1 Object-Oriented Thinking

CST141

2 Class Abstraction and Encapsulation

- Class abstraction is the separation of class implementation from class use
  - It is not necessary to understand the class’s implementation (its code) to use it in classes
  - Class encapsulation means that the details of implementation are hidden from the user
  - The class interface makes it possible to use a class without knowledge of its implementation

3 Interface vs. Implementation (Page 1)

- The interface (the class documentation) consists of:
  - The name of the class
  - A general description of the class
  - A list of constructors and methods
  - The return values and parameters for constructors and methods
  - The description of the purpose of each constructor and method
  - The constants and any other components

4 Interface vs. Implementation (Page 2)

- The interface does *not include* the class implementation:
  - The private data fields
  - Any public methods
  - The bodies (source code) for each method

5 Documentation

- “Document everything”
- *Write your comments first*
  - Before you write the method
  - If you do not know what to write, you probably do not understand fully what the method is supposed to do

6 Writing Class Documentation

- Your own classes can be documented the same way as are Java API library classes
  - Use your classes to *create* an interface, e.g. “library class”
- Others should be able to use your classes by reading the interface (documentation) without access to the implementation

7 Elements of Documentation (Page 1)

- Documentation for a *class* should include:
  - The class name (and inheritance hierarchy)
  - A comment describing the overall purpose, function, and characteristics of the class
  - A version number
  - The name of the author or authors
  - Documentation for each constructor and each method in the class

8 Elements of Documentation (Page 2)

- Documentation for *methods* (all methods including constructors) should include:

  - The method name, as well as a comment describing its purpose and function
  - The parameter names and types, including a description
  - The return type, including a description

9 The Javadoc Utility (Page 1)

- Javadoc.exe is a standard, convenient tool to document Java code (part of Java JDK)
- Requires *special formatting* of comments
- This utility reads the formatted comments, and automatically generates an HTML document based on those comments
- The HTML files provide convenience of hyperlinks from one document to another, as well as within each document

10 The Javadoc Utility (Page 2)

- Two kinds of Javadoc comments:
  - Class-level comments—provides overall description of the classes
  - Member-level comments—describes the purpose(s) of the members (e.g. usually the methods)
- Both types of comments always start with the characters /** and end with */

11 Class-Level Comments (Page 1)

- Class-level comments provide an overall description of the class
- Placed just above *class header*
  - May not be followed by any other elements before the class header (e.g. import)
- Generally contain *author* and *version number* tags, and a description of the class

12 Class-Level Comments (Page 2)

- Example class-level comment:

```
/**
 * The Payee class calculates payroll
 * for regular and overtime workers.
 * Users update data fields by calling
 * the setHoursWorked() and setPayRate()
 * methods.
 *
 * @author Carl B. Struck
 * @version 1.0
*/
public class Payee
{
```

13 Tags

- Tags are formatting elements that start with ampersand (@) character and are formatted in the documentation by Javadoc.exe utility
- The @author tag describes the author(s)

  @author Carl B. Struck
- The @version tag describes the version number or similar information

  @version 1.0

15 Member-Level Comments (Page 1)

- Member-level comments describe the fields, methods, and constructors

- Placed directly above each *method header*

16 Member-Level Comments (Page 2)

- Member-level tags may include:
  - The @param tag which describes each of the method’s required parameters
  - The @return tag describes the return value of a *non-void* method
  - The exceptions which the method throws (Chapter 14)

17 Member-Level Comments (Page 3)

- The @param tag describes each of the method’s required parameters
  - There may be more than one @param for a method if it takes more than one parameter
  - First word always is the parameter variable name and it will be followed by a hyphen (-) in the generated documentation
  - Example:

    @param hoursWorked the employee number of hours worked

18 Member-Level Comments (Page 4)

- Member-level comment with a @param tag:

```
/**
 * Mutator method for the hours worked
 * data field. Validates that hours
 * worked is between 0.25 and 60.0.
 *
 * @param hoursWorked the employee number
 *                  of hours worked
 */
public void setHoursWorked(int hoursWorked)
{
```

20 Member-Level Comments (Page 5)

- The @return tag describes the return value of a *non-void* method
- Example:

  @return Employee number of hours worked as a double

21 Member-Level Comments (Page 6)

- A member-level comment with a @return tag:

```
/**
 * Accessor method for the hours worked
 * data field.
 *
 * @return Employee number of hours
 *        worked as a double
 */
public String getHoursWorked()
{
```

32 Object Composition

- Composition is the relationship in which one object *contains* another object, e.g.:

  - A Date object instantiated within another class
  - A String object instantiated in another class
  - Any object of a programmer-defined class that is instantiated within another class
- Most professional applications include a large number of classes working together

33 Cohesion (Page 1)

- Cohesion is a measure of the number and the diversity of tasks for which a single unit is responsible
- If each unit is responsible for *one single* logical task, we say it has high cohesion
- Cohesion applies to classes and methods

34 Cohesion (Page 2)

- High cohesion makes it easier to:
  - Understand what a class or method does
  - Give it a descriptive name
  - Reuse the classes and/or methods

35 Cohesion (Page 3)

- Cohesion of classes:
  - Classes should represent one single, well defined entity and is *the single location* were all that functionality is managed
  - Cohesion of classes will result in better maintainability, reusability, and reliability

36 Cohesion (Page 4)

- Cohesion of methods:
  - A method should be responsible for one and only one well defined task
  - Easier to understand short cohesive methods rather than longer methods that carry out several tasks …
    - Even if the statements for several tasks could have been coded in a single method
  - Also a method's *name* should clearly state its function

37 Consistency (Page 1)

- Follow Java standards (conventions used by most programmers in industry):
  - Choose informative names for classes, data fields and methods
  - Do not choose different names for similar entities and operations …
    - Use the this reference for data fields in constructors and set methods with matching parameter names
    - This also is true in Java API classes, e.g. the String, StringBuilder and StringBuffer classes all have a length method with identical functionality

38 Consistency (Page 2)

- Follow Java standards (conventions used by most programmers in industry) (*con.*):
  - Place data field declarations before constructors
  - Place constructors before methods
  - Provide a no-argument constructor for defining a default instance of the class
    - Or have a good reason why not and document the reason, e.g. immutable objects

39 Encapsulation (Page 1)

- Encapsulation is achieved by making instance variables private
  - Also called "information hiding"

  - Only *what* a class can do should be visible to the outside, not *how* it does it

40 Encapsulation (Page 2)

- Through a public interface the private data can be used by the client class without *corrupting* that data
  - Only the class' own methods may directly inspect or manipulate its data fields
  - Protects data from the client but still allows the client to access the data
  - Makes the class easier to maintain since the functionality is managed in just one place
  -

41 Encapsulation (Page 3)

- Encapsulation is achieved by:
  - Making data fields (instance and static variables) private, and …
  - Having public *accessor* and *mutator* methods that give access to the data fields (of which the client does not know how they function)

42 Clarity

- Class members should be clear and easy to understand, e.g.:
  - Property values may be assigned in any order
  - Methods should be intuitive, e.g.:
    - Method substring(int beginIndex, int endIndex) is not since the string returned stops at endIndex – 1
  - Do not declare data fields that can be derived from other data fields, e.g.:
    - In the Payee class grossPay is *calculated* from hoursWorked and payRate

43 Completeness

- Classes are used by many clients and should be useful for a wide range of applications
- Provide a wide variety of properties and methods to meet all possible needs
- E.g. the String class has more than 40 methods

44 Instance vs. Static (Page 1)

- A variable or method that is dependent on a specific instance of the class should be an instance variable or method

45 Instance vs. Static (Page 2)

- A variable that is shared (one RAM location) by all instances of a class should be static
  - Static variables usually should be handled by static methods
  - Reference static members with the class name, e.g. JOptionPane.showMessageDialog()
  - Do not pass parameters for static variables to constructors which always are used to create an instance; rather include a static set method

46 The JOptionPane Class (Page 1)

- Class from the Java API library providing simple to use *popup dialogs* to prompt users for a value or to display information
- A member of the javax.swing class:
  - import javax.swing.JOptionPane;

47 The JOptionPane Class (Page 2)

- JOptionPane class can seem complex, but most methods are one-line calls to one of the four (4) static show*Xxx*Dialog methods

- Two of the methods are:
  - showInputDialog—prompts for some input
  - showMessageDialog—a message that tells the user about something that has happened

48 The JOptionPane Class (Page 3)

- These two methods showInputDialog and showMessageDialog are static:
  public static void showMessageDialog(Component *parent*, Object *message*)
- The syntax to call these methods uses the *class name*, not an object name), e.g.
  JOptionPane.showMessageDialog(null, pay1.toString());

49 The JOptionPane Class (Page 4)

- JOptionPane method calls *pause* program execution (blocks the caller until the user's interaction is complete)
- The Java API documentation for the class JOptionPane is located on-line at:
  - http://docs.oracle.com/javase/7/docs/api/javax/swing/JOptionPane.html

50 The showMessageDialog Method (Page 1)

- Displays output in a *message dialog* window
- The showMessageDialog is a method of the predefined JOptionPane class contained in the Java API library
- Alternative to println method which instead allows GUI (graphical user interface) output

51 The showMessageDialog Method (Page 2)

- Takes two required parameters:
  - The first is the keyword null
  - The second is the output *message* (String, etc.)
- Format:
  JOptionPane.showMessageDialog(null,
      message);
- Example:
  JOptionPane.showMessageDialog(null, pay1.toString());

52 The showInputDialog Method (Page 1)

- Accepts a String typed input from users in a textbox within the dialog window
- The showInputDialog is a member of the JOptionPane class
- Alternative to Scanner object which instead allows for GUI input

53 The showInputDialog Method (Page 2)

- The only required argument is a message
  - A *prompt* that tells the user what value should be keyed into the textbox
- The return value of the method is a String that is usually *assigned* to a variable

54 The showInputDialog Method (Page 3)

- Format:
  JOptionPane.showInputDialog(*message*);
- Example:
  String input = JOptionPane.showInputDialog("Enter hours worked");

55 Wrapper Classes (Page 1)

- Primitive types (byte, short, int, long, float, double, boolean and char) are not objects
- Wrapper classes, which allow primitives to be *treated like* objects, exist for every primitive:
  - Byte, Short, Integer, Long, Float, Double, Boolean and Character
- Located in the java.lang package so they do not need to be imported

56 Wrapper Classes (Page 2)

- Instantiated wrapper objects can hold and manipulate primitive values, e.g.:
  WrapperClass object = new *WrapperConstructor*(*primitiveValue*);
- Examples:
  Integer myIntegerObject = new Integer(myInteger);
  Integer myIntegerObject = new Integer(40);

64 Wrapper Classes (Page 4)

- All Java wrapper classes (except Character) have *parse* methods that can convert String format of a number to numeric value:
  Byte.parseByte(*string*)
  Short.parseShort(*string*)
  Integer.parseInt(*string*)
  Long.parseLong(*string*)
  Float.parseFloat(*string*)
  Double.parseDouble(*string*)
  Boolean.parseBoolean(*string*)

65 The Integer.parseInt Method

- A method from wrapper class Integer that converts String values to int type
  - May be necessary when an input method (e.g. showInputDialog()) returns a String
- Format:
  Integer.parseInt(*string*)
- Example:
  int age = Integer.parseInt(stringAge);

66 The Double.parseDouble Method

- A method from wrapper class Double that converts String values to double type
  - May be necessary when an input method returns a String
- Format:
  double.parseDouble(*String*)
- Example:
  double hoursWorked = Double.parseDouble(stringHours);

67 Return Values as Arguments to Another Method (Page 1)

- When the return value (result) of one method will serve as an argument to the next method …
- Rather than storing the return value in a separate variable …
- A common Java programmer practice is to insert the entire the first method call into the argument parentheses of the second method

68 Return Values as Arguments to Another Method (Page 2)

- Instead of:

```
String stringHours =
    JOptionPane.showInputDialog
        ("Enter hours worked");

double hoursWorked = Double.parseDouble(stringHours);
```

- Rather:

```
double hoursWorked = Double.parseDouble(
    JOptionPane.showInputDialog(
        "Enter hours worked") );
```