Task 5 – Population Class

This task sets up the Population class, featuring a list of individuals and a generation number as its fields. An initial population based on a population size is generated for the demo. Methods that calculate the fitness average, the most-fit-individual, and the population size have been implemented in this task. Selection methods have been left out since that is the next task. I also implemented a method to create a music individual given an individual number.

Demo

```
[1]> ( load "D:\\Programming Projects\\Lisp
Projects\\466Project\\acga.lisp" )
;; Loading file D:\Programming Projects\Lisp
Projects\466Project\acga.lisp ...
;; Loaded file D:\Programming Projects\Lisp
Projects\466Project\acga.lisp
T
[2]> ( population-demo )


Generation 0 population ...

1     #<MUSIC #x1AA05AA5>  0
2     #<MUSIC #x1AA05D11>  0
3     #<MUSIC #x1AA060BD>  0
4     #<MUSIC #x1AA06361>  0
5     #<MUSIC #x1AA06515>  0
6     #<MUSIC #x1AA067B9>  0
7     #<MUSIC #x1AA06A35>  0
8     #<MUSIC #x1AA06BE9>  0
9     #<MUSIC #x1AA06E8D>  0
10    #<MUSIC #x1AA07221>  0
11    #<MUSIC #x1AA07565>  0
12    #<MUSIC #x1AA07809>  0
13    #<MUSIC #x1AA07AAD>  0
14    #<MUSIC #x1AA07DA1>  0
15    #<MUSIC #x1AA08095>  0
16    #<MUSIC #x1AA08339>  0
17    #<MUSIC #x1AA084C5>  0
```

```
18    #<MUSIC #x1AA08719>  0
19    #<MUSIC #x1AA088F5>  0
20    #<MUSIC #x1AA08A59>  0
21    #<MUSIC #x1AA08C5D>  0
22    #<MUSIC #x1AA08F79>  0
23    #<MUSIC #x1AA091F5>  0
24    #<MUSIC #x1AA09421>  0
25    #<MUSIC #x1AA0973D>  0
26    #<MUSIC #x1AA09AD1>  0
27    #<MUSIC #x1AA09CFD>  0
28    #<MUSIC #x1AA09E61>  0
29    #<MUSIC #x1AA0A12D>  0
30    #<MUSIC #x1AA0A3D1>  0
31    #<MUSIC #x1AA0A50D>  0
32    #<MUSIC #x1AA0A699>  0
33    #<MUSIC #x1AA0A8ED>  0
34    #<MUSIC #x1AA0AC09>  0
35    #<MUSIC #x1AA0AE35>  0
36    #<MUSIC #x1AA0B1A1>  0
37    #<MUSIC #x1AA0B4E5>  0
38    #<MUSIC #x1AA0B621>  0
39    #<MUSIC #x1AA0B785>  0
40    #<MUSIC #x1AA0BA01>  0
41    #<MUSIC #x1AA0BCF5>  0
42    #<MUSIC #x1AA0BE59>  0
43    #<MUSIC #x1AA0C1C5>  0
44    #<MUSIC #x1AA0C3C9>  0
45    #<MUSIC #x1AA0C57D>  0
46    #<MUSIC #x1AA0C8C1>  0
47    #<MUSIC #x1AA0CC05>  0
48    #<MUSIC #x1AA0CEA9>  0
49    #<MUSIC #x1AA0D085>  0
50    #<MUSIC #x1AA0D329>  0
51    #<MUSIC #x1AA0D5F5>  0
52    #<MUSIC #x1AA0D7F9>  0
53    #<MUSIC #x1AA0D985>  0
54    #<MUSIC #x1AA0DCC9>  0
55    #<MUSIC #x1AA0E00D>  0
56    #<MUSIC #x1AA0E2D9>  0
57    #<MUSIC #x1AA0E48D>  0
58    #<MUSIC #x1AA0E351>  0
```

```
59    #<MUSIC #x1AA0E645>  0
60    #<MUSIC #x1AA0E7F9>  0
61    #<MUSIC #x1AA0E985>  0
62    #<MUSIC #x1AA0EC51>  0
63    #<MUSIC #x1AA0EE05>  0
64    #<MUSIC #x1AA0F009>  0
65    #<MUSIC #x1AA0F34D>  0
66    #<MUSIC #x1AA0F641>  0
67    #<MUSIC #x1AA0F8E5>  0
68    #<MUSIC #x1AA0FBB1>  0
69    #<MUSIC #x1AA0FEF5>  0
70    #<MUSIC #x1AA10121>  0
71    #<MUSIC #x1AA104B5>  0
72    #<MUSIC #x1AA10641>  0
73    #<MUSIC #x1AA10935>  0
74    #<MUSIC #x1AA10BD9>  0
75    #<MUSIC #x1AA10DDD>  0
76    #<MUSIC #x1AA10F91>  0
77    #<MUSIC #x1AA11235>  0
78    #<MUSIC #x1AA11411>  0
79    #<MUSIC #x1AA117A5>  0
80    #<MUSIC #x1AA119A9>  0
81    #<MUSIC #x1AA11C75>  0
82    #<MUSIC #x1AA11D89>  0
83    #<MUSIC #x1AA11FB5>  0
84    #<MUSIC #x1AA12259>  0
85    #<MUSIC #x1AA1245D>  0
86    #<MUSIC #x1AA126B1>  0
87    #<MUSIC #x1AA12905>  0
88    #<MUSIC #x1AA12A91>  0
89    #<MUSIC #x1AA12C95>  0
90    #<MUSIC #x1AA12F61>  0
91    #<MUSIC #x1AA13115>  0
92    #<MUSIC #x1AA13369>  0
93    #<MUSIC #x1AA134F5>  0
94    #<MUSIC #x1AA13681>  0
95    #<MUSIC #x1AA138AD>  0
96    #<MUSIC #x1AA13C41>  0
97    #<MUSIC #x1AA13F0D>  0
98    #<MUSIC #x1AA14099>  0
99    #<MUSIC #x1AA14315>  0
```

```
100   #<MUSIC #x1AA145C5>  0

Average fitness = 0.0

NIL
[3]>
```

Demo Code

```
; Demo to generate an initial population.
( defmethod population-demo ( &aux p )
    ( setf p ( initial-population ) )
    ( display p )
    ( format t "Average fitness = ~A~%~%" ( average p ) )
)
```

Code

```
; Method to generate a music sample given an individual number. Used for
the initial
; population creation.
( defmethod generate-music-sample ( i-num )
    ( setf melody1-notes ( generate-melody1 ) )
    ( setf melody2-notes ( generate-melody2 melody1-notes ) )
    ( setf melody3-notes ( generate-bassline ) )

    ( make-instance 'music
        :melody1 melody1-notes
        :melody2 melody2-notes
        :melody3 melody3-notes
        :str-representation ""
        :rank 0
        :num i-num
    )
)
```

```
; Global variable for the size of a population.
( defconstant *population-size* 100 )

; Global variable for the size of selection.
( defconstant *selection-size* 4 )

; Population Class -- consists of a list of music individuals and
generation number.
( defclass population ()
    (
        ( individuals :accessor population-individuals :initarg
:individuals )
        ( generation :accessor population-generation :initform 0 )
    )
)

; Calculates the size of a population.
( defmethod size ( ( p population ) )
    ( length ( population-indiviudals p ) )
)

; Displays the population in terms of salient information for the
developer.
; (all melodies are not displayed -> instead individual #,
; music object, and fitness are shown )
( defmethod display ( ( p population ) )
    ( terpri ) ( terpri )
    ( princ "Generation " )
    ( prin1 ( population-generation  p ) )
    ( princ " population ..." )
    ( terpri ) ( terpri )
    ( dolist ( i ( population-individuals p ) )

        ( prin1 ( music-num i ) )
        ( princ ( filler ( music-num i ) ) )

        ( prin1 i  )

        ( princ "  " )
        ( prin1 ( music-rank i ) )
```

```
        ( princ ( filler ( music-rank i ) ) )
         ( terpri )


    )
    ( terpri )
)

; Helper method to create space for individuals in population display
method.
( defmethod filler ( ( n number ) )
    (cond
        ( ( < n 10 ) "     " )
        ( ( < n 100 ) "    " )
        ( ( < n 1000 ) "   " )
        ( ( < n 10000 ) "  " )
        ( ( < n 100000 ) " " )
    )

)

; Method to generate the initial population based on a population size.
( defmethod initial-population ( &aux individuals )
    ( setf individuals () )
    ( dotimes ( i *population-size* )
        ( push ( generate-music-sample ( + i 1 ) ) individuals )
    )
    ( make-instance 'population :individuals ( reverse individuals ) )
)

; Method to calculate the average fitness of a population.
( defmethod average ( ( p population ) &aux ( total 0 ) )
    ( setf indiv-list ( population-individuals p ) )
    ( setf total ( sum ( mapcar #'music-rank indiv-list ) ) )
    ( float ( / total *population-size* ) )
)

; Method to calculate and display the highest-ranked music individual in a
list of
; music individuals.
```

```
( defmethod most-fit-individual ( ( l list ) &aux max-value max-individual
)
    ( setf max-individual ( max-val l 0 ) )
    ( setf max-value ( music-rank max-individual ) )
    max-individual
)

; Method to calculate the maximum value given a list.
( defmethod max-val ( ( l list ) current-max )
    (cond
        (( null l )
            current-max
        )
        (( or ( equal current-max 0 ) ( > ( music-rank ( car l ) ) ( 
music-rank current-max ) ) )
            ( max-val ( cdr l ) ( car l ) )
        )
        (t
            ( max-val ( cdr l ) current-max )
        )
    )
)


; Demo to generate an initial population.
( defmethod population-demo ( &aux p )
    ( setf p ( initial-population ) )
    ( display p )
    ( format t "Average fitness = ~A~%~%" ( average p ) )
)
```