More on Objects and Strings

Based on Introduction to Java Programming, Y. Daniel Liang, Brief Version, 10/E

1 Scope of Variables in Classes

- 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.

See ScopeAndVisibilityWithClasses.java

2 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.

Example: A constructor that takes parameters:

```java
class Car {
    private String color;
    private int year;
    private String make;
    ...

    public Car ( String color, int year, String make ) {
        this.color = color;
        this.year = year;
        this.make = make;
    }
    ...
}
```
The parameters to this constructor hide the data fields (because they have the same names as the data fields). We can use this reference to access the data fields of the object that is created by the constructor call.

Some programmers always use this reference with the data fields to distinguish them from variables that are local to the methods.

Example: Making a call from one constructor to another constructor:

```java
class Car {
    private String color;
    private int year;
    private String make;
    ...
    public Car ( String color, int year, String make ) {
        this.color = color;
        this.year = year;
        this.make = make;
    }
    public Car ( ) {
        this( "red", 2014, "Mazda" );
    }
    ...
}
```

The default constructor can pass "default" parameters to the other constructor instead of repeating the code of the other constructor. But the constructors cannot be called by name, instead we use the this reference with a parameter list to call another constructor.

**WARNING:** only a constructor can invoke another constructor using the this reference!

**WARNING:** when using this() to call another constructor from a constructor is must be the first thing the calling constructor does (i.e. the first line of code!)

See This.java

### 3 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.

### 4 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**). See Composition.java

### 5 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

- **StackOfCharacters()** Constructs an empty stack of characters with a default capacity of 16.
- **StackOfCharacters(capacity: int)** 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).
- **empty(): boolean** Returns true if the stack is empty, otherwise it returns false.
- **getSize(): int** Returns the number of elements in the stack.
- **getCapacity(): int** Returns the capacity of the stack.
- **push( value: Character ): void** Puts the character value at the top of the stack.
- **pop(): Character** Removes the character from the top of the stack and returns it. If stack is empty, returns null.
- **peek(): Character** Returns the character from the top of the stack (without removing it). If stack is empty, returns null.

See StackOfCharacters.java for implementation.

See TestingStack.java for an example how you should test your own classes.

### 6 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** :
### 7 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.

#### 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 The String class

For documentation see: [http://docs.oracle.com/javase/7/docs/api/java/lang/String.html](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html).

There are many different ways of constructing a String object, i.e., there are many constructors. For example:

```java
String s1 = "Hello world!";
String s2 = new String ("Hello NY!");
String s3 = new String ( s2 );
char [] letters = {'H', 'e', 'l', 'l', 'o', ' ', 'N', 'Y', 'U', '!'};
String s4 = new String (letters);
```

8.1 Immutable Objects

What happens when we execute the following lines of code?

1) String s1 = "I like C++";
2) String s2 = s1;
3) System.out.println(s1 + "\n" + s2);
4) s1 = "I like Java"
5) System.out.println(s1 + "\n" + s2);

String objects are examples of immutable objects. Immutable objects cannot be modified once they are created. See ImmutableStrings.java

8.2 String comparison

Comparison methods:

```java
equals( ... )
compareTo( ... )
equalsIgnoreCase ( ... )
compareToIgnoreCase ( ... )
```

8.3 Accessing parts of a string object and combining strings

Methods worth knowing:

```java
length ( ... )
charAt ( ... )
concat ( ... )
toLowerCase ( ... )
toUpperCase ( ... )
split ( ... )
replace ( ... )
```

See Strings.java