CSC220 (CSI)
Computational Problem Solving

Inheritance

The College of New Jersey

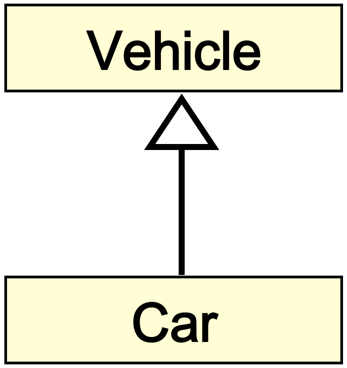
Please turn off your cell phone!

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

Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

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

```
//*******************************                                     *
//  Words.java       Author: Lewis
//
//  Demonstrates the use of an inh
//*******************************                                     *

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

      System.out.println("Number of pages: " + webster.getPages());

      System.out.println("Number of definitions: " +
                         webster.getDefinitions());

      System.out.println("Definitions per page: " +
                         webster.computeRatio());
   }
}
```

Output

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

```java
//********************************************************************
//  Book.java       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;
   }
}
```

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

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

   //  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;
   }
}
```

The protected Modifier

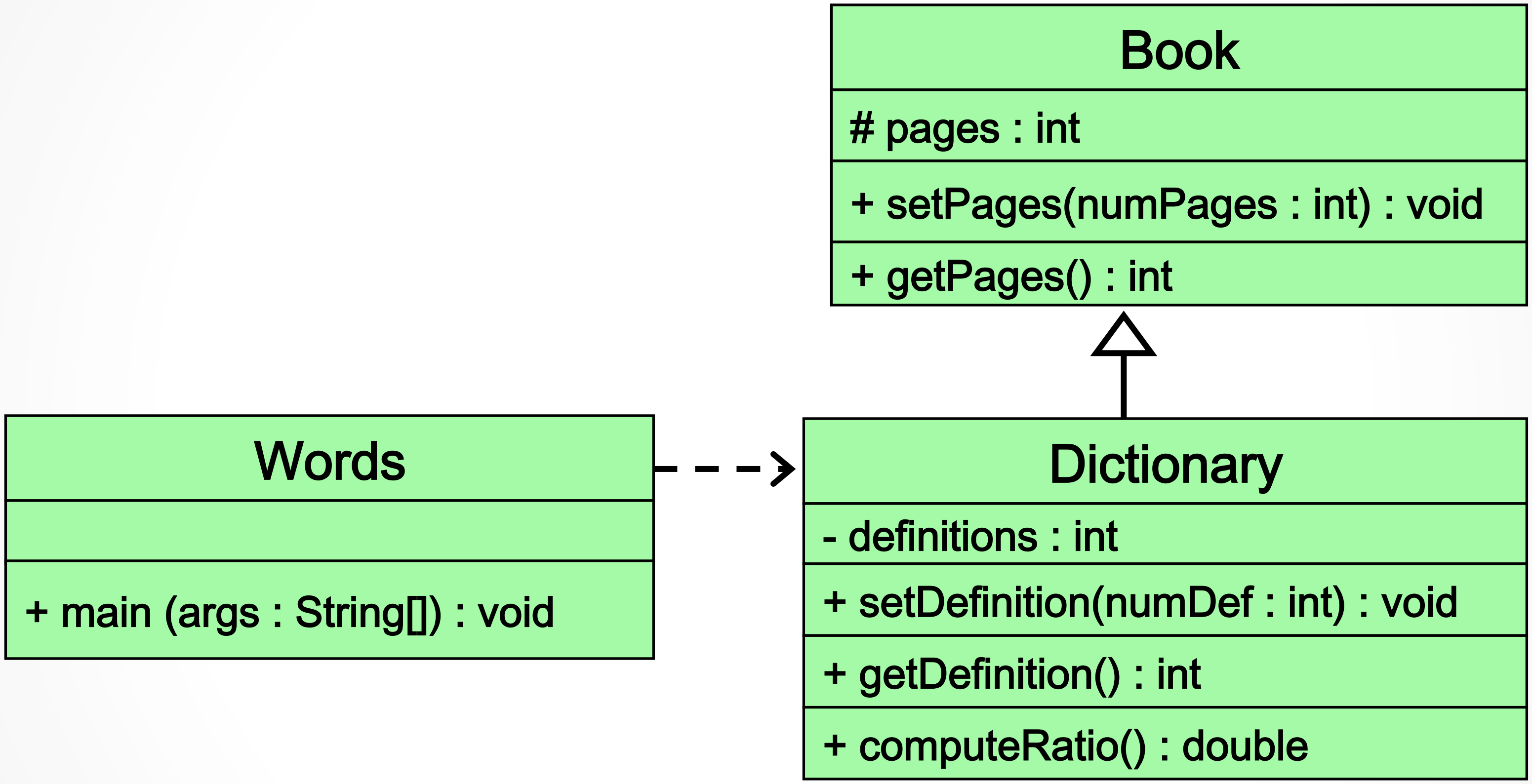
- Variables and methods declared with private visibility cannot be referenced in a child class

- They can be referenced in the child class if they are declared with public visibility -- but public variables violate the principle of encapsulation

- The `protected` modifier allows a child class to reference a variable or method in the child class

- A protected variable is also visible to any class in the same package as the parent class

Class Diagram for Words


Book
# pages : int
+ setPages(numPages : int) : void
+ getPages() : int
Words
+ main (args : String[]) : void
Dictionary
- definitions : int
+ setDefinition(numDef : int) : void
+ getDefinition() : int
+ computeRatio() : double

- Protected variables and methods can be shown with a # symbol preceding them in UML diagrams

The super Reference

- Constructors are **not** inherited, even though they have public visibility
- Yet we often use the parent's constructor to set up the "parent's part" of the object
- The `super` 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 first line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent's class

```java
//****************************                                    *****
//  Words2.java       Author:
//
//  Demonstrates the use of th
//****************************                                    *****

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);

      System.out.println("Number of pages: " + webster.getPages());

      System.out.println("Number of definitions: " +
                         webster.getDefinitions());

      System.out.println("Definitions per page: " +
                         webster.computeRatio());
   }
}
```

Output

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

```java
//********************************************************************
//  Book2.java       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;
   }

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

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

```java
//********************************************************************
//  Dictionary2.java       Author: Lewis/Loftus
//  Represents a dictionary, which is a book. Used to demonstrate
//  the use of the super reference.
//********************************************************************
public class Dictionary2 extends Book2 {
   private int definitions;
   //  Constructor: Sets up the dictionary with the specified number
   //  of pages and definitions.
   public Dictionary2(int numPages, int numDefinitions) {
      super(numPages);
      definitions = numDefinitions;
   }

   //  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

- Java 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 Java does not support it

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 type of the object executing the method determines which version of the method is invoked

```
//********************************************************************
//  Thought.java       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 void message(){
      System.out.println("I feel like I'm diagonally parked in a " +
                         "parallel universe.");
      System.out.println();
   }
}
```

```
//********************************************************************
//  Advice.java       Author: Lewis/Loftus
//  Represents some thoughtful advice. Used to demonstrate the use
//  of an overridden method.
//********************************************************************
public class Advice extends Thought{
   //  Prints a message. This method overrides the parent's version.
   public void message(){
      System.out.println("Warning: Dates in calendar are closer " +
                         "than they appear.");
      System.out.println();
      super.message();  // explicitly invokes the parent's version
   }
}
```

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.
```

```
//*
//
//
//
//*

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
   }
}
```

Overriding

- A method in the parent class can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- 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
  - lets you define a similar operation in different ways for different parameters
- Overriding
  - deals with two methods, one in a parent class and one in a child class, that have the same signature
  - lets you define a similar operation in different ways for different object types

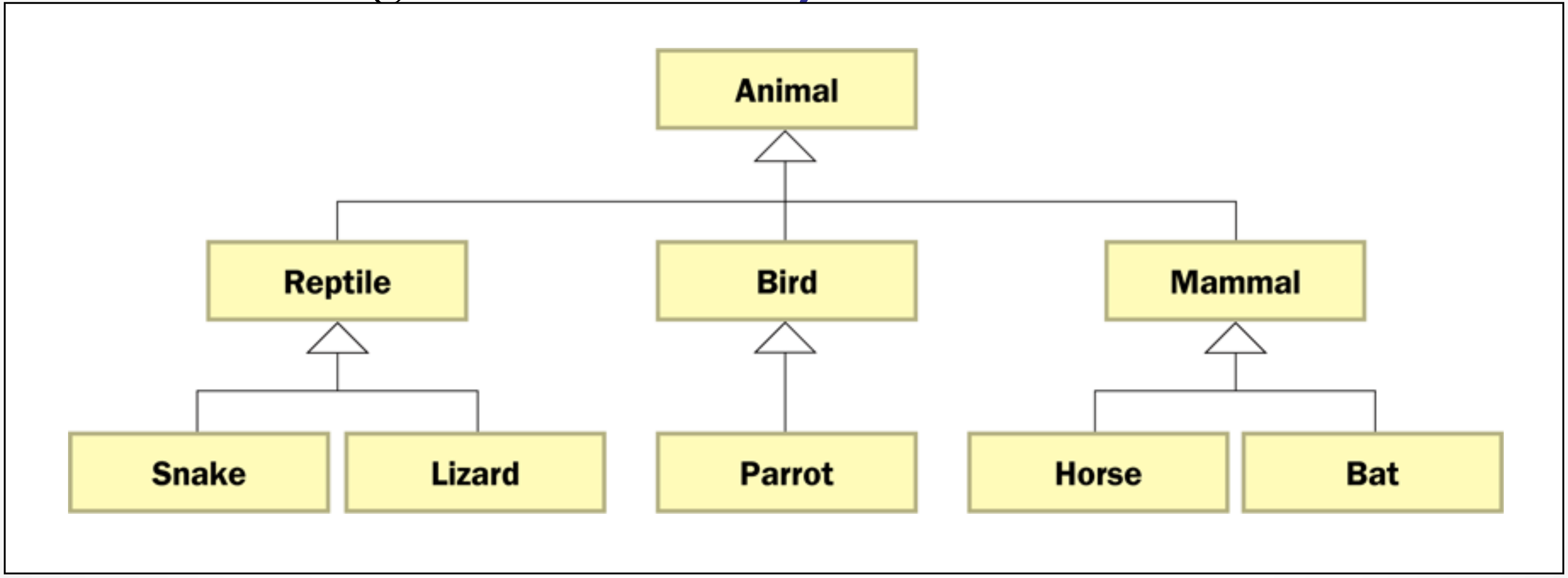
Quick Check

True or False?

- A child class may define a method with the same name as a method in the parent. True
- A child class can override the constructor of the parent class. False
- A child class cannot override a `final` 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

Class Hierarchies

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



- Two children of the same parent are called siblings
- Common features should be put as high in the hierarchy as is reasonable

The Object Class

- A class called `Object` is defined in the `java.lang` package of the Java 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
- 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 `equals` method of the `Object` class returns true if two references are aliases

Abstract Classes

- An abstract class is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract:

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

Abstract Classes

- An abstract class often contains abstract methods with no definitions (like an interface)
- Unlike an interface, the `abstract` modifier must be applied to each abstract method
- 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

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 `final` or `static`
- The use of abstract classes is an important element of software design – it allows us to establish common elements in a hierarchy that are too general to instantiate

Interface Hierarchies

- Inheritance can be applied to interfaces
- That is, one interface can be derived from another interface
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both interfaces
- Class hierarchies and interface hierarchies are distinct (they do not overlap)

Visibility Revisited

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

```java
//********************************************************************
//  FoodAnalyzer.java       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);

      System.out.println("Calories per serving: " +
                         special.caloriesPerServing());
   }
}
```

```java
//*******************************************************************
//  FoodItem.java       Author: Lewis/Loftus
//  Represents an item of food. Used as the parent of a derived class
//  to demonstrate indirect referencing.
//*******************************************************************
public class FoodItem {
   final private 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;
   }

   //  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);
   }
}
```

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

public class Pizza extends FoodItem
{
   //-----------------------------------------------------------------
   //  Sets up a pizza with the specified amount of fat (assumes
   //  eight servings).
   //-----------------------------------------------------------------
   public Pizza(int fatGrams)
   {
      super(fatGrams, 8);
   }
}
```

```
//**************                    *********************
//  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);

      System.out.println("Calories per serving: " +
                         special.caloriesPerServing());
   }
}
```

Output

```
Calories per serving: 309
```

Inheritance Design Issues

- Every derivation should be an is-a relationship
- 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
- Allow each class to manage its own data; use the `super` 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 `final` modifier is applied to a method, that method cannot be overridden in any derived classes
- If the `final` 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 final

Summary

- deriving new classes from existing classes
- the `protected` modifier
- creating class hierarchies
- abstract classes
- indirect visibility of inherited members
- designing for inheritance