# Short Duration Project 1: Hollow Priority Queue

## 1. Circle size and screen layout

The object I choose is circle. The colors I use are red, green and blue.

The screen size is 1920 * 980, which fits my screen very well. The painting area is 1550 * 980, other space is used for buttons panel and show panel. Button panel contains all buttons needed, such as Start button, 2 buttons for the user to choose order (Large to small, and Small to large), Next color/size buttons, and Finish button. Show panel prints the next priority circle in each heap. It also shows the initial circles for the user to choose and "seed" screen. Because this area is small, circles printed here are shrunken, to about 1/3 of their real sizes (diameters).

The diameter of circle is between 10 to 500. Because the screen height is 980, I set the maximum diameter 500 in order not to paint too large circle. For the minimum limit, I have tried 5, 10 and 20. The circle whose diameter is 5 actually looks more like a square on my screen, so I think 10 is better.

## 2. Overall classes hierarchy and framework

There are 15 classes, including one driver class (PanelDriver), one class CirclePanel, and one interface (ScreenCircleSize). CirclePanel holds all panels and listeners, and regulates heaps, record/track list and control visualization. It is explained in Section 6.

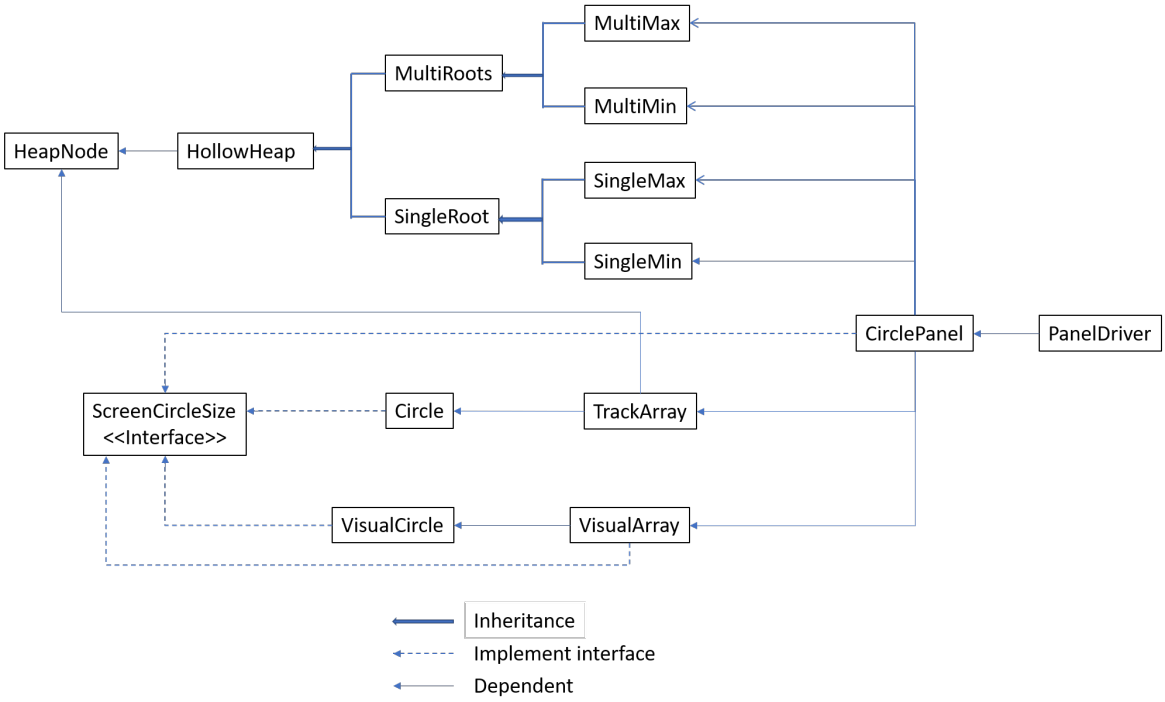
The internal data structure, hollow heap, has 8 classes, HeapNode, HollowHeap, MultiRoots, SingleRoot, MultiMax, MultiMin, SingleMax and SingleMin. HeapNode, HollowHeap, MultiRoots and SingleRoot are abstract classes. Details are in Section 3.

The track/record part, which stores the circle objects and their "locations" in heaps, includes 2 classes, Circle and TrackArray. TrackArray has a private class trackNode. This trackNode has 3 attributes, a pointer pointing to the Circle object, a pointer pointing to a HeapNode resides in color heap and represents this object, and a pointer pointing to a HeapNode stored in size heap. TrackArray is an array of trackNodes. When a Circle object is generated, 2 HeapNodes are generated too and reside in color and size heaps. A trackNode is generated to keep records of this circle, this color heap node and size heap node. Details are in Section 4.

The data structure for visualization has 2 classes, VisualCircle and VisualArray. VisualArray represents the whole screen, so it is a 2-D VisualCircle array whose size is 980 * 1550 (row * column). This is explained in Section 5.

The framework shown in *Fig 1* is an overall class hierarchy. Because each class has a lot of variables and methods, if they are all included, the figure would be too large to be shown in one page and too complex. So, a brief and overall framework is shown here and I will explain each part in Section 3, 4 and 5, with sub-hierarchy and details, respectively.

***Fig 1.***




## 3. Hollow heap structure.

The internal data structure has 8 classes.

HeapNode is the basic component of a heap. It has 5 attributes, size, rank, hollow (boolean), child (a HeapNode list), and the arrayLocation which is its TrackArray index. Size is the key value to organize the heap.

HollowHeap is an abstract class, which is a set of HeapNode. It has 2 child classes, MultiRoots and Single root, which are abstract class too. MultiRoots has 2 child classes, MultiMax and MultiMin. SingleRoot has 2 child classes too, SingleMax and SingleMin. *Fig 2* shows its class hierarchy.

Three important functions are: delete, deleteNext (deleteMin or deleteMax) and link function.

I will use min heap as the examples to explain. Following is part of the code of linkMin().

```
int s = 10;      //Our program has no chance to have a node whose rank is greater than 10.
HeapNode[] temp = new HeapNode[s];       // The index equals to node rank.
HeapNode current;
while (!root.isEmpty())
{
    current = root.removeFirst();
    while (temp[current.rank] != null)   //Another node has the same rank.
    {
        HeapNode heap2 = temp[current.rank];
        temp[current.rank] = null;
        if (current.compareTo(heap2) < 0)         //Combine them together to form a new subtree,
        {                                         //make smaller one the parent,
          current.child.addFirst(heap2);          //and increment rank by 1.
        } else
        {
          heap2.child.addFirst(current);
```

```
            current = heap2;
          }
          current.rank++;
          }
          temp[current.rank] = current;
      }
      for (int i = 0; i < s; i++)          //Add all ranked subtrees to list root.
      {
        if (temp[i] != null)
        root.add(temp[i]);
       }
}
```

For SingleMin, deleteNext() returns min node, adds all its children to a list (called root here, so root in single root heap represents a node list, not the real root node. The real root node is called min or max, respectively). Then remove all hollow nodes in list root, add their children to root too. Link all root nodes (via appropriate ranking), then find out the new/current min node, and melds (move all left nodes in root list and add them as the children of min node ignoring their ranks). Following is the code for deleteNext() in class SingleRoot.

```
public HeapNode deleteMin()
{
  HeapNode result = min;
  root = result.child;   //Move all children of min node to list root.
  result.child = new LinkedList<HeapNode>();

  removeHollowRoot();
  linkMin();             //Rank and link all root nodes
  findMin();             //Find new min node. This function also melds.
  return result;
}
```

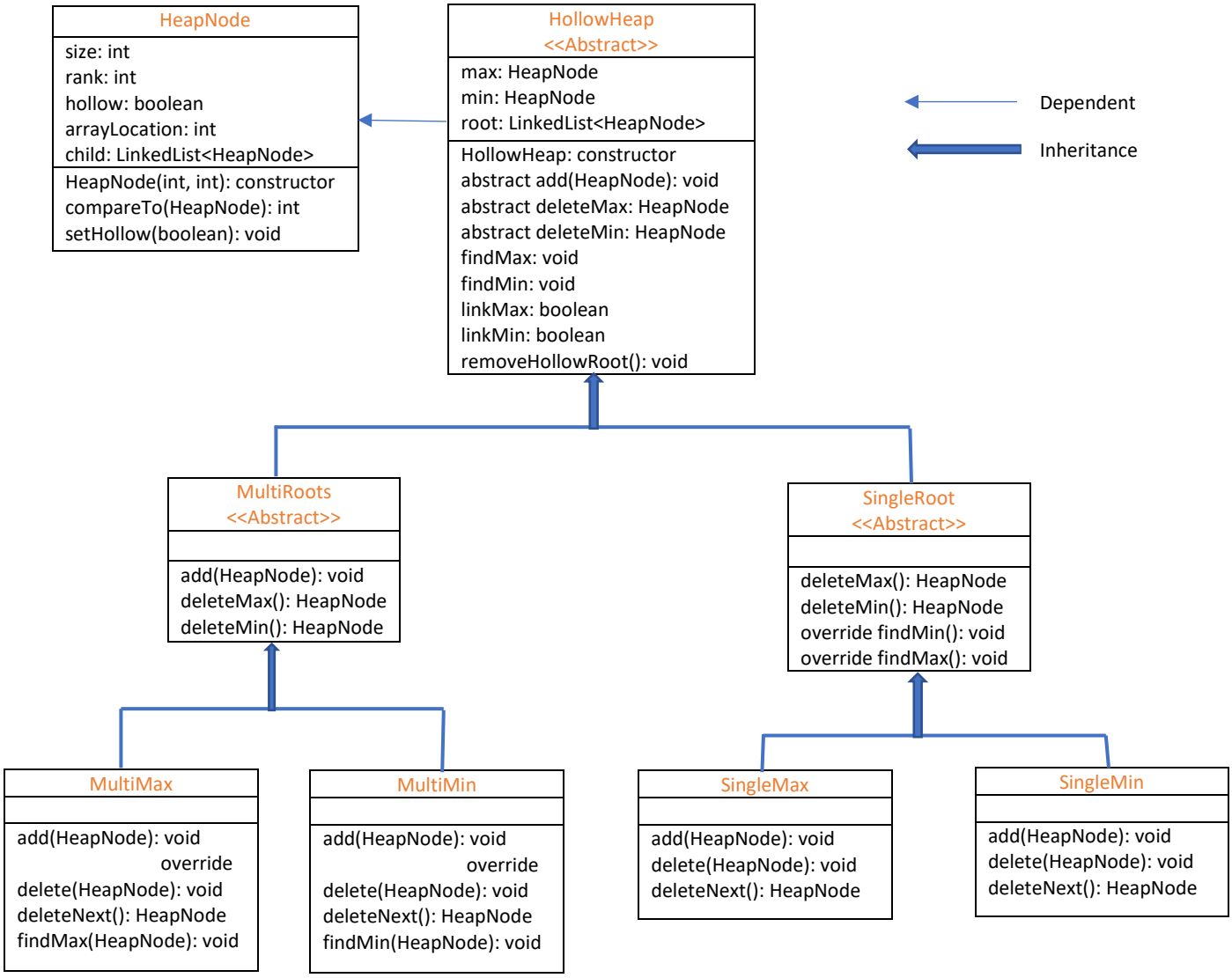
For MultiMin, a min heap with multi roots, deleteNext() returns min node, adds all its children to root, remove all hollow root nodes, link, and finds current min node. It does not need to meld.

When calling deleteMin to delete the min of a color heap, a HeapNode min is returned. This node has its own arrayLocation. Go to the TrackArray, find the trackNode stored in TrackArray[arrayLocation]. Then the object circle and a pointer pointing to size heap node can be easily found. Pass this size heap node pointer to delete(HeapNode) to remove it from the size heap. Following is part of the code for delete(HeapNode) in SingleMin class.  delete(HeapNode) in MultiMin uses similar method.

```
public void delete(HeapNode n)   //Delete a given node
{
  n.hollow = true;       //Make it hollow
  if (min == n)
  {
    deleteNext();
  } else
  {
    if (n.rank > 2)      //If its rank is greater than 2, move some children and meld with min.
    {
        for (int i = 0; i < n.child.size(); i++)
        {
          LinkedList<HeapNode> childNew = n.child;
          n.child = new LinkedList<HeapNode>();
          while (!childNew.isEmpty())
          {
            HeapNode t = childNew.removeFirst();
            if (t.rank != n.rank - 1 && t.rank != n.rank - 2)
            {
                min.child.add(t);
```

```
                } else
                {
                    n.child.add(t);
                }
            }
        }
    }
}
```

***Fig 2.***



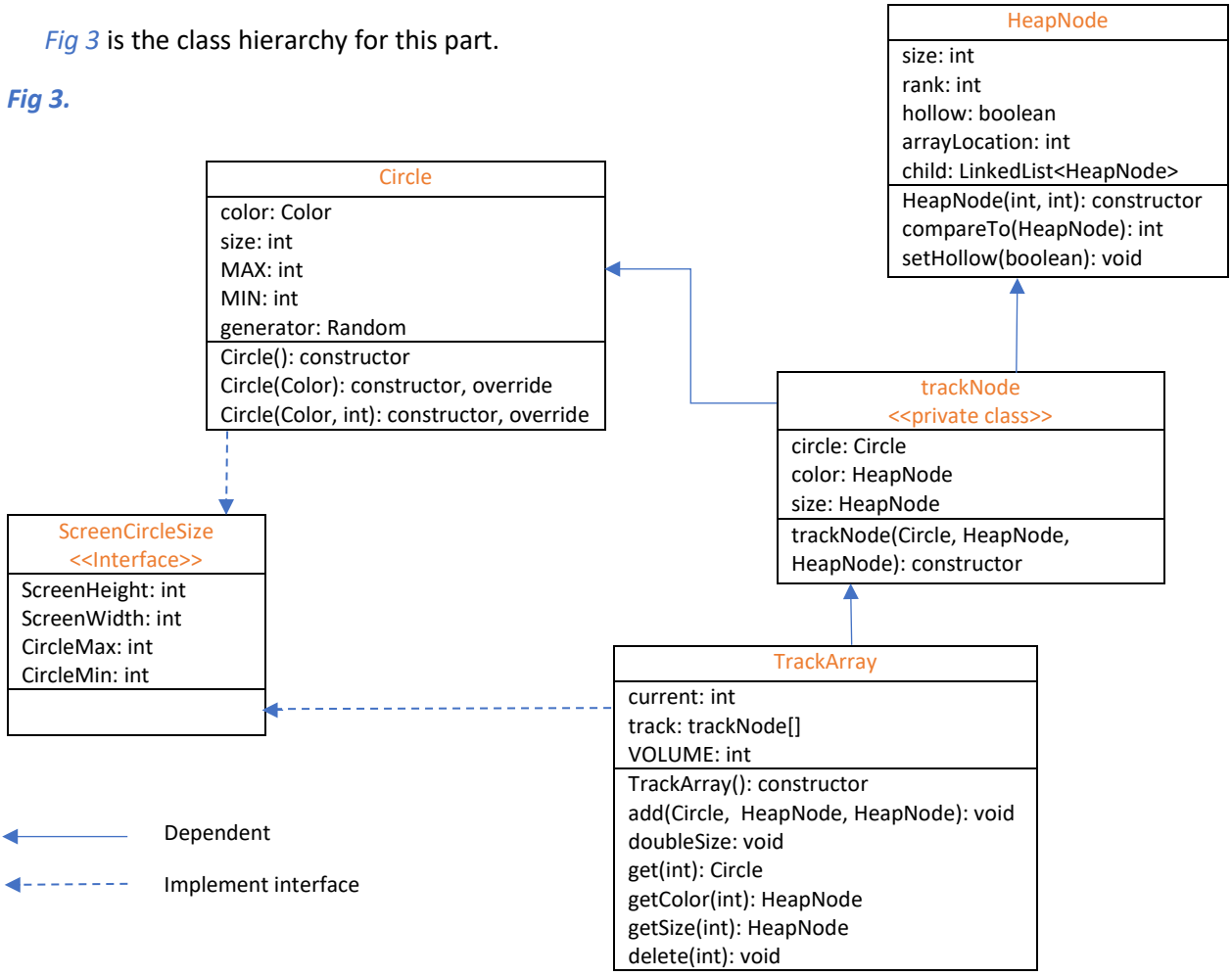

## 4. Storing circle and tracking its locations in heaps

The class Circle defines the user object circle and implements interface ScreenCircleSize. Attribute size is its diameter. An overridden constructor, Circle(Color, int), needs an integer augment, which is the largest circle diameter that the whole painting screen can hold currently. Using this constructor, we can just generate circles whose size is smaller than this limitation and thus can be successfully put into the screen. This important diameter limitation is found in class VisualArray, which is explained in Section 5.

The TrackArray class has a private class trackNode which is explained in Section 2.

*Fig 3* is the class hierarchy for this part.

***Fig 3.***




## 5. Visualization

It has 2 classes, VisualCircle and VisualArray.

VisualArray is a 2-D array which represents the whole painting screen. Each entry is a VisualCircle object, that denotes a point on screen.

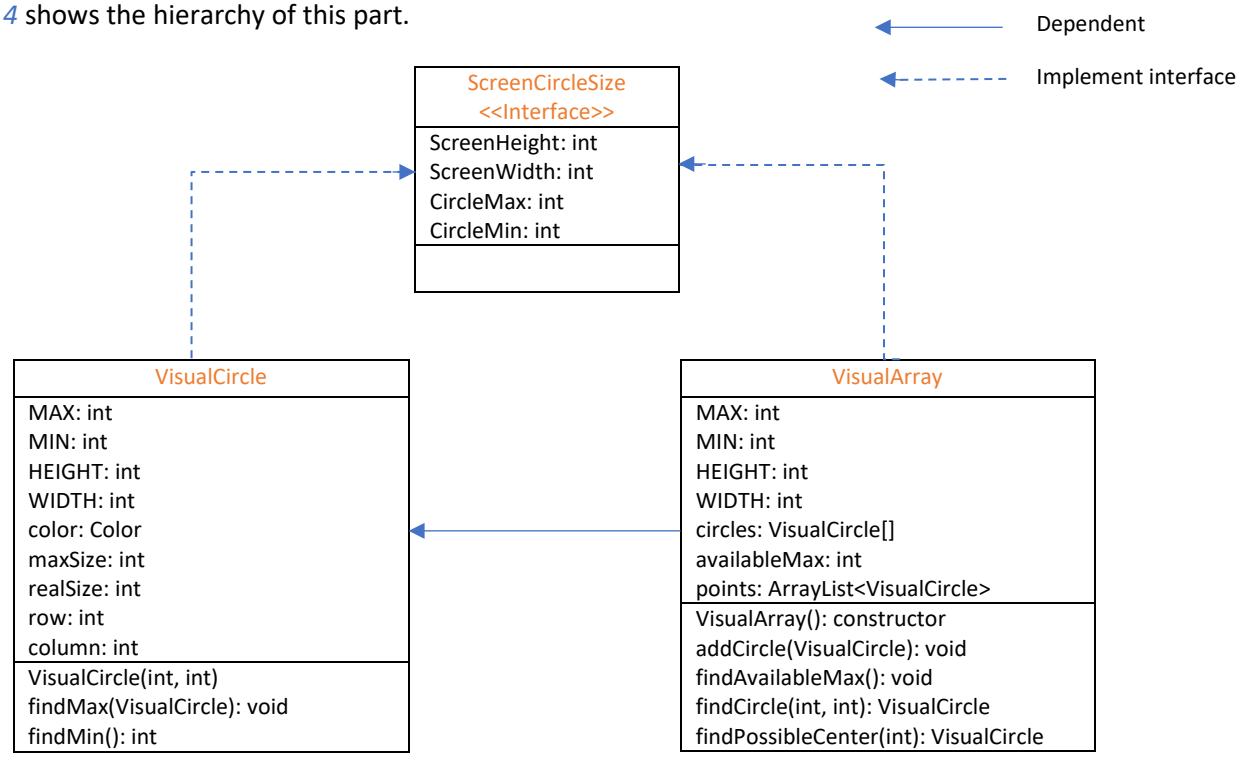
VisualCircle has 2 location attributes, row and column, keeps its location on screen. It has other 2 integer attributes, maxSize and realSize. For example, a VisualCircle p, row = 20, column = 20. Then its maxSize = 20 * 2 = 40, which means that the largest circle whose center is p cannot have its diameter bigger than 40. If we really draw a circle using p as the center, then the realSize is the diameter of this circle. So, when p is initialized, its maxSize = 40 and realSize = 0. Attribute color is the color of this circle.

VisualCircle array, let's call it vCircles, has an attribute points, which keep the list of all existing circles. It also has an important attribute availableMax, which represents the max diameter that the screen can fit by now. This value is used to set limitation of new generated objects, as I mentioned in Section 4. Another important method is findPossibleCenter(int s). It will screen all entries in array to first find a center which can just fit this size, in other words, its maxSize equals s. If this finding fails, it then

tries to find a bigger space. If this attempt fails too, s is too big for current screen, return null. This failure and returned null will give findPossibleCenter(int s) function caller some information, and caller will then remove too large circles which are not possible to be added. This will be explained further in Section 6.

*Fig 4* shows the hierarchy of this part.

***Fig 4.***




## 6. CirclePanel class

This class is very big. All used panels, label, buttons and listeners are defined in this class. Because all panels and methods have a lot of shared variables, so it is difficult and inconvenient if we split this big class into several classes, so I leave it a large class with about 900 lines of code and comments.

The CirclePanel object is a panel which contains 3 sub-panels, PaintPanel, ButtonPanel and ShowPanel. When CirclePanel object is initialized, a TrackArray track and a VisualArray vCircles are created and initialized. Three panels are added then.

PaintPanel, represents the painting area/screen. It has a listener to respond to initial circles locations choosing. It also has a counter which counts the total number of added circles.

ButtonPanel, holds all buttons.

Start button is used to generate initial circles to “seed” screen. I generate 5 circles with randomly chosen color and size. Circles are printed in ShowPanel, with shrunken size. A user can choose a circle by clicking on it to add it. The location is chosen by clicking on PaintPanel. The user can choose the sequence to add them to any locations. If the location the user chooses is too small to fit this circle, a warning is given. These 5 circles are added to VisualArray vCircles.

After program is initialized, the user then can choose an order for heap, either larger to smaller or smaller to larger by press button. Let me assume the user chooses button “Larger to smaller”. 4 max heaps are initialized by call method initializeMax(), including 3 color heaps, redHeap, greenHeap and blueHeap which are multi root heaps, and 1 size heap, sizeHeap which is a SingleRoot heap. I generate 10 circles for each color heap, thus size heap has 30 circles total. Following is the code used for generating 10 red circles. A new circle is generated and added to track array. 2 heapnodes are generated and added to either color heap or size heap.

```
Circle c;
HeapNode n, m;
int location, size;
for (int i = 0; i < 10; i++)
{
        c = new Circle(Color.RED, availableMax);
        size = c.size;
        location = track.current;
        n = new HeapNode(size, location);
        ((MultiMax) redHeap).add(n);
        m = new HeapNode(size, location);
        ((SingleMax) sizeHeap).add(m);
        track.add(c, n, m);
}
```

After initializing, the next priority circle of each heap is shown in ShowPanel, also with shrunken size. The user can choose the next one he wants to add by clicking buttons, “Next Red”, “Next Blue”, “Next Green” or “Next Size”.

Assume “Next Red” is chosen here. Button listener will call method findColorPop(Color.red) to pop the max of redHeap. This function also calls another function, checkEmptyHeap(color, availableMax) to check if redHeap is empty after this pop. I choose to add more circles/nodes to heap when heap is empty. This can give the user some very small circles to choose. If we choose to add more nodes to heap when, for example, there are 3 nodes left in redHeap, the user is always given bigger circles, because it is very likely that new added circles are bigger than old ones.

If this checkEmptyHeap(color, availableMax) function finds redHeap is empty, it will add 10 more red circles to track array, 10 nodes reside in redHeap and 10 nodes reside in sizeHeap. The circle diameter max limitation is the availbleMax, by which we avoid to generate too large circle which cannot be added.

Then VisualArray findPossibleCenter(int s) function is called to find a space for this pop-out circle. Here is part of code used in this step.

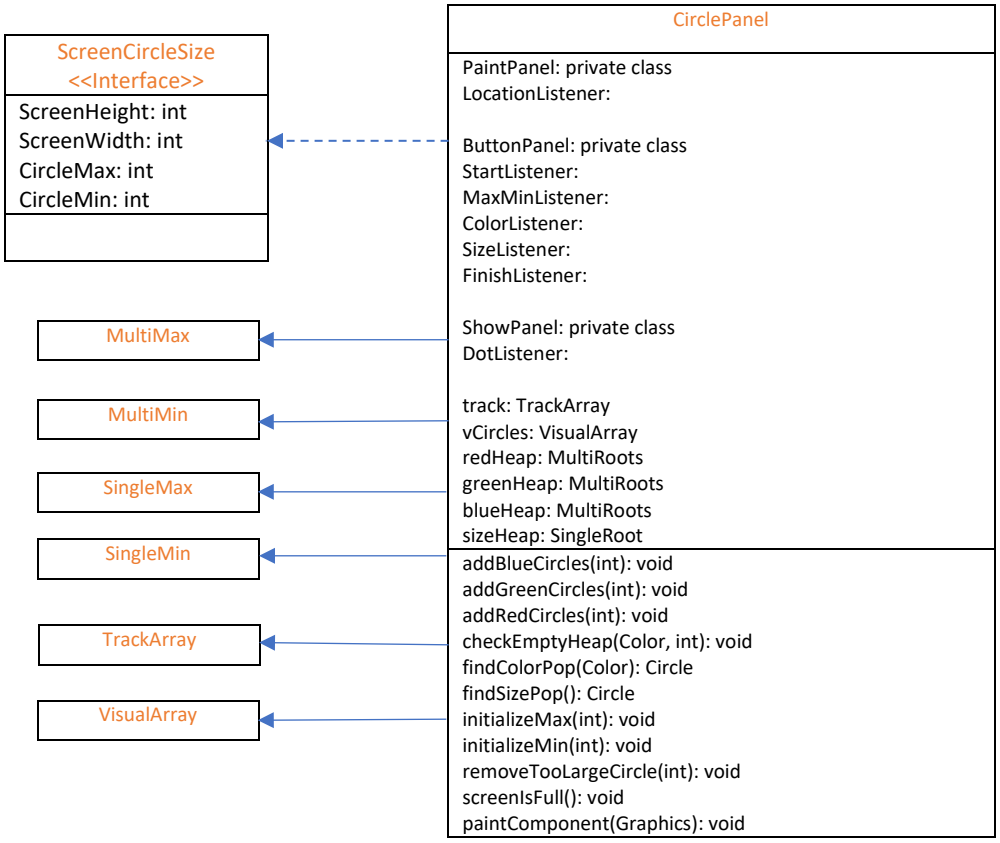
```
pop: the max circle returned by findColorPop(Color.red). The findColorPop(Color.red) also remove this max
from redHeap, sizeHeap and track array.
int size = pop.size;
Color c = pop.color;
VisualCircle target = vCircles.findPossibleCenter(size);
int availableMax = vCircles.availableMax;
if (availableMax == 0)  //The screen cannot hold even one more smallest circle now
{
        screenIsFull();
}
while (target == null)  //This one is big, but still can add some smaller circles
{
        pop = findColorPop(Color.red);  //Continue pop until we find a circle that small enough
        size = pop.size;
        c = pop.color;
```

```
            target = vCircles.findPossibleCenter(size);
            availableMax = vCircles.availableMax;
            if (availableMax == 0)
            {
                    screenIsFull();
            }
    }
    target.color = c;
    vCircles.addCircle(target);  //Add this circle to VisualArray
    vCircles.findAvailableMax();
    availableMax = vCircles.availableMax;
    removeTooLargeCircle(availableMax); //Remove too large circles from all heaps.
```

The user can keep adding until screen is full. A message will be given when he needs to stop. Or he can choose exit at any time by press button “Finish”.

Following is the class hierarchy. Some classes details are shown in Fig 2, 3 or 4, so I just show class names here indicating relationship between them and CirclePanle class.

***Fig 5.***




7. I designed and coded the project independently. And I executed the program for about 40 to 50 times and all experiments perform well without any problem.