6.1 The World

Our projects that extended the *World* contained three *import* statements, indicating that we need to use classes defined in three different libraries written by someone else.

To run a project that extends the *World* in Eclipse (or any other Java 1.5 IDE) we need to save a copy of the relevant library files in a folder, and specify in the project properties the file path to these libraries.

Managing the Libraries

- In the previous lab you have used one library file, *tester.jar*. In this lab you will use three additional library files. We suggest that you keep all library files in a common folder. For the purposes of this lab we will call this folder *JARs*.
- Copy into this folder the three library files *draw.jar*, *colors.jar*, and *geometry.jar*.
- In the *Project* menu select *Properties*.
- In the left pane select *Java Build Path*
- In the top menu line select *Libraries*
- On the right select *Add Variable* .... A pane with title *New Variable Classpath Entry* will open.
- Click on *Configure Variables...*
- Click on *New* to get the *New Variable Entry* pane
- Enter *draw* as *Name* and click on *File...* to select the *draw.jar* file in your *JARs* directory.
- Hit *OK*. A new entry should be visible under the *Classpath Variables*.
- Click again on *Configure Variables...* and follow the same steps to add the file *colors.jar* to the *Variables*, and to add the file *geometry.jar* to the *Variables*.
- Hit *Cancel* to get back to the main *Eclipse* environment.

  From now on all your projects will be able to use these libraries.

Configuring a Project with the World Library

Start a new project *BlobWorld*. Import the .java files from the *BlobWorld* folder. Notice that the files are marked with a number of errors. You need the *World* library.

To work with the libraries you need to add the three *Variable*s you defined earlier to this project. The process is similar to what you did earlier:

- In the *Project* menu select *Properties*.
- In the left pane select *java Build Path*
- In the top menu line select *Libraries*
- On the right select *Add Variable* .... A pane with title *New Variable Classpath Entry* will open.
- Click on *draw* entry in the list of available *Variables* and hit *OK*.
- You are back in the pane where you started adding a variable, but now, the entry for *draw* is available.

  Repeat the last two steps for the *colors* and *geometry Variables*.
- When you are done, hit *OK* to get back to you project environment.

You can now run your *BlobWorld* project. The key controls the movement of the ball, but the timer also moves the ball randomly on each tick. The *world* ends when the ball moves out of bounds.

Make sure you can run the project. Read the code, to see the design. It is nearly the same as what you have done in ProfessorJ. The method runWorld is invoked as a part of the test suite.

Making a Better World

Change the class *TimmerWorld* so that the world now consists of two *Blob*s, one that moves randomly on tick, and another that is controlled by the user through the arrow key events. Add the code that ends the world (using the *endOfWorld* method) when the two *Blob*s are overlapping (the distance between their centers is smaller than the sum of their radii). *Follow the Design Recipe.*

6.2 Quiz

6.3 Circular Data

We will now see in practice how to deal with circularly referential data similar to what we have seen in lectures.

In this part we'll visit a familiar concept where circular data exists – namely, buddy lists. These buddy lists could be IM buddy lists, ICQ ubuddy lists, or lists of friends on social networks. Inuitively a buddy list is just a username and a list of other buddies; the latter part is where we get circularity. So, start by working with the following files:

- Buddy.java
- ILoBuddy.java
- MTLoBuddy.java
- Examples.java

1. Create a project LabBuddies and import the four files listed above into the default package. Add the *tester.jar* library to the project as you have done before.

   All errors should have disappeared and you should be able to run the project.

2. Before we can design any methods for the lists of buddies, we need to ba able to make examples of buddy lists.

   Design the method *add* that adds a buddy to one person's *buddy list*. Add any additional methods you may need to make sure you can represent the following circle of buddies:

   - Tom's buddies are Jan and Tim
   - Tim's buddies are Dan and Jan and Tom
   - Jan's buddies are Tom and Tim
   - Dan's buddy (only one) is Tim

3. Now we would like to ask some pretty common questions, e.g

   - Does this person have this other person as a direct friend?

- How many buddies do two buddies have in common?
- Does this person have this other person as a "friend-of-a-friend"?

The purpose statements and the mehtod headers for the three methods are already given:

```
// returns true if this has that as a direct buddy
boolean hasDirectBuddy(Buddy that)

// returns the number of buddies this and that have in common
int countCommonBuddies(Buddy that)

// returns true if this has that as a direct or distant buddy
boolean hasDistantBuddy(Buddy that)
```

Start by implementing *hasDirectBuddy*. *Follow the Design Recipe!*

If you have time remaining, design the other two methods as well.