Chapter 3: Functional Programming

1. Functional gcd — recursion replacing a loop

```
int gcd( int u, int v)
{
    if (v == 0) return u;
    else return gcd(v, u % v);
}
```

After modifying them so that they can run as a program:

```
#include <stdio.h>

int gcd(int u, int v) {
    if (v == 0) {
        return u;
    }
    return gcd(v, u % v);
}

int main(void) {
    printf("gcd(48, 18) = %d\n", gcd(48, 18));
    printf("gcd(10, 35) = %d\n", gcd(10, 35));
    printf("gcd(270, 192) = %d\n", gcd(270, 192));
    return 0;
}
```

Purpose:

Implements the recursive gcd function to show the functional style of solving a problem by recursion instead of using a loop.

How To Run:

Compile and run with a C compiler:

```
gcc 01_gcd.c -o gcd
./gcd
```

Output:

```
gcd(48, 18) = 6
gcd(10, 35) = 5
gcd(270, 192) = 6
```

2. Scheme factorial with recursion

```
(letrec ((factorial
          (lambda (n)
            (if (= n 0) 1
                (* n (factorial (- n 1)))))))
  (factorial 10))
```

After modifying them so that they can run as a program:

```
(display "factorial(10) = ")
(display
  (letrec ((factorial
            (lambda (n)
              (if (= n 0)
                  1
                  (* n (factorial (- n 1)))))))
    (factorial 10)))
(newline)
```

Purpose:

Computes factorial in Scheme using letrec, showing a recursive function definition in a functional style.

How To Run:

Run in a Scheme interpreter, for example Racket:

```
racket 02_scheme_factorial_letrec.scm
```

Output:

```
factorial(10) = 3628800
```

3. Tail-recursive Scheme factorial

```
(define factorial
  (lambda (n result)
    (if (= n 1)
        result
        (factorial (- n 1) (* n result)))))
```

After modifying them so that they can run as a program:

```
(define factorial-iter
  (lambda (n result)
    (if (= n 0)
        result
        (factorial-iter (- n 1) (* n result)))))

(define factorial
  (lambda (n)
    (factorial-iter n 1)))

(display "factorial(10) = ")
(display (factorial 10))
(newline)
(display "factorial(6) = ")
```

```
(display (factorial 6))
(newline)
```

Purpose:
Computes factorial in Scheme using tail recursion, showing how recursion can be written in a more efficient functional form.

How To Run:
Run in a Scheme interpreter, for example Racket:
`racket 03_scheme_factorial_tail.scm`

Output:
```
factorial(10) = 3628800
factorial(6) = 720
```

4. Scheme map — higher-order function

```
(define map
  (lambda (f L)
    (if (null? L) '()
        (cons (f (car L)) (map f (cdr L))))))

(define square
  (lambda (x) (* x x)))

(define square-list
  (lambda (L) (map square L)))
```

After modifying them so that they can run as a program:

```
(define my-map
  (lambda (f L)
    (if (null? L)
        '()
        (cons (f (car L))
              (my-map f (cdr L))))))

(define square
  (lambda (x)
    (* x x)))

(define square-list
  (lambda (L)
    (my-map square L)))

(display "square-list '(1 2 3 4) = ")
```

```
(display (square-list '(1 2 3 4)))
(newline)
(display "my-map double '(1 2 3 4) = ")
(display (my-map (lambda (x) (+ x x)) '(1 2 3 4)))
(newline)
```

Purpose:
Defines a custom my-map function and uses it to build square-list, showing higher-order functions and list processing in Scheme.

How To Run:
Run in a Scheme interpreter:
`racket 04_scheme_map_square_list.scm`

Output:
```
square-list '(1 2 3 4) = (1 4 9 16)
my-map double '(1 2 3 4) = (2 4 6 8)
```

5. Scheme make-double — function returning a function

```
(define make-double
  (lambda (f)
    (lambda (x) (f x x))))

(define square (make-double *))
(define double (make-double +))
```

After modifying them so that they can run as a program:

```
(define make-double
  (lambda (f)
    (lambda (x)
      (f x x))))

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

(display "square 5 = ")
(display (square 5))
(newline)
(display "double 7 = ")
(display (double 7))
(newline)
```

Purpose:

Defines make-double, a function that returns another function, showing that functions are first-class values in functional programming.

How To Run:
Run in a Scheme interpreter:
`racket 05_scheme_make_double.scm`

Output:
```
square 5 = 25
double 7 = 14
```

6. ML append with pattern matching

```
fun append ([] , L) = L
  | append (h :: t , L) = h :: append (t , L);
```

After modifying them so that they can run as a program:

```
fun append ([] , L) = L
  | append (h :: t , L) = h :: append (t , L);

fun join [] = ""
  | join [x] = x
  | join (x :: xs) = x ^ ", " ^ join xs;

fun intListToString xs =
    "[" ^ join (List.map Int.toString xs) ^ "]";

val _ = print ("append([1, 2, 3], [4, 5]) = " ^
               intListToString (append ([1, 2, 3], [4, 5]))
^ "\n");
```

Purpose:
Implements the append function in Standard ML using pattern matching, showing recursive list processing without mutation.

How To Run:
Run in an SML environment such as SML:
`sml 06_ml_append.sml`

Output:
```
append([1, 2, 3], [4, 5]) = [1, 2, 3, 4, 5]
```

7. ML higher-order function make_double

```
fun make_double f = fn x => f (x,x);
```

```
val square = make_double (op * );
val double = make_double (op +);
```

After modifying them so that they can run as a program:

```
fun make_double f = fn x => f (x, x);

val square = make_double (op *);
val double = make_double (op +);

val _ = print ("square 5 = " ^ Int.toString (square 5) ^
"\n");
val _ = print ("double 7 = " ^ Int.toString (double 7) ^
"\n");
```

Purpose:

Defines the higher-order function make_double in Standard ML and uses it to create square and double, showing functions as input and output values.

How To Run:

Run in Standard ML:

```
sml 07_ml_make_double.sml
```

Output:

```
square 5 = 25
double 7 = 14
```

8. Haskell square_list with map

```
square_list = map (\x -> x * x)
```

After modifying them so that they can run as a program:

```
module Main where

squareList :: Num a => [a] -> [a]
squareList = map (\x -> x * x)

main :: IO ()
main = putStrLn $ "squareList [1,2,3,4] = " ++ show
(squareList [1,2,3,4] :: [Int])
```

Purpose:

Defines squareList in Haskell using map and an anonymous function, showing declarative list transformation with a higher-order function.

How To Run:

Compile and run with GHC:

```
ghc 08_haskell_square_list_map.hs -o hs_map
./hs_map
```

Output:

```
squareList [1,2,3,4] = [1,4,9,16]
```

9. Haskell list comprehension

```
square_list lis = [ x * x | x <- lis]
```

After modifying them so that they can run as a program:

```
module Main where
squareList :: Num a => [a] -> [a]
squareList lis = [x * x | x <- lis]

main :: IO ()
main = putStrLn $ "squareList [1,2,3,4] = " ++ show
(squareList [1,2,3,4] :: [Int])
```

Purpose:

Defines squareList in Haskell using list comprehension, showing a concise mathematical style for processing lists.

How To Run:

Compile and run with GHC:

```
ghc 09_haskell_square_list_comprehension.hs -o hs_comp
./hs_comp
```

Output:

```
squareList [1,2,3,4] = [1,4,9,16]
```

Summary

Chapter 3 explains functional programming as a style in which programs are treated as mathematical functions. It emphasizes recursion instead of loops, the absence of mutable state, referential transparency, and the idea that functions are first-class values that can be passed as arguments or returned as results. The chapter introduces major functional languages such as Scheme, ML, and Haskell, and shows how concepts like higher-order functions, pattern matching, tail recursion, delayed evaluation, and lambda calculus help create concise, predictable, and declarative programs.

Chapter 4: Logic Programming

1. Natural numbers in Prolog

```
natural(0).
natural(successor(X)) :- natural(X).
```

After modifying them so that they can run as a program:

```
natural(0).
natural(successor(X)) :- natural(X).

:- initialization(main).

main :-
    ( natural(0) ->
        writeln('natural(0) = yes')
    ;   writeln('natural(0) = no')
    ),
    ( natural(successor(successor(0))) ->
        writeln('natural(successor(successor(0))) = yes')
    ;   writeln('natural(successor(successor(0))) = no')
    ),
    ( natural(21) ->
        writeln('natural(21) = yes')
    ;   writeln('natural(21) = no')
    ),
    halt.
```

Purpose:
Implements a simple logic program with a fact and a recursive rule to show how Prolog represents knowledge and proves whether a statement is true.

How To Run:
Run in SWI-Prolog:

```
swipl -q -s 01_natural.pl
```

Output:

```
natural(0) = yes
natural(successor(successor(0))) = yes
natural(21) = no
```

2. Ancestor relation in Prolog

```
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
ancestor(X, X).
parent(amy, bob).
```

After modifying them so that they can run as a program:

```
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
ancestor(X, X).
parent(amy, bob).
```

```
:- initialization(main).

main :-
    findall(X, ancestor(X, bob), XS),
    format("ancestor(X, bob) -> X = ~w~n", [XS]),
    halt.
```

Purpose:
Defines a recursive logical relation and shows how Prolog can return multiple answers through search and backtracking.

How To Run:
Run in SWI-Prolog:
```
swipl -q -s 02_ancestor.pl
```

Output:
```
ancestor(X, bob) -> X = [amy,bob]
```

3. Prolog gcd with arithmetic evaluation

```
gcd(U, 0, U).
gcd(U, V, W) :- not(V = 0), R is U mod V, gcd(V, R, W).
```

After modifying them so that they can run as a program:
```
gcd(U, 0, U).
gcd(U, V, W) :-
    V =\= 0,
    R is U mod V,
    gcd(V, R, W).

:- initialization(main).

main :-
    gcd(15, 10, X),
    format("gcd(15, 10) = ~w~n", [X]),
    gcd(270, 192, Y),
    format("gcd(270, 192) = ~w~n", [Y]),
    halt.
```

Purpose:
Implements Euclid's algorithm in Prolog to show recursive problem solving together with arithmetic evaluation using is. The chapter presents this as a key Prolog example alongside append and reverse.

How To Run:
Run in SWI-Prolog:
```
swipl -q -s 03_gcd.pl
```

Output:
```
gcd(15, 10) = 5
```

```
gcd(270, 192) = 6
```

4. Prolog append

```
append([], Y, Y).
append([A|B], Y, [A|W]) :- append(B, Y, W).
```

After modifying them so that they can run as a program:

```
append([], Y, Y).
append([A|B], Y, [A|W]) :- append(B, Y, W).

print_splits([]).
print_splits([[X,Y]|Rest]) :-
    format("X = ~w, Y = ~w~n", [X, Y]),
    print_splits(Rest).

:- initialization(main).

main :-
    append([1,2], [3,4], Z),
    format("append([1,2], [3,4], Z) -> Z = ~w~n", [Z]),
    findall([X, Y], append(X, Y, [1,2]), Splits),
    writeln('append(X, Y, [1,2]) gives:'),
    print_splits(Splits),
    halt.
```

Purpose:
Implements list concatenation in Prolog and shows how the same logical definition can be used forward and backward through unification.

How To Run:
Run in SWI-Prolog:

```
swipl -q -s 04_append.pl
```

Output:

```
append([1,2], [3,4], Z) -> Z = [1,2,3,4]
append(X, Y, [1,2]) gives:
X = [], Y = [1,2]
X = [1], Y = [2]
X = [1,2], Y = []
```

5. Prolog reverse

```
reverse([], []).
reverse([H|T], L) :- reverse(T, L1),
 append(L1, [H], L).
```

After modifying them so that they can run as a program:

```
append([], Y, Y).
append([A|B], Y, [A|W]) :- append(B, Y, W).

reverse([], []).
```

```
reverse([H|T], L) :-
    reverse(T, L1),
    append(L1, [H], L).

:- initialization(main).

main :-
    reverse([1,2,3,4], L),
    format("reverse([1,2,3,4]) = ~w~n", [L]),
    halt.
```

Purpose:
Implements list reversal in Prolog using recursion and the previously defined append, showing declarative list processing. reverse appears with gcd and append in the chapter's main Prolog examples.

How To Run:
Run in SWI-Prolog:
```
swipl -q -s 05_reverse.pl
```

Output:
```
reverse([1,2,3,4]) = [4,3,2,1]
```

6. Prolog printpieces using fail and backtracking

```
printpieces(L) :- append(X, Y, L),
 write(X),
 write(Y),
 nl,
 fail.
```

After modifying them so that they can run as a program:
```
append([], Y, Y).
append([A|B], Y, [A|W]) :- append(B, Y, W).

printpieces(L) :-
    append(X, Y, L),
    format("X = ~w, Y = ~w~n", [X, Y]),
    fail.
printpieces(_).

:- initialization(main).

main :-
    writeln('All ways to split [1,2] using append/3:'),
    printpieces([1,2]),
    halt.
```

Purpose:
Shows how Prolog uses forced failure and backtracking to enumerate all solutions to a query.

How To Run:
Run in SWI-Prolog:
```
swipl -q -s 06_printpieces.pl
```

Output:
```
All ways to split [1,2] using append/3:
X = [], Y = [1,2]
X = [1], Y = [2]
X = [1,2], Y = []
```

7. Prolog writenum using cut

```
num(0).
num(X) :- num(Y), X is Y + 1.
writenum(I, J) :- num(X),
 I =< X,
 X =< J,
 write(X), nl,
 X = J, !,
 fail.
```

After modifying them so that they can run as a program:
```
num(0).
num(X) :- num(Y), X is Y + 1.

writenum(I, J) :-
    num(X),
    I =< X,
    X =< J,
    write(X), nl,
    ( X =:= J -> !
    ; fail
    ).

:- initialization(main).

main :-
    writenum(1, 10),
    halt.
```

Purpose:
Demonstrates how Prolog uses cut to control search and stop backtracking once the desired upper bound is reached.

How To Run:
Run in SWI-Prolog:
```
swipl -q -s 07_writenum.pl
```

Output:
```
1
2
```

3
4
5
6
7
8
9
10

8. The sieve of Eratosthenes in Prolog

```
primes(Limit, Ps) :- integers(2, Limit, Is),
 sieve(Is, Ps).
integers(Low, High, [Low|Rest]) :-
 Low =< High, !, M is Low+1,
 integers(M, High, Rest).
integers(Low, High, []).
sieve([], []).
sieve([I|Is], [I|Ps]) :- remove(I, Is, New),
 sieve(New, Ps).
remove(P, [], []).
remove(P, [I|Is], [I|Nis]) :-
 not(0 is I mod P), !, remove(P, Is, Nis).
remove(P, [I|Is], Nis) :-
 0 is I mod P, !, remove(P, Is, Nis).
```

After modifying them so that they can run as a program:

```
primes(Limit, Ps) :-
    integers(2, Limit, Is),
    sieve(Is, Ps).

integers(Low, High, [Low|Rest]) :-
    Low =< High, !,
    M is Low + 1,
    integers(M, High, Rest).
integers(_, _, []).

sieve([], []).
sieve([I|Is], [I|Ps]) :-
    remove(I, Is, New),
    sieve(New, Ps).

remove(_, [], []).
remove(P, [I|Is], [I|Nis]) :-
    I mod P =\= 0, !,
    remove(P, Is, Nis).
remove(P, [_|Is], Nis) :-
    remove(P, Is, Nis).

:- initialization(main).

main :-
```

```
    primes(30, Ps),
    format("primes(30) = ~w~n", [Ps]),
    halt.
```

Purpose:
Implements the sieve of Eratosthenes in Prolog to show a more substantial logic program using recursion, list processing, and cut. This appears in the chapter as a longer Prolog example.

How To Run:
Run in SWI-Prolog:
```
swipl -q -s 08_primes.pl
```

Output:
```
primes(30) = [2,3,5,7,11,13,17,19,23,29]
```

9. Quicksort in Prolog

```
qsort([], []).
qsort([H|T], S) :- partition(H, T, L, R),
 qsort(L, L1),
 qsort(R, R1),
 append(L1, [H|R1], S).
partition(P, [A|X], [A|Y], Z) :- A < P,
 partition(P, X, Y, Z).
partition(P, [A|X], Y, [A|Z]) :- A >= P,
 partition(P, X, Y, Z).
partition(P, [], [], []).
```

After modifying them so that they can run as a program:
```
append([], Y, Y).
append([A|B], Y, [A|W]) :- append(B, Y, W).

qsort([], []).
qsort([H|T], S) :-
    partition(H, T, L, R),
    qsort(L, L1),
    qsort(R, R1),
    append(L1, [H|R1], S).

partition(P, [A|X], [A|Y], Z) :-
    A < P,
    partition(P, X, Y, Z).
partition(P, [A|X], Y, [A|Z]) :-
    A >= P,
    partition(P, X, Y, Z).
partition(_, [], [], []).

:- initialization(main).

main :-
```

```
qsort([3,1,4,1,5,9,2,6], S),
format("qsort([3,1,4,1,5,9,2,6]) = ~w~n", [S]),
halt.
```

Purpose:
Implements quicksort in Prolog to show that logic programming can also express a full sorting algorithm with recursive partitioning. The chapter presents this as a Prolog quicksort example after discussing the limits of pure specification-style sorting.

How To Run:
Run in SWI-Prolog:
```
swipl -q -s 09_qsort.pl
```

Output:
```
qsort([3,1,4,1,5,9,2,6]) = [1,1,2,3,4,5,6,9]
```

Summary

Chapter 4 presents logic programming as a paradigm in which computation is expressed through logical facts, rules, and queries rather than step-by-step instructions. It introduces first-order predicate calculus, Horn clauses, resolution, and unification as the theoretical foundations of logic programming, then uses Prolog as the main practical example. The chapter shows how Prolog can solve problems such as gcd, append, reverse, and sorting through deduction and backtracking, while also discussing limitations such as control dependence, negation as failure, the closed-world assumption, and the difficulty of representing all of logic with Horn clauses alone.

Chapter 5: Object-Oriented Programming

1. Smalltalk message passing with Set

```
Set new
Set new size
Set new includes: 'Hello'
Set new add: 'Ken'
```

After modifying them so that they can run as a program:

```
| s |
s := Set new.

Transcript nextPutAll: 'size = '; nextPutAll: s size
printString; cr.
Transcript nextPutAll: 'includes Hello = '; nextPutAll: (s
includes: 'Hello') printString; cr.

s add: 'Ken'.

Transcript nextPutAll: 'includes Ken = '; nextPutAll: (s
includes: 'Ken') printString; cr.
```

Purpose:
Shows the basic object-oriented style of Smalltalk: creating objects, sending messages to them, and observing their state changes.

How To Run:
The code is run in a Smalltalk environment by opening a Transcript, pasting the code into it, and evaluating it using the “Do it” or “Show it” option; the output will then be displayed directly in the Transcript window, as described in the chapter.

Output:

```
size = 0
includes Hello = false
includes Ken = true
```

2. Smalltalk array control with to:do: and ifTrue:ifFalse:

```
| array |
array := Array new: 100.
1 to: array size do: [:i |
 i odd
 ifTrue: [array at: i put: 1]
 ifFalse: [array at: i put: 2]]
```

After modifying them so that they can run as a program:

```
| array |
array := Array new: 10.

1 to: array size do: [:i |
    i odd
        ifTrue: [array at: i put: 1]
        ifFalse: [array at: i put: 2]].

array do: [:element |
    Transcript nextPutAll: element printString; cr].
```

Purpose:
Shows how Smalltalk expresses loop and conditional control through message passing and blocks, instead of using conventional procedural syntax.

How To Run:
The code is run in a Smalltalk environment by opening a Transcript, pasting the code into it, and evaluating it using the "Do it" or "Show it" option; the output will then be displayed directly in the Transcript window, as described in the chapter.

Output:
```
1
2
1
2
1
2
1
2
1
2
```

3. Smalltalk Fraction and polymorphic max:

```
| oneHalf twoThirds |
oneHalf := Fraction numerator: 1 denominator: 2.
twoThirds := Fraction numerator: 2 denominator: 3.
(oneHalf max: twoThirds) printString
```

After modifying them so that they can run as a program:

```
| oneHalf twoThirds |
oneHalf := Fraction numerator: 1 denominator: 2.
twoThirds := Fraction numerator: 2 denominator: 3.

Transcript nextPutAll: 'max = '; nextPutAll: (oneHalf max:
twoThirds) printString; cr.
```

Purpose:

Shows inheritance and polymorphism in Smalltalk. The chapter uses Magnitude>>>max: together with subclass-specific comparison behavior to illustrate dynamic binding.

How To Run:
The code is run in a Smalltalk environment by opening a Transcript, pasting the code into it, and evaluating it using the "Do it" or "Show it" option; the output will then be displayed directly in the Transcript window, as described in the chapter.

Output:
```
max = 2/3
```

4. **Smalltalk Complex class**

```
| c1 c2 c3 |
c1 := Complex realPart: 3.0 imaginaryPart: 2.5.
c2 := Complex realPart: 6.0 imaginaryPart: 4.5.
c3 := c1 * c2.
Transcript nextPutAll: (c3 printString); cr
```

After modifying them so that they can run as a program:

```
Number subclass: #Complex
    instanceVariableNames: 'realPart imaginaryPart'
    classVariableNames: ''
    poolDictionaries: ''.

Complex class >> realPart: r imaginaryPart: i
    ^self new realPart: r imaginaryPart: i

Complex >> realPart: r imaginaryPart: i
    realPart := r.
    imaginaryPart := i.
    ^self

Complex >> realPart
    ^realPart

Complex >> imaginaryPart
    ^imaginaryPart

Complex >> * aNumber
    | rp ip |
    rp := self realPart * aNumber realPart - self
imaginaryPart * aNumber imaginaryPart.
    ip := self realPart * aNumber imaginaryPart + self
imaginaryPart * aNumber realPart.
    ^self class realPart: rp imaginaryPart: ip

Complex >> printOn: aStream
    realPart printOn: aStream.
```

```
    aStream nextPut: $+.
    imaginaryPart printOn: aStream.
    aStream nextPut: $i

| c1 c2 c3 |
c1 := Complex realPart: 3.0 imaginaryPart: 2.5.
c2 := Complex realPart: 6.0 imaginaryPart: 4.5.
c3 := c1 * c2.
Transcript nextPutAll: c3 printString; cr.
```

Purpose:
Shows how a new class is defined in Smalltalk, how instance variables are encapsulated behind methods, and how inherited arithmetic behavior can be extended with new methods. The chapter uses Complex as its main custom Smalltalk class example.

How To Run:
The code is run in a Smalltalk environment by opening a Transcript, pasting the code into it, and evaluating it using the "Do it" or "Show it" option; the output will then be displayed directly in the Transcript window, as described in the chapter.

Output:
```
6.75+28.5i
```

5. Smalltalk LinkedQueue

```
Collection subclass: #LinkedQueue
 instanceVariableNames:
 'front rear size '
 classVariableNames: ''
 poolDictionaries: ''
...
| q |
q := LinkedQueue new.
q addAll: #(1 2 3 4 5).
q dequeue.
q enqueue: 'Ken'.
q inspect
```

After modifying them so that they can run as a program:

```
Collection subclass: #LinkedQueue
    instanceVariableNames: 'front rear size'
    classVariableNames: ''
    poolDictionaries: ''.

LinkedQueue class >> new
    ^super new initialize

LinkedQueue >> initialize
    front := nil.
```

```
    rear := nil.
    size := 0.
    ^self

LinkedQueue >> enqueue: anObject
    | newNode |
    newNode := Association key: anObject value: nil.
    rear notNil ifTrue: [rear value: newNode].
    rear := newNode.
    front isNil ifTrue: [front := newNode].
    size := size + 1.
    ^anObject

LinkedQueue >> dequeue
    | oldNode |
    oldNode := front.
    front := front value.
    front isNil ifTrue: [rear := nil].
    size := size - 1.
    ^oldNode key

LinkedQueue >> size
    ^size

LinkedQueue >> add: anObject
    ^self enqueue: anObject

LinkedQueue >> do: aBlock
    | probe |
    probe := front.
    [probe notNil] whileTrue: [
        aBlock value: probe key.
        probe := probe value].

| q |
q := LinkedQueue new.
q addAll: #(1 2 3 4 5).
q dequeue.
q enqueue: 'Ken'.

q do: [:element |
    Transcript nextPutAll: element printString; cr].

Transcript nextPutAll: 'size = '; nextPutAll: q size
printString; cr.
```

Purpose:
Shows object-oriented design with a custom collection class, encapsulated state, queue operations, inheritance from Collection, and reuse through add: and do:. The chapter presents LinkedQueue as a full Smalltalk class example.

How To Run:
The code is run in a Smalltalk environment by opening a Transcript, pasting the code into it, and evaluating it using the “Do it” or “Show it” option; the output will then be displayed directly in the Transcript window, as described in the chapter.

Output:

```
2
3
4
5
Ken
size = 5
```

6. Java Complex and TestComplex

```java
import numbers.Complex;
public class TestComplex{
 static public void main(String[] args){
  Complex c1 = new Complex(3.5, 4.6);
  Complex c2 = new Complex(2.0, 2.0);
  Complex sum = c1.add(c2);
  System.out.println(sum);
 }
}
```

After modifying them so that they can run as a program:

```java
public class TestComplex {
    static class Complex {
        private double re, im;

        public Complex() {
            this(0, 0);
        }

        public Complex(double realPart, double imaginaryPart) {
            re = realPart;
            im = imaginaryPart;
        }

        public double realPart() {
            return re;
        }

        public double imaginaryPart() {
            return im;
        }

        public Complex add(Complex c) {
```

```
            return new Complex(this.realPart() + c.realPart(),
                               this.imaginaryPart() + c.imaginaryPart());
        }

        public Complex multiply(Complex c) {
            return new Complex(this.realPart() * c.realPart() -
                               this.imaginaryPart() * c.imaginaryPart(),
                               this.realPart() * c.imaginaryPart() +
                               this.imaginaryPart() * c.realPart());
        }

        public String toString() {
            return this.realPart() + "+" + this.imaginaryPart() + "i";
        }
    }

    public static void main(String[] args) {
        Complex c1 = new Complex(3.5, 4.6);
        Complex c2 = new Complex(2.0, 2.0);
        Complex sum = c1.add(c2);
        System.out.println("sum = " + sum);
    }
}
```

Purpose:
Shows encapsulation in Java with private instance variables, accessor methods, constructor chaining, method calls on objects, and overriding toString. The chapter uses Complex as its main Java class example.

How To Run:
Compile and run with a Java compiler:

```
javac TestComplex.java
java TestComplex
```

Output:
```
sum = 5.5+6.6i
```

7. Java generic iterators with map and reduce

```
public static<E> E reduce(E baseElement,
 ReduceStrategy<E> strategy,
 Collection <E> c){
 for (E element : c)
  baseElement = strategy.combine(baseElement, element);
 return baseElement;
}
```

After modifying them so that they can run as a program:

```
import java.util.*;

public class TestIterators {
    interface MapStrategy<E, R> {
        R transform(E element);
    }

    interface ReduceStrategy<E> {
        E combine(E x, E y);
    }

    static class Iterators {
        public static <E, R> Collection<R> map(MapStrategy<E, R> strategy,
                                               Collection<E> c) {
            Collection<R> results = new ArrayList<R>();
            for (E element : c) {
                results.add(strategy.transform(element));
            }
            return results;
        }

        public static <E> E reduce(E baseElement,
                                   ReduceStrategy<E> strategy,
                                   Collection<E> c) {
            for (E element : c) {
                baseElement = strategy.combine(baseElement, element);
            }
            return baseElement;
        }
    }

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        for (int i = 1; i <= 8; i++) {
            list.add(i);
        }

        Collection<Integer> squares =
            Iterators.map(new MapStrategy<Integer, Integer>() {
                public Integer transform(Integer i) {
                    return i * i;
                }
            }, list);

        Integer sum =
            Iterators.reduce(0, new ReduceStrategy<Integer>() {
```

```
                public Integer combine(Integer a, Integer b) {
                    return a + b;
                }
            }, squares);

        System.out.println("squares = " + squares);
        System.out.println("sum = " + sum);
    }
}
```

Purpose:
Shows Java interfaces, generics, polymorphism, and higher-level reusable collection operations. The chapter discusses defining map, filter, and reduce in Java through strategy interfaces and static utility methods.

How To Run:
Compile and run with a Java compiler:

```
javac TestIterators.java
java TestIterators
```

Output:
```
squares = [1, 4, 9, 16, 25, 36, 49, 64]
sum = 204
```

8. C++ abstract class with pure virtual area

```
class ClosedFigure{
 public:
 virtual double area() = 0;
 ...
};
class Rectangle : public ClosedFigure{
 public:
 double area(){
 return length * width; }
 ...
};
class Circle : public ClosedFigure{
 public:
 double area(){
 return PI*radius*radius; }
 ...
};
```

After modifying them so that they can run as a program:

```
#include <iostream>
#include <iomanip>
```

```cpp
using namespace std;

class ClosedFigure {
public:
    virtual double area() const = 0;
    virtual const char* name() const = 0;
    virtual ~ClosedFigure() {}
};

class Rectangle : public ClosedFigure {
public:
    Rectangle(double l, double w) : length(l), width(w) {}
    double area() const override {
        return length * width;
    }
    const char* name() const override {
        return "Rectangle";
    }
private:
    double length, width;
};

class Circle : public ClosedFigure {
public:
    Circle(double r) : radius(r) {}
    double area() const override {
        return 3.141592653589793 * radius * radius;
    }
    const char* name() const override {
        return "Circle";
    }
private:
    double radius;
};

int main() {
    Rectangle r(3.0, 4.0);
    Circle c(2.0);

    cout << fixed << setprecision(2);
    cout << r.name() << " area = " << r.area() << endl;
    cout << c.name() << " area = " << c.area() << endl;
    return 0;
}
```

Purpose:

Shows inheritance, abstract classes, pure virtual functions, and overridden methods in C++. The chapter presents ClosedFigure with pure virtual area as the standard example of abstraction in C++.

How To Run:
Compile and run with a C++ compiler:

```
g++ 08_closed_figure.cpp -o closed_figure
./closed_figure
```

Output:

```
Rectangle area = 12.00
Circle area = 12.57
```

9. C++ static binding and dynamic binding

```cpp
class A
{
 public:
 void p()
 { cout << "A::p\n" ;
 }
 virtual void q()
 { cout << "A::q\n" ;
 }
 void f()
 { p();
   q();
 }
};
class B : public A
{
 public:
 void p()
 { cout << "B::p\n" ;
 }
 void q()
 { cout << "B::q\n" ;
 }
};
```

After modifying them so that they can run as a program:

```cpp
#include <iostream>
using namespace std;

class A {
public:
    void p() {
```

```
        cout << "A::p" << endl;
    }

    virtual void q() {
        cout << "A::q" << endl;
    }

    void f() {
        p();
        q();
    }
};

class B : public A {
public:
    void p() {
        cout << "B::p" << endl;
    }

    void q() override {
        cout << "B::q" << endl;
    }
};

int main() {
    cout << "Object case:" << endl;
    A a;
    B b;
    a.f();
    b.f();
    a = b;
    a.f();

    cout << "Pointer case:" << endl;
    A aObj;
    B bObj;
    A* ap = &aObj;
    B* bp = &bObj;
    ap->f();
    bp->f();
    ap = bp;
    ap->f();

    return 0;
}
```

Purpose:

Shows the difference between static binding and dynamic binding in C++. The chapter uses this exact style of example to show that only virtual methods are dynamically bound, and that the object must be accessed through a pointer or reference for the dynamic effect to appear fully.

How To Run:
Compile and run with a C++ compiler:

```
g++ 09_dynamic_binding.cpp -o dynamic_binding
./dynamic_binding
```

Output:
```
Object case:
A::p
A::q
A::p
B::q
A::p
A::q
Pointer case:
A::p
A::q
A::p
B::q
A::p
B::q
```

Summary

Chapter 5 focuses on object-oriented programming as a paradigm designed to improve software reuse, modifiability, and component independence. It explains the key ideas of objects, classes, encapsulation, inheritance, polymorphism, and dynamic binding, using Smalltalk as the purest object-oriented example and then comparing it with Java and C++. The chapter shows how object-oriented languages organize software around interacting objects, how inheritance and method overriding support code reuse, and how different languages balance flexibility, abstraction, and runtime efficiency in their implementations of object-oriented features.