Problems On Classes in Java

In this problem set, we will present a variety of simple classes. In these examples, the primary purpose of the class is to collect data elements that are related to one another in a single object. In such examples, the behavior provided *beyond set and get functions* is quite limited. We will call such classes *data buckets* because their primary purpose is to collect data.

Problem 1: Person

In this problem, we will explore the building a `Person` class that collects information about a person. Such information might be the starting point for an online address-telephone list or might be a component in a database system.

Let us first explore the data in the `Person` class. A simple approach is to have one field, `name`, that contains all parts of a person's name: first name, middle name, and family name. However, this design implies that when you need an individual part of the name then you must somehow extract that part from `name`. A better design is to have three fields, `firstName`, `middleName`, and `familyName`, to hold the information as separate data from the start.

The next question is whether the data should be `public` or `protected`. It may be argued that since the `Person` class is a data bucket why not make access as simple as possible by making the data fields `public`? This design might be represented by:

```
public class Person {
    public String familyName = "";
    public String firstName = "";
    public String middleName = "";
}
```

Since this class does not have an explicit constructor, Java will automatically supply a default constructor that leaves the member data fields with the initializations given above. Thus, if we now introduce a `Person` object

```
Person X = new Person();
```

then we may directly access the three fields as `X.familyName`, `X.firstName`, and `X.middleName` and these fields initially contain the empty `String`: `""`. Initializing with the empty `String` is much more convenient than initializing with `null` since making tests for `null` is a pain. This observation reveals, however, the fatal flaw in permitting the fields to be `public`. If the fields are `public`, there is nothing to prevent another programmer from writing code that says, for example:

```
X.familyName = null;
```

Thus, even if we intend the class to maintain its data as non-**null** **String** objects, if the data is **public** we have no control over what another programmer will do.

This means that it is a better design to make the data **protected** and to provide access via *set* and *get* functions. Let us illustrate this with the **familyName** field:

```
protected String familyName = "";

// later on in the class definition ...

public void setFamilyName(String name) {
    if (name == null)
        familyName = "";
    else
        familyName = name;
}

public String getFamilyName() { return familyName; }
```

This coding means that **setFamilyName** prevents the assignment of a **null** value to the internal **familyName** data field of a **Person** object.

The next question we can ask is whether or not we should have additional fields in the **Person** class. Several possibilities come to mind:

(1) A field, **title**, to hold *honorific information* such as **Mr.**, **Ms.**, **Miss**, **Mrs.**, **Dr.**, **Prof.**, etc.

(2) A field, **extra**, to hold extra tags such as **Sr.**, **Jr.**, **III**, etc.

(3) A field, **nickName**, to hold a person's nickname.

(4) A field, **identifier**, to hold some identifier data to distinguish people with the same name information.

We explicitly *do not* wish to attach address information since that will be dealt with using separate classes below.

We leave it to you to decide exactly how you wish to build the data in the class, that is, what fields to include and what fields to exclude. Let us now consider the question of *constructors*. Usually, it is a good idea to have a *default constructor* so that an object can be constructed with no immediate information thus leaving modification of its data until later. It is also a good idea to have a *maximal constructor* that permits *all data fields to be set at the time of construction*. What about constructors that fall in between these extremes? We recommend at least:

```
public Person(String familyName) { ... }

public Person(String familyName, String firstName) { ... }

public Person(String familyName, String firstName, String middleName) { ... }
```

Under no circumstances should any of the constructors do direct assignment to the data fields. The appropriate *set* functions should be called to avoid **null** assignments.

Finally, we come to member functions beyond *constructors* and *set* and *get* functions. For such a simple class built with internal **String** data, the only obvious functions are those that combine the **String**'s in various ways. Here are some that come to mind.

The Java **toString()** function should be implemented to *combine all of the internal data* into one big **String**. For this, we recommend traditional *comma-separated-format*:

```
familyName, firstName, middleName, title, extra, nickName, identifier
```

Of course, what you combine depends on your choice of data fields. The definition of the **toString()** function should look like:

```
public String toString() {
    return familyName + ", " + firstName + ", " + ... ;
}
```

It will be useful to have member functions that combine only parts of the data into larger String objects. For example:

```
firstName familyName
firstName middleName familyName extra
title firstName familyName
title firstName middleName familyName extra
```

Exactly how you name these functions is up to you. One particular problem, however, deserves a general solution, namely, *what to do about the blank spacing between* **String**'s *if one of the two* **String**'s *is empty*. In that case, we want to omit the blank as well. You will go crazy if you try to implement this requirement inline in each case. Instead, we recommend the definition of a **static** function that handles this issue once and for all:

```
/**
 * Join will return a blank separated String by combining the two given input
 * String objects if both are non-null and non-empty.
 *
 * Otherwise, join will return one of the two input String's as appropriate.
 */
public static String join(String s, String t) {

    if ((s == null) || (s.length() == 0))
        return t;

    if ((t == null) || (t.length() == 0))
        return s;

    return s + " " + t;
}
```

Notice that this **static** function does test for **null**. The reason is that you might want to use it in other programs and might even want to place it in a Java library. In such situations, you want the function definition to be as robust as possible.

Problem 2: StreetAddress

Define a Java class **`StreetAddress`** that combines the information needed to specify a street postal location, that is, a street number, a street, and an apartment or office number within the building at that location. Define member data, constructors, set and get functions, and **`String`** output functions as appropriate following the model of the **`Person`** class.

Problem 3: Town

Define a Java class **`Town`** that combines the information needed to specify a town postal location, that is, a town, state or province, country, and postal equivalent of the US zipcode. Define member data, constructors, set and get functions, and **`String`** output functions as appropriate following the model of the **`Person`** class.

Problem 4: FullAddress

Define a Java class **`FullAddress`** that combines one object of class **`StreetAddress`** and one object of class **`Town`** to define the full information required to specify a complete address. Define member data, constructors, set and get functions, and **`String`** output functions as appropriate following the model of the **`Person`** class.

Problem 5: TelephoneNumber

Define a Java class **`TelephoneNumber`** that combines the information needed to specify a telephone number. Be concerned about differences in telephone numbers for different countries and decide how you will handle this in the member data definitions and set functions. What should be the defaults for the member data? Should this class contain information about whether the telephone number will connect to a person via wires, to a person via a cell phone, to a fax machine, or to a computer modem bank or is it better not to specify such information? Define constructors, set and get functions, and **`String`** output functions as appropriate following the model of the **`Person`** class.

Problem 6: PersonInfo

Define a Java class **`PersonInfo`** that combines the information needed to specify a person's name, *zero or more* full addresses at which the person may live or work, and *zero or more* telephone numbers. Define member data, constructors, set and get functions, and **`String`** output functions as appropriate following the model of the **`Person`** class.

Note: This final problem is significantly harder than the previous problems because of the requirement *zero or more* for the full addresses and telephone numbers. If you wish, you may simplify the problem to require *exactly one* full address and telephone number.