Module 7

Advanced Class Features

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

Relevance

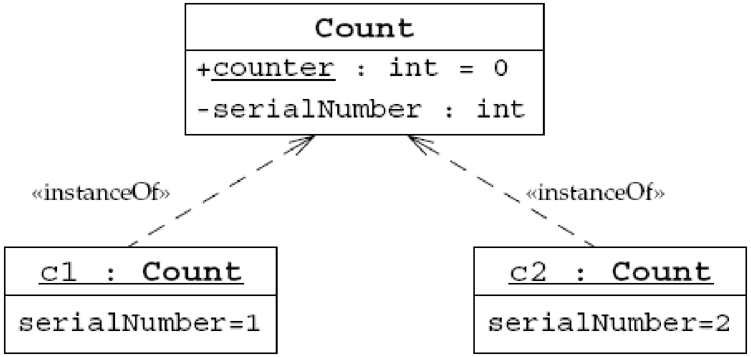
- How can you create a constant?
- How can you declare data that is shared by all instances of a given class?
- How can you keep a class or method from being subclassed or overridden?

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:



Count

+counter : int = 0

-serialNumber : int

«instanceOf»

«instanceOf»

c1 : Count

serialNumber=1

c2 : Count

serialNumber=2

```
public class Count {
  private int serialNumber;
  public static int counter = 0;

  public Count() {
    counter++;
    serialNumber = counter;
  }
}
```

Class Attributes

If the static member is `public`:

```
public class Count1 {
  private int serialNumber;
  public static int counter = 0;
  public Count1() {
    counter++;
    serialNumber = counter;
  }
}
```

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

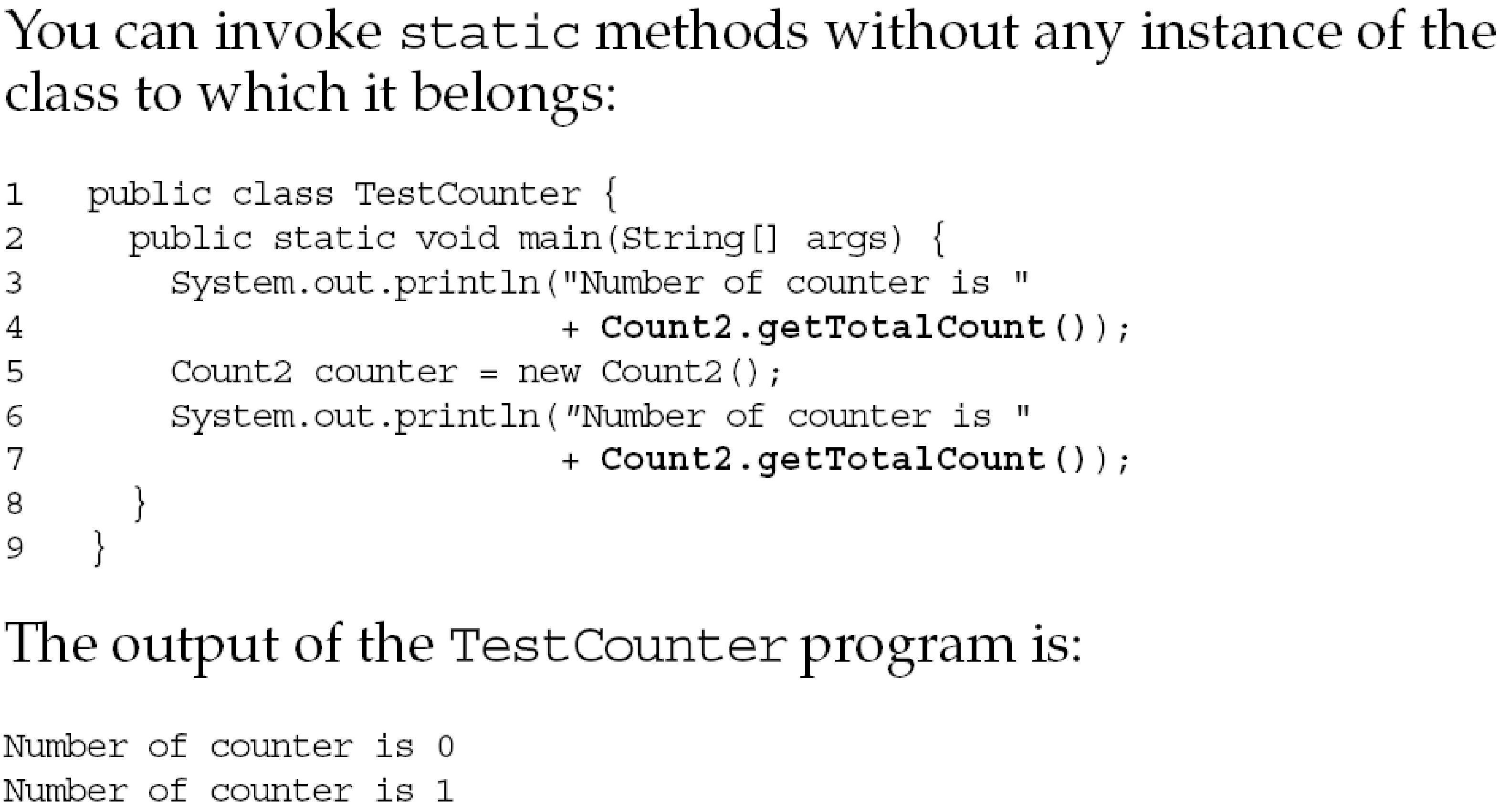
```
public class OtherClass {
  public void incrementNumber() {
    Count1.counter++;
  }
}
```

Class Methods

You can create `static` methods:

```
public class Count2 {
  private int serialNumber;
  private static int counter = 0;

  public static int getTotalCount() {
    return counter;
  }

  public Count2() {
    counter++;
    serialNumber = counter;
  }
}
```

Class Methods



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

```
public class TestCounter {
  public static void main(String[] args) {
    System.out.println("Number of counter is "
                       + Count2.getTotalCount());
    Count2 counter = new Count2();
    System.out.println(″Number of counter is "
                       + Count2.getTotalCount());
  }
}
```

The output of the `TestCounter` program is:

```
Number of counter is 0
Number of counter is 1
```

Class Methods

Static methods cannot access instance variables:

```
public class Count3 {
  private int serialNumber;
  private static int counter = 0;

  public static int getSerialNumber() {
    return serialNumber;  // COMPILER ERROR!
  }
}
```

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

```
public class Count4 {
  public static int counter;
  static {
    counter = Integer.getInteger("myApp.Count4.counter").intValue();
  }
}
```

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

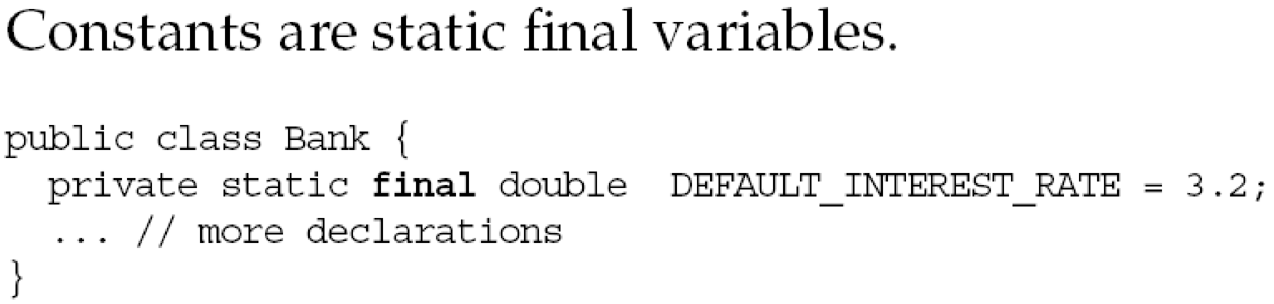
The output of the `TestStaticInit` program is:

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

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.

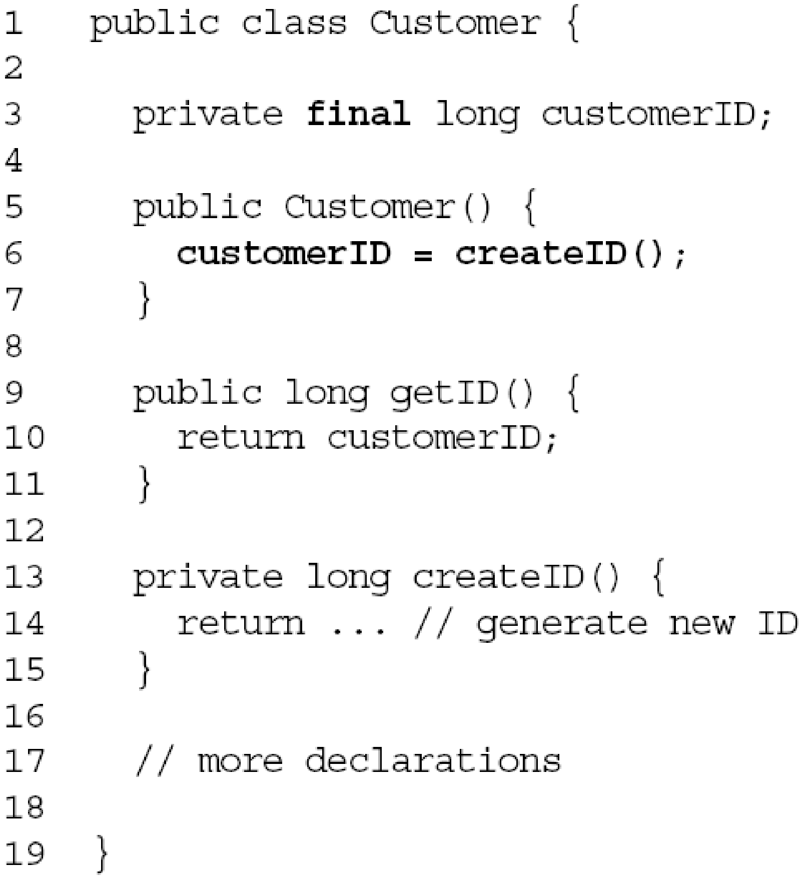
Final Variables



Constants are static final variables.

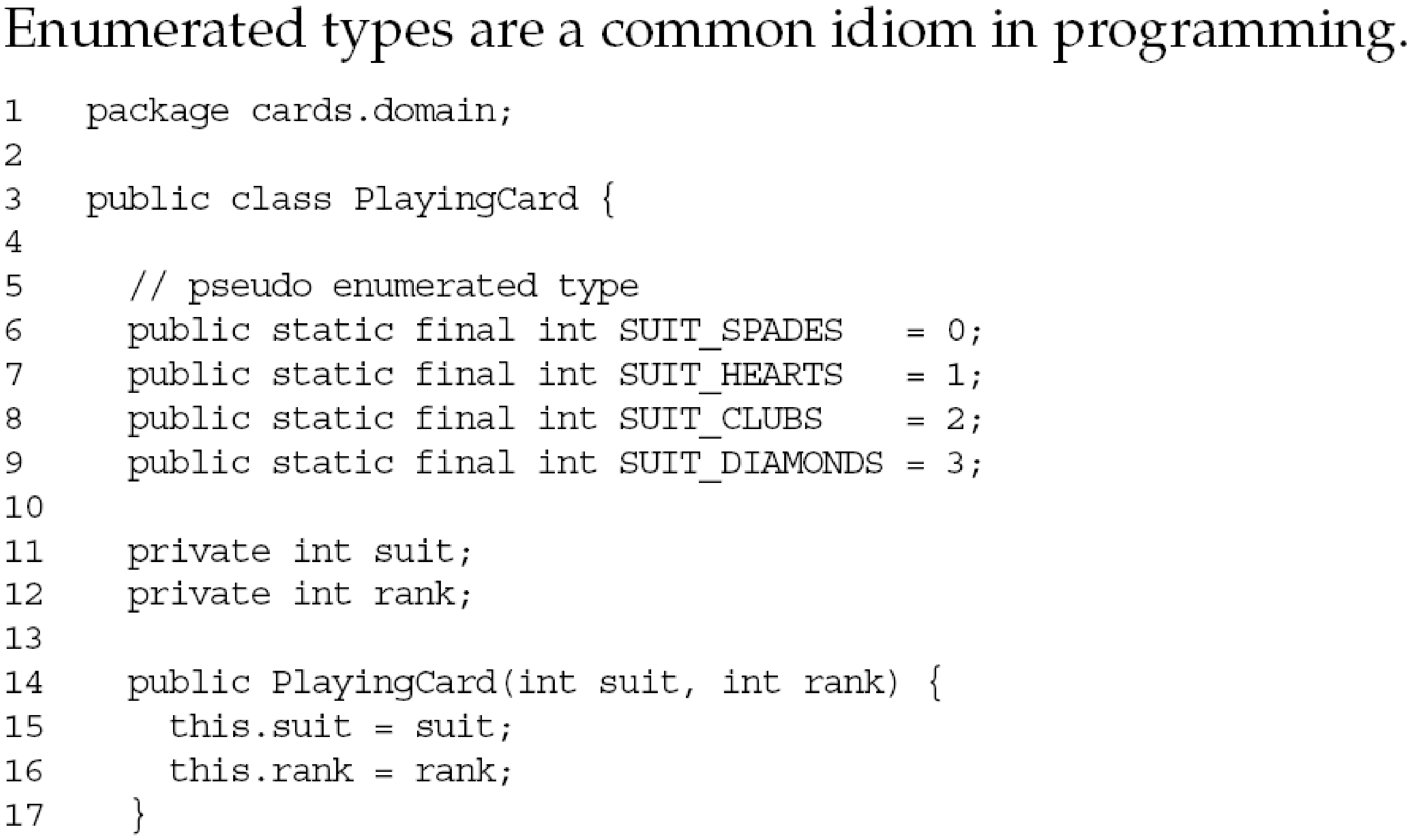
```
public class Bank {
  private static final double  DEFAULT_INTEREST_RATE = 3.2;
  ... // more declarations
}
```

Blank Final Variables



```
public class Customer {

  private final long customerID;

  public Customer() {
    customerID = createID();
  }

  public long getID() {
    return customerID;
  }

  private long createID() {
    return ... // generate new ID
  }

  // more declarations

}
```

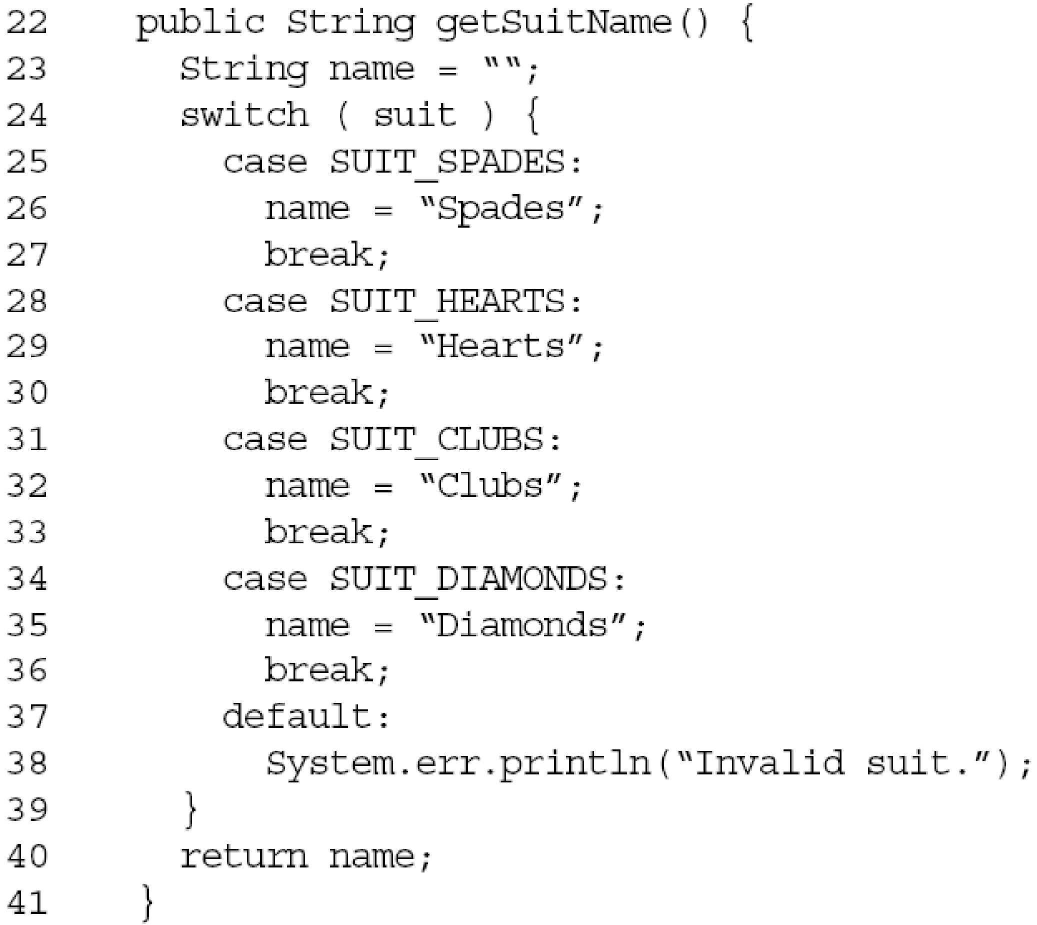
Old-Style Enumerated Type Idiom



Enumerated types are a common idiom in programming.

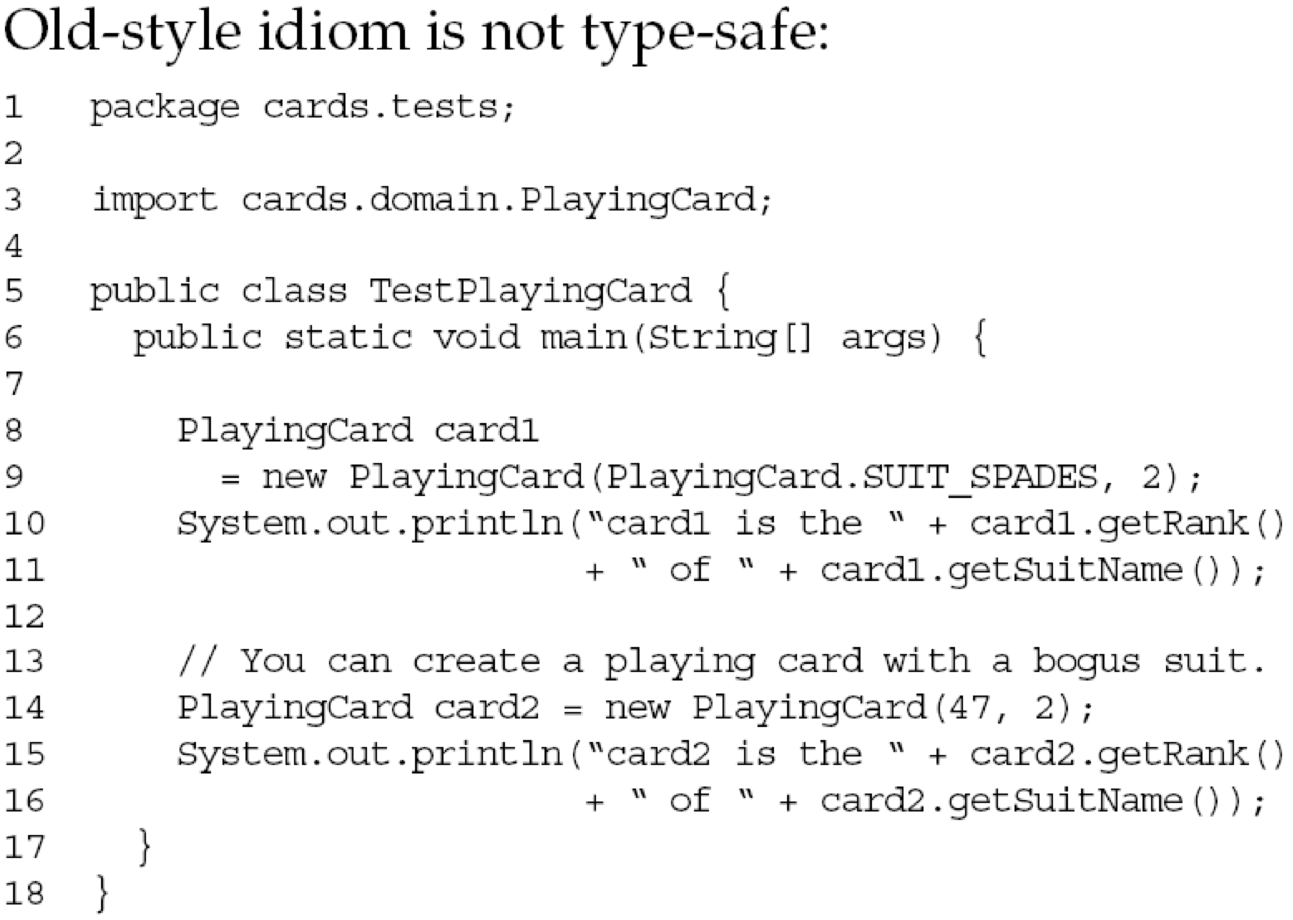
```
package cards.domain;

public class PlayingCard {

  // pseudo enumerated type
  public static final int SUIT_SPADES   = 0;
  public static final int SUIT_HEARTS   = 1;
  public static final int SUIT_CLUBS    = 2;
  public static final int SUIT_DIAMONDS = 3;

  private int suit;
  private int rank;

  public PlayingCard(int suit, int rank) {
    this.suit = suit;
    this.rank = rank;
  }
```

Old-Style Enumerated Type Idiom



```
public String getSuitName() {
  String name = ““;
  switch ( suit ) {
    case SUIT_SPADES:
      name = “Spades”;
      break;
    case SUIT_HEARTS:
      name = “Hearts”;
      break;
    case SUIT_CLUBS:
      name = “Clubs”;
      break;
    case SUIT_DIAMONDS:
      name = “Diamonds”;
      break;
    default:
      System.err.println(“Invalid suit.”);
  }
  return name;
}
```

Old-Style Enumerated Type Idiom



Old-style idiom is not type-safe:

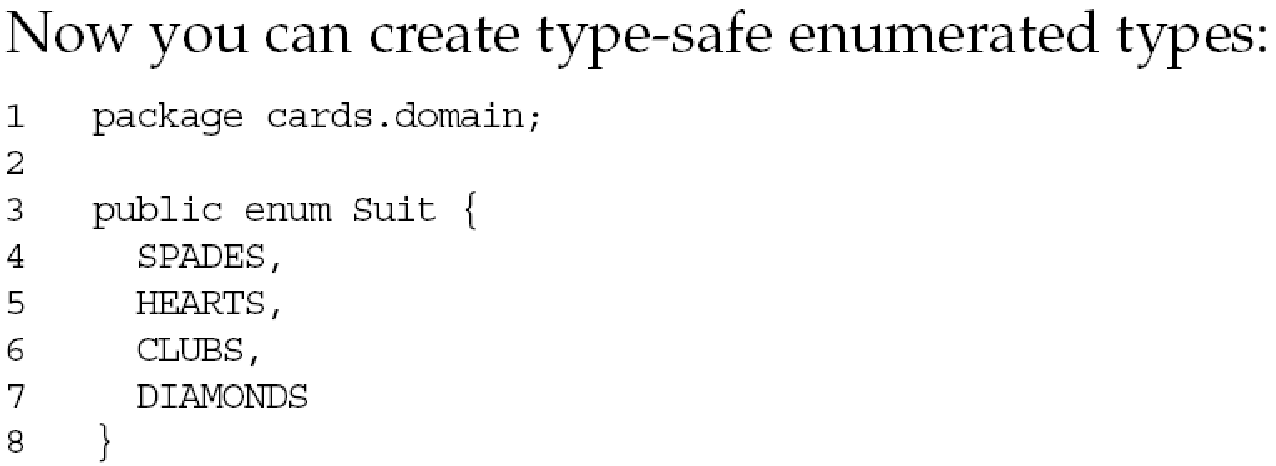
```
package cards.tests;

import cards.domain.PlayingCard;

public class TestPlayingCard {
  public static void main(String[] args) {

    PlayingCard card1
      = new PlayingCard(PlayingCard.SUIT_SPADES, 2);
    System.out.println(“card1 is the “ + card1.getRank()
                       + “ of “ + card1.getSuitName());

    // You can create a playing card with a bogus suit.
    PlayingCard card2 = new PlayingCard(47, 2);
    System.out.println(“card2 is the “ + card2.getRank()
                       + “ of “ + card2.getSuitName());
  }
}
```

Old-Style Enumerated Type Idiom

This enumerated type idiom has several problems:

- Not type-safe
- No namespace
- Brittle character
- Uninformative printed values

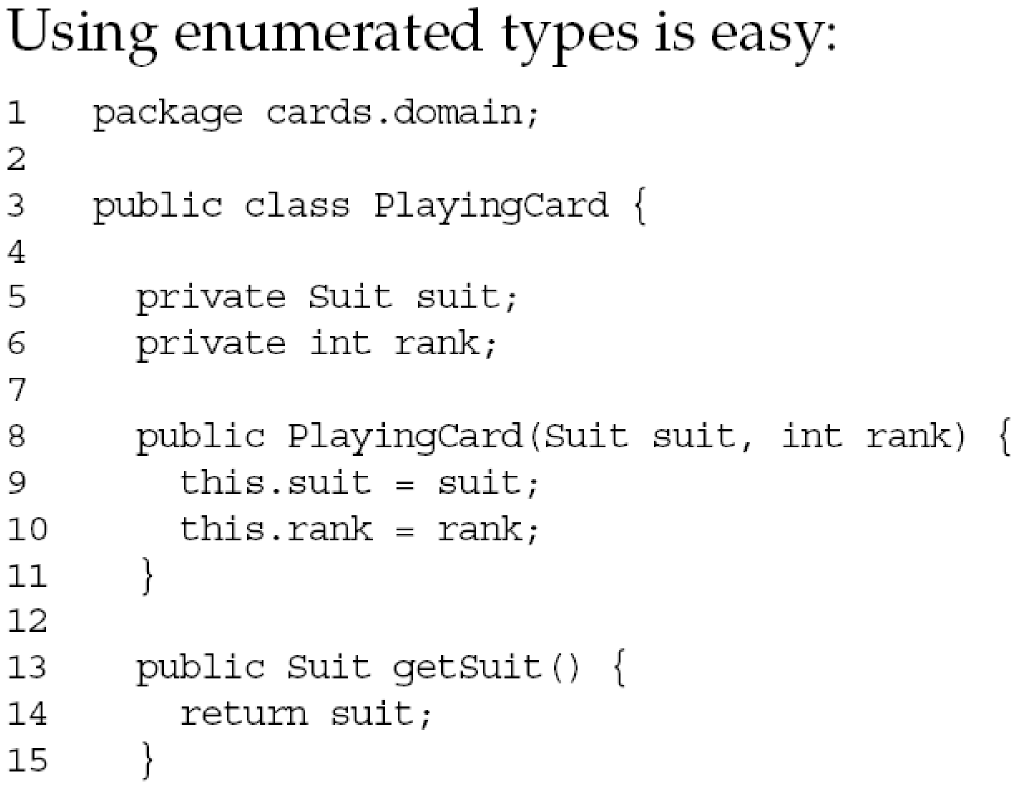
The New Enumerated Type



Now you can create type-safe enumerated types:

```java
package cards.domain;

public enum Suit {
  SPADES,
  HEARTS,
  CLUBS,
  DIAMONDS
}
```

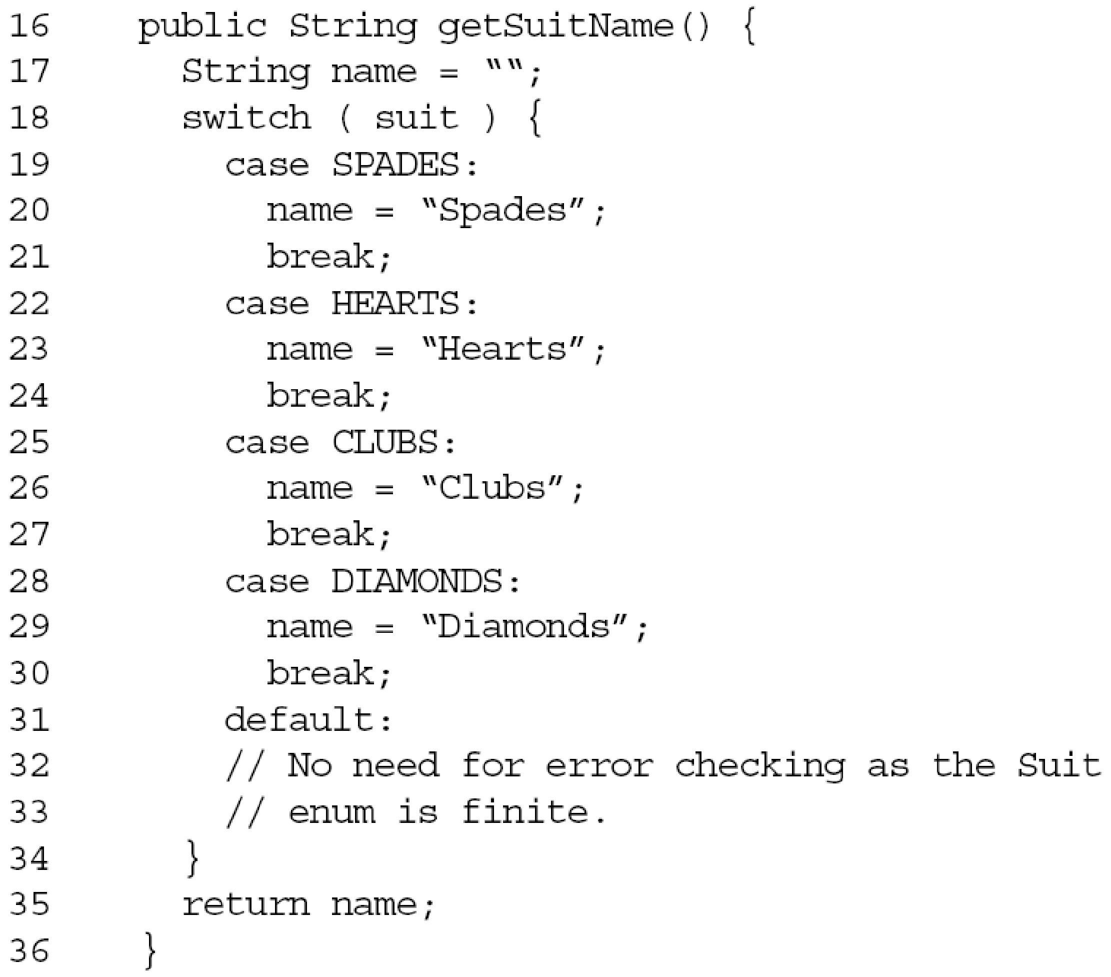
The New Enumerated Type



Using enumerated types is easy:

```
package cards.domain;

public class PlayingCard {

  private Suit suit;
  private int rank;

  public PlayingCard(Suit suit, int rank) {
    this.suit = suit;
    this.rank = rank;
  }

  public Suit getSuit() {
    return suit;
  }
```

The New Enumerated Type



```
public String getSuitName() {
  String name = “”;
  switch ( suit ) {
    case SPADES:
      name = “Spades”;
      break;
    case HEARTS:
      name = “Hearts”;
      break;
    case CLUBS:
      name = “Clubs”;
      break;
    case DIAMONDS:
      name = “Diamonds”;
      break;
    default:
    // No need for error checking as the Suit
    // enum is finite.
  }
  return name;
}
```

The New Enumerated Type

Enumerated types are type-safe:

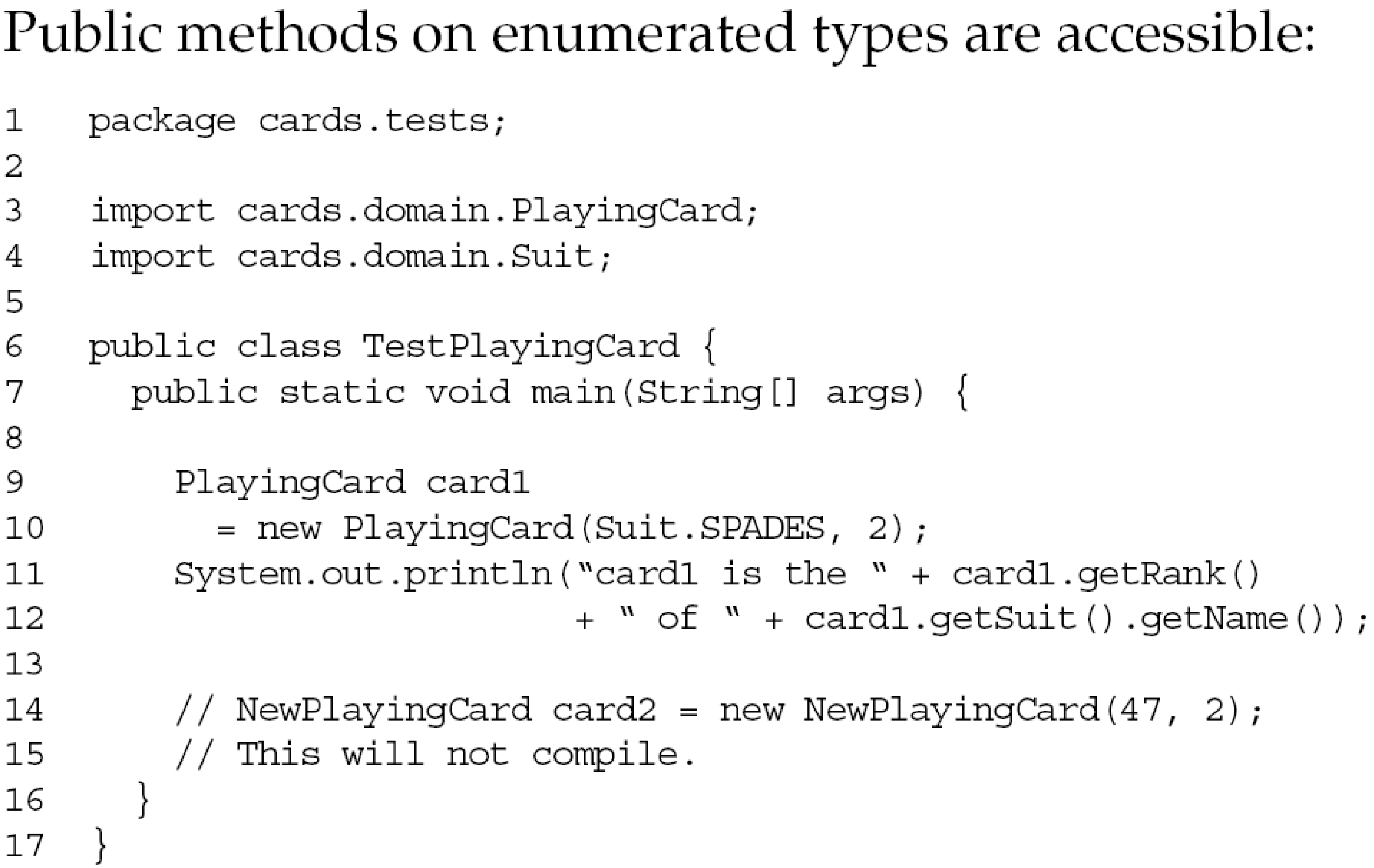
```java
package cards.tests;

import cards.domain.PlayingCard;
import cards.domain.Suit;

public class TestPlayingCard {
  public static void main(String[] args) {

    PlayingCard card1
      = new PlayingCard(Suit.SPADES, 2);
    System.out.println(“card1 is the “ + card1.getRank()
                       + “ of “ + card1.getSuitName());

    // PlayingCard card2 = new PlayingCard(47, 2);
    // This will not compile.
  }
}
```

Advanced Enumerated Types

Enumerated types can have attributes and methods:

```
package cards.domain;

public enum Suit {
  SPADES   (“Spades”),
  HEARTS   (“Hearts”),
  CLUBS    (“Clubs”),
  DIAMONDS (“Diamonds”);

  private final String name;

  private Suit(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

Advanced Enumerated Types



Public methods on enumerated types are accessible:

```
package cards.tests;

import cards.domain.PlayingCard;
import cards.domain.Suit;

public class TestPlayingCard {
  public static void main(String[] args) {

    PlayingCard card1
      = new PlayingCard(Suit.SPADES, 2);
    System.out.println(“card1 is the “ + card1.getRank()
                        + “ of “ + card1.getSuit().getName());

    // NewPlayingCard card2 = new NewPlayingCard(47, 2);
    // This will not compile.
  }
}
```

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:

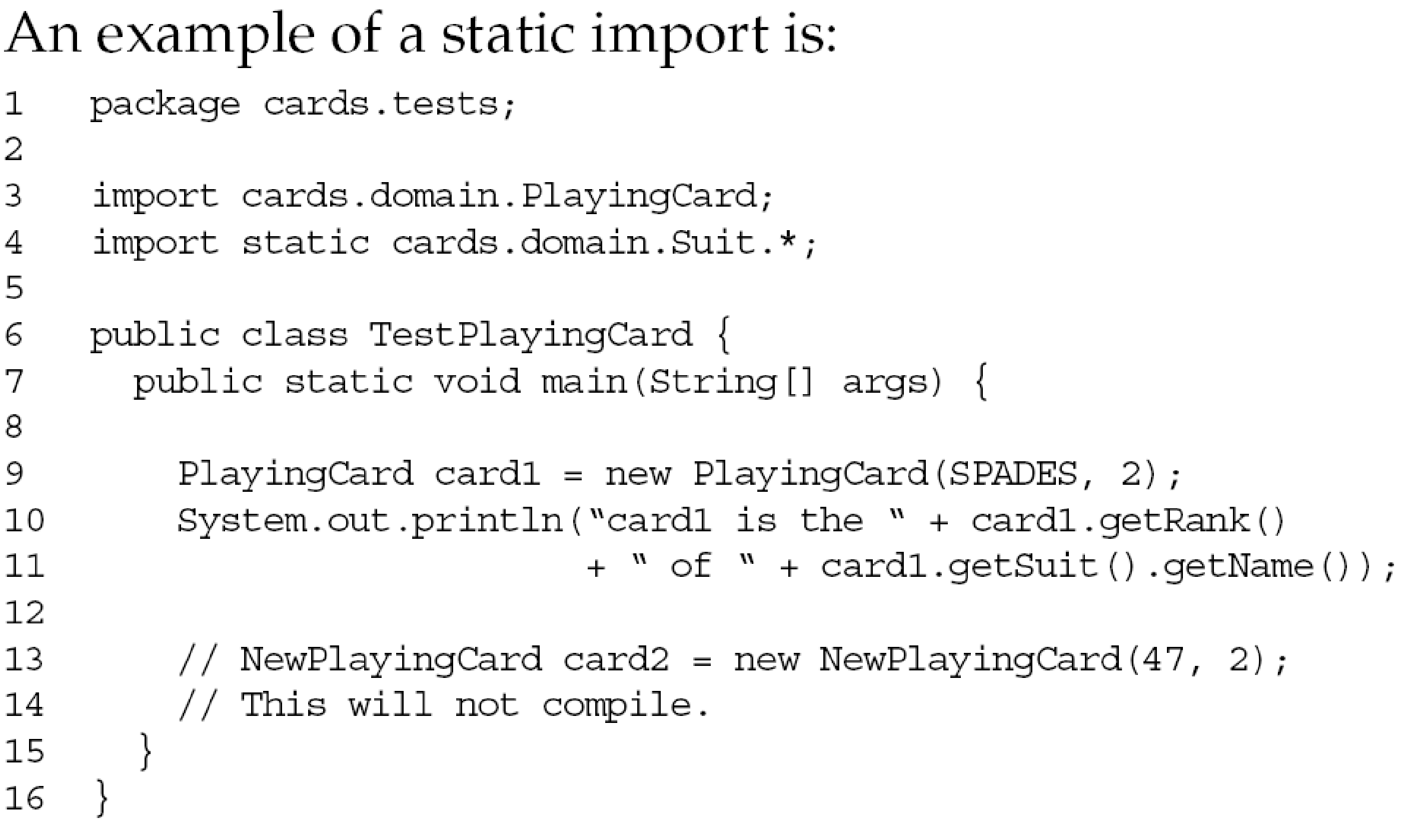
```
import static cards.domain.Suit.SPADES;
OR
import static cards.domain.Suit.*;
```

- There is no need to qualify the static constants:

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

- *Use this feature sparingly.*

Static Imports

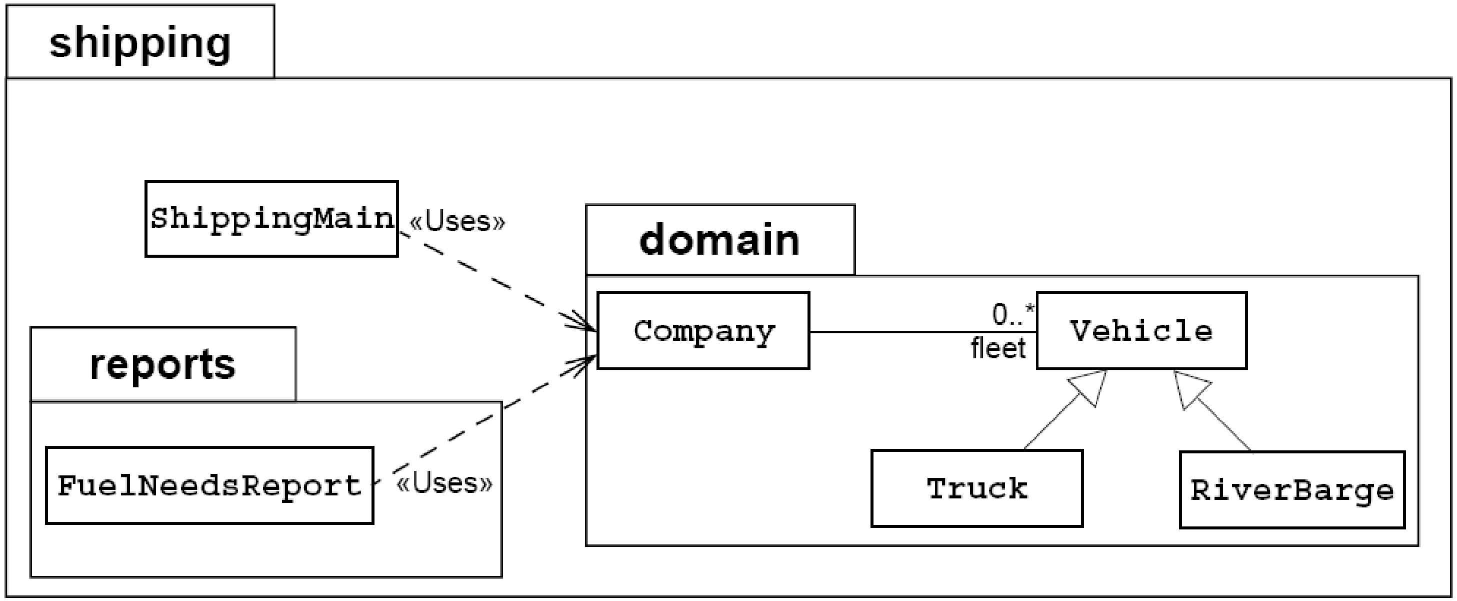


An example of a static import is:

```
package cards.tests;

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

public class TestPlayingCard {
  public static void main(String[] args) {

    PlayingCard card1 = new PlayingCard(SPADES, 2);
    System.out.println(“card1 is the ‟ + card1.getRank()
                        + ‟ of ‟ + card1.getSuit().getName());

    // NewPlayingCard card2 = new NewPlayingCard(47, 2);
    // This will not compile.
  }
}
```

Abstract Classes

The design of the Shipping system looks like this:



Abstract Classes

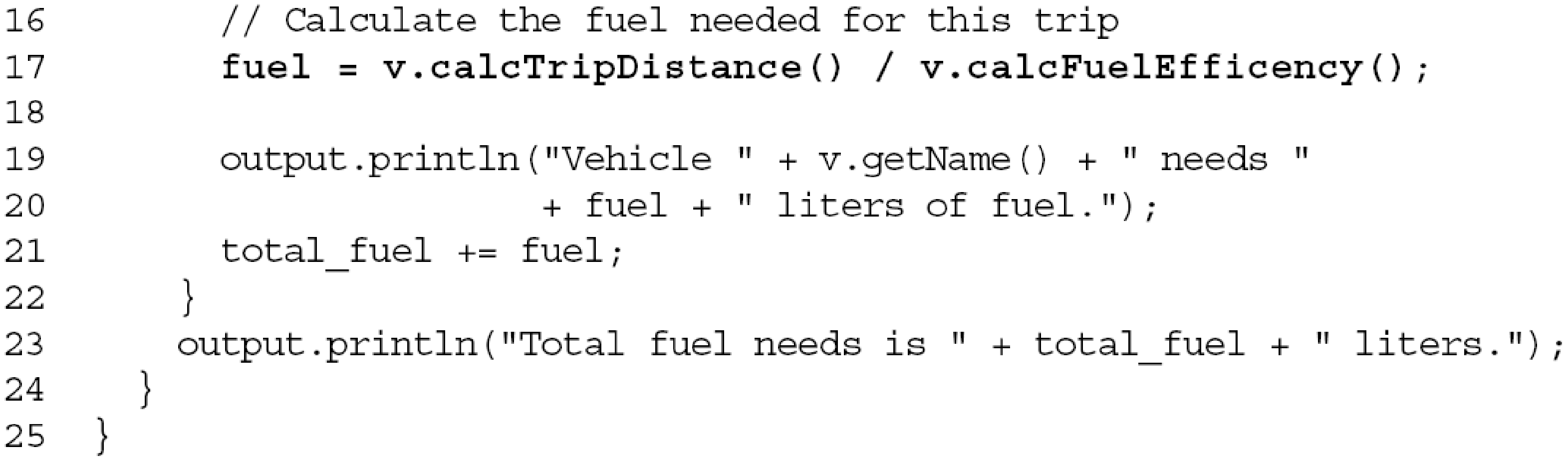
Fleet initialization code is shown here:

```
public class ShippingMain {
  public static void main(String[] args) {
    Company c = new Company();

    // populate the company with a fleet of vehicles
    c.addVehicle( new Truck(10000.0) );
    c.addVehicle( new Truck(15000.0) );
    c.addVehicle( new RiverBarge(500000.0) );
    c.addVehicle( new Truck(9500.0) );
    c.addVehicle( new RiverBarge(750000.0) );

    FuelNeedsReport report = new FuelNeedsReport(c);
    report.generateText(System.out);
  }
}
```

Abstract Classes

```java
public class FuelNeedsReport {
  private Company company;

  public FuelNeedsReport(Company company) {
    this.company = company;
  }

  public void generateText(PrintStream output) {
    Vehicle1 v;
    double fuel;
    double total_fuel = 0.0;

    for ( int i = 0; i < company.getFleetSize(); i++ ) {
      v = company.getVehicle(i);

```

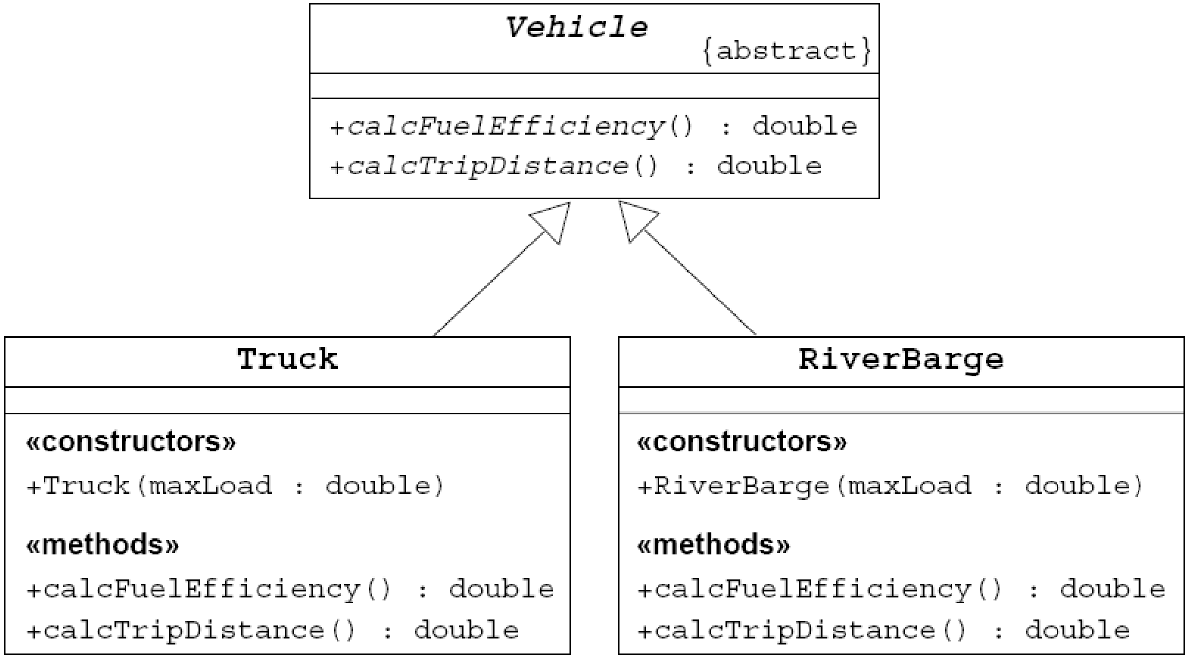
Abstract Classes



```
      // Calculate the fuel needed for this trip
      fuel = v.calcTripDistance() / v.calcFuelEfficency();

      output.println("Vehicle " + v.getName() + " needs "
                     + fuel + " liters of fuel.");
      total_fuel += fuel;
    }
    output.println("Total fuel needs is " + total_fuel + " liters.");
  }
}
```

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.



Vehicle {abstract}

+*calcFuelEfficiency*() : double

+*calcTripDistance*() : double

Truck

«constructors»

+Truck(maxLoad : double)

«methods»

+calcFuelEfficiency() : double

+calcTripDistance() : double

RiverBarge

«constructors»

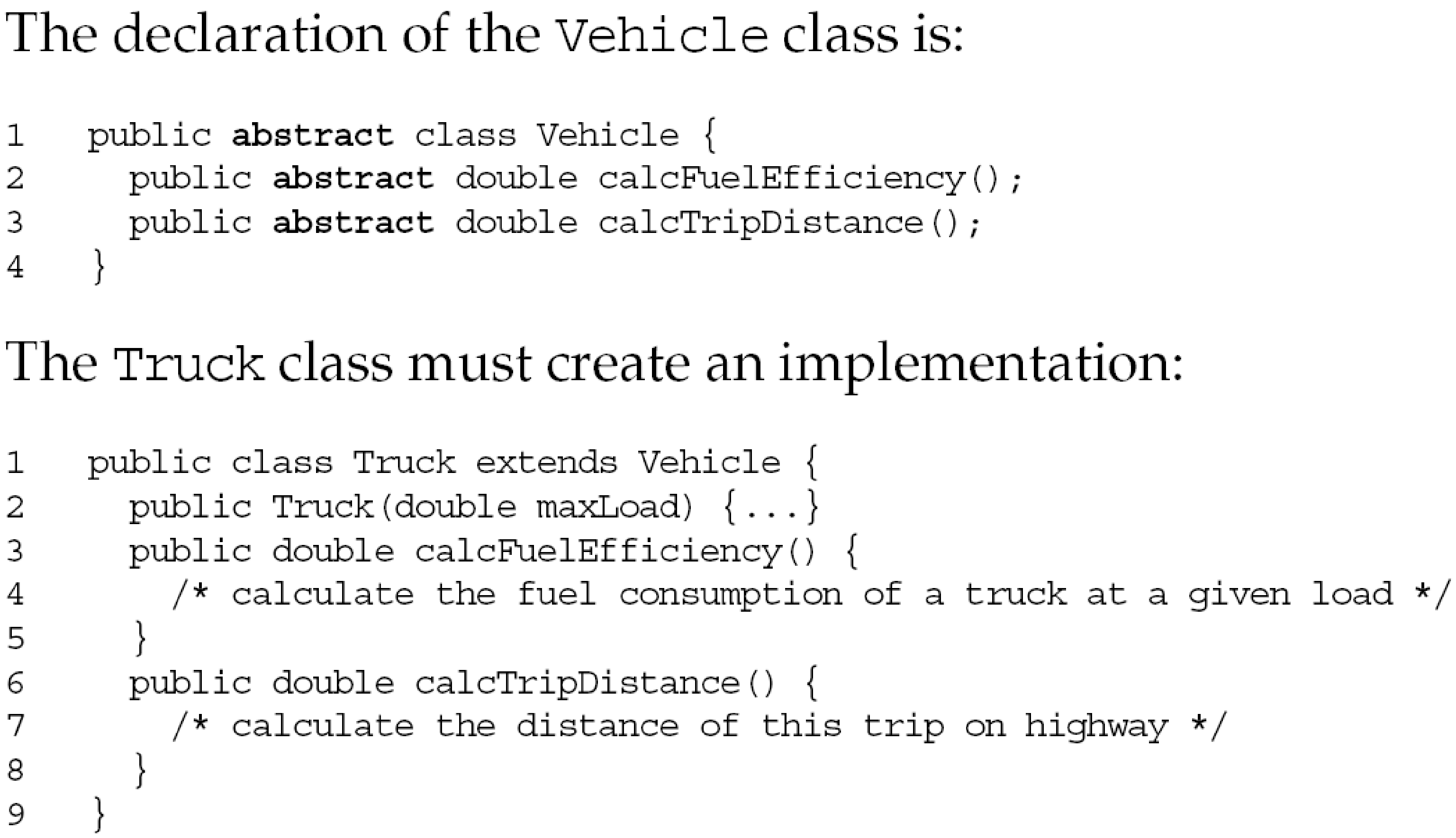
+RiverBarge(maxLoad : double)

«methods»

+calcFuelEfficiency() : double

+calcTripDistance() : double

The Solution



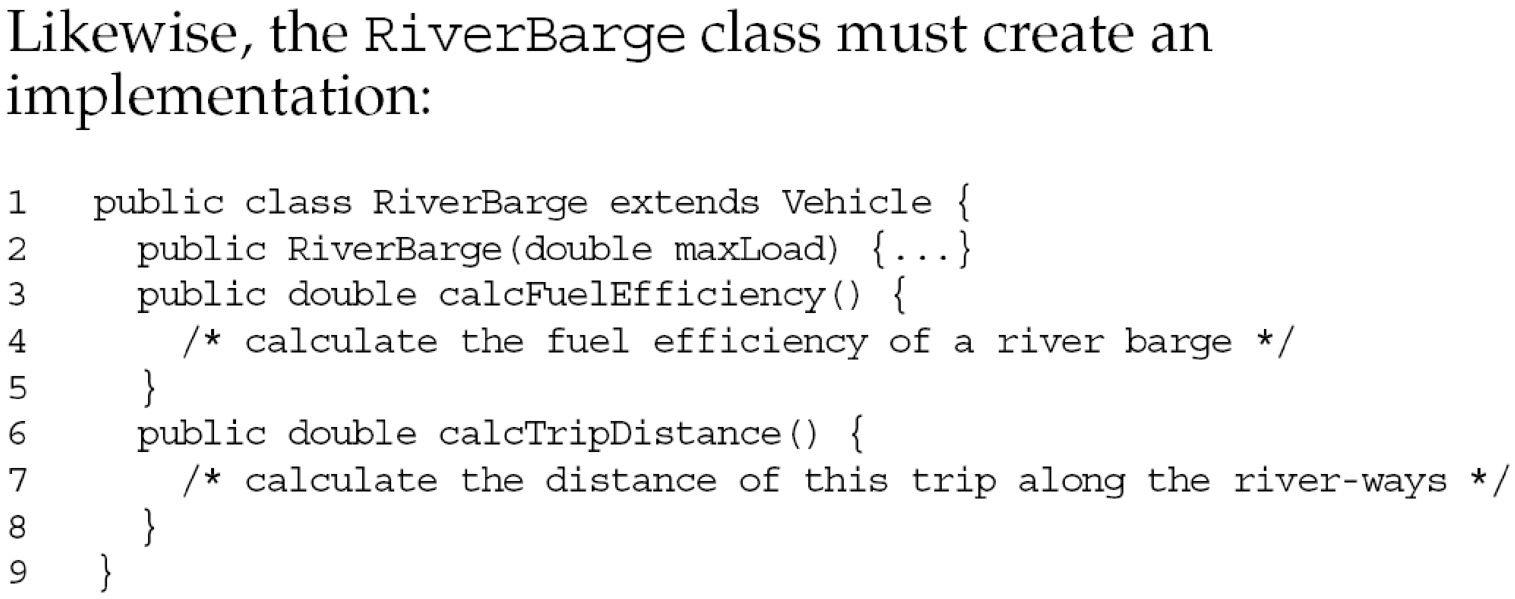
The declaration of the Vehicle class is:

```
public abstract class Vehicle {
  public abstract double calcFuelEfficiency();
  public abstract double calcTripDistance();
}
```

The Truck class must create an implementation:

```
public class Truck extends Vehicle {
  public Truck(double maxLoad) {...}
  public double calcFuelEfficiency() {
    /* calculate the fuel consumption of a truck at a given load */
  }
  public double calcTripDistance() {
    /* calculate the distance of this trip on highway */
  }
}
```

The Solution



Likewise, the RiverBarge class must create an implementation:

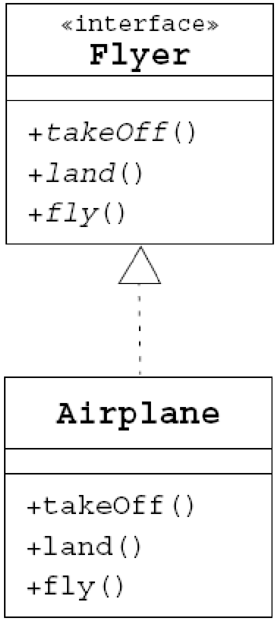
```
public class RiverBarge extends Vehicle {
  public RiverBarge(double maxLoad) {...}
  public double calcFuelEfficiency() {
    /* calculate the fuel efficiency of a river barge */
  }
  public double calcTripDistance() {
    /* calculate the distance of this trip along the river-ways */
  }
}
```

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

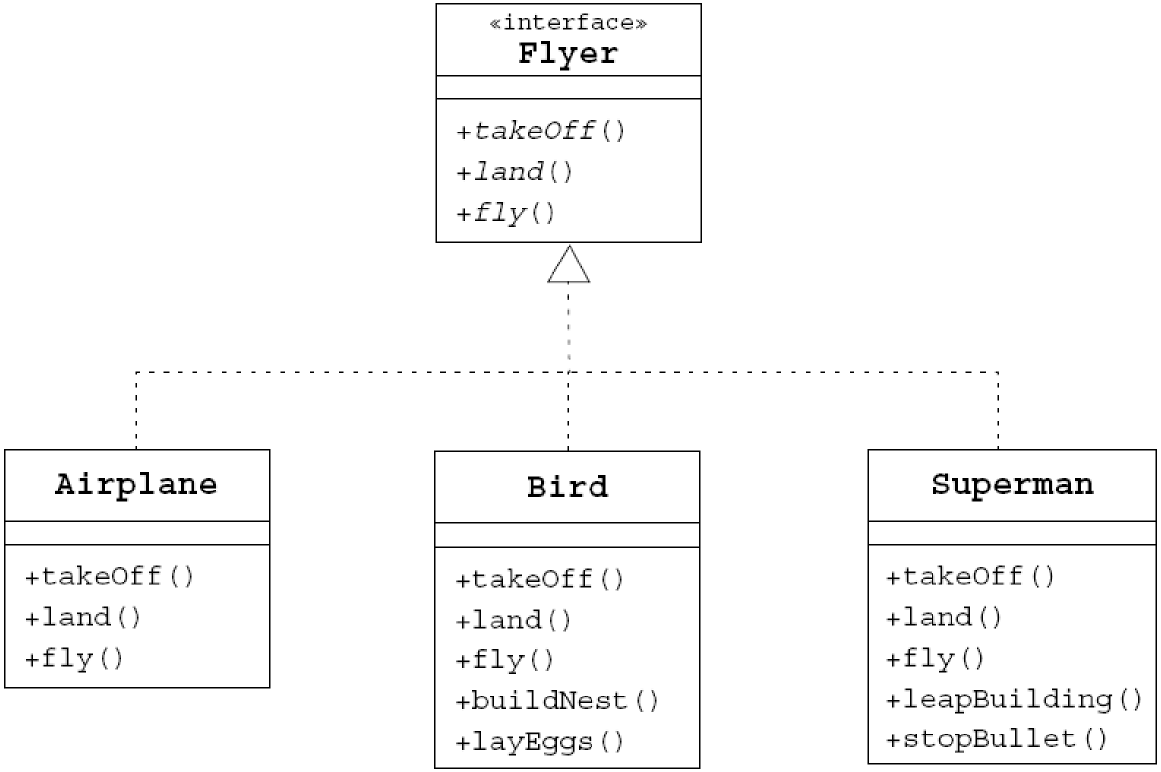


```
public interface Flyer {
  public void takeOff();
  public void land();
  public void fly();
}
```

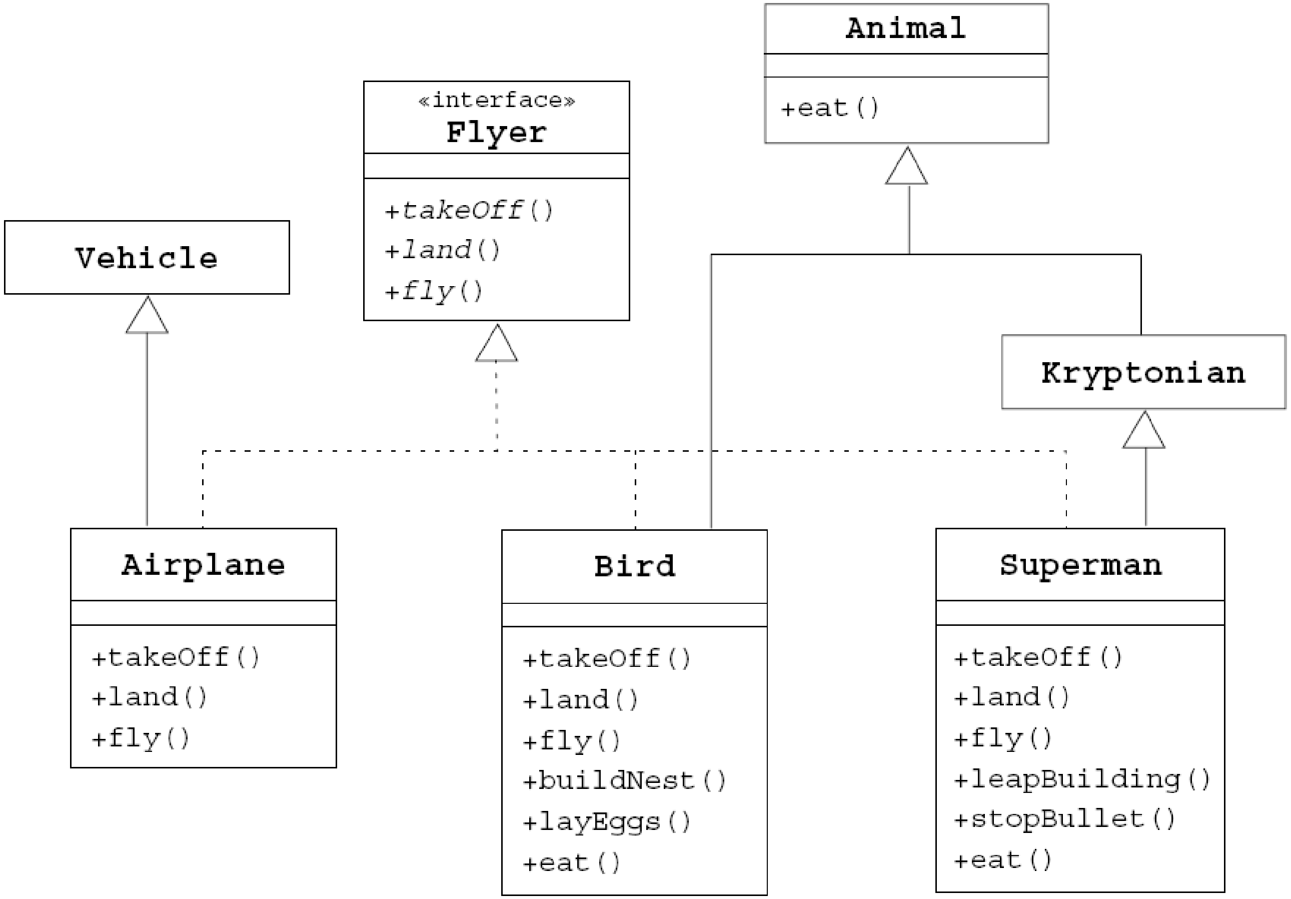
The Flyer Example

```
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



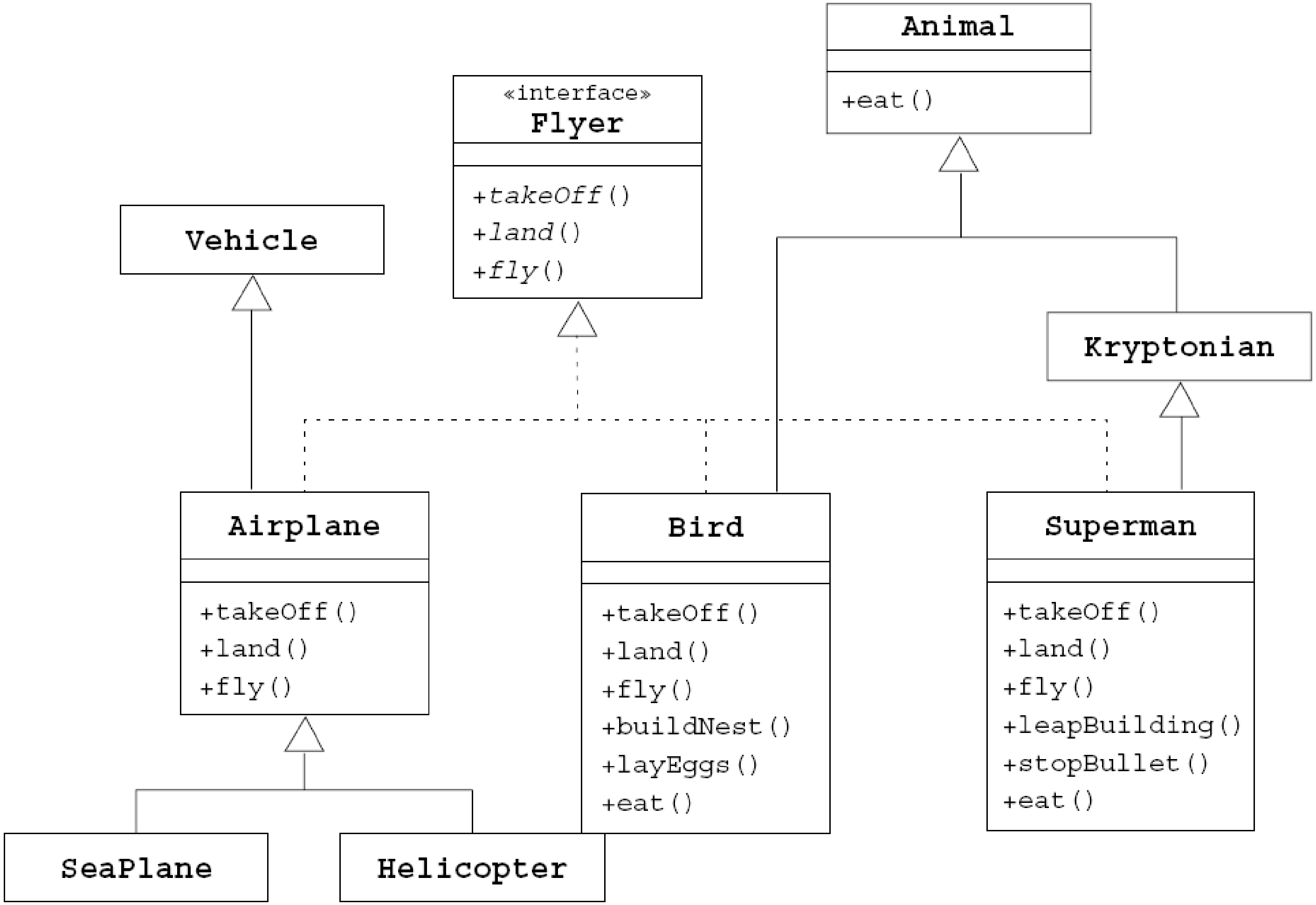
The Flyer Example



The Flyer Example

```
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 */ }
}
```

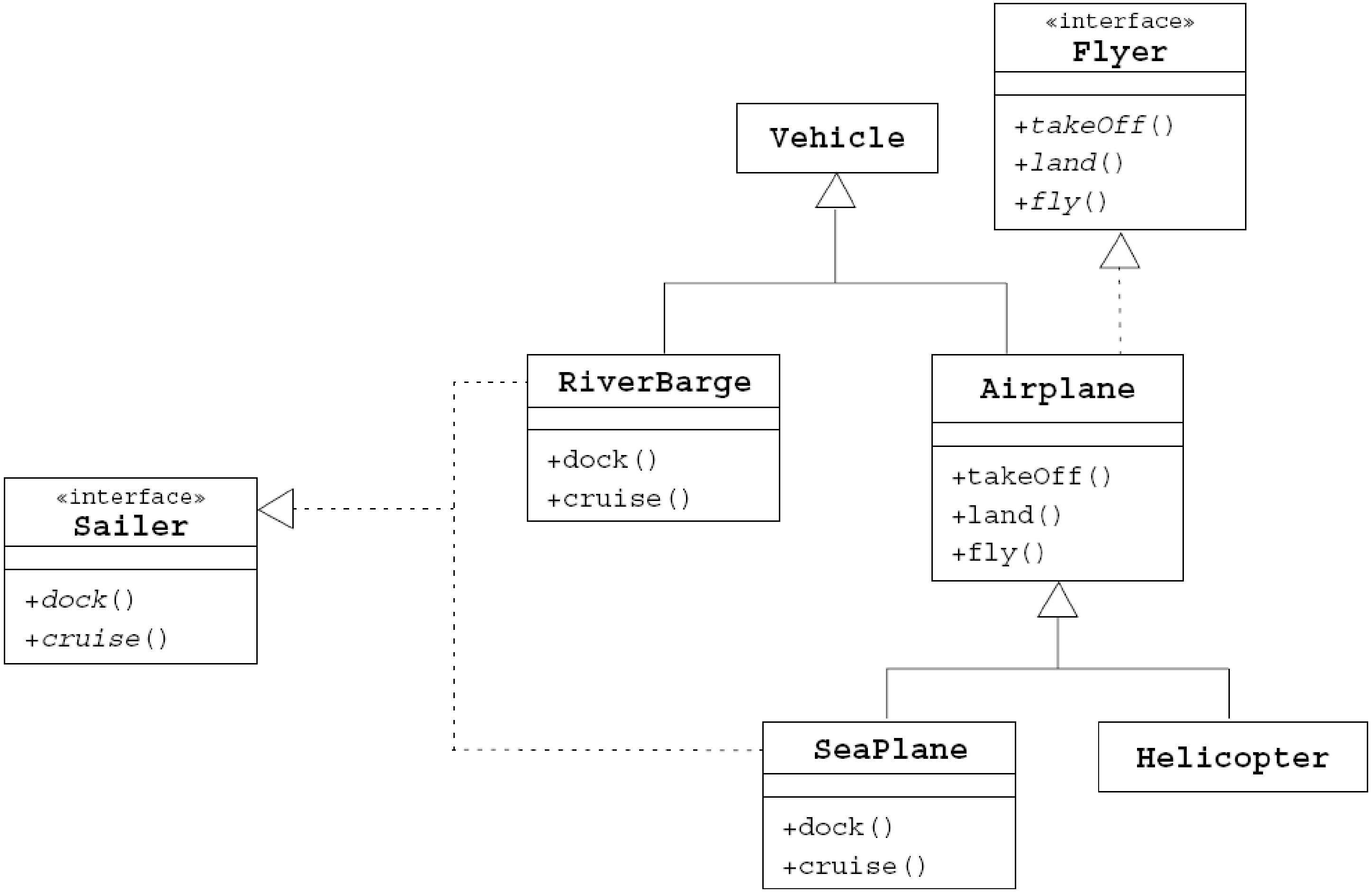
The Flyer Example



The Flyer Example

```
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

```
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