Homework must be submitted by mail to cs558acc@cs.pdx.edu. All submitted files (just one for this assignment) should be “bundled” together into a single mail message using the Unix `shar` command. If you use a WinTel mailer, copy your submissions over to the CS unix system first and `shar` and mail from there. It is your responsibility to submit the homework in the proper format.

All programs mentioned can be downloaded from the course web page.

1. File `hw8_1.java` contains an interpreter for an object-oriented, untyped variant of the “toy language” from recent Java-based homeworks. A program now consists of an expression to be evaluated (corresponding to the “main program”) followed by a set of class definitions. Class definitions consist of a class name, a superclass name, and a list of members. A member is either a field or a function. A field has an identifier and an initializing expression. A function has a name, a parameter name, and a body expression. Expressions are much as before, except that: there is no `Letfun` construct (the only functions are class member functions), the format of applications has changed to specify an object expression and a member function name from that object to be applied, there is a `New` constructor expression to create new objects of a specified class, and there is a `Seq` expression form that represents one or more expressions to be evaluated in sequence.

The full AST is shown below. Note that for the first time we permit an arbitrary number of children for certain node types. Since `Program` and `Class` can have only one possible label, we omit it in the concrete representation. An example program is in `hw8_1.input`.

```
Program : Program → Exp Class1 ... Classn

Class (Id) (Id) : Class → Member1 ... Membern

Field (Id) : Member → Exp
Fun (Id) (Id) : Member → Exp

Let (Id) : Exp → Exp Exp
Add : Exp → Exp Exp
Sub : Exp → Exp Exp
Mul : Exp → Exp Exp
Div : Exp → Exp Exp
Num (Int) : Exp → ε
Var (Id) : Exp → ε
App (Id) : Exp → Exp Exp
Ifzero : Exp → Exp Exp Exp
Asgn (Id) : Exp → Exp
New (Id) : Exp → ε
Seq : Exp → Exp1 ... Expn
```

The semantics of the language are similar to those of other class-based object-oriented languages such as Java or Smalltalk. Classes are arranged in a hierarchy, with an (empty) built-in class.
Object at the root. Objects are created (via new) as members of a named class. A newly created object has all the members of that class, plus the members of all its ancestors.

The initial value of each field is specified by an expression in the field definition. Fields can contain integers or other objects. The intention is that fields of a object are only accessible for reading and writing within functions defined within the class of that object or its subclasses. (Actually, they are also accessible from functions defined in its superclasses, but that is really a kind of bug in my implementation; if this were a typed language, such accesses could be prohibited statically.)

All function applications are “virtual” (in C++ terminology); i.e., when a function is applied, its name is searched for first in the receiving class, then in its superclass, and so on. Within any function body, the identifier “this” is always pre-defined to refer to the receiving object of the function call.

As usual, further details of the semantics may be revealed by experimenting with the interpreter and reading its source code!

Your assignment is to extend the interpreter in two ways. Combine your modifications into a single class sol8_1 in file sol8_1.java and make this file the (sole) contents of your submitted bundle.

(a) Add a new expression form

\[ \text{Instanceof (Id) : Exp} \rightarrow \text{Exp} \]

The expression \( \text{Instanceof } c \ e \) should return the integer 0 if expression \( e \) evaluates to an object which was created by \( \text{New } c' \) where \( c' \) is \( c \) or a subclass of \( c \); otherwise it should return a non-zero integer. Using \text{Instanceof} as the test expression of an \text{Ifzero} gives a way to branch according to the runtime class of an object. (This is essentially the same as the \text{instanceof} operator in Java.)

(b) Add a static checking phase to the interpreter, to be executed after parsing but before evaluation. Your checker should detect any class whose objects would have duplicate field names. If it finds such a class, it should terminate execution with the message:

Checking failed: objects of class \( c \) have duplicate field name \( f \)

where \( c \) is a class name and \( f \) is the duplicated field name. If there is more than one such error in the program, you only need to report (any) one of them.

Notes:

(1) Duplicate field names in an object can arise either directly, because the same field is declared twice in the object’s class definition, or indirectly because the same field is declared in the object’s class and in one of its ancestors in the inheritance hierarchy. You must report both kinds of problems.

(2) If objects of class \( c \) have duplicate field names, and class \( d \) is a subclass of \( c \), then objects of class \( d \) will necessarily also have duplicate field names. In such situations, you should report an error for class \( c \) rather than for class \( d \), i.e., always report a duplication as high up in the inheritance hierarchy.
hierarchy as possible. (In implementing this point, remember that classes need not appear in any particular order in the program text.)

(3) Your check must be static, i.e., it must be done before evaluation is attempted. In particular, this means that it must detect and report duplicate field names even for classes that are never instantiated with a New operator.