Spring, 1994 Name: _____

Com S 541 — Programming Languages 1

Test on Smalltalk and OOP

Special Directions for this Test

This test has 5 questions and pages numbered 1 through 6.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
When you write programs on this test, you may use anything documented in the *Overview and Specification of the built-in types in Little Smalltalk*, TR91-22. You can define methods and classes not specifically asked for if they are useful to your programming.
Make it clear by indentation where methods start and stop.

General Directions from the "Course Specification"

Short answer questions will be graded on the basis of completeness (did you list all the relevant parts of the answer?) soundness (did you make any errors of fact or contradict yourself?) and clarity. You can help ensure clarity by giving examples where possible.
For essays, critical justifications, and the like there may be no one right answer; so the criteria for judgement will be whether your essay is:

- clear (you use examples where appropriate, you write clear sentences, diagrams, and you avoid excess verbosity),
- sound (you start from facts or reasonable assumptions, your logic is convincing), and
- complete (you consider relevant aspects of the problem and alternatives).

The highest grade will be earned by an answer that is, in addition, concise and elegant; that is, answers that say the most with the least amount of excess. Extra consideration will be given to answers that are especially creative. For programming problems, there is usually more than one right answer, hence it is important to present with your program evidence that it is correct. Answers to programming questions will be scored on the following points:

- elegance and clarity (conciseness, using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution, and how clear and convincing your argument is that the code is correct)
- correctness (whether your program solves the given problem correctly in the given programming language)

You should try to eliminate all syntax errors from your programs, as they will affect both the correctness and clarity of your solution; you may receive no points for programs with major syntax errors. Run-time space and time efficiency of programs is not important, unless the problem says so. Also, checking of external inputs, prompting, labeling output, and other conventions for dealing with humans are not important (for this class), unless the problem says so.
The breakdown of points between these categories is roughly as follows.

- 50% for elegance and clarity (using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution),
- 50% for correctness (whether your program solves the given problem correctly in the given subset of the given language)

You are encouraged to use auxiliary subroutines whenever they are helpful.

1. (15 points) In the context to Smalltalk, (a) what is a protocol? How does a protocol differ from the notions of: (b) class, (c) object, and (d) message? (Don't just define these other terms, describe how they compare to the notion of a protocol.) (e) Why are protocols important in Smalltalk and object-oriented programming?

2. (15 points) (a) What does inheritance mean for instance variables and methods in Smalltalk? (b) Give specific examples from your homework to illustrate your description. (c) What is the importance of inheritance in Smalltalk? Include in your answer something about (d) the advantages of inheritance in Smalltalk, as compared to programming languages without it.

3. In this problem you will define a class `BankAccount`. The abstract value of a `BankAccount` object is its balance, which is a number.

   (a) (5 points) What would you type to the Little Smalltalk system prompt to define the class `BankAccount`?

   (b) (5 points) Define the method `new` for this class, so that a `BankAccount` object starts out with a balance of zero (0).

   (c) (15 points) Using the convention that a completely blank line separates methods, write code for the methods specified below. (These methods are specified in the notation of TR #91-22.)

   balance *self: BankAccount* → *m: Number*
   Ensures: *m* is the balance of *self*.

   deposit: *self: BankAccount*, *m: Number*
   Requires: *m* is positive.
   Modifies at most: *self*.
   Effect: make the balance of **post**(*self*) be the balance of **pre**(*self*) plus *m*.

   withdraw: *self: BankAccount*, *m: Number*
   Requires: *m* is positive and *m* is no larger than the balance of *self*.
   Modifies at most: *self*.
   Effect: make the balance of **post**(*self*) be the balance of **pre**(*self*) minus *m*.

   payInterestAtRate: *self: BankAccount*, *r: Number*
   Requires: *r* is between 0 and 1 (inclusive).
   Modifies at most: *self*.
   Effect: make the balance of **post**(*self*) be the balance of **pre**(*self*) plus *r* times the balance of **pre**(*self*).

4. In this problem you will define a class that implements the type `PlusAccount`, which is a subtype of `BankAccount`. The abstract value of a `PlusAccount` object is a pair of numbers: a balance and a checking account balance.

   (a) (5 points) What would you type to the Little Smalltalk system prompt to define the class `PlusAccount`? You should do this in a way that makes it very easy to implement all the methods specified in the prevous problem.

   (b) (5 points) Define the method `new` for this class so that both the balance and the checking account balance start at zero (0).

   (c) (15 points) Using the convention that a completely blank line separates methods, write any code needed to define the methods specified below, as well as those the methods `balance`, `deposit:`, and `withdraw:` as specified for `BankAccount` in the previous problem. (That is, the specification of `payInterestAtRate:` given below, which pays interest on both parts of the `PlusAccount`'s abstract value, should be used instead of the specification given in the previous problem.)

   payInterestAtRate: *self: PlusAccount, r: Number*
   Requires: r is between 0 and 1 (inclusive).
   Modifies at most: *self*.
   Effect: make the balance of **post**(*self*) be the balance of **pre**(*self*) plus r times the balance of **pre**(*self*), and make the checking account balance of **post**(*self*) be the checking account balance of **pre**(*self*) plus r times the checking account balance of **pre**(*self*).

   checkingBalance *self: PlusAccount* → *m: Number*
   Ensures: m is the checking account balance of *self*.

   checkingDeposit: *self: PlusAccount, m: Number*
   Requires: m is positive.
   Modifies at most: *self*.
   Effect: make the checking account balance of **post**(*self*) be the checking account balance of **pre**(*self*) plus m.

   payCheckFor:numbered: *self: PlusAccount, m: Number, seq: Integer*
   Requires: m is positive and m is no larger than the balance of *self* plus the checking account balance of self.
   Modifies at most: *self*.
   Effect: if the checking account balance of **pre**(*self*) is at least as large as m, then make the checking account balance of **post**(*self*) be the checking account balance of **pre**(*self*) minus m. Otherwise (if the checking account balance of **pre**(*self*) is smaller than m), make the checking account balance of **post**(*self*) be 0 and make the balance of **post**(*self*) be the balance of **pre**(*self*) plus the checking account balance of **pre**(*self*) minus m. (Note: it's okay that *seq* isn't used.)

   (There is a blank page following this that you can use if there isn't enough room on this page.)

(This page left (mostly) blank as space to write part of your answer to the previous problem.)

5. In this problem you will define a class that implements the type `HistoryAccount`, which is a subtype of `PlusAccount`. The abstract value of a `HistoryAccount` object is a pair of numbers, a balance and a checking account balance, and a mapping from check numbers to numbers, called the object's history.

   (a) (5 points) What would you type to the Little Smalltalk system prompt to define the class `HistoryAccount`? You should do this in a way that makes it very easy to implement all the methods of `HistoryAccount`.

   (b) (5 points) Define the method `new` for this class, so that a `HistoryAccount` object starts out with a balance of zero (0), a checking account balance of zero (0), and an empty history mapping.

   (c) (10 points) Using the convention that a completely blank line separates methods, write any code needed to define the method specified below, as well as other methods specified for `PlusAccount` in the previous problem (including those from its supertype `BankAccount`). (That is, the specification of `payCheckFor:numbered:` given below, which pays a check and remembers the check number and amount, should be used instead of the specification given in the previous problem.)

   payCheckFor:numbered: *self: HistoryAccount, m: Number, seq: Integer*

   Requires: *m* is positive and *m* is no larger than the balance of *self* plus the checking account balance of self.

   Modifies at most: *self*.

   Effect: if the checking account balance of **pre**(*self*) is at least as large as *m*, then make the checking account balance of **post**(*self*) be the checking account balance of **pre**(*self*) minus *m*. Otherwise (if the checking account balance of **pre**(*self*) is smaller than *m*), make the checking account balance of **post**(*self*) be 0 and make the balance of **post**(*self*) be the balance of **pre**(*self*) plus the checking account balance of **pre**(*self*) minus *m*. IN ADDITION, the history of **post**(*self*) is made to map *seq* to *m*, but is otherwise the same as the history of **pre**(*self*).

   historyAt: *self: HistoryAccount, seq: Integer* $\rightarrow$ *m: Number*

   Requires: *seq* is in the domain of the history of *self*.

   Ensures: *m* is the amount paid for the check numbered *seq*.