5 All are equal, but some are more equal than others.

Goals:

Learn how to determine the equality of two objects in a Java program.

The definitions of all the classes are already provided. The classes include a method *translate* in the class *CartPt* and the method *move* in the remaining classes. Both methods consume the distance *dx* and *dy* by which the items should be moved or translated. Additionally, some sample data is also given. Your goal is to design the method *same* that determines whether the values of two objects are the same (according to our definition of *sameness*).

5.1 Equality of simple classes

We start with our class of CartPt. The class is defined as follows:

```
+----------------------------------+
| CartPt                           |
+----------------------------------+
| int x                            |
| int y                            |
+----------------------------------+
| CartPt translate(int dx, int dy) |
+----------------------------------+
```

The method *move* is defined as follows:

```
// translate the position of this point by the given dx and dy
CartPt translate(int dx, int dy){
  return new CartPt(this.x + dx, this.y + dy);
}
```

Our tests are designed as follows:

```
CartPt pt1 = new CartPt(20, 30);

boolean testMove = check pt1.move(-5, 8) expect new CartPt(15, 38);
```

Of course we know, that the *check* form compares the values of *pt1* and the new *CartPt* we specified as the expected result. To replace this test by our own, we need a method in the class *CartPt* that determines whether this point is the same as the given one.

```
// is this point the same as the given one?
boolean same(CartPt that){...}
```

Design this method.

5.2 Equality of classes with containment

We now want to see if two stars in our Shooting Stars program are the same. Here is the class diagram for the class *Star*:

```
+---------------------------+
| Star                      |
+---------------------------+
| CartPt loc                |
| int lifespan              |
+---------------------------+
| Star move(int dx, int dy) |
+---------------------------+
```

Design the method *same* that determines whether two stars are the same. We consider two stars to be the same if they are at the same location and have the same lifespan.

Rewrite the tests as follows (remember, as the star moves, it also decreases its lifespan):

CartPt pt1 = *new CartPt*(20, 40);
boolean testTranslate = *pt.translate*(3, −5).*same*(*new CartPt*(23, 35));

Star star = *new Star*(*this.pt1*, 9);
boolean testMove = *star.move*(3, −5).*same*(*new Star*(*new CartPt*(23, 35), 8);

5.3 Quiz

You have 10 minutes.

5.4 Equality of unions with self-reference

We will continue with the class *Star* and the classes that represent a list of *Star*s.

5.4.1 Problem analysis

We need the method *same* to compare one instance of the list of *Star*s with another instance. The argument must be of the type *ALoStars*, because it can be an empty list as well as a nonempty list.

5.4.2 Purpose and Header

```
// determine whether this list of Stars is the same as the given one
abstract boolean same(ALoStars that);
```

This method needs to be defined in every subclass of the *ALoStars* class.

5.4.3 Examples

Empty list can appear both as the instance that invokes the method (*this*) and as the argument to the method (*that*). Additionally, we need to think what happens when two lists contain the same objects, but not in the same order. It is much easier to compare two lists if the order of the items is the same in both of them. We defer till later the work on sorting the lists, and require here that the comparison succeeds only if the objects appear in the same order in both lists.

Our examples then need to address all of these possibilities:

```
// Sample data
Star s1 = new Star(new CartPt(20, 40), 10);
Star s2 = new Star(new CartPt(30, 40), 5);
Star s3 = new Star(new CartPt(10, 30), 8);
Star s4 = new Star(new CartPt(10, 50), 10);

LoStars mtstars = new MTLoStars();
LoStars list1 = new ConsLoStars(s1, mtstars);
LoStars list2 = new ConsLoStars(s2, list1);
LoStars list3 = new ConsLoStars(s3, list2);
LoStars list4 = new ConsLoStars(s4, list3);

// dealing with the empty list
mtstars.same(new MTLoStars()) --> true
mtstars.same(list1) --> false
list1.same(mtstars) --> false
list4.same(mtstars) --> false

// dealing with a list with one item
list1.same(new ConsLoStars(s1, mtstars)) --> true
list1.same(list2)) --> false
list1.same(new ConsLoStars(s2,
           new ConsLoStars(s3, mtstars))) --> false
```

```
// dealing with a list with more than one item
list4.same(new ConsLoStars(s1, mtstars)) --> false
list4.same(new ConsLoStars(s4, new ConsLoStars(s3, list2)) --> true
list4.same(new ConsLoStars(s4, new ConsLoStars(s2, mtstars)) --> false
```

5.4.4 Template

Again, there is no data in *MTLoStars*, so we only need a template for the class *ConsLoStars*:

```
... this.first ...
... this.first.same(Star ...) ...

... this.rest ...
... this.rest.same(ALoStars ...) ...
```

5.4.5 Body

As usual we try to do the simple case first.

class MTLoStars

In the class *MTLoStars* we have two pieces of data: *this* and *that*. We do not know whether *that* is an instance of *MTLoStars* or of the *ConsLoStars*.

However, we do know that *this* is an instance of *MTLoStars*. We decide to use this fact and design a helper method that consumes an argument of the type *MTLoStars* and determines whether some list is the same as the given empty list:

```
// determine whether this list is the same as given empty list
boolean sameMTLoStars(MTLoStars other)
```

and the body of our method becomes:

```
return that.sameMTLoStars(this);
```

What we see here is the reversal of the role of the two pieces of data involved in the method: the argument of the unknown type invokes the helper method, using the data of a known type (*this*) as the argument. But that means, the method *sameMTLoStars* has to be implemented for all lists - whether an empty or the constructed one. Let us finish the design of the bodies (a trivial task) - leaving the examples to you (Make sure you do them!):

```
in the class ALoStars:
// determine whether this list is the same as given MTLoStars list
abstract boolean MTLoStars(MTLoStars other);
```

```
in the class MTLoStars:
// determine whether this list is the same as given MTLoStars list
boolean sameMTLoStars(MTLoStars other){
  return true;
}

in the class ConsLoStars:
// determine whether this list is the same as given MTLoStars list
abstract boolean sameMTLoStars(MTLoStars other){
  return false;
}
```

class ConsLoStars

Here we run into the same problem. If the second list was an instance of *ConsLoStars*, we could compare the *first* in each and the *rest* in each. But *that* can also be an instance of *MTLoStars*.

We again wish for a helper method that gets as argument an instance of *ConsLoStars*:

```
// determine whether this list is the same as given ConsLoStars list
boolean sameConsLoStars(ConsLoStars other)
```

and the body of the original method becomes:

```
return that.sameConsLoStars(this);
```

This method is invoked by the list that can be either an instance of *MT-LoStars* or an instance of *ConsLoStars*, but we know its argument is an instance of *ConsLoStars*. Again, we must define this method for both the *MT-LoStars* and the *ConsLoStars* class.

Here is the variant in the *MTLoStars* class:

```
// determine whether this list is the same as given ConsLoStars list
boolean sameConsLoStars(ConsLoStars other){
  return false;
}
```

All that remains is the body of the method in the *ConsLoStars* class. But here both *this* and *other* are of the type *ConsLoStars*. The template gives us the following:

```
... this.first ...        ... other.first ...   -- Star
... this.rest ...         ... other.rest ...    -- LoStars
```

... *this.first.same(Star ---)* ... -- *boolean*
... *this.rest.same(ALoStars ---)* ... -- *boolean*
... *other.first.same(Star ---)* ... -- *boolean*
... *other.rest.same(ALoStars ---)* ... -- *boolean*

We see that we can compare the two *first* fields (both of the type Star):

this.first.same(other.first)

which invokes the method *same* in the class Star.

Reading the purpose statement for *this.rest.same* method invocation we get:

determine whether the rest of this list is the same as given list

and all we have to do is to use the *other.rest* as its argument, to compare the *rests* of the two lists:

this.rest.same(other.rest)

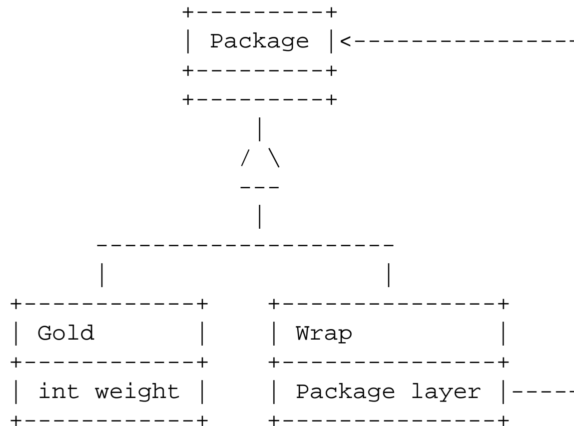
which invokes the method *same* in the class that is the runtime type of *this.rest*.

The body of the method is then:

// determine whether this list is the same as given ConsLoStars list
boolean sameConsLoStars(ConsLoStars other){
  return this.first.same(other.first) && this.rest.same(other.rest);
}

5.5 On your own ...

Here is another class hierarchy. Design the method *same* that determines whether two packages are the same.



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