierarchies
- Abstract Classes
- Inheritance and Visibility
- Designing for Inheritance

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*



Class Hierarchies

- Two children of the same parent are called *siblings*
- Common features should be put as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
- Therefore, a child class inherits from all its ancestor classes
- The system executes the first override-virtual method found in the hierarchy
- There is no single class hierarchy that is appropriate for all situations

The Object Class

- A class called `Object` is defined in the `cs.lang` Namespace of the C# standard class library
- All classes are derived from the `Object` class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
- Therefore, the `Object` class is the ultimate root of all class hierarchies

The Object Class

- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `Tostring` method is defined in the `Object` class
- Every time we define the `Tostring` method, we are actually overriding an inherited definition
- The `Tostring` method in the `Object` class is defined to return a string that contains the name of the object's class along with a hash code

The Object Class

- The `Equals` method of the `Object` class returns true if two references are aliases
- We can override `Equals` in any class to define equality in some more appropriate way
- As we've seen, the `string` class defines the `Equals` method to return true if two `string` objects contain the same characters
- The designers of the `string` class have overridden the `Equals` method inherited from `Object` in favor of a more useful version

Outline

- Creating Subclasses
- Overriding Methods
- Class Hierarchies
- Abstract Classes
- Inheritance and Visibility
- Designing for Inheritance

Abstract Classes

- Super classes are more general and subclasses are more specific.
- Sometimes a base class is so general that it doesn't make sense to actually instantiate it (i.e. create an object from it).
- Such a class is primarily a grouping place for common data and behaviors of subclasses -- an **abstract class**.
- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept

Abstract Classes

- We use the modifier `abstract` on the class header to declare a class as abstract:

```
public abstract class Shape
{
    // class contents
}
```

- An abstract class cannot be instantiated
- Now that Shape is abstract, this would be illegal:
  - `Shape s = new Shape();`
  - Specifically, it's `new Shape();` that is illegal

Abstract Classes

- An abstract class often contains abstract methods with no definitions
- An abstract method is a method with the keyword **abstract**, and it ends with a semicolon instead of a method body.
  - Private methods and static methods may not be declared **abstract**
- Also, an abstract class typically contains non-abstract methods with full definitions
- A class declared as abstract does not have to contain abstract methods -- simply declaring it as abstract makes it so
- So An *abstract class* is a class
  - defined with the modifier **abstract** OR
  - that contains an abstract method OR
  - that does not provide an implementation of an inherited abstract method

Abstract Classes

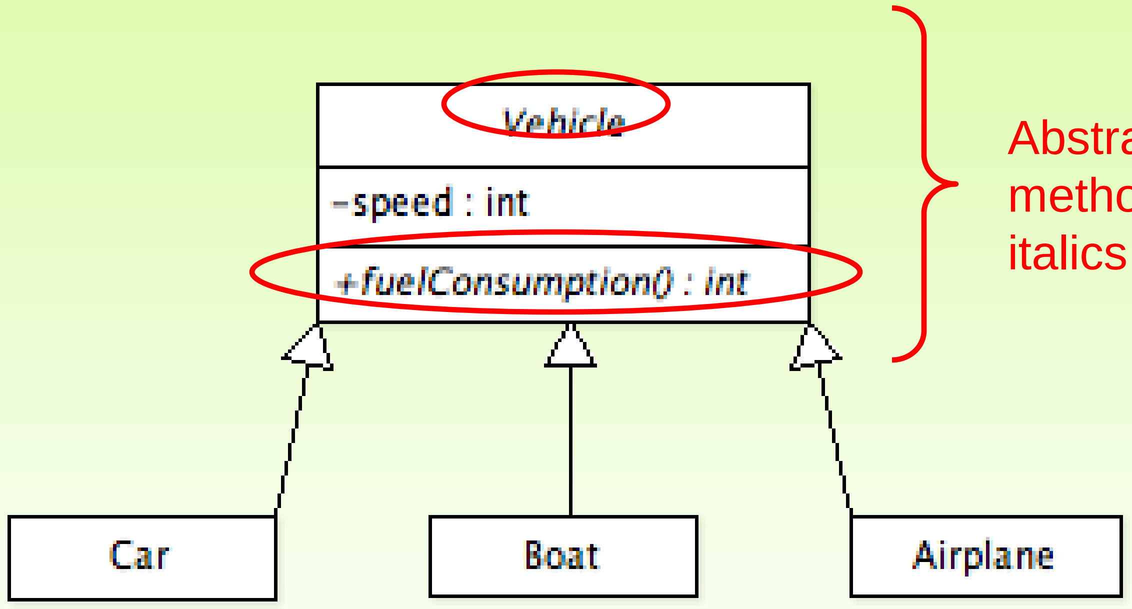
- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract
- An abstract method cannot be defined as sealed or static
- However an abstract classes can have static methods which can be called directly from the abstract class
- You can also override a static method if needed
- When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method

Abstract Methods

- Methods can be abstract as well:
- An abstract method is a method signature without a definition
- Abstract methods can *only* be created inside abstract classes
- The main purpose of an abstract method is to be overridden in derived classes (with the same signature)
- Example:

```
public abstract class Shape  // Shape is an abstract class
{ public abstract double findArea(); // findArea is an abstract method
  // other methods and data
}
```

An Abstract Class in UML



Abstract classes and methods are shown in italics font.

Quick Check

What are some methods defined by the Object class?

What is an abstract class?

Quick Check

What are some methods defined by the Object class?

```
ToString()
Equals(Object obj)
```

What is an abstract class?

An abstract class is a placeholder in the class hierarchy, defining a general concept and gathering elements common to all derived classes. An abstract class cannot be instantiated.

Outline

- Creating Subclasses
- Overriding Methods
- Class Hierarchies
- ➡ Inheritance and Visibility
- Designing for Inheritance

Visibility Revisited

- It's important to understand one subtle issue related to inheritance and visibility
- All variables and methods of a parent class, even private members, are inherited by its children
- As we've mentioned, private members cannot be referenced by name in the child class
- However, private members inherited by child classes exist and can be referenced indirectly

Visibility Revisited

- Because the parent can refer to the private member, the child can reference it indirectly using its parent's methods
- The `base` reference can be used to refer to the parent class, even if no object of the parent exists
- See `FoodAnalyzer.cs`
- See `FoodItem.cs`
- See `Pizza.cs`

```
//********************************************************************
//  FoodAnalyzer.cs       Author: Lewis/Loftus
//
//  Demonstrates indirect access to inherited private members.
//********************************************************************

public class FoodAnalyzer
{
   //-----------------------------------------------------------------
   //  Instantiates a Pizza object and prints its calories per
   //  serving.
   //-----------------------------------------------------------------
   public static void Main (string[] args)
   {
      Pizza special = new Pizza (275);

      Console.WriteLine ("Calories per serving: " +
                         special.caloriesPerServing());
   }
}
```

```
//**************                                **********************
//  FoodAnalyzer.
//
//  Demonstrates                                vate members.
//********************************************************************

public class FoodAnalyzer
{
   //-----------------------------------------------------------------
   //  Instantiates a Pizza object and prints its calories per
   //  serving.
   //-----------------------------------------------------------------
   public static void Main (string[] args)
   {
      Pizza special = new Pizza (275);

      Console.WriteLine ("Calories per serving: " +
                         special.caloriesPerServing());
   }
}
```

Output

```
Calories per serving: 309
```

```
//**********************************************************************
//   FoodItem.cs        Author: Lewis/Loftus
//
//   Represents an item of food. Used as the parent of a derived class
//   to demonstrate indirect referencing.
//**********************************************************************

public class FoodItem
{
   Private const int CALORIES_PER_GRAM = 9;
   private int fatGrams;
   protected int servings;

   //-----------------------------------------------------------------
   //   Sets up this food item with the specified number of fat grams
   //   and number of servings.
   //-----------------------------------------------------------------
   public FoodItem (int numFatGrams, int numServings)
   {
      fatGrams = numFatGrams;
      servings = numServings;
   }

continue
continue

   //-----------------------------------------------------------------
   //  Computes and returns the number of calories in this food item
   //  due to fat.
   //-----------------------------------------------------------------
   private int calories()
   {
      return fatGrams * CALORIES_PER_GRAM;
   }

   //-----------------------------------------------------------------
   //  Computes and returns the number of fat calories per serving.
   //-----------------------------------------------------------------
   public int caloriesPerServing()
   {
      return (calories() / servings);
   }
}
```

```
//********************************************************************
//   Pizza.cs       Author: Lewis/Loftus
//
//   Represents a pizza, which is a food item. Used to demonstrate
//   indirect referencing through inheritance.
//********************************************************************

public class Pizza : FoodItem
{
   //-----------------------------------------------------------------
   //  Sets up a pizza with the specified amount of fat (assumes
   //  eight servings).
   //-----------------------------------------------------------------
   public Pizza (int fatGrams): base(fatGrams, 8)
   {

   }
}
```

Outline

- Creating Subclasses
- Overriding Methods
- Class Hierarchies
- Abstract Classes
- Inheritance and Visibility
- ➡ Designing for Inheritance

Designing for Inheritance

- As we've discussed, taking the time to create a good software design reaps long-term benefits
- Inheritance issues are an important part of an object-oriented design
- Properly designed inheritance relationships can contribute greatly to the elegance, Maintainability, and reuse of the software
- Let's summarize some of the issues regarding inheritance that relate to a good software design

Inheritance Design Issues

- Every derivation should be an is-a relationship
- Think about the potential future of a class hierarchy, and design classes to be reusable and flexible
- Find common characteristics of classes and push them as high in the class hierarchy as appropriate
- Override methods as appropriate to tailor or change the functionality of a child
- Add new variables to children, but don't redefine (shadow) inherited variables

Inheritance Design Issues

- Allow each class to manage its own data; use the `base` reference to invoke the parent's constructor to set up its data
- Override general methods such as `Tostring` and `Equals` with appropriate definitions
- Use abstract classes to represent general concepts that derived classes have in common
- Use visibility modifiers carefully to provide needed access without violating encapsulation

Restricting Inheritance

- If the `sealed` modifier is applied to an entire class, then that class cannot be used to derive any children at all
- Therefore, an abstract class cannot be declared as sealed
- If the `sealed` modifier is applied to a method, that method cannot be overridden in any derived classes

Restricting Inheritance

In C# Methods are sealed by default, if they're not declared as virtual and not overriding another virtual method

A method, or property on a derived class that is overriding a virtual member of the base class can declare that member as sealed.

This negates the virtual aspect of the member for any further derived class. This is accomplished by putting the **sealed keyword before the override** keyword in the class member declaration. For example:

```
public class D : C

{ public sealed override void DoWork() { }  }
```

Sealed method is used to define the overriding level of a virtual method. Sealed keyword is always used with override keyword.