BBM 102 – Introduction to Programming II

Encapsulation

Today

- Information Hiding
- Encapsulation
- Pre- and Postcondition Comments
- The public and private Modifiers
- UML Class Diagrams
- Overloading
- Packages

Information Hiding

- Programmer using a class method need not know details of implementation
  - Only needs to know *what* the method does

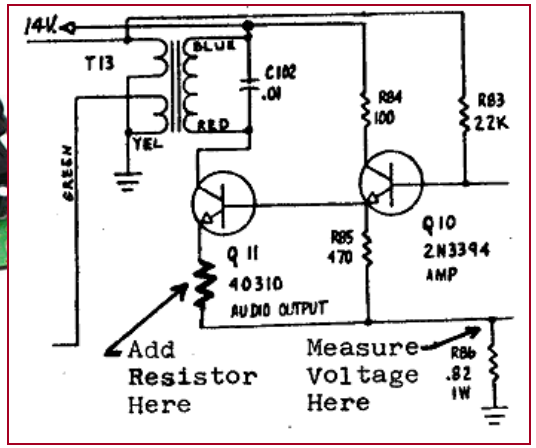
- **Information hiding:**
  - Designing a method so it can be used without knowing details

- Also referred to as ***abstraction***

- Method design should separate *what* from *how*

Encapsulation

- **Encapsulation**: Hiding implementation details of an object from its clients.
  - Encapsulation provides *abstraction*.
    - separates external view (behavior) from internal view (state)
  - Encapsulation protects the integrity of an object's data.



When Creating Classes

- When creating the public interface of a class, give careful thought and consideration to the *contract* you are creating between yourself and users (other programmers) of your class

- Use ***preconditions*** to state what you assume to be true before a method is called
  - caller of the method is responsible for making sure these are true

- Use ***postconditions*** to state what you guarantee to be true after the method is done if the preconditions are met
  - implementer of the method is responsible for making sure these are true

Pre- and Postcondition Comments

- Precondition comment
  - States conditions that must be true before method is invoked

- Example

```
/**
 Precondition: The instance variables of the calling
 object have values.
 Postcondition: The data stored in (the instance variables
 of) the receiving object have been written to the screen.
*/
public void writeOutput()
```

Pre- and Postcondition Comments

- Postcondition comment
  - Tells what will be true after method is executed

- Example

```
/**
 Precondition: years is a nonnegative number.
 Postcondition: Returns the projected population of the
 receiving object after the specified number of years.
*/
public int predictPopulation(int years)
```

Visibility Modifiers

- All parts of a *class* have visibility modifiers
  - Java keywords
  - **public**, protected, **private**
  - do not use these modifiers on local variables (syntax error)

- **public** means that constructor, method, or field may be accessed outside of the class.
  - part of the interface
  - constructors and methods are generally public

- **private** means that part of the class is hidden and inaccessible by code outside of the class
  - part of the implementation
  - data fields are generally private

The `public` and `private` Modifiers

- Type specified as **`public`**
  - Any other class can directly access that object by name
- Classes are generally specified as **`public`**
- Instance variables are usually not **`public`**
  - Instead specify as **`private`**

Private fields

- A field can be declared *private*.
  - No code outside the class can access or change it.

```
private type name;
```

  - Examples:

```
private int id;
private String name;
```

- Client code sees an error when accessing private fields:

```
PointMain.java:11: x has private access in Point
System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
                                      ^
```

Accessing private state

- We can provide methods to get and/or set a field's value:

```
// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}

// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}
```

- Client code will look more like this:

```
System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
p1.setX(14);
```

Programming Example

```
public class Rectangle
{
    private int width;
    private int height;
    private int area;

    public void setDimensions (int newWidth, int newHeight)
    {
        width = newWidth;
        height = newHeight;
        area = width * height;
    }
    public int getArea ()
    {
        return area;
    }
}
```

Note `setDimensions` method : This is the only way the `width` and `height` may be altered outside the class

- Statement such as

  `box.width = 6;`

  is illegal since width is `private`
- Keeps remaining elements of the class consistent

Point class

```
// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }

    public void translate(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
```

Client code

```
public class PointMain4 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);

        // print each point
        System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");

        // move p2 and then print it again
        p2.translate(2, 4);
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");
  }
}
```

OUTPUT :

```
p1 is (5, 2)
p2 is (4, 3)
p2 is (6, 7)
```

Encapsulation

- Consider example of driving a car
  - We see and use break pedal, accelerator pedal, steering wheel – know what they do
  - We do not see mechanical details of how they do their jobs

- Encapsulation divides class definition into
  - *Class interface*
  - *Class implementation*

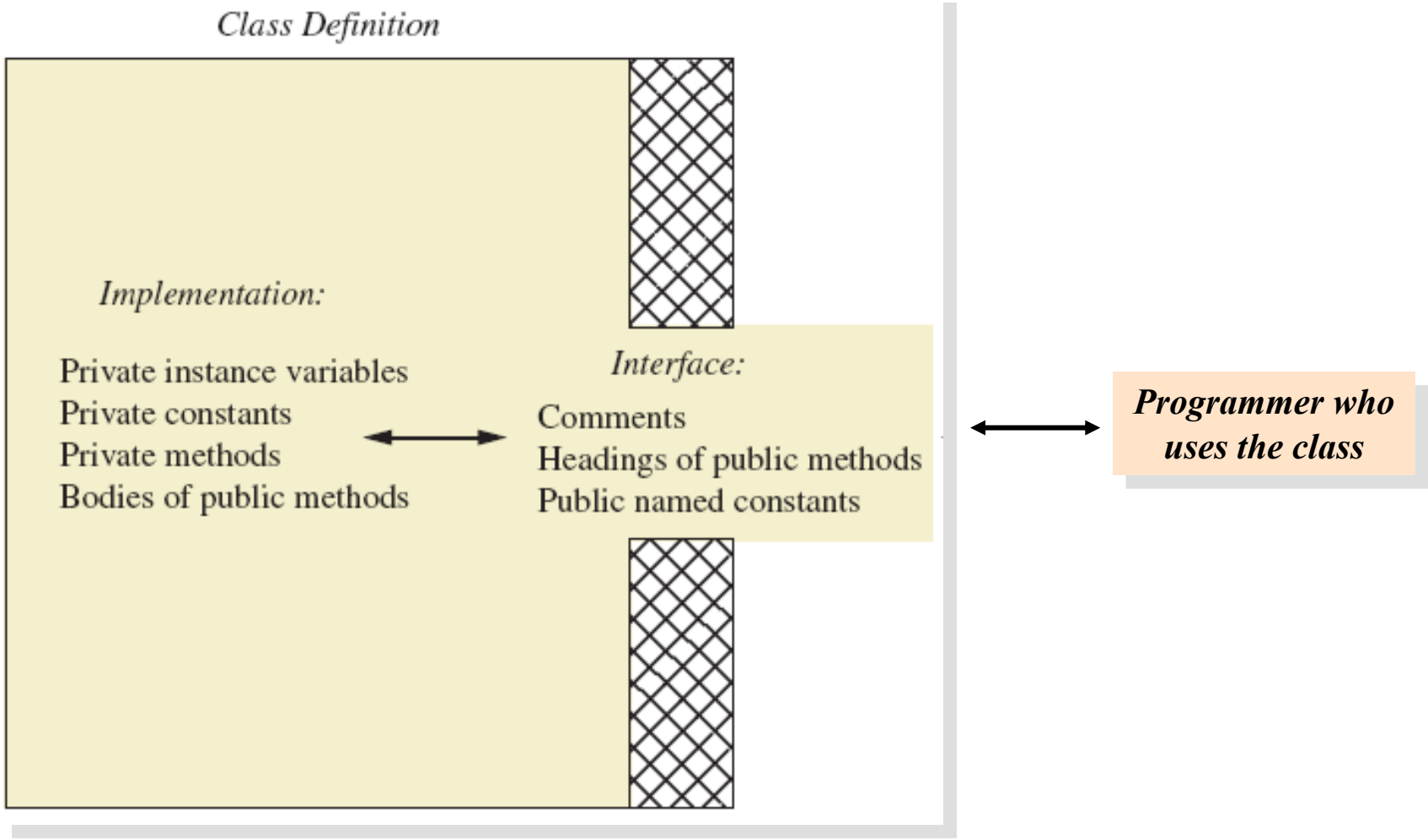
Encapsulation

- *Class interface*
  - Tells what the class does
  - Gives headings for public methods and comments about them

- *Class implementation*
  - Contains private variables
  - Includes definitions of public and private methods

Encapsulation

- A well encapsulated class definition



Encapsulation – Best Practices

- Preface class definition with comment on how to use class
- Declare all instance variables in the class as private.
- Provide public accessor methods to retrieve data and provide public methods to manipulate data

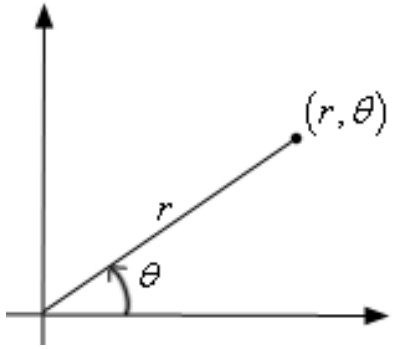
  Such methods could include public mutator methods.
- Place a comment before each public method heading that fully specifies how to use method.
- Make any helping methods private.
- Write comments within class definition to describe implementation details.

Benefits of encapsulation

- Provides abstraction between an object and its clients.

- Protects an object from unwanted access by clients.
  - A bank app forbids a client to change an `Account`'s balance.

- Allows you to change the class implementation.
  - `Point` could be rewritten to use polar coordinates (radius *r*, angle ϑ), but with the same methods.



- Allows you to constrain objects' state (**invariants**).
  - Example: Only allow `Points` with non-negative coordinates.

Software Development Observations

- Interfaces change less frequently than implementations.
- When an implementation changes, implementation-dependent code must change accordingly.
- Hiding the implementation reduces the possibility that other program parts will become dependent on class-implementation details.

Outline

Time1.java

(1 of 2)

```
1  // Fig. 8.1: Time1.java
2  // Time1 class declaration maintains the time in 24-hour format.
3
4  public class Time1
5  {
6     private int hour;   // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // set a new time value using universal time; ensure that
11    // the data remains consistent by setting invalid values to zero
12    public void setTime( int h, int m, int s )
13
14       hour = ( ( h >= 0 && h < 24 ) ? h : 0 );   // validate hour
15       minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
16       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
17    } // end method setTime
18
```

private instance variables

Declare **public** method **setTime**

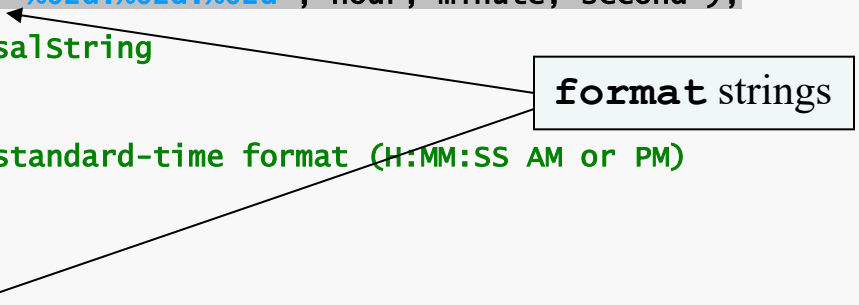
Validate parameter values before setting instance variables

Outline

Time1.java

(2 of 2)

```
19    // convert to String in universal-time format (HH:MM:SS)
20    public String toUniversalString()
21    {
22       return String.format( "%02d:%02d:%02d", hour, minute, second );
23    } // end method toUniversalString
24
25    // convert to String in standard-time format (H:MM:SS AM or PM)
26    public String toString()
27    {
28       return String.format( "%d:%02d:%02d %s",
29          ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
30          minute, second, ( hour < 12 ? "AM" : "PM" ) );
31    } // end method toString
32 } // end class Time1
```



Outline

Time1Test.java
(1 of 2)

```
1  // Fig. 8.2: Time1Test.java
2  // Time1 object used in an application.
3
4  public class Time1Test
5  {
6     public static void main( String args[] )
7     {
8        // create and initialize a Time1 object
9        Time1 time = new Time1(); // invokes Time1 constructor
10
11       // output string representations of the time
12       System.out.print( "The initial universal time is: " );
13       System.out.println( time.toUniversalString() );
14       System.out.print( "The initial standard time is: " );
15       System.out.println( time.toString() );
16       System.out.println(); // output a blank line
17
```

Create a **`Time1`** object

Call **`toUniversalString`** method

Call **`toString`** method

Outline

Time1Test.java

(2 of 2)

```
18       // change time and output updated time
19       time.setTime( 13, 27, 6 );
20       System.out.print( "Universal time after setTime is: " );
21       System.out.println( time.toUniversalString() );
22       System.out.print( "Standard time after setTime is: " );
23       System.out.println( time.toString() );
24       System.out.println(); // output a blank line
25
26       // set time with invalid values; output updated time
27       time.setTime( 99, 99, 99 );
28       System.out.println( "After attempting invalid settings:" );
29       System.out.print( "Universal time: " );
30       System.out.println( time.toUniversalString() );
31       System.out.print( "Standard time: " );
32       System.out.println( time.toString() );
33    } // end main
34 } // end class Time1Test
```

Call **`setTime`** method

Call **`setTime`** method with invalid values

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Software Development Observations & Tips

- When one object of a class has a reference to another object of the same class, the first object can access all the second object's data and methods (including those that are `private`).

- When implementing a method of a class, use the class's *set* and *get* methods to access the class's `private` data. This simplifies code maintenance and reduces the likelihood of errors.

- This architecture helps hide the implementation of a class from its clients, which improves program modifiability

`final` Instance Variables

- `final` instance variables
  - Keyword `final`
    - Specifies that a variable is not modifiable (is a constant)
  - `final` instance variables can be initialized at their declaration
    - If they are not initialized in their declarations, they must be initialized in all constructors

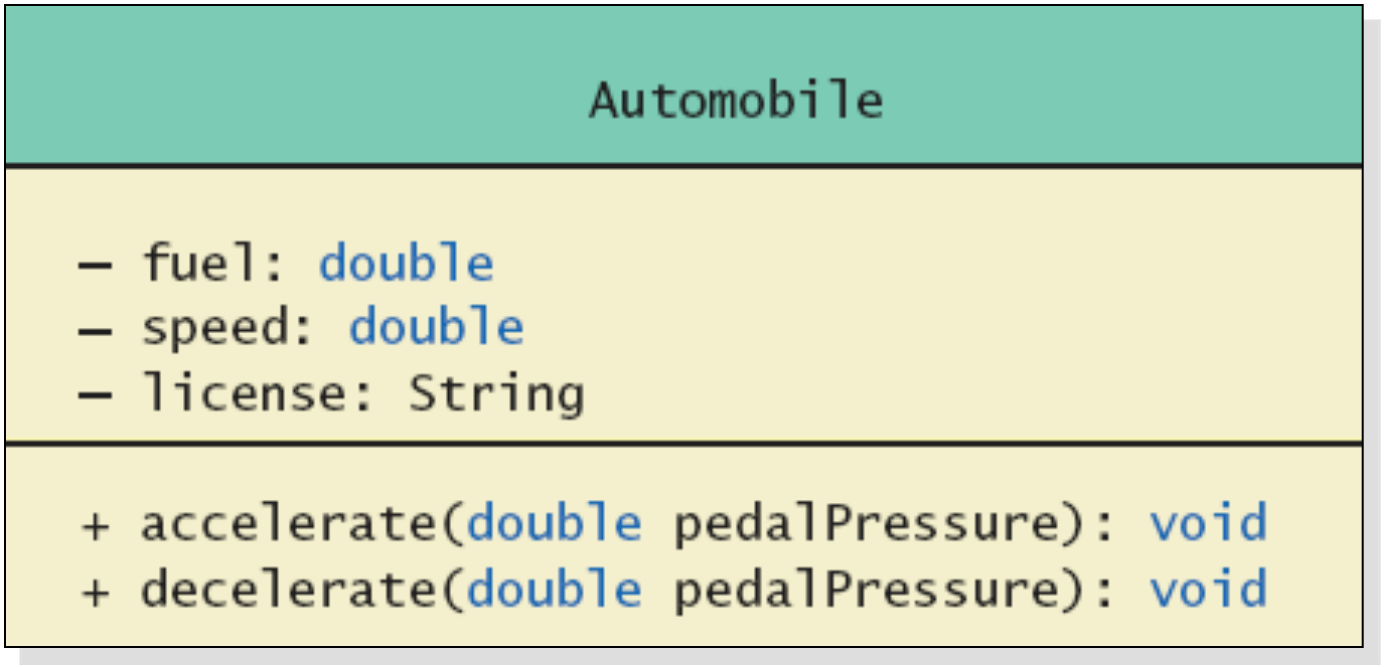
- If an instance variable should not be modified, declare it to be `final` to prevent any erroneous modification.

`static final` Instance Variables

- A `final` field should also be declared `static` if it is initialized in its declaration.
- Once a `final` field is initialized in its declaration, its value can never change.
- Therefore, it is not necessary to have a separate copy of the field for every object of the class.
- Making the field `static` enables all objects of the class to share the `final` field.
- Example: `public static final double PI = 3.141592;`

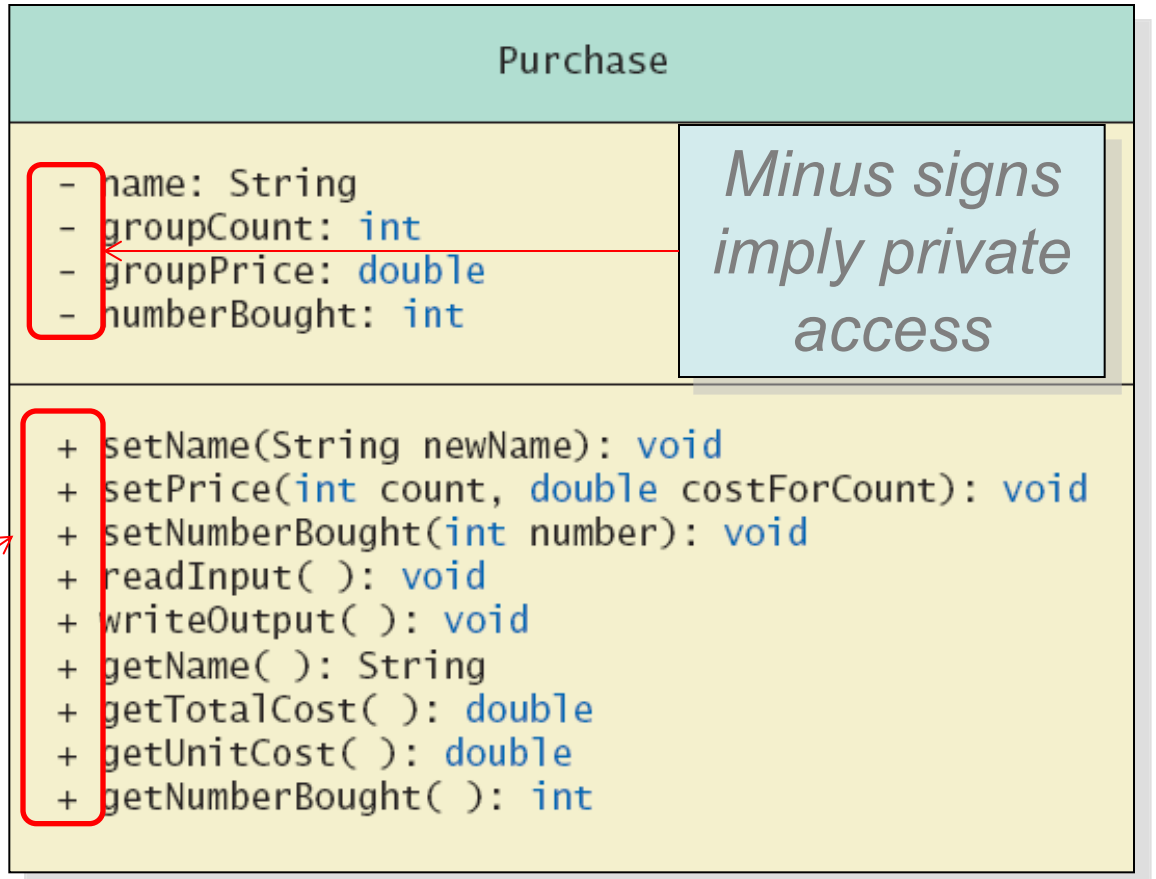
UML Class Diagrams

- An automobile class outline as a UML class diagram



UML Class Diagrams

- Example:
  `Purchase`
  class



UML Class Diagrams

- Contains more than interface, less than full implementation
- Usually written *before* class is defined
- Used by the programmer defining the class
  - Contrast with the *interface* used by programmer who uses the class

Packages and Importing

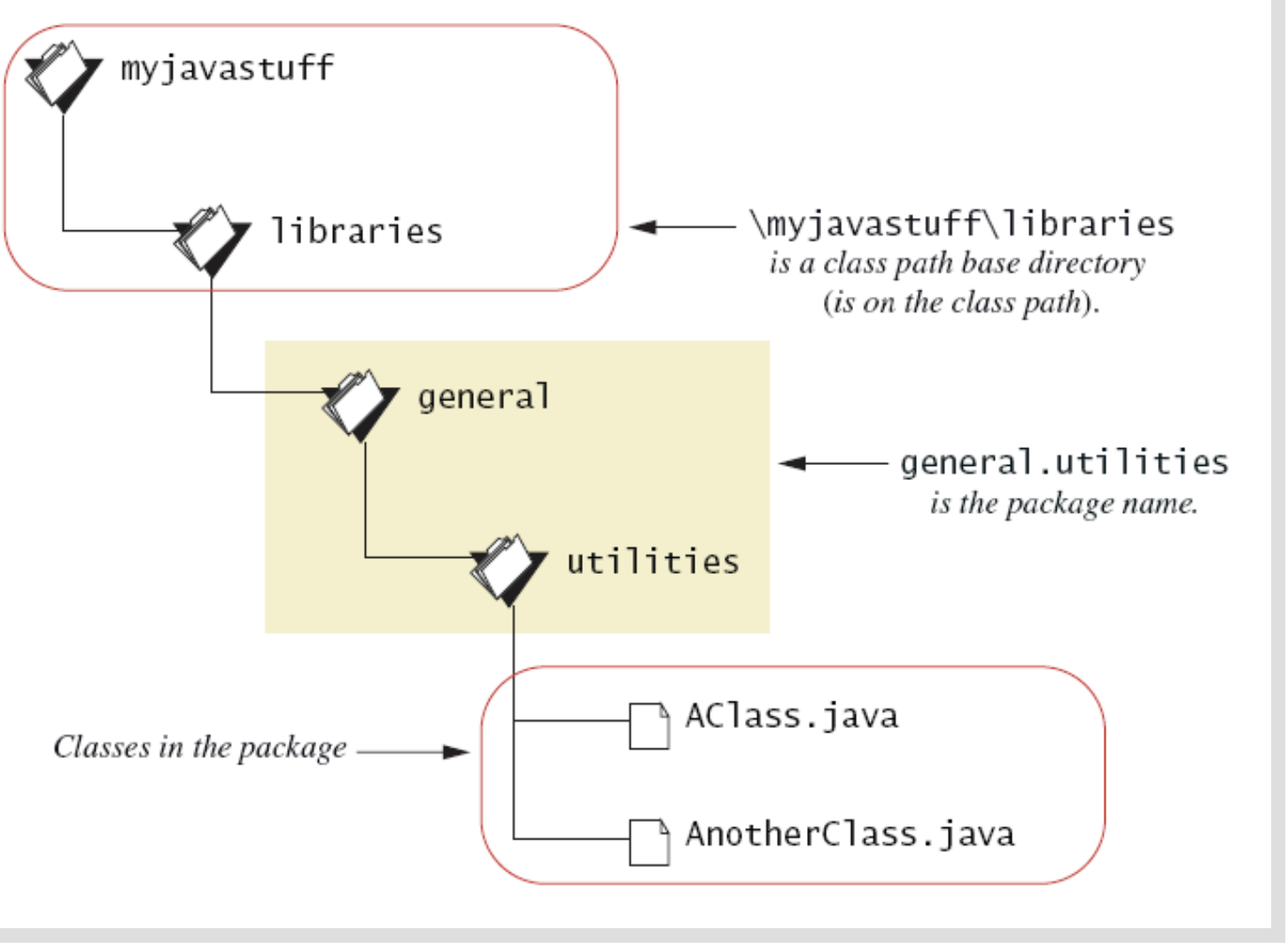
- A **package** is a collection of classes grouped together into a folder
- Name of folder is name of package
- Each class
  - Placed in a separate file
  - Has this line at the beginning of the file
    `package Package_Name;`
- Classes use packages by use of **`import`** statement

Package Names and Directories

- Package name tells compiler path name for directory containing classes of package
- Search for package begins in class path base directory
  - Package name uses dots in place of / or \
- Name of package uses relative path name starting from any directory in class path

Package Names and Directories

- A package name



Time Class Case Study: Creating Packages

- To declare a reusable class
  - Declare a `public` class
  - Add a `package` declaration to the source-code file
    - must be the very first executable statement in the file
      - Package name example: `com.deitel.jhtp6.ch08`
      - `package` name is part of the fully qualified class name
        - » Distinguishes between multiple classes with the same name belonging to different packages
        - » Prevents name conflict (also called name collision)

Example

- Time1.java

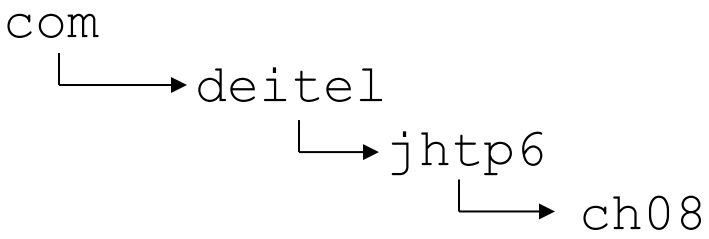
```
1  // Fig. 8.18: Time1.java
2  // Time1 class declaration maintains the time in 24-hour format.
3  package com.deitel.jhtp6.ch08;
4
5  public class Time1
6  {
7     private int hour;   // 0 - 23
8     private int minute; // 0 - 59
9     private int second; // 0 - 59
10
11    // set a new time value using universal time; perform
12    // validity checks on the data; set invalid values to zero
13    public void setTime( int h, int m, int s )
14    {
15       hour = ( ( h >= 0 && h < 24 ) ? h : 0 );   // validate hour
16       minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
17       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
18    } // end method setTime
19 }
```

`package` declaration

`Time1` is a **`public`** class so it can be used by importers of this package

Time Class Case Study: Creating Packages (Cont.)

- Compile the class so that it is placed in the appropriate package directory structure
  - Example: our package should be in the directory



  - `javac` command-line option `-d`
    - `javac` creates appropriate directories based on the class's `package` declaration
    - A period (`.`) after `-d` represents the current directory

Time Class Case Study: Creating Packages (Cont.)

- Import the reusable class into a program
  - Single-type-import declaration
    - Imports a single class
    - Example: `import java.util.Random;`
  - Type-import-on-demand declaration
    - Imports all classes in a package
    - Example: `import java.util.*;`

Name Clashes

- Packages help in dealing with name clashes
  - When two classes have same name

- Different programmers may give same name to two classes
  - Ambiguity resolved by using the package name

Overloading Basics

- When two or more methods have same name within the same class
- Java distinguishes the methods by number and types of parameters
  - If it cannot match a call with a definition, it attempts to do type conversions
- A method's name and number and type of parameters is called the *signature*

Programming Example

```
/** This class illustrates overloading. */
public class Overload {

public static void main (String [] args) {
   double average1 = Overload.getAverage (40.0, 50.0);
   double average2 = Overload.getAverage (1.0, 2.0, 3.0);
   char average3 = Overload.getAverage ('a', 'c');
       System.out.println ("average1 = " + average1);
       System.out.println ("average2 = " + average2);
       System.out.println ("average3 = " + average3); }

public static double getAverage (double first, double second) {
       return (first + second) / 2.0; }

public static double getAverage (double first, double second,
double third) { return (first + second + third) / 3.0; }

public static char getAverage (char first, char second) {
       return (char) (((int) first + (int) second) / 2); }
}
```

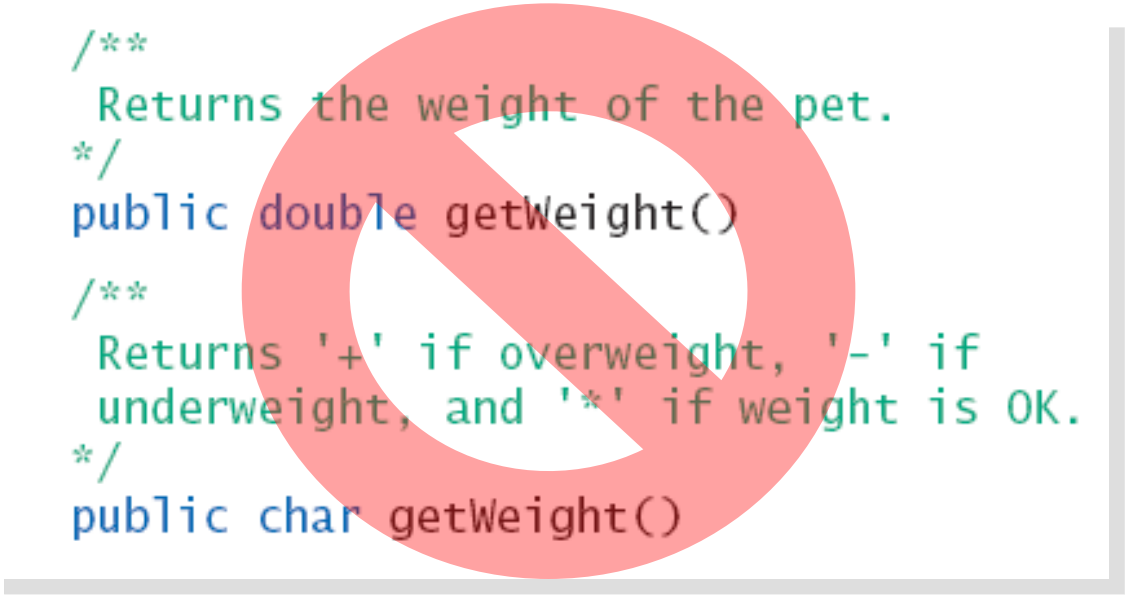
```
average1= 45.0
average2= 2.0
average3 = b
```

Overloading and Type Conversion

- Overloading and automatic type conversion can conflict
- Remember the compiler attempts to overload before it does type conversion
- Use descriptive method names, avoid overloading when possible

Overloading and Return Type

- You must not overload a method where the only difference is the type of value returned



```
/**
 Returns the weight of the pet.
*/
public double getWeight()

/**
 Returns '+' if overweight, '-' if
 underweight, and '*' if weight is OK.
*/
public char getWeight()
```

Acknowledgments

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012
  - Java - How to Program, Paul Deitel and Harvey Deitel, Prentice Hall, 2012
  - Mike Scott, CS314 Course notes, University of Texas Austin