kcr2118
COMSW1007 HOMEWORK 2

Part I - Theory

1. A total ordering of "Person"s would have to be asymmetric and transitive: this means that we should order Person by an attribute which will always be placed where it is. For example, we could order Person alphabetically, using their last name. In this way "Friedrich" will always go before "Gonzalez" will always come before "Birva." In cases where a last name begins with the same letter as another, we will look at the next letter to find which comes first, and so on. In case the last name is the same, we will then move onto the Person's first name. Were two Persons to have the same name, the list will be ordered by SS #, with the one beginning with the smaller number first.
Ex.
"Pollini, Maurizio" > "Hernandez, Maria" > "Castor, Pollux" 324 > "Castor, Pollux" 254

2.
<u>Array List</u>
- Cohesion - ArrayList is very cohesive: it represents the concept of a list well, and has all the intuitive methods associated with it.
- Completeness - the ArrayList seems to be complete, for the majority of its purposes. It does implement other interfaces, like List and Collection, so it is not entirely complete on its own.
- Convenience - the most convenient method of ArrayList is its ability to expand and accommodate more elements. It is also easier to add, and remove elements, making it more accessible than Arrays.
- Clarity - the names and methods of ArrayList are clear, for the most part. The only thing that may surprise the user is its synchronization: ArrayList requires external synchronization, the encapsulation of the ArrayList object when adding, deleting or resizing takes place, in order to ensure the same copy is kept across classes.
- Consistency - the "add" method is either return void or return type boolean, as is the "remove" method. Other than that, everything seems consistent.

<u>Scanner</u>
- Cohesion - Scanner acts exactly as a reader of files should.
- Completeness - all the methods the user needs are readily available using Scanner.
- Convenience - it is very convenient to parse a file using Scanner, and not have to worry about closing it. Also, the user can have different Scanners at a time, which allow for interaction with files and the user.
- Clarity - although Scanner is very thorough in the types it will parse and recognize, the abundance of methods may confuse the user. For example, "next" and "nextLine" are distinctly different, but do not indicate so in their names.
- Consistency - the "has" and "next" methods consistently return booleans and types, respectively, and are intuitive for the user.

Date

- Cohesion - Date is now just used as a very long type, which represents dates in a numerical fashion without the added functionality of a date *class.*
- Completeness - Since most constructors and vital methods (like "setDate," "setHour," and "getDate" were deprecated, Date is not very complete.
- Convenience - the Date class was not convenient for different time zones, changing to daylight savings', and accounting for the fact that our measurement of years is not exactly at 365 days.
- Clarity - the Date class often confused users in their indexing of months, which started at 0.
- Consistency - the methods of the Date class seemed to be fairly consistent and self-explanatory.

3.

The author gives a definition of abstraction applicable to the use of interfaces: "a simplification of something much more complicated that is going on under the covers." Interfaces are meant to do just that, albeit without elaborating exactly on how to do so. An interface will include the methods necessary for the representation of an object, but the implementation and specialization of the methods is only done when a class extends an interface. An interface may simplify an object or concept to its bare minimum, but it only goes so far: the programmer must still device a proper way to make the methods work. This relates to the author's final point: at the end of the day, we have to look twice at what appears to make our lives easier. An interface must still be understood abstractly in order for us to be able to use it. The programmer must know to override the methods in the interface, must be able to recognize when an interface is "abstract" and has overridden its methods already, and must keep in mind that the visibility and parameters of an interface's methods must remain as they are given.

The Java API also asks us to take certain things for granted: for example, the user of Serializable has no idea how the interface really works, and in some cases, would not be able to solve any problems accordingly. Object-oriented programming itself is a form of abstraction, for we do not necessarily have to code a complex “automobile” class when we have made the wheel, tired and velocities work together to create a whole.

The author eventually concludes that abstraction is dragging us down because when it is obvious that the simplification cannot hold forever, we must deal with the patching up of the slip. This makes for a rather pessimistic outlook on how abstraction is able to achieve its purpose: it is not always the case that an abstraction leaks on the job. Sometimes, and probably, I dare say, most of the time, an abstraction serves its function well, otherwise we would not continue using them. Perhaps the only thing that is necessary to change about how we use abstraction in the future is to be more transparent. It seems that the point of contention is the nasty surprise a programmer will get when she finds that she does not understand an error because it is based on the abstraction of a more complex topic. In that case, it may serve us better to come fully prepared to address these situations.