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*, and *package*
- 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

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*.

Software Engineering

Toolkits / Frameworks / Object APIs (1990s–Up)				
Java 2 SDK	AWT / J.F.C./Swing	Jini™	JavaBeans™	JDBC™

Object-Oriented Languages (1980s–Up)					
SELF	Smalltalk	Common Lisp Object System	Eiffel	C++	Java

Libraries / Functional APIs (1960s–Early 1980s)				
NASTRAN	TCP/IP	ISAM	X-Windows	OpenLook

High-Level Languages (1950s–Up)				Operating Systems (1960s–Up)			
Fortran	LISP	C	COBOL	OS/360	UNIX	MacOS	Microsoft Windows

Machine Code (Late 1940s–Up)

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
  - Inheritance
  - Polymorphism

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 maxLoad;
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  }
```

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
```

Information Hiding

The problem:

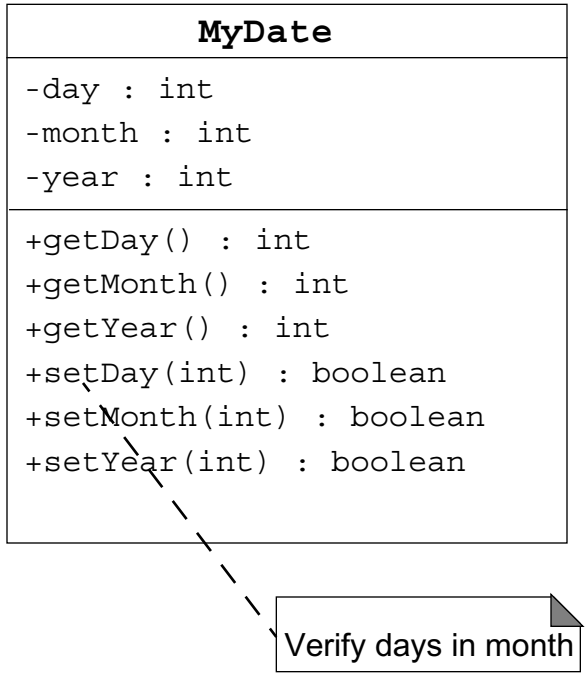
MyDate
+day : int +month : int +year : int

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:

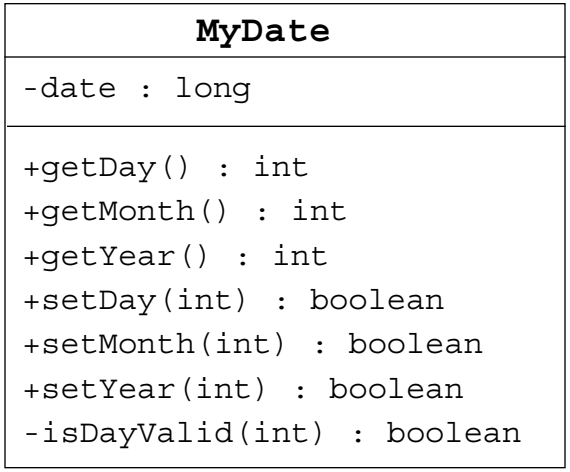


Client code must use setters and getters to access internal data:

```
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



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 = 42;
7     }
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.

Source File Layout

- Basic syntax of a Java source file is:

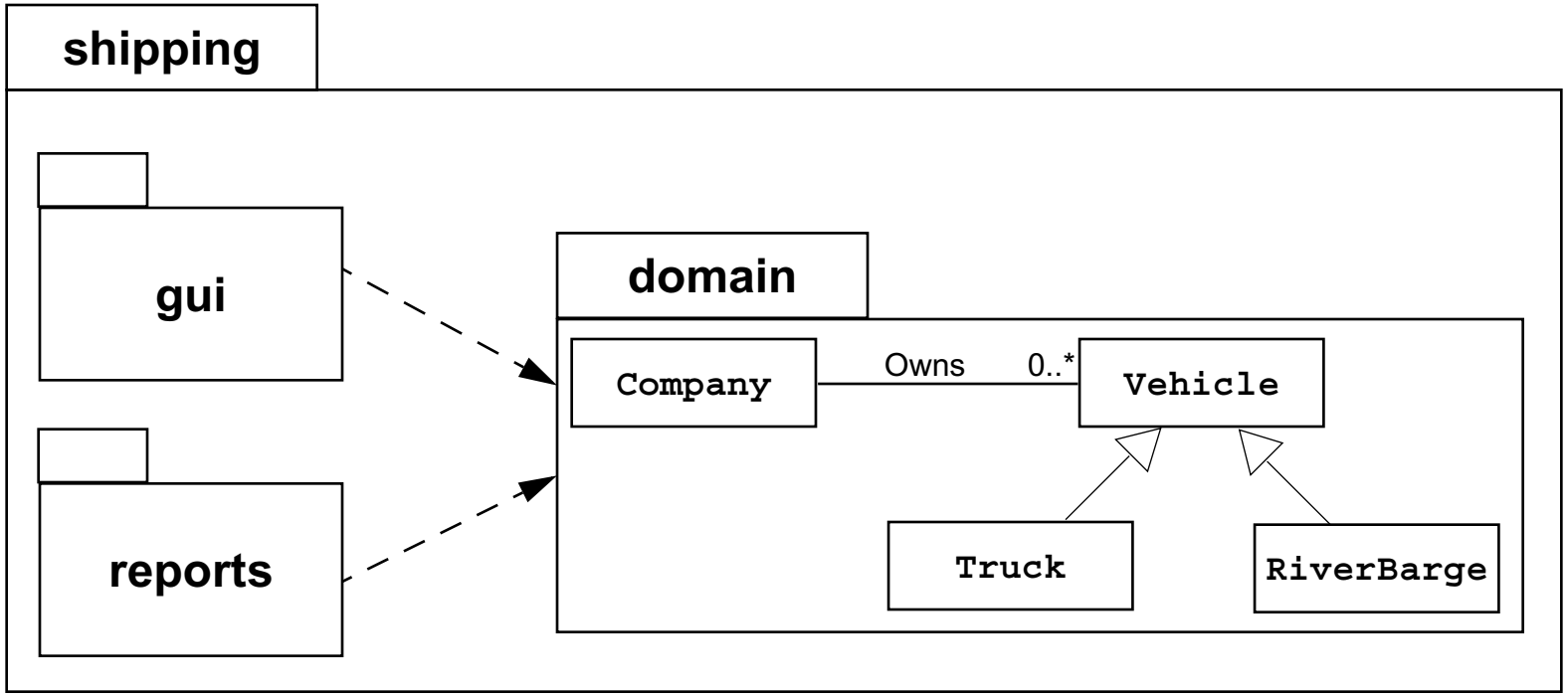
```
[<package_declaration>]
<import_declaration>*
<class_declaration>+
```

- 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  }
```

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>]*.*;
```

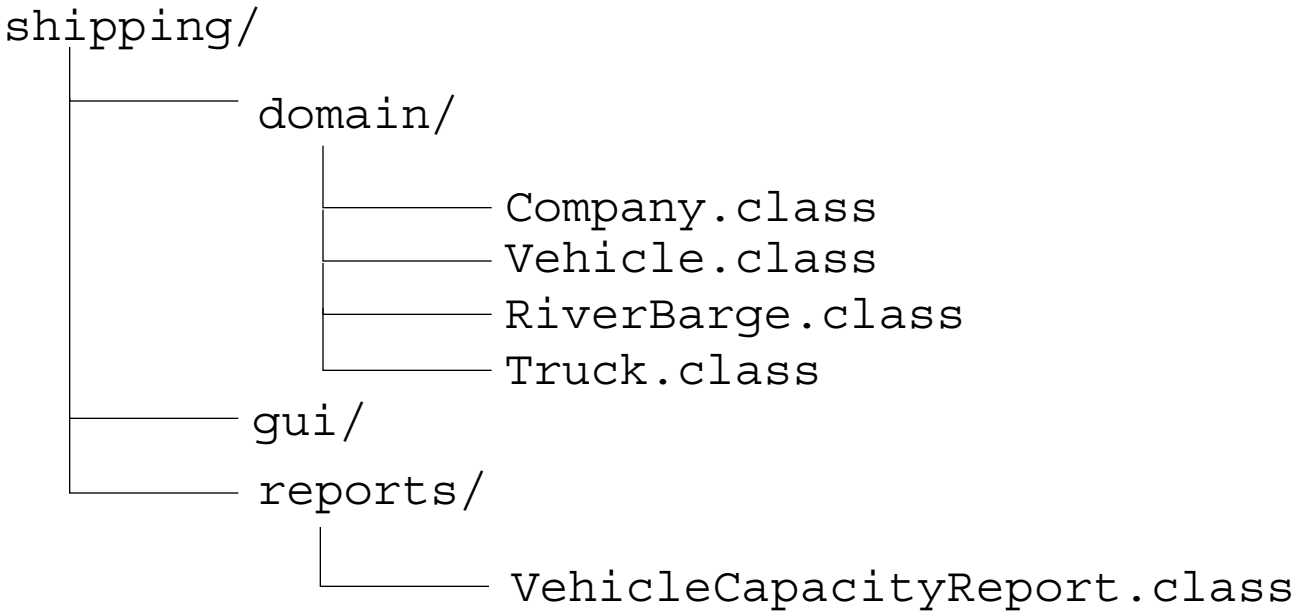
- Examples of the statement are:

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

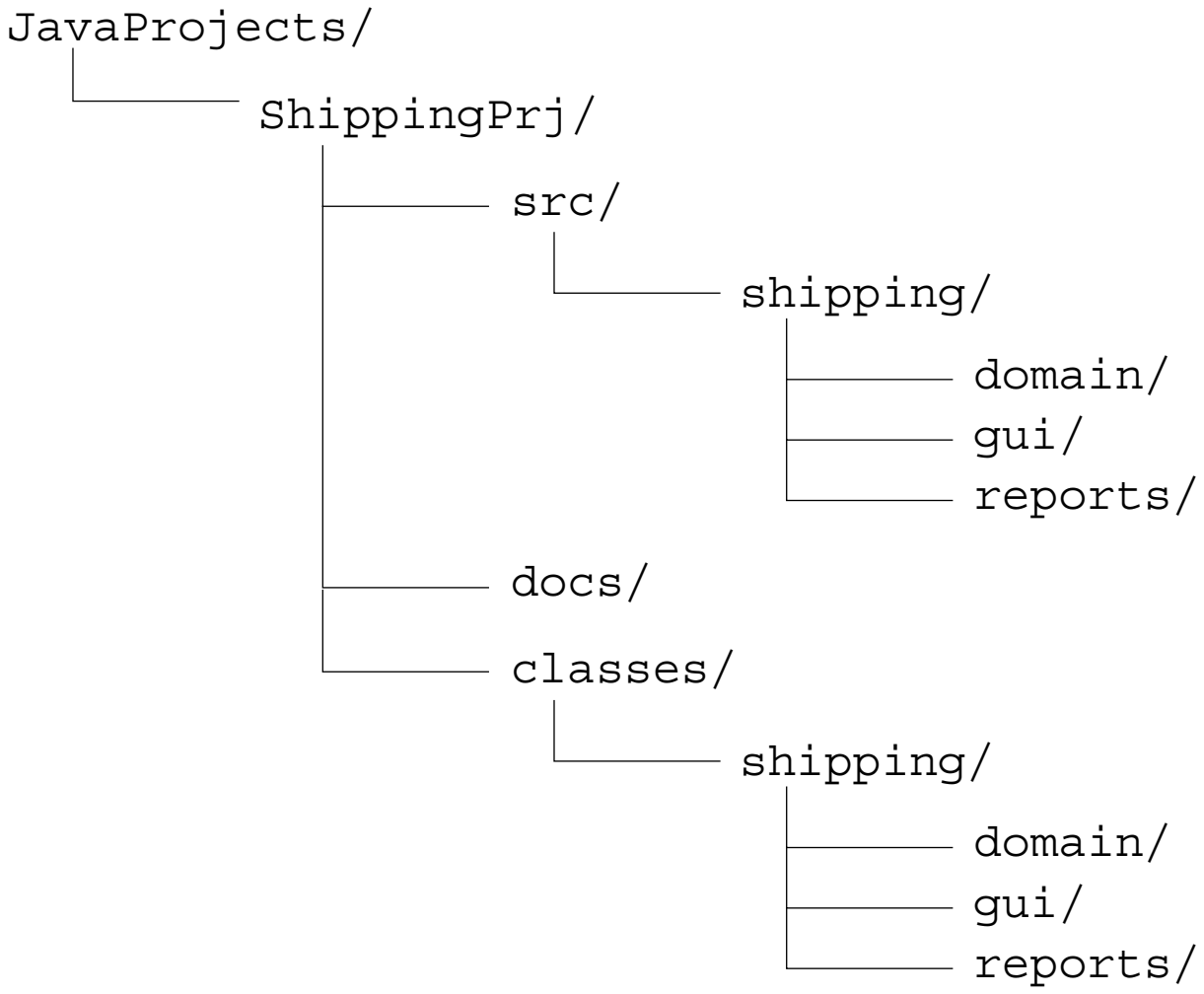
- The import statement does the following:
  - Precedes all class declarations
  - Tells the compiler where to find classes

Directory Layout and Packages

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



Development



Compiling Using the `-d` Option

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

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

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.

Java Technology API Documentation