# Module 7

## Advanced Class Features

# Objectives

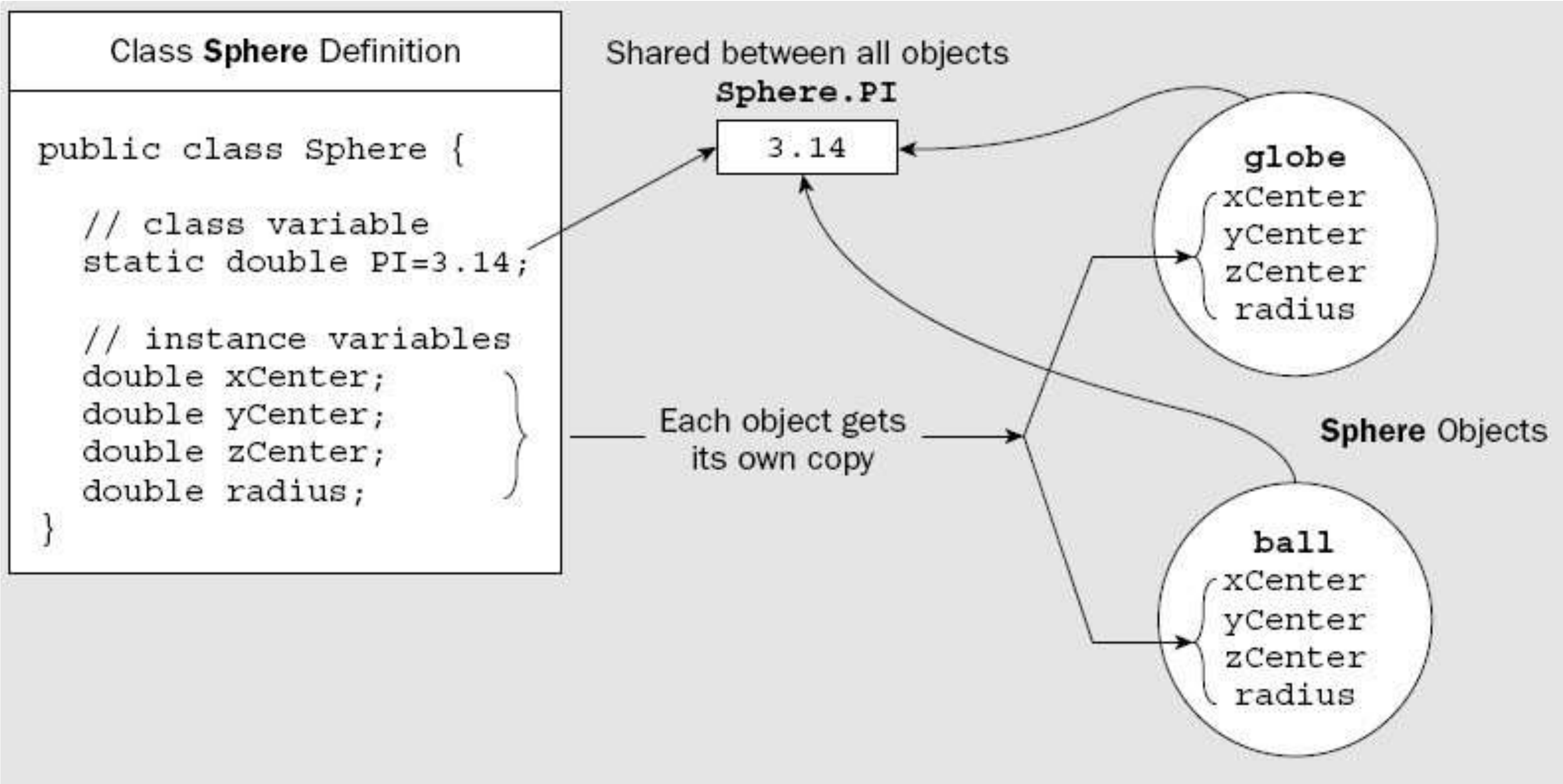
- Create static attributes, methods, and initializers
- The Singleton Design Pattern
- Create final classes, methods, and variables
- Constants are static final attributes.
- Create abstract classes and methods
- Template Method Design Pattern
- Create and use an interface
- Inner Classes

# The *static* Keyword

# The `static`Keyword

- The `static`keyword is used as a modifier on attributes, methods, and initializers.
- 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 and Instance Attributes

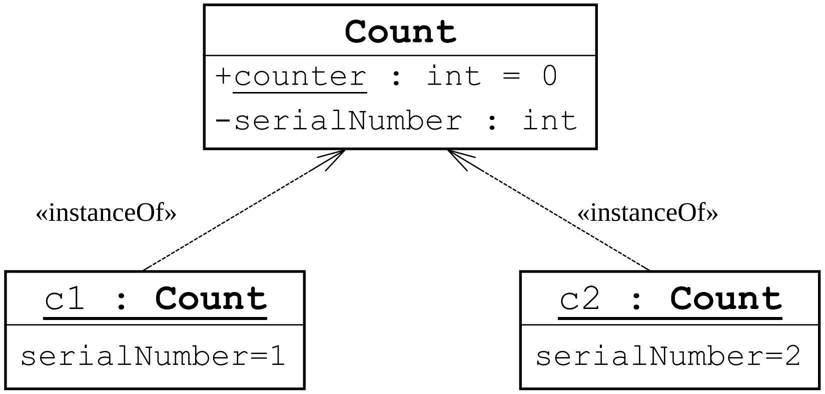


Class **Sphere** Definition

```
public class Sphere {

   // class variable
   static double PI=3.14;

   // instance variables
   double xCenter;
   double yCenter;
   double zCenter;
   double radius;
}
```

# Class Attributes

Class attributes are shared among all instances of a class:



```
public class Count
  { private int
  serialNumber;              = 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;

  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;

  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();
 }
}
```

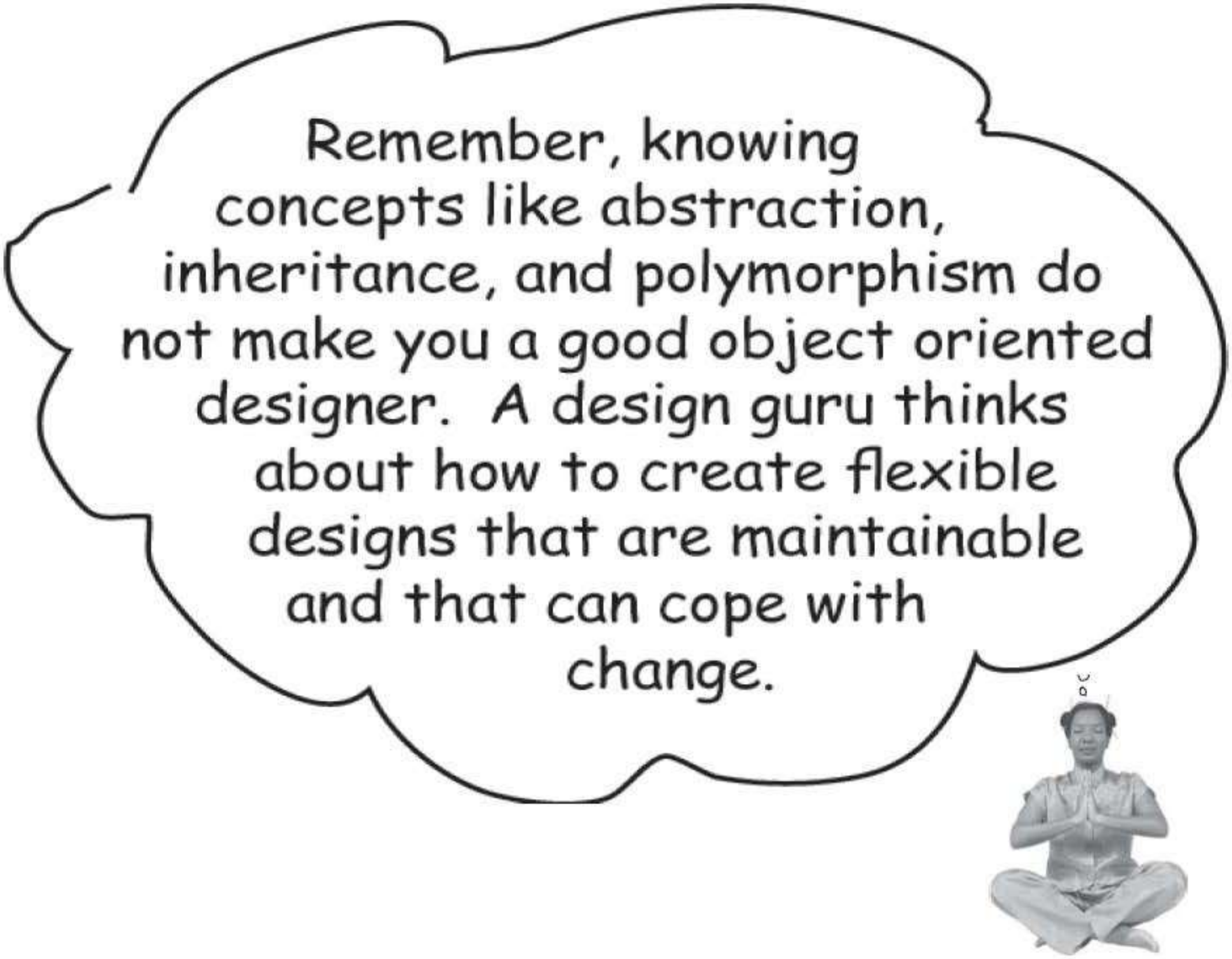
*Integer.getInteger()* Determines the integer value of the system property with the specified name.

```
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
```

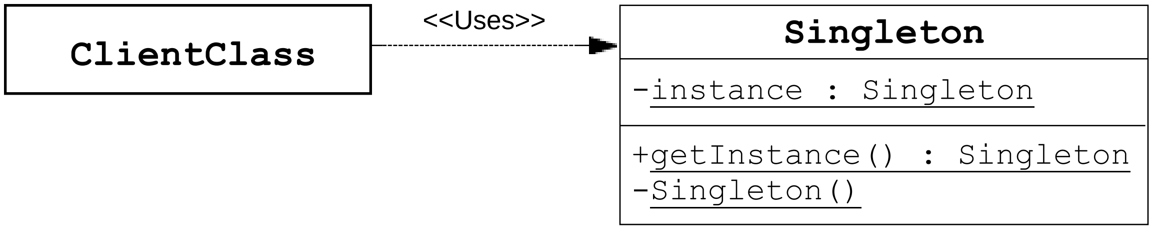
# The Singleton Design Pattern

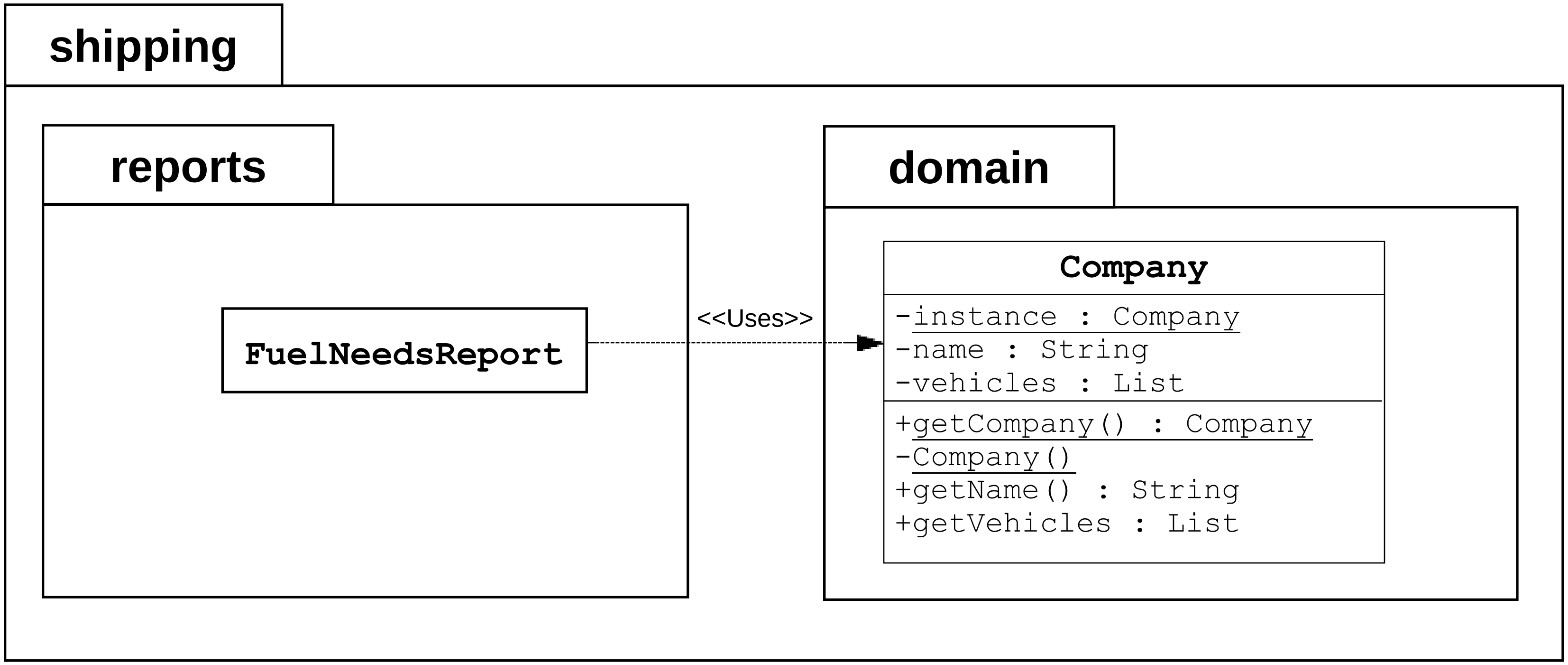

Remember, knowing concepts like abstraction, inheritance, and polymorphism do not make you a good object oriented designer. A design guru thinks about how to create flexible designs that are maintainable and that can cope with change.

# Design Patterns

- **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
  - holds just a single class!

# The Singleton DesignPattern





# Implementing the Singleton Design Pattern

## The Singleton code:

```
package shipping.domain;

public class Company {
  private static Company instance = new Company();
  private String name;
  private Vehicle[] fleet;

  public static Company getCompany() {
    return instance;
  }

  private Company() {...}

  // more Company code ...
}
```

## Usage code:

```
package shipping.reports;

import shipping.domain.*;

public class FuelNeedsReport {
  public void generateText(PrintStream output)
    { Company c = Company.getCompany();
    // use Company object to retrieve the fleet vehicles
  }
}
```

# The *final* Keyword

# 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.

# Blank Final Variables

```
public class Customer {

  private final long customerID;

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

  public long getID()
    { return
    customerID;

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

  // more declarations

}
```

# Final Variables

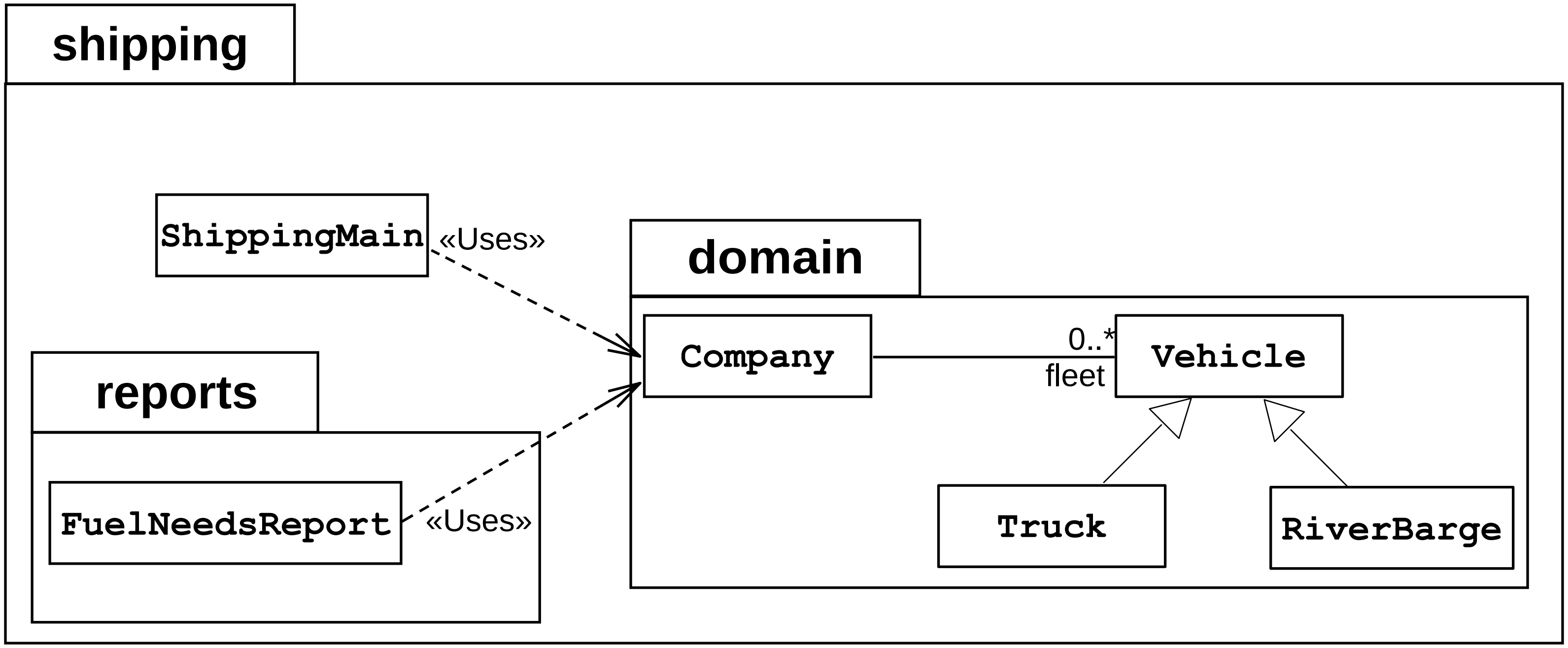
Constants are static final variables.

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

# The *abstract* Keyword

# 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) );
                  new Truck(15000.0) );
    c.addVehicle( new RiverBarge(500000.0) );
                  new Truck(9500.0) );
    c.addVehicle( new RiverBarge(750000.0) );

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

# Abstract Classes

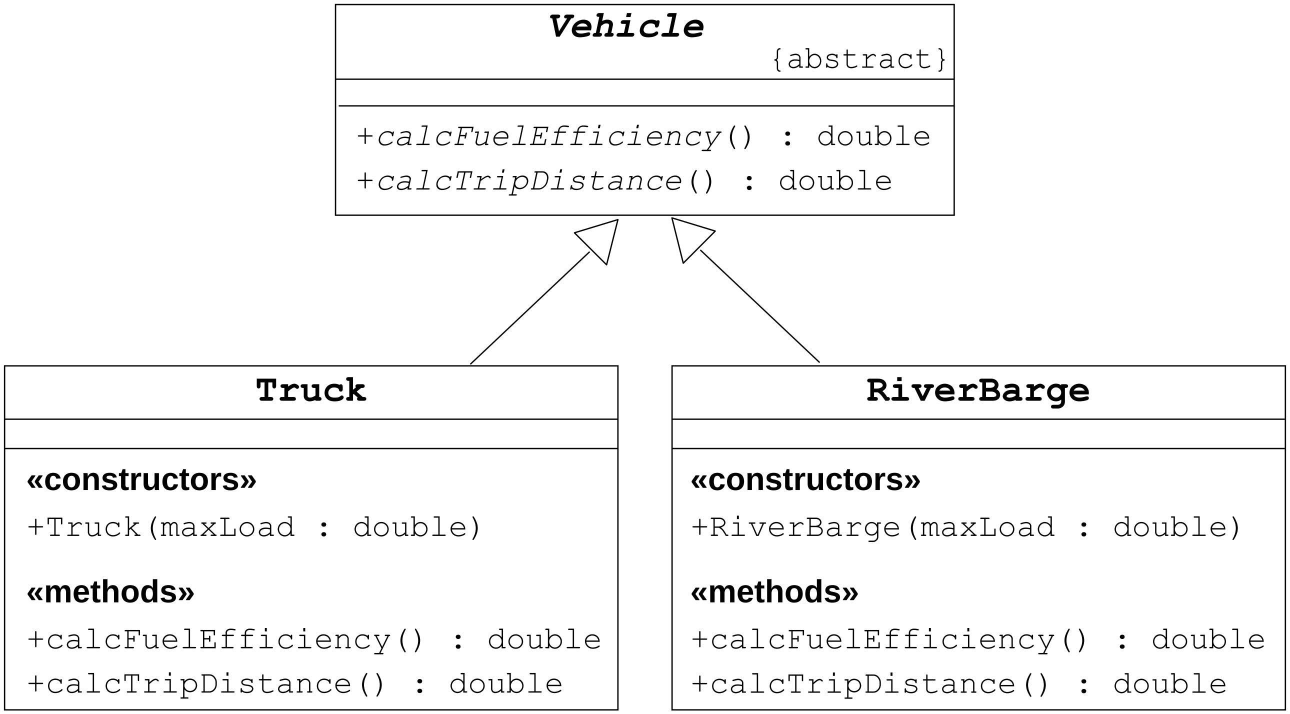
```
public class FuelNeedsReport
  { private Company company;

  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:

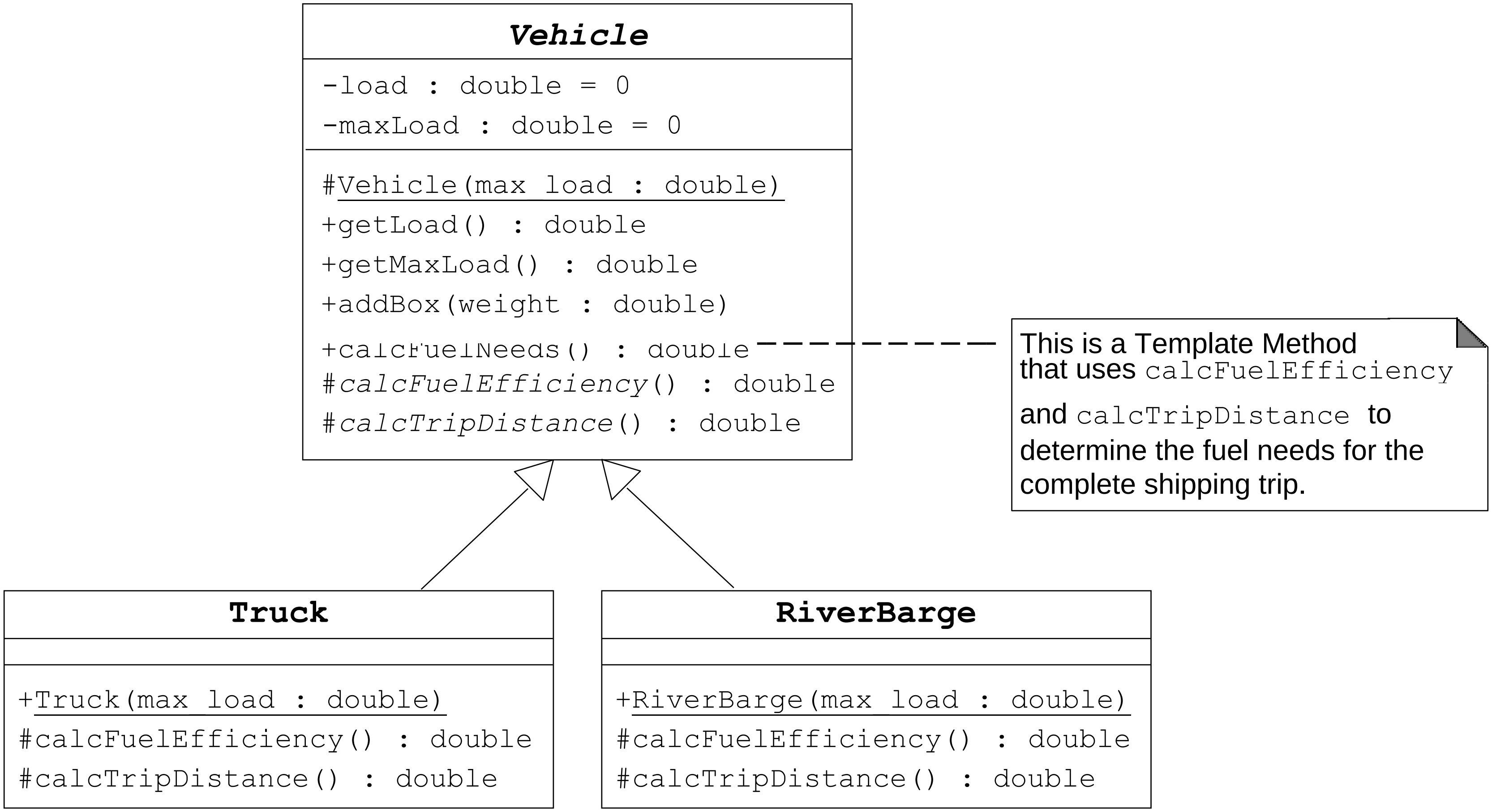
```
public class Truck extends Vehicle
  { public Truck(double maxLoad)
  {...} public double
    /* calculate the fuel consumption of a truck at a given load */
  }
  public double calcTripDistance() {
    /* calculate the distance of this trip on highway */
  }
}
```

# 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
  }
    /* calculate the distance of this trip along the river-ways */
  }
}
```

# Template Method Design Pattern



```
Vehicle
-load : double = 0
-maxLoad : double = 0
#Vehicle(max_load : double)
+getLoad() : double
+getMaxLoad() : double
+addBox(weight : double)
+calcFuelNeeds() : double
#calcFuelEfficiency() : double
#calcTripDistance() : double
```

This is a Template Method that uses `calcFuelEfficiency` and `calcTripDistance` to determine the fuel needs for the complete shipping trip.

```
Truck
+Truck(max_load : double)
#calcFuelEfficiency() : double
#calcTripDistance() : double
```

```
RiverBarge
+RiverBarge(max_load : double)
#calcFuelEfficiency() : double
#calcTripDistance() : double
```

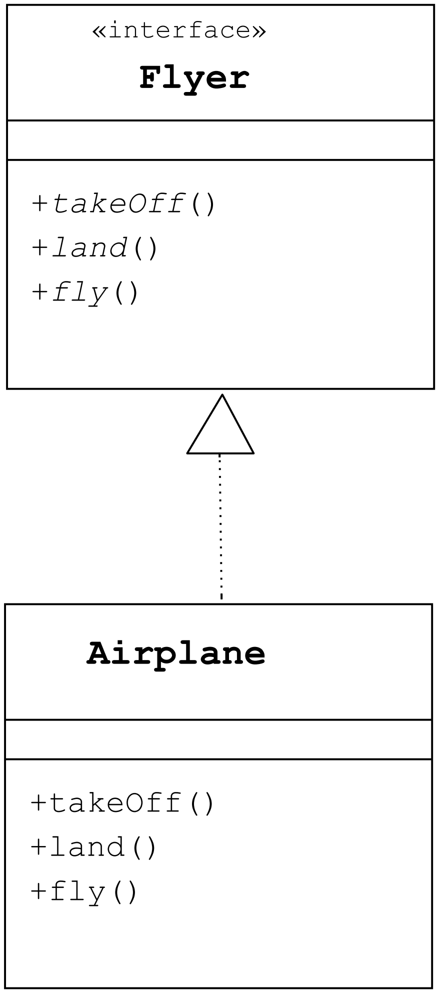
# Interfaces

# 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



«interface»
**Flyer**

+*takeOff*()
+*land*()
+*fly*()

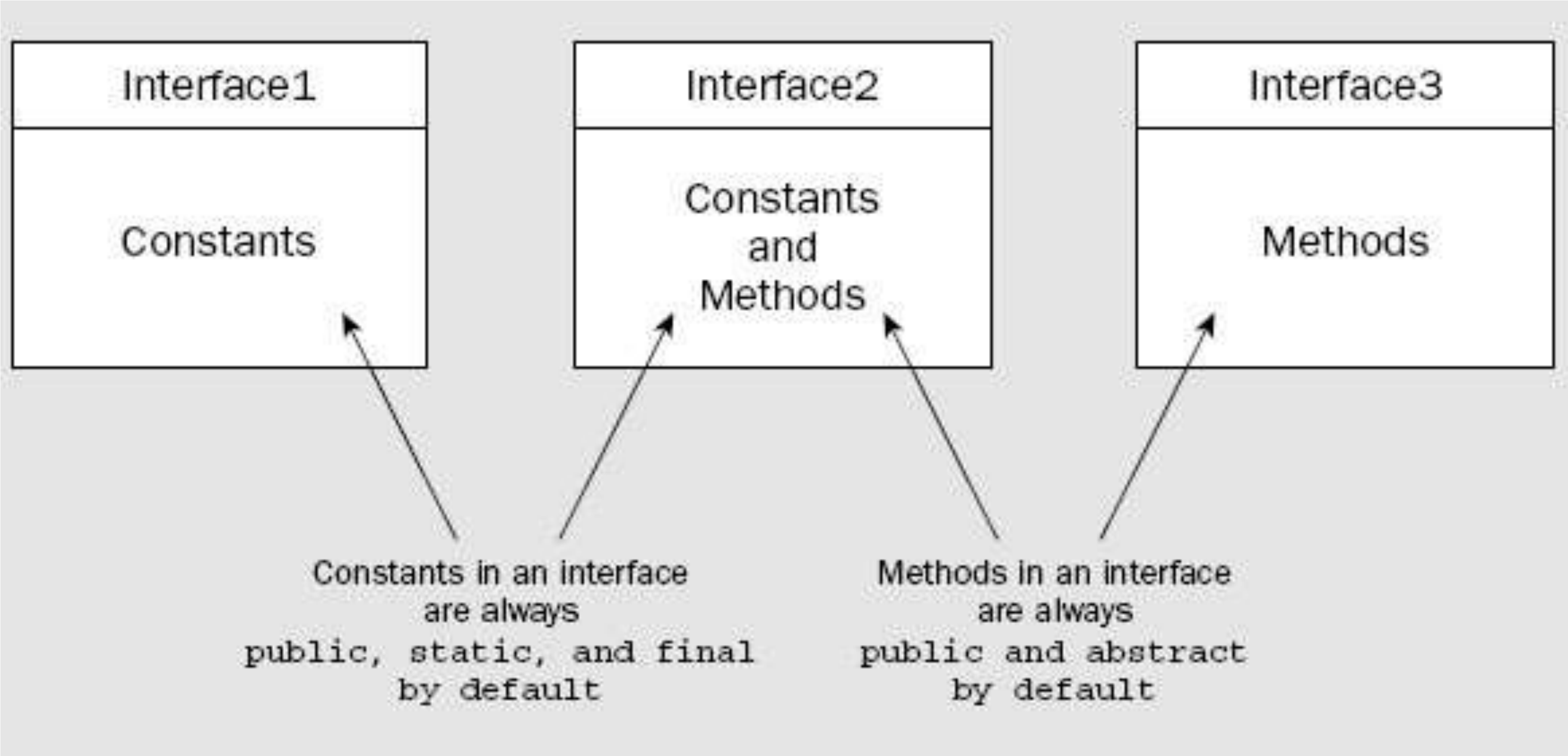
**Airplane**

+takeOff()
+land()
+fly()

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

}
```

# The Flyer Example

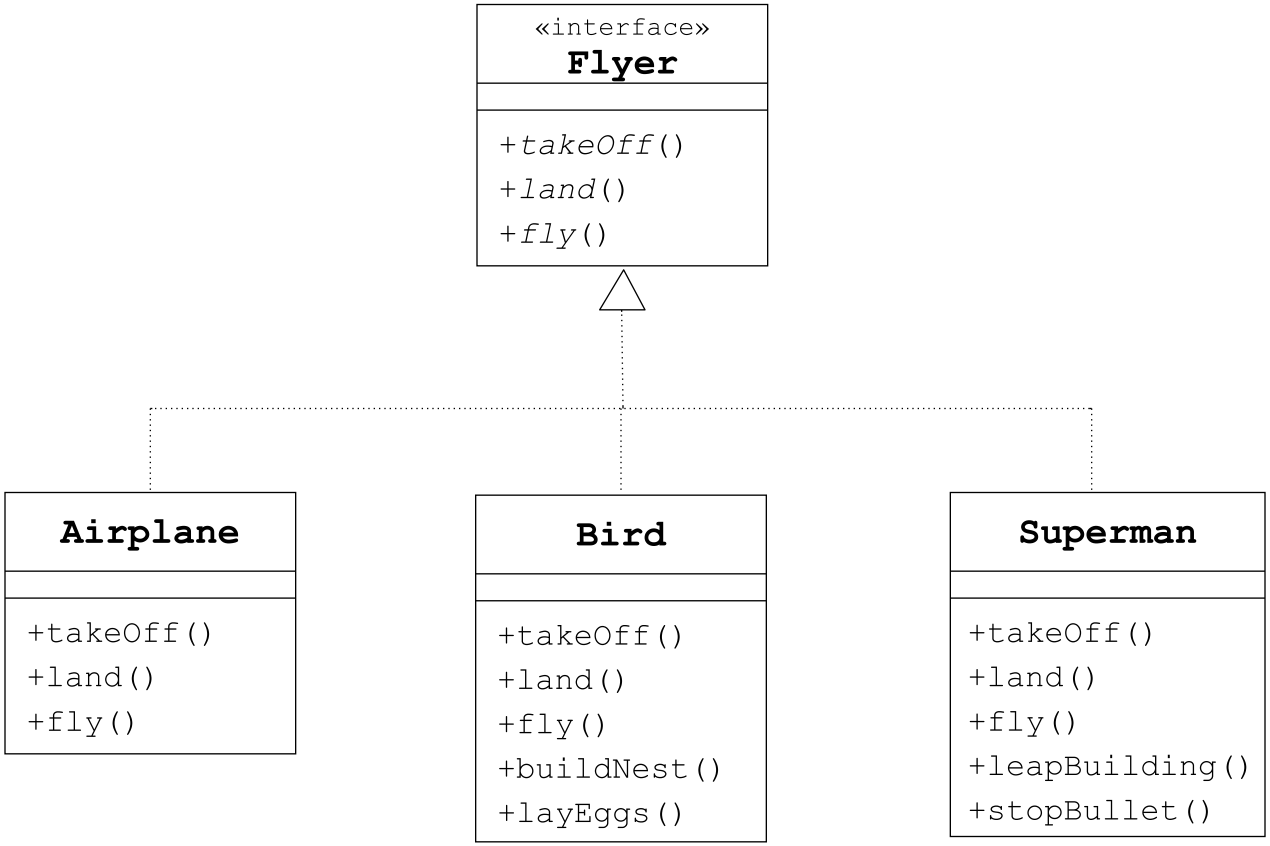
```
public class Airplane implements Flyer {
  public void takeOff() {
    // accelerate until lift-off
    // raise landing gear
  }
  public void land() {
    // lower landing gear
    // decelerate and lower flaps until touch-down
    // apply brakes
  }
  public void fly() {
    // keep those engines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



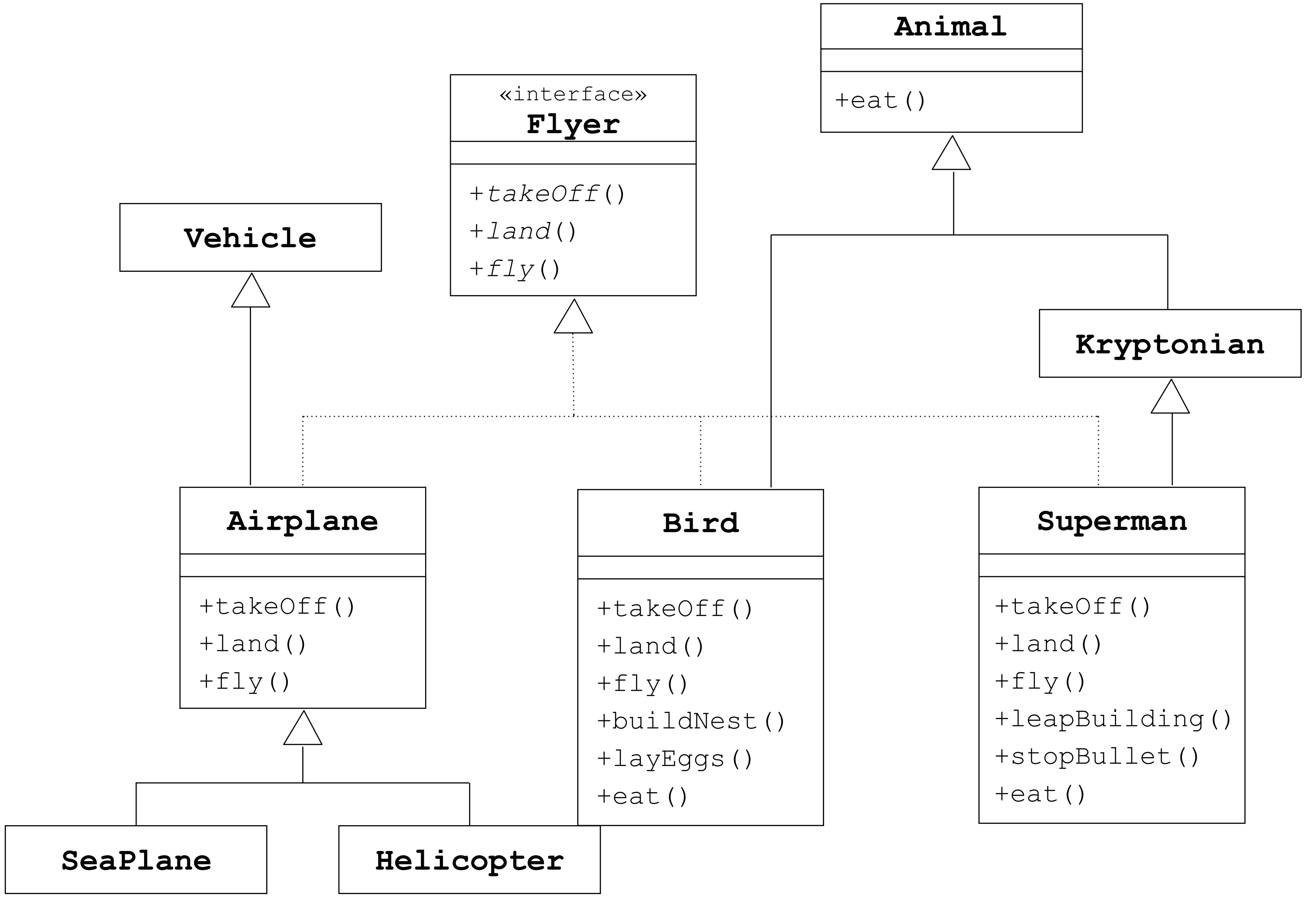
Constants in an interface are always `public, static, and final` by default

Methods in an interface are always `public and abstract` by default

# The Flyer Example



# The Flyer Example



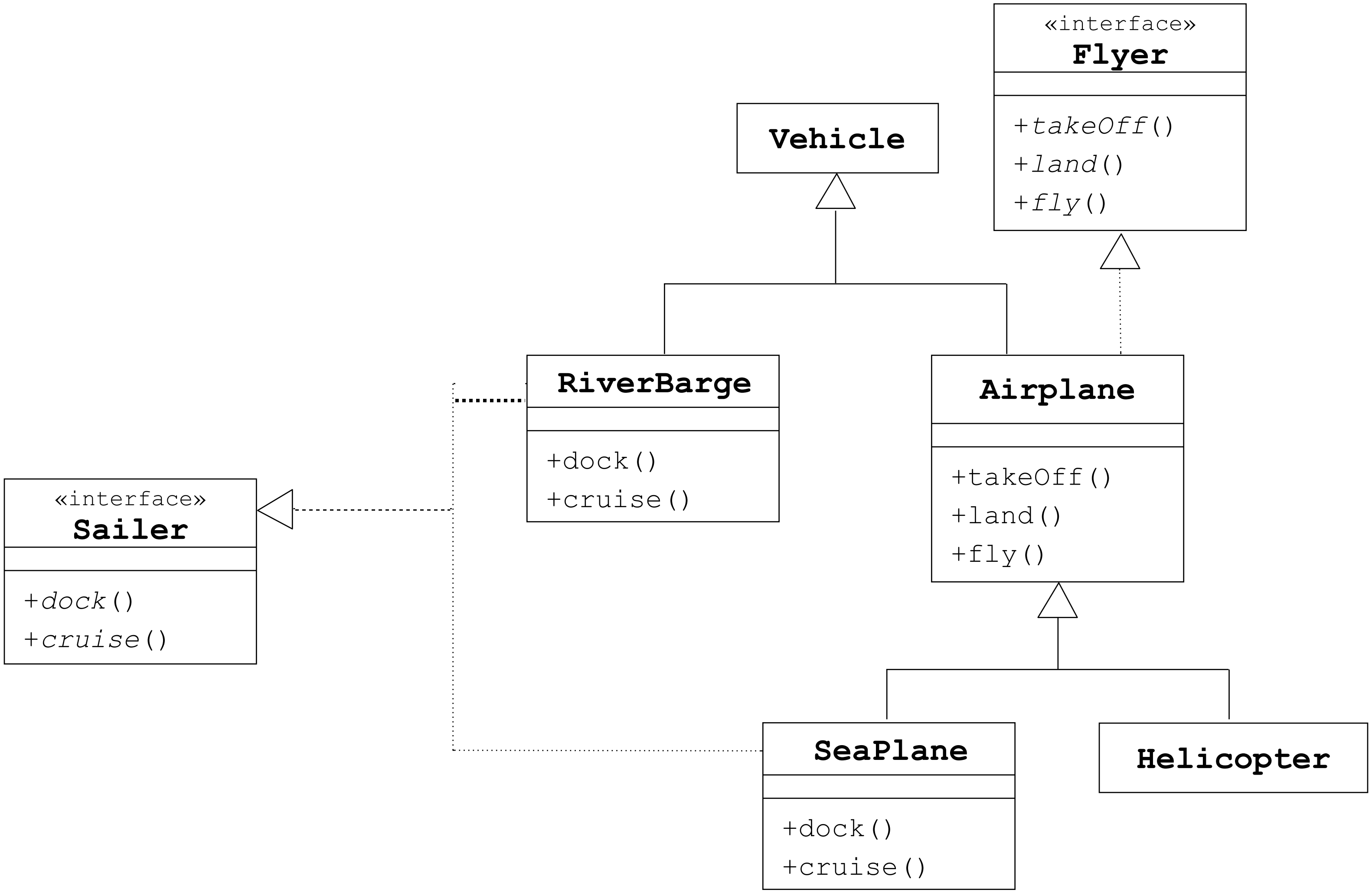
# The Flyer Example

```
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

```
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

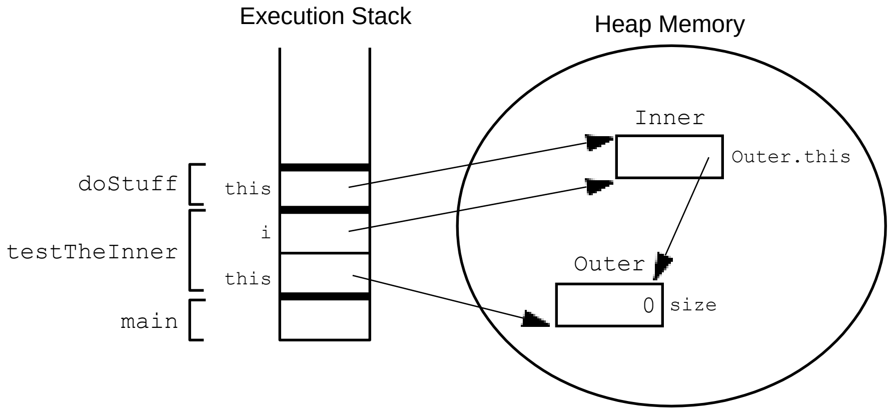
# Inner Classes

# Inner Classes

- Added to JDK 1.1
- Allow a class definition to be placed inside another class definition
- Group classes that logically belong together
- Have access to their enclosing class’s scope

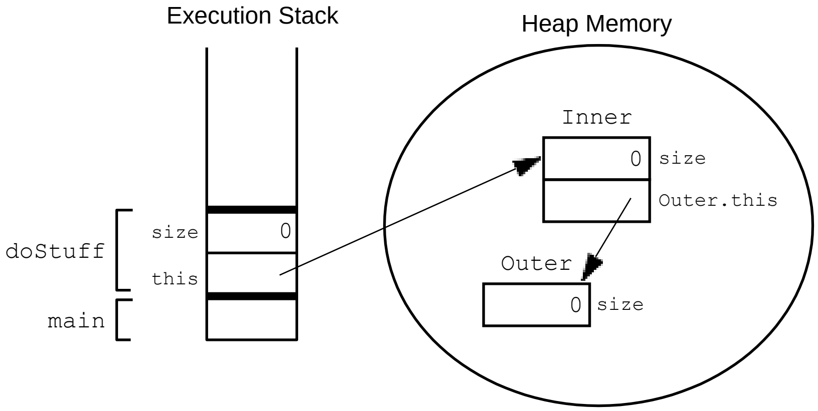
# Inner Class Example

```
public class Outer1
  { private int size;

  /* Declare an inner class called "Inner" */
  public class Inner {
    public void doStuff() {
      // The inner class has access to ’size’ from Outer
      size++;
    }
  }

  public void testTheInner() {
    Inner i = new Inner();
    i.doStuff();
  }
}
```



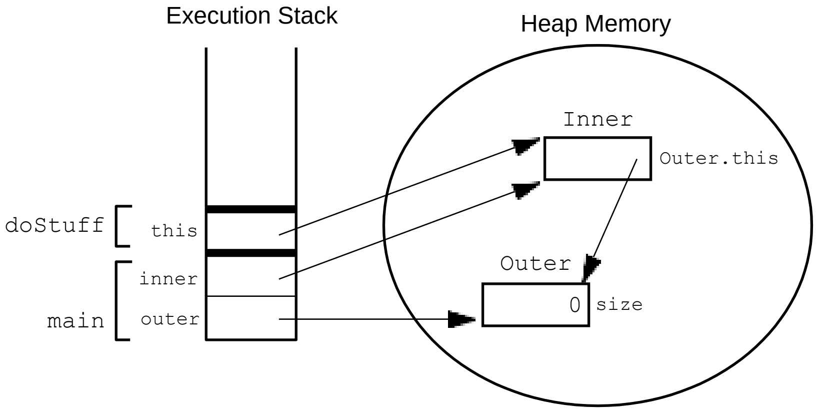
# Inner Class Example

```
public class Outer3
  { private int size;

  public class Inner
    { private int

    public void doStuff(int size) {
      size++;             // the local parameter
      this.size++;        // the Inner object attribute
      Outer3.this.size++;  // the Outer3 object attribute
    }
  }
}
```



# Inner Class Example

```
public class Outer2 {
  private int size;

  public class Inner {
    public void doStuff() {
      size++;
    }
  }
}
```



```
public class TestInner {
  public static void main(String[] args) {
    Outer2 outer = new Outer2();

    // Must create an Inner object relative to an Outer
    Outer2.Inner inner = outer.new Inner();
    inner.doStuff();
  }
}
```

# Properties of InnerClasses

- You can use the class name only within the defined scope, except when used in a qualified name. The name of the inner class must differ from the enclosing class

- The inner class can be defined inside a method. Only local variables marked as `final` can be accessed by methods within an inner class.

# Inner Class Example

```java
public class Outer4
  { private int size =

  public Object makeTheInner(int localVar)
    { final int finalLocalVar = 6;

    // Declare a class within a method!?!
    class Inner {
      public String toString() {
        return ("#<Inner size=" + size +
                // " localVar=" + localVar + // ERROR: ILLEGAL
                "finalLocalVar=" + finalLocalVar + ">");
      }
    }

    return new Inner();
  }

  public static void main(String[] args)
    { Outer4 outer = new Outer4();
    Object obj = outer.makeTheInner(47);
    System.out.println("The object is " + obj);
  }
}
```

# Properties of InnerClasses

- The inner class can use both class and instance variables of enclosing classes and local variables of enclosing blocks
- The inner class can be defined as `abstract`
- The inner class can have any access mode
- The inner class can act as an interface implemented by another inner class

# Properties of InnerClasses

- Inner classes that are declared `static` automatically become top-level classes
- Inner classes cannot declare any `static` members; only top-level classes can declare `static` members
- An inner class wanting to use a `static` member must be declared `static`

# Java Native Interface

# Native Methods

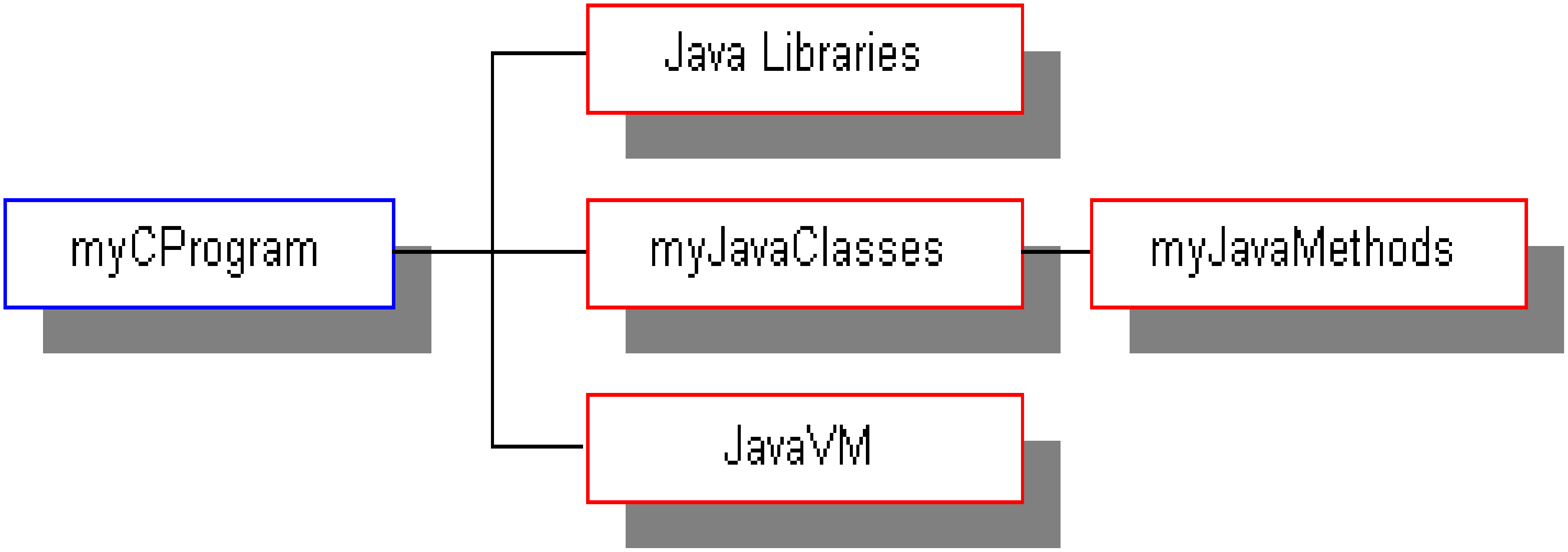
- **In a class , including a method that is implemented in some *other programming language*, such as C or C++, external to the Java Virtual Machine.**
- **To specify such a method within a class definition,you use the keyword native in the declaration of the method. For example:**

  *public native long getData();*

  - **The method will have *no body* in Java since it is defined elsewhere, where all the work is done, so the declaration ends with a semicolon.**
  - **The implementation of a native method will need to use an interface to the Java environment.**
  - **The standard API for implementing native methods in C, for example,is called JNI — the Java Native Interface.**

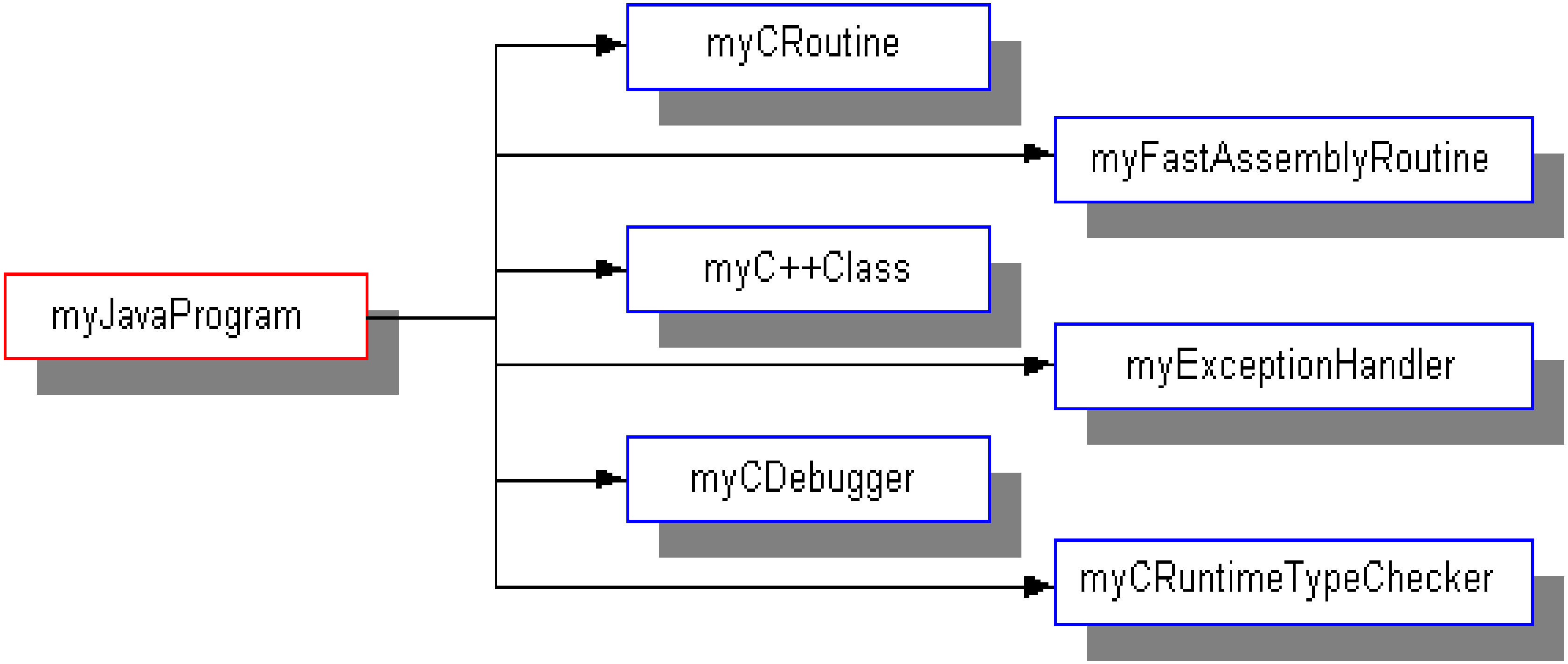
# JNI

- a legacy C program can use the JNI to link with Java libraries, call Java methods, use Java classes, and so on.

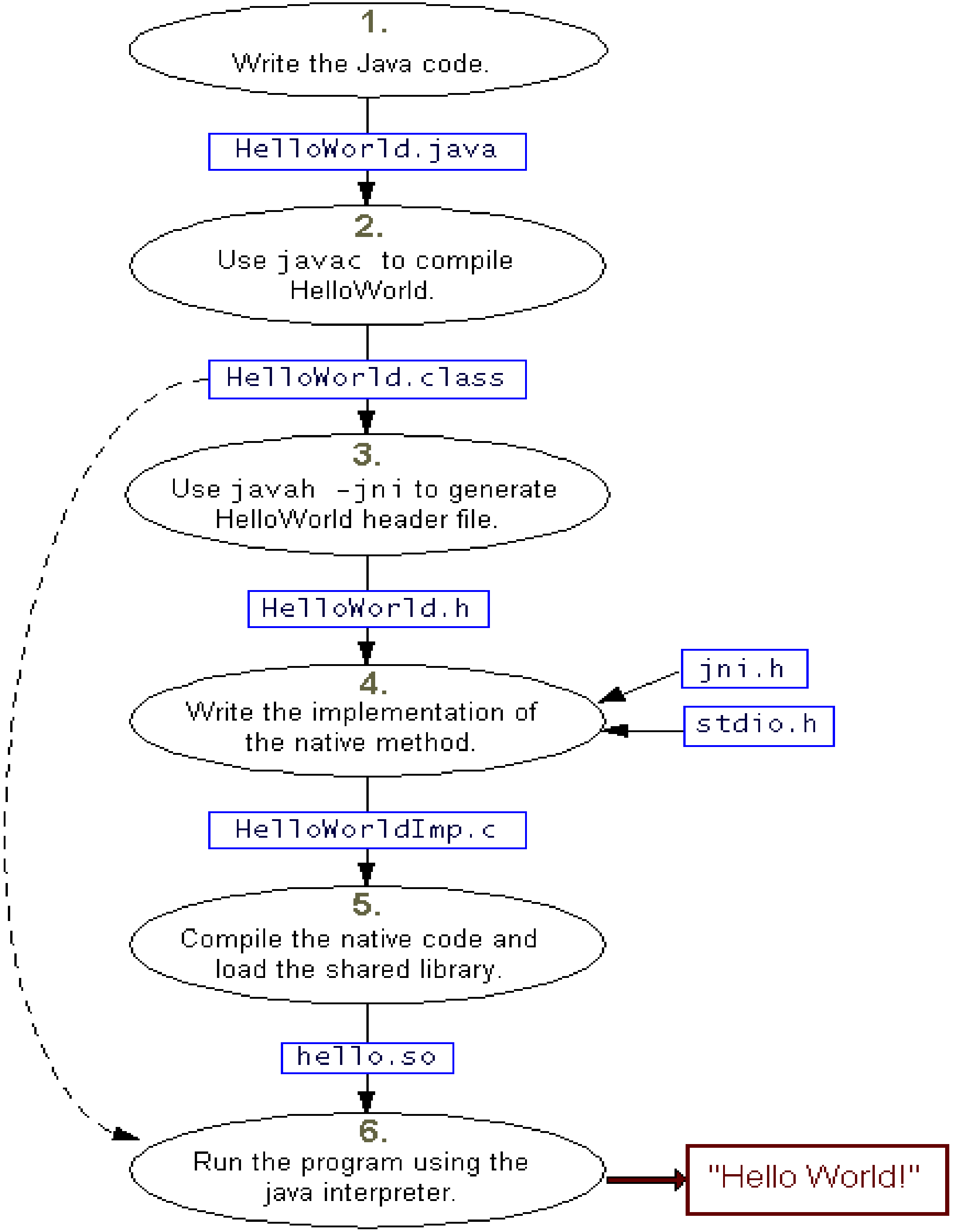


# JNI

- utilizing the JNI from a Java program, including calling C routines, using C++ classes, calling assembler routines, and so on.



# JNI


1.
Write the Java code.
HelloWorld.java
2.
Use javac to compile HelloWorld.
HelloWorld.class
3.
Use javah -jni to generate HelloWorld header file.
HelloWorld.h
jni.h
stdio.h
4.
Write the implementation of the native method.
HelloWorldImp.c
5.
Compile the native code and load the shared library.
hello.so
6.
Run the program using the java interpreter.
"Hello World!"

# Summary

- **static** variables, methods, initializers, and inner class
- **final** classes, methods, and variables
- **abstract** classes and methods
- **interface**
- Inner Class
- Singleton Design Pattern
- Template Method Design Pattern