**#1 Chapter 3 Problem 3.9**

****

**#2 Chapter 3 Problem 3.22**

**(a)** The Scheme function below named multiply takes two arguments and returns a single value which is the product of the two arguments. Scheme is not using currying since the function is returning a single value. It is not breaking the multiply function into a sequence of multiple functions, one for each argument. If it used currying this function would consume one argument and return another function with the second parameter as its argument.

**(define** multiply**(lambda** **(**x y**)**

**(if** **(null?** x**)** **'0**

**(if** **(null?** y**)** **'0**

**(\*** x y**)))))**

**(**multiply 4 4**)**

**=>** 16

**(b)** The make-double function defined below assumes that F is a function with two parameters and creates the function that repeats the parameter x in a call to F. The returned value of make-double is a function value created with this statement : **(lambda (x) (F x x))).** The square function uses this returned function definition and applies it to its parameters. In this way a call to square with the parameter 4 will result in 16. This example was explained on pages 61 and 62 of the textbook.

**(define** make-double**(lambda** **(**F**)**

**(lambda** **(**x**)** **(**F x x**))))**

**(define** square **(**make-double **\*))**

**(**square 4**)**

**=>** 16

**#3 Chapter 3 Problem 3.34**

The function call in the last line of code below uses the two functions defined above it to get a list of even integers in a generator-filter style. Lazy evaluation is used by using the Scheme keywords delay and force.

**(define** **(**getEvens n L**)**

**(if** **(=** n 0**)** **'()**

**(if** **(even?** n**)** **(cons** **(car** **(force** L**))** **(**getEvens **(-** n 1**)(cdr** **(force** L**))**

**(**getEvens **(-** n 1**)** **(cdr** L**)))))**

**(define** **(**intlist m n**)**

**(if** **(>** m n**)** **'()**

**(cons** m **(delay** **(**intlist **(+** 1 m**)** n**))))**

**(**getEvens 10 **(delay** **(**intlist 1 10**)))**

**#4 Chapter 5 Problems 5.20**

Encapsulation is an important concept in OOP because it allows for data and implementation details to be hidden from the user of the classes. This ensures data is not corrupted and methods of the class continue to function as expected. Both Java and C++ use the public, private, and protected modifiers but in slightly different ways. Smalltalk does not use public, private, and protected keywords, but instead encapsulates internal class details by default. Users of a Smalltalk object other than *self* cannot access its instance variables without sending a message to that object. In C++ and Java class fields and methods are made private to restrict external usage. To gain access to or update private class data fields one must use the public mutator and accessor methods defined in the class. Only by using these public methods can a user of an object change the objects state. Protected members can only be accessed by a subclass of its class. In Java the class declaration and the implementation of the methods are stored in one file. However, C++ often places them in different files – a **.cpp** file and a corresponding **.h** file. When data or methods are declared public, subclasses and any other class working on the object can use that objects public methods. Declaring all instance variables and methods public is bad programming style and does not conform to the OOP notion of data encapsulation. Below are examples of how C++, Java, and Smalltalk enforce encapsulation.

**C++ Example Magic\_Square.h**

Notice how the private and public keyword modifiers define sections where all methods and data in that section take on the meaning of the access modifier.

#ifndef Magic\_Square\_H

#define Magic\_Square\_H

class MagicSquareCreator

**{**

private**:**

static const int \_ROWS **=** 4**;**

static const int \_COLUMNS **=** 4**;**

static const int \_REF\_MS\_VALUE **=** 34**;**

static const int referenceSquare**[**\_ROWS**][**\_COLUMNS**];**

int magicSquareMatrix**[**\_ROWS**][**\_COLUMNS**];**

public**:**

MagicSquareCreator**();**

int getCellValue**(**const int**&,** const int**&)** const**;**

bool getMagicSquare**(**int anArray**[][**4**],** int**,** int**)**const**;**

bool getMagicSquare**(**int v**,** int anArray**[][**\_COLUMNS**],** int**,** int**);**

int getMsValue**()**const**;**

**};**

#endif

**Java Example LoopEvent.java**

Notice that with Java you must explicitly declare each instance variable and method either private, public, or protected by placing the appropriate keyword before its definition.

public class LoopEvent **extends** SoundEvent **{**

public static final String STATE\_WAITING **=** "waiting"**;**

public static final String STATE\_RUNNING **=** "running"**;**

private String currentState**;**

public LoopEvent**(**AudioStream audio**)** **{**

**super(**audio**);**

currentState **=** STATE\_WAITING**;**

**}**

protected void processEvent**(**String event**)** **throws** InterruptedException **{**

System**.**out**.**println**(**"Got " **+** event **+** " event"**);**

**}**

**}**

**Smalltalk example Complex.st**

Notice that Smalltalk provides accessor methods with the same name of the instance variables. Without accessor methods the instance variables are for private use of the class only. The ^ symbol is the return symbol in Smalltalk and is used to return the two instance variables in the example below.

Number subclass: **#Complex**

instanceVariableNames:

'realPart imaginaryPart '

. . .

imaginaryPart

"Answer the imaginary part."

**^**imaginaryPart

realPart

"Answer the real part."

**^**realPart

. . .

**#5 Chapter 5 Problem 5.23**

The fact that Smalltalk does not have a generic collections library in my opinion puts Smalltalk programmers at a slight disadvantage. I believe this to be so because the generic collections of Java and the Template collections of C++ allow for parametric polymorphism which is a very powerful and extremely useful concept in OOP. It allows for many collection types to leverage a single type definition. This in turn requires dynamic binding at runtime of the collection objects. In Smalltalk you cannot do this and so a collection of a single type will be statically bound at compile time. A Smalltalk programmer could argue that this ensures code reliability since all types are known at compile time. However, I would argue that by using generic collections and parametric polymorphism your code is more flexible and less prohibitive in nature which allows for more choices to the programmer with an additional advantage of having to write less code.