Chapters 10: Thinking in Objects

Based on *Introduction to Java Programming*, Y. Daniel Liang, Brief Version, 9/E

**Topics Covered**

1. Immutable Classes and Objects 2
2. Scope of Variables (the class perspective) 2
3. this Reference 2
4. Class Abstraction and Encapsulation 3
5. Object Composition 3
6. Designing a Stack Class 3
7. Class Design Guidance 4
8. Wrapper Classes for Primitive Data Types 5
9. `BigInteger` and `BigDecimal` classes 6
1 Immutable Classes and Objects

The content of an immutable object cannot change.

Requirements for an immutable class:

- all data fields are private,
- no setters (mutators) that allow modification of the data fields,
- no getter (accessors) that return a reference to a data field.

See Supplement III AB (on author’s website) for more.

2 Scope of Variables (the class perspective)

- Data fields, both static and instance, are class’s variables, i.e., their scope is the entire class (independent of their access specifiers). Data fields have class scope.
- Variables declared and defined inside methods are local variables (their scope is local) and can be accessed only from within the method in which they were declared.
- Data fields and methods can be declared in arbitrary order in the class definition. It is often a good practice to list all the data fields together before the methods.
- Local variables whose name is the same as class’s variables hide the class’s variable within that method. Hidden variables are variables with class scope that are hidden by local variables.

3 this Reference

this refers to the object itself. Within the class definition you can replace any mention of a data field or a method by preceding it with this.

Why do we need a reference to itself?

- accessing hidden data fields,
- invoking constructor from another constructor.
4 Class Abstraction and Encapsulation

Class abstraction separates class implementation from how the class is used.

Class’s contract is the collection of methods and fields that are accessible from outside of the class, together with the description of how these members are expected to behave. (The user of the class does not need to know the details of how the methods are implemented.)

Class encapsulation means that the class implementation is (and should be) hidden from the user of the class.

Abstract data type (ADT) is another name for the class for which the implementaiton is hidden.

5 Object Composition

An object can contain another object. This relationship is called has-a relationship.

Example:

Student object has-a:

• name, which is a String object
• date of birth, which is a Date object
• address, which is an Address object
• ...

The "owner" object is called aggregating object (it’s class is aggregating class). The "subject" object is called aggregated object (it’s class is aggregated class).

6 Designing a Stack Class

A stack is a data structure that stores the data in the last-in, first-out (LIFO) fashion. New items can be added to the top of the stack. Items can be removed only from the top of the stack.

StackOfCharacters class contract

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StackOfCharacters()</td>
<td>Constructs an empty stack of characters with a default capacity of 16.</td>
</tr>
<tr>
<td>StackOfCharacters(capacity: int)</td>
<td>Constructs an empty stack of characters with a specified capacity (if provided capacity is less than or equal to zero, the capacity is set to default 16).</td>
</tr>
<tr>
<td>empty(): boolean</td>
<td>Returns true if the stack is empty, otherwise it returns false.</td>
</tr>
</tbody>
</table>
### Class Design Guidance

**Cohesion**
- The class should describe a single thing.

**Consistency with Java programming style and conventions**
- Place data fields before constructors and constructors before the other methods.
- Provide default (no-arg) constructors.
- Use standard methods’ and fields’ names, for example `length`, `compareTo`, `toString`.
- Implement `toString` method.
- Implement `compareTo` method.

**Encapsulation = Hiding implementation details**
- Make all data fields private.
- Provide accessors (get methods) for fields that should be readable.
- Provide mutators (set methods) for fields that should be writable.
- Make helper methods private.
- Helper methods are the methods that should not be called from outside of the class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSize()</td>
<td>Returns the number of elements in the stack.</td>
</tr>
<tr>
<td>getCapacity()</td>
<td>Returns the capacity of the stack.</td>
</tr>
<tr>
<td>push( value: Character ): void</td>
<td>Puts the character <code>value</code> at the top of the stack.</td>
</tr>
<tr>
<td>pop()</td>
<td>Removes the character from the top of the stack and returns it. If stack is empty, returns null.</td>
</tr>
<tr>
<td>peek()</td>
<td>Returns the character from the top of the stack (without removing it). If stack is empty, returns null.</td>
</tr>
</tbody>
</table>

See `StackOfCharacters.java` for implementation.
Clarity

- Provide easy to explain contract: methods should implement simple tasks.
- Methods should be independent, i.e., calling one method should not fail because another method has not been called first. Provide graceful way of quitting the method rather than letting it crash the program.
- Use intuitive meaning of names.
- Use independent data fields: Do not keep multiple data fields that can be derived one from another. Having data fields that are not independent implies that they all need to be modified when one of them changes. Exception to the rule: when computing the value is very costly.

Completeness

- Provide full and general functionality for the class (not only things you need in your next assignment). Think of all the different ways in which the class can be used.

8 Wrapper Classes for Primitive Data Types

Java provides class wrappers for all primitive data types:

- Character
- Byte
- Short
- Integer
- Long
- Float
- Double
- Boolean

You should familiarize yourself with the methods provided in these classes. They may come very handy in many situations.

A primitive type value can be automatically converted to an object using a wrapper class, and vice versa, depending on the context.

Example:
Integer intObject = new Integer(5);

is the equivalent to

Integer intObject = 5;

You can also write

Integer intObject = 5;
int primitiveInt = intObject;

Converting a primitive value to a wrapper object is called **boxing**. The reverse conversion is called **unboxing**.

9 **BigInteger** and **BigDecimal** **classes**

see LargeFactorial.java