IANT Educom Private Limited

Module 7

Advanced Class Features

Objectives

Upon completion of this module, you should be able to:

- 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

This module completes the discussion about the object-oriented features of the Java technology programming language.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2008 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G.2

IANT Educom Private Limited

Relevance

Discussion – The following questions are relevant to the material presented in this module:

- 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?

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

IANT Educom Private Limited

The `static` Keyword

The `static` keyword declares members (attributes, methods, and nested classes) that are associated with the class rather than the instances of the class.

The following sections describe the most common uses of the static keyword: class variables and class methods.

Class Attributes

Sometimes it is desirable to have a variable that is shared among all instances of a class. For example, you could use this variable as the basis for communication between instances or to keep track of the number of instances that have been created (see Figure 7-1).

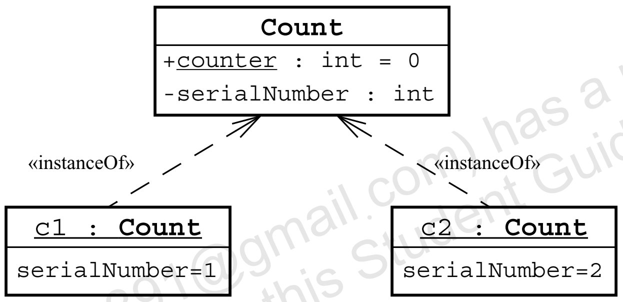

Figure 7-1 UML Object Diagram of the `Count` Class and Two Unique Instances

You achieve this shared effect by marking the variable with the keyword `static`. Such a variable is sometimes called a *class variable* to distinguish it from a member or instance variable, which is not shared.

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

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

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Copyright 2008 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G.2

IANT Educom Private Limited

In this example, every object that is created is assigned a unique serial number, starting at 1 and counting upwards. The variable counter is shared among all instances, so when the constructor of one object increments counter, the next object to be created receives the incremented value.

A static variable is similar in some ways to a global variable in other languages. The Java programming language does not have global variables, but a static variable is a single variable that is accessible from any instance of the class.

If a static variable is not marked as private, you can access it from outside the class. To do this, you do not need an instance of the class, you can refer to it through the class name.

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

Class Methods

Sometimes you need to access program code when you do not have an instance of a particular object available. A method that is marked using the keyword static can be used in this way and is sometimes called a *class method*.

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

  public static int getTotalCount() {
    return counter;
  }

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

You should access methods that are static using the class name rather than an object reference, as follows:

```
public class TestCounter {
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp301@gmail.com) has a non-transferable license to use this Student Guide.

IANT Educom Private Limited

```
  public static void main(String[] args) {
    System.out.println("Number of counter is "
                         + Count2.getTotalCount());
    Count2 count1 = 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
```

Because you can invoke a static method without any instance of the class to which it belongs, there is no this value. The consequence is that a static method cannot access any variables other than the local variables, static attributes, and its parameters. Attempting to access non-static attributes causes a compiler error.

Non-static attributes are bound to an instance and can be accessed through instance references only.

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

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

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2008 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G.2

IANT Educom Private Limited

You should be aware of the following when using static methods:

- You cannot override a `static` method but you can hide it.

  For a method to be *overridden* it must be non-static. Two static methods with the same signature in a class hierarchy simply means that there are two independent class methods. If a class method is applied to an object reference, the method invoked is the one for the class for which the variable was declared.

- The `main()` method is a `static` method because the JVM does not create an instance of the class when executing the `main` method. So if you have member data, you must create an object to access it.

Static Initializers

A class can contain code in *static blocks* that are not part of normal methods. Static block code executes once when the class is loaded. If a class contains more than one static block, they are executed in the order of their appearance in the class.

```
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 code on Line 4 of the `Count4` class uses a static method of the `Integer` class `getInteger(String)`, which returns an `Integer` object that represents the value of a system property. This property, named `myApp.Count4.counter`, is set on the command line using the `-D` option. The `intValue` method on the `Integer` object returns the value as an `int`.

The result is the following:

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

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp391@gmail com) has a non-transferable license to use this Student Guide.

IANT Educom Private Limited

The `final` Keyword

This section describes final classes, final methods, and final variables.

Final Classes

The Java programming language permits you to apply the keyword `final` to classes. If you do this, the class cannot be subclassed. For example, the class `java.lang.String` is a `final` class. This is done for security reasons, because it ensures that if a method references a string, it is a definite string of class `String` and not a string of a class that is a modified subclass of `String` that might have been changed.

Final Methods

You can also mark individual methods as `final`. Methods marked `final` cannot be overridden. For security reasons, you should make a method `final` if the method has an implementation that should not be changed and is critical to the consistent state of the object.

Methods declared `final` can be optimized. The compiler can generate code that causes a direct call to the method, rather than the usual, virtual-method invocation that involves a runtime lookup. Methods marked as `static` or `private` can be optimized by the compiler as if they had been marked `final`, because dynamic binding cannot be applied in either case.

Final Variables

If a variable is marked as `final`, the effect is to make it a constant. Any attempt to change the value of a `final` variable causes a compiler error. The following example shows a `final` variable that is defined properly:

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

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp891@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2008 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G.2

IANT Educom Private Limited

Note – If you mark a variable of reference type (that is, any class type) as `final`, that variable cannot refer to any other object. However, you can change the object's contents, because only the reference itself is `final`.

Blank Final Variables

A *blank final variable* is a final variable that is not initialized in its declaration. The initialization is delayed. A blank final instance variable must be assigned in a constructor, but it can be set once only. A blank final variable that is a local variable can be set at any time in the body of the method, but it can be set once only. The following code fragment is an example of how a blank final variable can be used in a class:

```
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
}
```

Mayur Patel (mayurp331@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

IANT Educom Private Limited

Enumerated Types

A common idiom in programming is to have finite set of symbolic names that represent the values of an attribute. For example, to represent the suits of playing cards you might create a set of symbols: SPADES, HEARTS, CLUBS, and DIAMONDS. This is often called an *enumerated type*.

Note – Do not get confused by the term *enumerated type* and the `java.util.Enumeration` class. They are unrelated in all respects.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2008 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G.2

IANT Educom Private Limited

Old-Style Enumerated Type Idiom

Code 7-1 illustrates the old-style (before J2SE version 5.0) idiom for creating an enumerated type using integer constants for the symbolic names of the enumerated values.

Code 7-1 Old-Style Enumerated Type Example

```
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;
  }

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

Note – The `rank` attribute is also a good candidate for an enumerated type, but this code does not use an enumerated type for brevity.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayur391@gmail.com) has a non-transferable license to use this Student Guide.

IANT Educom Private Limited

Furthermore, the user interface will need to display strings rather than integers to represent the card suit. Code 7-2 shows the `getSuitName` method of the `PlayingCard` class. Notice Lines 37 and 38; this code is used when the `suit` attribute holds an invalid integer value.

Code 7-2 The `getSuitName` Method

```
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:
38          System.err.println("Invalid suit.");
39      }
40      return name;
41    }
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2008 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G.2

IANT Educom Private Limited

Code 7-3 shows the `TestPlayingCard` program that creates two playing card objects and then displays their rank and suit. This program illustrates the primary problem with the old-style of enumerated type idiom. The `PlayingCard` constructor takes two arguments: an integer for the suit and an integer for the rank. The first call to the `PlayingCard` constructor (Line 9) uses an appropriate symbolic constant, but the second call to the `PlayingCard` constructor (Line 14) uses an arbitrary integer value. Both constructor calls are valid from the perspective of the compiler.

Code 7-3 The `TestPlayingCard` Program

```
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  }
19
```

However, when this program is executed you can see that the second card object is invalid and does not display properly:

```
> java cards.tests.TestPlayingCard
card1 is the 2 of Spades
Invalid suit.
card2 is the 2 of
```

The statement `Invalid suit.` is the error message sent by the `getSuitName` method on Line 38 in Code 7-2.

This enumerated type idiom has several problems:

- Not type-safe – Because a `suit` attribute is just an `int`, you can pass in any other `int` value where a suit is required. A programmer could also apply arithmetic operations on two suits, which makes no sense.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

IANT Educom Private Limited

- No namespace – You must prefix constants of an `int` enum with a string (in this case, `SUIT_`) to avoid collisions with other `int` enum types. For example, if you used an `int` enum for the rank value, then you would probably create a set of `RANK_XYZ int` constants.
- Brittle character – Because `int` enums are compile-time constants, they are compiled into clients that use them. If a new constant is added between two existing constants or the order is changed, clients must be recompiled. If they are not, they will still run, but their behavior will be undefined.
- Uninformative printed values – Because they are just `int` values, if you print one out all you see is a number, which tells you nothing about what it represents, or even what type it is. That is why the `PlayingCard` class needs the `getSuitName` method.

The New Enumerated Type

The Java SE version 5.0 of the Java programming language includes a type-safe enumerated type feature. Code 7-4 shows an enumerated type for the suits of a playing card. You can think of the `Suit` type as a class with a finite set of values that are given the symbolic names listed in the type definition. For example, `Suit.SPADES` is of type `Suit`.

Code 7-4 The `Suit` Enumerated Type

```
package cards.domain;

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

Code 7-5 shows the `PlayingCard` class using the `Suit` type for the data type of the `suit` attribute.

Code 7-5 The `PlayingCard` Class Using the `Suit` Type

```
package cards.domain;

public class PlayingCard {

  private Suit suit;
  private int rank;
```

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Copyright 2008 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G.2

IANT Educom Private Limited

```

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

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

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.
Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

IANT Educom Private Limited

This solves the type-safety issues with the old-style enumerated type idiom. Code 7-6 shows an updated test program. Line 14, if uncommented, would result in a compiler error because the `int` value of `47` is not of type `Suit`.

Code 7-6 The `TestPlayingCard` Program

```
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.getSuitName());
13
14      // PlayingCard card2 = new PlayingCard(47, 2);
15      // This will not compile.
16    }
17  }
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2008 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G.2

IANT Educom Private Limited

Advanced Enumerated Types

Unfortunately, the `PlayingCard` class still requires a `getSuitName` method to display a user-friendly name in the user interface. If the program were to display the actual `Suit` value, it would show the symbolic name of the type value; for example, `Suit.SPADES` would display as `SPADES`. This is more user-friendly than “`0`,” but is still not as user-friendly as `Spades`.

Furthermore, the name of the suit should not be part of the `PlayingCard` class, but rather part of the `Suit` type. The new enumerated type feature permits both attributes and methods, just like regular classes. Code 7-7 shows a refinement of the first `Suit` type (Code 7-4 on page 7-13) with a `name` attribute and `getName` method. Notice the use of proper information hiding with the private attribute and the public accesser method.

Code 7-7 The `Suit` Type with an Attribute

```
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  }
```

An `enum` constructor should always use the private accessibility. The arguments to the constructor are supplied after each declared value. For example, on Line 4 the string `“Spades”` is the argument to the enum constructor for the `SPADES` value. Enumerated types can have any number of attributes and methods.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

IANT Educom Private Limited

Finally, Code 7-8 shows the modified test program that uses the new `getName` method on the `Suit` type. On Line 12, the `getSuit` method returns a `Suit` value and the `getName` method returns the `name` attribute of that `Suit` value.

Code 7-8 The `TestPlayingCard` Program Using `Suit` Methods

```
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  }
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2008 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G.2

IANT Educom Private Limited

Static Imports

If you have to access the static members of a class, then it is necessary to qualify the references with the class from which they come. This is also true of the enumeration type values. Line 10 of Code 7-8 on page 7-17 shows accessing the `SPADES` value of the `Suit` type using the dot-notation `Suit.SPADES`.

J2SE version 5.0 provides the static import feature that enables unqualified access to static members without having to qualify them with the class name. Code 7-9 shows the use of static imports. Line 4 tells the compiler to include the enumerated type values (or any static member of that type) in the symbol table when compiling this program. Therefore, Line 9 can use `SPADES` without the `Suit.` namespace prefix.

Code 7-9 The `TestPlayingCard` Program Using Static Imports

```
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   }
```

Caution – Use static imports sparingly. If you overuse the static import feature, it can make your program unreadable and unmaintainable, polluting its namespace with all of the static members that you import. Readers of your code (including you, a few months after you wrote it) will not know from which class a static member comes. Importing *all* of the static members from a class can be very harmful to readability; if you need one or two members only, import them individually. Used appropriately, static import can make your program *more* readable, by removing the boilerplate of repetition of class names.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp301@gmail.com) has a non-transferable license to use this Student Guide.

IANT Educom Private Limited

Abstract Classes

In the shipping example, suppose that the system needed to supply a weekly report that lists each vehicle in the company's fleet and the fuel needs for their upcoming trips. Assume that the Shipping System has a `ShippingMain` class that populates the company's vehicle fleet list and generates the Fuel Needs report.

Figure 7-2 shows the UML model of the company and its heterogeneous collection of vehicles (the `fleet` association).

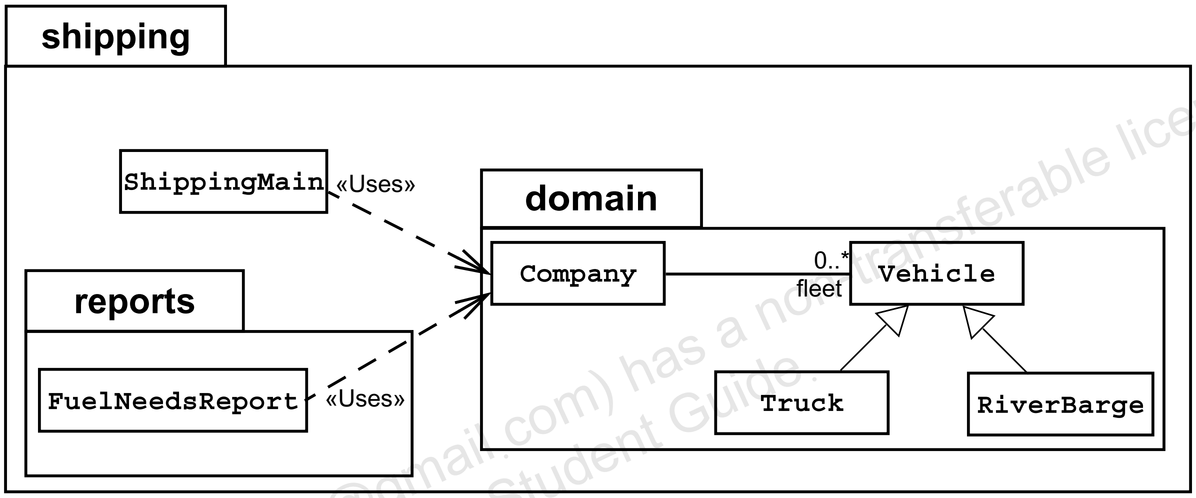

Figure 7-2 UML Model of the Company Fleet

The `ShippingMain` program shows the initialization of an example fleet.

```
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);
  }
}
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.
Mayur Patel (mayurp201@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2008 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G.2

IANT Educom Private Limited

You should write the report code as follows:

```
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);

      // 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 calculation for the fuel requirements is the trip distance (in kilometers) divided by the vehicle's fuel efficiency (in kilometers per liter).

The Problem

The calculations to determine fuel efficiency of a truck as compared with a river barge might differ radically. The `Vehicle` class can not supply these two methods, but its subclasses (`Truck` and `RiverBarge`) can.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

IANT Educom Private Limited

The Solution

The Java programming language enables a class designer to specify that a superclass declares a method that does not supply an implementation. This is called an *abstract method*. The implementation of this method is supplied by the subclasses. Any class with one or more abstract methods is called an *abstract class* (see Figure 7-3).

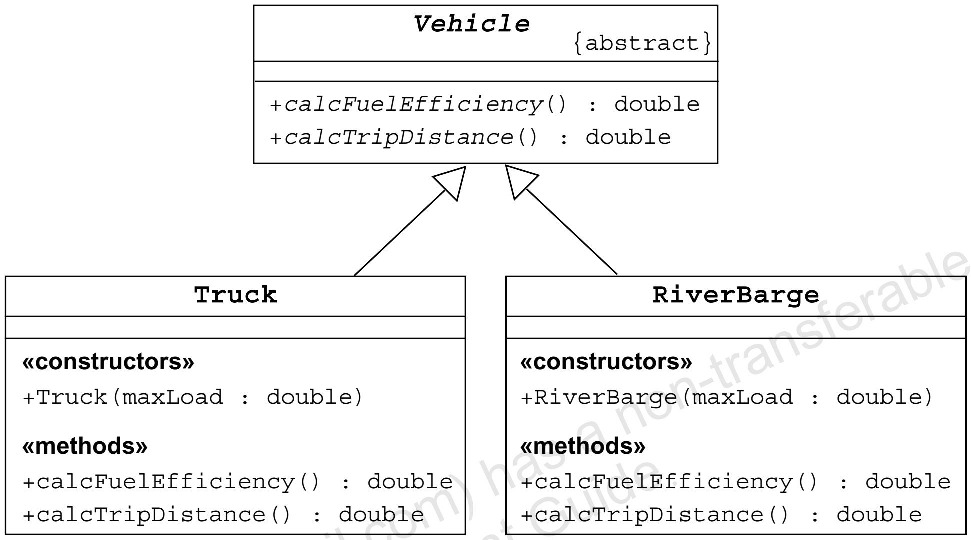
Vehicle
{abstract}
+calcFuelEfficiency() : double
+calcTripDistance() : double
Truck
«constructors»
+Truck(maxLoad : double)
«methods»
+calcFuelEfficiency() : double
+calcTripDistance() : double
RiverBarge
«constructors»
+RiverBarge(maxLoad : double)
«methods»
+calcFuelEfficiency() : double
+calcTripDistance() : double

Figure 7-3 UML Model of the Abstract `Vehicle` Class

Figure 7-3 presents a UML model of the solution. `Vehicle` is an abstract class with two public, abstract methods.

Note – UML uses the italic font to indicate abstract elements in a class diagram. You may also indicate an abstract class with the `{abstract}` constraint flag in the name compartment.

The Java compiler prevents you from instantiating an abstract class. For example, the statement `new Vehicle()` is illegal.

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:

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.
Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide

Copyright 2008 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G.2

IANT Educom Private Limited

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

The `RiverBarge` class must create an implementation:

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

However, abstract classes can have data attributes, concrete methods, and constructors. For example, the `Vehicle` class might include `load` and `maxLoad` attributes and a constructor to initialize them. It is a good practice to make these constructors `protected` rather than `public`, because it is only meaningful for these constructors to be invoked by the subclasses of the abstract class. Making these constructors protected makes it more obvious to the programmer that the constructor should not be called from arbitrary classes.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurbg1@gmail.com) has a non-transferable license to use this Student Guide.

IANT Educom Private Limited

Interfaces

The *public interface* of a class is a contract between the *client code* and the class that provides the service. Concrete classes implement each method. However, an abstract class can defer the implementation by declaring the method to be abstract, and a Java interface declares only the contract and no implementation.

A concrete class implements an interface by defining all methods declared by the interface. Many classes can implement the same interface. These classes do not need to share the same class hierarchy. Also, a class can implement more than one interface. This is described in the following sections.

As with abstract classes, use an interface name as a type of reference variable. The usual dynamic binding takes effect. References are cast to and from interface types, and you use the `instanceof` operator to determine if an object's class implements an interface.

Note – All methods declared in an interface are `public` and `abstract`, no matter if we explicitly mention these modifiers in the code or not. Similarly, all attributes are `public`, `static` and `final`; in other words, you can only declare constant attributes.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2008 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G.2

IANT Educom Private Limited

The Flyer Example

Imagine a group of objects that all share the same ability: they fly. You can construct a public interface, called `Flyer`, that supports three operations: `takeOff`, `land`, and `fly`. (see Figure 7-4).

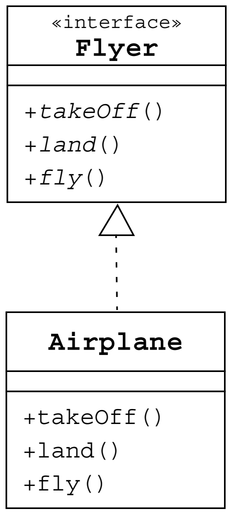

Figure 7-4 The `Flyer` Interface and `Airplane` Implementation

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

```
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
  }
}
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.
Mayur Patel (mayur551@gmail.com) has a non-transferable license to use this Student Guide.

IANT Educom Private Limited

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

There can be multiple classes that implement the `Flyer` interface, as shown in Figure 7-5. An airplane can fly, a bird can also fly, Superman can fly, and so on.

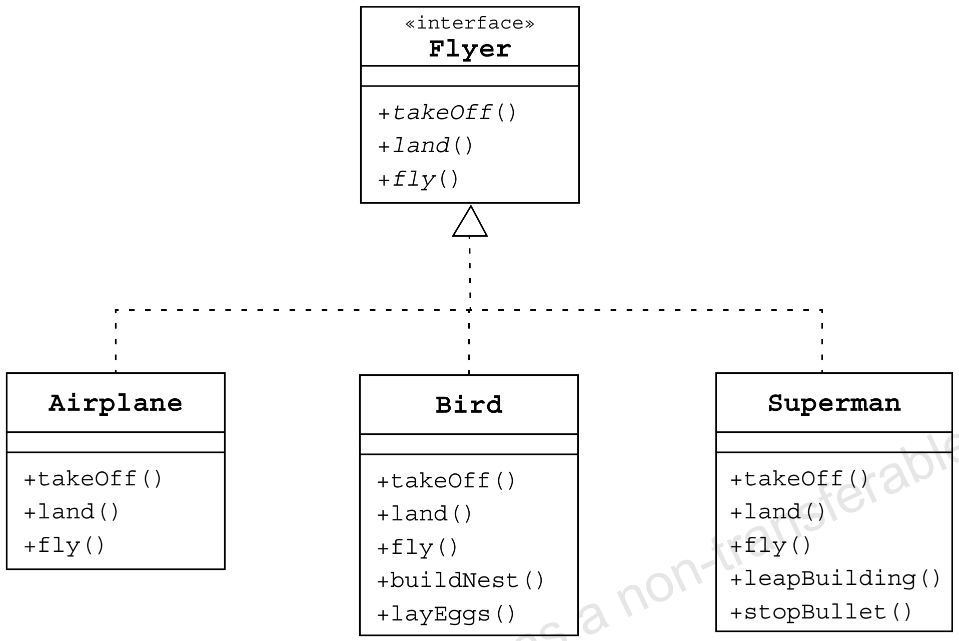

Figure 7-5 Multiple Implementations of the `Flyer` Interface

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

An `Airplane` is a `Vehicle`, and it can fly. A `Bird` is an `Animal`, and it can fly. These examples show that a class can inherit from one class but also implement some other interface.

IANT Educom Private Limited

This sounds like multiple inheritance, but it is not quite that. The danger of multiple inheritance is that a class could inherit two distinct implementations of the same method. This is not possible with interfaces because an interface method declaration supplies no implementation, as shown in Figure 7-6.

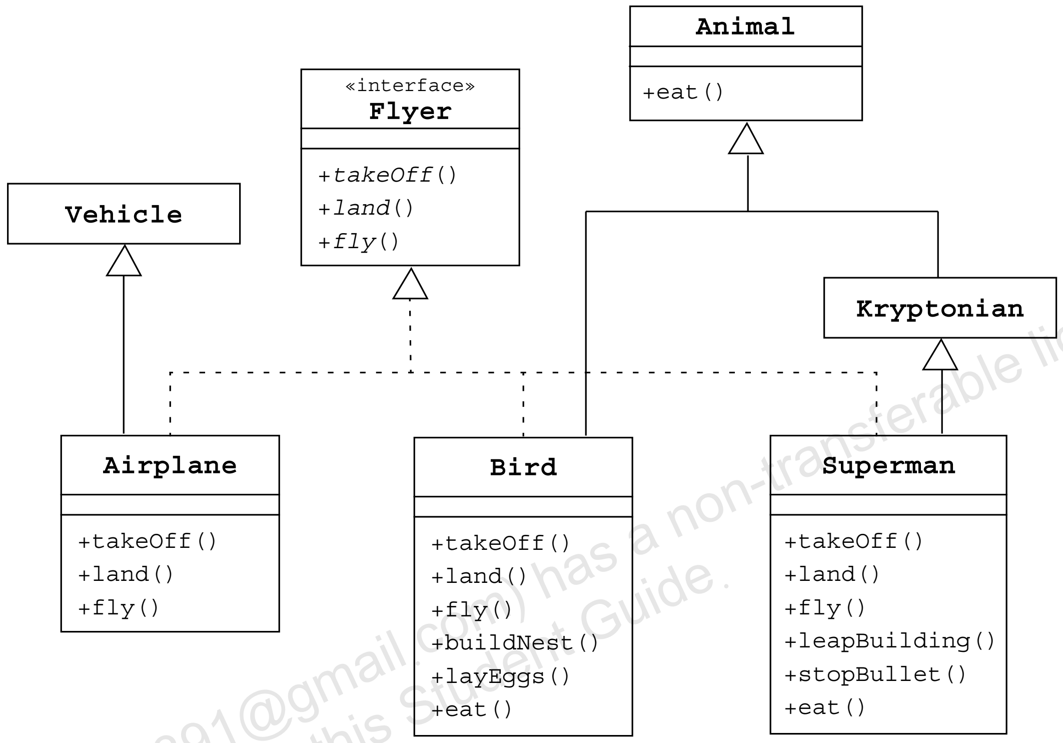

Figure 7-6 A Mixture of Inheritance and Implementation

The following describes the Bird class:

```
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 extends clause must come before the implements clause. The Bird class can supply its own methods (buildNest and layEggs), as well as override the Animal class methods (eat).

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.
Mayur Patel (mayur.p391@gmail.com) has a non-transferable license to use this Student Guide.

IANT Educom Private Limited

Suppose that you are constructing an aircraft control software system. It must grant permission to land and take off for flying objects of all types. This is shown in Figure 7-7.

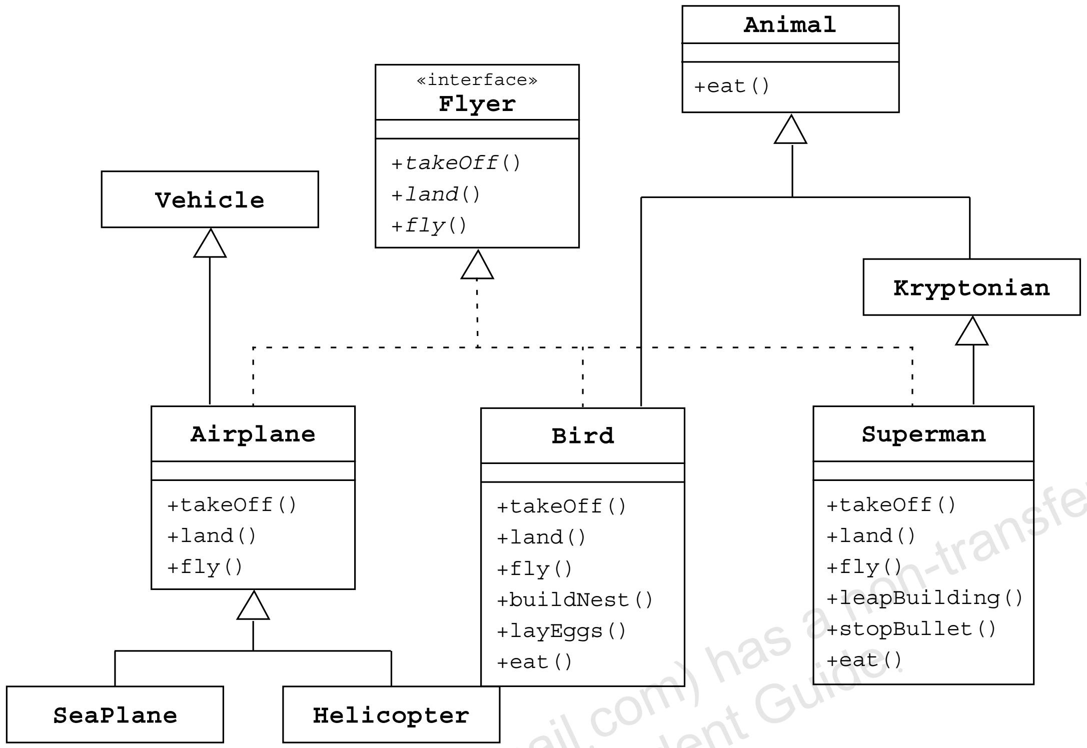

Figure 7-7 Class Hierarchy for the Airport Example

The code for the airport could look like the following:

```
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();
  }
}
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.
Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2008 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G.2

IANT Educom Private Limited

Multiple Interface Example

A class can implement more than one interface. Not only can the SeaPlane fly, but it can also sail. The SeaPlane class extends the Airplane class, so it inherits that implementation of the Flyer interface. The SeaPlane class also implements the Sailer interface. This is shown in Figure 7-8.

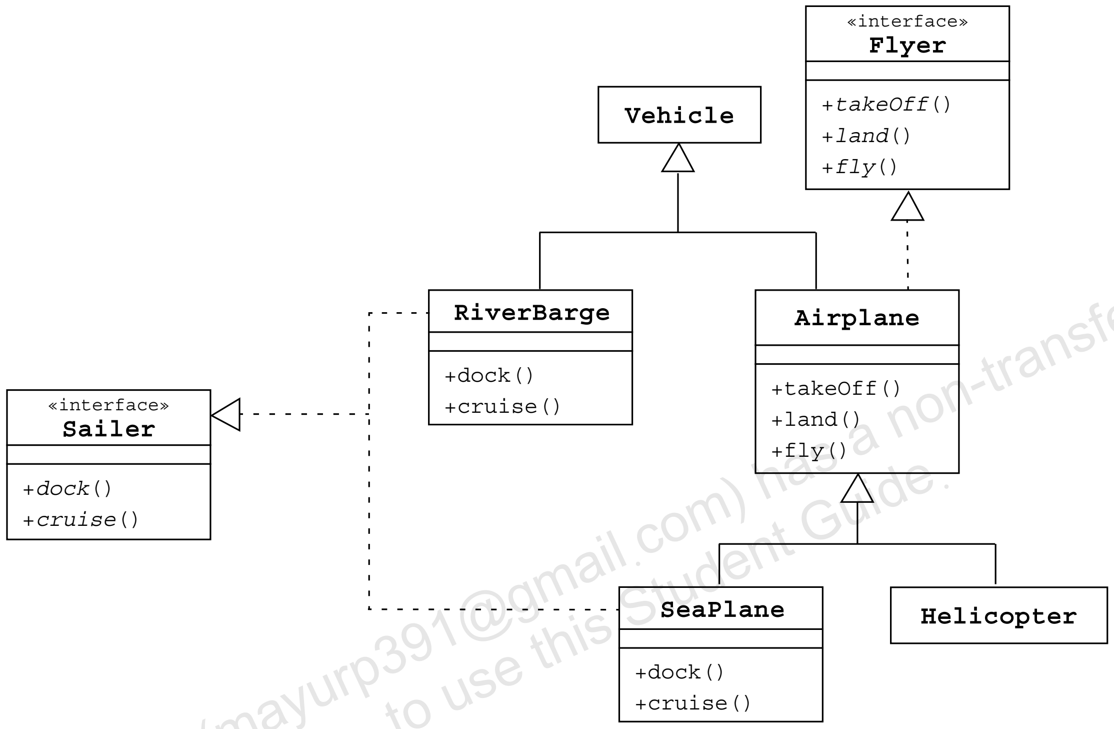

Figure 7-8 An Example of Multiple Implementations

Now write the Harbor class to give docking permission:

```
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();
  }
}
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

IANT Educom Private Limited

The seaplane can take off from Metropolis airport and dock in Boston harbor.

Uses of Interfaces

You use interfaces to:

- Declare methods that one or more classes are expected to implement
- Reveal an object's programming interface without revealing the actual body of the class (this can be useful when shipping a package of classes to other developers)
- Capture similarities between unrelated classes without forcing a class relationship
- Simulate multiple inheritance by declaring a class that implements several interfaces

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.

Copyright 2008 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G.2

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Mayur Patel (mayurp391@gmail.com) has a non-transferable license to use this Student Guide.