Programming in Java

Advanced Class Features

Hua Huang, Ph.D.

Spring 2019

Objectives

- Create **static** **variables**, **methods**, and **initializers**
- Create **final** **classes**, **methods**, and **variables**
- Create and use **enumerated types**
- Use the **static import** statement
- Create **abstract** **classes** and **methods**
- Create and use an **interface**
- **Singleton** design pattern
- **Nested classes** and its usage

Relevance

- How can you create a **constant**?
- How can you declare data that is **shared** by all instances of a given class?
- How can you keep a class or method from being subclassed or overridden?

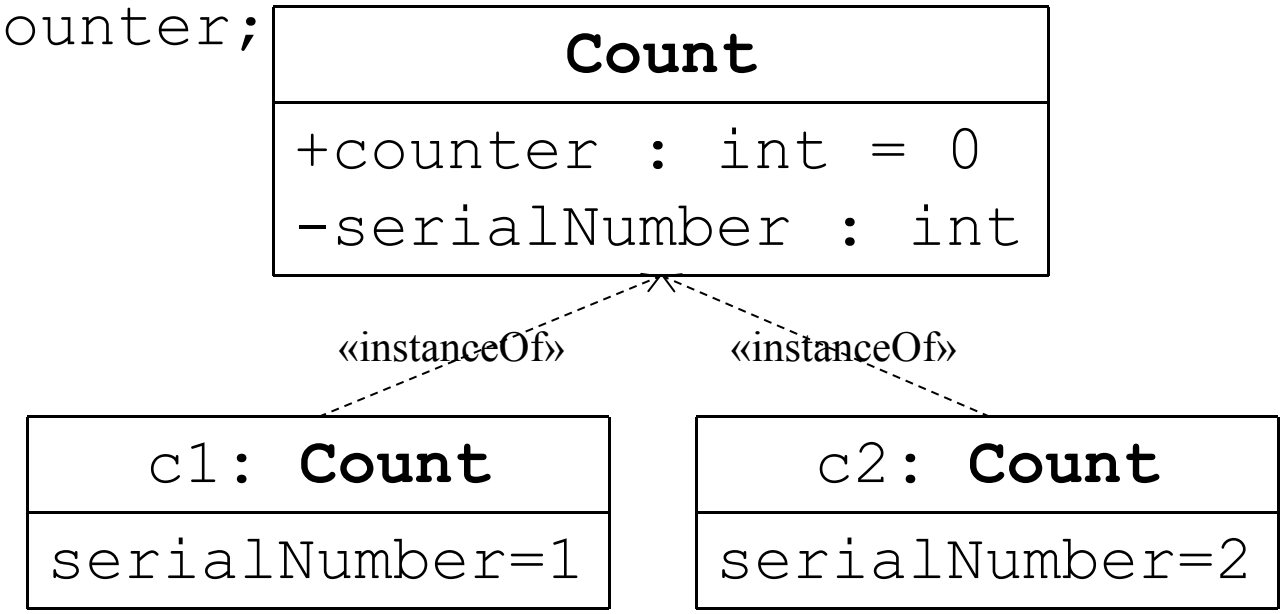
The `static` Keyword

- The **`static`** keyword is used as a **modifier** on **variables**, **methods**, and **nested classes**.
- The **`static`** keyword declares the attribute or method is **associated with the class** as a whole rather than any particular instance of that class.
- Thus **static members** are often called **class members**, such as class attributes or class methods.

Class Attributes

- Class attributes are shared among all instances of a class:

```
28    public class Count {
29    private int serialNumber;
30        public static int counter = 0;
31
32        public Count() {
33            counter++;
34            serialNumber = counter;
35        }
36    }
```



Class Attributes(Cont.)

- If the static member is `public`:

```
1     public class Count1 {
2       private int serialNumber;
3       public static int counter = 0;
4       public Count1() {
5         counter++;
6         serialNumber = counter;
7       }
8     }
```

- it can be accessed from outside the class without an instance:

```
1     public class OtherClass {
2       public void incrementNumber() {
3         Count1.counter++;
4       }
5     }
```

Class Methods

- You can create **static** methods:

```
1     public class Count2 {
2         private int serialNumber;
3         private static int counter = 0;
4
5         public static int getTotalCount(){
6             return counter;
7         }
8
9         public Count2() {
10            counter++;
11            serialNumber = counter;
12        }
13    }
```

Class Methods(Cont.)

- You can invoke **`static`** methods without any instance of the class to which it belongs:

```
1     public class TestCounter {
2       public static void main(String[] args) {
3         System.out.println("Number of counter is "
4         + Count2.getTotalCount());
5         Count2 counter = new Count2();
6         System.out.println("Number of counter is "
7         + Count2.getTotalCount());
8       }
9     }
```

Question: What is the output of the `TestCounter` program ??

Class Methods(Cont.)

- Static methods **cannot** access **instance variables**:

```
1     public class Count3 {
2       private int serialNumber;
3       private static int counter = 0;
4
5       public static int getSerialNumber() {
6         return serialNumber;
7       }
8     }
```

Question: Is there any problem with this program??

Static Initializers

- A class can contain code in a **static block** that does **not exist within a method body**.
- **Static block code executes once only**, when the class is loaded.
- Usually, a static block is used to initialize static (class) attributes.

Static Initializers(Cont.)

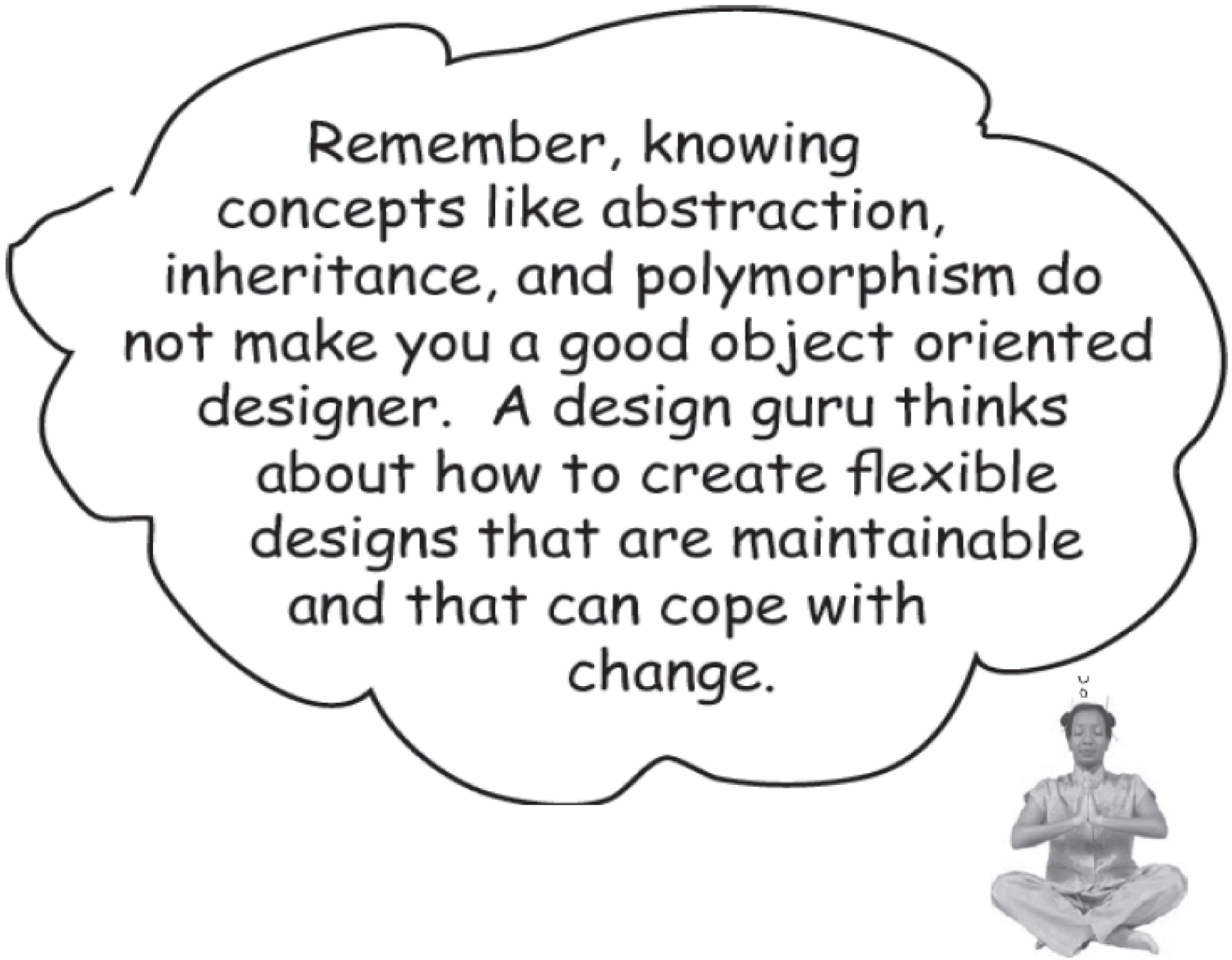
```
1     public class Count4 {
2       public static int counter;
3       static {
4         counter =
Integer.getInteger("myApp.Count4.counter").intValue();
5       }
6     }
```

```
1     public class TestStaticInit {
2       public static void main(String[] args) {
3         System.out.println("counter = "+ Count4.counter);
4       }
5     }
```

The output of the `TestStaticInit` program is:

```
java -DmyApp.Count4.counter=47 TestStaticInit
counter = 47
```

Design Patterns



Design Patterns(Cont.)

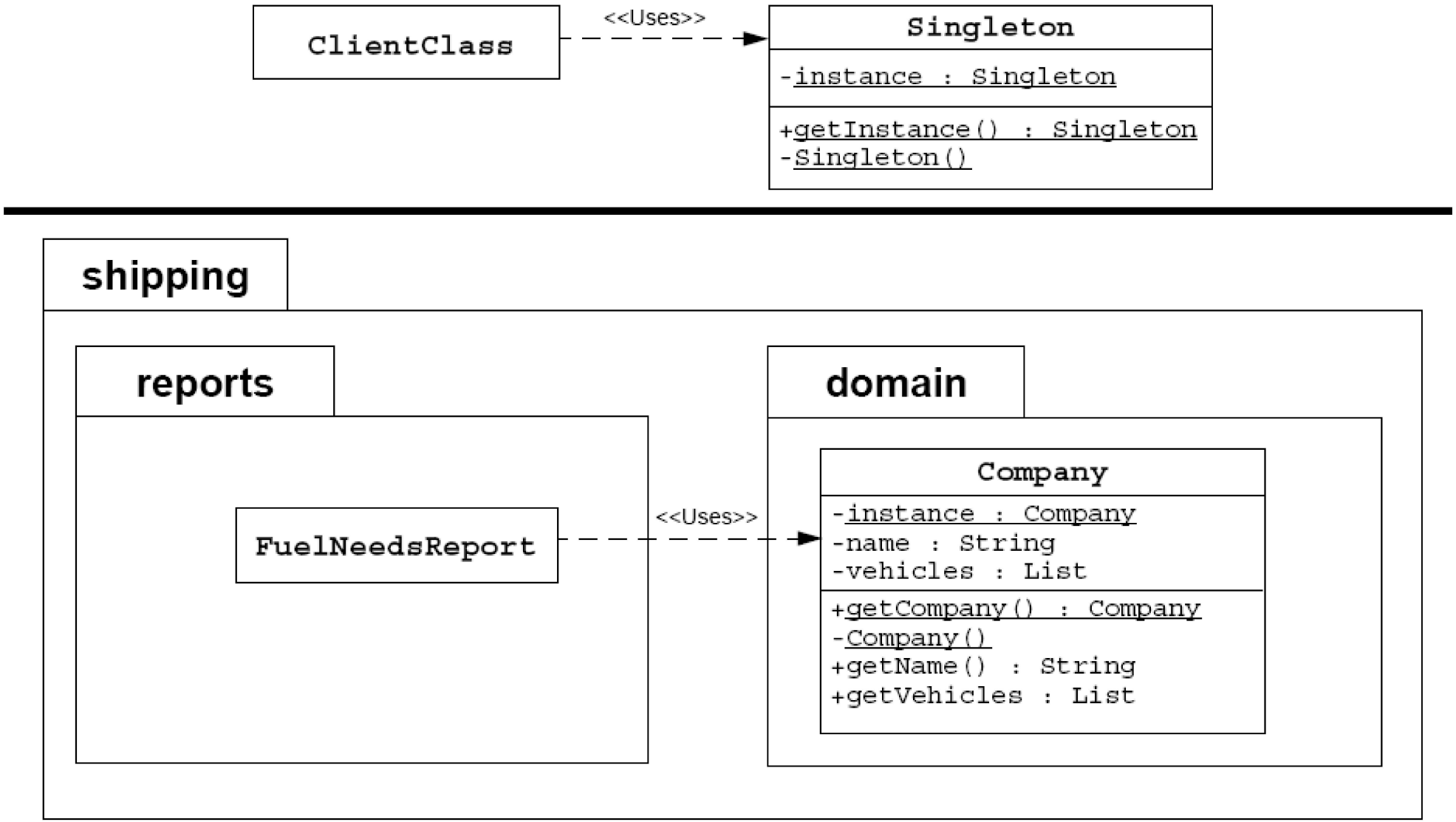
- Someone has already solved your problems.

  Instead of code reuse, with patterns you get **experience reuse**.

- The **Singleton** is the simplest in terms of its class diagram.

  Holds just a single instance!!

The Singleton Design Pattern



Implementing the Singleton Design Pattern

- The Singleton code:

```
1 package shipping.domain;
2
3 public class Company {
4     private static Company instance = new Company();
5     private String name;
6     private Vehicle[] fleet;
7
8     public static Company getCompany() {
9       return instance;
10    }
11
12    private Company() {...}
13
14    // more Company code ...
15 }
```

Implementing the Singleton Design Pattern(Cont.)

- Usage code:

```
1     package shipping.reports;
2
3     import shipping.domain.*;
4
5     public class FuelNeedsReport {
6         public void generateText(PrintStream output) {
7             Company c = Company.getCompany();
8             //use Company object to retrieve the fleet vehicles
9         }
10    }
```

The `final` Keyword

- You **cannot subclass a final class**.
- You **cannot override** a **final method**.
- A **final variable** is a **constant**.
- You can **set a final variable once only**, but that assignment can occur independently of the declaration; this is called a **blank final variable**.
  - A **blank final instance attribute** must be set in every **constructor**.
  - A **blank final method variable** must be set in the method body before being used.

Final Variables

- Constants are static final variables.

```
public class Bank {
   private static final double DEFAULT_INTEREST_RATE = 3.2;
   ... // more declarations
}
```

Blank Final Variables

```
1       public class Customer {
2
3       private final long customerID;
4
5       public Customer() {
6         customerID = createID();
7       }
8
9       public long getID() {
10        return customerID;
11      }
12
13      private long createID() {
14        return ... // generate new ID
15      }
16
17      // more declarations
18
19      }
```

Old-Style Enumerated Type Idiom

- **Enumerated types** are a common idiom in programming.

```
1       package cards.domain;
2
3       public class PlayingCard {
4
5         // pseudo enumerated type
6         public static final int SUIT_SPADES     = 0;
7         public static final int SUIT_HEARTS     = 1;
8         public static final int SUIT_CLUBS      = 2;
9         public static final int SUIT_DIAMONDS   = 3;
10
11        private int suit;
12        private int rank;
13
14        public PlayingCard(int suit, int rank) {
15          this.suit = suit;
16          this.rank = rank;
17        }
```

```
22    public String getSuitName() {
23      String name = "";
24      switch ( suit ) {
25        case SUIT_SPADES:
26          name = "Spades";
27          break;
28        case SUIT_HEARTS:
29          name = "Hearts";
30          break;
31        case SUIT_CLUBS:
32          name = "Clubs";
33          break;
34        case SUIT_DIAMONDS:
35          name = "Diamonds";
36          break;
37        default:  System.err.println("Invalid suit.");
38      }
39      return name;
40    }
```

Old-Style Enumerated Type Idiom(Cont.)

- Old-style idiom is **not type-safe**:

```
1       package cards.tests;
2
3       import cards.domain.PlayingCard;
4
5       public class TestPlayingCard {
6         public static void main(String[] args) {
7
8           PlayingCard card1
9             = new PlayingCard(PlayingCard.SUIT_SPADES, 2);
10            System.out.println("card1 is the " + card1.getRank()
11              + " of " + card1.getSuitName());
12
13            // You can create a playing card with a bogus suit.
14            PlayingCard card2 = new PlayingCard(47, 2);
15            System.out.println("card2 is the " + card2.getRank()
16              + " of " + card2.getSuitName());
17        }
18      }
```

Old-Style Enumerated Type Idiom(Cont.)

- This enumerated type idiom has several problems:
  - Not type-safe
  - No namespace
  - Brittle character
  - Uninformative printed values

The Enumerated Type

- Now you can create **type-safe enumerated types**:

```
1     package cards.domain;
2
3     public enum Suit {
4        SPADES,
5        HEARTS,
6        CLUBS,
7        DIAMONDS
8     }
9
```

The Enumerated Type(Cont.)

- Using enumerated types is easy:

```
1     package cards.domain;
2
3     public class PlayingCard {
4
5       private Suit suit;
6       private int rank;
7
8       public PlayingCard(Suit suit, int rank) {
9         this.suit = suit;
10        this.rank = rank;
11      }
12
13      public Suit getSuit() {
14        return suit;
15      }
```

```
16    public String getSuitName() {
17      String name = "";
18      switch ( suit ) {
19        case SPADES:
20          name = "Spades";
21          break;
22        case HEARTS:
23          name = "Hearts";
24          break;
25        case CLUBS:
26          name = "Clubs";
27          break;
28        case DIAMONDS:
29          name = "Diamonds";          break;
30        default: // No need for error checking as the Suit
31        // enum is finite.
32      }
33      return name;
34    }
```

The Enumerated Type(Cont.)

- Enumerated types are **type-safe**:

```
1     package cards.tests;
2
3     import cards.domain.PlayingCard;
4     import cards.domain.Suit;
5
6     public class TestPlayingCard {
7       public static void main(String[] args) {
8
9         PlayingCard card1 = new PlayingCard(Suit.SPADES, 2);
10        System.out.println("card1 is the " + card1.getRank()
11          + " of " + card1.getSuitName());
12
13        // PlayingCard card2 = new PlayingCard(47, 2);
14        // This will not compile.
15      }
16    }
```

Advanced Enumerated Types

- Enumerated types can have **attributes** and **methods**:

```
1     package cards.domain;
2
3     public enum Suit {
4       SPADES(”Spades”),
5       HEARTS(”Hearts”),
6       CLUBS(”Clubs”),
7       DIAMONDS(”Diamonds”);
8
9       private final String name;
10
11      private Suit(String name) {
12        this.name = name;
13      }
14
15      public String getName() {
16        return name;
17      }
18    }
```

Advanced Enumerated Types(Cont.)

- Public methods on enumerated types are accessible:

```
1       package cards.tests;
2
3       import cards.domain.PlayingCard;
4       import cards.domain.Suit;
5
6       public class TestPlayingCard {
7         public static void main(String[] args) {
8
9           PlayingCard card1
10            = new PlayingCard(Suit.SPADES, 2);
11          System.out.println("card1 is the " + card1.getRank()
12            + " of " + card1.getSuit().getName());
13
14          // NewPlayingCard card2 = new NewPlayingCard(47, 2);
15          // This will not compile.
16        }
17      }
```

Static Imports

- A static import imports the static members from a class:

```
import static <pkg_list>.<class_name>.<member_name>;
```

OR

```
import static <pkg_list>.<class_name>.*;
```

- A static import imports members **individually** or **collectively**:

```
import static cards.domain.Suit.SPADES;
```

OR

```
import static cards.domain.Suit.*;
```

- There is no need to qualify the static constants:

```
PlayingCard card1 = new PlayingCard(SPADES, 2);
```

- *Use this feature sparingly. // Not recommended*

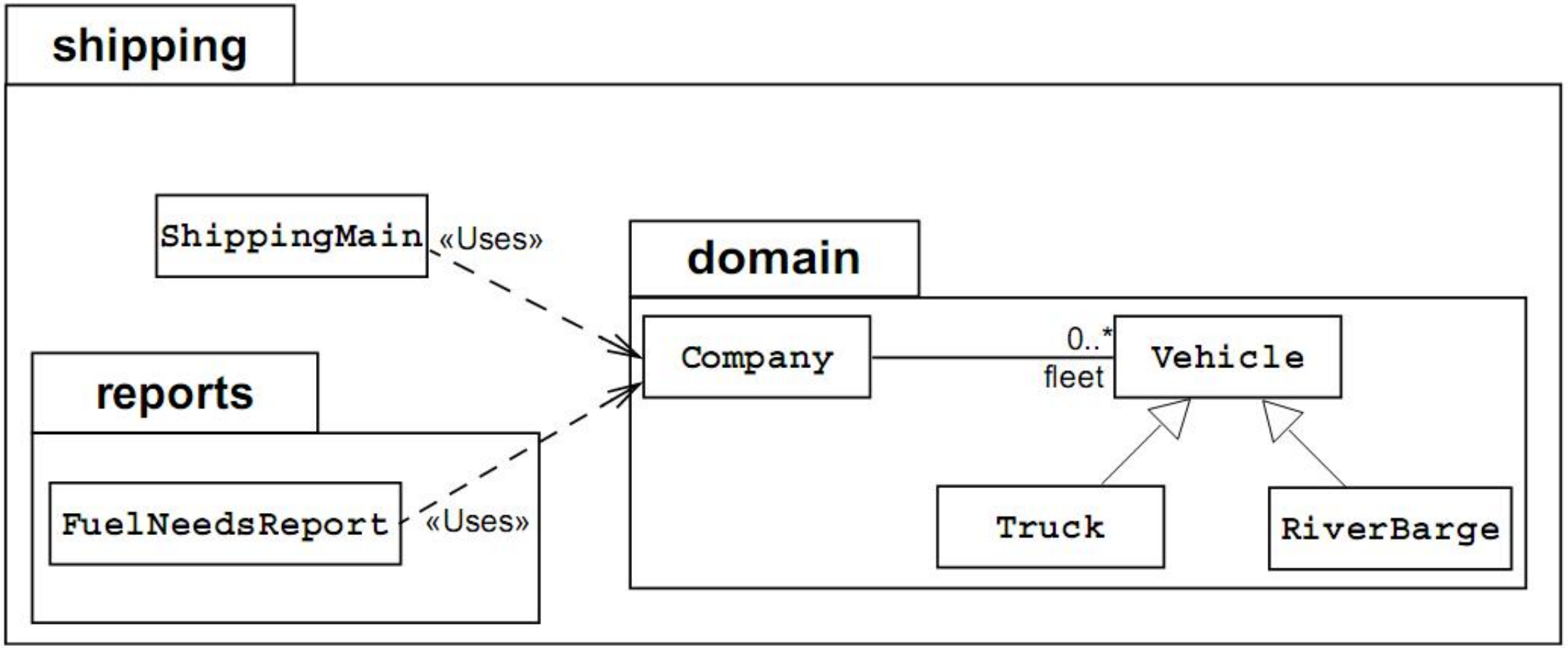
Static Imports(Cont.)

- An example of a static import is:

```
1     package cards.tests;
2
3     import cards.domain.PlayingCard;
4     import static cards.domain.Suit.*;
5
6     public class TestPlayingCard {
7       public static void main(String[] args) {
8
9         PlayingCard card1 = new PlayingCard(SPADES, 2);
10        System.out.println("card1 is the " + card1.getRank()
11          + " of " + card1.getSuit().getName());
12
13        // NewPlayingCard card2 = new NewPlayingCard(47, 2);
14        // This will not compile.
15      }
16    }
```

Abstract Classes

- The design of the Shipping system looks like this:



Abstract Classes(Cont.)

- Fleet(车队) initialization code is shown here:

```
1     public class ShippingMain {
2       public static void main(String[] args) {
3         Company c = new Company();
4
5         // populate the company with a fleet of vehicles
6         c.addVehicle( new Truck(10000.0) );
7         c.addVehicle( new Truck(15000.0) );
8         c.addVehicle( new RiverBarge(500000.0) );
9         c.addVehicle( new Truck(9500.0) );
10        c.addVehicle( new RiverBarge(750000.0) );
11
12        FuelNeedsReport report = new FuelNeedsReport(c);
13        report.generateText(System.out);
14      }
15    }
```

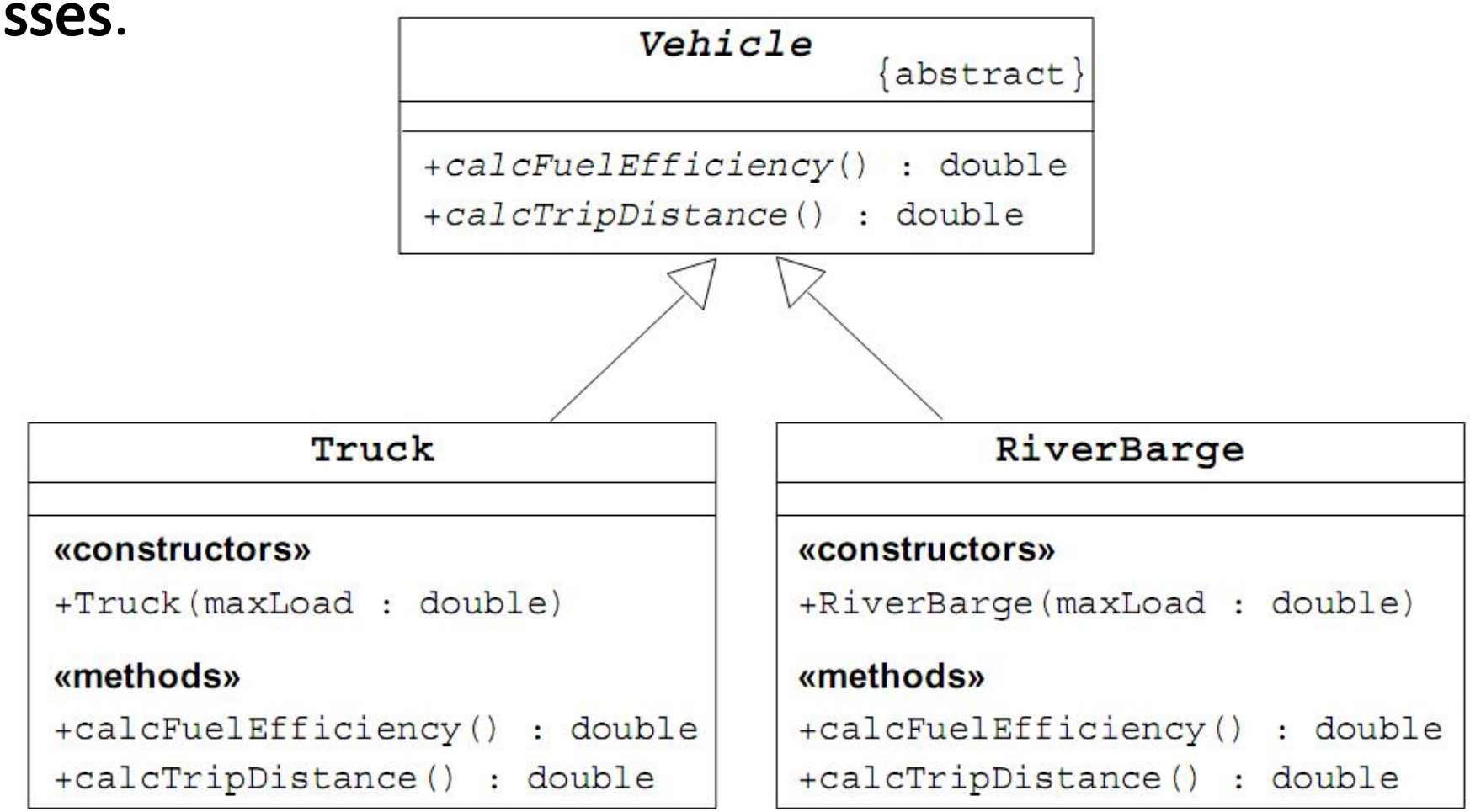
Abstract Classes(Cont.)

```
1     public class FuelNeedsReport {
2         private Company company;
3
4         public FuelNeedsReport(Company company) {
5             this.company = company;
6         }
7
8         public void generateText(PrintStream output){
9             Vehicle1 v;
10            double fuel;
11            double total_fuel = 0.0;
12
13            for( int i = 0;i < company.getFleetSize();i++ ){
14                v = company.getVehicle(i);
15
```

```
16          // Calculate the fuel needed for this trip
17          fuel = v.calcTripDistance() / v.calcFuelEfficency();
18
19          output.println("Vehicle " + v.getName() + " needs "
20            + fuel + " liters of fuel.");
21          total_fuel += fuel;
22        }
23        output.println("Total fuel needs is " + total_fuel + " liters.");
24      }
25    }
```

The Solution

- An **abstract class** models a class of objects in which the **full implementation is not known** but is **supplied by the concrete subclasses**.



The Solution(Cont.)

- The declaration of the `Vehicle` class is:

```
1    public abstract class Vehicle {
2      public abstract double calcFuelEfficiency();
3      public abstract double calcTripDistance();
4    }
```

- The **`Truck`** class must create an implementation:

```
1    public class Truck extends Vehicle {
2      public Truck(double maxLoad) {...}
3      public double calcFuelEfficiency() {
4        /* calculate the fuel consumption of a truck at a given load */
5      }
6      public double calcTripDistance() {
7        /* calculate the distance of this trip on highway */
8      }
9    }
```

The Solution(Cont.)

- Likewise, the **RiverBarge** class must create an implementation:

```
1     public class RiverBarge extends Vehicle {
2       public RiverBarge(double maxLoad) {...}
3       public double calcFuelEfficiency() {
4         /* calculate the fuel efficiency of a river barge */
5       }
6       public double calcTripDistance() {
7         /* calculate the distance of this trip along the river-ways */
8       }
9     }
```

Interfaces

- A **public interface** is a **contract** between **client code** and the **class that implements that interface**.
- A Java **interface** is a formal declaration of such a contract in which **all methods contain no implementation**.
- Many unrelated classes can implement the same interface.
- A class can implement many unrelated interfaces.
- Syntax of a Java class is as follows:

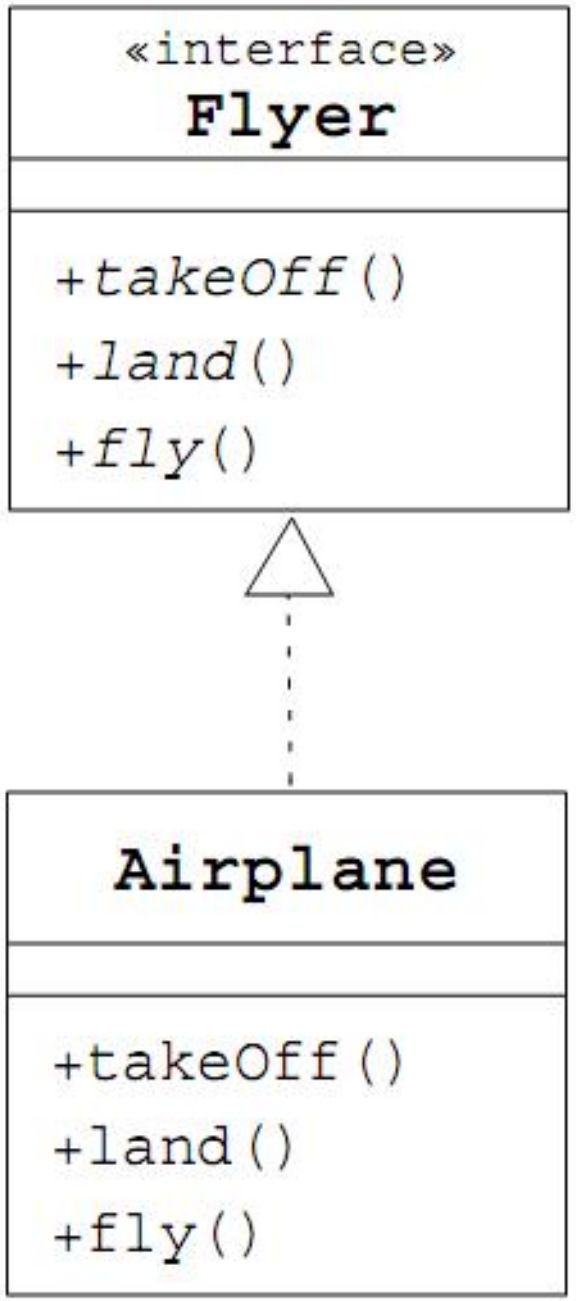
<modifier> class <name> [**extends <superclass>**] [**implements <interface>** [,**<interface>**]*] {

    <member_declaration>*

}

The Flyer Example

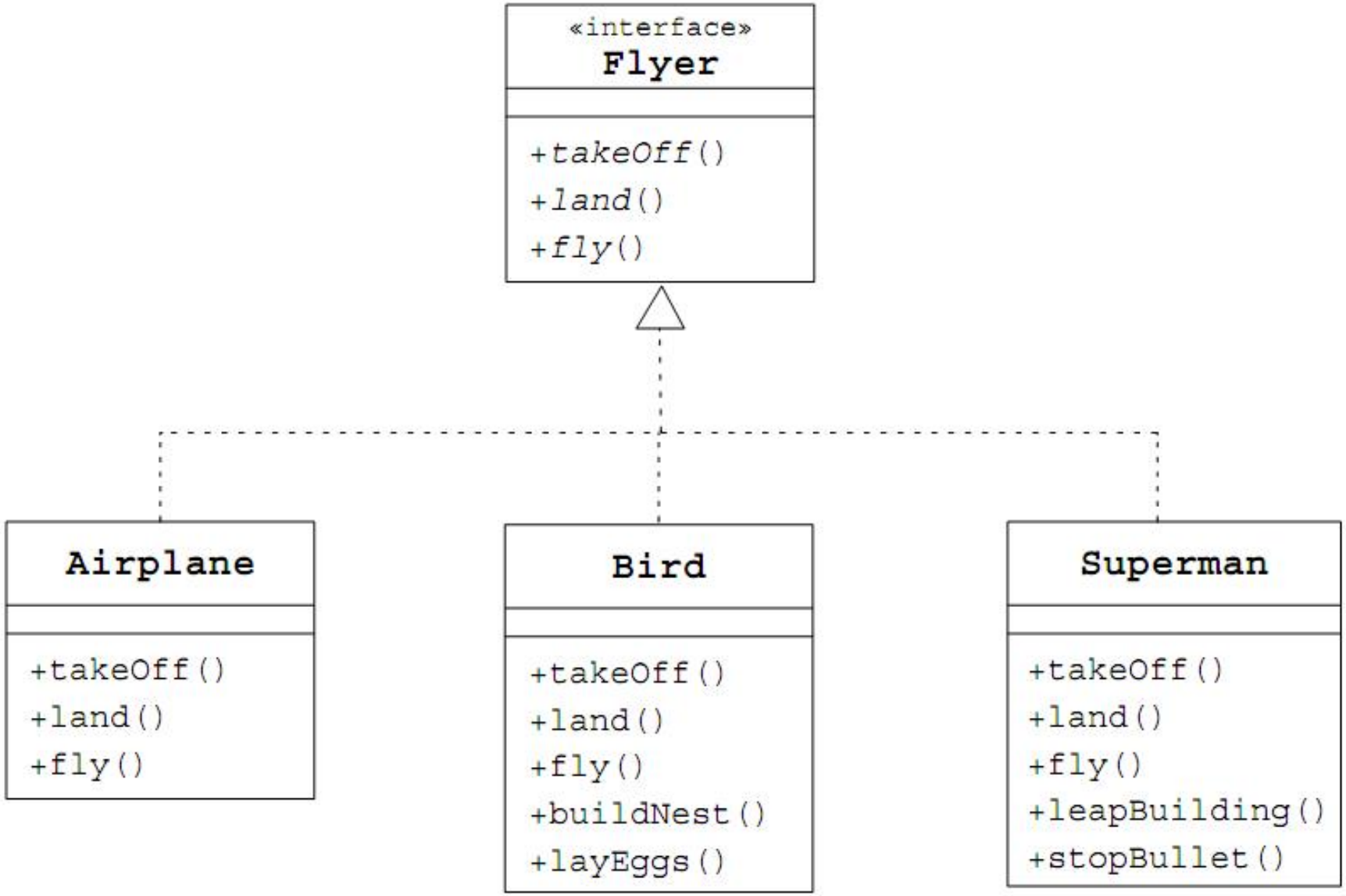
```
public interface Flyer {
    public void takeOff();
    public void land();
    public void fly();
}
```



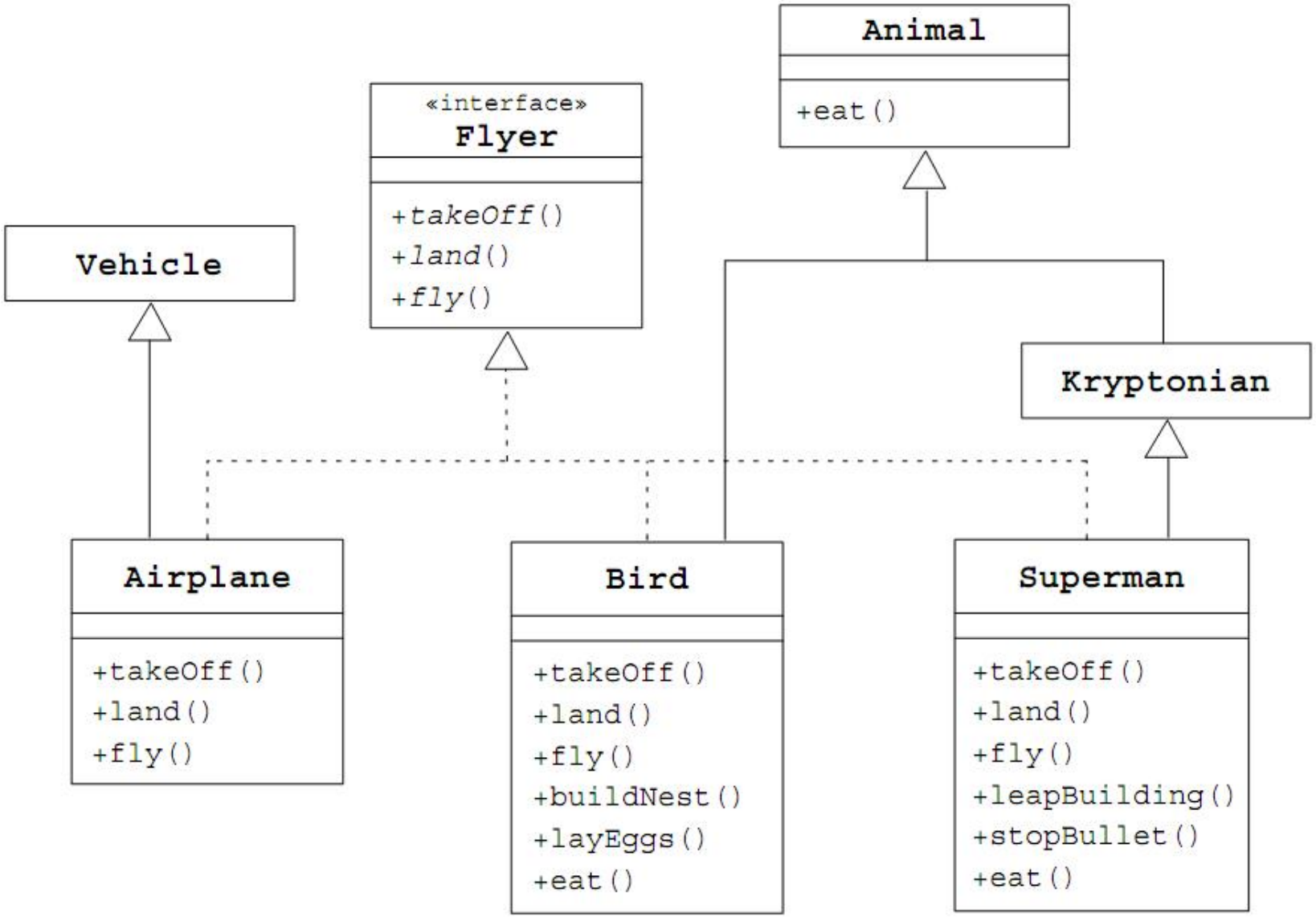
The Flyer Example(Cont.)

```
public class Airplane implements Flyer {
   public void takeOff() {
      // accelerate until lift-off
   // raise landing gear
   }
   public void land() {
      // lower landing gear
      // decelerate and lower flaps until touch-down
      // apply brakes
   }
   public void fly() {
      // keep those engines running
   }
}
```

The Flyer Example(Cont.)



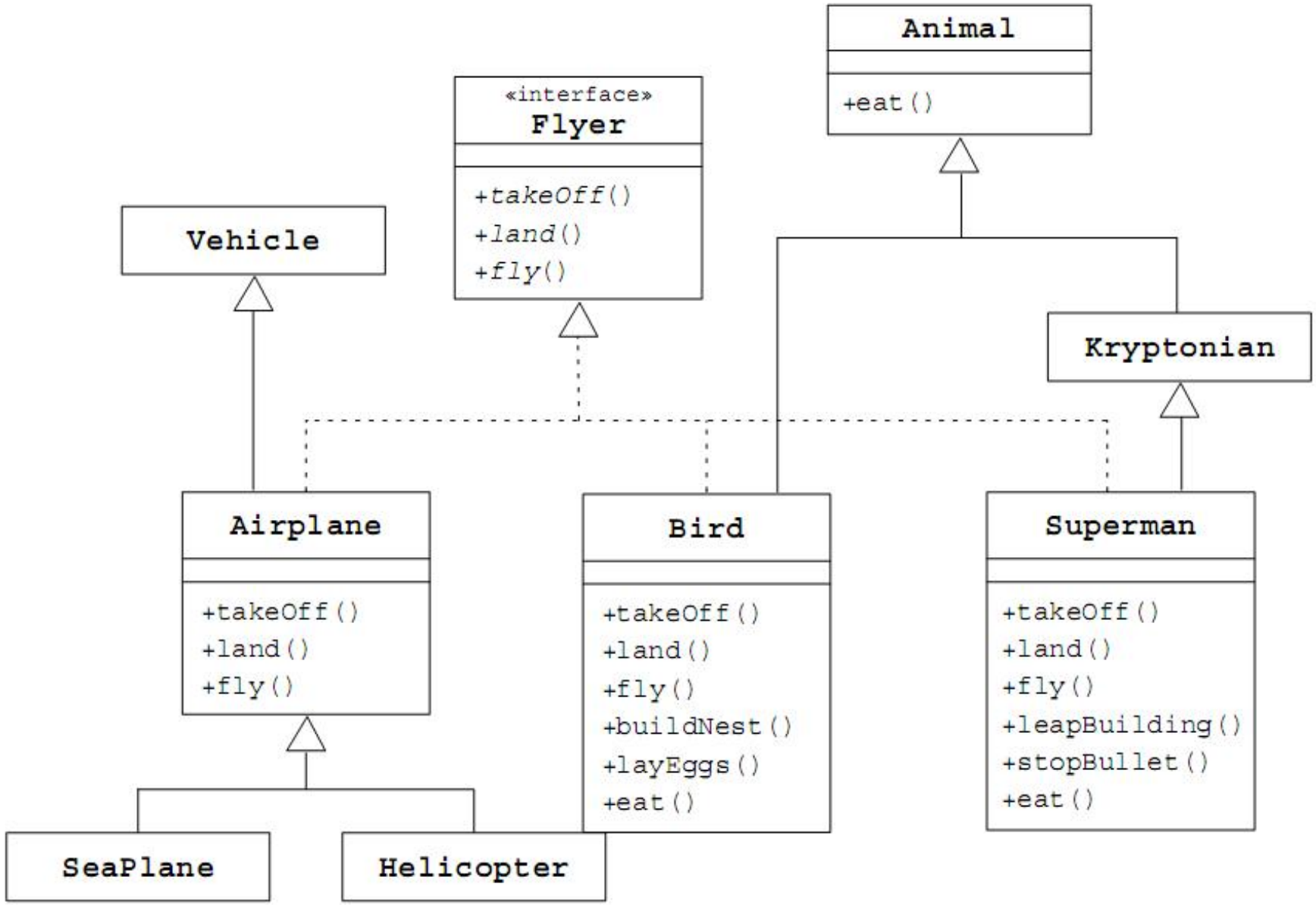
The Flyer Example(Cont.)



The Flyer Example(Cont.)

```
public class Bird extends Animal implements Flyer {
    public void takeOff() { /* take-off implementation */ }
    public void land() { /* landing implementation */ }
    public void fly() { /* fly implementation */ }
    public void buildNest() { /* nest building behavior */ }
    public void layEggs() { /* egg laying behavior */ }
    public void eat() { /* override eating behavior */ }
}
```

The Flyer Example(Cont.)



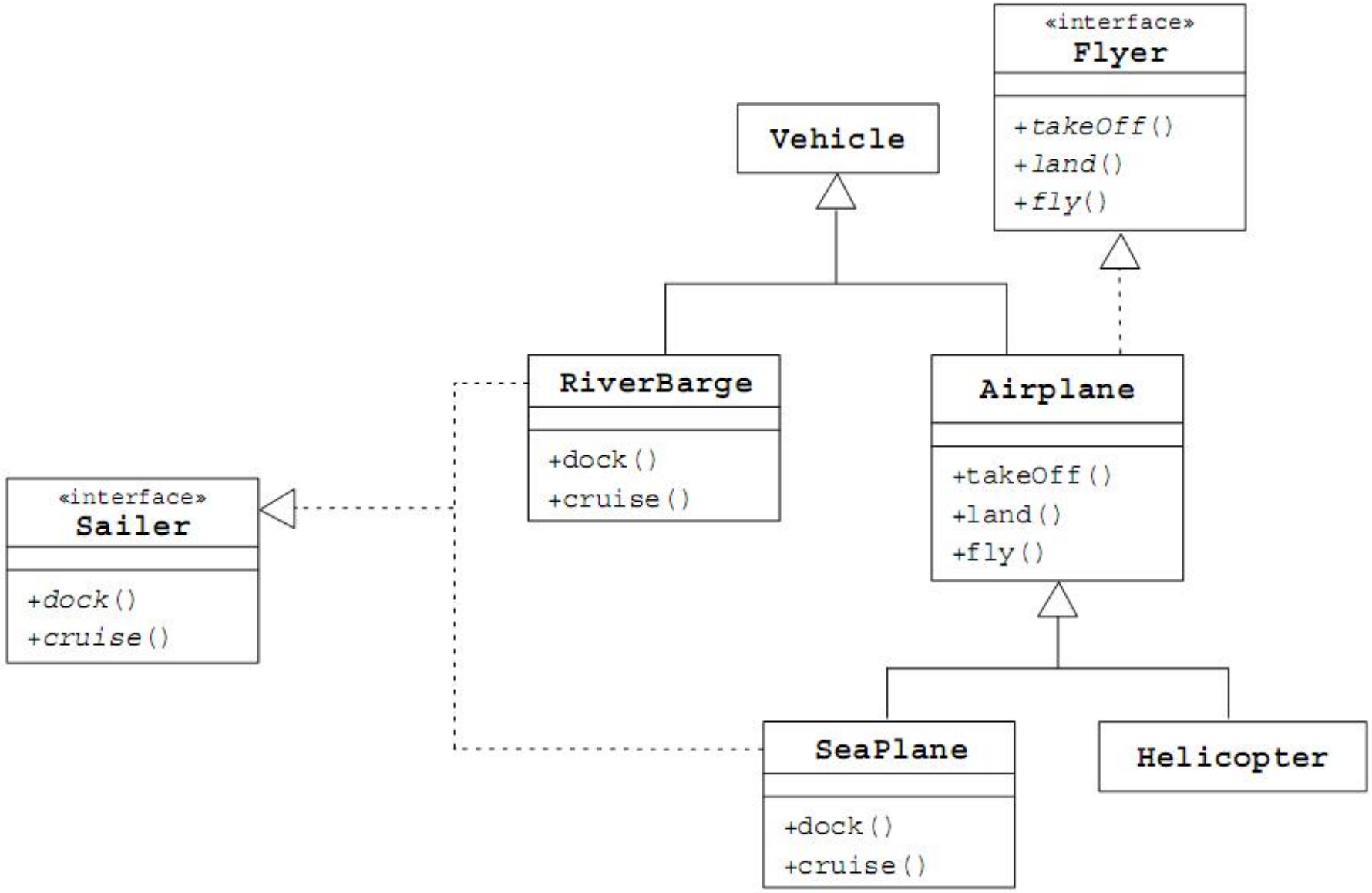
The Flyer Example(Cont.)

```
public class Airport {
  public static void main(String[] args) {
    Airport metropolisAirport = new Airport();
    Helicopter copter = new Helicopter();
    SeaPlane sPlane = new SeaPlane();

    metropolisAirport.givePermissionToLand(copter);
    metropolisAirport.givePermissionToLand(sPlane);
  }

  private void givePermissionToLand(Flyer f) {
    f.land();
  }
}
```

Multiple Interface Example



Multiple Interface Example(Cont.)

```
public class Harbor {
  public static void main(String[] args) {
    Harbor bostonHarbor = new Harbor();
    RiverBarge barge = new RiverBarge();
    SeaPlane sPlane = new SeaPlane();

    bostonHarbor.givePermissionToDock(barge);
    bostonHarbor.givePermissionToDock(sPlane);
  }

  private void givePermissionToDock(Sailer s) {
    s.dock();
  }
}
```

Uses of Interfaces

- Interface uses include the following:
  - Declaring **methods** that one or more classes are expected to implement
  - Determining an object's programming interface without revealing the actual body of the class
  - Capturing **similarities** between unrelated classes without forcing a class relationship
  - **Simulating multiple inheritance** by declaring a class that implements several interfaces

Interface Declarations and Usage Rules

- Comparing Java technology types:
  - Concrete class
  - Abstract class
  - Interface
- Facts about an interface:
  - All interface **methods** are **public** and **abstract**.
  - All interface **fields** are **static** and **final**.
- What an interface can contain:
  - **Constants** (that is **public static final** fields)
  - **Method interfaces** (but no method bodies)

Nested Classes

```
1     public class StackOfInts {
2       private int[] stack;
3       private int next = 0; // index of last item in stack + 1
4
5       public StackOfInts(int size) { //...}
6       public void push(int on) { // ... }
7       public int pop(){ // ... }
8
9       private class StepThrough {
10        private int i = 0;
11        public void increment() { // ... }
12        public int current() { // ... }
13        public boolean isLast(){ // ... }
14      }
15
16      public StepThrough stepThrough() { // ... }
17      public static void main(String[] args) { // ... }
18    }
```

Nested Classes(Cont.)

- **Nested class** have **complete access** to all **fields** and **methods** of the outer class.
- Nested classes are **often used for**:
  - New levels of **encapsulation**
  - **Improved readability** and **maintainability** of your code
  - Additional levels for organizing a class hierarchy

Examining the Nested Class Syntax

```
1     [public] class OuterClass {
2        ...
3        [public|protected|private|static]class NestedClass {
4           ...
5        }
6     }
```

- Nested class categories:
  - **Non-static nested classes**, which are called **inner classes**
  - **Static nested classes**

Syntax for Inner Classes

```
class OuterClass {
   ...
   class InnerClass {
      ...
   }
   ...
}
```

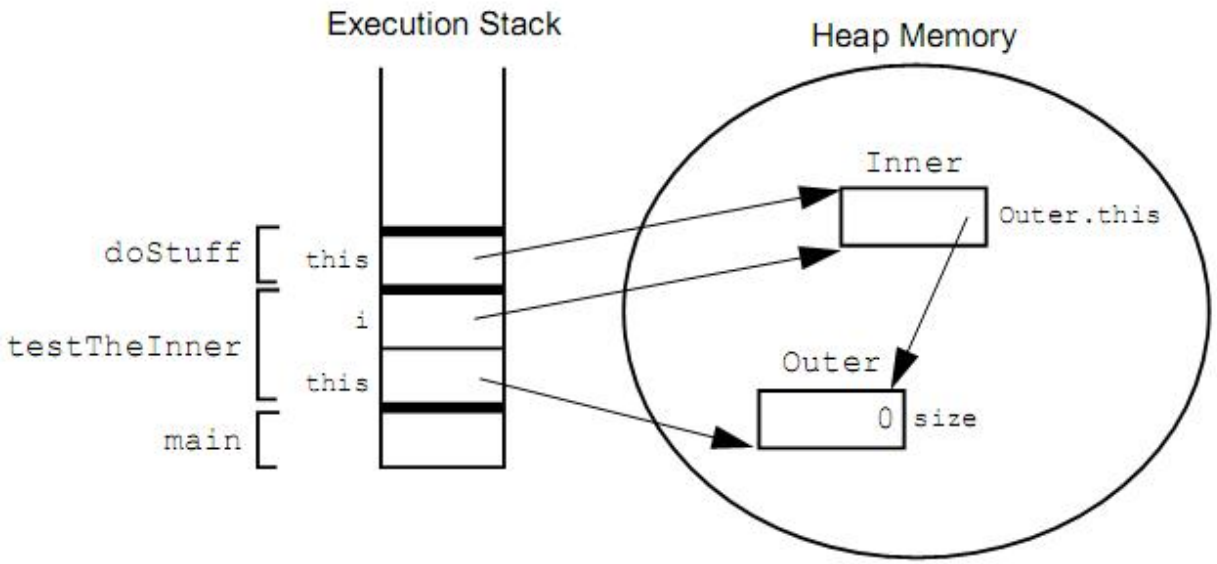
Inner Classes

- Objects that are **instances of an inner class** exist inside an instance of the outer class, and therefore have direct access to the methods and fields of its enclosing instance.
- To **instantiate an inner class**, you **must** first instantiate the outer class. Then, create the inner class object instance within the outer object with the following **syntax**:

```
OuterClass.InnerClass innerObject
      = outerObject.new InnerClass();
```

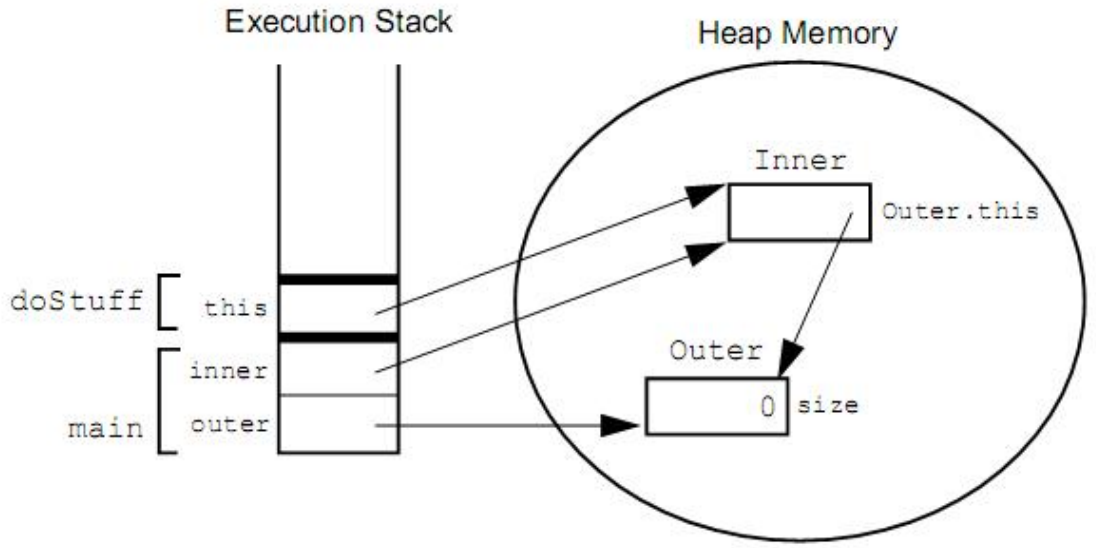
Inner Classes Examples

```
1   public class Outer1 {
2     private int size;
3
4     /* Declare an inner class called "Inner" */
5     public class Inner {
6       public void doStuff() {
7         // The inner class has access to 'size' from Outer
8         size++;
9       }
10    }
11
12    public void testTheInner() {
13      Inner i = new Inner();
14      i.doStuff();
15    }
16  }
```



Inner Classes Examples(Cont.)

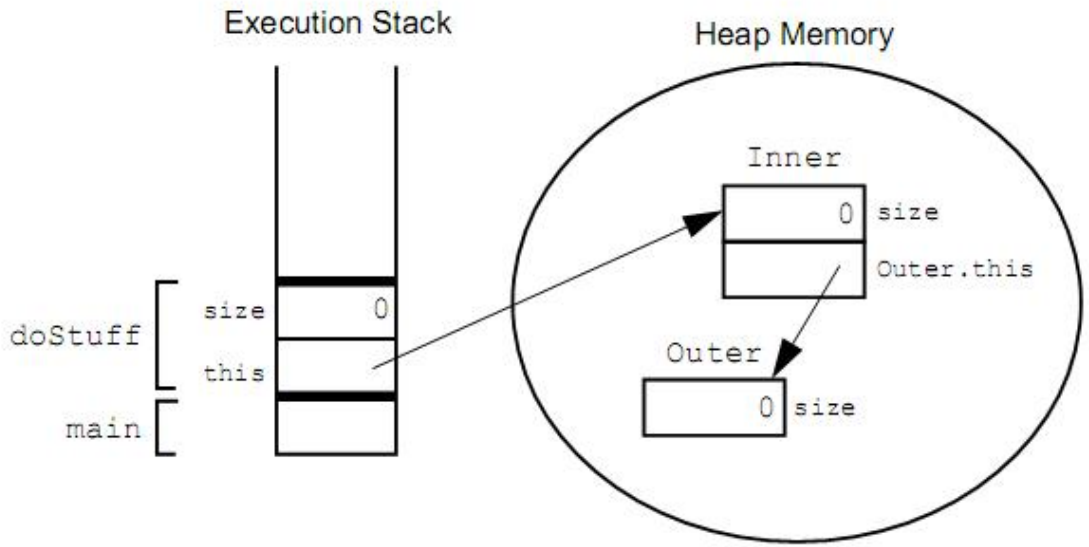
```
1   public class Outer2 {
2     private int size;
3
4     public class Inner {
5       public void doStuff() {
6         size++;
7       }
8     }
9   }
```




```
1   public class TestInner {
2     public static void main(String[] args) {
3       Outer2 outer = new Outer2();
4
5       // Must create an Inner object relative to an Outer
6       Outer2.Inner inner = outer.new Inner();
7       inner.doStuff();
8     }
9   }
```

Inner Classes Examples(Cont.)

```
1   public class Outer3 {
2     private int size;
3
4     public class Inner {
5       private int size;
6
7       public void doStuff(int size) {
8         size++;              // the local parameter
9         this.size++;         // the Inner object attribute
10        Outer3.this.size++;  // the Outer3 object attribute
11      }
12    }
13  }
```



Local Inner Classes

Definition:

- A class **declared inside a block of code in a method body**.
- Two kinds of inner classes:
  - •Local classes
  - •Anonymous inner classes

Properties:

- Only **visible** within **the block of code in which it is defined**
- Have **runtime access** to everything that is normally accessible from within the block of code in which they are defined (the method's parameters, local variables, and the members of the outer class)

Local Inner Classes(Cont.)

```
class OuterClass {
   ...
   ...
   void aMethod{
      ...
      InnerClass x = new InnerClass();
      ...
      class InnerClass {
         ...
      }
      ...
      // do something with instance variable x
      ...
   }
   ...
}
```

Local Inner Class Example

```
1   public class Outer4 {
2     private int size = 5;
3
4     public Object makeTheInner(int localVar) {
5       final int finalLocalVar = 6;
6
7       // Declare a class within a method!?!
8       class Inner {
9         public String toString() {
10          return ("#<Inner size=" + size +
11                  // " localVar=" + localVar + // ERROR: ILLEGAL
12                  "finalLocalVar=" + finalLocalVar + ">");
13        }
14      }
15
16      return new Inner();
17    }
18
19    public static void main(String[] args) {
20      Outer4 outer = new Outer4();
21      Object obj = outer.makeTheInner(47);
22      System.out.println("The object is " + obj);
23    }
24  }
```

Anonymous Inner Class

- Definition: An inner class with no declared name. Syntax:

```
new existingTypeName ( [ argumentList ] ) {
   // class code here
}
```

Example of an Anonymous Inner Class

```
1     metropolisAirport.givePermissionToLand(new AirPlane(){
2       public void land() {
3         //do something that only UFOs can do...
4       } //close land method
5     } //close anonymous inner class
6     ) //close argument to givePermissionToLand method
7     ;     //end statement started at line 1
8
```

Anonymous Inner Class: Properties

- Anonymous inner classes are most useful under the following circumstances:
  - When the **declaration and usage** of the class are **adjacent**
  - When the **class code is short**

- **An anonymous class** is:
  - Always an **inner class** and always **implicitly final**
  - **Never abstract and never static**

Static Nested Classes

- Definition: A static nested class is **a static class defined in the namespace of a top-level class**.
- **Restrictions**: A static nested class **cannot refer directly to instance variables** or **methods** defined **in its enclosing class**.

```
public class Outer {
   ...
   public static class StaticNested {
      ...
   }
   ...
}
```

Static Nested Classes(Cont.)

- To reference a static nested class, use the **outer class name**:

```
Outer.StaticNested
```

- Example of the syntax to create an instance of a static nested class:

```
Outer.StaticNested nestedObject = new Outer.StaticNested();
```

Summary

- **static** variables, methods, and initializers
- **final** classes, methods, and variables
- **abstract** classes and methods
- **interface**
- **enumerate** Types
- Singleton Design Pattern
- Nested Class

Questions or Comments?