Thinking in Objects

Inmutable Objects

An object whose contents may not be changed once it is instantiated and initialized in
the constructor
- All data fields are private
- There are no set (mutator) for any data field
- No get (accessor) can return a reference to a data field that is mutable

Scope

The scope of a local variable is the block in which it is declared:
- Instance variables and static class variables have class scope even though declared
  with the access modifier private
- Accessible from any method in the class
- Local variables (including parameters) have method scope
- Accessible only in the method in which declared

Scope

Scope may limited further by subordinate blocks with methods, e.g.
- An if or loop (for or while) block
- Any set of braces within the method

Life Time

The lifetime (or simply life) of a variable is the time of execution (the time is exists)
within the block in which it is declared:
- For a static class variable as soon as the class is loaded into memory
- For an instance variable as soon as the object is instantiated and as long as it is in
  memory
- Within a method only as long as the method still is executing
- Or within lesser blocks as well

The this Reference

Every object has a reference to itself in the keyword this
Reference to instance members (variables and methods) with the prefix this, e.g.
this.hoursWorked
this.getHoursWorked()
- Refers to the private instance variable of the object ...
- Not the local variable within the method

The this Reference

For example (although the convention in this case is to not use the this reference):
public void printPayee()
{
    System.out.println("Hours worked: " + this.hoursWorked );
    System.out.println("Pay rate: " + this.payRate );
    System.out.println("Gross pay: " + this.grossPay() );
}

The this Reference and Parameters
It is common to define *parameter* variable names that are the same as *instance variable* names in set methods

In this case the local parameter name takes precedence over the instance variable name

– The instance variable becomes *hidden*

Prefix this reference to the instance variable name to access to it

### The *this* Reference and Parameters (Page 2)

For example, in the following set method:

```java
public class Payee
{
    private double hoursWorked;
    private double payRate;

    ...

    public setPayRate(double payRate)
    {
        this.payRate = payRate;
    }
}
```

### Class Variables and Parameters (Page 1)

If a *parameter* variable name is the same as a static *class variable* name in a set method or constructor ...

Prefix the class name to the static variable name to access to it, e.g. *ClassName.staticVariable*

### Class Variables and Parameters (Page 2)

For example if payRate is static:

```java
public class Payee
{
    private double hoursWorked;
    private static double payRate;

    ...

    public static payRate(double payRate)
    {
        Payee.payRate = payRate;
    }
}
```

### Using this to Invoke Constructor (Page 1)

Use of this by itself in one constructor refers to another constructor in the class

– A good habit when there is more than one constructor in the class

Must be the first statement in a constructor before any other statement

### Using this to Invoke Constructor (Page 2)

Example:

```java
public class Payee
```
```java
{  
    private double hoursWorked;
    private double payRate;
    public Payee()
    {
        this(0, 0);
    }
    public Payee(double hoursWorked, double payRate)
    {
        this.hoursWorked = hoursWorked;
        this.payRate = payRate;
    }
}
```

26 **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

27 **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

28 **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

29 **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

30 **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
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

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

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

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 */

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

Class-Level Comments (Page 2)

Example class-level comment:

```java
/**
 * The Payee class calculates payroll
 * for regular and overtime workers.
 * Users update data fields by calling
 * the setHoursWorked() and setPayRate() methods.
 */

public class Payee
```
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

Member-Level Comments  (Page 1)

- Member-level comments describe the fields, methods, and constructors
- Placed directly above each method header

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)

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

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)
{

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

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
*/

```java
public String getHoursWorked()
{
    // Class implementation
}
```

---

**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.

---

**Cohesion**

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.

---

**Cohesion**

High cohesion makes it easier to:

- Understand what a class or method does
- Give it a descriptive name
- Reuse the classes and/or methods

---

**Cohesion**

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

---

**Cohesion**

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

---

**Consistency**

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
62 **Consistency**  
Follow Java standards (conventions used by most programmers in industry) \( (\text{con.}) \):  
- 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

63 **Encapsulation**  
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

64 **Encapsulation**  
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  

65 **Encapsulation**  
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)

66 **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

67 **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

68 **Instance vs. Static**  
A variable or method that is dependent on a specific instance of the class should be an instance variable or method

69 **Instance vs. Static**  
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()`
The JOptionPane Class (Page 1)

Class from the Java class library providing simple to use popup dialogs to prompt users for a value or to display information.

JOptionPane class can seem complex, but most methods are one-line calls to one of the four (4) static showXxxDialog methods.

A member of the java.util class:

- import javax.swing.JOptionPane;

The JOptionPane Class (Page 2)

Two of the methods are:

- showInputDialog—prompts for some input
- showMessageDialog—a message that tells the user about something that has happened

The syntax of these static methods uses the class name, not an object name, e.g.

input = JOptionPane.showMessageDialog( "Enter hours worked" );

The JOptionPane Class (Page 3)

The JOptionPane showInputDialog() method is static:

public static void showMessageDialog( Component parent, Object message )

That is why objects need not be instantiated from JOptionPane before its static methods are called, e.g.

JOptionPane.showMessageDialog( null, pay1.toString() );

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

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.

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() );

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

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

The showInputDialog Method (Page 3)
- Format:
  JOptionPane.showMessageDialog( message );
- Example:
  input = JOptionPane.showInputDialog( "Enter hours worked" );

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

Wrapper Classes (Page 2)
- Instantiated wrapper objects can hold and manipulate primitive values, e.g.:
  WrapperClass object = new WrapperConstructor(primitiveValue);
- Example:
  Integer myIntegerObject = new Integer(myInteger);

Wrapper Classes (Page 3)
- Wrapper classes do not have no-argument constructors since they are immutable
  - Once instantiated their values are unchangeable
- Each wrapper class includes a conversion method, e.g. doubleValue(), floatValue(), intValue(), longValue(), shortValue()
- Each wrapper class includes compareTo() and valueOf() methods

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)

The Integer.parseInt Method
- A method from wrapper class Integer that converts String values to int data type
  - Can be necessary when an input method (e.g. showInputDialog()) returns a String
- Format:
Integer.parseInt(string)

Example:
int age = Integer.parseInt(stringAge);

90 The Double.parseDouble Method

A method from wrapper class Double that converts String values to double data type
– Can be necessary when an input method returns a String

Format:
double.parseDouble(String)

Example:
double hoursWorked = Double.parseDouble(stringHours);

91 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

92 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") );

97 Autoboxing (Page 1)

Java provides for automatic conversion between primitive and wrapper class types:
– Conversion from primitive to wrapper is called boxing
– Conversion from wrapper to primitive is called unboxing

98 Autobooking (Page 2)

Declaring a wrapper object:
   Integer integerObject = new Integer(20);
Is equivalent to (boxing):
   Integer integerObject = 20;
Example of unboxing:
   System.out.println(integerObject);

99 The BigInteger Class

Used to declare and represent an integer of unlimited size and precision
Example:
   BigInteger x = new BigInteger("9394789256343923");
   BigInteger y = new BigInteger("34826493");
BigInteger z = x.multiply(y);

100 **The BigDecimal Class**

Used to declare and represent an integer of unlimited size and precision

Example:

```java
BigDecimal x = new BigDecimal(2.3);
BigDecimal y = new BigDecimal(0.0000000000000004);
BigDecimal z = x.multiply(y);
```