Programming in Java
Object-Oriented Programming

Hua Huang, Ph.D.

Spring 2019

Objectives

- Define modeling concepts: **abstraction**, **encapsulation**, and **packages**
- Discuss why you can reuse Java technology application code
- Define **class**, **member**, **attribute**, **method**, **constructor**, and **package**
- Use the **access modifiers** **private** and **public** as appropriate for the guidelines of **information hiding and encapsulation**
- **Invoke a method** on a particular object
- Java **Coding Conventions**

Relevance

- What is your understanding of **software analysis** and **design**?
- What is your understanding of **design and code reuse**?
- What features does the Java programming language possess that make it an object-oriented language?
- Define the term **object-oriented**.

The Analysis and Design Phase

- **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

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**(封装)
  - **Inheritance**(继承)
  - **Polymorphism**(多态)

Declaring Java Technology Classes

Basic syntax of a Java class:

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

Example:

```
public class Vehicle {
  private double maxLoad;
  public void setMaxLoad(double value){
    maxLoad = value;
  }
}
```

Associating a New Class With a Package

```
package finance; // package statement

class Stock {
// Internals of the class declarations not shown
}
```

- The **package** statement provides a **namespace**:
  - **First non-comment line**
  - **Can be omitted** resulting in **default package**
  - Package name used in forming **fully qualified name** of class, for example `finance.Stock`

- To be detailed later...

Declaring the Foreign Classes Used by the New Class

```
package finance;

import java.util.Date;
// Additional import statements if required go here

class Stock {
// Implementation of the stock class
}
```

- Where declared?
- When required?

```
import java.util.*;
```

- The `java.lang` package

Declaring Attributes

- Basic syntax of an attribute:

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

  - Syntax: ***data_type identifier;***

    ```
    double price;
    ```

  - Syntax: ***data_type identifier = initial_value;***

    ```
    double price = 25.50;
    ```

  - Syntax: ***data_type identifier1, identifier2, identifier3;***

    ```
    Date birthday, anniversary;
    ```

- Fields represent attributes (or state)

Examples:

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

Declaring Methods

- Basic syntax of a method:

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

- Methods provide behavior.

Examples:

```
public class Dog {
  private int weight;
  public int getWeight() {
    return weight;
  }
  public void setWeight(int newWeight) {
    if ( newWeight > 0 ) {
      weight = newWeight;
    }
  }
}
```

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

Declaring Constructors

- Basic syntax of a constructor:

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

- Constructors provide dynamic initialization of fields.

Example:

```
public class Dog {

  private int weight;

  public Dog() {
    weight = 42;
  }
}
```

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.

The Default Constructor

```
package finance;

import java.util.Date;

class Stock {
  // Fields declarations
  String symbol;
  double price;
  Date date;

  // No constructors declared

  // Method declarations

}
```

Explicit No-arg Constructor

```
package finance;

import java.util.Date;

class Stock {
  // Field declarations
  String symbol;
  double price;
  Date date;

  // Constructor declarations
  Stock() {
    date = new Date();
  }

  // Method declarations not shown for clarity reasons

}
```

Overloading Constructors

- As with methods, constructors can be overloaded. An example is:

```
public  Employee(String name, double salary, Date DoB)
public  Employee(String name, double salary)
public  Employee(String name, Date DoB)
```

- **Argument lists must differ**.
- You can use the `this` reference at **the first line** of a constructor to call another constructor.

Overloading Constructors

```
public class Employee {
  private static final double BASE_SALARY = 15000.00;
  private String name;
  private double salary;
  private Date    birthDate;

  public Employee(String name, double salary, Date DoB) {
    this.name = name;
    this.salary = salary;
    this.birthDate = DoB;
  }
  public Employee(String name, double salary) {
    this(name, salary, null);
  }
  public Employee(String name, Date DoB) {
    this(name, BASE_SALARY, DoB);
  }
  // more Employee code...
}
```

Source File Layout

- Basic syntax of a Java source file is:

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

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

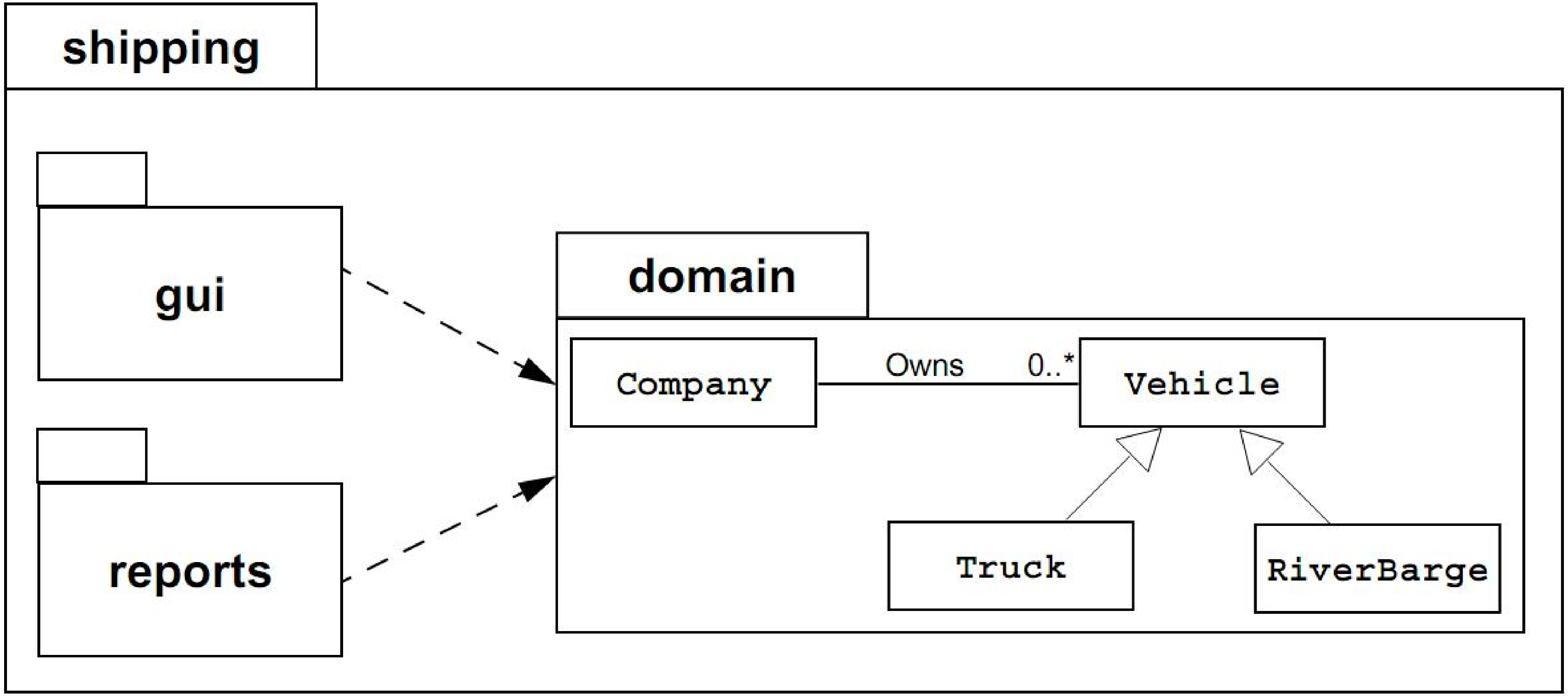
```
package shipping.reports;

import shipping.domain.*;
import java.util.List;
import java.io.*;

public class VehicleCapacityReport {
  private List vehicles;
  public void generateReport(Writer output) {...}
}
```

Software Packages

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



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>]*.*;
```

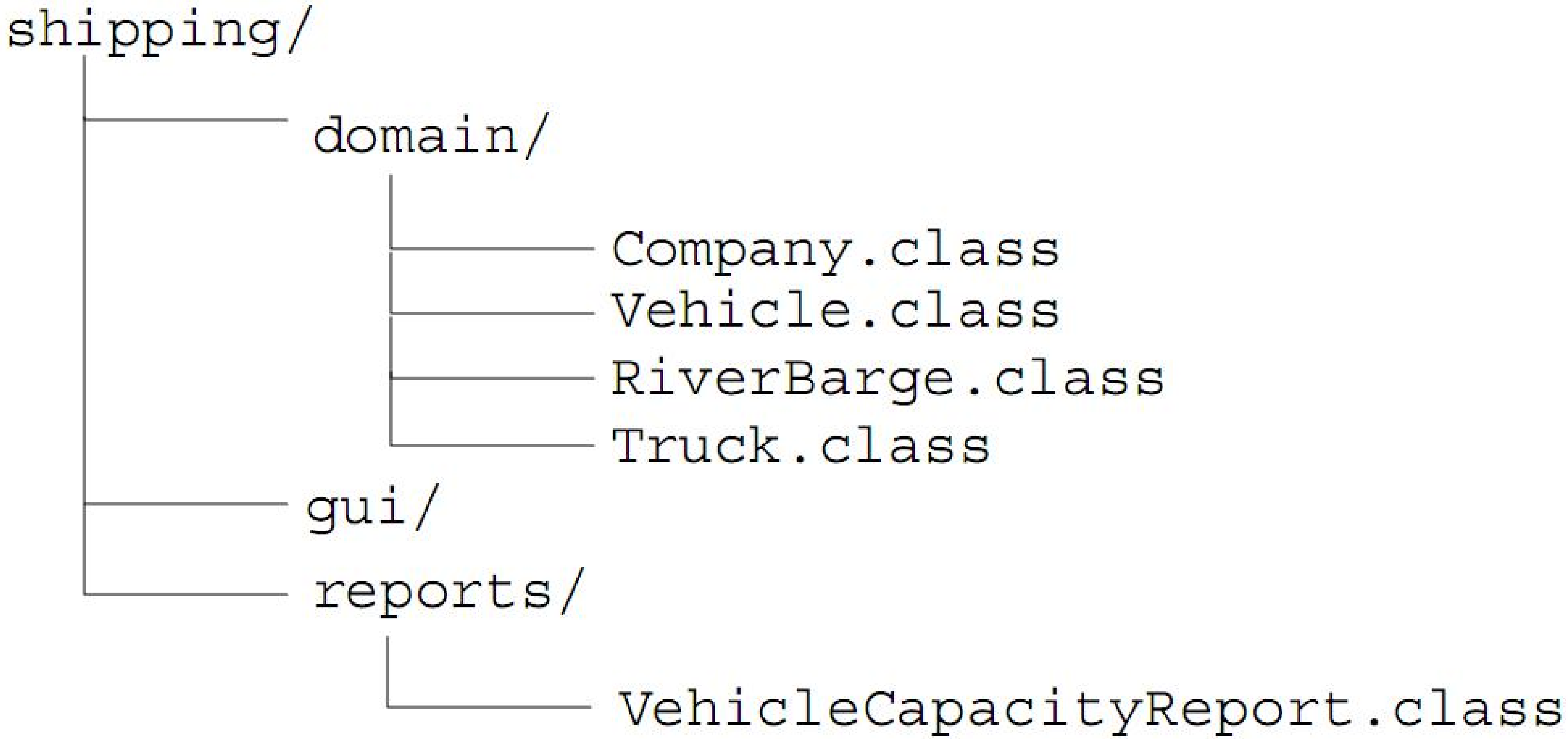
- Examples of the statement are:

```
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

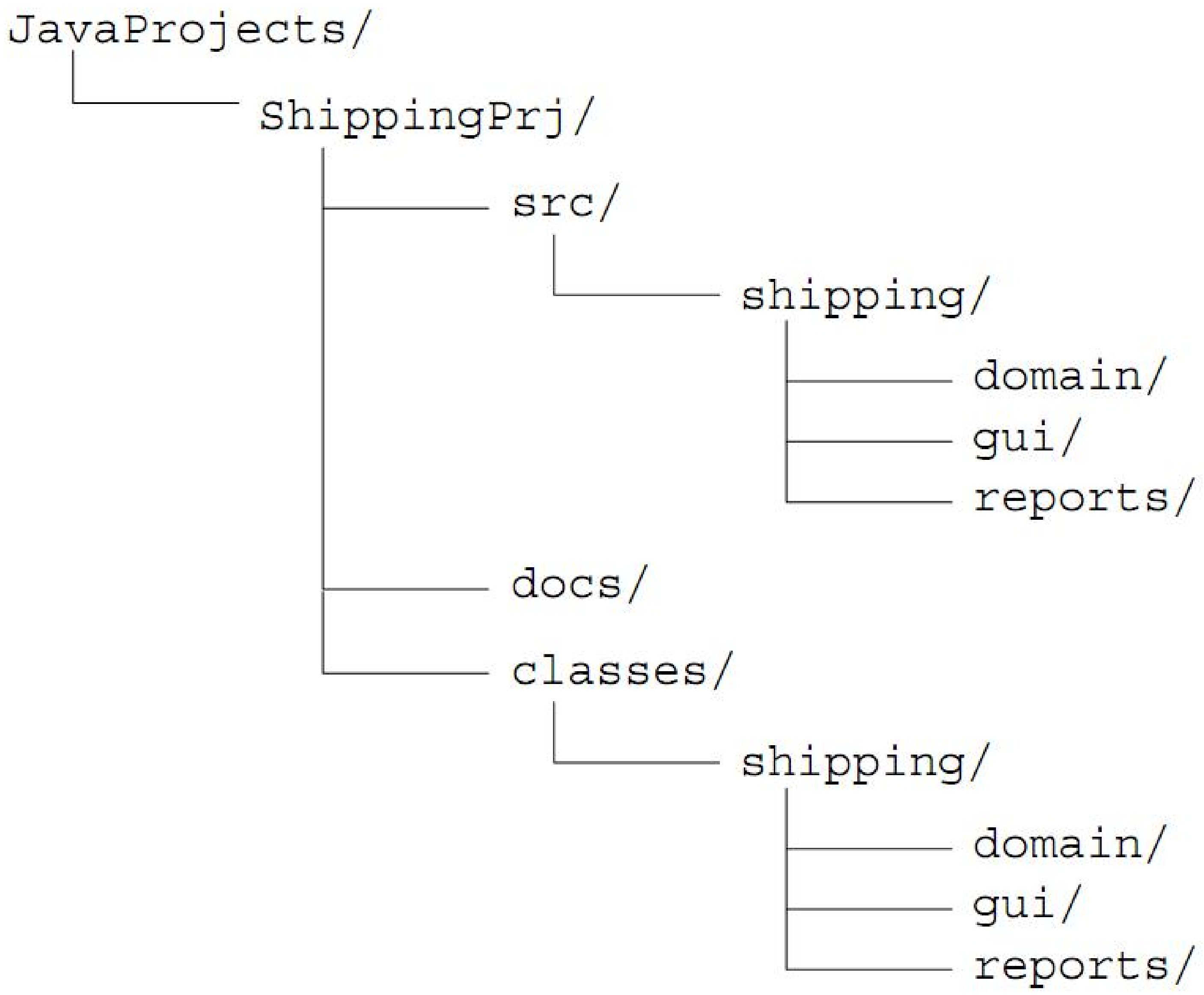
Directory Layout and Packages

- 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



```
JavaProjects/
    ShippingPrj/
        src/
            shipping/
                domain/
                gui/
                reports/
        docs/
        classes/
            shipping/
                domain/
                gui/
                reports/
```

Compiling Using the `-d` Option

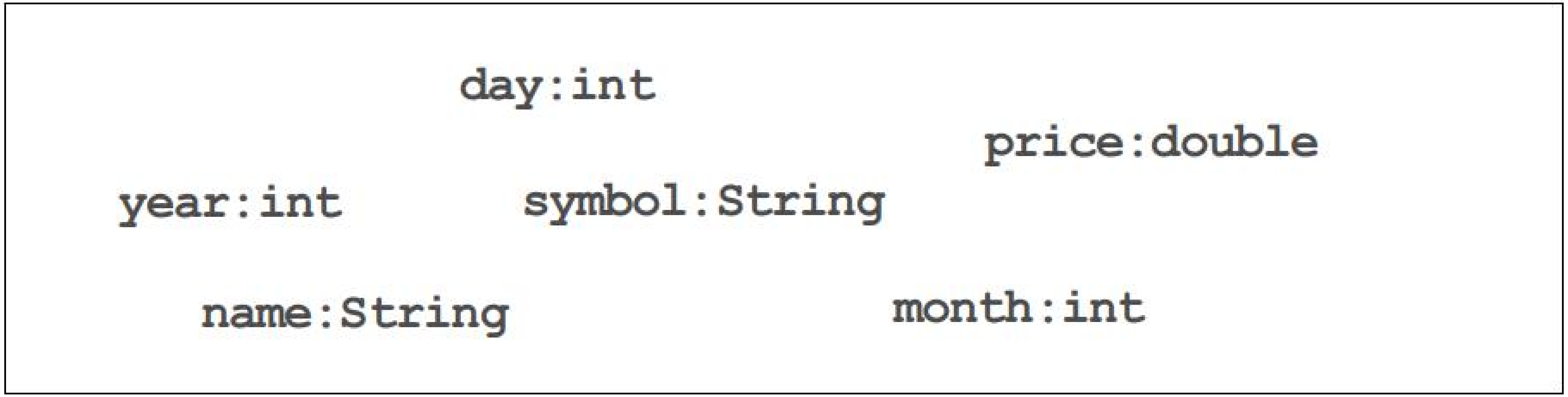
- `cd JavaProjects/ShippingPrj/src`

- `javac -d ../classes shipping/domain/*.java`

- `java -cp ../classes shipping.domain.MyClass`

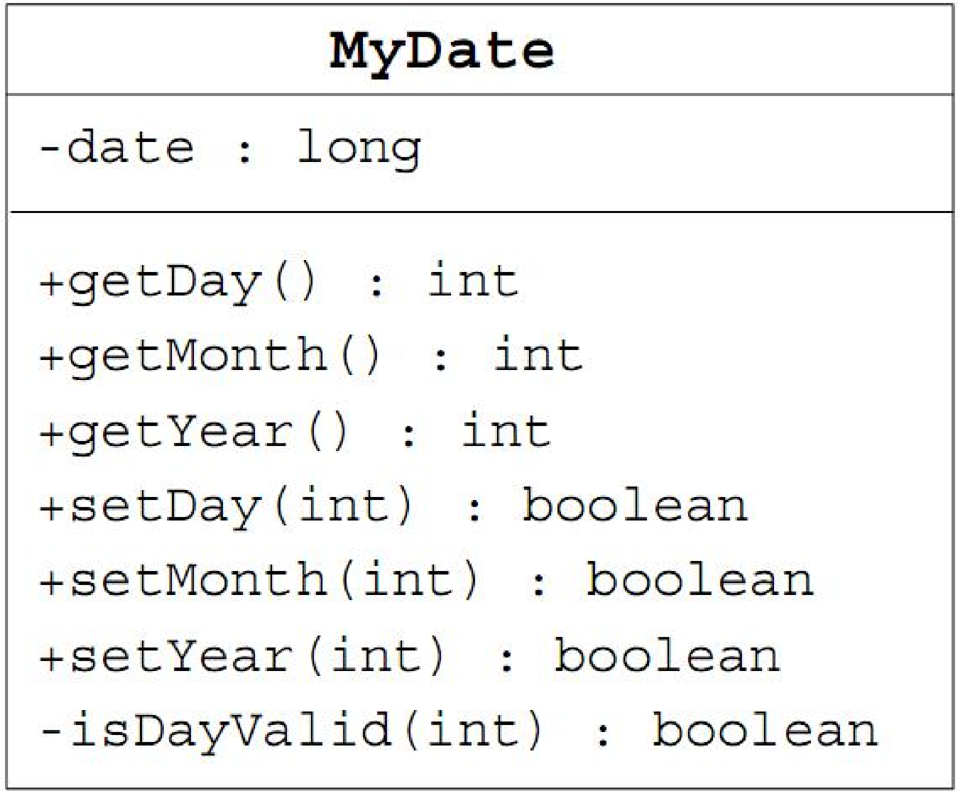
Concept of Encapsulation

- Unencapsulated data



Why Encapsulation ?

- **Hides the implementation details** of a class
- Forces the user to use an **interface** to access data，Protecting **data integrity**
- 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
```

Information Hiding

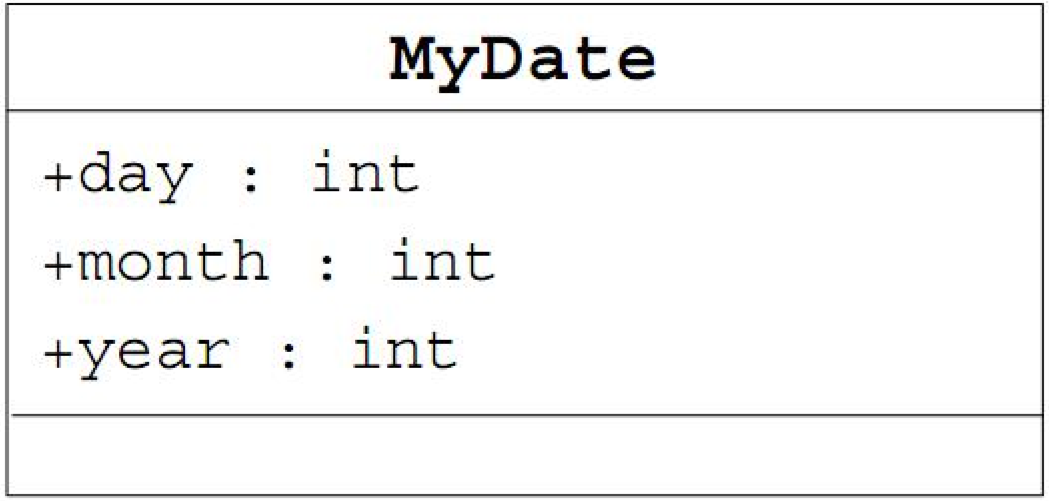
Client code has direct access to internal data (d refers to a MyDate object):

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

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

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

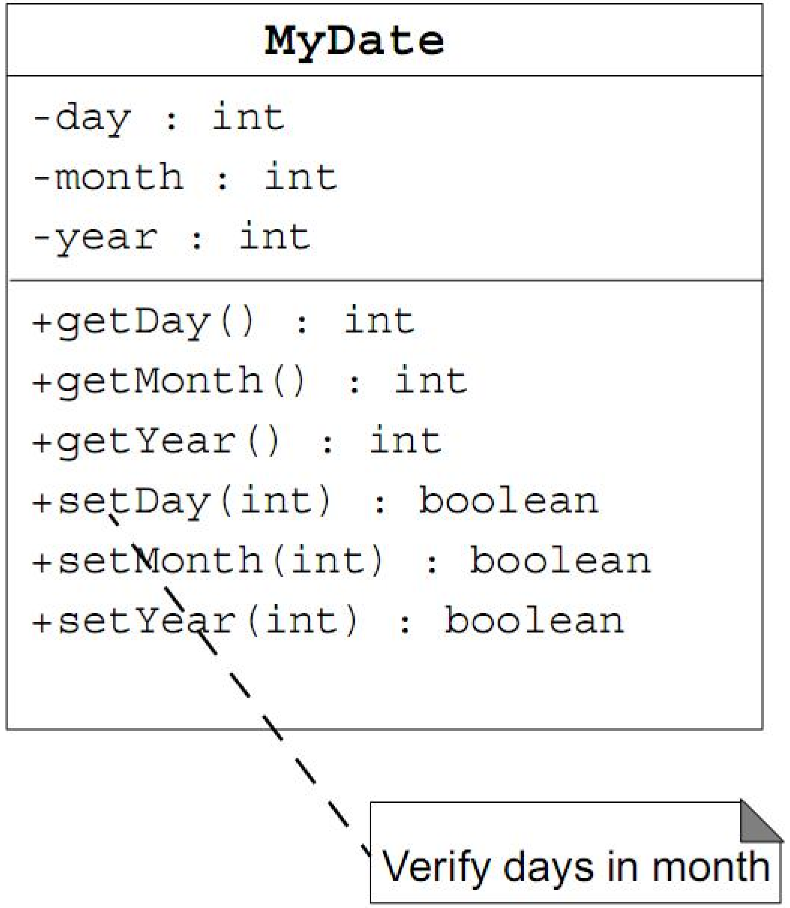
The problem:



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

Information Hiding(Cont')

The solution:



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

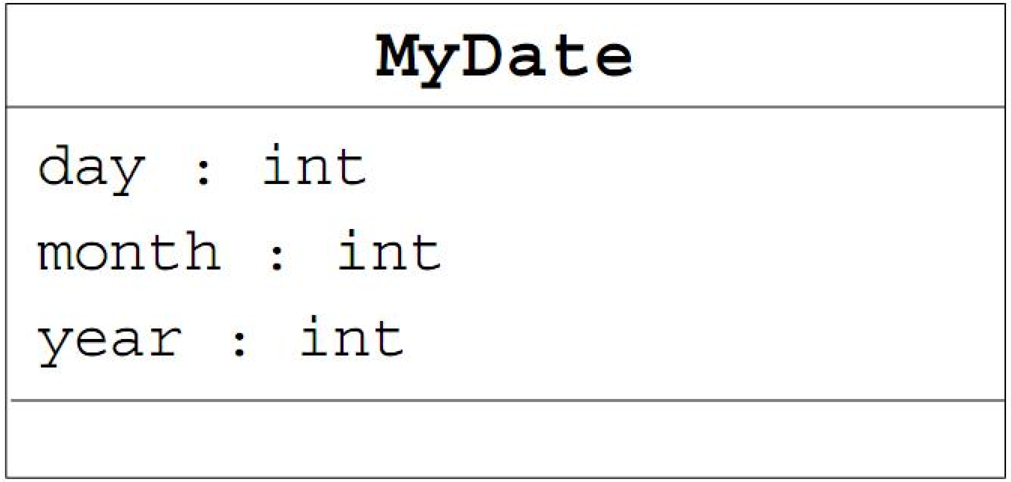
```
MyDate d = new MyDate();
d.setDay(32);
// invalid day, returns false

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
```

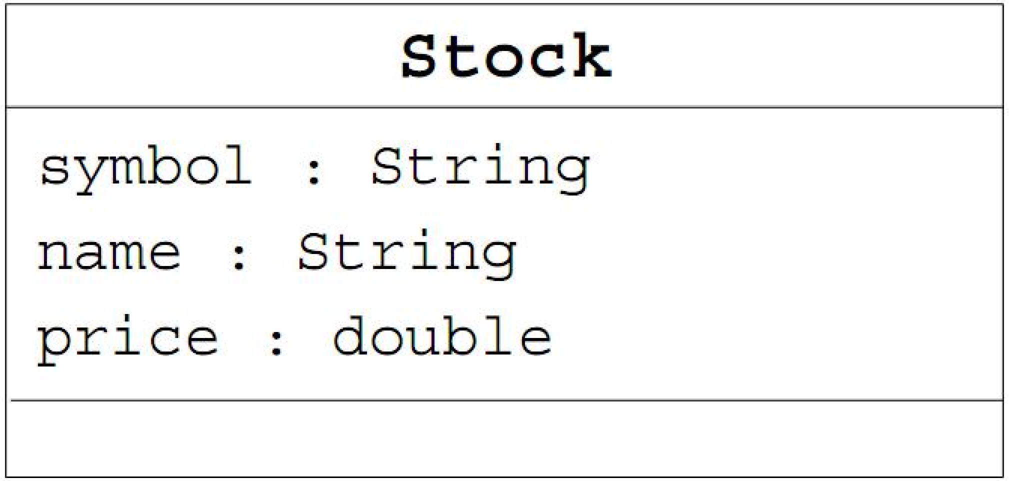
Encapsulation Steps

- Encapsulation Step 1: **Group-Related Data**



MyDate

day : int
month : int
year : int

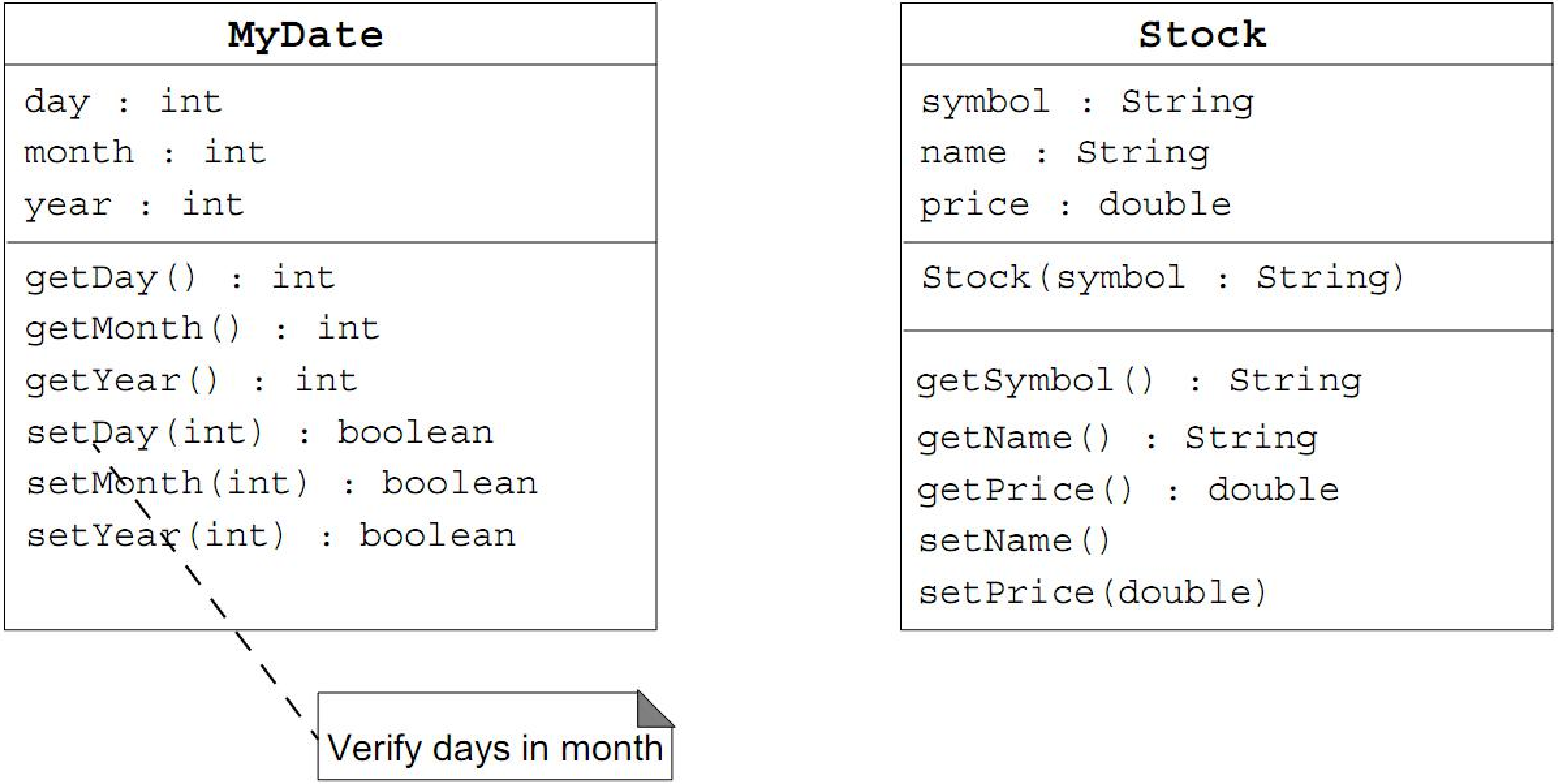


Stock

symbol : String
name : String
price : double

Encapsulation Steps(Cont')

- **Group Data With Behavior**



MyDate

day : int
month : int
year : int

getDay() : int
getMonth() : int
getYear() : int
setDay(int) : boolean
setMonth(int) : boolean
setYear(int) : boolean

Verify days in month

Stock

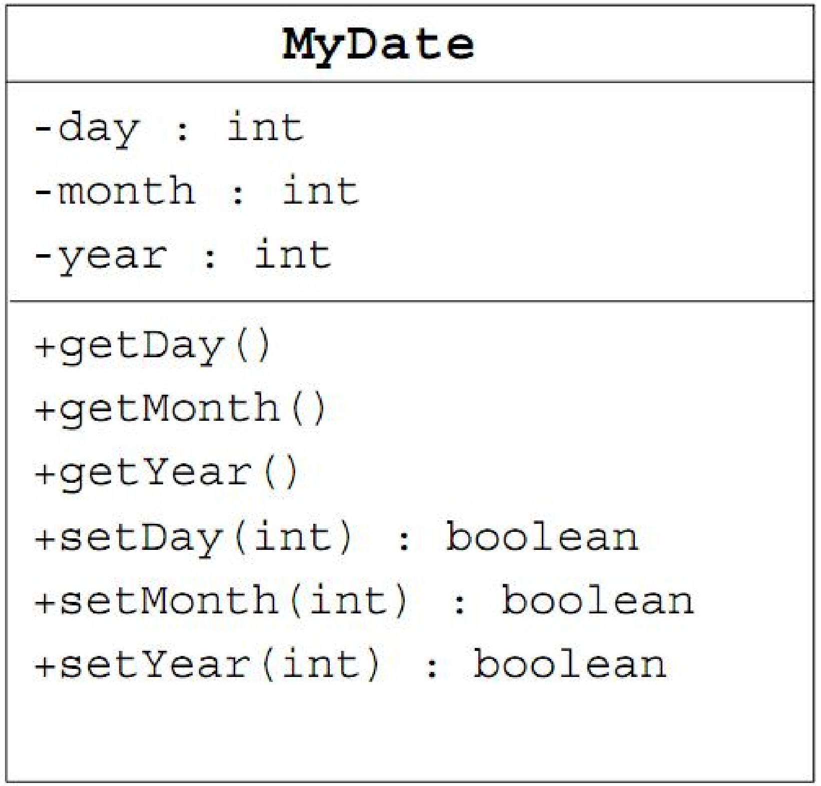
symbol : String
name : String
price : double

Stock(symbol : String)

getSymbol() : String
getName() : String
getPrice() : double
setName()
setPrice(double)

Encapsulation Steps(Cont')

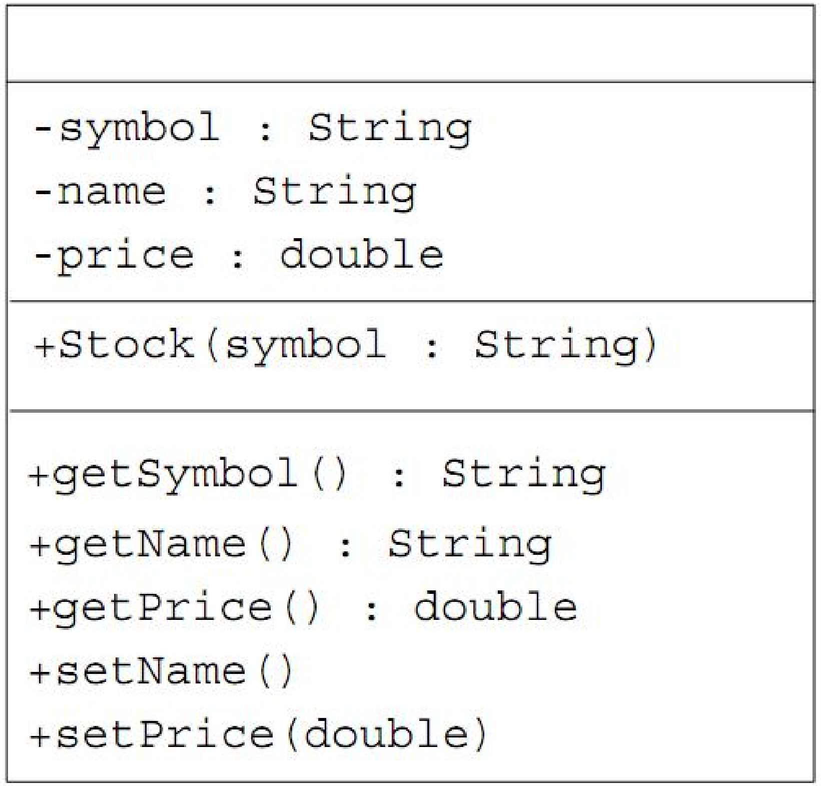
- Implement **Access Control**



MyDate

-day : int
-month : int
-year : int

+getDay()
+getMonth()
+getYear()
+setDay(int) : boolean
+setMonth(int) : boolean
+setYear(int) : boolean



-symbol : String
-name : String
-price : double

+Stock(symbol : String)

+getSymbol() : String
+getName() : String
+getPrice() : double
+setName()
+setPrice(double)

+ symbol represents external or public access

- symbol represents internal or private access

Implementing Encapsulation in Java

- The **package statement**:
  - A class in a package is **visible** and therefore **accessible** to all other classes in the **same package**.
  - A class marked **public** is visible to classes in **other packages**.
  - A class **not marked public** is **hidden** to classes in **other packages**.
- The **class** statement encapsulates attributes, constructors, and methods into a single unit that can be compiled.

Implementing Encapsulation in Java (Cont')

- Access modifiers:
  - **private**
  - **default**
  - **protected**
  - **public**

Modifier	**Same Class**	**Same Package**	**Subclass**	**Universe**
private	Yes			
default	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

Encapsulation Examples

```
package com.abc.util;

public class Date {
  private int day;

  public Date() {//... }

  public void addDays(int days) { }
    int getDaysInMonth(int month) { }
}
```

Encapsulation Examples

```
package com.abc.brokerage;

public class Stock {
private String symbol;
  public Stock(String symbol, double price) { }

  public String getSymbol() { }
  public void setSymbol(String symbol) { }
}
```

```
package com.abc.brokerage;
import abc.util.Date;

class StockAnalyzer {
  private MyDate date;

  double sell(Stock stock, int quantity) { }
  public double buy(Stock stock, int quantity) { }
}
```

Coding Conventions

- Packages:

```
com.example.domain;
```

- Classes, interfaces, and enum types:

```
SavingsAccount
```

- Methods:

```
getAccount()
```

- Variables:

```
currentCustomer
```

- Constants:

```
HEAD_COUNT
```

Coding Conventions(Cont')

- Control structures:

```
if ( condition ) {
  statement1;
} else {
  statement2;
}
```

- Spacing:
  - Use **one statement per line**.
  - Use **two or four spaces for indentation**.
- Comments:
  - Use `//` to comment inline code.
  - Use `/** documentation */` for class members.

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

Questions or Comments?