# Inheritance

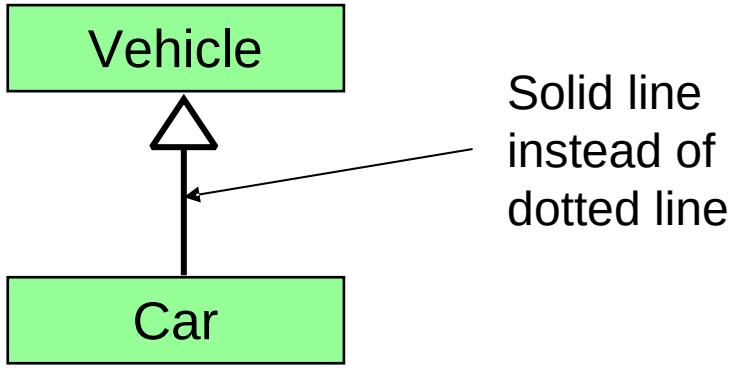
- Inheritance
- Reserved word *protected*
- Reserved word *super*
- Overriding methods
- Class Hierarchies
- Reading for this lecture: L&L 8.1 – 8.5

# Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class, superclass*, or *base class*
- The new class is called the *child class, subclass* or *derived class*
- 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 is based on an *is-a* relationship
- The child *is a* more specific version of the parent
- Inheritance relationships are shown in a UML class diagram using a solid arrow with an unfilled triangular arrowhead pointing to the parent class (Note: Similar graphic notation as Interface)



# Inheritance

- *Software reuse* is a fundamental benefit of inheritance
- 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
- However, a programmer can tailor a derived class as needed by adding new variables or by “overriding” some of the inherited methods

# Deriving Subclasses

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

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

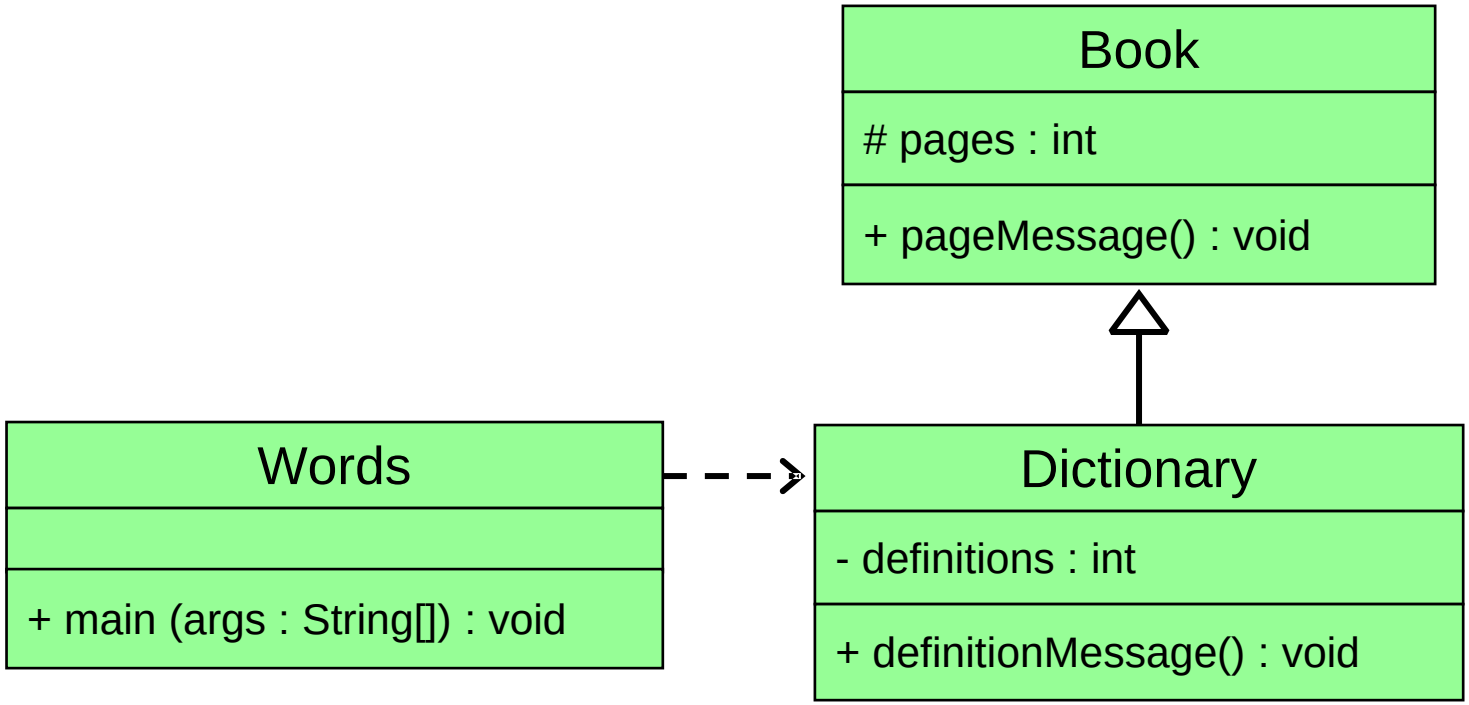
# The protected Modifier

- 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 by name 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 directly in the child class
- It provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility
- A protected variable is visible in any class that is a child of the class where it is defined
- A protected variable is also visible in any class that is in the same package as the class where it is defined (we don't use packages in this course)
- Protected variables and methods can be shown with a # symbol in UML class diagrams

# Class Diagram for Words



# The super 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 `super` reference can be used to refer to the parent class and invoke the parent's constructor

```
public class Child extends Parent
{
public Child()
    {
          super();   // a call to Parent()
          // plus whatever code we need for Child
    }
}
```

# The super Reference

- 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 (with a dot .) other variables and methods defined in the parent's class

# Multiple Inheritance

- *Multiple inheritance* allows a class to be derived from two or more classes inheriting members of all parents
- Collisions (such as the use of the same variable name in two or more parents) must be resolved
- Java does not support multiple inheritance (Some other languages such as C++ do)
- In Java, a class can have only one direct parent
- The use of interfaces in Java gives us most of the benefits of multiple inheritance without the problems

# 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 class of the object used to execute an overridden method determines which version of the method is invoked
- In the child, the method in the parent class can be invoked explicitly using the `super` reference
- If a method is declared in the parent class with the `final` modifier, it cannot be overridden

# Overriding Variables

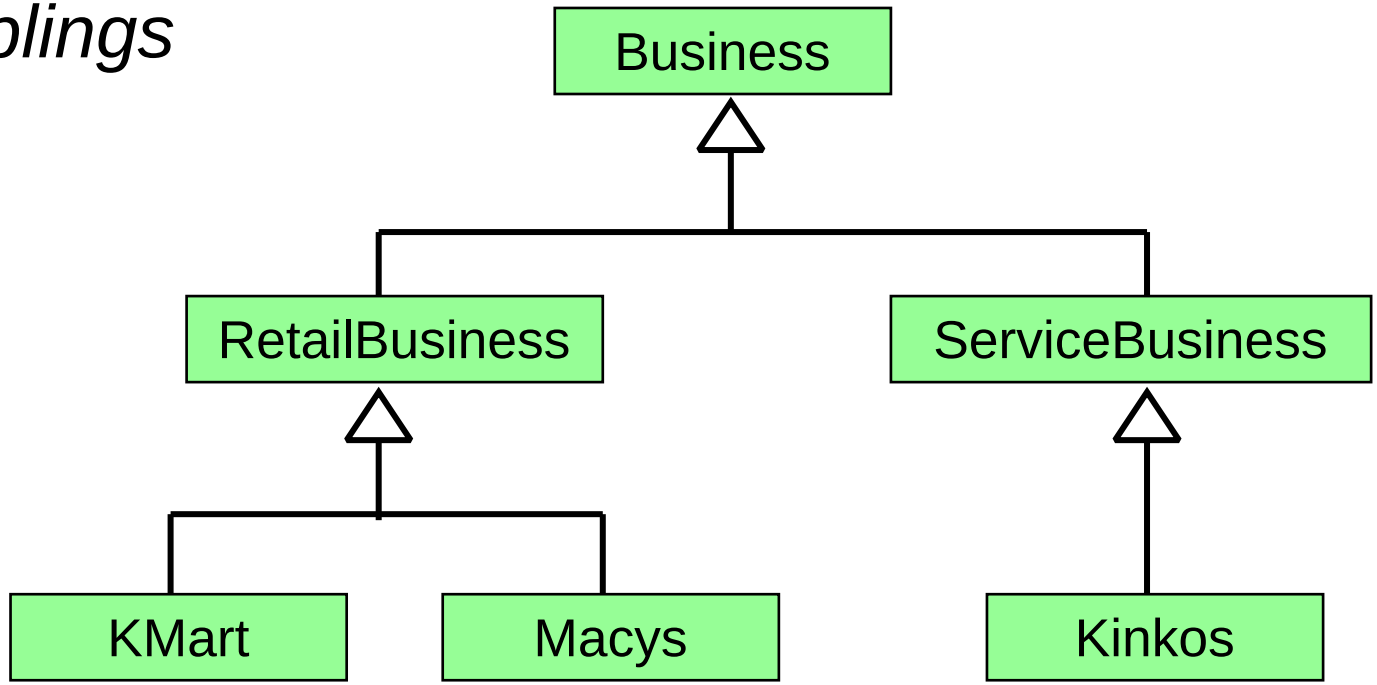
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

# Overloading vs. Overriding

- Overloading deals with multiple methods with the same name in the same class that have different signatures (different parameter lists)
- Overriding deals with two methods (one in a parent class and one in a child class) that have the same signature (same parameter list)
- Overloading lets you define a similar operation in different ways (using different input parameters)
- Overriding lets you redefine a parent’s method in a child (using the same input parameters)

# 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*



# Class Hierarchies

- A child class inherits from all its ancestor classes
- An inherited variable, constant, or method is passed continually down the line (unless it is declared *private*)
- Common features should be put as high in the hierarchy as is reasonable
- There is no single class hierarchy that is appropriate for all situations

# 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
- 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
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class along with other information (e.g. the address of its location in memory)

```
System.out.println(new Object());
java.lang.Object@952905
```

- Every time we define the `toString` method, we are actually overriding the inherited definition

# The Object Class

- The `equals` method of the `Object` class returns true if the 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 the 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

# Abstract Classes

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

```
public abstract class Product
{
    // 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 some non-abstract methods with their 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 all 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 class hierarchy that are too generic to instantiate