13

Object-Oriented Programming

Exercises

13.1 Using Java as an example:

- Code reuse: inhertiance, interfaces. In the case of an interface, any class implementing the *Comparable* interface can be sorted or stored in an ordered colection.
- Type safety: in downcasting using an explicit cast, Java always checks the class (type) of an object. Similarly, the `instanceof` operator allows one to check the run time type of an object.
- Abstraction: abstract classes and interfaces allow one to deal with an abstraction. Consider the *Expression* class in the abstract syntax of Clite; it allows one to deal with the concept of an expression without worrying about the kind of expression an object may be.
- Encapsulation: both classes and packages provide encapsulation mechanisms. The visiblity attributes (`protected`, `private`) allow the programmer to hide the representation of an object from the client.

13.2

13.3 When you want to say about classes: "an x is an a and a b and a c," you are defining one class by of combining features from multiple classes. In C++ this act is called *multiple inheritance*; it carries some sticky baggage because each class can have an implementation of the same method.

In Java, you can perform the same act, but since x can only extend one of the three classes a, b, or c, only one implementation can be provided. The other two must be interface classes, which are not allowed to provide implementations of methods. So the C++ problems in combining multiple

classes using multiple inheritance do not occur in Java using multiple interfaces.

13.4 (a) One obvious problem is that arrays are pseudo objects and distinct from other `Collection` classes such as `ArrayList`. `Hashtable` extends `Dictionary`, but no other class does; a `Vector` and a `Hashtable` are both automatically synchronized giving a performance penalty. A *Stack* extends a *Vector* and so as all of a *Vector*s methods, as well as being synchronized.

(b) C++ comes with the *Standard Temple Library*, which contains collections comparable to those of Java. Until Java 5, the major difference was that the C++ collections supported generics or templates and Java did not. This distinction disappeared with the advent of Java 5.

The other major difference is that one can ignore templates in Java since the Java class hierarchy is a tree rooted in the *Object* class. THe C++ class hierarchy is not a tree. Hence, one cannot ignore templates when declaring a collection.

(c) Although there are many excellent collections for Ada (see, for example, the *Public Ada Library*), no standard library of collections is provided with Ada.

13.5 Ada 95, as an extension of Ada 83, is not a true object-oriented language; you can write entirely procedural code using the Ada 83 subset. A class in Ada corresponds to a package containing a main type; Ada is far less object-oriented than Java.

Similarly, Perl is a procedural language to which classes were added. Most of Perl, including the builtin libraries, constitute an imperative scripting language. Objects can be ignored.

Ruby is truly an object-oriented language. Like Smalltalk, everything in Ruby is an object, including control structures.

13.6

13.7

13.8

13.9 See the Clite sources available at the Instructor web site.

13.10 Students should start with the complete Clite sources available at the Instructor web site. Also, see the file `AbstractSyntax.java` in the *ch12code* directory. This file contains the `typeOf`, `V`, and `M` functions for *Expression*s, which can serve as a model for reimplementing the rest of Clite in a more object-oriented style.

13.11 The Clite test program is provided in the software distribution as the text file `ootest.cpp` in the *ch13code* directory. It computes the real roots of a quadratic equation using Newton's method. Compare your results with those of the Clite interpreter.

13.12 Unary addition can be discarded by the parser. Unary minus can be converted to a binary subtraction from zero. The not operator and cast conversions are much more difficult to deal with. There seems to be no real advantage into converting these to binary operators.

13.13 See the files `Concordance.java`, `MakeConcordance.java`, `Report.java`, `Document.java` in the *ch13code* directory. The requested modification is trivial; just report the number of entries in the list for each word as part of the report.

13.14 This is a simple test of `Concordance.java`.

13.15 These are simple experiments with `Concordance.java`.

13.16 See file `Knight.java` in the *ch13code* directory; students also need the file `Queens.java` in the student source code directory for this chapter.

13.17 See [Wirth, 1976, p. 146] for the required attempt method:

```
public void attempt(int level) {
   Enumeration e = b.moves(level);
   while (e.hasMoreElements()) {
      Object move = e.nextElement( );
      if (b.valid(level, move)) {
         b.record(leve, move);
         if (b.done(level))
            b.display( );
         else
            attempt(level+1);
         b.undo(level, move);
      } // if valid
   } // while
} // attempt
```

Additionally, a `display` method was added to the `Backtracker` interface. Also note that this version of `attempt` does not return a `boolean`.

A simple exercise is to convert the enumeration to an iterator. Another is to use generics and a for each loop.

13.18 Using classes that define the abstract syntax of Clite:

(a) `Expression` is a client of `Conditional`.

(b) `Value` is a subclass of `Expression`.

13.19 See distributed software for the `Clite` interpreter.

13.20 The addition of generics complicates the syntax and semantics of the language, but simplifies writing programs in the language. This tradeoff of power vs. convenience is an ongoing one in the design and implementation of programming languages. For example, to write a loop only a `while` is needed, but most languages also provide a `for` and a `do while`.

Templates in Java simplify programming by eliminating the need for most downcasts (since collections can store any form of object) and allow the use of the for all loop introduced in Java 5. Both of these are conveniences but logically unnecessary.

13.21 This is a fairly large undertaking. Essentially, each class in the abstract syntax would need to have both a V (validity) method and an M (meaning) method. The class `Expression` and its subclasses must also implement a `typeOf` method.

The code is simpler because you no longer need a dispatching function for abstract syntax classes and, for example, the meaning functions do not have to be passed as an argument the abstract syntax they are dealing with.

By limiting the scope of the abstract syntax actually used, this can be made into a non-Team Project.

(a) Consider the *Assignment* class, for example:

```
class Assignment extends Statement {
   ...
   public void V(TypeMap tm) { ... }
   public State M(State sigma) { ... }
}
```

Of course, the meaning function of an *Expression* and its subclasses would return a *Value*.

(b) Note that the abstract syntax being evaluated is not being passed as an argument.

(c) A subset of Clite including only assignments, values, variables, and binary expressions can be done by an individual.

13.22 This exercise should be ignored (it properly belongs in Chapter 18 on Correctness).

13.23 This exercise should be ignored (it properly belongs in Chapter 18 on Correctness).

13.24 See the file `Polynomial.py` in the *ch13code* directory.

13.25 See the file `Polynomial.py` in the *ch13code* directory.

13.26 See the file `Fraction.py` in the *ch13code* directory.

13.27 a. A Set in theory is equivalent to a Map, since a map $k \rightarrow v$ can always be converted to a tuple $< k, v >$ and vice veras. In Java, a `HashSet` lacks mnay of the methods of a `HashMap`, so the latter is preferred for this application.

b. `hash_multimap`, `hash_map`, `hash_multiset`, `hash_set`.

c. None.