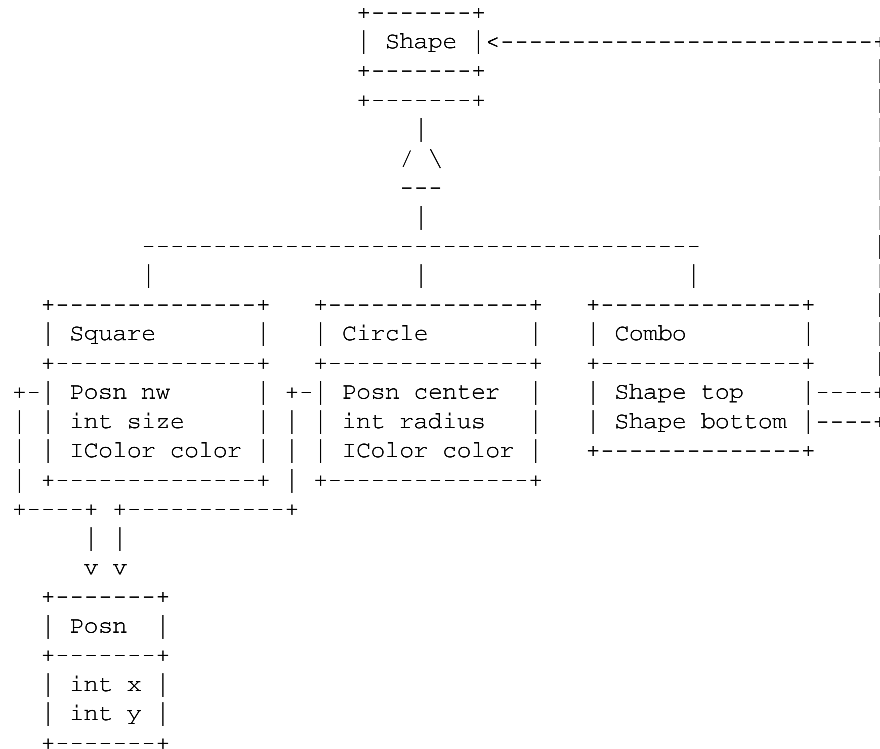
4 Designing Methods – Part 2

Methods for Unions and Self-Referential Data

We will focus on geometric shapes - a circle, a square, and a shape that is a combination of two shapes, the top and the bottom one. Recall the data definition as given by the class diagram:



1. Design the method *totalArea* that computes the total area of a shape. For the shape that consists of two components add the areas - as if you were measuring how much paint is needed to paint all the components.

   You will need to use math functions, such as square root. The following example shows how you can use the math function, and how to test *double*s for equality. (You can only make sure they are different only within some given tolerance.)

```
class Foo{
  double x;

  Foo(double x){
    this.x = x;}

  double squareRoot(){
    return Math.sqrt(this.x);}
}

class Examples {
  Examples () {}

  Foo f = new Foo(16.0);

  boolean testSquared =
    check this.f.squareRoot() expect 4.0 within 0.01;
}
```

2. Design the method *moveBy* that produces a new shape moved by the given distance in the vertical and horizontal direction.

3. Design the method *isWithin* that determines whether the given point is within this shape.

4. Of course, we would like to draw the shapes on a canvas. Design the method *drawShape* that draws *this* shape on the given *Canvas*. The following code (that can be written within the *Examples* class) shows how you can draw one circle:

```
import draw.*;
import colors.*;
import geometry.*;

class Examples{
  Examples() {}

  Canvas c = new Canvas(200, 200);

  boolean makeDrawing =
    this.c.show() &&
    this.c.drawDisk(new Posn(100, 150), 50, new Red());
}
```

The three *import* statements on the top indicate that we are using the code programmed by someone else and available in the libraries named *draw*, *colors*, and *geometry*. Open the *Help Desk* and look under the *Teachpacks* for the teachpacks for *How to Design Classes* to find out more about the drawing and the *Canvas*.

4.1 Shark and Fish

Fish swim across the screen from right to left. There may be one fish, or a whole school of fish, or none at a time. A hungry shark swims in place at the left side, moving only up and down, controlled by the `"up"` and `"down"` keys. It tries to catch and eat the fish. It gets bigger with each fish it eats, it keeps getting hungrier and hungrier as the time goes on between the catches. It dies of starvation, if not fed in time.

Once you design all methods for these classes, you can see the animation and play the game by importing the necessary library classes, changing to the *ProfessorJ Intermediate Language* and adding the phrase *extends World* after the *class OceanWorld* beginning of the *OceanWorld* definition.

Design this game.

The Shark class

1. Design the class *Shark* to represent a shark swimming up and down.
2. Design the method *onKeyEvent* that consumes a *String* that represents the key a user pressed and produces a new *Shark* at the location determined by the key the user pressed. If the user pressed the `"up"` key the new *Shark* has moved up, if the user pressed the `"down"` key, the new *Shark* has moved down. If the user pressed any other key, the method produces the same *Shark* that invoked the method.
3. Design the method *draw* that consumes a *Canvas* and draws the shark on the *Canvas* in its current position.
4. Design the method *onTick* that produces a *Shark* a bit hungrier than before, until the shark is declared dead. Think of the good way to represent the *liveness* of the *Shark*.

The Fish

1. Design the class *Fish* to represent one fish swimming in the ocean.

2. Design the method *onTick* that produces a fish moved to the left by a fixed distance.

   Note: Later we may add some random movement up and down, or even randomly change the speed of swimming. However, when designing a complex program, it is the best to get down the basic functionality and add more features later. This is called *iterative refinement*. (You may have seen it already in the previous semester.)

3. Design the method *escaped* that produces a boolean value *true* if the fish swam outside of the visible *Canvas*, i.e. its horizontal coordinate is negative.

4. Design the method *draw* that that consumes a *Canvas* and draws the fish on the *Canvas* in its current position.

The Ocean World

We now design the whole scene: the fish and the shark swimming in the blue ocean, the shark eating the fish, all swimming along.

1. Design the class *OceanWorld* that consists of one *Shark* and one *Fish*. We also specify the size of the ocean scene (that will become the size of the *Canvas*.

   (Note: Again, we choose to handle the simple case first: one fish only.)

2. Design the method *sharkFoundFish* that determines whether a shark has found a fish. In which class should this method be defined? Can it be useful if we change the problem to include a whole school of fish?

3. Design the method *eatFish* in the class *Shark* that (no pun intended) consumes the given *Fish* and produces a fatter *Shark*.

4. Design the method *onTick* that produces a new *OceanWorld* as follows:

   - An escaped fish is replaced by a new one that appears at a random height, colse to the right edge of the Canvas.
   - If the shark gets close to the fish, it eats the fish, gets fatter and a new fish appears at a random height on the right hand side.

- If the shark dies of starvation, the method produces *endOfWorld* with the message announcing the shark's demise.
- If none of the above applies, the fish moves as given by its *onTick* method, and the shark starves as given by its *onTick* method.

Make sure you test this method carefully before running any simulation.

5. Design the method *onKeyEvent* that produces a new *OceanWorld* with the same *Fish* as before and the shark moved in response to the key pressed by the user in the manner already determined by the *onKeyEvent* method in the *Shark* class.
6. Design the method *draw* that draws the ocean scene: the blue ocean, the shark, and the fish.
7. With just a little help you can now play the game. The TAs will tell you how.

Note: The following code can be used to generate a random height for the new fish:

```
// produce a random initial height of the fish
int randomHeight(){
  return new Random().nextInt() %
}
```

When you add this method you must also add to the beginning of the program the following *import* statemet:

```
import java.util.Random;
```