COP 2250 – Java Programming – Chapter 10 – Object-Oriented Thinking

As Liang states on page 366, this chapter is about the difference between procedural and object-oriented programming, with a focus on objects and class design.

Class Abstraction and Encapsulation

* Abstraction refers to the distinction between using a class and its implementation.
* Put another way, one can use a class without knowing the details of how it was coded.
* The implementation can be hidden from users, or encapsulated in a “black box”.
* Why would this encapsulation be desirable?

Examine class Loan on page 368-369. Note the UML diagram, too. Then try TestLoanClass

Thinking in Objects

* Read section 10.3 on page 370 to learn the benefits of thinking in objects.

Examine class BMI on page 371-372. Note the UML diagram, too. Then try UseBMIClass

Class Relationships

* Understanding the principles of relationship is useful when designing an OOP program.
* The key relationships are association, aggregation, composition, and inheritance (studied later in this course)

The Association Relationship

* This is a binary relationship because it describes activity between two classes.
* Association is defined as a structural relationship that conceptually means that the two components are linked to each other.
* This kind of relation is also referred to as a using relationship, where one class instance uses the other class instance or vice-versa, or both may be using each other.
* The lifetime of instances of the two classes are independent of each other and there is no ownership between two classes.
* See page 373 for a discussion and UML diagram for association.
* Refer to Figure 10.5 to see how association is implemented in class fields and methods.

Aggregation and Composition

* Aggregation is like association but with an additional point that there is an ownership.
* One object can “own” one or more other objects. This is called aggregation.
* The owner is the aggregating class. The class owned is the aggregated class or object.
* Aggregation is characterized by the “**has a**” relationship.
* If the aggregating class has exclusive ownership of the aggregated class, the relationship is said to be composition. A Car object “has a” Motor object, for example.
* Figure 10.6 shows a UML diagram used for these relationships.
* When we study inheritance in Chapter 11 we will also encounter the “**is a**” relationship.

Examine class Course on page 377. Note the UML diagram, too. Then try TestCourse

Designing a Class for Stacks

* See the Case Study on page 378.

Examine class StackOfIntegers on page 379 and the UML diagram on page 378, too. Then try TestStackOfIntegers

Processing Primitive Data Type Values as Objects

* Primitives like int, double, char, etc. are not object types, but each has its corresponding **wrapper** type in the Java API.
* Note the UML diagrams for the Integer and Double wrapper classes on page 381.
* Note the methods on page 381-382
* The static **valueOf()** method in all wrappers creates a wrapper instance from a String.
* The “parse” methods can parse numbers into **different number systems** with a specified **radix**.

Automatic Conversion Between Primitive and Wrapper Types

* This is known as boxing and unboxing (page 383-384).
* In **autoboxing**, a wrapper object is created with simplified syntax (top page of 384).
* Wrapper data can be used with the same syntax as with primitives in **autounboxing**.

BigInteger and BigDecimal

* The **java.math** package has these classes for extra-large and very precise numbers.
* There is no upper limit for the size of a BigInteger instance and no limit to the precision of a BigDecimal instance.

Try LargeFactorial

The String Class

* Note the three ways to create a String object on page 386.

String Class Methods

* Study the UML diagrams and examples for these String method groupings:
  + Replacing and Splitting Strings: see page 387-388.
  + Matching, Replacing, and Splitting with Patterns: see page 388-389.
  + Conversion between Strings and Arrays: see page 389-390.
  + Formatting Strings: see page 390.

StringBuilder and StringBuffer Classes

* These classes were developed to get around the issue that a String is immutable.
* Objects of these classes can be altered by editing and appending.
* The two classes are essentially the same except that StringBuffer methods are synchronized and therefore thread-safe.
* Note the UML diagram and examples for StringBuilder on pages 393-394.
* Note also the additional StringBuilder methods in the UML diagram on page 395.

Palindromes Case Study (see page 396)

* Note the steps needed to check a palindrome by ignoring alphanumeric characters.

Try PalindromeIgnoreNonAlphanumeric