Classes

```
[ClassModifiers] class ClassName
        [extends SuperClass]
        [implements Interface$_1$, Interface$_2$, …] {
   ClassMemberDeclarations
}
```

- Class modifiers
  - visibility: package versus `public`
  - `abstract`
  - `final`
- `extends` clause specifies the superclass
- `implements` clause specifies the interfaces being implemented

Design principles

Design principles in this course and used by the design patterns

- Use abstraction whenever possible
  - introduce (abstract) superclass (or interfaces) in order to implement or define common behavior
  - nothing should be implemented twice
- Program to an interface, not an implementation
- Favor aggregation/composition over inheritance
  - delegation
- Design for change and extension

Inheritance

- Parent/child, superclass/subclass
- Instances of child inherit data and behavior of parent
- `implements`
  - inheritance of specification
- `extends`
  - Subclassing
    - a subclass *extends* the capability of its superclass; the subclass inherits features from its superclass and adds more features
    - every instance of a subclass is an instance of the superclass
  - inheritance of code and specification
  - overriding
    - Polymorphism
- Subclass as an extension of behavior (specialization)
- Subtype as a contraction of value space (specialization)

Overriding versus Overloading

- Overloading
  - methods
  - same name, different signatures
  - same class or subclass
  - effect – multiple methods with same name
  - **do not overuse** (readability of programs)
  - overloading should be used only in two situations:
    1. When there is a general, non-discriminative description of the functionality that fits all the overloaded methods.
    2. When all the overloaded methods offer the same functionality, with some of them providing default arguments.
- Overriding
  - instance methods
  - same name, signature and result type
  - in subclass
  - effect – replacement implementation
    - access superclass version via `super`

Forms of inheritance

- Inheritance for specification
  - parent provides specification
    - abstract classes
    - interfaces
  - behaviour implemented in child
- Inheritance for extension
  - adding behaviour
- Inheritance for specialization
  - child is special case
  - child overrides behavior to extend

- Inheritance for construction
  - inherit functionality
  - *ad hoc* inheritance
- Inheritance for limitation
  - restricting behavior
- Inheritance for combination
  - combining behaviors
  - multiple inheritance
  - only through interfaces in Java

Inheritance for Specification: Java interface

Ch.8.4, Budd: Understanding Object-Oriented Programming with Java

```
interface ActionListener {
  public void actionPerformed (ActionEvent e);
}


class CannonWorld extends Frame {
  …

    // a fire button listener implements the action
    // listener interface
  private class FireButtonListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
    … // action to perform in response to button press
    }
  }
}
```

Inheritance for Specification: abstract class

Ch.8.4, Budd: Understanding Object-Oriented Programming with Java

```
public abstract class Number {
   public abstract int intValue();
   public abstract long longValue();
   public abstract float floatValue();
   public abstract double doubleValue();
   public byte byteValue()
        { return (byte) intValue(); }
   public short shortValue()
        { return (short) intValue(); }
}
```

Inheritance for Extension

Ch.8.4, Budd: Understanding Object-Oriented Programming with Java

```
class Properties extends Hashtable {
   …
   public synchronized void load(InputStream in)
        throws IOException {…}

   public synchronized void save(OutputStream out,
        String header) {…}

  public String getProperty(String key) {…}

  public Enumeration propertyNames() {…}

  public void list(PrintStream out) {…}
}
```

Inheritance for Specialization

```
public class MyCar extends Car {
   …
   public void startEngine() {
        motivateCar();
        super.startEngine();
   }
   …
}
```

Inheritance for Construction

Ch.8.4, Budd: Understanding Object-Oriented Programming with Java

```
class Stack extends LinkedList {

    public Object push(Object item)
         { addElement(item); return item; }

   public boolean empty()
         { return isEmpty(); }

   public synchronized Object pop() {
         Object obj = peek();
         removeElementAt(size() - 1);
         return obj;
    }

   public synchronized Object peek()
         { return elementAt(size() - 1); }
}
```

Inheritance for Limitation

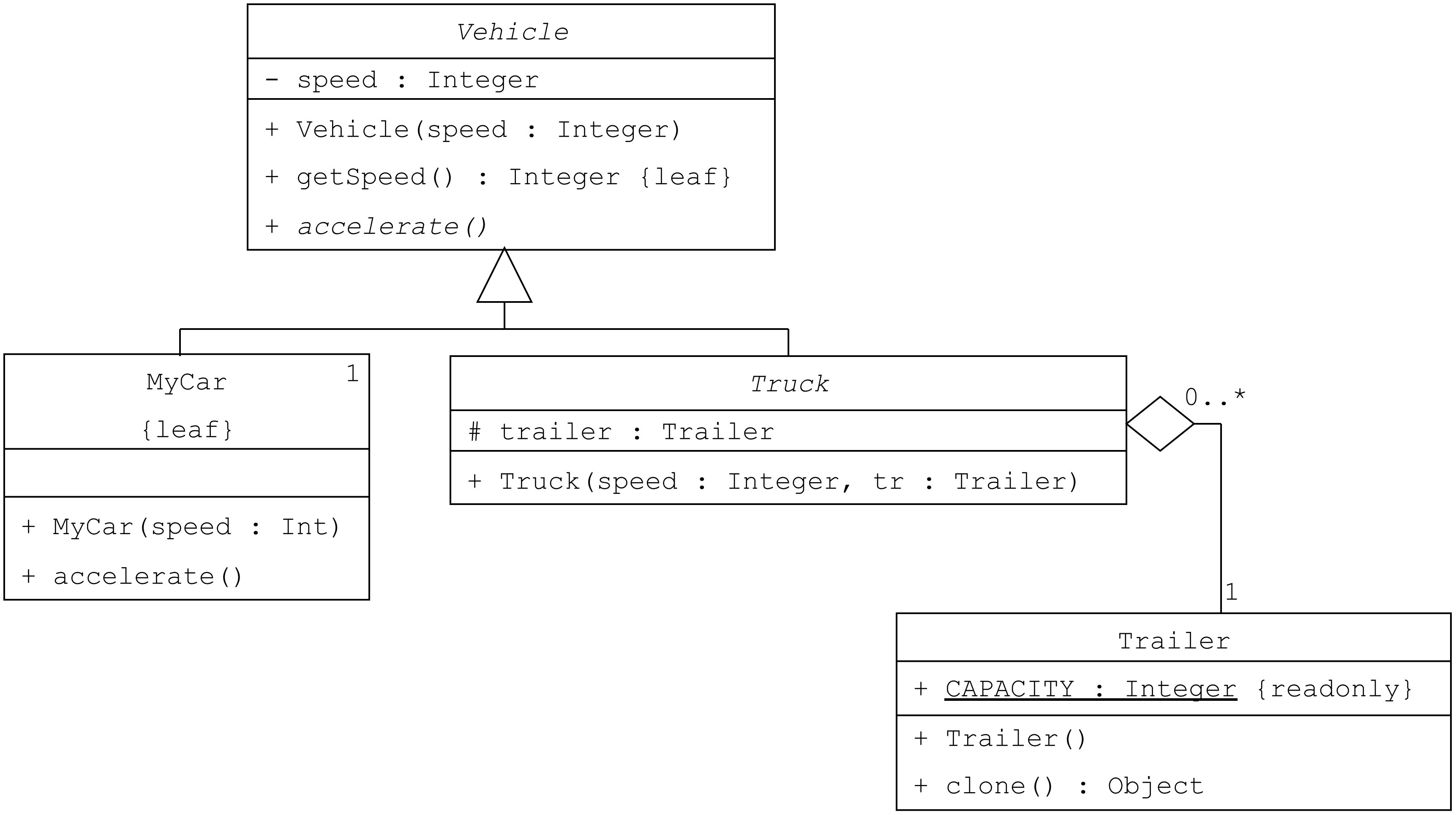
Ch.8.4, Budd: Understanding Object-Oriented Programming with Java

```
class Set extends LinkedList {
  // methods addElement, removeElement, contains,
  // isEmpty and size are all inherited from LinkedList

  public int indexOf(Object obj) {
    System.out.println(“Do not use Set.indexOf”);
    return 0;
  }

  public Object elementAt(int index) {
    return null;
  }
}
```

Inheritance for Combination

```
public class Mouse extends Vegetarian implements Food {
   …
   protected RealAnimal getChild() {
        …
   }
   …
   public int getFoodAmount() {
        …
   }
   …
}
```

UML Class diagrams



Vehicle

- speed : Integer

+ Vehicle(speed : Integer)

+ getSpeed() : Integer {leaf}

+ *accelerate()*

MyCar

{leaf}

1

+ MyCar(speed : Int)

+ accelerate()

Truck

# trailer : Trailer

+ Truck(speed : Integer, tr : Trailer)

0..*

1

Trailer

+ CAPACITY : Integer {readonly}

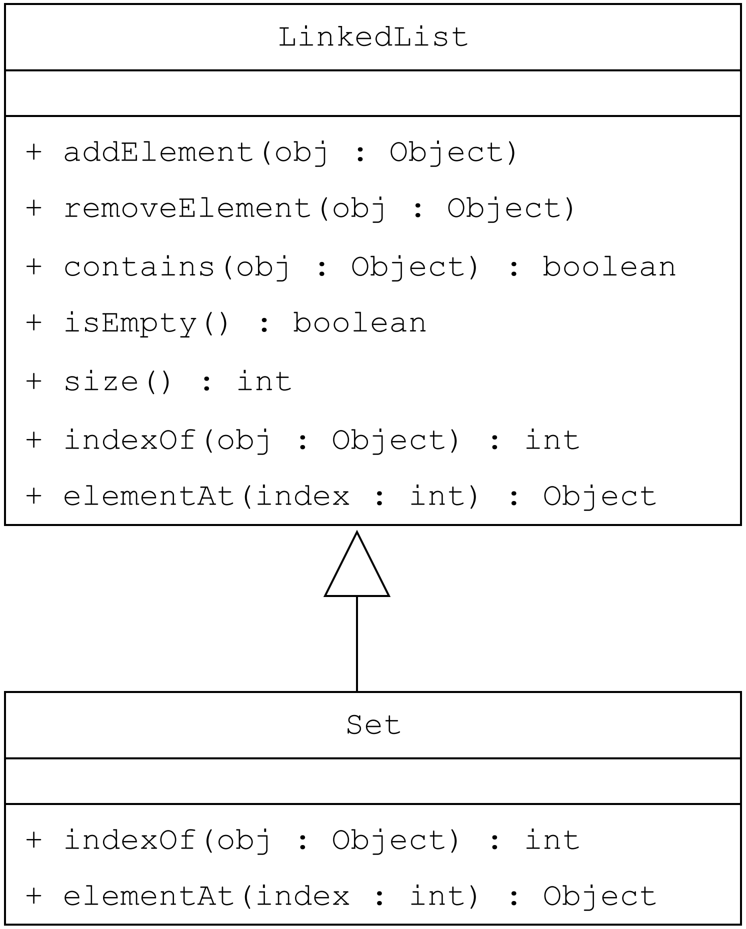
+ Trailer()

+ clone() : Object

Composition

- Composition ≡ *has-a* relationship (strong ownership)
- Inheritance ≡ *is-a* relationship
- Inheritance versus composition
  - desire to reuse existing implementation
  - subclass inherits specification and all methods and variables
  - composition allows selective reuse

UML Class diagrams
Inheritance versus Composition



LinkedList

+ addElement(obj : Object)

+ removeElement(obj : Object)

+ contains(obj : Object) : boolean

+ isEmpty() : boolean

+ size() : int

+ indexOf(obj : Object) : int

+ elementAt(index : int) : Object

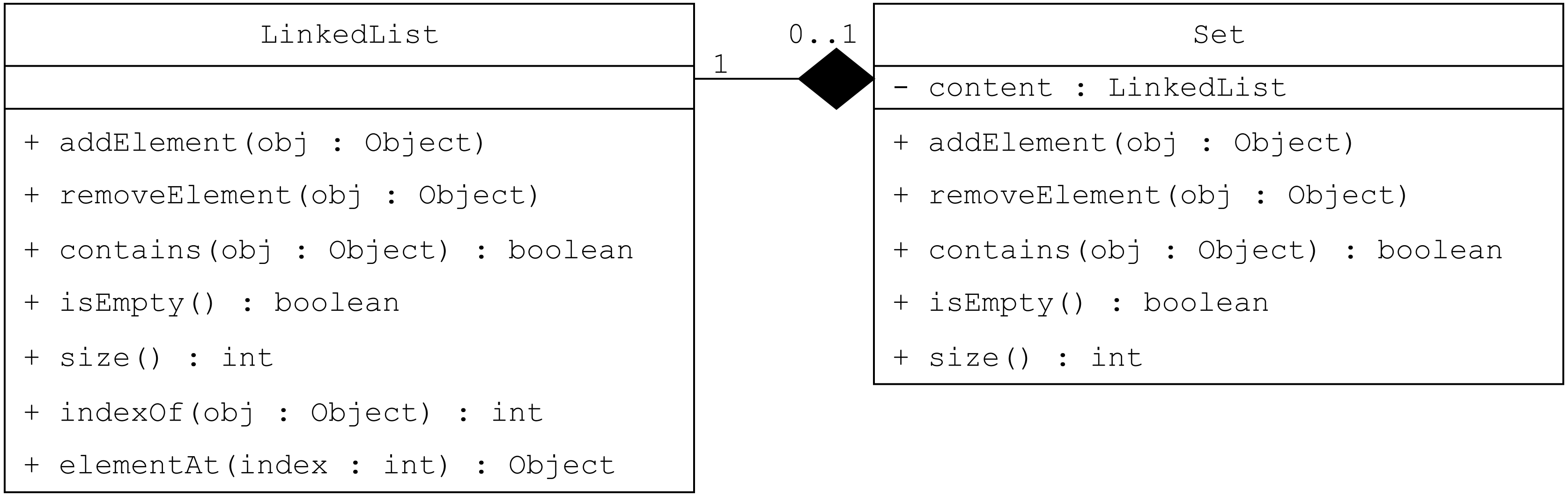
Set

+ indexOf(obj : Object) : int

+ elementAt(index : int) : Object

```
public int indexOf(Object obj) {
 System.out.println(“Do not use Set.indexOf”);
 return 0;
}
public Object elementAt(int index) {
 System.out.println(“Do not use Set.elementAt”);
 return null;
}
```

UML Class diagrams
Inheritance versus Composition



```
LinkedList

+ addElement(obj : Object)
+ removeElement(obj : Object)
+ contains(obj : Object) : boolean
+ isEmpty() : boolean
+ size() : int
+ indexOf(obj : Object) : int
+ elementAt(index : int) : Object
```

```
Set
- content : LinkedList
+ addElement(obj : Object)
+ removeElement(obj : Object)
+ contains(obj : Object) : boolean
+ isEmpty() : boolean
+ size() : int
```

```
public void addElement(Object obj) {
   content.addElement(obj);
}
…
public int size(){
   return content.size();
}
```