04 Inheritance

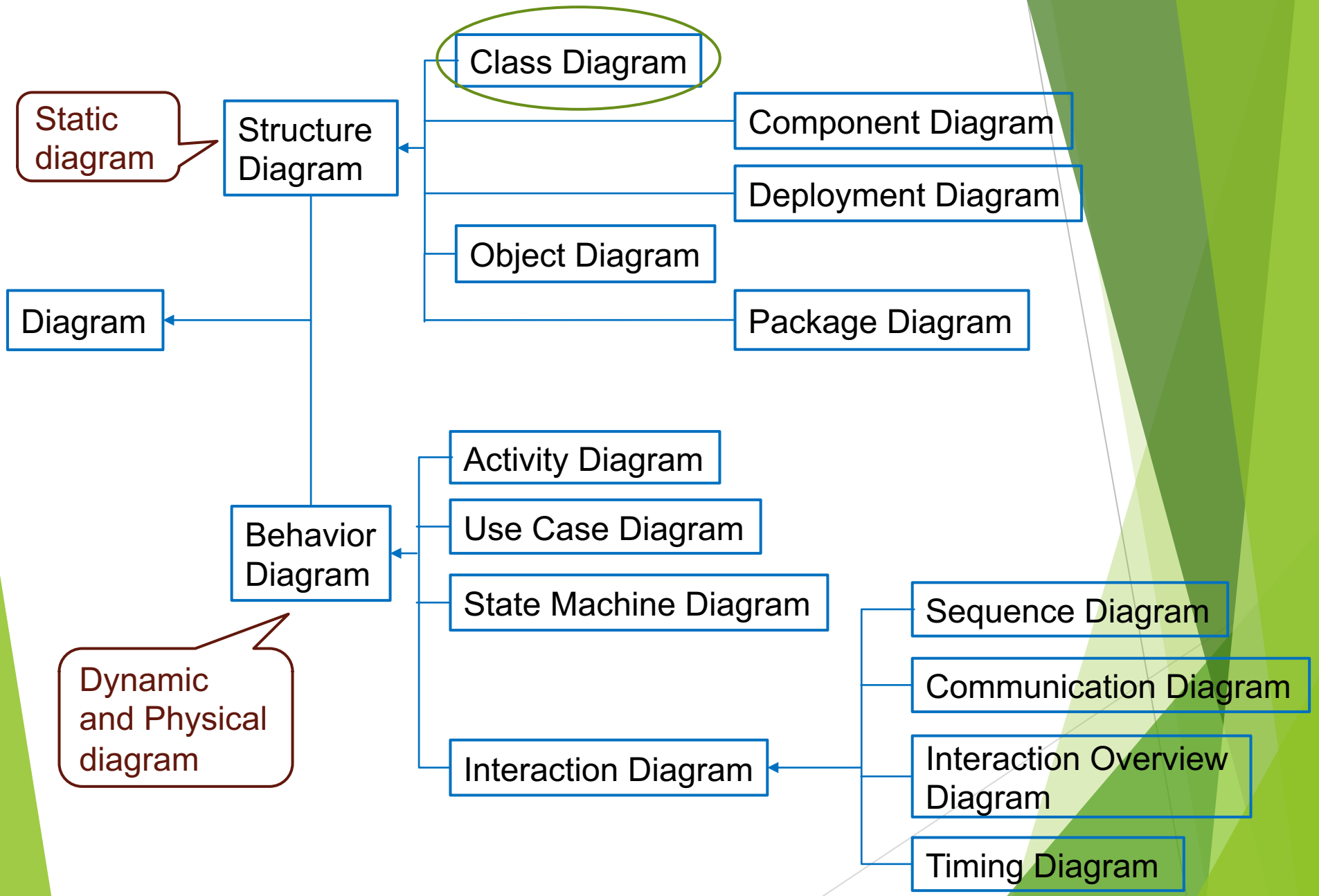
Introduction to OOA OOD and UML

2022 Spring

College of Information Science and Engineering

Ritsumeikan University

Yu YAN


Class Diagram
Static diagram
Structure Diagram
Component Diagram
Deployment Diagram
Object Diagram
Diagram
Package Diagram
Activity Diagram
Behavior Diagram
Use Case Diagram
State Machine Diagram
Dynamic and Physical diagram
Sequence Diagram
Communication Diagram
Interaction Diagram
Interaction Overview Diagram
Timing Diagram

Variable Types

- When we introduce a field, we need to decide what type it is: a primitive or a class:
  - The primitive could include:
    - boolean {true, false},
    - byte {0~255, can be stored in 8-bit byte},
    - char {value from 0 to 255, interpreted as characters}
    - double {4.9E-324},
    - float {1.4E-45, 3.4028235E38, 1.79769313486231570E308},
    - int {-2147483648, 2147483647, stored in 4 bytes},
    - long {-9223372036854775808, 9223372036854775807, 8 bytes},
    - short {-32768, 32767, stored in 2 bytes}
  - Classes that we ourselves are designing
  - Classes from the pattern and frameworks that we have chosen to use

Visibility of Fields

- As well as providing the names and types of fields, we must declare their visibility
- The visibility of a field specifies which pieces of code are allowed to read or modify the value
  - Private (shown by "–" in UML): Only visible with in the defining class
  - Package (shown by "~" in UML): Visible within the defining class and to all classes in same package
  - Protected (shown by "#" in UML): Visible within the defining class, to all classes in the same package, and to all subclasses of the defining class (whether inside or outside the package)
  - Public (shown by "+" in UML): Visible everywhere

Visibility of Messages

- Visibilities can be applied to messages too:
  - Public: if it's part of the interface of the package
  - Package: if it's implementation code to be used by the class itself and by classes in the same package
  - Protected: if it's implementation code to be used by the class itself, by its subclasses and by classes in the same package
  - Private: if it's implementation code for use by this class only (which decreases coupling and allows the compiler to do more optimizations, as with fields)

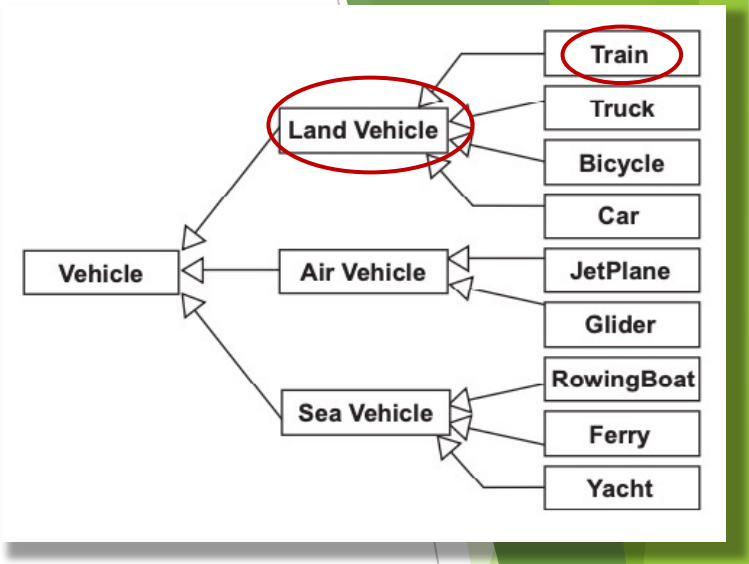
Accessors

- Accessor messages come in two varieties:
  - getters: it returns the value of a field
  - setters: it sets the field to a new value
- It is a good idea to provide accessor messages for fields:
  - Accessors allow us to centralize access to fields, making maintenance easier
  - Accessors are also easy for a compiler to optimize

```
private int count;

public int getCount() {
   return count;
}
public void setCount(int c) {
   count = c;
}
...
```

- An example of class hierarchy:
  - Specialization/Generalization: *Train* is more specialized than *LandVehicle*. *LandVehicle* is more generalized than *Train*.
  - Inheritance: *Train* inherits characteristics of *LandVehicle.*
  - Parent/child: *LandVehicle* is the **parent** of *Train*; *Train* is a **child** of *LandVehicle*.
  - Superclass/subclass: *LandVehicle* is the **superclass** of *Train*; *Train* is a **subclass** of *LandVehicle.*
  - Base/derived: *LandVehicle* is the **base** from which *Train* is **derived.**



Outline

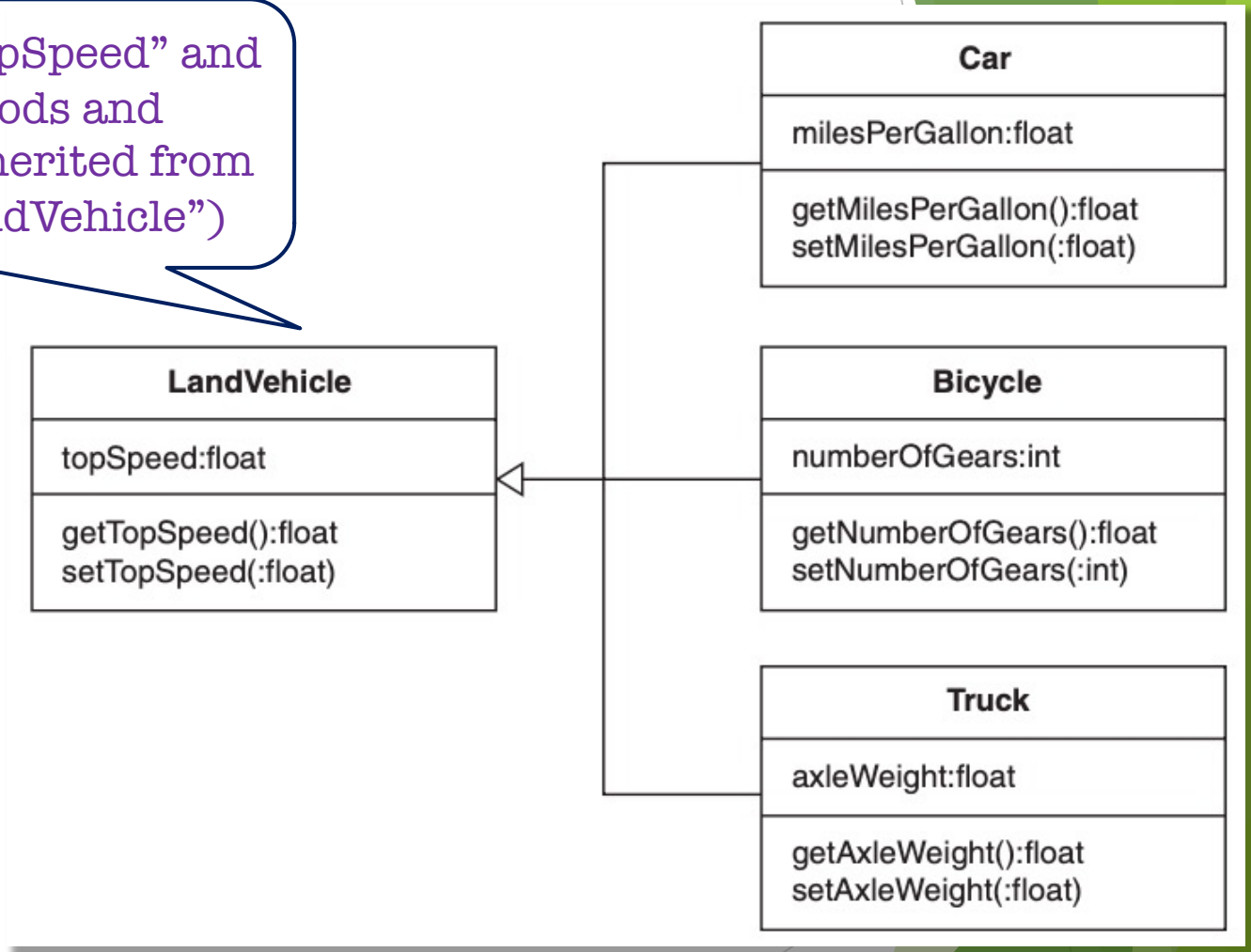
- The Definition of Inheritance
- Case Study: Design a Class Hierarchy
- Abstract and Concrete Classes
- Case Study: Implement a Stack Class
- Multiple Inheritance
- Summary and Class Vacabularies
- Exercise 04

What Is Inheritance?

- Inheritance allows that a class gets some of its characteristics from a parent class and then adds unique features of its own
  - The characteristics of a parent class (superclass) can be shared with the whole child classes (subclasses).
  - A subclass inherits all of the fields, messages and methods from its superclass.
- From a programming point of view, we want inheritance because:
  - It benefits both the developing team and other developers who might want to reuse code.
  - We will have less code to write by using it. (Information and behavior in one class can be shared in related subclasses).
  - It is natural. This is one of the prime motivations for object orientation in the first place.

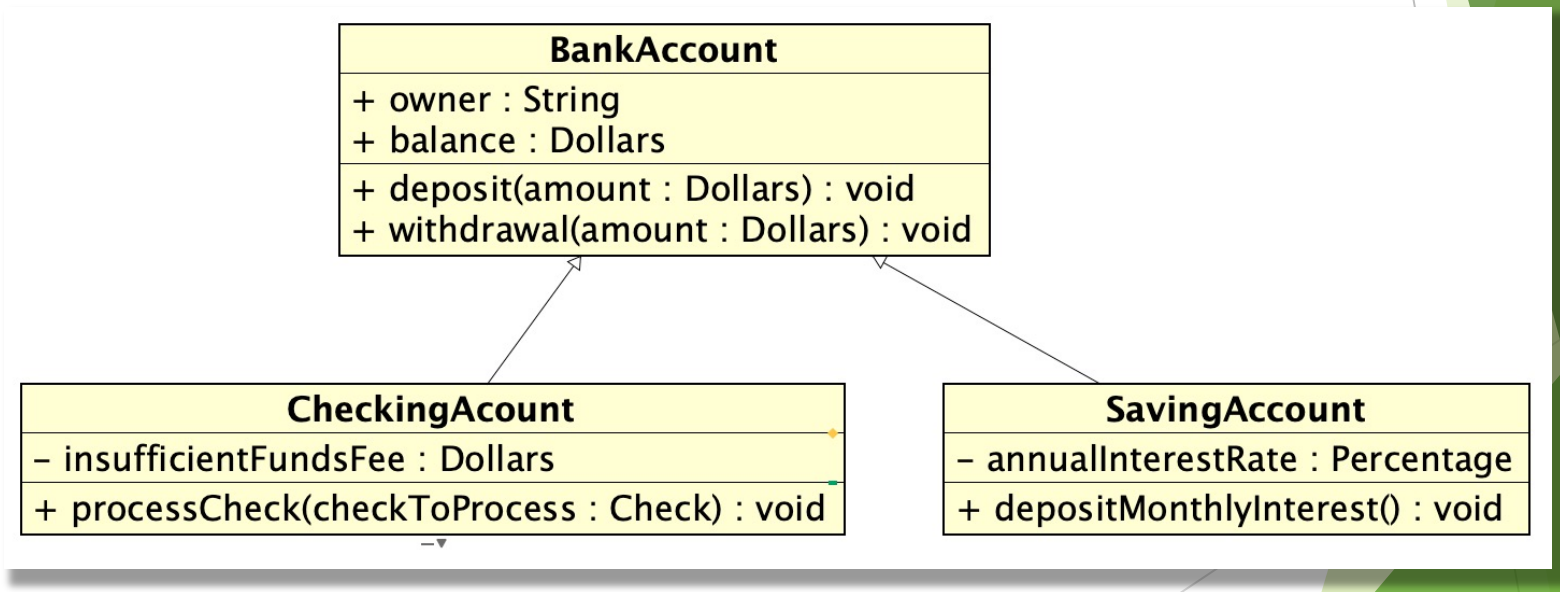
An Example of Inheritance

"car" can use "getTopSpeed" and "setTopSpeed" methods and "topSpeed" field (inherited from its parent class "LandVehicle")



UML Rules for Inheritance

- Connector type "generalization" is used.
- Fields and methods listed in a superclass will not be listed in its subclasses, even subclasses use them.
  - If some fields or methods are used only in one subclass, they should not be used in its superclass.



Outline

- The Definition of Inheritance
- Case Study: Design a Class Hierarchy
- Abstract and Concrete Classes
- Case Study: Implement a Stack Class
- Multiple Inheritance
- Summary and Class Vacabularies
- Exercise 04

A Collection

- A collection is data structure that holds objects. There are four styles of collection:
  - List: A collection that keeps all of its objects in the order in which they were inserted.
  - Bag: A collection that doesn't keep its objects in order.
  - LinkedList: A collection that keeps its objects in order using an implementation of a sequence of objects in which each object points to the next in the sequence.
  - ArrayList: A collection that keeps its objects in order using an array, a sequence of adjacent memory locations.

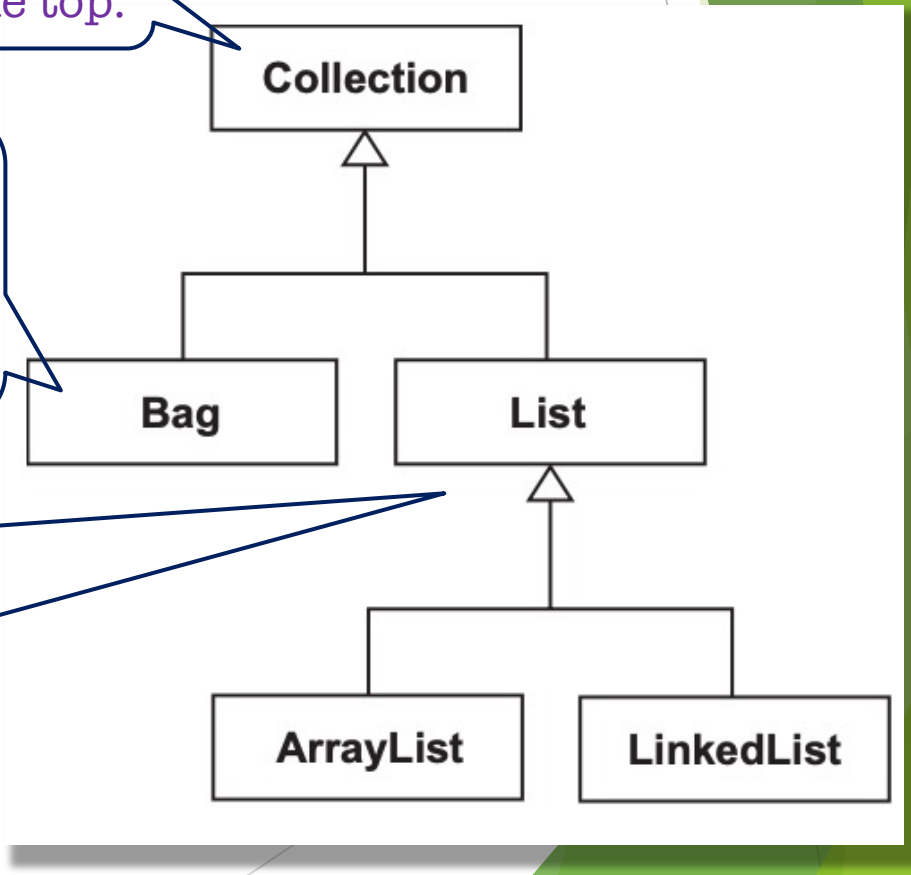
Problem: we would like to place the four styles of collections into an inheritance hierarchy

- Step #01: decide the class hierarchy

Because all styles of collections are collections, "collection" goes to the top.

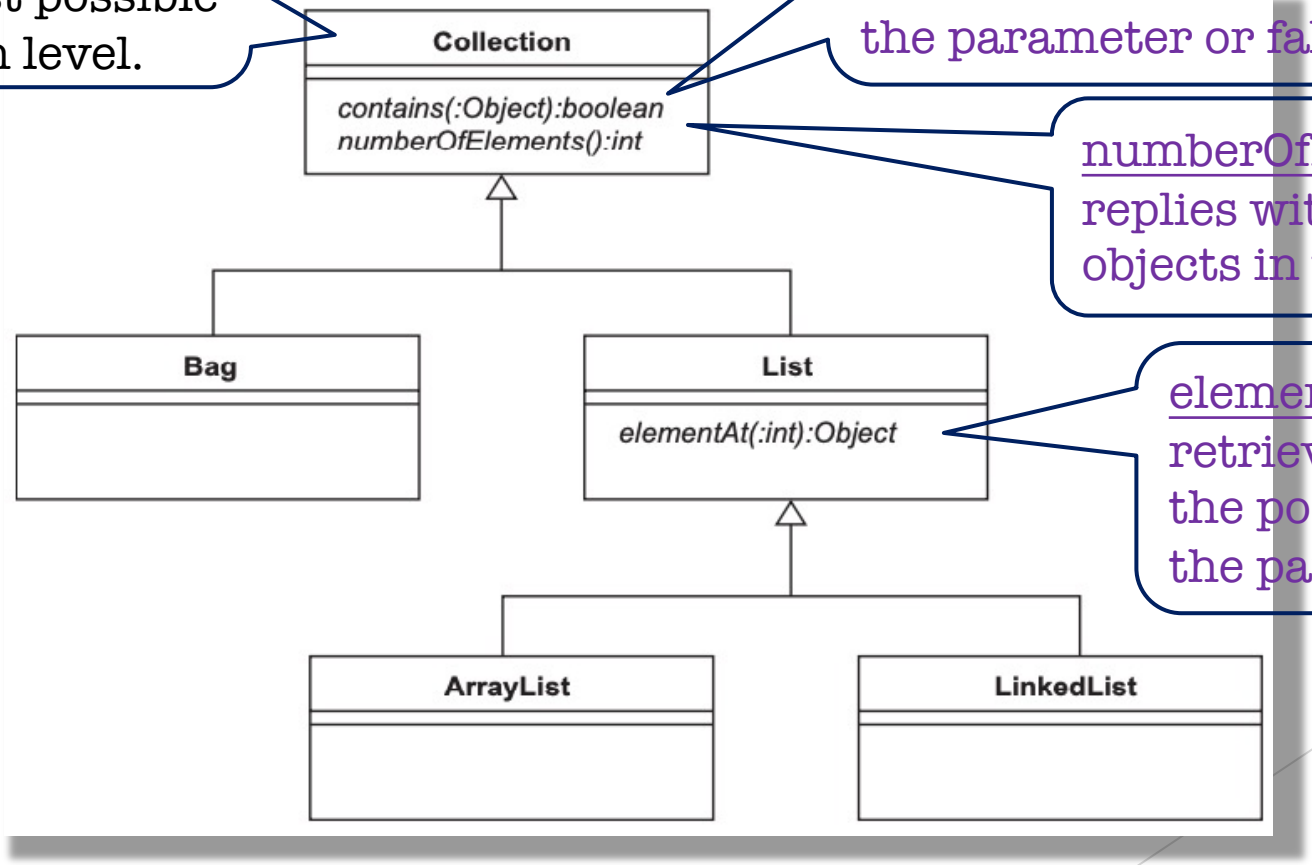
"List", "ArrayList" and "LinkedList" keep their objects in order, but "Bag" doesn't.So "Bag" should be in a separate branch from others.

"List" has no comment to internal implementation, but "LinkedList" and "ArrayList" do. So "List" is the superclass of "ArrayList" and "LinkedList.".



- Step #02: Add messages to the hierarchy

Try to place them at the most possible high level.



contains(:Object):boolean searches for an object in the collection and returns true if the receiver contains the parameter or false otherwise.

numberOfElements():int replies with the number of objects in the collection.

elementAt(:int):Object retrieves the object at the position indicated by the parameter.

Outline

- The Definition of Inheritance
- Case Study: Design a Class Hierarchy
- Abstract and Concrete Classes
- Case Study: Implement a Stack Class
- Multiple Inheritance
- Summary and Class Vacabularies
- Exercise 04

An Abstract Class

- An abstract method has no method body, but a concrete method does
  - Usually, abstract methods are written in *italic* in UML diagrams
    One line code example (Java):

```
public abstract int numberOFElements();
```

- An abstract class is a class with at least one abstract method
  - The abstract method may be introduced on the class itself, or it may be inherited from a superclass.

Understandings of An Abstract Class

- An interesting analogy: peeling a piece of fruit.

We know that we can take the skin off any piece of fruit, but we can't describe how to do in a way that will work well for every variety.

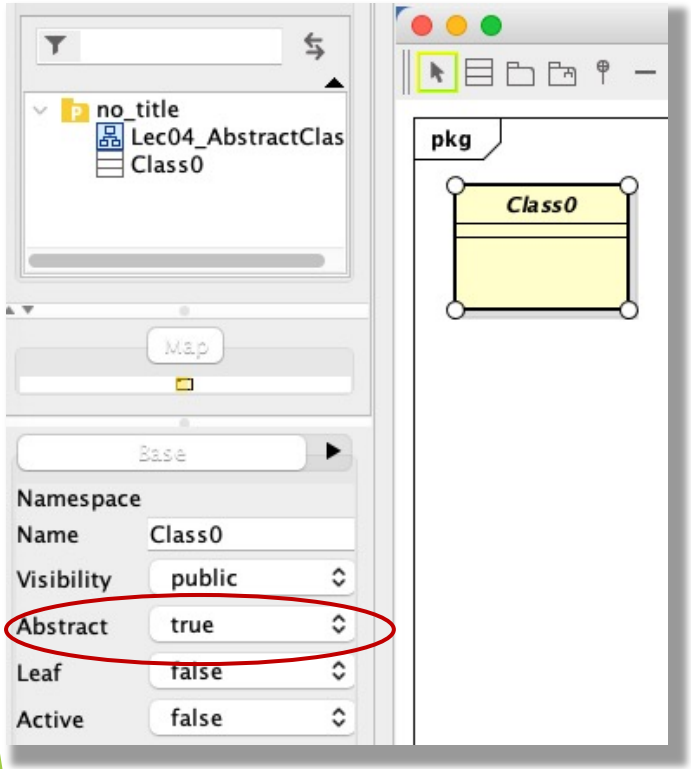
In the real world: if I give you some money and ask you to go to the shop to buy a piece of fruit, you're likely to ask "What kind of fruit would you like", because you need a concrete request.

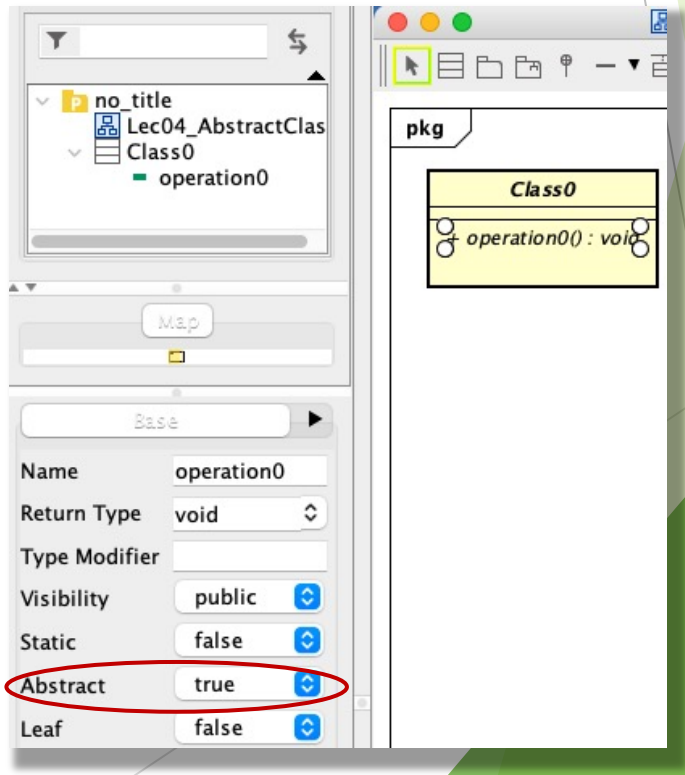
- Most OO languages regards creating instances of abstract classes as illegal.
- Most superclasses are abstract classes
- Subclasses are supposed to have some extra knowledge and behavior than superclasses that they inherit

In the real world: let's say "Fruit" is the superclass of "Orange". If there is no other fruits but orange existing, why don't we just call orange as fruit?

Implement an Abstract Class via UML

- Set a class as “abstract” by setting “Abstract” of the class as “true”.
- Set a method as “abstract” by setting “Abstract” of the method as “true”.





Redefining Methods

- Methods/Abstracts declared in a superclass can be redefined in its inherited classes. In a redefined method:
  - Form #01: the declaration of the method stays the same, but lines of codes are replaced. For example, there is one more parameter in the method.
  - Form #02: the specification of the method are changed. For example, a subclass can turn a private method into a public method.
- One thing we must be careful when we try to redefine a method is:
  - We are just adding additional knowledge and behaviors in the subclass, and we should make sure that the superclass definition still does everything it used to – this increases code sharing and simplifies maintenance.
- Every OO language allows a redefined method to invoke the one on its superclass.

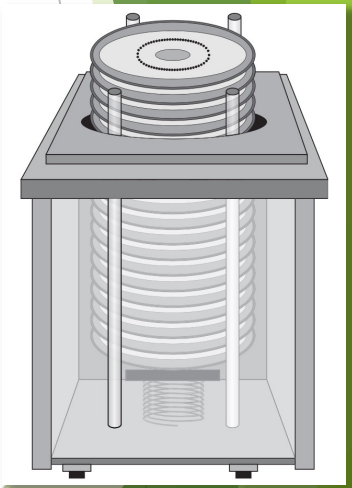
Outline

- The Definition of Inheritance
- Case Study: Design a Class Hierarchy
- Abstract and Concrete Classes
- Case Study: Implement a Stack Class
- Multiple Inheritance
- Summary and Class Vacabularies
- Exercise 04

A Stack

- One example of the stack data structure is "a spring-loaded plate dispenser"
  - "Push an object onto the top", "peek at the top object, see if the stack is empty", and "pop an object off the top"

- Thus, in order to implement a stack, we need the following four messages/abstracts:
  - push(): to add an object to the top of the stack.
  - peek(): Object to return the object on the top of the stack.
  - isEmpty():boolean to return true if there are no objects on the stack.
  - pop():Object to remove an object from the top of the stack and return it.



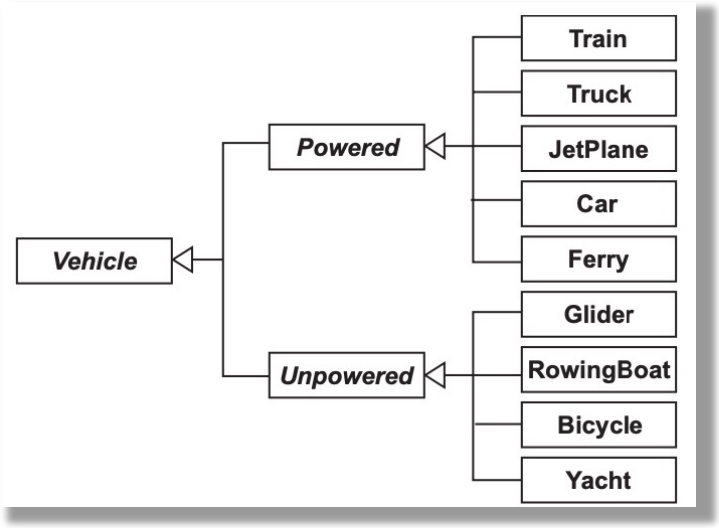
https://www.youtube.com/watch?v=ICraN2GvIuQ

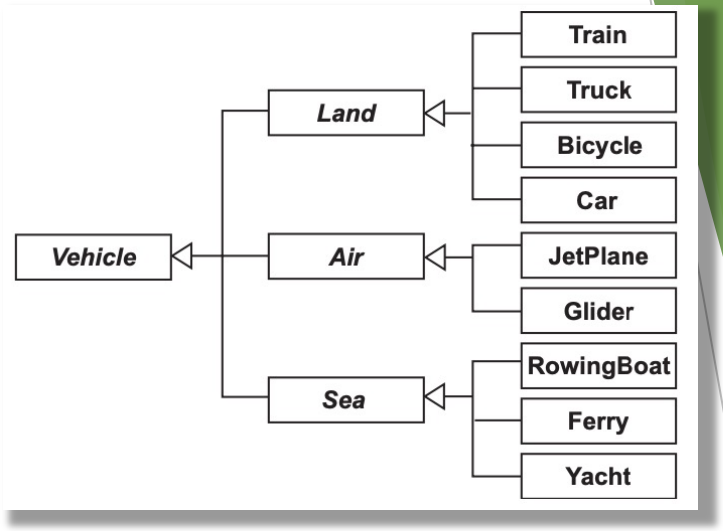
Reuse the *LinkedList* Class

- There are four messages/abstracts that should be able to reuse:
  - addElement(): add an object to the end of the list.
  - lastElement(): Object to return the object at the end of the list.
  - numberOfElement: int to return the number of objects in the list.
  - removeLastElement(): to remove the object at the end of the list.
- In order to incorporate the existing *LinkedList* behavior into our *Stack* class, we could use inheritance.
- The *Stack* class will be a subclass of the *LinkedList* class.

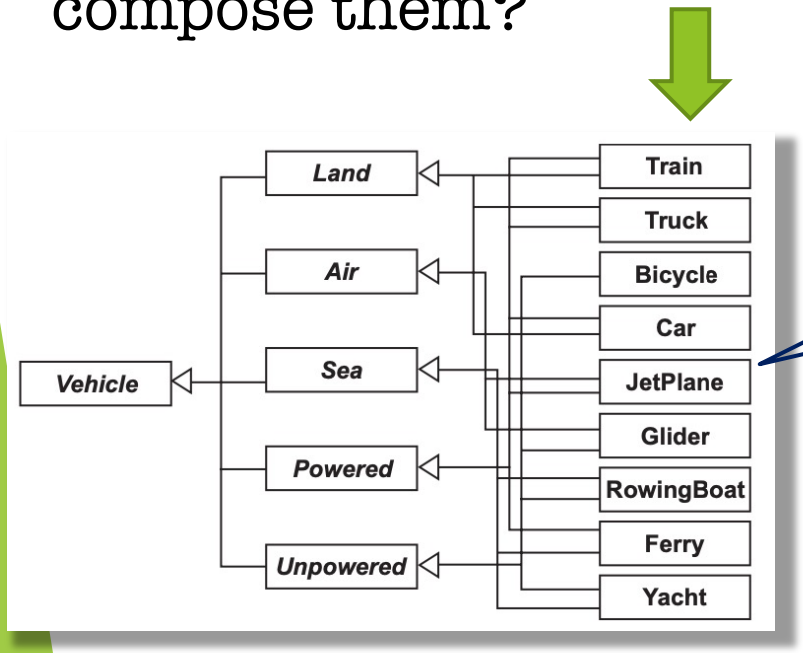
Outline

- The Definition of Inheritance
- Case Study: Design a Class Hierarchy
- Abstract and Concrete Classes
- Case Study: Implement a Stack Class
- Multiple Inheritance
- Summary and Class Vacabularies
- Exercise 04





- Two types of classification give different diagrams, how to compose them?



A subclass can have more than one parent (multiple inheritance). But it is more difficult for design and understanding.

- Here's a summary of the multiple inheritance facilities available in common languages:
  - Eiffel provides *straightforward multiple inheritance*, *plus private inheritance and mix-in inheritance*.
  - Smalltalk provides only single inheritance.
  - C++ provides some multiple inheritance facilities.
  - Java provides single inheritance for classes but multiple inheritance for interfaces(abstract classes that have no methods).

Outline

- The Definition of Inheritance
- Case Study: Design a Class Hierarchy
- Abstract and Concrete Classes
- Case Study: Implement a Stack Class
- Multiple Inheritance
- Summary and Class Vacabularies
- Exercise 04

Class Vocabularies

Inheritance, Abstract and Concrete Methods, Abstract and Concrete Classes, Redefinition, Multiple Inheritance

Summary

- The concept of inheritance in OOP and UML design
- The definitions of abstract and concrete classes

Outline

- The Definition of Inheritance
- Case Study: Design a Class Hierarchy
- Abstract and Concrete Classes
- Case Study: Implement a Stack Class
- Multiple Inheritance
- Summary and Class Vacabularies
- Exercise 04

Exercise 04

- Deadline: **2022/05/12 (Thur.) 9:00**
- Please submit your answer file to “Exercise 04” under “Assignments” tab in Manaba +R
- Please put all of the answers in one “.pdf” file. The file name will be “**UML_Ex04_Your name.pdf**”
- The maximum points for “Exercise 04” will be **3p**

Tasks

- Task #01: In UML diagrams, how are abstract classes distinguished from concrete classes? Please choose only one option (1p)
  - a) Concrete classes are shown as boxes with dashed outlines.
  - b) Labels on abstract classes are shown in italics.
  - c) Labels on concrete classes are shown in italics.
  - d) Abstract classes are shown as boxes with dashed outlines.
- Task #02: Why is the ability to redefine a method important in OOP? You can choose multiple options (2p)
  - a) Because it allows us to add extra work to a method.
  - b) Because it allows us to introduce abstract methods that are redefined as concrete methods
  - c) Because it allows us to provide a more accurate or faster definition in a subclass
  - d) Because it allows us to disable a method in a subclass
  - e) Because it allows us to change the meaning of a method

Tasks

- Task #03 (5p): Please draw a class diagram (with Astah UML) to describe a Stack and a LinkedList as shown in slide #22 and slide #23. Export your diagram as a PNG/JEPG image and insert the diagram image in your report.
- Tips:
  - ✓ The relationship between the Stack and the LinkedList are supposed to be "Inheritance".
  - ✓ Think about the fields for both the classes Stack and LinkedList by yourself. The fields are supposed to have their names, types and visibilities.
  - ✓ Think about the messages for both the classes Stack and LinkedList by yourself. The messages are supposed to have their names, parameters(names and types), return value types and visibilities.
  - ✓ For both the classes Stack and LinkedList, for at least one of the fields, the visibility is supposed to be "private". In addition, for each "private" field, there are supposed to be a set of "getter" and "setter" methods. Be careful about the parameters and return value types of the "getters" and "setters".