Module 2

Object-Oriented Programming

Objectives

- Define modeling concepts: *abstraction*, *encapsulation*, and *packages*
- Discuss why you can reuse Java technology application code
- Define *class*, *member*, *attribute*, *method*, *constructor*
- Use the access modifiers `private` and `public` as appropriate for the guidelines of encapsulation
- Invoke a method on a particular object
- Use the Java technology application programming interface (API) online documentation
- *this*
- *package* and *import*

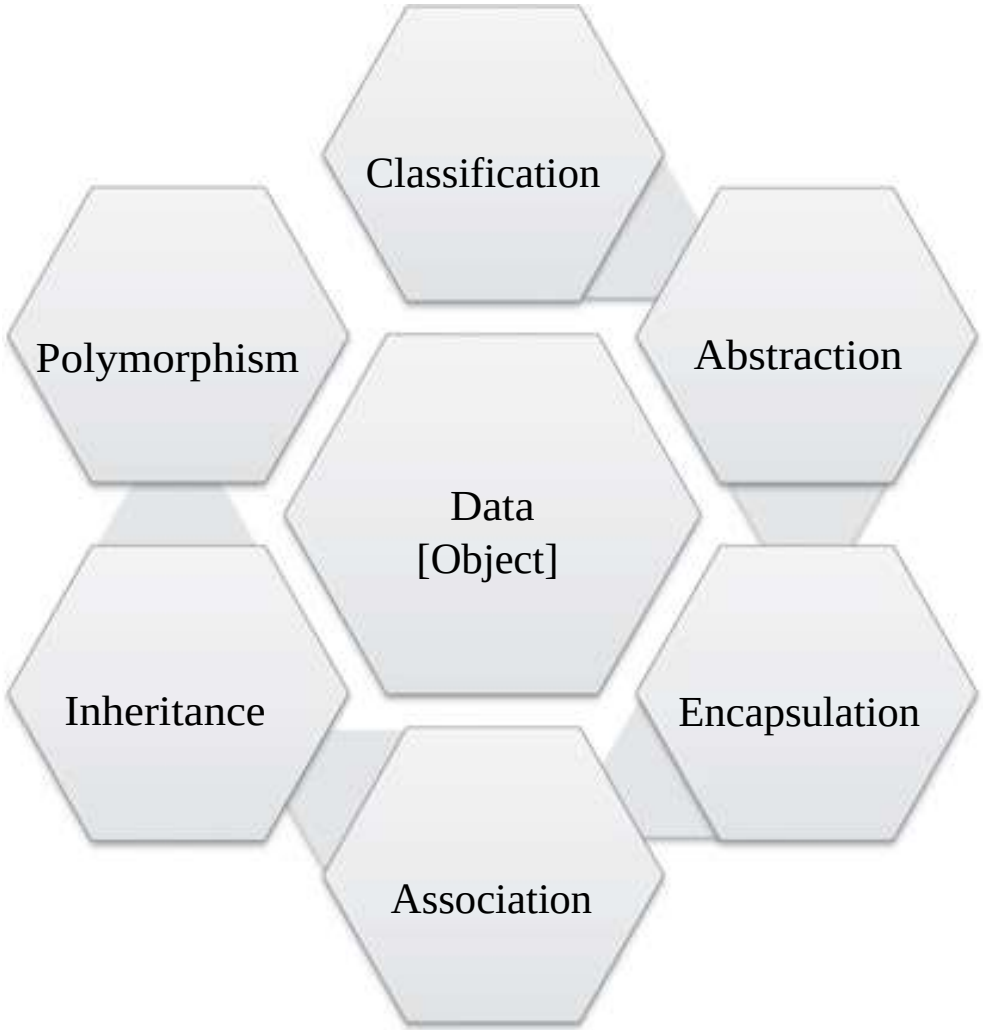
Analysis and Design

- Analysis describes *what* the system needs to do:

  Modeling the real-world, including actors and activities, objects, and behaviors
- Design describes *how* the system does it:
  - Modeling the relationships and interactions between objects and actors in the system
  - Finding useful abstractions to help simplify the problem or solution

Six fundamentals make up OO

- Classification (grouping)
- Abstraction (representing)
- Encapsulation (modularizing)
- Association (relating)
- Inheritance (generalizing)
- Polymorphism (executing)



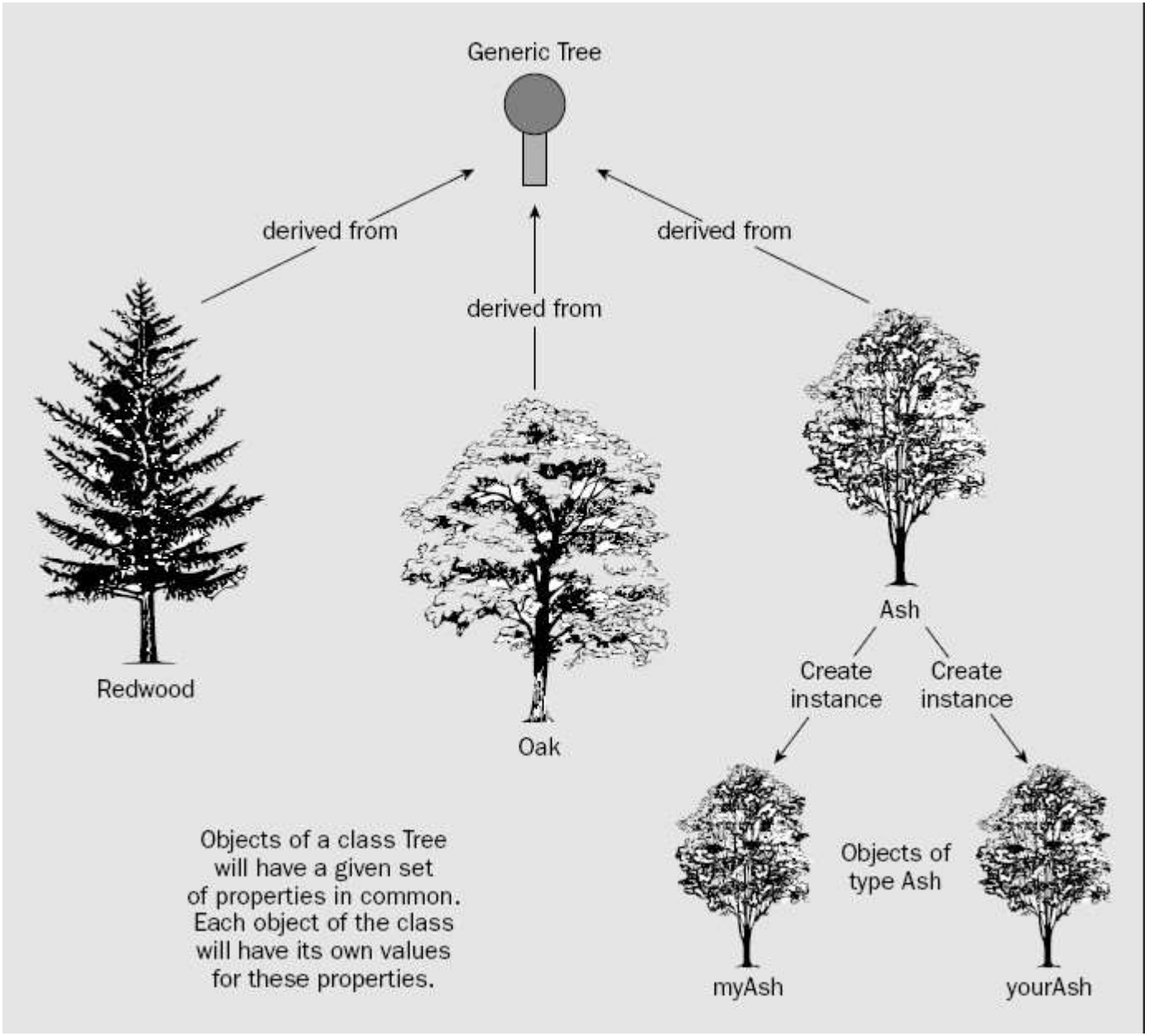
Abstraction

Abstraction

- Functions – Write an algorithm once to be used in many situations
- Objects – Group a related set of attributes and behaviors into a class
- Frameworks and APIs – Large groups of objects that support a complex activity; Frameworks can be used *as is* or be modified to extend the basic behavior

Classes as Blueprints for Objects

- In manufacturing, a blueprint describes a device from which many physical devices are constructed.
- In software, a class is a description of an object:
  - A class describes the data that each object includes.
  - A class describes the behaviors that each object exhibits.
- In Java technology, classes support three key features of object-oriented programming (OOP):
  - Encapsulation
  - Inheritance
  - Polymorphism



The ones below are real objects with multiple instances. Each object has a unique identifier.

Tese names with boxes around them are *ABSTRACTIONS*. They form the basis for classes.

Frog Object-1, Frog Object-2, and so on...

Hat Object-1, Hat Object-2, and so on...

Clock Object-1, Clock Object-2, and so on...

Cat Object-1, Cat Object-2, and so on...

Abstracted to → FROG

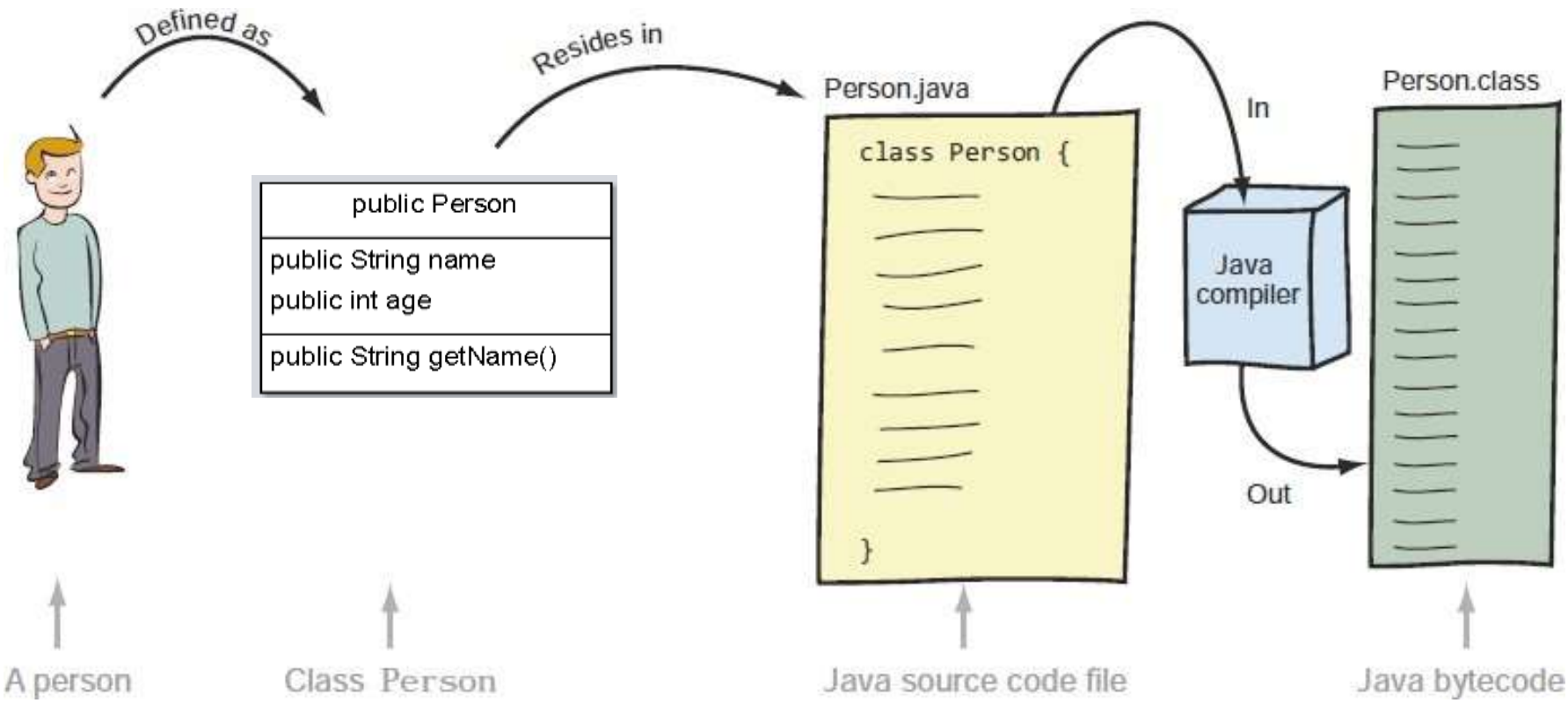
Abstracted to → HAT

Abstracted to → CLOCK

Abstracted to → CAT

Contains common characteristics of frog (which become attributes and behavior)

Good classification leads to creation of good abstractions.



Declaring Java Technology Classes

- Basic syntax of a Java class:

```
<modifier>* class <class_name> {
  <attribute_declaration>*
  <constructor_declaration>*
  <method_declaration>*
}
```

- Example:

```
1   public class Vehicle
2     { private double
3     public void setMaxLoad(double value) {
4       maxLoad = value;
5     }
6   }
```

Declaring Attributes

- Basic syntax of an attribute:

  <modifier>* <type> <name> [**=** ***<initial_value>]*** **;**

- Examples:

```
1   public class Foo {
2     private int x;
3     private float y = 10000.0F;
4     private String name = "Bates Motel";
5   }
```

Declaring Methods

- Basic syntax of a method:

```
<modifier>* <return_type> <name> ( <argument>* ) {
  <statement>*
}
```

- Examples:

```
1   public class Dog
2     { private int weight;
3     public int getWeight() {
4       return weight;
5     }
6     public void setWeight(int newWeight) {
7       if ( newWeight > 0 ) {
8         weight = newWeight;
9       }
10    }
11  }
```

Declaring Constructors

- Basic syntax of a constructor:

```
[<modifier>] <class_name> ( <argument>* ) {
  <statement>*
}
```

- Example:

```
1   public class Dog {
2
3     private int weight;
4
5     public Dog()
6        { weight =
7        42;
8   }
```

The Default Constructor

- There is always at least one constructor in every class.
- If the writer does not supply any constructors, the default constructor is present automatically:
  - The default constructor takes no arguments
  - The default constructor body is empty
- The default enables you to create object instances with `new` *`Xxx`*`()` without having to write a constructor.

Accessing Object Members

- The *dot* notation is: ***<object>.<member>***
- This is used to access object members, including attributes and methods.
- Examples of dot notation are:

```
d.setWeight(42);
d.weight = 42;  // only permissible if weight is public
```

Declaring Object Reference Variables

- Syntax:

  Classname identifier;

- Example:

```
Shirt myShirt;
```

Instantiating an Object

Syntax:

```
new Classname()
```

Initializing Object Reference Variables

- The assignment operator
- Example:

```
myShirt = new Shirt();
```

Declaring Object References, Instantiating Objects, and Initializing Object References

Example:

```
1   class ShirtTest {

2

3   public static void main (String args[]) {
4
5       Shirt myShirt = new Shirt();
6
7       myShirt.displayInformation();
8
9     }
10  }
```

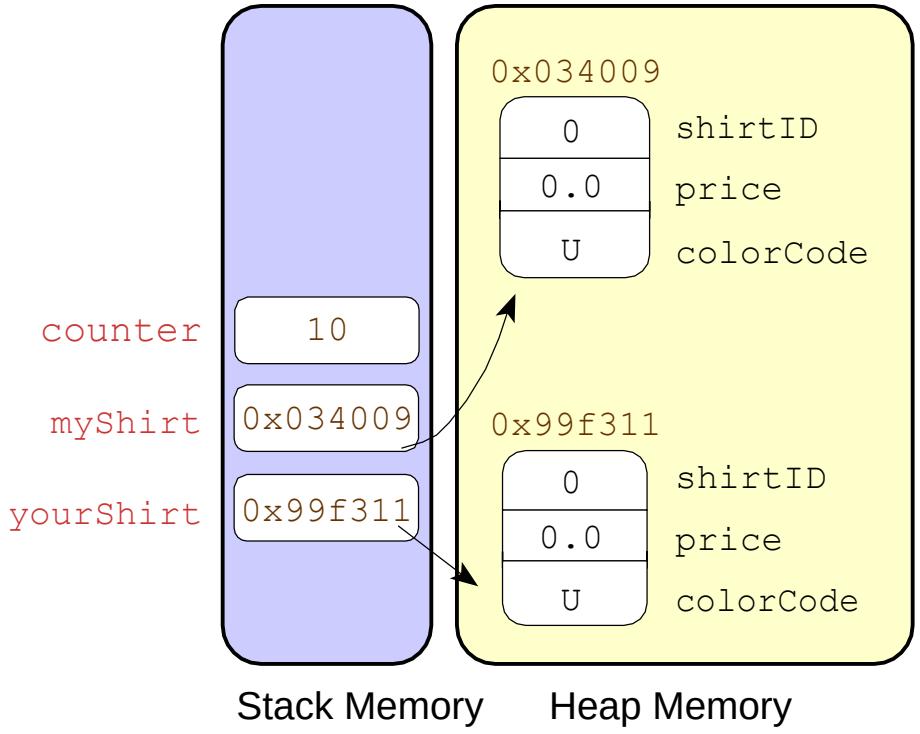
Using an Object Reference Variable to Manipulate Data

```
1   public class ShirtTestTwo {
2
3     public static void main (String args[]) {
4
5       Shirt myShirt = new Shirt();
6       Shirt yourShirt = new Shirt();
7
        myShirt.displayInformation();
9       yourShirt.displayInformation();
10
11      myShirt.colorCode='R';
12      yourShirt.colorCode='G';
13
14      myShirt.displayInformation();
15      yourShirt.displayInformation();
16
17    }
18  }
```

Storing Object Reference Variables in Memory

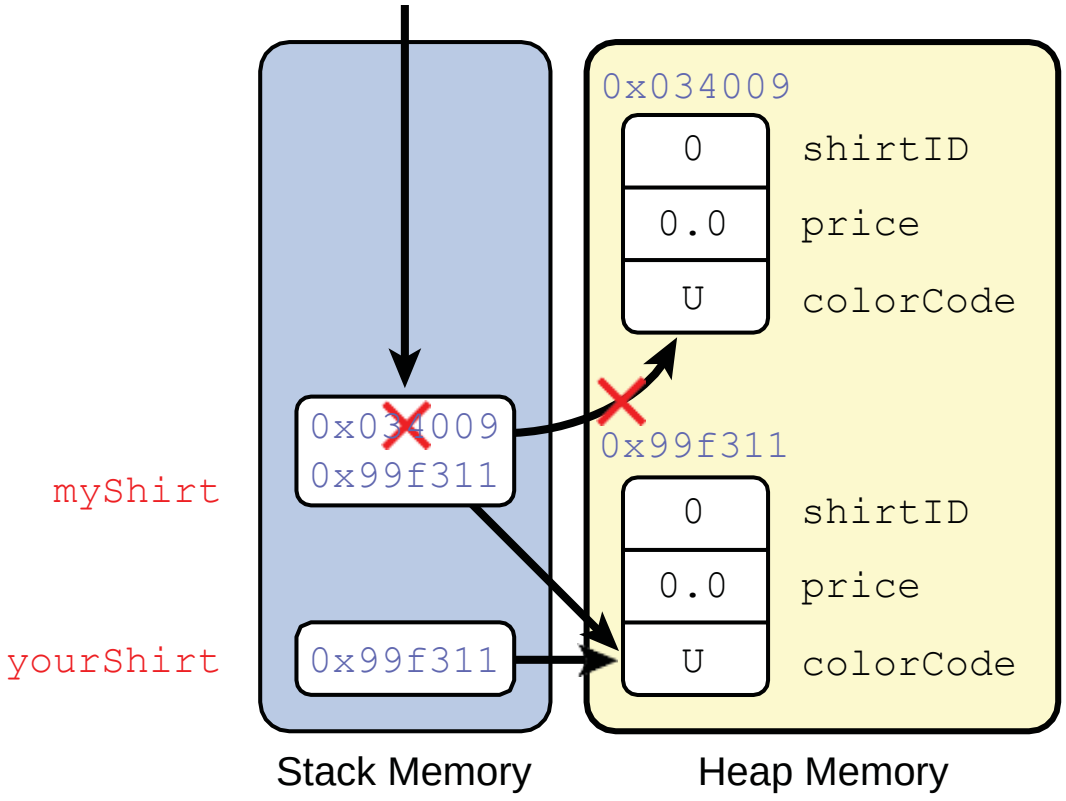
```
public static void main (String args[]) {

  int counter;
  counter = 10;
  Shirt myShirt = new Shirt ( );
}
```



Assigning an Object Reference From One Variable to Another

```
1   Shirt  myShirt = new Shirt( );
2   Shirt  yourShirt = new Shirt( );
3   myShirt = yourShirt;
```



API Documentation

Using the Java Technology API Documentation

- A set of Hypertext Markup Language (HTML) files provides information about the API.
- A frame describes a package and contains hyperlinks to information describing each class in that package.
- A class document includes the class hierarchy, a description of the class, a list of member variables, a list of constructors, and so on.

Investigating the Java Class Libraries

OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP Java SE 14 & JDK 14

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

SEARCH: Search

Module java.base
Package java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>, Constable, ConstantDesc

---

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence, Constable, ConstantDesc
```

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Using the `String` Class

- Creating a `String` object with the `new` keyword:

```
String myName = new String("Fred Smith");
```

- Creating a `String` object without the `new` keyword:

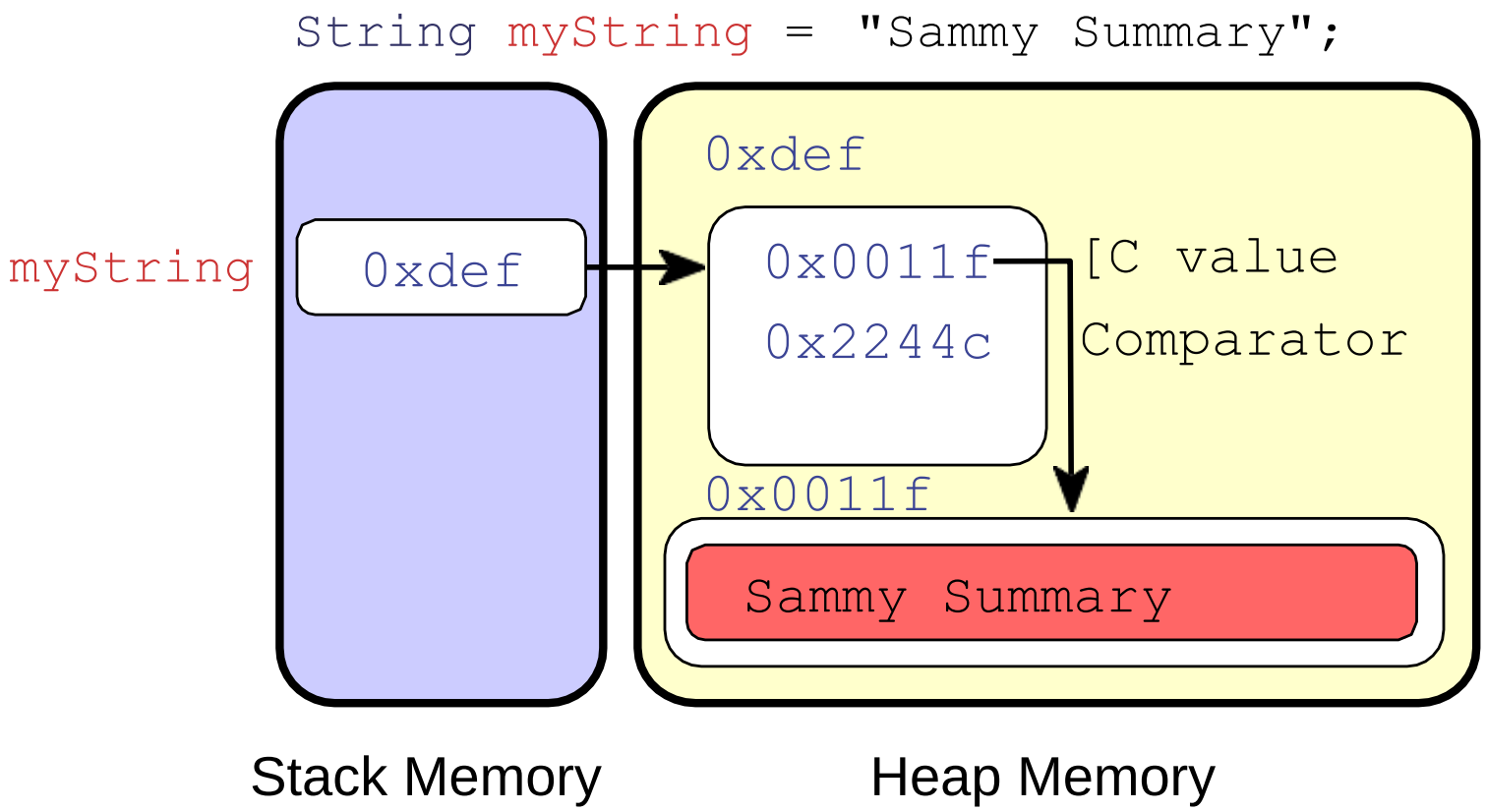
```
String myName = "Fred Smith";
```

- Methods of String class:

```
String myName = "Fred Smith"
char c = myName.charAt(0); //'F'
String lastName = myName.substring(5);//"Smith"
String s = myName.substring(6,8);//"mit"
lastName.equals(s);//false
s = s.toUpperCase();//"MIT"
```

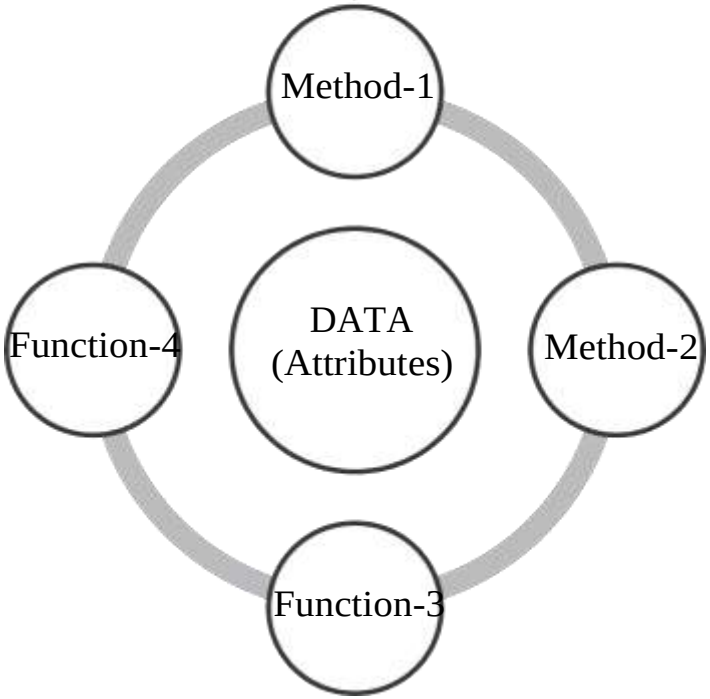
- Reverse: StringBuffer or StringBuilder class

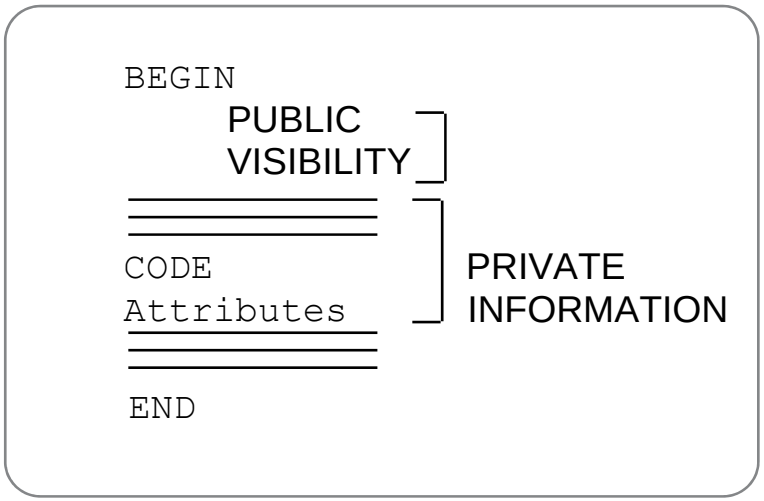
Storing `String` Objects in Memory



Encapsulation

Encapsulation means DATA are "wrapped" with METHODS in a meaningful way; the data's public visibility provides the *only* way to access them





Information Hiding

The problem:

MyDate
+day : int +month : int +year : int

Client code has direct access to internal data (`d` refers to a `MyDate` object):

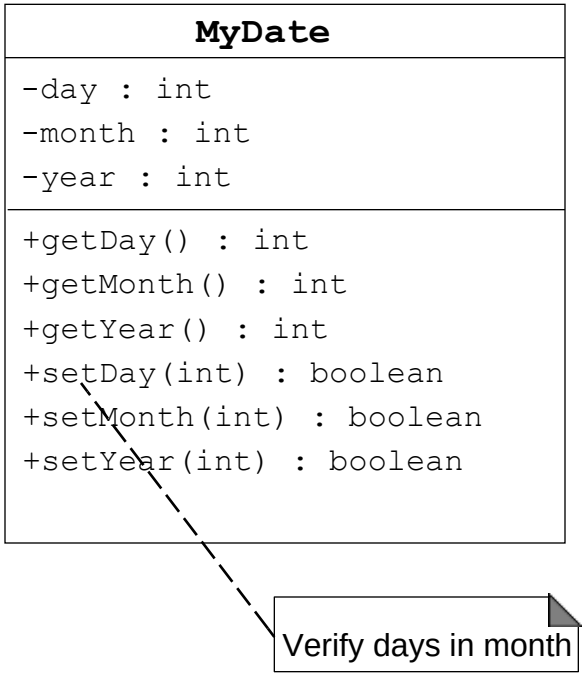
```
d.day = 32;
// invalid day

d.month = 2; d.day = 30;
// plausible but wrong

d.day = d.day + 1;
// no check for wrap around
```

Information Hiding

The solution:



Client code must use setters and getters to access internal data:

```
MyDate d = new MyDate();
d.setDay(32);
// invalid day, returns false

d.setMonth(2);
d.setDay(30);
// plausible but wrong,
// setDay returns false

d.setDay(d.getDay() + 1);
// this will return false if wrap around
// needs to occur
```

Encapsulation

- Hides the implementation details of a class
- Forces the user to use an interface to access data
- Makes the code more maintainable

MyDate
-date : long
+getDay() : int +getMonth() : int +getYear() : int +setDay(int) : boolean +setMonth(int) : boolean +setYear(int) : boolean -isDayValid(int) : boolean

The *this* Reference

The `this` Reference

Here are a few uses of the `this` keyword:

- To resolve ambiguity between instance variables and parameters
- To access the current object in methods or constructors

The `this` Reference

```
1   public class MyDate {
2     private int day = 1;
3     private int month = 1;
4     private int year = 2000;
5
6     public MyDate(int day, int month, int year) {
7       this.day   = day;
8       this.month = month;
9       this.year  = year;
10    }
11    public MyDate(MyDate date) {
12      this.day   = date.day;
13      this.month = date.month;
14      this.year  = date.year;
15    }
```

The `this` Reference

```
16
17    public MyDate addDays(int moreDays)
18      { MyDate newDate = new MyDate(this);
19      newDate.day = newDate.day + moreDays;
20      // Not Yet Implemented: wrap around code...
21      return newDate;
22    }
23    public String toString() {
24      return "" + day + "-" + month + "-" + year;
25    }
26  }
```

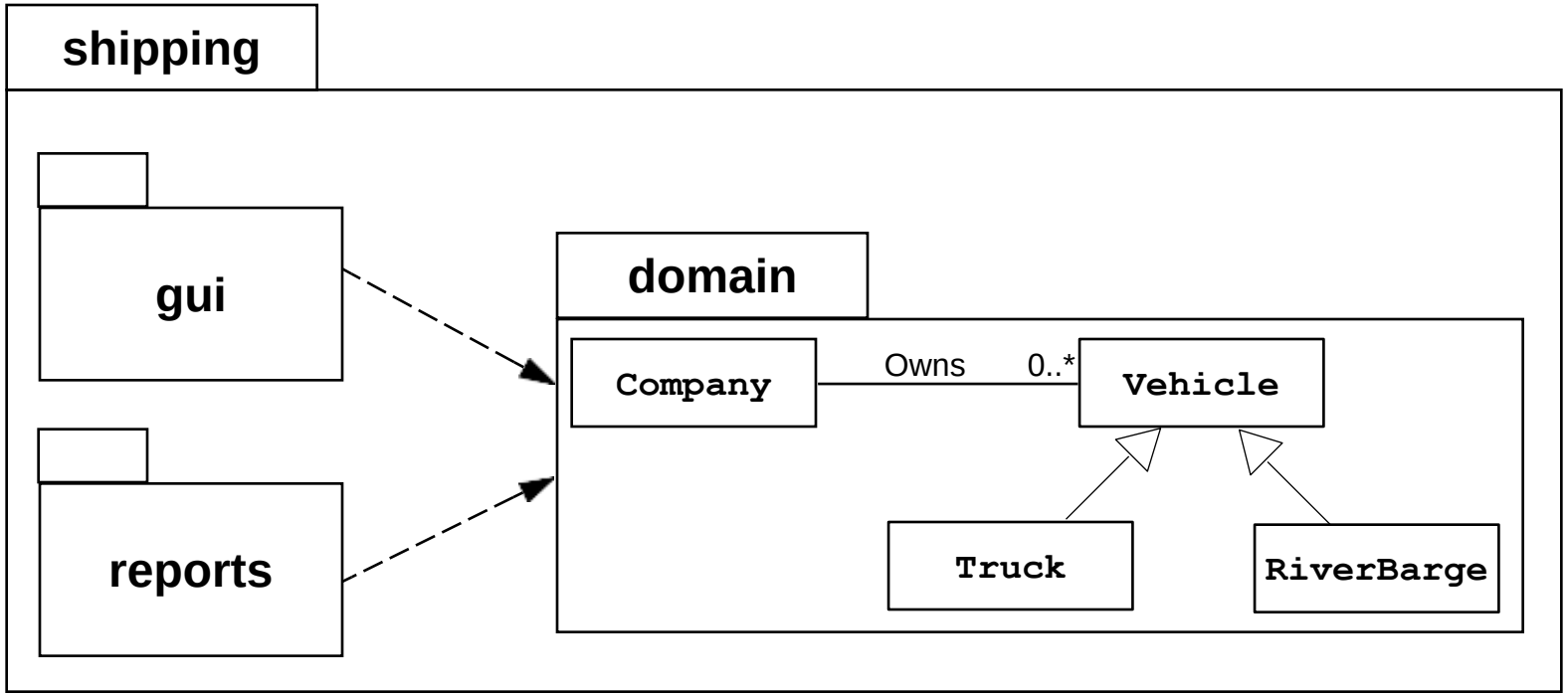
The `this` Reference

```
1   public class TestMyDate {
2     public static void main(String[] args)
3       { MyDate my_birth = new MyDate(22, 7,
4       1964); MyDate the_next_week =
5
6       System.out.println(the_next_week);
7     }
8   }
```

Packages

Software Packages

- Packages help manage large software systems.
- Packages can contain classes and sub-packages.

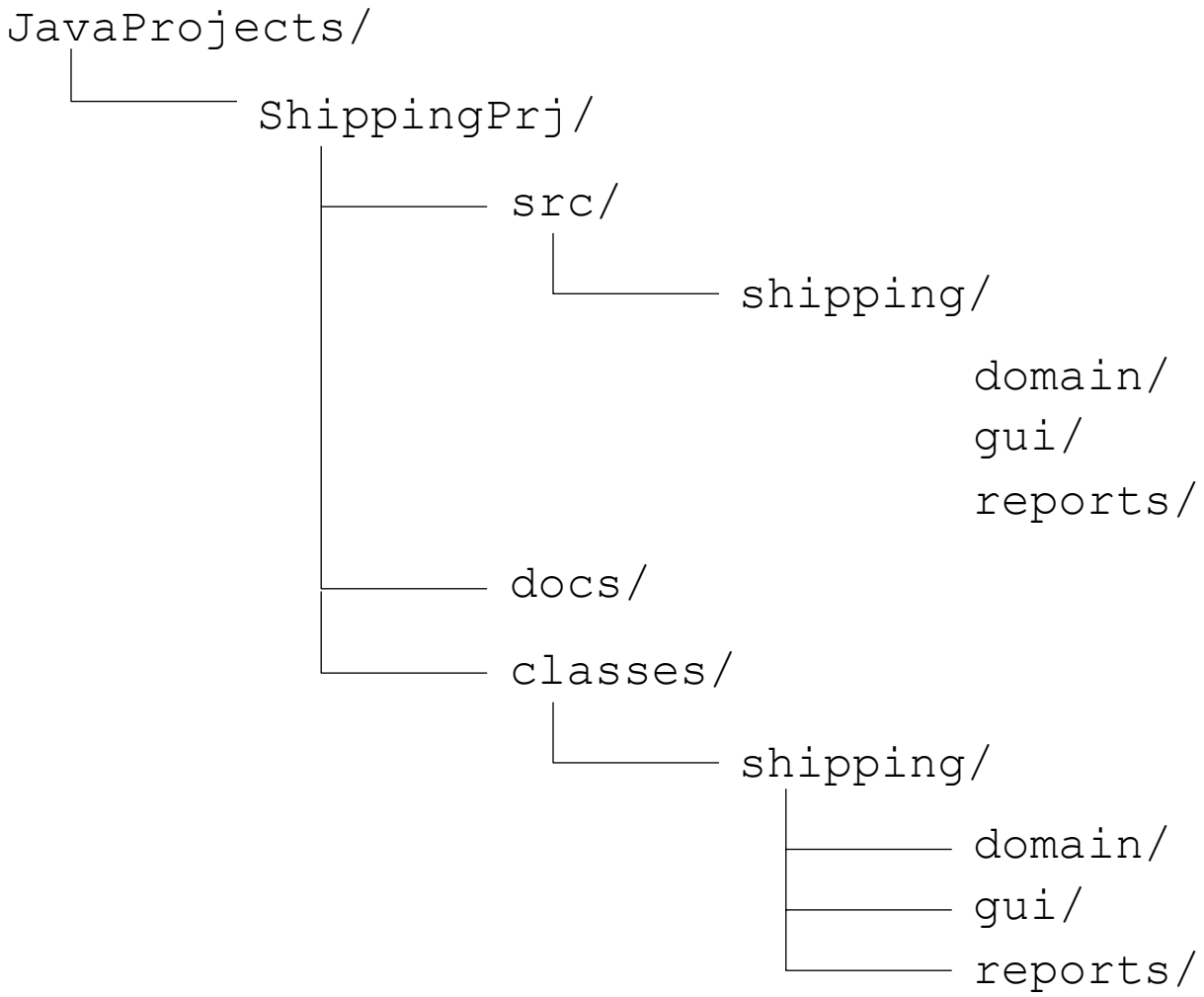


Directory Layout and Packages

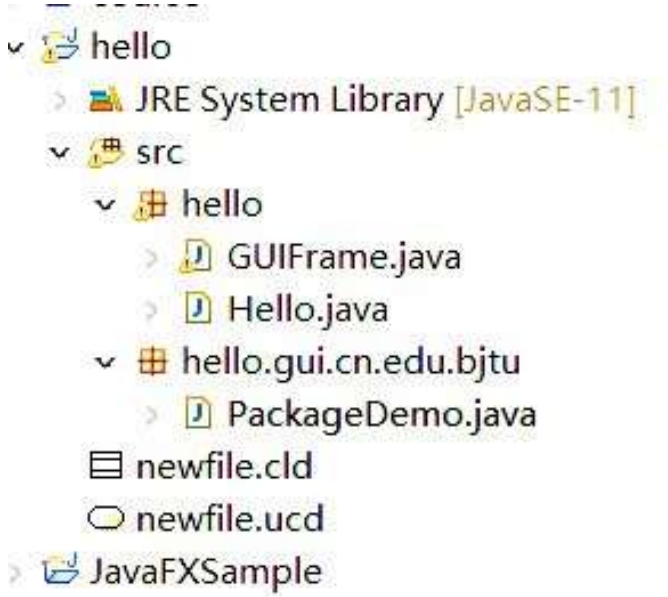
- Packages are stored in the directory tree containing the package name.
- An example is the shipping application packages.

```
shipping/
        domain/
           ├──── Company.class
           ├──── Vehicle.class
           ├──── RiverBarge.class
           └──── Truck.class
        gui/
        reports/
           └──── VehicleCapacityReport.class
```

Development



Package



The `package` Statement

- Basic syntax of the `package` statement is:

  `package` ***`<top_pkg_name>[.<sub_pkg_name>]*`***`;`

- Examples of the statement are:

```
package shipping.gui.reportscreens;
```

- Specify the package declaration at the beginning of the source file.
- Only one package declaration per source file.
- If no package is declared, then the class is placed into the default package.
- Package names must be hierarchical and separated by dots.

The `import` Statement

- Basic syntax of the `import` statement is:

```
import <pkg_name>[.<sub_pkg_name>]*.<class_name>;
OR
import <pkg_name>[.<sub_pkg_name>]*.*;
```

- Examples of the statement are:

```
import java.util.List;
import java.io.*;
import shipping.gui.reportscreens.*;
```

- The import statement does the following:
  - Precedes all class declarations
  - Tells the compiler where to find classes

Source File Layout

- Basic syntax of a Java source file is:

[<package_declaration>]
<import_declaration>*
<class_declaration>+

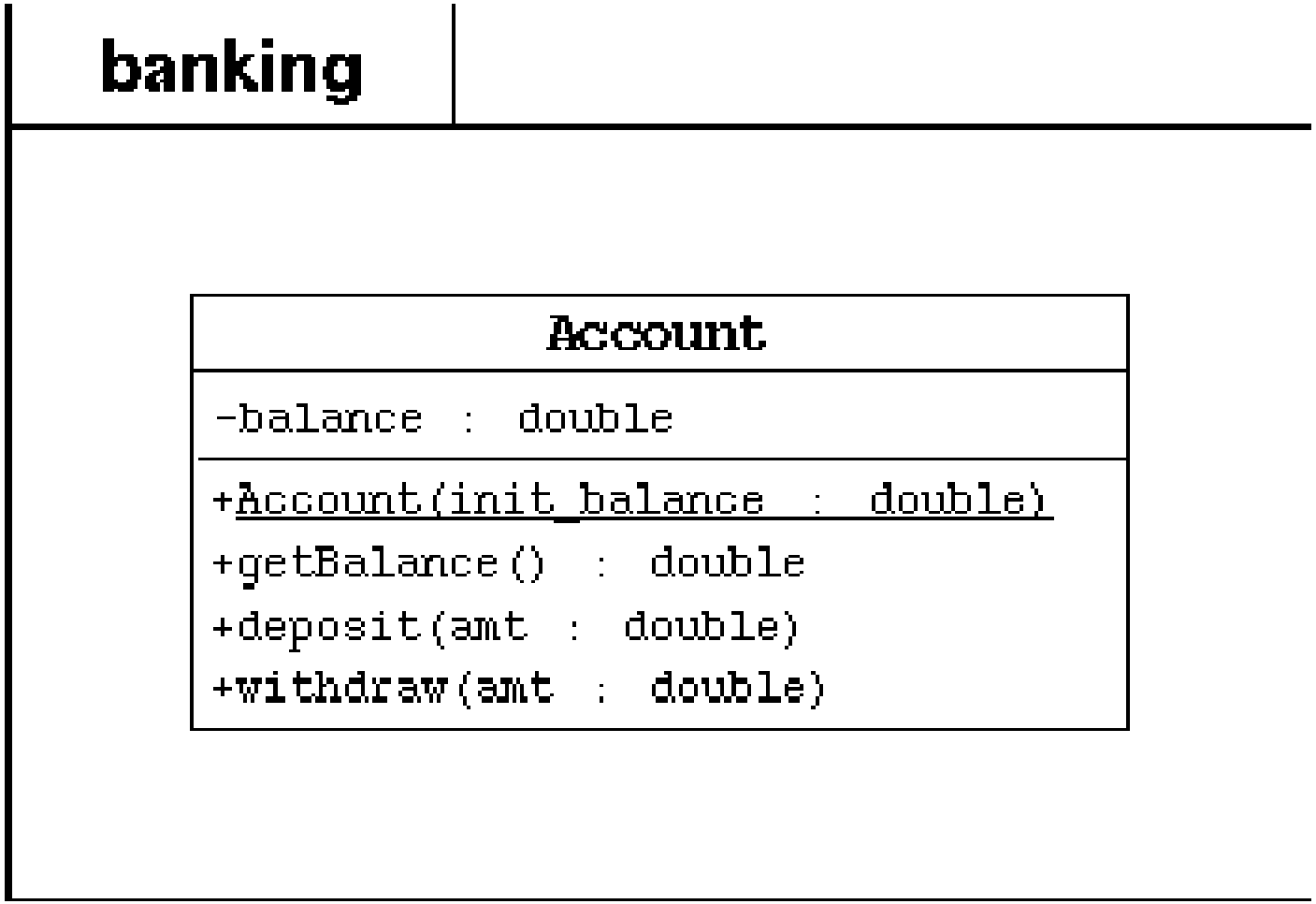
- For example, the `VehicleCapacityReport.java` file is:

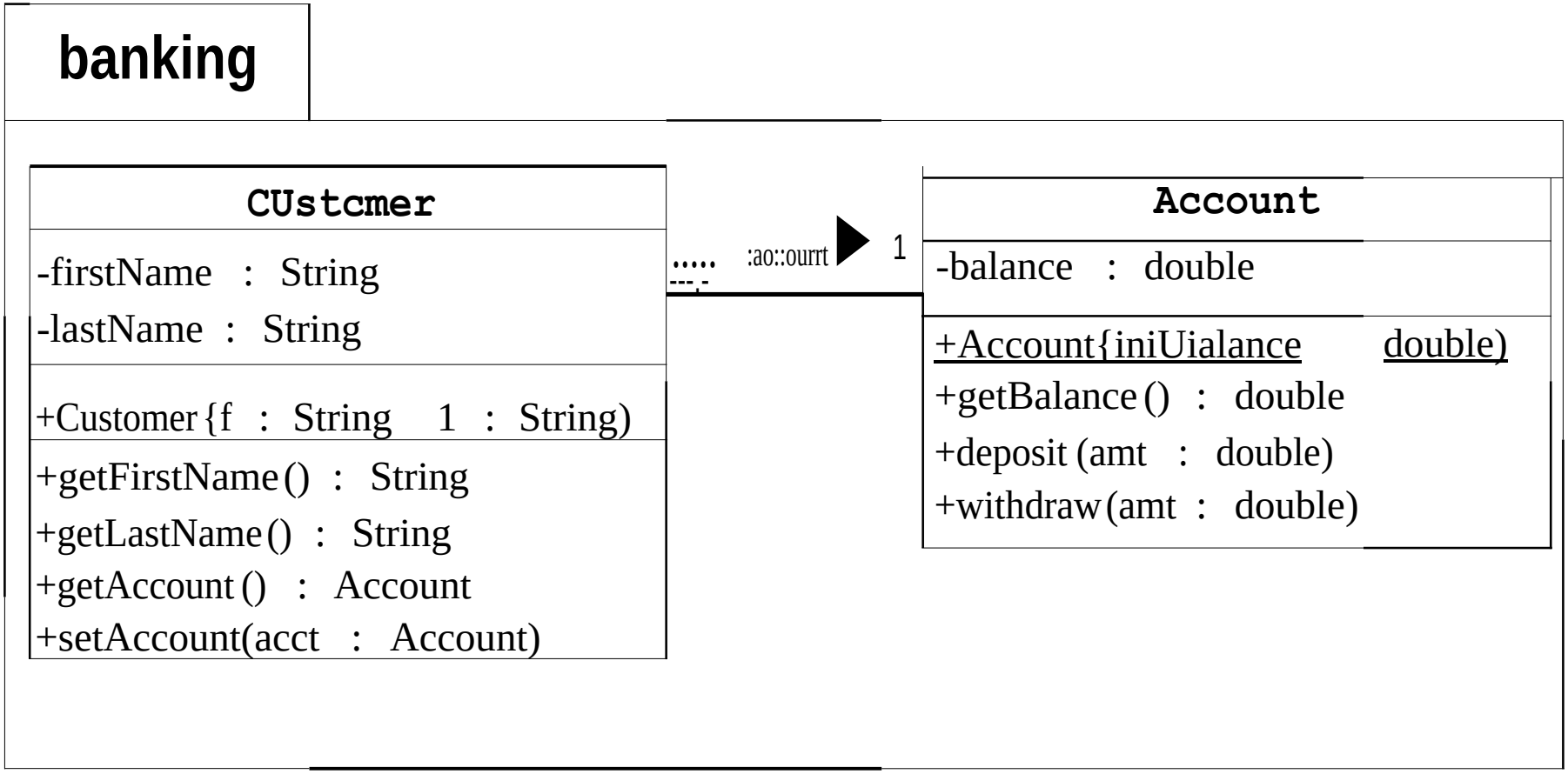
```
1   package shipping.reports;
2
3   import shipping.domain.*;
4   import java.util.List;
5   import java.io.*;
6
7   public class VehicleCapacityReport {
8     private List vehicles;
9     public void generateReport(Writer output) {...}
10  }
```

Compiling Using the `-d` Option

```
cd JavaProjects/ShippingPrj/src
javac -d ../classes shipping/domain/*.java
```

Package

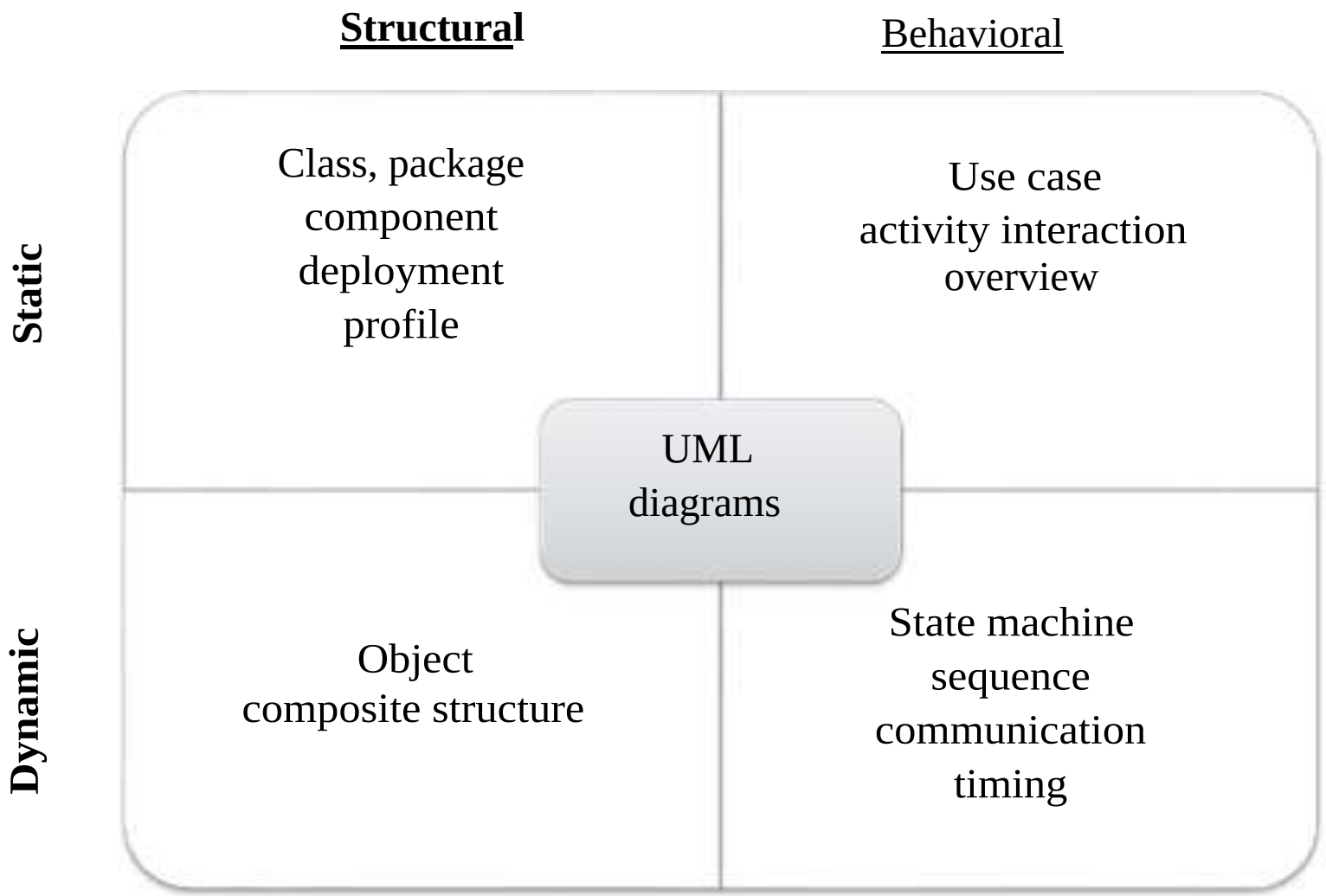

banking
Account
-balance : double
+Account(init_balance : double)
+getBalance() : double
+deposit(amt : double)
+withdraw(amt : double)


banking
CUstcmer
-firstName : String
-lastName : String
+Customer{f : String 1 : String)
+getFirstName() : String
+getLastName() : String
+getAccount() : Account
+setAccount(acct : Account)
:ao::ourrt
1
Account
-balance : double
+Account{iniUialance double)
+getBalance() : double
+deposit (amt : double)
+withdraw(amt : double)

Terminology Recap

- Class – The source-code blueprint for a run-time object
- Object – An instance of a class;
  also known as *instance*
- Attribute – A data element of an object;
  also known as *data member*, *instance variable*, and *data field*
- Method – A behavioral element of an object;
  also known as *algorithm*, *function*, and *procedure*
- Constructor – A *method-like* construct used to initialize a new object
- Package – A grouping of classes and sub-packages

UML diagrams



Summary

- Abstraction
- Java Classes and Objects
- Object Reference Variables
- **String** and **StringBuilder** Class
- Encapsulation: **public** and **private**
- The **this** Reference
- Java API Documentation
- Packages and **import**