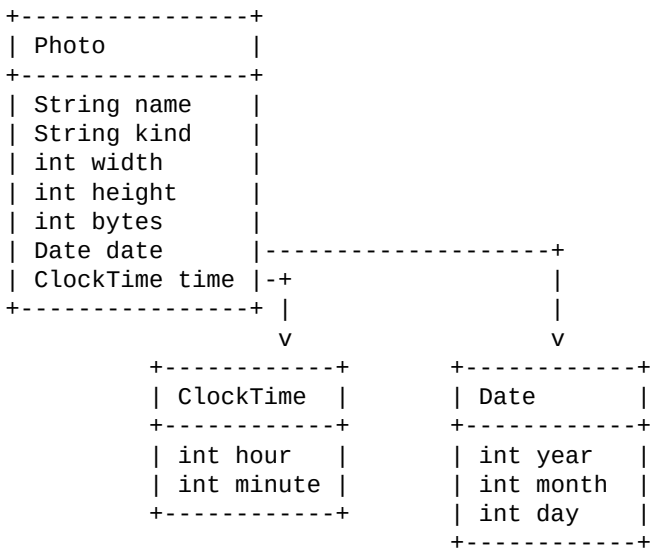
3 Designing Methods — Part 1

Methods for Simple Classes and Classes with Containment

For this series of exercises use the classes from Lab 1 that represented photo images and included the date and time information. Recall the class diagram:



Recall some examples of the information we wish to represent:

- Picture of a river (jpeg) that is 3456 pixels wide and 2304 pixels high, using up 3,614,571 bytes

  — taken on September 23, 2007 at 9:50 am.

- Picture of a mountain (jpeg) that is 2448 pixels wide and 3264 pixels high, using up 1,276,114 bytes

  — taken on November 11, 2007 at 11:30 am.

- Picture of a group of people (gif) that is 545 pixels wide and 641 pixels high, using up 13,760 bytes

  — taken on November 11, 2007 at 9:30 pm.

- Picture of a plt icon (bmp) that is 16 pixels wide and 16 pixels high, using up 1334 bytes

  — taken on September 23, 2007 at 11:30 pm.

You can work with these data definitions, or, if you used different names in the previous lab, you can use the code from that lab, as long as you can represent all the information given in the examples.
Note: Make sure you understand the data definitions, can translate these examples to data, and converesely, translate any instance of data defined for these classes into the information the data represents.

Design Recipe for a simple method definition

Recall from the lectures that in a class based language every method is defined in a class that is most relevant, it is then invoked by the instance of that class, and *this* instance becomes the first argument for the method.

Below is an example of the design of a method that computes the number of pixels in a photo image:

- Step 1: Problem analysis and data definition.

  The method deals with *Photo*s and so it needs to be defined in the class *Photo*. Each instance of a *Photo* has all the information we need to solve the problem - we do not need any additional data to be given. The result is an integer.

  We will use the following data in our examples. For your work add at least one more instance of each class.

```
// Examples for the class ClockTime
ClockTime ct1 = new ClockTime(21, 50);
ClockTime ct2 = new ClockTime(11, 30);
ClockTime ct3 = new ClockTime(9, 50);


// Examples for the class Date
Date d1 = new Date(2007, 9, 23);
Date d2 = new Date(2007, 11, 7);
Date d3 = new Date(2007, 9, 25);


// Examples for the class Photo
Photo river = new Photo("River", "jpeg", 3456, 2304, 3614571,
                        this.d1, this.ct3);
Photo mountain = new Photo("Mountain", "jpeg", 2448, 3264, 1276114,
                           this.d2, this.ct2);
Photo people = new Photo("People", "gif", 545, 641, 13760,
                         this.d2, this.ct1);
Photo icon = new Photo("PLTicon", "bmp", 16, 16, 1334,
                       this.d1, this.ct2);
```

- Step 2: The purpose statement and the header.

  // to compute the number of pixels in this photo
  int pixels(){...}

- Step 3: Examples.

  people.pixels() −−−> 349345
  icon.pixels() −−−> 256

- Step 4: The template.

  int pixels(){
  ... *this.name* ... −−− *String*
  ... *this.kind* ... −−− *String*
  ... *this.width* ... −−− *int*
  ... *this.height* ... −−− *int*
  ... *this.bytes* ... −−− *int*
  ... *this.date* ... −−− *Date*
  ... *this.time* ... −−− *ClockTime*

  We will only need *this.width* and *this.height*.

- Step 5: The method body.

  // to compute the number of pixels in this photo
  int pixels(){
  return this.width ∗ *this.height*;
  }

- Step 6: Tests.

  ProfessorJ provides a special way of running the tests. A *check* expression

  check *test method invocation* expect *expected test result*

  produces the test result as a *boolean* value and all test results are reported in a separate display. The following code:

  // Tests for the method pixels:
  boolean testPixels = (*check this.people.pixels*() *expect* 349345) *&&*
  (*check this.icon.pixels*() *expect* 256);

shows the tests for our method.

1. Design the method *timeToDownload* that determines how long it will take to download this image, if we know the number of bytes we can download in one second.

2. Design the method *takenBefore* that determines whether this picture was taken before another picture.

   Remember: One task, one method. Make each class responsible for its data.

3. In the class *ClockTime* design the method that advances the time by the given number of minutes. Use helper methods as needed.

4 Quiz

5 Designing Methods – Part 2

Methods for Unions

For this part use the data for camera shots (photos or videos) from the previous lab. When designing methods for unions, we first need to define the purpose and the header in the *interface* that represents the union, then work on designing the method body in each class that implements the interface *following the desing recipe*.

1. Design the method *timeToDownload* that determines how long it will take to download this shot, if we know the number of bytes we can download in one second.

2. Design the method *takenOn* that determines whether this shot has been taken on a given day.

6 Designing Methods — Part 3

We continue with the the theme of the photo images. Our goal is to design methods that answer questions about list of images and manipulate these lists.

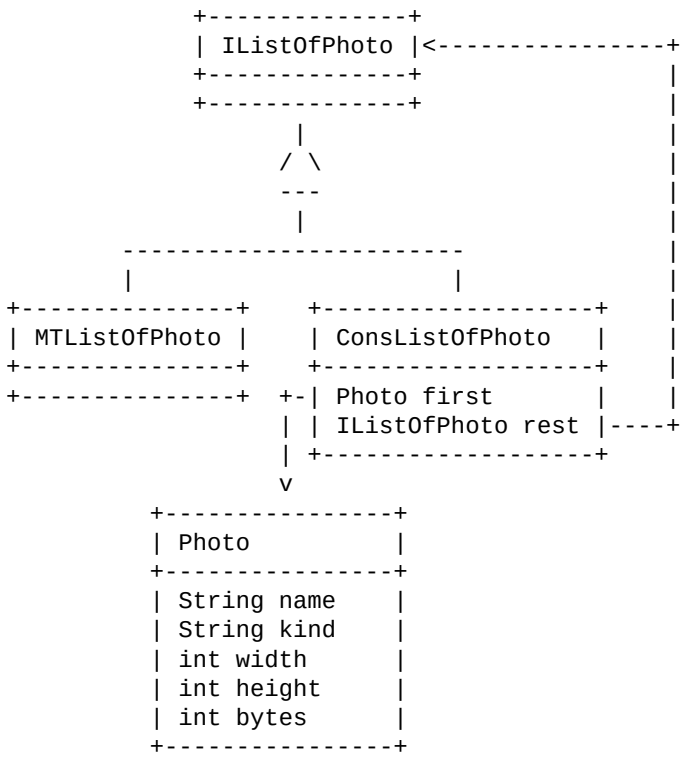
We will also work with the geometric shapes and learn to draw their images on the *Canvas*.

6.1 Methods for Self-Referential Data

In the previous lab you designed a list of photo images. We first design the method that counts the images in our list. (We use the simpler version of the class *Photo* that was defined in Lab 1.)

Note: Of course, you will quickly realize that this method will look the same regardless of what are the pieces of data contained in the list. We will address that issue later on, once we are comfortable with dealing with lists that contain specific items.

Recall that the class diagram for a list of photo images looked as follows:



Design Recipe for a method definition for unions of data

Below is an example of the design of a method that counts the number of pictures in a list of photo images.

The method deals with *IListOfPhotos*. We have an interface *IListOfPhotos* and two classes that implement the interface, *MTListOfPhotos* and *Con-*

sListOfPhotos. When the DESIGN RECIPE calls for the method purpose statement and the header, we include the purpose statement and the header in the interface *IListOfPhotos* and in all the classes that implement the interface.

Including the method header in the interface serves as a contract that requires that all classes that implement the interface define the method with this header. As the result, the method can be invoked by any instance of a class that implement the interface - without the need for us to distinguish what is the defined type of the object.

We can now proceed with the DESIGN RECIPE.

- Step 1: Problem analysis and data definition.

  The only piece of data needed to count the number of elements in a list is the list itself. The result is an integer.

  We will use the following data in our examples. For your work add at least one more instance of each class.

```
// Examples for the class Photo
Photo river = new Photo("River", "jpeg", 3456, 2304, 3614571);
Photo mountain = new Photo("Mountain", "jpeg", 2448, 3264, 1276114);
Photo people = new Photo("People", "gif", 545, 641, 13760);
Photo icon = new Photo("PLTicon", "bmp", 16, 16, 1334);

IListOfPhotos mtlist = new MTListOfPhotos();
IListOfPhotos list1 = new ConsListOfPhotos(this.river, this.mtlist);
IListOfPhotos list2 = new ConsListOfPhotos(this.mountain,
                                           new ConsListOfPhotos(this.people,
                                           new ConsListOfPhotos(this.icon, this.mtlist)));
```

- Step 2: The purpose statement and the header.

```
// to count the number of pictures in this list of photos
int count(){...}
```

  In the interface *IListOfPhotos* we write:

```
// to count the number of pictures in this list of photos
int count();
```

  indicating there is no definition for this method.

  We now have to design the method separately for each of the two classes.

- Step 3: Examples.

  We make examples for the empty list, a list with one element and a longer list:

  mtlist.count() −−−> 0
  list1.count() −−−> 1
  list2.count() −−−> 3

- Step 4: The template.

  We need to look separately at the two classes that implement the method.

  class MTListOfPhotos: The class has no member data and there is no other data available. It is clear that the method will always produce the same result, the value 0.

  We can finish the steps 4. and 5. right away — the method body becomes:

```
// to count the number of pictures in this list of photos
int count() {
  return 0;
}
```

  The template for the class *ConsListOfPhotos* includes the two fields, *this.first* and *this.rest*. However, just as in HtDP, we recognize that *this.rest* is a data of the type *IListOfPhotos* and so it can invoke the method *count* that is now under development. The template then becomes:

  class ConsListOfPhotos

```
int count(){
 ... this.first ...          --- Photo
 ... this.rest ...           --- IListOfPhotos

 ... this.rest.count() ...   --- int
```

  Recall the purpose statement for the method *count*:

```
// to count the number of pictures in this list of photos
```

That means the purpose of the method invocation *this.rest.count*() is

```
// to count the number of pictures in the rest of this list of photos
```

When designing methods for self-referential data, make sure you say out loud (or at least understand clearly) the purpose statement as applied to the self-referential method invocation.

- Step 5: The method body.

  We have already finished the method body for the class *MTListOfPhotos*. In the class *ConsListOfPhotos* the method body is:

```
// to count the number of pictures in this list of photos
int count(){
  return 1 + this.rest.count();
}
```

- Step 6: Tests.

  We can now convert our examples into tests:

```
// Tests for the method count:
boolean testPixels = (check this.mtlist.count() expect 0) &&
                     (check this.list1.count() expect 1) &&
                     (check this.list2.count() expect 3);
```

Design the methods that will help you in dealing with your photo collection:

1. Before burning a CD of your photos, you want to know what is the total size in bytes of all photos in the list of photos. The method *totalSize* should help you. Design it.

2. You now want to go over the list of photos and select only the photos in the *jpeg* format. Design the method *onlyJpeg* to help you with this task.

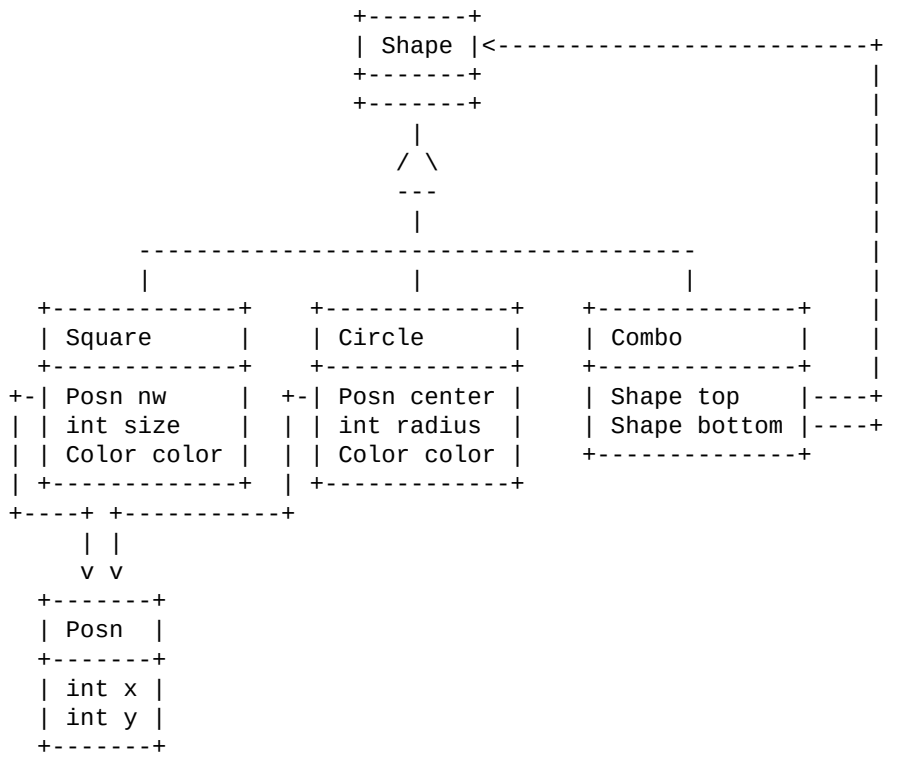
3. Finally, you want to sort the list of photos by the name of the image (as typically these are generated by your camera and represent the date and time when the photo was taken). Of course, you design the method *sortByDateTime*.

   Note: If you are running out of time, sort only by date.

More Methods for Self-Referential Data
Graphics

You may finish this part at home as a part of your Portfolio.

In the previous lab you defined classes that represent different 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 show 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*.

Save all your work — the next lab will build on the work you have done here!