CS100J Spring 2005 Assignment A2
Due (submitted on the CMS) on Thursday, 17 February

Introduction

Read the WHOLE handout before you begin to write the assignment.

CBC is planning on launching a gritty new drama series, called *The Greases*, kind of like *As the World Turns*. The producers of this show have a problem, however: the Grease family is so large that they have trouble keeping track of who is related to whom, what the relation is between any two characters, and which couples can't get married because they're cousins.

That's where you come in, as the CBC's resident computer expert. They've asked you to devise a system that will keep track of all the family members in this show and allow them to add characters and keep track of characters as the show progresses. Make sure that the Java class you implement to store information on Grease family members is a good one --or they may not renew your job for next season.

Requirements

For this assignment, you are required to design two classes. Class `FamilyMember` is the real one. It has lots of fields and methods, but each method is simple. If you do one thing at a time, not worrying about the overall picture, you should have little trouble.

Class `FamilyMemberTester` is used to test class `FamilyMember`. We will describe how to do this on 8-10 February. Do not think too much about it when first reading this handout.

Class FamilyMember

An instance of class `FamilyMember` represents a single member of a family (in this case, one of the Grease family members). It has several fields that one might use to describe a relative, as well as methods that operate on these fields.

FamilyMember Variables

Class `FamilyMember` will have the following 11 fields:

- first name (a `String`)
- last name (a `String`)
- age in years (an `int`)
- gender (a `String`)
- day of birth (an `int`)
- month of birth (an `int`)
- criminal record (a `boolean`)
- father (a `FamilyMember` object)
- mother (a `FamilyMember` object)
- number of children (an `int`)
- family size (a static `int`)

Here are a few specifications about the fields of the `FamilyMember` class:

- The gender field is one of two `String` values: `"male"` or `"female"`.
- The day of birth is in the range 1..31, representing the day within the month. The month of birth is in the range 1..12, representing a month from January to December. Do not worry about invalid dates like 31 February; do **not** write code that checks whether dates are valid: assume they are valid. *Error-checking may be penalized.*
- The criminal-record field contains `true` if the family member has a criminal record and `false` otherwise.

- The father and mother variables are references to the `FamilyMember` objects that correspond to this person's parents. They are `null` if not known.
- The family-size variable contains the number of family members that have been recorded (created) so far. They do not have to be directly related to one another, since non-relatives are sometimes included in this type of 'family'. Whenever a new `FamilyMember` object is created, this value should increase by 1.

FamilyMember Methods

Class `FamilyMember` supports the following methods. Pay close attention to the parameters and return values of each method. The descriptions, while informal, are complete.

Constructor	Description
`FamilyMember(String first, String last, String gender, int day, int month, int age)`	Constructor: a new `FamilyMember`. Parameters are, in order, the first name and last name of the person, their gender, the day and month of their birthday, and their age. The new person is not a criminal.
`FamilyMember(String first, String last, String gender, FamilyMember father, FamilyMember mother, int day, int month, int age)`	Constructor: a new `FamilyMember`. Parameters are, in order, the first name and last name of the person, their gender, their father and mother (not `null`), the day and month of their birthday, and their age. The new person is not a criminal.
`FamilyMember(String first, String last, String gender, FamilyMember father, FamilyMember mother)`	Constructor: a new `FamilyMember`. Parameters are (in order) the first name and last name of the person, their gender (`"male"` or `"female"`), their father and mother (not `null`). The new person was born on 1 January, is 0 years old, and is not a criminal.

Method	Description
`getFirstName()`	= the first name of this family member (a `String`)
`getLastName()`	= the last name of this family member (a `String`)
`getName()`	= the full name of this family member (a `String`). The format of the name is: the first name, a blank char., and finally the last name. E.g.: "John Mac" is correct; "Mac, John" iswrong.
`getAge()`	= the age of this family member (an `int`)
`getGender()`	= the gender of this family member (a `String`).
`getDOB()`	= the day of the month this family member was born (an `int`).
`getMOB()`	= the integer that represents the family member's month of birth (an `int`).
`getFather()`	= (the name of the object representing) the father of this family member (a `FamilyMember`).
`getMother()`	= (the name of the object representing) the mother of this family member (a `FamilyMember`).
`getNumberChildren()`	= the number of children of this person (an `int`).
`getFamilySize()`	Static method. = the number of people in the entire family --the number of `FamilyMember` objects created thus far (an `int`).
`isCriminal()`	= "this family member is a criminal" (a `boolean`)
`setFirstName(String n)`	Set the first name of this family member to `n`.
`setLastName(String n)`	Set the last name of this family member to `n`.
`setAge(int age)`	Set this family member's age to `age`.
`setGender(String g)`	Set the gender of the family member to `g` (one of `"male"` and `"female"`).
`setDOB(int i)`	Set the day of the month this family member was born to `i`.
`setMOB(int i)`	Set the month of birth for this family member to `i`.
`setGuilt(boolean b)`	Set whether this family member is guilty of a crime to `b` (`true` or `false`).
`setFather(FamilyMember fm)`	Set this person's father to `fm`. Precondition: This person's father is `null`, `fm` is not `null`, and `fm` is male.
`setMother(FamilyMember fm)`	Set this person's mother to `fm`. Precondition: This person's mother is `null`, `fm` is not `null`, and `fm` is female.

`isOlder(FamilyMember fm)`	= "this family member is older than fm" (a `boolean`). Use only the age field in making this comparison, not the day and month fields. Precondition: `fm` is not null.
`areSameAge(FamilyMember fm1, FamilyMember fm2)`	Static function. = "`fm1` and `fm2` are not null and are the same age" (a `boolean`). Use only the age field in making this comparison, not the day/month fields.
`areRelated(FamilyMember fm1, FamilyMember fm2)`	Static function. = "`fm1` and `fm2` are not null and have the same last name" (a `boolean`). Capitalization is not important --e.g. "Grease" and "grease" are the same last name.
`isBrother(FamilyMember fm)`	= "`fm` is this family member's brother" (a `boolean`). Note: a guy is called your brother if you and he (has to be a "he") are different and have at least one parent in common. Precondition: `fm` is not `null`.
`isSister(FamilyMember fm)`	= "`fm` is this family member's sister" (a `boolean`). Note: a gal is called your sister if you and she (has to be a "she") are different and have at least one parent in common. Precondition: `fm` is not `null`.
`areSiblings(FamilyMember fm1, FamilyMember fm2)`	Static method. = "`fm1` and `fm2` are not null and `fm1` and `fm2` are siblings (brothers or sisters)" (a `boolean`).
`isMotherOf(FamilyMember fm)`	= "this family member is `fm`'s mother" (a `boolean`). Precondition: `fm` is not `null`.
`isFatherOf(FamilyMember fm)`	= "this family member is `fm`'s father" (a `boolean`). Precondition: `fm` is not `null`.
`isParentOf(FamilyMember fm)`	= "this family member is `fm`'s parent" (a `boolean`). Precondition: `fm` is not `null`.
`areTwins(FamilyMember fm1, FamilyMember fm2)`	Static method. = "`fm1` and `fm2` are not null and `fm1` and `fm2` are siblings and have the same age, birth day, and birth month" (a `boolean`).
`isEvilTwin(FamilyMember fm)`	= "this family member is an evil twin of `fm` --'evil' means having a criminal record. Precondition: `fm1` is not `null`.

Make sure that the names of your methods match those listed above **exactly**, including capitalization. The number of parameters and their order must also match. The best way to do this is to copy and paste. Our testing will be expecting those method name names and parameters, so any mismatch will fail during our testing. Parameter names will not be tested --you can change the parameter names if you want.

Each method **must** be preceded by an appropriate specification, as a Javadoc comment. The best way to ensure this is to copy and paste. Note that a precondition should not be tested by the method; it is the responsibility of the caller to ensure that the precondition is met. As an example, in method `isMotherOf`, the method body should not test whether `fm` is null. However, in function `areSiblings`, the tests for `fm1` and `fm2` not `null` must be made.

The number of children of a newly created person is 0. Whenever person P is made the mother or father of another person, P's number of children should increase by 1.

It is possible for person P1 to be P2's mother, and visa versa, at the same time. We do not check for such strange occurrences.

Your method bodies should have **no if** statements. Your method bodies should contain only assignments and return statements. Points will be deducted if **if** statements are used.

Class FamilyMemberTester

How do you know whether class `FamilyMember` that you are designing is correct? The only way you can be sure is to test it, to see if it does what it is supposed to do. It is not enough simply to try out your class `FamilyMember` in the interactions pane. Every time you write a method for your class `FamilyMember`, you should also write a couple of tests for it. And run your collection of tests frequently to make sure that everything works correctly.

The `FamilyMemberTester` is the JUnit test suite you'll design, which performs these testing tasks for you. Make sure that your test suite adheres to the following principles:

- For each method in your class `FamilyMember`, your test suite should have **at least** one test case that tests that method.
- The more interesting or complex a method is, the more test cases you should have for it. What makes a method 'interesting' or complex can be the number of interesting combinations of inputs that method can have, the number of different results that the method can have when run several times, the different results that can arise when other methods are called before and after this method, and so on.
- Test very basic methods early in your test suite; then move on to more complex ones. (Make sure that the car parts work on their own before you take the whole car for a test drive.)
- Don't try to test too many things in a single test case. Each test case should test only a couple of conditions.

Remember that if you change static variables in an early test, they will retain their values in later tests. Also, the tests are not necessarily run in the order in which you list them in your test suite. So when testing static variables, record their initial value at the beginning of the test, and test that the *change* in the value is what you expect.

Hints and Tips

- We suggest that you proceed as follows.

    - First, declare the fields in class `FamilyMember`, compiling often as you proceed.
    - Second, write the first constructor and all the getter methods of class `FamilyMember`.
    - Third, put a method in class `FamilyMemberTester` to make sure that the first constructor and all the getter methods work.
    - Fourth, for each other constructor in turn, write the constructor and add a method in `FamilyMemberTester` to test it (do one at a time).
    - Fifth, write each of the setter methods and add a method in `FamilyMemberTester` to test them.
    - Sixth, write each of the comparison methods and add a method in `FamilyMemberTester` to test them.

    At each step, make sure all methods are correct before proceeding to the next step. When adding a new method, cut and paste the comment and the header from the assignment handout.

- Submit only files that end with the `.java`. Be careful about this, because in the same place as your `.java` files you may also have files that end with the `.class` suffix, but otherwise have the same name. In particular:

    - Microsoft operating systems hide file extensions by default. You should change this so that the extensions ALWAYS appear! Don't let Microsoft tell you what to do.
    - DrJava creates "backup files", which means that you'll see file names like "`FamilyMember.java~`". The `~` indicates that the file is a backup file, which happens to be an older version of your program.

    Since `.class` files cannot be read by TAs or run with our testing programs, submitting the wrong files creates havoc with grading your assignment. Every year, a half-dozen students submit the wrong file. Don't do it!

- **Do not** use `if` statements when completing this assignment. For boolean expressions the `&&` (AND) and `||` (OR) operators are sufficient to implement all the methods shown above. You will lose points for using `if` statements.
- Some of the `FamilyMember` methods can be implemented easily by using other `FamilyMember` methods that you have already created. Look for these case, and take advantage of them as much as possible.
- Methods `toUpperCase` and `toLowerCase` in class `String` may be useful.
- Remember that a `String` literal is enclosed in double quotation marks and a `char` literal is enclosed in single quotation marks.
- Use method `.equals` to compare objects for equality and `==` to compare primitive values for equality.
- Only object variables can have the value `null`. So comparisons between primitive types and `null` are not legal.
- To create a JUnit test suite, select menu item `File ->?New JUnit Test Case`, and then replace the `testX` method with many methods that test your `FamilyMember` functionality.