[Have api out while working through case study]

go to dr java, find link to source code & download.
move GridWorldCode in downloads to your folder for code as a folder.

Unit 8. GridWorld Notes

[i keep my projects separate, under boxBug file store BoxBugProject]

Running Gridworld using DrJava.

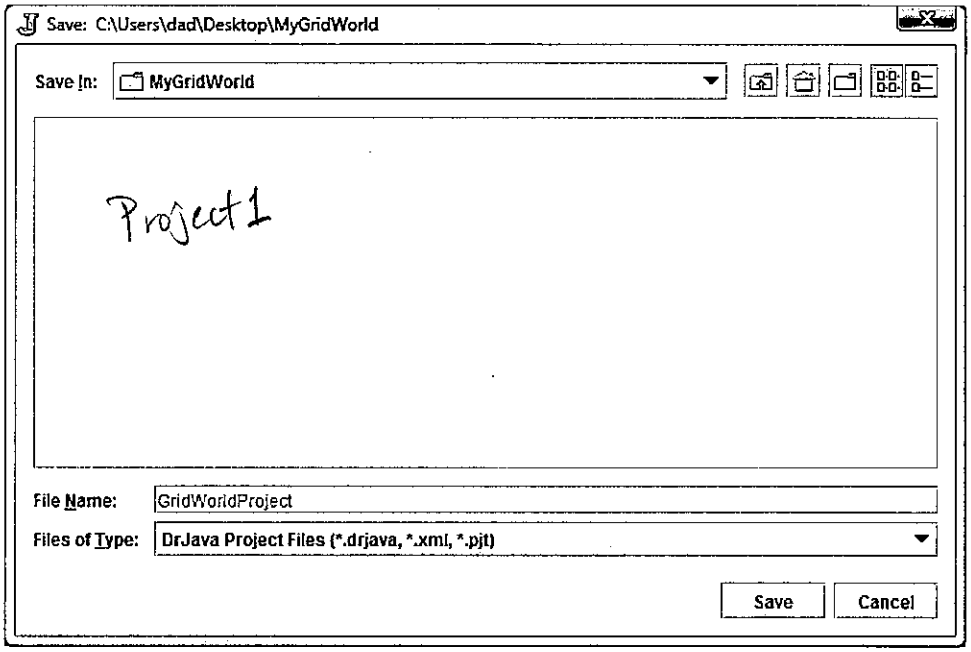
1. Create a folder. Copy BugRunner.java and gridworld.jar file into the folder.

project new

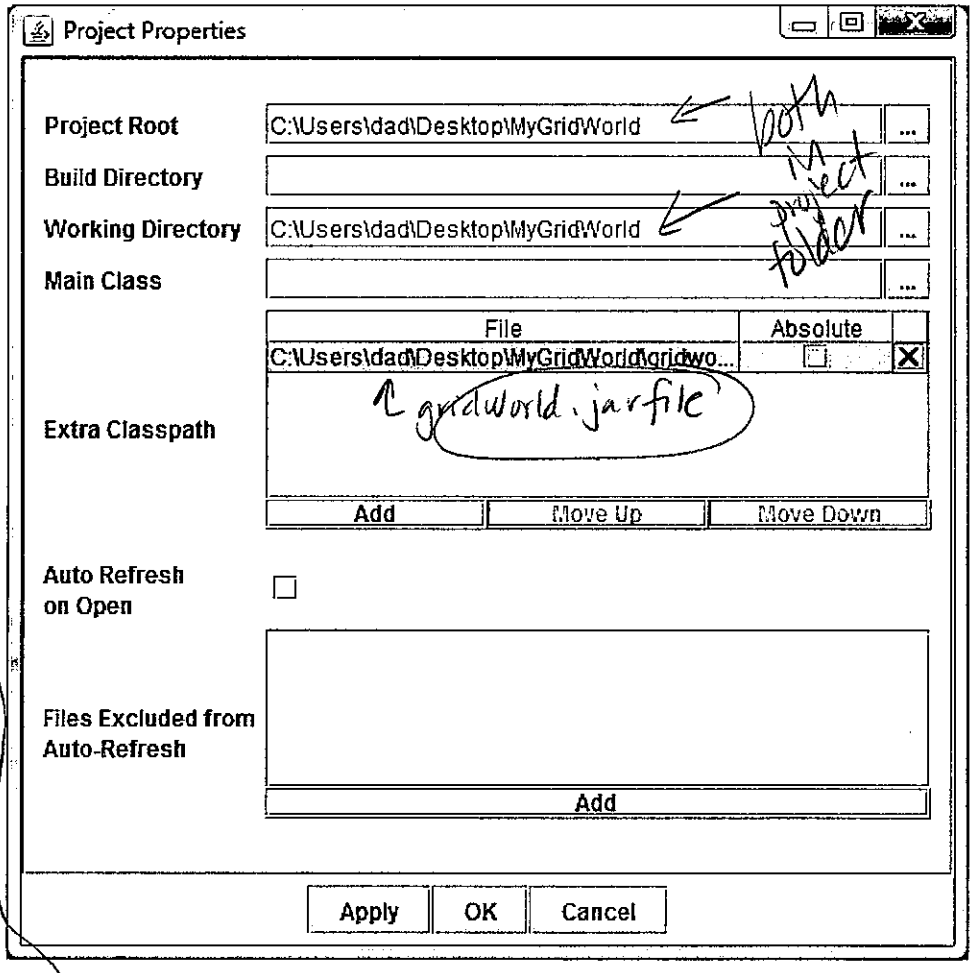
2. Open DrJava and create a Project. Save the project in the folder you just created.

i use their pre-made first project folder. i copied gridworld.jar there.

When you click Save, the Project Properties window appears.



3. Click the Add button and select the gridworld.jar file. hit apply Then Click OK.



path in project folder

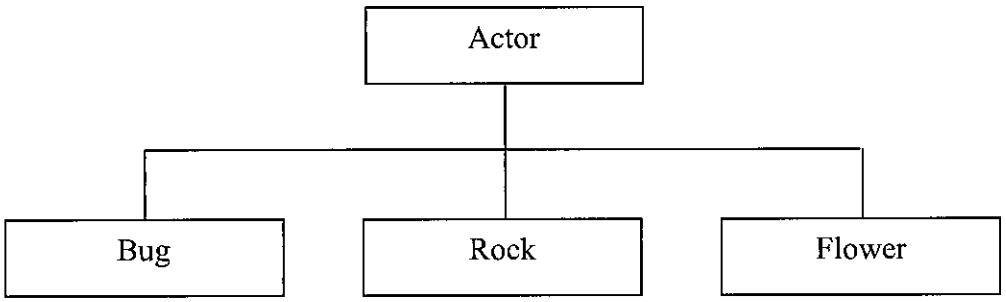
4. In DrJava, open the BugRunner file. Compile the project and run it.

for other projects same.
open
Runner file,
then
compile
project
then run

Part 1. Observing and Experimenting with GridWorld

```
import info.gridworld.actor.ActorWorld;
import info.gridworld.actor.Bug;
import info.gridworld.actor.Rock;

public class BugRunner {
        public static void main(String[] args) {
                ActorWorld world = new ActorWorld();
                world.add( new Bug() );        // adds a bug to a random empty spot
                world.add( new Rock() );       // adds a rock to a random empty spot
                world.show();                  // displays the grid
        }
}
```



The value of an object's instance variables is sometimes called the object's state. An actor's state is represented by four instance variables (these may not be the actual names):

1. direction
2. color
3. position
4. type

When the Step button is clicked, the move() method for each actor in the grid is called. If the Run button is clicked, then it is as if the Step button were being clicked again and again at whatever speed the slider control is set at.

How does a Bug act?

1. If possible, it will more one square in the direction it is pointing. It can move on a diagonal.
2. When it moves, it leave behind a flower
3. If its way is blocked by a wall, rock, or other bug, the bug turns 45° right
4. If its way is blocked by a flower, the bug walks on top of it

The Model-View-Controller (MVC) Approach to Developing Programs. In this approach each class falls into one of three categories.

- Model application data, logic, functions
- View the representation to users
- Control accepts input & commands

Using this approach one can modify the "look" of a program without having to change any of the model classes. Conversely, one could change how we model the behavior of objects without having to change the look of the program. Gridworld has been developed using this approach. We are only concerned with those classes that "model" the behavior of different actors in the grid. Those classes that handle the display and the interaction with the user are considered "black box" classes.

Part 2. Bug Variations. Here is the code for the Bug class's act method:

```
public void act() {
        if ( canMove() )
                move();
        else
                turn();
}
```

Extending the Bug class. Create another project and do the following:

1. Paste a copy of the gridworld.jar into the project folder.
2. Find the boxBug folder. Copy and paste the three files (BoxBug.gif, BoxBug.java, and BoxBugRunner.java) into the folder.
3. Compile and run the project.

```
import info.gridworld.actor.ActorWorld;
import info.gridworld.grid.Location;
import java.awt.Color;

public class BoxBugRunner{
        public static void main(String[] args){
                ActorWorld world = new ActorWorld();
                BoxBug alice = new BoxBug(6);
                alice.setColor(Color.ORANGE);
                BoxBug bob = new BoxBug(3);
                world.add(new Location(7, 8), alice);
                world.add(new Location(5, 5), bob);
                world.show();
        }
}
```

the length of the sides of the "box" in steps

straight 6 steps & then right go 6 steps then right again six steps

step awhile to get used to it. if wall turn right 90°

Part 3. GridWorld Classes and Interfaces.

You want to get very comfortable with the classes and interfaces that make up Gridworld. You will have most of the appendices available to you whenever you have a quiz or exam (including the AP exam) but the better you know the material the faster you can solve the problems.

Some methods of the Location class.

Headers		Comments
int	getRow()	
int	getCol()	
Location	getAdjacentLocation(int direction)	
int	getDirectionToward(Location target)	

There are also many static constants that can be useful

The grid inside the ActorWorld is represented by the Grid interface. By using an interface, one can create an ActorWorld that consists of a bounded or unbounded grid and every actor still interacts with the grid using the same set of methods. We will always be working with a bounded grid.

Some methods of the Grid interface.

Headers		Comments
int	getNumRows()	
int	getNumCols()	
boolean	isValid(Location loc)	
E	get(Location loc)	Precondition: loc is valid
ArrayList<Location>	getOccupiedLocations()	
ArrayList<Location>	getValidAdjacentLocations()	
ArrayList<Location>	getEmptyAdjacentLocations ()	
ArrayList<Location>	getOccupiedAdjacentLocations()	
ArrayList<E>	getNeighbors()	

The grid interface also specifies put and remove methods but don't use them.

You should understand all the methods in the Actor class. To add or remove an actor from the grid, use these methods of the Actor class.

```
public void putSelfInGrid( Grid<Actor> gr, Location loc)
public void removeSelfFromGrid()
```

Important. When an actor is created, getLocation and getGrid both return null because the actor has not yet been added to the ActorWorld. It is only after an actor is added that these methods return non-null values.

Part 4. Interacting Objects

Objects of the Critter class interact with other actors around them. Its act method consists of calls to the following methods.

Method	Description	Postcondition
getActors()	Returns an array list of the actors in the neighboring locations.	The state of all actors is unchanged.
processActors(ArrayList<Actor> actors)	Removes all actors from the grid that are specified in the array list (except for rocks and critters).	(1) The state of all actors in the grid other than this critter and the elements of actors is unchanged. (2) The location of this actor is unchanged.
getMoveLocations()	Returns an array list of empty valid neighboring locations.	The state of all actors is unchanged.
selectMoveLocations(ArrayList<Location> locs)	Returns a randomly selected location from the array list. If the size of locs is zero, then return the current location.	(1) The return location is an element of locs, this critter's current location, or null. (2) The state of all actors is unchanged.
makeMove(Location loc)	Moves the critter to the given location loc, or removes the critter from its grid if loc is null. Note. An actor may be added to the old location	(1) getLocation() == loc (2) The state of all actors other than those at the old and new locations is unchanged.

Some rules for extending the Critter class.

- Do NOT override the act method. Override one or more of the five above methods.
- When you override any of the five above methods, you must maintain the postconditions.

Appendix A.
Jar Files, Packages, and Import Statements.

A **JAR** (**J**ava **Ar**chive) file consists of class files, images, and other resources. It is a common way to distribute java applications or libraries.

A **Java package** is a mechanism for organizing Java classes into namespaces. Think of a namespace as a collection of classes that all have unique names. "Java packages can be stored in compressed files called JAR files, allowing classes to download faster as a group rather than one at a time. Programmers also typically use packages to organize classes belonging to the same category or providing similar functionality." (Wikipedia)

Java has a number of core packages. Here are three:

java.lang	Fundamental java classes including the String and Math classes. This package is automatically "imported."
java.util	Contains various utility classes such as the Scanner class.
java.awt	Contains various GUI related classes including the Color class.

If your program wants to use the Scanner and ArrayList classes (which are both in the java.util package), then you should write this.

```
import java.util.Scanner;
import java.util.ArrayList;
```

If you write this:

```
import java.util.*;
```

then you are making every class in the java.util package available to your class.

IMPORTANT. There are certain naming conventions used with packages that can be misleading. For instance, here are the names of two packages in java:

```
java.awt
java.awt.font
```

If you wrote

```
import java.awt.*;
```

you would be importing all the classes in java.awt but not the java.awt.font package. java.awt.font is a separate package but named in this way because the font package is related to awt package.

Here are two packages in GridWorld

Name	Includes these classes and interfaces
info.gridworld.grid	
info.gridworld.actor	