CollegeBoard
inspiring minds™

GridWorld:

A Curriculum Module for AP® Computer Science

2010

Curriculum Module

The College Board

The College Board is a not-for-profit membership association whose mission is to connect students to college success and opportunity. Founded in 1900, the College Board is composed of more than 5,700 schools, colleges, universities and other educational organizations. Each year, the College Board serves seven million students and their parents, 23,000 high schools, and 3,800 colleges through major programs and services in college readiness, college admission, guidance, assessment, financial aid and enrollment. Among its widely recognized programs are the SAT®, the PSAT/NMSQT®, the Advanced Placement Program® (AP®), SpringBoard® and ACCUPLACER®. The College Board is committed to the principles of excellence and equity, and that commitment is embodied in all of its programs, services, activities and concerns.

For further information, visit www.collegeboard.com.

Visit the College Board on the Web: www.collegeboard.com.

Contents

Introduction

Fran Trees

"A case study describes a programming problem, the process used by an expert to solve the problem, and one or more solutions to the problem. Case studies emphasize the decisions encountered by the programmer and the criteria used to choose among alternatives." [1] Case studies have been part of the AP® Computer Science curriculum since 1995. These studies allow students the benefits of an apprenticeship by being able to work with large programs written by experts. Students can be guided through programming design issues early in your course rather than postponing the complexity of program design until the language has been mastered. Case studies also provide an excellent vehicle for assessing a student's knowledge and are great tools for promoting teamwork and active learning.

The teaching modules included here provide instructional strategies and presentation suggestions related to the GridWorld case study (http://www.collegeboard.com/student/testing/ap/compsci_a/case.html). The authors of the modules are experienced AP teachers and university professors who have used case studies in an introductory Java course and are excited about sharing their favorite activities with you.

After an overview of the GridWorld case study, highlighting ways to incorporate the case study into an AP Computer Science course, each of the five parts of the case study are addressed by supplying you with a variety of ideas and tools. The instructional units include teaching strategies, presentation suggestions and activities (including lab assignments) that are designed to address the various ability levels and learning styles of the students in your class. Throughout the document there are student worksheets that can be printed and distributed to students at various times in your course, either as reviews of GridWorld concepts or as exercises and homework assignments while working with GridWorld.

1. M.J. Clancy, and M.C. Linn, "Case Studies in the Classroom," *Proceedings of the 23rd SIGCSE Technical Symposium on Computer Science Education* (Kansas City, Mo., March, 1992).

Overview

Judy Hromcik

GridWorld is a great tool for teaching inheritance to computer science students. In the past, many of us have taught the case study as an add-on at the end of the course, just before the AP Exam. In this article, I suggest ways to integrate the case study throughout the year by using it as a vehicle to teach students inheritance, to teach students how to create and use objects, and to facilitate the instruction of other non-OOP computer science concepts. I will also discuss some of the technical issues with setting up and running the case study code.

The gridworld.jar file

As with any case study, there are always obstacles to overcome before you and your students will be able to compile and run the case study code. The case study requires students to examine a large body of code and add to that body of code. The classes that are already written have been compiled into ***.class*** files and are contained in the ***gridworld.jar*** file found in your distribution files from the College Board (http://www.collegeboard.com/student/testing/ap/compsci_a/case.html). A classpath must be set to this jar file or students will not be able to compile and run the case study code. If you are unfamiliar with this process, the GridWorld case study files from the College Board include a document, "GridWorld Installation Guide," that should be read before setting up projects for your students.

I use JCreator and BlueJ in my classroom. In BlueJ, you can set the classpath to the jar file by adding it as a library in the preferences; you only need one copy of the jar file. In a controlled lab setting, this is very painless, and any case study project will run from that point on without worrying about the jar file. I have found this to be convenient when teaching workshops with teachers and students in a lab.

Using just one copy of the jar file in JCreator poses more problems than it does in BlueJ, because there are so many ways to set the classpath to the jar file. Each project must set a classpath to the jar file or you must configure the JCreator environment to add the jar file. For many reasons, this has not worked well for my students when they work at home on GridWorld projects. When transporting project folders between school and home, the classpath must be reset in the project or you must count on the students configuring the JCreator environment correctly at home. In order to make case study projects completely portable, I have my students copy the jar file into each project's folder and add the jar file to the project. Once the project is saved, students can transport the project folder between computers with Java and JCreator installed and the project will always find the jar file.

Import statements

The GridWorld case study uses packages. This requires the programmer to use import statements or use a very long class name when declaring and instantiating actors, bugs, critters, etc. For example, the `Bug` class is a part of the `info.gridworld.actor` package. The fully qualified name of the `Bug` class is actually `info.gridworld.actor.Bug`. To be able to just type `Bug` when declaring and instantiating a bug, the programmer must write the following import statement:

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

The following chart lists the classes and interfaces in each package:

Package	Classes and Interfaces
`info.gridworld.actor`	`ActorWorld, Actor, Bug, Critter, Rock, Flower`
`info.gridworld.grid`	`Grid, AbstractGrid, BoundedGrid, UnboundedGrid, Location`
`info.gridworld.world`	`World`

In addition to these classes, students must remember to include import statements for the standard Java classes that they use. A common one needed is the `java.awt.Color` class. While import statements are not tested on the AP Exam, students need to understand when to include them when necessary.

Integrating GridWorld in the AP® Computer Science curriculum

The purpose of a case study is to allow students to study, modify and add to a large body of code. In an Object Oriented language, it also requires students to understand and use interacting classes. The GridWorld case study was designed to allow teachers to introduce the material to students early in the AP Computer Science course and then use it throughout the school year. AP Computer Science teachers have the task of figuring out how and when to teach the case study material.

If you compare GridWorld to the Marine Biology Simulation (MBS), you will notice that there are fewer interacting classes in GridWorld than there are in MBS. This may lull you into the false sense of security that you can teach GridWorld faster than you could MBS; I have not found that to be true. The complexity is still there, and Part 4 requires solid knowledge of the interacting classes. Some teachers introduce GridWorld early and throughout the year come back to the case study as the topics covered in GridWorld come up in the curriculum. Some teachers plan to teach the case study as a review of the AP Computer Science course right before the exam. Initially, I was going to follow the materials and cover them in order as I got to those topics in the curriculum,

but I have since changed my mind; I am now using GridWorld to help reinforce the use of objects and inheritance. I am also using GridWorld to help teach loops and conditional statements, `ArrayList`, classes, and anything else I can think of as I teach the curriculum throughout the year.

I introduce my students to OOP using Jeroo (Jeroo: http://home.cc.gatech.edu/dorn/jeroo), a wonderful book and program by Dean Sanders and Brian Dorn from the University of Missouri. In Jeroo, students make one kind of object, a Jeroo, and then send messages to the Jeroos to make them act. They also modify the Jeroo class by adding new methods. Once this unit is finished, I introduce the students to Java. Early in the Java introduction, it is difficult to keep students in the OO frame of mind. Java does not have a lot of simple classes that students can easily use as they are introduced to the language basics. It is difficult to leverage what the students learned about creating and using objects in Jeroo as they move into Java.

Stacey Armstrong (http://apluscompsci.com) from Cypress Woods High School in Houston, Texas, has solved this problem by using GridWorld as a key curriculum component in AP Computer Science, developing GridWorld lessons and labs for students to use from the very beginning of the course and throughout the year. These labs allowed me to seamlessly move from Jeroo to Java and keep OO alive. After students finish Part 1 of the case study, the `Actor`, `Location` and `ActorWorld` classes can be introduced. Introductory labs have students write a runner class that simply creates a world and adds actors to the world.

```
public class ActorRunner
{
    public static void main(String[] args)
    {
        ActorWorld world = new ActorWorld();
        Actor a1 = new Actor();
        world.add(a1);
        world.add(new Location(2,2), new Actor());
        world.show();
    }
}
```

The first labs only require that students know how to call the constructors for the `Actor` and `Location` classes and the constructor, `add`, and `show` methods for the `ActorWorld` class. Students are essentially doing the same tasks in these programs as they did at the beginning of the Jeroo unit.

After the GridWorld introduction, the Java language basics can be taught: primitive variables, references, input, output, assignment statements, arithmetic, etc. While teaching these basics, students can input row and column values to make new `Location` objects, and then create `Actor` objects to place in an `ActorWorld`.

After the Java language basics have been covered, you could begin teaching simple inheritance by making new types of actors. One such actor from Armstrong's

materials is the `MoveLeftActor`. Only the `Actor act` method is overridden and the `getDirection and moveTo` methods of `Actor` are used. The `Location` class methods `getAdjacentLocation`, `getRow`, `getCol` and the `Location` constants are also introduced. Students learn how to get the adjacent location to the left of the actor and then move to that location. No checks are added to keep the actor from going out of the grid. The focus of this lab and others like it is to introduce inheritance and to learn how to use some of the methods found in the `Actor` and `Location` classes. Later on, after conditional statements have been taught, this lab is redone to check for boundaries. The `Actor's getGrid` method is introduced along with the `Grid` interface's `isValid`, `getNumRows` and `getNumCols` methods. Before Part 2 is taught, students are introduced to a substantial number of methods found in the case study's interacting classes. Many of these methods are not introduced in the narrative until Part 3. This simplifies teaching Part 3 and gives the teacher some good classes to use early in the AP Computer Science curriculum.

Once conditionals are taught, students can begin on Part 2. New instance variables, constructors, `super()` and `super.method()` can be taught using the `Bug` class as a base class. Essentially, Part 2 can be taught as is in the narrative. The 2007–2008 *AP Computer Science Special Focus Edition: GridWorld Case Study* has additional `Bug` labs as well that can also be added to the exercises found in the narrative. Parts 3 and 4 can be taught after loops and `ArrayList`. Part 4 is a great unit to help you teach the intricacies of inheritance.

GridWorld can be used to enhance your teaching of other programming concepts such as conditionals and loops. Cay Horstmann has a Web page (http://www.horstmann.com/gridworld/extending-gridworld.html) that shows how to use the GridWorld environment to teach other concepts in AP Computer Science. My favorite is `LoopWorld`. Using this project, students can place `Actor` objects in the grid with a loop and see the objects appear one at a time. Students can "see" loops and nested loops in action, and they can continue to instantiate and use objects. The graphical nature of this code can assist your visual learners. Additional extensions of the GridWorld case study can also be found at the above-referenced website. If you and your students visit this page, you will find that you can also use the GridWorld framework to create games.

As I work with GridWorld, I see more ways to integrate worlds, locations, grids and actors to help reinforce object instantiation and object use, and to help teach classes, inheritance, conditionals, loops, input, etc. The GridWorld classes help me to teach objects first and to teach them often.

Part 4 of the GridWorld Case Study

The `Critter` class was designed with an `act` method that always performs five basic behaviors:

1. Get a list of actors for the critter to process in some way.
2. Process those actors.

3. Get a list of possible move locations for the critter.
4. Select one of those locations.
5. Make the move.

This is what all critters do when they act. New types of critters should do this as well. Therefore new `Critter` subclasses override one or more of the five methods found in the `act` method. The `act` method should not be overridden. Students should remember that a `Critter` is defined to be an actor that performs the five basic behaviors in the `act` method. If a new actor is being defined and needs to perform other behaviors, consider overriding the `Actor` class.

A word of caution about Part 4: Read all of the descriptions and postconditions for these five methods found in `act`. They must be followed when your students create new `Critter` subclasses. The postconditions and additional comments about the postconditions for each of these five methods are listed below:

Critter Method	Postconditions and Comments
`getActors`	Postcondition: The state of all actors is unchanged. No changes can be made to any actors anywhere, in or out of the grid. This means you cannot move actors in the grid or add actors to the grid or change any actor's attributes.
`processActors`	Postcondition: (1) The state of all actors in the grid other than this critter and the elements of <code>actors</code> is unchanged. (2) The location of this critter is unchanged. For actors currently in the grid, only this critter and the actors in the getActors() ArrayList can change. New actors can be added to unoccupied locations.
`getMoveLocations`	Postcondition: The state of all actors is unchanged. No changes can be made to any actors anywhere, in or out of the grid. This means you cannot move actors in the grid or add actors to the grid or change any actor's attributes.
`selectMoveLocation`	Postcondition: (1) The returned location is an element of locs, this critter's current location, or null. (2) The state of all actors is unchanged. No changes can be made to any actors anywhere, in or out of the grid. This means you cannot move actors in the grid or add actors to the grid or change any actor's attributes
`makeMove`	Postcondition: (1) getLocation() == loc. (2) The state of all actors other than those at the old and new locations is unchanged. Only the actor and the actor at the move location can change.

Suppose you have a new type of critter that keeps a log of all of the places that it visits. It contains an `ArrayList` private instance variable to store these locations. Which method should update the `ArrayList`? You might choose `selectMoveLocation`. This is a reasonable place to update the list, but the postconditions do not allow any changes to be made to the critter. The `ArrayList` should be updated in `makeMove`. Be sure to teach these postconditions to your students. Students who violate the postconditions when writing free-response answers may not receive full credit for their solutions.

The GridWorld case study can and should be used throughout the year to introduce and enhance numerous topics, such as OOP fundamentals, decision making, looping and inheritance. GridWorld can be used to help students better understand and grasp any of the AP Computer Science A topics. The graphical nature of GridWorld makes it very attractive to visual learners, and it also serves as a game creation platform, thus making it an incredible recruiting tool. GridWorld is robust and has a rich source of materials to help us teach AP Computer Science.

Unit Plan: Parts 1, 2 and 3

Kathleen A. Larson

Parts 1, 2 and 3 of the GridWorld case study (http://www.collegeboard.com/student/testing/ap/compsci_a/case.html) can be taught during a contiguous time period or may be spread over many weeks interspersed with other AP Computer Science course material. This instructional unit is designed to be implemented either way, at the teacher's discretion, and provides a variety of ways to introduce the case study parts. It includes activities and lesson plans for each part; diagnostic, formative and summative assessment tools; accommodations for different types of learners; a teacher self-evaluation; and resources.

GridWorld topics covered in this unit are:

- Part 1: Observing and Experimenting with GridWorld
- Part 2: Bug Variations
- Part 3: GridWorld Classes and Interfaces

The GridWorld case study is a required element of the AP Computer Science curriculum and constitutes a significant part of the AP Computer Science Examination. It is incumbent upon the teacher to provide ample experience with the case study for students to learn the important concepts illustrated in it and for them to achieve success on the exam.

Worksheets for Parts 1–3 of GridWorld, written by Joe Coglianese, are found in the appendixes of this article.

Introducing GridWorld

The purposes of a carefully planned introduction are as follows:

- Motivate students to want to learn the new material
- Focus their attention on this topic
- Activate prior knowledge by connecting the new learning to what each student already knows, in order to create brain synapses that will enable real learning to occur
- Diagnostic assessment, to find out what students already know in order to build on their prior knowledge

The following are five possible GridWorld introductory activities:

1. One simple way to introduce any new material is a KWL (Know, Want to know and Learned) chart (see Appendix A). Give each student a paper divided into three columns labeled "Know," "Want to Know" and "Learned." At the beginning of study, the students are asked to fill out the first two columns with a list of what they already know about the topic (case studies, grids and GridWorld in particular) and what they want to know. At the end of a module (all three parts or each part individually), students will complete the third column with what they have learned. Give the students two or three minutes to fill out the first two columns. On a transparency or on the whiteboard, create a similar chart. Allow five minutes for students to tell you what they know and what they want to know, and to student's list many of their answers on the master chart. After completing the module, you and the students can fill in what they have learned. This activity satisfies the required parts of an introduction, especially by giving you feedback for diagnostic assessment; you will build on this information.

2. When the students enter the classroom, have a "Gummy Bug" (see Appendix B) on each student's desk. Tell them they have one minute to write down three ways their bug might interact with its environment. After one minute, tell the students to share their ideas in small groups and decide on the best three ideas; allow two minutes for this activity. Each group should choose one spokesperson to report the group's three ideas to the class. Students may eat their bugs.

3. As students enter the classroom, have them take a folded piece of paper from a paper bag (basket, hat …); the papers will say either "bug," "flower" or "rock." In one corner of the room have a large picture of a bug; in another corner, a picture of a flower; and in a third corner, a picture of a rock. Tell the students they have two minutes to find their corners and discuss how their selected character behaves. You might provide props or costumes to enhance the role-playing aspect of this activity. At the end of two minutes, ask each group to act out the behaviors of their selected character. Have the rest of the class try to guess what the "acting" group is doing.

4. Make up tic-tac-toe game boards on sheets of paper, and distribute them to pairs of students. Make game pieces that are either bugs or rocks (see Appendix B). Give one player of each pair three bug pieces, and give three rock pieces to the other player in the pair. Have them play tic-tac-toe with the pieces five times and report the number of wins. (Alternatively the players could draw a bug or a rock in the position on the paper game board.) Ask the class to name other games that are played on a grid and suggest extra credit for any of the pairs who, by the time the module is completed, come up with rules for a new game that is played on a grid and that uses bugs, rocks and, possibly, flowers.

An alternative to traditional tic-tac-toe is a version played in reverse. Pairs of students start with a 4 x 4 (or N x N) grid and take squares using bugs and rocks. The object is to *avoid* getting three of the same marker in a row. Ask students to discuss their choice of moves and describe a winning strategy.

5. Ask each student to recall an incident that happened to him or her in their childhood involving a bug, a flower or a rock. Have them draw a depiction of the story. Then have the students form small groups to share their stories and create one single story out of their individual stories. They may use the pictures they already have or create a single picture that combines the stories. Each group then has two minutes to tell their combined story to the class.

Each of these activities is intended to introduce GridWorld on the first day you use the case study. Each in some way uses social interaction (cooperative learning) and active involvement. Parts of the activities are geared toward different types of learners (i.e., visual, auditory or kinesthetic), and each activity allows the teacher to grab the students' attention and to attach new learning from GridWorld to students' prior knowledge.

At the end of any of these introductory activities, inform the class that you are introducing them to the GridWorld case study. Explain to the students the duration of the class work with the case study (typically three weeks) and that the class will revisit the case study periodically throughout the rest of the school year. Also explain that the case study will help them learn several of the primary concepts of this course; that they will see model code that will help them learn how to organize and write their own programs; that they will work with graphic characters, developing special abilities for them, and make the characters move around a special world on the screen; and that the case study is part of their preparation for the AP Computer Science Exam.

Note: Each of the following lesson plans is intended for one 45- to 50- minute class period.

Part 1: Activity/Lesson Plan 1 — GridWorld Part 1

Students will experiment with GridWorld to learn how the environment and actors behave.

Learning Goals

- Students will learn the capabilities of bugs, rocks, flowers and the Graphic User Interface (GUI).
- Students will develop investigative techniques to be used in understanding GridWorld.

Lesson Objectives

- Given a working version of GridWorld and an investigative worksheet, the student will be able to activate the GUI in Step and Run modes and determine the characteristics and capabilities of the GUI, a bug, a flower and a rock with 100 percent accuracy.
- Given a working version of GridWorld and a set of questions to be explored, the student will be able to work cooperatively with another student to investigate adding actors to the grid and to determine what those actors can and cannot do and how the grid can be used with 100 percent accuracy.

Materials

- GridWorld Case Study Part 1: Observing and Experimenting with GridWorld (http://www.collegeboard.com/student/testing/ap/compsci_a/case.html)
- "Running the Demo" Reading Worksheet (see Appendix C)
- "Exploring Actor" Reading Worksheet (see Appendix D)

Anticipatory Set

Assuming this lesson is taught on the first day of the unit, one of the introductory activities listed above may serve as the anticipatory set for this lesson. If, however, this lesson is not taught on the first day, the following introduction may be used.

Play the game, *Find the Route*, found at http://math2.eku.edu/Greenwell/MAT303. In this game, an individual player must find a route from the upper left-hand corner to the lower right-hand corner of a grid according to prescribed rules. This activity focuses the students' attention on the grid. Ask them to contrast the grid in this game with the Cartesian coordinate system.

Lesson Development

Project the instructor's computer screen and demonstrate how to open and run the GridWorld demo program. Instruct the students to follow the steps with you, making sure they are able to open and run GridWorld. Hand out Part 1 of the case study and the above-referenced worksheets. Instruct the students to complete the Running the Demo worksheet independently, following the instructions and answering the questions, then compare answers with another student. The pairs of students should then work on the Exploring Actor worksheet activities and questions. Students may refer to Part 1 of the case study as needed. Circulate about the classroom, answering students' questions and helping as needed, but allowing the students to do their own investigation. Monitor the students to be sure that they are finding the worksheet answers through their investigation. This is your formative assessment, the time during which you determine to what extent each student is mastering the material.

Accommodations

Students who need extra help may receive it by working with a partner and by the teacher circulating about the room to provide feedback and guidance regarding their work. Those who finish early can investigate features of the case study further.

Closure

At the close of class, take a few minutes to ask the students to summarize what they learned. Ask them not only for the facts that they learned about the GUI and the actors, but also ask them to discuss the investigative strategies they used to find answers to the worksheet questions.

Assignment

Read Part I and answer of the GridWorld Student Manual and complete "Do You Know? Set 1." For homework and as a summative assessment, tell the students to write a short letter to their best friend describing five things they learned about the GUI, five things they learned about a bug, two things they learned about a rock and three things they learned about a flower.

Rubric for a Letter to a Friend

Topic Evaluation		Point Value	Points Earned
Content:		10	
Five valid things learned about a Bug	5		
Two valid things learned about a Rock	2		
Three valid things learned about a Flower	3		
Style:		10	
Correct friendly letter style	5		
Free of spelling and grammar errors	5		

Instruct the students to begin keeping a GridWorld journal. The first entry is to describe what they learned in this lesson and what it means to them in terms of the AP Computer Science course. (See Appendix E.)

Assessment

- Diagnostic assessment occurs by asking questions during the anticipatory set in order to find out what the students already know about a grid.
- Formative assessment occurs through questioning and observation during lesson development.
- Summative assessment occurs with the letter written to a friend. Give the students a rubric for the letter and grade according to the rubric.

Evaluation

Formulate questions you will ask yourself after teaching this lesson, such as the following:

- Was my anticipatory set effective in activating prior knowledge and motivating the lesson?
- Do the students understand how to use the features of the GUI?
- Do the students understand what the objects in the grid can do?
- Do the students know how to instantiate `Actors`, `Bugs` and `Rocks` in the GUI?
- Do the students understand the difference between accessor and modifier method calls?
- In teaching this lesson, what did I do well?
- What could I have done differently?

Part 1: Activity/Lesson Plan 2 — GridWorld Part 1 (continued)

After introducing the students to GridWorld in Part 1, use the GridWorld Role Play to transition into Part 2. The role play provides complete directions and a complete script for implementation.

Role Play for Activity 2

A Role and Name sign is helpful in identifying the players. Print each role individually using a very large font size, centered on a sheet of 8 x 11 inch paper, landscape style. Print the script using regular font size and landscape style, and print any supplementary pages that accompany the role. Insert the role type and script/supplementary forms into a transparent sheet protector. Tie a string or ribbon to the holes in the sheet protector so it can be hung around the player's neck. Using an erasable marker, write the player's first name on the outside of the sheet protector. (Sheet protectors can be reused for any role play, provided names are not written with permanent markers.)

Costume pieces may be used to enhance the role play. Don't go for elaborate, expensive items. Find bug antennae, baseball caps in the shape of bugs, flower headpieces and crumpled material to look like rocks. Homemade headpieces work equally well. The point is simply to give the impression of the role, not to dress up. Hats or headpieces work best. The addition of hats isn't necessary to the role play; it adds a sense of identity with the role and the silliness of wearing bug antennae or a flower or rock hat adds to the fun.

Do not expect to complete one iteration of the Role Play in a single class period, unless your school is on block scheduling. This lesson typically takes two class periods. To do justice to the lessons learned from this activity, it can't be rushed. Taking time now pays big dividends in students' subsequent understanding of the case study concepts.

Learning Goal

- Students will understand the GridWorld design.

Lesson Objective

- Given a role in the GridWorld Role Play, the student will act out the role to the best of his or her ability.

Notes for the GridWorld Role Play

Early in the school year and prior to teaching the case study, take time to use the First Day Role Play (see Appendix B). Students often have little or no previous experience doing a role play. (Perhaps the teacher has little experience directing a role play as well.) The First Day Role Play is a series of short scripts in which everyone can take part. Not only do the scripts allow the teacher and students to see how a role play works but each script actually illustrates an object-oriented programming (OOP) concept.

Have the students sit in a circle and make sure each one has a role for at least one of the scripts. Most important, take time for debriefing after role playing a script. Ask the students how it felt to play a role and what they learned both in terms of the action of playing a part and in terms of the OOP concept illustrated. The teacher as well as the students should examine the role play from both perspectives. These role plays are fun; they usually provide a real break from normal classroom routine. Kinesthetic learners typically understand the concepts better by using this approach, and the whole experience creates enthusiasm and motivation for learning.

Experience with the First Day Role Play is crucial to the success of the subsequent GridWorld Role Play.

Materials

- GridWorld Role Play scripts
- Copies of GridWorld Role Play accompanying notes pages for the roles
- Ball of yarn or string
- Hats or other props to signify roles

- Large grid drawn on erasable plastic sheet for the Display to use
- Erasable markers or stick-on bugs, flowers and rocks for the Display to use

Anticipatory Set

Minimally, one iteration of the role play requires approximately 90 minutes. For many teachers this means allowing at least two class days. The role play includes many tips for success, not the least of which is the teacher's considered decision about which students should take on the most active roles. To maximize the use of classroom time, hand out the roles the day before the role play is to be done and tell the students to become familiar with their scripts prior to class.

Students have observed GridWorld in action from Part 1, but they haven't looked at any code up to this point. They soon will do so, but by experiencing the role play first they will understand the code better when they examine it. Some teachers believe that the opposite is true — that students should study the code and then role play it. This has not been my experience.

When students come into the classroom, tell them to get into character and review their scripts.

Lesson Development

Once again have the students sit in a circle for the role play. If there are more students than roles, have the "extra" students act as a "buddy" for each role, helping the player to find his or her part when told to take an action. If the class is small, a student could take on more than one of the smaller roles. In that case, there should be a way for the student to identify himself or herself in the current role. Signs hung around the students' necks (see Materials list above) help to identify their roles. You might find hats that signify the part being played as well (again, see Materials).

An effective method for keeping track of the hierarchy of method calls is tossing a ball of yarn. The first student to begin the role play holds an end of the yarn. As each object is called upon to execute a method, the ball is tossed to the person playing that role. Along the way the object making the method call hangs on to his or her spot on the yarn. As an object completes its method, the ball is returned to the player who made the call. Eventually the complex hierarchy of method calls is completed and the ball goes back to the player who started the whole chain of event calls. This dynamic illustration of how the objects interact is effective in helping students understand GridWorld code. Later, when studying the code, students are reminded of who did what and in what order during the role play.

The GridWorld Role Play includes Time-for-a-Commercial-Break intervals, during which the teacher and students review what has occurred up to that point. This is an opportunity to check on student understanding.

Closure

Debriefing is a critical part of any role play. Activate the students' higher learning skills of analysis, synthesis and evaluation by asking questions and initiating a discussion of what occurred during the role play. What do they see as the overall GridWorld design? Which responsibilities belong to which objects? Why? Does this make sense? What did they learn about object instantiation? What did they observe about object interaction? From their observations, what can they say about parameters and return values? More questions will come to mind as you watch the role play in action.

Accommodations

The role play involves students in the Social Interaction and Information Processing models of education. Students are actively involved in their education. Visual, auditory and kinesthetic learners are all engaged during the role play. Repetition of steps provides reinforcement for those who need extra practice to understand concepts, while discussion and debriefing questions challenge those who are ready for enrichment.

Assessment

- Diagnostic assessment occurs as the students recall what they learned from GridWorld Part 1.
- Formative assessment takes place during the "commercial breaks" in the role play.
- Summative assessment consists of each student writing a journal entry commenting on his or her role, what he or she learned from participation, five elements of GridWorld each observed during the role play, and two questions they plan to investigate as they study the Java code.

Evaluation

Formulate questions you will ask yourself after teaching this lesson, such as the following:

- Did my anticipatory set actually activate prior knowledge in the students?
- Did I implement the role play effectively?
- Did the students understand what a `Bug`, `Rock` and `Flower` can do?
- Did the students understand responsibilities and the hierarchy of method calls?
- Was the debriefing effective?
- Is there anything I could have done better or anything I would change the next time I teach this lesson?

- Is there anything I did particularly well and would repeat the next time I teach this lesson?

Part 2: Activity/Lesson Plan 3 — GridWorld Part 2

Students study the `BoxBug` and `BoxBugRunner` classes. Using `BoxBug` as a template, they create variations of the `Bug` class that have different characteristics; this facilitates learning about inheritance. This is a Direct Instruction lesson because studying the GridWorld code and learning how to modify it is new material to the students.

Learning Goal

- The students will learn how to create and run their own variations on a class.

Lesson Objective

- Given the `BoxBug` and `BoxBugRunner` classes as templates, the student will design and implement a variation of a subclass of `Bug` with 100 percent accuracy.

Materials

- GridWorld Student Manual Part 2: Bug Variations (page 10) (http://www.collegeboard.com/student/testing/ap/compsci_a/case.html)
- GridWorld Student Manual Testable Code for APCS A/AB (http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/compsci_a_exam_appendix.pdf)
- "Bug Class" Reading Worksheet (see Appendix F)
- "Runner Class" Reading Worksheet (see Appendix G)

Anticipatory Set

Tell the students they have one minute to jot down one thing they have in common with everyone else in the class and one thing they can *do* that is unique about them, something that sets each one apart from everyone else in the class. After one minute the students will find a partner and share what they wrote. After another minute ask the pairs to share their ideas of what is common to the students in the class. Write their ideas on the board. Then ask each one to share what their partner's unique ability is. Tell the students that while we have common characteristics, in some way we are all unique variations of the `Student` class, and that in this lesson they will learn about a variation of the `Bug` class called a `BoxBug`. Inform the students that they will work in pairs to design and implement another variation (that you assign), and then they will use their experience and creativity to design a Bug variation of their own. Inform the students that this code is testable on the AP Computer Science Exam.

Lesson Development

Demonstrate the `BoxBugRunner` class that instantiates a `BoxBug`. Students will imitate the teacher's steps, observing how a `BoxBug` behaves. They will be given copies of Part 2 and Appendix C. Introduce the `BoxBug` class and discuss the code line by line, explaining that `BoxBug` is a subclass of (extends) the Bug class, to be studied in Part 3. The constructor, private instance variables and act method will be carefully discussed. In addition, a runner class must be created for each `Bug` variation. The `BoxBugRunner` class provides a template. There should be continuous didactic questioning of the students to assess how well they understand the new material. These questions are short and factual, aimed at the lowest level of learning. Think of this as the "I do" part of the lesson, in which the teacher does most of the talking and demonstrates what is to be learned.

After studying `BoxBug`, students will work in pairs, one computer per pair, adapting the `BoxBug` code to create a `CircleBug`, as described in Part 2. Generally during this process (the "we do" part of the lesson), the instructor and the students work together. In this instance, the model has been modified to involve the students in pair programming. Students will take turns being the "programmer," the person at the keyboard actually typing the code, and the "navigator," the person studying the modifications for errors and making suggestions about additional modifications to be made. It is important that students swap roles midstream; insist that each person experiences both roles. During this part of the lesson, walk around the room observing student interaction and progress, offering suggestions, and keeping the pairs on task.

When a pair has successfully completed `CircleBug`, they will separate and work independently during the third part of the lesson. In this, the "you do" part, each student will again adapt the `BoxBug` code to create his or her own variation. If they are having a hard time coming up with ideas, try a short brainstorming session in which variations are called out and recorded on the board.

The resulting `BoxBug` variation constitutes the summative assessment for each student. The instructor's usual grading rubric for programs will be applied. (See Materials list above for an example.)

Accommodations

For students who need extra help, the instructor will be available to discuss programming concepts and techniques. Students who are ready for a greater challenge may work on more complex variations or instantiate two objects of the same or different types to see how they interact on the grid.

Closure

Review with the students what they learned from studying the `BoxBug` and `BoxBugRunner` code. Ask them to think about how these concepts apply to programming aside from the case study.

Assignment

Students will complete the `Bug` class "Bug Reading Worksheet" and the "Runner Class Reading Worksheet " on concepts from Part 2. The worksheets will be collected and graded as a summative assessment.

Assessment

- Diagnostic assessment occurs during the anticipatory set when the instructor determines if students understand the concept of variation.
- Formative assessment will be ongoing with questioning and observation during the "I do" and "we do" parts of the lesson.
- Summative assessment will be based on the students' individual variations based on `BoxBug` and the Part 2 worksheets.

Evaluation

Formulate questions you will ask yourself after teaching this lesson, such as the following:

- Did my anticipatory set grab the students' attention and help me to diagnose prior knowledge?
- Was my demonstration and explanation clear?
- Were the students able to work together successfully to practice the concepts?
- Did I provide enough help?
- Is there anything I could have done to improve the lesson?
- Is there anything I did well and would like to repeat?

Part 3: Activity/Lesson Plan 4 — GridWorld Part 3

This lesson examines the `Location` class and applies the `Location` class in a program separate from GridWorld. The model for this lesson is Information Processing because the topic facilitates the students' research about the class and finding an application other than the case study.

Learning Goals

- Students will understand the `Location` class.
- Students will work together to develop an alternate application for the `Location` class.

Lesson Objectives

- Given Part 3 and the GridWorld Student Manual Appendix B, the student will complete a worksheet on the `Location` class with 90 percent accuracy.
- Given the `Location` class, the student will work with a small group to design an application for Location in a situation different from the case study.

Materials

- GridWorld Part 3: GridWorld Classes and Interfaces (page 16)
- GridWorld Student Manual Testable Application Programming Interface (API) (http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/compsci_a_exam_appendix.pdf)
- Location Reading Worksheet (see Appendix H)

Anticipatory Set

Give each student a compass (see Appendix B). Tell them to use the compass to face in various directions, then pick one direction and turn a half-circle from there. Ask them to describe in degrees the direction they are facing upon each turn. Ask them to tell you how many degrees they have turned with each new move.

Inform the students that in this lesson they will investigate the `Location` class in GridWorld. Tell them that they will complete a worksheet based on the `Location` class and then design an application for the `Location` class different from GridWorld.

Lesson Development

Arrange students into pairs and give each pair a compass. Have them follow a set of directions to find an object hidden in the classroom. Each pair will have a unique set of directions and object to find (toy or edible rock, flower, or bug). The directions given in terms of objects, constants and methods of the `Location` class instruct the pairs to face different compass directions, turn through different compass angles and move a specified number of steps to find their hidden object. This part of the activity should take 5 to 7 minutes.

When students have found their hidden objects, discuss the meaning of the instructions and the constant values from `Location`. Give the students a copy of Part 3 and the GridWorld Testable Application Programming Interface (API) containing the `Location` class. They will continue working in pairs to understand `Location`. Ask them to write down any questions they have and submit them to you. Address the questions one at a time during a class discussion.

Student pairs will then design an application of the `Location` class other than the case study. The design phase is the focus in this exercise. Student pairs will present their application designs to the class, specifying the problem being solved and how `Location` is used in the design. The class will question the presenters, who are expected to defend and justify their applications.

The design process does not require actual coding and implementation. Extra credit could be awarded to pairs that develop a running program within the following 10 days.

Accommodations

Pairs will be formed in such a way that students who need extra help are paired with students who are likely to help them without taking over. Students who are more capable may be selected to help others in the pairs or may work on coding and implementing their designs.

Closure

Students will be asked to explain strategies they used find their hidden object. Discuss how the pairs investigated the `Location` class. Review the `Location` class methods and constants.

Assignment

Students will complete the "Location Reading Worksheet" and turn it in at the beginning of the next class.

Assessment

- Diagnostic assessment occurs through the anticipatory set activity, which allows the instructor to assess the students' prior knowledge of compass locations.

- Formative assessment takes place during the lesson as the instructor observes and questions the students' progress in completing the assignment.
- Summative assessment is achieved through the worksheet.

Evaluation

Formulate questions you will ask yourself after teaching this lesson, such as the following:

- Did the compass activity at the beginning of class activate students' prior knowledge of compass directions and angles?
- Did the search for the hidden objects cause the students to use and understand `Location` class objects, constants and methods?
- Did I perform my role as facilitator in a way that was helpful but not interfering?
- What worked well with this lesson?
- Is there anything I would change when I present this lesson again?

Part 3: Activity/Lesson Plan 5 — GridWorld Part 3 (continued)

A surprising number of fundamental concepts can be introduced to students in a highly motivating way through the case study. This lesson examines the `Grid` interface and introduces preconditions. It is taught using the direct instruction, due to the fact that most of the material is new to the students. In the latter part of the lesson, students work cooperatively to understand method signatures.

This lesson could (and perhaps should) be divided into two separate lessons. The first would be direct instruction about interfaces and preconditions. The second would be social interaction (cooperative learning) in which the students develop skills in reading and interpreting an API, in particular those listed in the GridWorld Testable API.

Learning Goals

- Students will understand the purpose and appreciate the design advantages of creating `Grid` as an interface.
- Students will understand the implications of preconditions.
- Students will work together to develop skills for reading an API and analyzing method signatures.

Lesson Objectives

- Given the "Grid Reading Worksheet," the student will complete the worksheet with 90 percent accuracy.
- Given the GridWorld Testable API, the student will be able to describe what must be true when a method is invoked from examining the precondition with 100 percent accuracy.
- Given the GridWorld Testable API, the student will work with a small group to list the methods, preconditions, parameters, parameter types and return types of the `Grid` interface and the Location class, describing the differences between methods whose names are nearly the same with 100 percent accuracy.

Materials

- First Day Role Play
- GridWorld Classes and Interfaces Grid Reading Worksheet (see Appendix I)
- GridWorld Student Manual Testable API (http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/compsci_a_exam_appendix.pdf)

Anticipatory Set

Begin the class by recalling one aspect of the First Day Role Play. Have one student be a Calculator and another, a Lazy Calculator with a Helper. (Both of these are classes in the First Day Role Play.) Issue the same command to each player ("Alex, add 2, 4." "Pat, add 2, 4."). Each should return the correct value, determined in different ways. Remind the class that the command and the parameters were the same in both cases, so you didn't have to change the way the request was made, and in both cases the result was predictable and consistent.

Now ask the students if they recall the choices they have for a grid when they run any of the bug runner classes. They should reply "a bounded grid and an unbounded grid." Ask them if `Bug` behavior is predictable and consistent in either grid, and how the two grids are alike and different; then have them list their responses on the board. They may run `BugRunner` or some other runner class with each grid and test out their answers. Inform the students that the grid in GridWorld is an interface, not a class, and that the bounded grid and the unbounded grid are classes that are said to "implement" the `Grid` interface. Inform them that in this lesson they will learn about interfaces in Java and study the `Grid` interface as an example of how to design and use an interface.

Lesson Development

Explain that interface types are used to express operations which are common to related classes. The interface specifies a set of methods and their signatures but does not provide implementation. Each class that implements the interface must include the common methods and their signatures, but each class most likely implements the

methods differently. Continue along these lines, discussing how an interface works and emphasizing the design advantages of creating an interface. Tell the students that an interface is not a class because there is no constructor and there are no instance variables, only method signatures. An interface doesn't have "state" and the behavior isn't completely specified. Talk about the ease with which a student can select the bounded or unbounded grid and how the `Grid` interface allows this to happen. The methods in the interface are invoked the same way for either type of grid, and the result is consistent and predictable. All of this is direct instruction, but you should continually ask questions to check for student understanding. Encourage your students to express in their own words the points you have made, and use any misunderstandings you hear as an opportunity to activate their higher-level thinking skills and help them synthesize the concepts.

With the students, draw a high-level diagram of the interface and the classes that implement it. Show the method signatures but no code. This is "we do" component of direct instruction.

Introduce the students to the purpose of a precondition. Think of a precondition as a promise made to the method, a guarantee that a given condition will be true when the method is invoked. Ask the class how that guarantee can be made. Ask them about the responsibility of the code that calls the method. Only if the precondition is met does the method guarantee a correct result.

In terms of the AP Exam (or good programming, for that matter), remind students that when they write code that invokes a method having a precondition, they must make sure their code assures that the precondition is met before making the method call.

Discuss method signatures in the `Grid` interface. (This is more of the "we do" part of the lesson.) Look only at the signatures, the part they need to know how to use from the GridWorld Testable API.

For the "you do" part of the lesson, have the students complete the Grid worksheet. Move about the room, checking to see how they are doing, offering suggestions or answering questions as needed.

Form small cooperative-learning groups of three or four students. The groups will analyze the `Grid` interface method signatures and, for each signature, list the name, number of parameters, type and name of each parameter, and return type. For similar signatures, have the groups write down exactly what each method does and how the return values differ. If groups complete their analysis of the `Grid` method signatures early, ask them to analyze the Location class methods in the same way.

Accommodations

Groups will be assigned in such a way that a less knowledgeable or able student will have the support of a more knowledgeable peer. All students will be responsible for understanding the interface. Those who catch on quickly will be expected to help anyone in the group who struggles with the concepts. Monitor groups to be sure this is happening.

Closure

Ask students to summarize what they learned about interfaces in today's lesson. Randomly select a spokesperson from each cooperative learning group to list the group's findings about one of the method signatures. Check for agreement between groups. Remind the students to always be aware of the properties of a method signature, especially when reading or writing code on the AP Exam.

Assignment

For homework, students will complete a similar analysis of the `Comparable` interface. The homework will be collected and graded.

Assessment

- Diagnostic assessment occurs during the anticipatory set when students are asked to recall their observations about the role play and the grid.
- Formative assessment is ongoing during the lesson. Didactic questioning during the direct instruction segment, followed by teacher observation as the groups analyze the Grid interface, provides continuous feedback to the teacher.
- Summative assessment is met through the Grid worksheet, group reports and individual homework assignments.

Evaluation

Formulate questions that you will ask yourself after teaching this lesson:

- Did I get the students' attention and activate their prior knowledge through my introduction to the lesson?
- Did the students understand the differences between an interface and a class that implements the interface?
- Were the students able to analyze the `Grid` interface and understand the method signatures? Do they understand the significance of the preconditions? Will they be able to use the methods correctly?
- Is there anything I would change about this lesson in the future?
- What did I do right that I would like to remember and repeat?

Part 3: Activity/Lesson Plan 6 — GridWorld Part 3 (continued)

In this lesson the students will investigate the `Actor` class and classes that extend `Actor`. They will discover the important ideas in creating new subclasses and the key idea of overriding the `act` method. This lesson uses the Information Processing Model because students will investigate reading material on their own to answer the question of finding two sets of five important ideas in the reading assignment.

This is the first of a two-part lesson. This plan and Activity/Lesson Plan 7 should be presented on consecutive days. They can be combined if the school is on block scheduling.

Learning Goals

- The students will understand the `Actor` class and design principles associated with `Actor`.
- The students understand inheritance and will learn how to extend a superclass to create a subclass.

Lesson Objectives

- Given Part 3 of the Student Manual and Student Manual Appendixes B and C, the student will be able to investigate inheritance and class design and make two lists, each containing five important ideas they have found.
- Given the Actor Reading Worksheet, the student will complete the worksheet with 90 percent accuracy.
- Given the Extending Actor Reading Worksheet, the student will complete the worksheet with 90 percent accuracy.

Materials

- GridWorld Part 3: GridWorld Classes and Interfaces (page 16) (http://www.collegeboard.com/student/testing/ap/compsci_a/case.html)
- GridWorld Student Manual Testable API (http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/compsci_a_exam_appendix.pdf)
- GridWorld Student Manual Testable Code for APCS A/AB (http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/compsci_a_exam_appendix.pdf)
- Actor Reading Worksheet (see Appendix J)
- Extending Actor Reading Worksheet (see Appendix K)

- Play-Doh in several colors
- Colored pipe cleaners
- Large square grid paper (1-inch or more)

Anticipatory Set

When the students arrive in the classroom, they will find small containers of Play-Doh and colored pipe cleaners (see Appendix B) available at a central table. They will use their imagination to create bugs from the craft materials. Each student will take a sheet of large scale (one inch or larger) graph paper and will design a movement pattern for his or her bug. Remind the students that their bugs must remain within the grid. Students will give their bugs a descriptive name that characterizes the bug's unique movement style, and they will share their creations with the class. Allow about 7 minutes for this activity.

Inform the students that, in this lesson, they will learn about the `Actor` class, inheritance and code reuse. `Actor` is a superclass, and `Bug` is a subclass of `Actor`. Tell the students that they will learn how to create their own subclass and implement that class. In this lesson, they will discover some important design principles and then, in a subsequent lesson, they will apply those principles in a group assignment. Students will then design their own class.

Lesson Development

Start by having the students read the `Actor` and Extending `Actor` sections of Part 3 of the GridWorld Student Manual; examine the `Actor`, `Rock` and `Flower` API in the GridWorld Testable API; and examine the `Bug` and `BoxBug` subclasses in the GridWorld Testable Code for APCS A/AB. As they read, instruct the students to individually make lists of the important things they learn about the `Actor` class, about extending Actor and about the classes that extend `Actor`. Ask them to think about class design and what a subclass inherits from the superclass. Tell them that when they finish the reading they are to select the five ideas they think are the most important from each list and to make new lists, which will be collected, putting those five ideas in order from most to least important. Allow 15 minutes for this work, and then collect the lists.

Have the students return to the reading and complete the Actor and Extending Actor Reading Worksheets. While they are doing that, compile the student lists into two lists from the most frequent to the least frequent responses. You may need to combine very similar responses into single statements. Arrange the compiled lists in order of most to least important, using your educated opinion. (You may need to create the compiled lists after class. You will be using these lists to introduce the lesson tomorrow.) Check the students' progress with the worksheets and provide help as needed.

Accommodations

For students who need extra help making the lists, you can offer suggestions as you move about the room monitoring student progress. Students who need help with the worksheets will be encouraged to discuss the questions with you during your open period or after school..

Closure

At the end of class, ask the students what they learned about the `Actor` superclass and the `Rock`, `Flower`, `Bug` and `BoxBug` subclasses from their investigation of the API and code. Ask them to describe how they investigated the reading and code to find the main ideas. Summarize the design principles found in this part of the case study.

Assignment

Complete the Actor and Extending Actor Reading worksheets for homework. These will be collected and graded.

Assessment

- Diagnostic assessment concerning what students know about the classes will occur as you listen to the students talk about their bug models.
- Formative assessment will be ongoing during the lesson as you observe students making their lists, ask individual students questions, and respond to their comments and questions.
- Summative assessment will occur through the lists and worksheets they complete and hand in.

Evaluation

Formulate questions you will ask yourself after teaching this lesson:

- Did making the bug models, creating a movement pattern for them, and naming the models to represent their pattern focus the students' attention on the lesson and activate prior knowledge?
- Did I give adequate instructions for the investigative part of the lesson?
- Did reading for the big ideas and then reading to complete the worksheets accomplish my goal of having students learn about design principles?
- Did having the students investigate on their own achieve my goals? Did they find the information I intended for them to find? Did they learn anything about how to investigate?
- What would I do differently if I taught this lesson again?

- What did I do well and what would I repeat if I taught this lesson again?

Note: Here are a few possible ideas students might list.

About Actor:

- Use `putSelfInGrid` to insert an `Actor` into the grid.
- Use `removeSelfFromGrid` to remove an `Actor` from the grid.
- The `act` method defines an `Actor`'s behavior.
- An `Actor` has color, location and direction.
- The `moveTo` method allows the actor to move to any valid location.

About extended `Actor` (They probably will find many more ideas for this category):

- `Bug`, `Rock` and `Flower` extend `Actor`.
- `Rock` overrides `act` by doing nothing — empty method.
- `Flower` overrides `act` by darkening on each step.
- The `canMove` method in `Bug` determines if a `Bug` can move.
- The `instanceOf` method determines whether an object is a member of a class.

Part 3: Activity/Lesson Plan 7 — GridWorld Part 3 (continued)

This is a continuation of Activity/Lesson Plan 6 and should be implemented on the day following Plan 6. Students will work in small groups to design and implement the Jumper exercise found in Part 3 of the Student Manual. They will then design their own subclasses, determining which class they should extend, and making design decisions about the methods to override or to create. This lesson uses the Social Interaction (Cooperative Learning) Model.

Learning Goals

- The students will learn how to design and implement a subclass. They will learn to use inherited methods and how to override methods from the super or parent class.
- The students will learn to work cooperatively to design and implement a subclass.

Lesson Objectives

- Given the Jumper assignment from Part 3 of the GridWorld Student Manual, the student will work cooperatively in a small group to design and implement the Jumper class with 100 percent accuracy.
- Given Part 3 and Appendixes B and C of the GridWorld Student Manual, the student will successfully design and implement his or her own subclass of `Actor`, using an appropriate reimplementation of the `act` method and following a model similar to `BoxBug` and `Jumper`.

Materials

- GridWorld Part 3: GridWorld Classes and Interfaces (page 16) (http://www.collegeboard.com/student/testing/ap/compsci_a/case.html)
- GridWorld Student Manual Testable API (http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/compsci_a_exam_appendix.pdf)
- GridWorld Student Manual Testable Code for APCS A/AB (http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/compsci_a_exam_appendix.pdf)
- Results of the previous day's lesson (five important ideas in two categories) set up as hidden answers to a version of the Family Feud game. These can be on an overhead transparency or covered on the board.

Anticipatory Set

Tell the students that they are going to play Family Feud. Inform them that you have taken their "Top Five" important ideas in the two categories, `Actor` and Extending `Actor`, from their previous lesson and tabulated the results. Divide the students into two "families" and play two rounds based on those lists. Be sure that the teams are balanced in terms of overall academic strength. The topics will be: The Five Most Important Ideas About the `Actor` Class and The Five Most Important Ideas About Extending the `Actor` Class. Each team will choose a captain. Play the Five Most Important Ideas About the `Actor` category first. Follow the game rules for *Family Feud*; if you're not sure how the game is played, see the Resources section for links to rules and an online version, or ask your students.

Tell the class that, now that they have reviewed what they found out about `Actor` and extending `Actor`, they are going to work in small groups to learn how to design and develop a subclass of `Actor`: the `Jumper` class.

Lesson Development

Separate the class into small groups. Tell the students to follow directions found in Part 3 of the GridWorld Student Manual for the `Jumper` exercise, deciding within the group how to answer the questions, design the class and implement the code. They are to

answer the questions in writing (using a word processor is preferred). Their answers will be collected as well as their completed code. Urge them to use the GridWorld Testable Code for APCS A/AB and Testable API as references. While they work, walk around the classroom monitoring group progress and answering questions as needed. Be sure all students are actively involved. Each student should take a turn entering code while the others make suggestions and check for errors. For the coding part, you may decide to have the students work in pairs with one student typing and the other navigating (making suggestions and watching for errors). If you use this technique, insist that the pairs switch roles at least once during coding. Tell the class their grade will be based on both the success of the completed project and their level of participation. They will be evaluating themselves and the other group members at the end of class. Collect their answers to the Jumper exercise questions; code may be submitted electronically.

Accommodations

Students will be assigned to groups in such a way that a more able or knowledgeable student can help a less able student. Groups will be reminded that they are to cooperate and be sure all members understand every step of the assignment. The teacher should also monitor the groups and provide help as needed.

Closure

Groups with working code will run their applications for the class to see. Ask the groups to name one thing they learned through this exercise. Each group must contribute something not already mentioned. Ask them to talk about how the groups answered the questions, designed their `Jumper` class and implemented the code. What worked best in this group situation? Was anything difficult? Did they overcome the difficulties? How? Remind them that programmers usually work in teams in the workplace and that it's essential that they all cooperate and contribute. Give them a copy of the individual group member evaluation form (see Materials list above) to complete.

Assignment

Each student will design and implement a subclass of his or her own choice. They may work on this at home and in class the following day. Students should also write a journal entry, describing what they have learned in Part 3 of the GridWorld case study. (See Materials list above for a description.)

Assessment

- Diagnostic assessment occurs during the anticipatory set as students participate in the game. Ensure that everyone is involved in deciding on a group response to the game.
- Formative assessment occurs during the lesson as the teacher moves about the room observing, listening and questioning the groups.

- Summative assessment is through the answers to the questions turned in by each group, the successful design and implementation of the `Jumper` class (examining the code as well as the run), and through the group evaluation forms.

Evaluation

Formulate questions you will ask yourself after teaching this lesson:

- Did the Family Feud game grab the students' attention, focus them on the lesson and activate prior knowledge?
- Were my directions to the groups clear? Did they understand how to work together to solve the problem?
- During closure did I ask appropriate questions to help the students summarize the lesson? Did I help them understand the importance of working together? Did we discuss techniques for success in cooperative groups?
- Is there anything I will change if I teach this lesson again?
- Is there anything that went very well and that I will repeat in future lessons?

Appendix A

KWL Chart for Introducing GridWorld Activities

We are about to begin a study involving bugs, critters and other objects situated in a world composed of a two-dimensional grid. Thinking about this scenario, list in the left-hand column of the chart below what you already *know* about this topic. In the middle column, list what you *want* to know about this topic. The right-hand column is where you will list what you have *learned* upon completing this module. You have three minutes to fill in the first two columns before we share what each of you has written with the class.

KNOW	WANT to know	LEARNED

Appendix B

Resources

- GridWorld Student Manual
- GridWorld Installation Guide
- Oriental Trading Company (http://www.orientaltrading.com) — an excellent source of many items at very reasonable prices
  - gummy bugs
  - costume pieces for bug or critter items, antennae, hats, etc.
  - compasses
- Games for the classroom played on a grid (http://math2.eku.edu/Greenwell/MAT303/)
- First Day Role Play and GridWorld Role Play (http://www.cs.sbu.edu/dlevine/RolePlay/roleplay.html)
- Sheet protectors (which can be purchased at stores such as Staples and Office Max)
- Costume pieces (which can be purchased from costume shops, craft stores and dollar stores, and at online sites, such as Oriental Trading Company, for very little money)
- Small, very inexpensive containers of Play-Doh or similar brands (which can be found at "dollar" stores; you can also make your own using recipes found online)
- Bags of multicolored pipe cleaners (which can be found at craft stores such as Michaels)
- Rules for playing *Family Feud* can be found at: http://en.wikipedia.org/wiki/Family_Feud#Basics
- An online version of *Family Feud* is available at: http://www.gamefools.com/onlinegames/free/FamilyFeud2.html

Lab Grading Rubric

Name _____

Lab # _____ Grade _____/10

Description _____

Each item has a value of one point. Half-points may be given.

Code:
Correct class structure
Descriptive identifiers
Well commented
Logically structured
Well-designed test class

Implementation:
Does the code run?
Test cases provided?
Output matches test cases?
Output readable?
Text output grammatically correct?

Comments:

Individual Feedback on Cooperative Learning Group Work

Name_____Project_____

Group Name _____

What I did to help my group's success today:

My participation was:

_____ 5 _____4 _____3 _____2 _____1
great good OK poor not acceptable

What I will do the same next time we work together:

What I will do differently:

The rate of my group members' participation:

Name

_____ _____5 ____4 ____ 3 ____ 2 ____1

_____ _____5 ____4 ____ 3 ____ 2 ____1

_____ _____5 ____4 ____ 3 ____ 2 ____1

_____ _____5 ____4 ____ 3 ____ 2 ____1

_____ _____5 ____4 ____ 3 ____ 2 ____1
(5 = the highest, 1 = the lowest)

My suggestions, ideas on how to improve CL group work, comments on the project:

Appendix C

GridWorld Case Study Part 1 — Running the Demo Observing and Experimenting in GridWorld

1. Set up GridWorld on your computer.

Running the Demo

2. Read pages 4–6 of the Student Manual.
3. Run the `BugRunner.java`. If the bug is facing the bounding wall or a rock, close the GUI and repeat Step 3.
4. What two actors appear in the GridWorld GUI?
5. How many cells are in the Grid?
6. Press the **Step** button.
7. What actor(s) appears in the Grid?
8. Where is the new actor relative to where the bug was?
9. Where did the bug move?
10. How has the rock changed?
11. Press the **Step** button again.
12. How many actors are in the world?
13. How has the actor in the bug's original location changed?
14. How has the bug changed?
15. How has the rock changed?
16. How many actors will be in the grid if you press the **Step** button again?
17. Press the **Run** button.
18. How is the **Run** button different from the **Step** button?
19. What effect does the slider have?
20. Would the grid fill up with flowers if the GUI was left running indefinitely? Why?

21. Restart the BugRunner.java . Press on an empty cell. How many constructors are in the drop-down menu?
22. How many of the constructors are for Actor?
23. Press the Step button.
24. What new constructors are now in the drop-down menu?
25. What determines the list of constructors in the drop-down menu?
26. Select the constructor info.gridworld.actor.Actor(). What image appears?
27. When a constructor requires a parameter what appears on screen?
28. Complete the **Do You Know**? Set 1.

GridWorld Case Study Part 1 — Running the Demo Observing and Experimenting in GridWorld

Reading Worksheet (Solutions)

1. Set up GridWorld on your computer.

Running the Demo

2. Read pages 4–6 of the Student Manual.
3. Run the `BugRunner.java`. If the bug is facing the bounding wall or a rock, close the GUI and repeat Step 3.
4. What two actors appear in the GridWorld GUI? `Bug` and `Rock`
5. How many cells are in the grid? 100 cells
6. Press the **Step** button.
7. What actor(s) appears in the grid? A `Flower`
8. Where is the new actor relative to where the bug was? The flower is at the `Bug's` former location.
9. Where did the bug move? Forward one cell
10. How has the rock changed? The rock has not changed.
11. Press the **Step** button again.
12. How many actors are in the world? In most cases, four (`Bug`, `Rock` and two `Flowers`). However, if the `Bug` is at an edge or facing an adjacent rock, there would only be three actors after Step.
13. How has the actor in the bug's original location changed? The flower's color has darkened.
14. How has the bug changed? It moved forward one cell if it wasn't at an edge or facing an adjacent rock.
15. How has the rock changed? The rock has not changed.
16. How many actors will be in the grid if you press the **Step** button again? Five (another flower), unless the bug is at an edge or facing an adjacent rock.
17. Press the **Run** button.

18. How is the **Run** button different from the **Step** button? The **Run** button carries out a series of steps.
19. What effect does the slider have? It changes the delay between steps.
20. Would the grid fill up with flowers if the GUI was left running indefinitely? Why? No, the flowers only appear where the bug moved, and the bug will move in a repeated path.
21. Restart the `BugRunner.java`. Press on an empty cell. How many constructors are in the drop-down menu? Five
22. How many of the constructors are for `Actor`? One
23. Press the **Step** button.
24. What new constructors are now in the drop-down menu?
    `info.gridworld.actor.Flower(java.awt.Color)`
    `info.gridworld.actor.Flower()`
25. What determines the list of constructors in the drop-down menu? The objects that have been placed in the grid.
26. Select the constructor `info.gridworld.actor.Actor()`. What image appears? Theater Mask for comedy
27. When a constructor requires a parameter, what appears on screen? Dialogue window
28. Complete the **Do You Know?** Set 1.

Appendix D

GridWorld Case Study Part 1 — Exploring Actor Observing and Experimenting in GridWorld Reading Worksheet

Exploring Actor State and Behavior

1. Read pages 7–8 of the Student Manual.
2. What is in the drop-down menu when you click on a cell containing an `Actor`?
3. Click on the bug. Explore the methods.
4. How many methods are in the drop-down menu when you click on the cell containing the bug?
5. Which four methods are **not** inherited from the `Actor` class?
6. Which methods are accessor methods?
7. What is the prefix of all modifier methods?
8. Which accessor methods do not have a companion modifier method?
9. Explain the difference between what happens when you invoke the method `setDirection(45)` and the `turn` method?
10. Change the direction so that all the actors are facing the left edge of the grid. What method did you use?
11. Complete the **Exercises** on page 8.
12. How many `Actors` can be in a cell?

13. Try a negative number as an argument in `setDirection`.

Degrees	Compass Directions
-90	
-270	
-180	
-45	
-315	
-360	
-225	
-135	
-720	

14. How many different directions can a bug appear to face?

GridWorld Case Study Part 1 — Exploring Actor Observing and Experimenting in GridWorld Reading Worksheet (Solutions)

Exploring Actor State and Behavior

1. Read pages 7–8 of the Student Manual.
2. What is in the drop-down menu when you click on a cell containing an `Actor`? The methods that you can invoke on this `Actor` object.
3. Click on the bug. Explore the methods.
4. How many methods are in the drop-down menu when you click on the cell containing the bug? 14
5. Which four methods are **not** inherited from the `Actor` class? `act`, `canMove`, `move` and `turn`
6. Which methods are accessor methods? `getColor`, `getDirection`, `getGrid` and `getLocation.toString` are often considered to be accessor methods because they return information about the object and do not change the state of the object.
7. What is the prefix of the modifier methods? `Set`
8. Which accessor methods do not have a companion modifier method? `getGrid` and `getLocation`
9. Explain the difference between what happens when you invoke the method `setDirection(45)` and the `turn` method? `setDirection(45)` is inherited from `Actor` and turns the bug northeast. The `turn` method rotates the bug 45 degrees right.
10. Change the direction so that all the actors are facing the left edge of the grid. What method did you use? `setDirection(270)`
11. Complete the **Exercises** on page 8.
12. How many `Actors` can be in a cell? One

13. Try a negative number as an argument in `setDirection`.

Degrees	Compass Directions
-90	West
-270	East
-180	South
-45	Northwest
-315	Northeast
-360	North
-225	Southeast
-135	Southwest
-720	North

14. How many different directions can a bug face? Eight

Appendix E

Journal Descriptions

Submitting the journal electronically has the advantage of the teacher being able to use track changes for comments.

Journal 1

Start your journal by describing your experience in class today. Write about three things you learned in your investigation and one question you have that was not answered so far. Discuss how today's lesson relates to you and to what you expect to learn in AP Computer Science.

Journal 2

Write about your role in the role play. What did you learn from playing the part? What were your responsibilities in that role? With which other roles did your role collaborate? How did you feel when you were playing the part? What value did the role play have for you?

Journal 3

Write about what you learned from the readings, activities and assignments in Part 3 of the case study. What was especially interesting or motivating for you, and why? What did you learn about working with others in the class? How has Part 3 increased your understanding of the principles of computer science? How will you apply concepts from Part 3 to classwork outside the case study? Is there anything you wish would have been covered in more detail? What unanswered questions do you have? Somewhere in your journal begin one sentence with the words "I wonder … " and another with the words "I believe … "

Rubric for Reflective Journal

Name_____

CRITERIA		POINTS	Self-Evaluation
1. Content of the Entries		**10**	
• Substantive reactions to the reading material	**4**		
• Thoughtful reflections about class discussions and activities	**3**		
• Insightful applications of course content	**3**		
2. Structure of the Journal and Each Entry		**6**	
• Has a reflective title	**1**		
• The entry reflects the major concepts discussed during the period of reflection	**2**		
• Evaluates what has been learned introspectively	**3**		
3. Originality		**4**	
• Appealing visual presentation of the text entries	**2**		
• Mode of expressing the thought in writing	**2**		
Total		**20**	

Comments:

Appendix F

GridWorld Case Study Part 2 — Bug Class Bug Variations Reading Worksheet

Methods of the `Bug` Class

1. Read page 10 of the Student Manual.
2. What three methods specify how the bugs move and turn?
3. What methods are invoked in the `act` method?
4. Why does the bug's `act` method invoke the `canMove` method?
5. When does a bug turn?
6. Draw a grid where a bug can never move. Explain how the bug would act in this grid.
7. What happens to a flower when a bug moves forward onto it?
8. When a bug moves, what does it leave in its previous location?
9. How the bug moves is determined by which method?

Extending the `Bug` Class

10. Read page 11 of the Student Manual.
11. When creating new types of bugs, which method is most likely to be overridden?
12. What does it mean to have an *overridden* method?
13. How many new methods need to be added to a new `Bug` class?
14. What are the new instance variables for the `BoxBug` class?
15. What are the values of the `sideLengths` for the two `BoxBugs` in the picture?
16. Write a `turnAround` method for `BackAndForthBug`. `turnAround` that makes a `BackAndForthBug` face the opposite direction.
17. Write an act method for `BackAndForthBug`. When a `BackAndForthBug`'s path is blocked, it should turn around. Otherwise it acts like a `Bug`.
18. Do questions **Do You Know?** Set 2.

GridWorld Case Study Part 2 — Bug Class
Bug Variations
Reading Worksheet (Solutions)

Methods of the `Bug` Class

1. Read page 10 of the Student Manual.
2. What three methods specify how the bugs move and turn? `canMove`, `move` and `turn`
3. What methods are invoked in the `act` method? `canMove`, `move` and `turn`
4. Why does the bug's `act` method invoke the `canMove` method? To test if the bug can move forward into an empty cell or a cell containing a flower
5. When does a bug turn? When `canMove` returns false
6. Draw a grid where a bug can never move. Explain how the bug would act in this grid. Possible solution: 1 x 1 grid. A bug would turn every time the act was invoked.
7. What happens to a flower when a bug moves forward onto it? The flower is removed from the grid.
8. When a bug moves, what does it leave in its previous location? A flower
9. How the bug moves is determined by which method? `act` (`move`, `canMove` and `turn` play roles, too!)

Extending the Bug Class

10. Read page 11 of the Student Manual.
11. When creating new types of bugs, which method is most likely to be overridden? `act`
12. What does it mean to have an *overridden* method? A subclass's method has the same name and parameters as the superclass, but gives its own definition of the method.
13. How many new methods need to be added to a new `Bug` class? 0
14. What are the new instance variables for the `BoxBug` class? `sideLength` and `steps`
15. What are the values of the `sideLengths` for the two `BoxBugs` in the picture? 6 and 3

16. Write a `turnAround` method for `BackAndForthBug`. `turnAround` that makes a `BackAndForthBug` face the opposite direction.

```
public void turnAround()
{
        setDirection(180 + getDirection());
}
```

17. Write an act method for `BackAndForthBug`. When a `BackAndForthBug`'s path is blocked, it should turn around. Otherwise it acts like a `Bug`.

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

18. Do questions **Do You Know?** Set 2.

Appendix G

GridWorld Case Study Part 2 — Runner Class Bug Variations Reading Worksheet

Runner Classes

1. Read page 12 of the Student Manual.
2. What is required to observe the behavior of one or more actors?
3. What is an `ActorWorld` object?
4. What method is used to place an object into the grid?
5. What two `ActorWorld` methods appear in the `BoxBugRunner`?
6. What does it mean to have an *overloaded* method?
7. What must you do to demonstrate your own classes extending `Bug`?
8. Explain how the two different `add` methods work.
9. Do the **Exercises** at the end of Part 2.
10. Write a runner class for a `StudentBug` class. The `StudentBug` has a constructor that takes a color. Add a `Bug`, two `StudentBugs` and a `Rock` into the world, and test your class.

GridWorld Case Study Part 2 — Runner Class
Bug Variations
Reading Worksheet (Solutions)

Runner Classes

1. Read page 12 of the Student Manual.
2. What is required to observe the behavior of one or more actors? A "runner" class
3. What is an `ActorWorld` object? The object that keeps track of everything in the grid
4. What method is used to place an object into the grid? `Add`
5. What two `ActorWorld` methods appear in the `BoxBugRunner`? `Add` and `show`
6. What does it mean to have an *overloaded* method? Methods have the same name but differ from each other in terms of the type, number and/or order of parameters.
7. What must you do to demonstrate your own classes extending `Bug`?
   You must write a "runner" class.
8. Explain how the two different `add` methods work. The `add` method with `Actor` and `Location` parameters places an actor at a specific location. The `add` method with only an `Actor` parameter places an actor at a random empty location.
9. Do the **Exercises** at the end of Part 2.
10. Write a runner class for a `StudentBug` class. The `StudentBug` has a constructor that takes a color as a parameter. Add a `Bug`, two `StudentBugs` and a `Rock` into the world, and test your class.

```
public class StudentBugRunner
{
    public static void main(String[] args)
    {
        ActorWorld world = new ActorWorld();
        Bug ann = new Bug();
        StudentBug bob = new StudentBug(Color.BLUE);
        StudentBug claire = new StudentBug(Color.YELLOW);
        Rock darrel = new Rock();
        world.add(new Location(2, 5), ann);
        world.add(new Location(3, 5), bob);
        world.add(new Location(4, 6), claire);
        world.add(new Location(2, 1), darrel);
        world.show();
    }
```

Appendix H

GridWorld Case Study Part 3 — Location GridWorld Classes and Interfaces Reading Worksheet

1. Read page 16 of the Student Manual.
2. What does a grid contain?
3. Any instances of a class that extends the `Actor` class must ***is-a***
4. How many classes implement the `Grid` interface?
5. What does an `Actor` know?
6. Answer the following **is-a** questions:
    a. `Bug` is-a_____ .
    b. `Flower` is-a _____ .
    c. `BoundedGrid` is-a _____ .
    d. `Actor` is-a _____ .
7. Answer the following **has-a** questions:
    a. Actor has-a _____ .
    b. Bug has-a_____ .

The `Location` Class

8. Read pages 16–19 of the Student Manual.
9. Every `Actor` that appears in the grid must have a_____ .
10. Instances of what class represent locations in the grid?
11. What does *encapsulate* mean?
12. The `Location` class encapsulates what aspect of the grid?
13. What relationship does `Location` provide methods for?
14. Cells that run vertically up and down the grid are called what (rows or columns)?
15. What is the smallest row number?

16. What unit of measure is used to represent compass directions?
17. How many constants based on compass directions are provided in the `Location` class?

Question 18 refers to the following line of code from the `Location` class.

```
public static final int NORTH = 0;
```

18. What does `static final` mean?
19. Write an example of how to use the `Location` constant `EAST` in some bug class.
20. Write a segment of code to have `Bug bugBoy` change its color to blue if it is facing `NORTH` and to change its color to red if it is not facing north.
21. What is the data type of `Location.SOUTHWEST`?
22. What is the 0 direction for the `Location` class?
23. What is the 450 direction for the `Location` class?
24. What is the value of `Location.NORTHWEST`?
25. What is the numerical value of `Location.EAST + Location.SOUTH`?
26. How many constants based on the turn angle are provided in the `Location` class?
27. What is the data type of `Location.LEFT`?
28. What is the constant for a 0 degree turn angle for the `Location` class?
29. What is the value of `Location.HALF_LEFT`?
30. What is the value of `Location.FULL_CIRCLE + Location.HALF_LEFT`?
31. Write a segment of code to get an actor to turn around.
32. What two parameters are required of the `Location` constructor?
33. Create a reference `Location` named `loc1`. Instantiate the `Location` object with a row number of 3 and a column number of 2.
34. How many new accessor methods exist for the `Location` class?
35. Write the code to output the row for `loc1`.
36. Write the code to output the column for `loc1`.
37. List the `Location` methods that give information about the relationship between locations and directions.

38. What direction is an actor traveling if rows are increasing and columns are decreasing?
39. What is compared in the `equals` method of `Location`?
40. What is compared in the `compareTo` method of `Location`?
41. Write a segment of code to output the name of `Bug bugBoy` or `Bug bugGirl`, with the lowest `Location`.
42. When will the `loc1.compareTo(loc2)` method return a positive number? Assume: `loc1 = new Location(row1,col1) and loc2= new Location(row2,col2)` (Describe every situation.)
43. Do questions **Do You Know?** Set 3.

GridWorld Case Study Part 3 — Location
GridWorld Classes and Interfaces
Reading Worksheet (Solutions)

1. Read page 16 of the Student Manual.
2. From your reading, what does a grid contain? Instances of the `Actor` class
3. Any instances of a class that extends the `Actor` class must ***is-a***. `Actor`
4. How many classes implement the `Grid` interface? Two
5. What does an `Actor` know? Their `Grid`, their current location, their direction and their color
6. Answer the following **is-a** questions:
   a. `Bug` is-a `Actor`.
   b. `Flower` is-a `Actor`.
   c. `BoundedGrid` is-a `Grid`.
   d. `Actor` is-a `Object`.
7. Answer the following **has-a** questions:
   a. `Actor` has-a `Location` or `Grid` (or `Color` or `int (direction)`).
   b. `Bug` has-a `Location` or `Grid` (or `Color` or `int (direction)`).

The `Location` Class

8. Read pages 16–19 of the Student Manual.
9. Every `Actor` that appears in the grid must have a `Location`.
10. Instances of what class represent locations in the grid? `Location`
11. What does *encapsulate* mean? The way to interact with a class is visible, but how it's implemented is hidden. "Don't tell me how you do it; just do it."
12. The `Location` class encapsulates what aspect of the grid? Coordinates
13. What relationship does `Location` provide methods for? Relationship between locations and compass directions
14. Cells that run vertically up and down the grid are called what (rows or columns)? Columns
15. What is the smallest row number? 0

16. What unit of measure is used to represent compass directions? Degrees

17. How many constants based on compass directions are provided in the `Location` class? Eight

Question 18 refers to the following line of code from the `Location` class.

```
public static final int NORTH = 0;
```

18. What does `static final` mean? It denotes a constant, so the value of `North` cannot be changed.

19. Write an example of how to use the `Location` constant `EAST` in some `Bug` class.
`Location.EAST`

20. Write a segment of code to have `Bug bugBoy` change its color to blue if it is facing north and change its color to red if it is not facing north.

```
if (bugBoy.getDirection() == Location.NORTH)
    bugBoy.setColor(Color.BLUE);
else
    bugBoy.setColor(Color.RED);
```

21. What is the data type of `Location.SOUTHWEST`? `int`

22. What is the 0 direction for the `Location` class? `NORTH`

23. What is the 450 direction for the `Location` class? `EAST`

24. What is the value of `Location.NORTHWEST`? 315

25. What is the numerical value of `Location.EAST + Location.SOUTH`? 270

26. How many constants based on the turn angle are provided in the `Location` class? Seven

27. What is the data type of `Location.LEFT`? `int`

28. What is the constant for a 0 degree turn angle for the `Location` class? `AHEAD`

29. What is the value of `Location.HALF_LEFT`? -45

30. What is the value of `Location.FULL_CIRCLE + Location.HALF_LEFT`? 315

31. Write a segment of code to get an actor to turn around.
`setDirection(getDirection() + Location.HALF_CIRCLE);`

32. What two parameters are required of the `Location` constructor? row, column

33. Create a reference `Location` named `loc1`. Instantiate the `Location` object with a row number of 3 and a column number of 2.

```
Location loc1 = new Location(3, 2);
```

34. How many new accessor methods exist for the `Location` class? Two

35. Write the code to output the row for `loc1`. `System.out.println("loc1 is at row: " + loc1.getRow());`

36. Write the code to output the column for `loc1`. `System.out.println("loc1 is at col: " + loc1.getCol());`

37. List the `Location` methods that give information about the relationship between locations and directions. `getAdjacentLocation`, `getDirectionToward`

38. What direction is an actor traveling if rows are increasing and columns are decreasing? Southwest

39. What is compared in the `equals` method of `Location`? row, and column

40. What is compared in the `compareTo` method of `Location`? row, then column

41. Write a segment of code to output the name of `Bug bugBoy` or `Bug bugGirl`, with the lowest `Location`.

```
If(bugBoy.getLocation().compareTo(bugGirl.getLocation())< 0)
     System.out.println("bugBoy comes first");
else
     System.out.println("bugGirl comes first");
```

42. When will `loc1.compareTo(loc2)` return a positive number? Assume: `loc1 = new Location(row1,col1)` and `loc2= new Location(row2,col2)` (Describe every situation.) If `row1` is greater than `row2` or if `row1` equals `row2` and `col1` is greater than `col2`

43. Do questions **Do You Know?** Set 3.

Appendix I

GridWorld Case Study Part 3 — Grid GridWorld Classes and Interfaces Reading Worksheet

The `Grid` Interfaces

1. Read pages 20–21 of the Student Manual.
2. The `Grid` in `BugRunner.java` contains what type of objects?
3. Why can you not construct a `Grid` object?
4. Declare a `Grid<Actor>` reference named `grd`.
5. Assign `grd` a new `BoundedGrid` of `Actor` objects that is 25 x 25.
6. What method checks to see if a `Location` is contained in the `Grid`?
7. Write a segment of code to output if a location (25, 25) is contained in `grd`. What is the output of executing this code?
8. What must not be passed in as an argument in all case study methods?
9. Write a segment of code to output if a `Location loc` is contained in `grd`.
10. List the three methods that add, remove and retrieve objects from a `Grid`.
11. Why do all three of these methods return `E`?
12. Declare `Actor act1` to reference the `Actor` at Location `loc1` in `grd`.
13. What type is returned by the `getOccupiedLocations` method?
14. Write a segment of code to output the total number of occupants in `grd`.
15. What is returned by `grd.getValidAdjacentLocations(new Location(0,0))`?
16. What is returned by `grd.getValidAdjacentLocations(new Location(25, 25))`?
17. What methods allow the user to determine the size of the `Grid`?
18. What is returned by these methods for an `UnboundedGrid`?

19. Create and initalize `numCells` to hold the number total number of cells in `Grid someGrid`.
20. Do questions **Do You Know?** Set 4.

GridWorld Case Study Part 3 — Grid GridWorld Classes and Interfaces Reading Worksheet (Solutions)

The `Grid` Interfaces

1. Read pages 20–21 of the Student Manual.
2. The `Grid` in `BugRunner.java` contains what type of objects? `Actor`
3. Why can you not construct a `Grid` object? `Grid` is an interface.
4. Declare a `Grid<Actor>` reference named `grd`. `Grid<Actor> grd;`
5. Assign `grd` a new `BoundedGrid` of `Actor` objects that is 25 x 25.
   ```
   grd = new BoundedGrid<Actor>(25, 25);
   ```
6. What method checks to see if a `Location` is contained in the `Grid`? `isValid`
7. Write a segment of code to output if the location (25, 25) is contained in `grd`. What is the output of executing this code?
   ```
   if (grd.isValid(new Location(25, 25)))
       System.out.println("(25,25) is a valid location in grd");
   else
       System.out.println("(25,25) is an invalid location
   in grd"); Output:

       (25, 25) is an invalid location in grd
   ```
8. What must not be passed in as an argument in all case study methods? `null`
9. Write a segment of code to output if a `Location loc` is contained in `grd`.
   ```
   if (grd.isValid(loc))
       System.out.println(loc + " is a valid location in grd");
   else
       System.out.println(loc + " is an invalid location in grd");
   ```
10. List the three methods that add, remove and retrieve objects from a `Grid`. `put, remove, get`
11. Why do all three of these methods return `E`? The `Grid` interface contains objects of `E` type.
12. Declare `Actor act1` to reference the `Actor` at `Location loc1` in `grd`.
    ```
    Actor act1 = grd.get(loc1);
    ```

13. What type is returned by the `getOccupiedLocations` method?
    `ArrayList<Location>`

14. Write a segment of code to output the total number of occupants in `grd`.
    ```
    ArrayList<Location> all = grd.getOccupiedLocations();
    System.out.println("The number of objects in the grid
    is: "
                                                + all.size());
    ```

15. What is returned by `grd.getValidAdjacentLocations(new Location(0,0))`?
    [(0, 1), (1, 0), (1, 1)]

16. What is returned by `grd.getValidAdjacentLocations(new Location(25, 25))`? [(24, 24)]

17. What methods allow the user to determine the size of the `Grid`? `getNumRows`, `getNumCols`

18. What is returned by these methods for an `UnboundedGrid`? –1

19. Create and initalize `numCells` to hold the number total number of cells in `Grid someGrid`.
    ```
    int numCells = someGrid.getNumRows() * someGrid.
    getNumCols();
    ```

20. Do questions **Do You Know?** Set 4.

Appendix J

GridWorld Case Study Part 3 — Actor GridWorld Classes and Interfaces Reading Worksheet

The `Actor` Class

1. Read pages 22–23 of the Student Manual.
2. How many assessor methods exist for the `Actor` class?
3. Declare a `Color` variable named `clr` to store the color of `Actor actr`.
4. Declare an `int` variable named `dir` to store the direction of `actr`.
5. Declare a `Grid<Actor>` variable named `grd` to store the grid of `actr`.
6. Declare a `Location` variable named `loc` to store the location of `actr`.
7. Declare a `Location` variable named `nextLoc` to store the adjacent location directly in front of `actr`. (You may use the variables from previous questions in the following questions.)
8. Declare an `ArrayList<Location>` variable named `emptyNbr` to store the empty neighboring locations of `actr`.
9. How does an `Actor` add itself to the grid?
10. Write a segment of code to add an `Actor` into all the empty neighbors of `actr`.
11. Why does the `removeSelfFromGrid` method have no parameters?
12. Write a segment of code to remove all `Actors` that are neighbors of `actr`.
13. Why not just `put` and `remove` actors directly from the grid?
14. Write code to check if `grd` contains an `Actor` at `Location mysteryLoc`. If `mysteryLoc` is empty, add an `Actor` there.
15. Write code to check if `grd` contains an `Actor` at `Location otherLoc`. If it does, remove it from the grid.
16. What happens when an actor calls the `moveTo` method using a location already occupied by another actor?
17. Write a statement to have `Actor act1` move to the location of `Actor act2`.

18. After the code for problem 15 is executed, what is the location of the original actor at `otherLoc`?
19. How many new modifier methods exist for the `Actor` class?
20. Write a statement to have `Actor act1` be the color `Color.BLUE`.
21. Write a statement to check if `Actor act1` and `Actor act2` are different colors. If they are, change `act2` to `act1`'s color.
22. Write a statement to check if `Actor act1` and `Actor act2` face different directions. If they do, have `act2` face the same direction as `act1`.
23. Write code to check if `grd` contains an `Actor` at `Location otherLoc`. If `otherLoc` is not empty, remove the `Actor` there.
24. Write code to change all the `Actors` in `actr`'s grid to be `actr`'s color.
25. How is the behavior of a class that extends `Actor` defined?
26. What is the behavior of the `Actor` class?
27. How can you extend the `Actor` class and keep the behavior of the Actor class?
28. What classes provide examples of overriding the `act` method?
29. Do questions **Do You Know?** Set 5.

GridWorld Case Study Part 3 — Actor GridWorld Classes and Interfaces Reading Worksheet (Solutions)

The `Actor` Class

1. Read pages 22–23 of the Student Manual.
2. How many assessor methods exist for the `Actor` class? Four
3. Declare a `Color` variable named `clr` to store the color of `Actor actr`.
```
Color clr = actr.getColor();
```
4. Declare an `int` variable named `dir` to store the direction of `actr`.
```
int dir = actr.getDirection();
```
5. Declare a `Grid<Actor>` variable named `grd` to store the grid of `actr`.
```
Grid<Actor> grd = actr.getGrid();
```
6. Declare a `Location` variable named `loc` to store the location of `actr`.
```
Location loc = actr.getLocation();
```
7. Declare a `Location` variable named `nextLoc` to store the adjacent location directly in front of `actr`. (You may use the variables from previous questions in the following questions.)
```
Location nextLoc = loc.getAdjacentLocation(dir);
```
8. Declare an `ArrayList<Location>` variable named `emptyNbr` to store the empty neighboring locations of `actr`.
```
ArrayList<Location> emptyNbr =
    grd.getEmptyAdjacentLocations(loc);
```
9. How does an `Actor` add itself to the grid? `putSelfInGrid`
10. Write a segment of code to add an `Actor` into all the empty neighbors of `actr`.
```
for (Location emptyLoc : emptyNbr)
    new Actor().putSelfInGrid(grd, emptyLoc);
```
11. Why does the `removeSelfFromGrid` method have no parameters? The actor knows its location and grid, so it does not need any other information.

12. Write a segment of code to remove all Actors that are neighbors of actr.

```
ArrayList<Location> occNbr =
    grd.getOccupiedAdjacentLocations(loc);
for (Location neighborLoc: occNbr)
{
        Actor neighbor = grd.get(neighborLoc);
        neighbor.removeSelfFromGrid();
    }
```

13. Why not just put and remove actors directly from the grid? The put and remove methods do not update the Actor object to reflect the change in state of the object. This could leave an actor in an inconsistent state with the grid.

14. Write code to check if grd contains an Actor at Location mysteryLoc. If mysteryLoc is empty, add an Actor there.

```
if (grd.isValid(mysteryLoc) && grd.get(mysteryLoc)==
null)
new Actor().putSelfInGrid(grd, mysteryLoc);
```

15. Write a segment of code to check if grd contains an Actor at Location otherLoc. If it does, remove it from the grid.

```
Actor someActor = grd.get(otherLoc);
If (someActor != null)
someActor.removeSelfFromGrid();
```

16. What happens when an actor calls the moveTo method using a location already occupied by another actor? The other actor removes itself from the grid and the current actor takes its location.

17. Write a statement to have Actor act1 move to the location of Actor act2.

```
act1.moveTo(act2.getLocation());
```

18. After the code for problem 15 is executed, what is the location of the original actor at otherLoc? null

19. How many new modifier methods exist for the Actor class? Two (setColor, setDirection)

20. Write a statement to have Actor act1 be the color Color.BLUE.

```
act1.setColor(Color.BLUE);
```

21. Write a statement to check if Actor act1 and Actor act2 are different colors. If they are, change act2 to act1's color.

```
if (! act1.getColor().equals(act2.getColor()))
act2.setColor(act1.getColor());
```

22. Write a statement to check if `Actor act1` and `Actor act2` face different directions. If they do, have `act2` face the same direction as `act1`.

```
if (act1.getDirection() != act2.getDirection())
act2.setDirection(act1.getDirection());
```

23. Write a segment of code to check if `grd` contains an `Actor` at `Location otherLoc`. If `otherLoc` is not empty, remove the `Actor` there.

```
Actor someActor = grd.get(otherLoc);
if( someActor != null)
someActor.removeSelfFromGrid();
```

24. Write code to change all the `Actors` in `actr`'s grid to be `actr`'s color.

```
Grid<Actor> grd = actr.getGrid();
Color clr = actr.getColor();
ArrayList<Location>allActorLoc=grd.
getOccupiedLocations();
for (Location occupied: allActorLoc)
{
        Actor temp = grd.get(occupied);
        temp.setColor(clr);
}
```

25. How is the behavior of a class that extends `Actor` defined? By overriding `act`

26. What is the behavior of the `Actor` class? It reverses the direction of the actor.

27. How can you extend the `Actor` class and keep the behavior of the `Actor` class? Do not override the `act` method.

28. What classes provide examples of overriding the `act` method? `Bug`, `Flower` and `Rock`

29. Do questions **Do You Know?** Set 5.

Appendix K

GridWorld Case Study Part 3 — Extending Actor GridWorld Classes and Interfaces Reading Worksheet

Extending the Actor Class

1. Read pages 24–25 of the Student Manual.
2. How many classes are given as subclasses of the `Actor` class?
3. How are the behaviors of the subclasses specified to be different than `Actor`?
4. Describe the behavior of the `Actor` class.
5. Describe the behavior of the `Rock` class.
6. Describe the behavior of the `Flower` class.
7. What does a bug do when the location in front is occupied by a rock?
8. What does a bug do when the location in front is occupied by a flower?
9. List the auxiliary methods of the `Bug` class.
10. Write two Boolean expressions (one resulting in *true* and one resulting in *false*) demonstrating the use of the `instanceof` operator and using the `Rock` reference `rck` (you may not use the word `Rock` for this solution).
11. Is `instanceof Actor` always true for any occupant in a `Grid<E>`?
12. Write a segment of code to remove all the `Rocks` in `Grid<Actor> grd`.
13. If the adjacent location in front of `Bug bg` is a `Flower`, change the flower's color to `bg`'s color.
14. If the adjacent location to a `Bug` is `null`, what does that mean about that grid location?
15. Why does the `canMove` method need to check if the neighbor is `null` first?
16. Why does the `canMove` method need to check if a bug is in the grid?
17. What does the `turn` method for bug do?
18. When is the `turn` method called by a bug method?

19. Do questions **Do You Know?** Set 6.
20. Read **What Makes It Run?** (page 27)

GridWorld Case Study Part 3 — Extending Actor
GridWorld Classes and Interfaces
Reading Worksheet (Solutions)

Extending the `Actor` Class

1. Read pages 24–25 of the Student Manual.
2. How many classes are given as subclasses of the `Actor` class? Three
3. How are the behaviors of the subclasses specified to be different than `Actor`? By overriding the `act` method
4. Describe the behavior of the `Actor` class. Flips (turns) back and forth
5. Describe the behavior of the `Rock` class. Does nothing
6. Describe the behavior of the `Flower` class. Darkens the color of the flower, without moving
7. What does a bug do when the location in front is occupied by a rock? Turns right 45 degrees
8. What does a bug do when the location in front is occupied by a flower?
   The flower removes itself from the grid and the bug moves into the location in front.
9. List the auxiliary methods of the `Bug` class. `canMove, move, turn`
10. Write two Boolean expressions (one resulting in *true* and one resulting in *false*) demonstrating the use of the `instanceof` operator and using the `Rock` reference `rck` (you may not use the word `Rock` for this solution).

```
rck instanceof Actor                (true)
rck instanceof Location             (false)
```

11. Is `instanceof Actor` always true for any occupant in a `Grid<E>`? No, because a `Grid` can contain objects that are not `Actors`.
12. Write a segment of code to remove all the `Rocks` in `Grid<Actor> grd`.

```
ArrayList<Location>allActorLoc=grd.getOccupiedLocations();
for (Location occupied : allActorLoc)
{
        Actor temp = grd.get(occupied);
        if (temp instanceOf Rock)
          temp.removeSelfFromGrid();
}
```

13. If the adjacent location in front of `Bug bg` is a `Flower`, change the flower's color to `bg`'s color.

```
Grid<Actor> gr = bg.getGrid();
Location loc = bg.getLocation();
Location neighLoc =loc.getAdjacentLocation(bg.getDirection());
if (gr.isValid(neighLoc))
{
        Actor actr = gr.get(neighLoc);
        if (actr != null && actr instanceof Flower)
              actr.setColor(bg.getColor());
}
```

14. If the adjacent location to a `Bug` is `null`, what does that mean about that grid location? It is empty or invalid.

15. Why does the `canMove` method need to check if the neighbor is `null` first? Short circuit evaluation

16. Why does the `canMove` method need to check if a bug is in the grid? It is possible that an actor has been removed from the grid by another actor.

17. What does the `turn` method for bug do? Turns 45 degrees to the right

18. When is the `turn` method called by a bug method? When the location in front is blocked

19. Do questions **Do You Know?** Set 6.

20. Read **What Makes It Run**? (page 27)

Unit Plan: Part 4 — Critters

Mike Lew

Introduction

Students are always curious about the application of what they learn in class to the "real world." In this project, students will model an ecosystem, a task that has been and is still undertaken by many biologists and environmental scientists the world over. They will use the GridWorld simulation to model an Amazon rainforest, although any ecosystem can be simulated (desert, tundra, arctic, etc.).

Students will simulate one of three different types of symbiotic relationships and a predator-prey relationship. The "hook" to this project is that students are able to design their own creatures to place into the ecosystem along with creatures designed by other students. The resulting interaction is one that students (and I) enjoy seeing and analyzing. For this project, I assume that students are familiar with the basic operation of GridWorld and have studied inheritance and polymorphism in addition to having a working knowledge of class structure (data members and methods).

The project consists of four parts. In the first part, students will design and write class definitions and implementations for two creatures that are involved in one of three types of symbiotic relationships: mutualistic (both creatures benefit from each other), commensalistic (one creature benefits, the other is unaffected) and parasitic (one creature is harmed or weakened but not directly killed off). At least one of these creatures must inherit from `Critter` or a subclass or `Critter`. Students may choose which symbiotic relationship to model.

The second part requires students to design and write class definitions and implementations for two creatures that exist in a predator-prey relationship (one creature "eats" the other creature). Again, at least one of the two creatures must inherit from `Critter` or a subclass of `Critter`.

The third part of the project requires students to design and write a class definition and implementation for an "environmental feature" with which their creatures can interact (trees, rivers, streams, holes, night/day, iceberg, cactus, etc.). Most of these features will inherit from `Actor` instead of `Critter`, and students will define how their creatures interact with the environmental feature. This part is geared toward students who are able to implement such things as trees (tunnels for creatures to burrow into), queues (logs for creatures to walk through) and stacks (places for creatures to store food). However, this part can also accommodate students who can use `ArrayLists` to design simple features such as nests. Again, any of these environmental features will inherit from `Actor` or a subclass of `Actor` and include one or more of these data structures (tree, set, stack, queue, etc.) in their private data member list.

Finally, in the fourth part, after all creatures and "environmental features" have been written, the students will then integrate these objects into one working ecosystem, where they will be able to see how their own creatures interact with other creatures and "environmental features." Here students can see that modifying the methods called in the `act` method, but not the `act` method itself (e.g., by NOT removing or renaming the methods called in the `act` method), is a powerful mechanism to create a related "family" of creatures who behave or "act" in the same basic manner. In addition, students will also see that overriding the `act` method of an `Actor` may be desirable and/or necessary thing to do. In any case, all objects can and will be processed polymorphically because they all possess an `act` method.

I have done a similar type of project with the Marine Biology Simulation. My students were very excited to complete a project in which they were involved with the design decisions of the "fish" and the "environmental features" of the "ocean." In addition to being the highlight of the year for my students, the final integration of all objects into one working program illustrated the value of designing classes that could be used without modification in a large-scale project and the issues that arise when certain design decisions are made.

Learning Goals

- To understand that there are five methods which control the "acting" of a `Critter` (`getActors`, `processActors`, `getMoveLocations`, `selectMoveLocation` and `makeMove`).
- To understand that any or all of these five methods can be overridden to alter creature behavior.
- To write a class definition that inherits from `Critter` and overrides one or more of the five methods called in the `act` method.
- To understand why the `act` method should not be overridden.
- To understand the importance of clearly stated design specifications.
- To be able to integrate written class implementations into a larger project.

Instructional Considerations

This lesson assumes that the GridWorld simulation has been introduced and that students are familiar with class definitions, inheritance and polymorphism. If students have not been introduced to class design, this project may serve as their first experience with designing a class and writing a class definition based on their own specifications. If this is the case, it may be fruitful for each student to give a 4- to 5-minute oral presentation of

their ideas for the specifications of their class design. Allowing students to verbalize their thoughts requires them to think through their thought processes (metacognition) and justify their design decisions to themselves. Students may, and probably will, make design changes based on feedback from the teacher and/or their peers. Allowing each student to present their creature designs (and corresponding class names) will allow for other students/groups to know what other creatures and features will be in the ecosystem in Part 4.

Teaching Timeline (Each day described below is a 55-minute class period.)

Day 1: Introduce the project. Separate large classes into small groups (more discussion on this below). Goal for students: (1) define class names for their creatures, (2) define creature behavior, (3) define new instance variables for their creatures, and (4) define which methods in act will be overridden for their creatures.

Day 2: Students present ideas for creatures in Part I — Symbiotic relationships. Share class names and creature behavior with other students/groups so that all know what creatures will eventually exist in the "ecosystem."

Day 3: Lab time — Write complete class definitions for the creatures presented on Days 1–2 and create images for their new creatures. Test class implementation in a test program. During this time, the teacher should move among individuals or groups to check on student progress and address any problems.

Day 4: Students present ideas for creatures for Part II — Predator-prey relationships. Share class names and creature behaviors as in Day 2.

Day 5: Lab time — Write complete class definitions for creatures presented on Day 4, and create images for their new creatures. Test class implementation in a test program. The teacher should again move about the class to check on student progress.

Day 6: Students present ideas for Part III — Environmental features. Share class names and environmental feature behavior as in Day 2.

Day 7: Lab time — Write complete class definitions for environmental feature described on Day 6 and create images for their new feature. Test class implementation in a test program.

Day 8: Integration of some or all of creatures and environmental features into one working program.

Instructional Delivery Ideas

Three different ways this project can be introduced to address class size, student abilities (differentiation) and learning styles are provided below.

Class Size

With class sizes that are relatively small (less than 16 students), this project could be individually assigned to each student to be completed independently. With class sizes such as these, individual students are responsible for their own assignments, but I assign "buddy" partners with whom they can discuss problems they encounter. Most of the simple (coding) problems are usually resolved by the students helping one another via their preassigned partners. In addition, the process of students assisting each other is a valuable learning experience for the "helper," because he or she gains experience in debugging code other than his or her own. If there is a problem that cannot be solved by the students, I assist in the debugging process. However, when students come to me for help, I also require them to describe the problem in detail. It is interesting to hear students solve their own problem through the process of verbalizing it!

A strategy that could be used with larger class sizes is to break the class into pairs. In my experience, the dynamics of two students works the best. A third student can sometimes be good in the case of debates about design decisions, but I have found that the programming work is too diluted with three or more students. Grouping helps classroom management and allows the teacher more time to meet with the students. In a class of 30, it is much easier to meet and discuss issues with 15 pairs than with 30 individual students.

Student Abilities

Each year I have students with very diverse experience and aptitudes. I always have many students who have never programmed before, and I usually have one or two students who have been programming for three or more years. In addition, I also have a spread of aptitudes. Each year, approximately a third of the class catches on very quickly and needs very little assistance. Another third of the class needs a key example or two to catch on, and the last third usually needs individualized help to one degree or another.

This project can be conducted with these different aptitude levels, as well. After introducing the project and talking about the possibilities for different creatures, I will allow the students to begin to work on the design of their creature. The students with a high aptitude for programming will quickly begin to work and will enjoy the challenge of designing a class of their own. If one or more of these students finishes early, they can then begin work on the next part of the project. This prevents boredom and the question, "What do I do next?" The "middle" group of students will also begin work and usually do not have problems until they actually get into the coding of their classes. When problems arise during the coding phase, I will offer suggestions to help guide them through their problems and toward possible solutions. I believe that greater understanding (and satisfaction) arises when they "figure it out for themselves" using this guided inquiry-based instructional technique. During this work time for

the middle group of students, I will give more detailed instruction to the last group of students who need help. I can have a small group discussion addressing their specific questions while the others are at work. If needed, I will give ideas for creatures and walk them through the design and coding process. I gauge their understanding to ensure that I do not give them too much information, but instead constantly put them in a position to think and proceed successfully.

I have found that allowing students to design their own objects, whether they are fish, aliens or creatures, gives them a sense of ownership of their project. I only assign the specifications for what needs to be done and how it will be assessed, but they have the freedom to design something of their own. I have found this to be true for all types of students: The ownership component of their project goes a long way in terms of their intrinsic motivation level.

Learning Styles

Throughout the course, my delivery of the material includes auditory, kinesthetic and visual components; this project is no different. When students present their ideas as outlined in the "Teaching Timeline" above, I require them to speak, draw and act out (mini role play) how their creatures will behave. Specifically, students describe verbally how their creature will behave. They are given a 2-minute time limit to clearly and succinctly describe their creatures. This 2-minute limit forces the students to "get to the point" and remove any extraneous detail from their description. During their verbal description, they are required to draw diagrams of their creature in a grid and show how their creature will move and/or process other creatures in different situations. Diagrams help the other students (and the teacher!) to understand the behavior of their creature. Finally, as a mini "role play," the students are required to perform an enactment of how their creatures will behave using their peers as "creatures" in the grid. (I have the students create a "grid" by moving their desks into a two-dimensional array arrangement.) In using this three-pronged approach to teaching the material, I can anecdotally say that many more students gain a deeper understanding of the material/problem at hand than when I did not require all three learning modalities. For students who do not master these concepts as quickly, I often find that addressing these three areas a second time and at a slower pace helps these students grasp and internalize the material.

Student Assignments

Below are the formal assignments with design requirements for each of the four parts of the project. These may be altered in any way to fit the dynamics of your class.

Description for Teacher: Part I — Symbiotic Relationships

Each student (or group) writes class definitions for two "critter-like" creatures. At least one of the creatures must be derived from `Critter`. These new creatures will have a symbiotic relationship (students will have to do some research on the animals they wish to model). The emphasis of this part of the project is to reinforce the idea that the `act` method calls five other methods (`getActors`, `processActors`, `getMoveLocations`, `selectMoveLocation` and `makeMove`) and that it is these methods that should be overridden to create new behaviors. The `act` method itself should not be overridden and if `act` needs to be overridden, then what the student is attempting to create should not be derived from `Critter`.

Part I — Symbiotic Relationships

Student Directions

You have been hired to write a program that simulates the *symbiotic* (mutualistic, commensalistic or parasitic) relationship between two kinds of creatures in an ecosystem you choose. A mutualistic relationship is one where both creatures benefit from each other's existence; a commensalistic relationship is one where one creature benefits from the relationship, but the other is unaffected; and a parasitic relationship is one where one creature benefits from the relationship and the other is harmed but not necessarily killed. You will write the class definitions for your creatures and incorporate them into a working program that tests the functionality of the creatures.

Part I Requirements

1. At least one of the two creatures must inherit from `Critter` or a subclass of `Critter`.
2. The two creatures must exhibit one of the three types of symbiotic behavior.
3. The creature(s) that inherits from `Critter` must NOT override the `act` method.
4. The creature(s) that inherits from `Critter` should override one or more of the "Big 5" methods called in the `act` method: `getActors`, `processActors`, `getMoveLocations`, `selectMoveLocation` and `makeMove`.
5. All of the overridden "Big 5" methods must satisfy that method's stated postconditions.
6. Your new creature may add up to two new instance variables and two new methods.

7. The two creatures should only interact with and/or kill off one creature outside of its symbiotic relationship.
8. The new creatures (Java files and images) should be integrated into one working GridWorld program.

Part I Formative Assessments/Grading Rubrics

With a creative design project such as this, I have found it very useful to have a grading criterion (rubric) in place that illustrates how the project will be evaluated. In short, my grading rubric is a one-for-one match with the project requirements. I show the students at the beginning of the project what the grading criteria are. This helps the students understand how their work will be evaluated. Teachers can determine the point value for each criterion to fit their grading scales.

Part I Project Criteria

1. Does at least one creature inherit from `Critter` or a subclass of `Critter`?
2. Do the two creatures exhibit some form of symbiotic behavior?
3. Does the creature(s) that inherits from `Critter` NOT override the `act` method?
4. Does the creature(s) that inherits from `Critter` override one or more of the "Big 5" methods called in the `act` method: `getActors`, `processActors`, `getMoveLocations`, `selectMoveLocation` and `makeMove`?
5. Do all of the overridden "Big 5" methods satisfy that method's stated postconditions?
6. Does the creature(s) that inherits from `Critter` add at most only two new instance variables and at most only two new methods?
7. Do the two creatures interact with and/or kill off only one creature outside of its symbiotic relationship?
8. Are the new creatures (Java files and images) correctly integrated into a working GridWorld program?

Description for Teacher: Part II — Predator-Prey Relationships

Each student (or group) writes a detailed description for two creatures, at least one of which must be derived from the original `Critter` class. These new creatures will have a predator-prey relationship. However, instead of writing the class definitions and implementations themselves, the students (or groups) will exchange the program specifications with other students/groups and attempt to write the class definitions for the program specifications they were given. This will require the students to (1) write program descriptions clearly for others to read and implement, and (2) understand the conceptual framework for the `Critter` class and GridWorld as a whole in order to write a coherent program description.

Part II — Predator-Prey Relationships

Student Directions

You have been hired to write a program that simulates the *predator-prey* relationship between creatures in an ecosystem. You will write the class definitions for the creature descriptions that you were given and incorporate them into a working program that tests the functionality of the creatures.

Part II Requirements

1. At least one of the two creatures must inherit from `Critter` or a subclass of `Critter`.
2. The two creatures must exhibit a predator-prey relationship (i.e., one creature must "eat" the other).
3. The creature(s) that inherits from `Critter` must NOT override the `act` method.
4. The creature(s) that inherits from `Critter` should override one or more of the "Big 5" methods called in the `act` method: `getActors`, `processActors`, `getMoveLocations`, `selectMoveLocation` and `makeMove`.
5. All of the overridden "Big 5" methods must satisfy that method's stated postconditions.
6. Each creature that inherits from `Critter` adds at most only two new instance variables and at most two new methods.
7. The two creatures only interact with and/or kill off one creature outside of its predator-prey relationship.
8. The new creatures (Java files and images) are integrated into one working GridWorld program.

Part II Formative Assessments/Grading Rubrics

1. Does at least one creature inherit from `Critter` or a subclass of `Critter`?
2. Do the two creatures exhibit a predator-prey relationship?
3. Does the creature(s) that inherits from `Critter` NOT override the "`act`" method?
4. Does the creature(s) that inherits from `Critter` override one or more of the "Big 5" methods?
5. Do all of the overridden "Big 5" methods satisfy that method's stated postconditions?
6. Does the creature(s) that inherits from `Critter` add at most only two new instance variables and at most two new methods?
7. Do the two creatures interact with and/or kill off only one creature outside of its predator-prey relationship?
8. Are the new creatures (Java files and images) correctly integrated into a working GridWorld program?

Description for Teacher: Part III — Environmental Features

Although outside the scope of creating a creature that inherits from `Critter`, the "Environmental Features" component of this project allows for creatures implemented in Parts I and II to interact with other objects.

Each student/group will write a class description for an environmental change and/or an environmental feature with which the `Critter` will interact. Examples include streams or mountains, logs that creatures walk through (queues), creature hills (stacks), underground creature dwellings (trees), eggs laid in nests (`ArrayLists`), nocturnal behavior, etc.

Part III — Environmental Features

Student Directions

The company that has hired you wants to add an environmental feature found in the ecosystem to be added to the simulation. This feature should interact with at least one creature in the simulation. You will write the class definition for the feature and incorporate it into a working program that tests its functionality.

Part III Requirements

1. The environmental feature should inherit from `Actor` or a subclass of `Actor`.
2. At least one of the creatures written in Parts I and II should interact with the environmental feature. This will require modification of the creature to recognize the environmental feature.
3. An image for the environmental feature should be created.
4. The environmental feature should use one or more of the following containers: `ArrayList` (AP Computer Science A only), stack, queue, tree, set or linked list (for those students who have studied these more advanced data structures).

Part III Formative Assessments / Grading Rubrics

1. Does the environmental feature inherit from `Actor` or a subclass of `Actor`?
2. Do one or more of the creatures written in Parts I and II interact with the environmental feature?
3. Does the new environmental feature have its own image?
4. Does the environmental feature use one or more of the following containers: `ArrayList` (AP Computer Science A only), stack, queue, tree, set, or linked list (for those students who have studied these more advanced data structures)?

Description for Teacher: Part IV — The Complete Ecosystem

The final part of the project is to have the entire class incorporate all of the creatures and environmental features into one working program (ecosystem). If all of the creatures have kept their respective `act` methods intact, this should be a relatively short and easy component of the project. I have done this with the Marine Biology Simulation and the students were excited to see what happened with their creations!!

Part IV — The Complete Ecosystem

Student Directions

In this final part of the project, you will integrate all of the creatures designed by you and your classmates into one working "ecosystem." You will compile and run the project to see the interaction of all the objects in this ecosystem. If all goes well, you will see some very interesting behavior!

Part IV Requirements

You should give copies of your creatures and environmental feature Java files and images to each class member and/or group in your class.

Your creatures and environmental features (Java files and images) should be directly usable in a standard GridWorld project (i.e., each new class should have an `act` method that can be called polymorphically).

Part IV Formative Assessments/Grading Rubrics

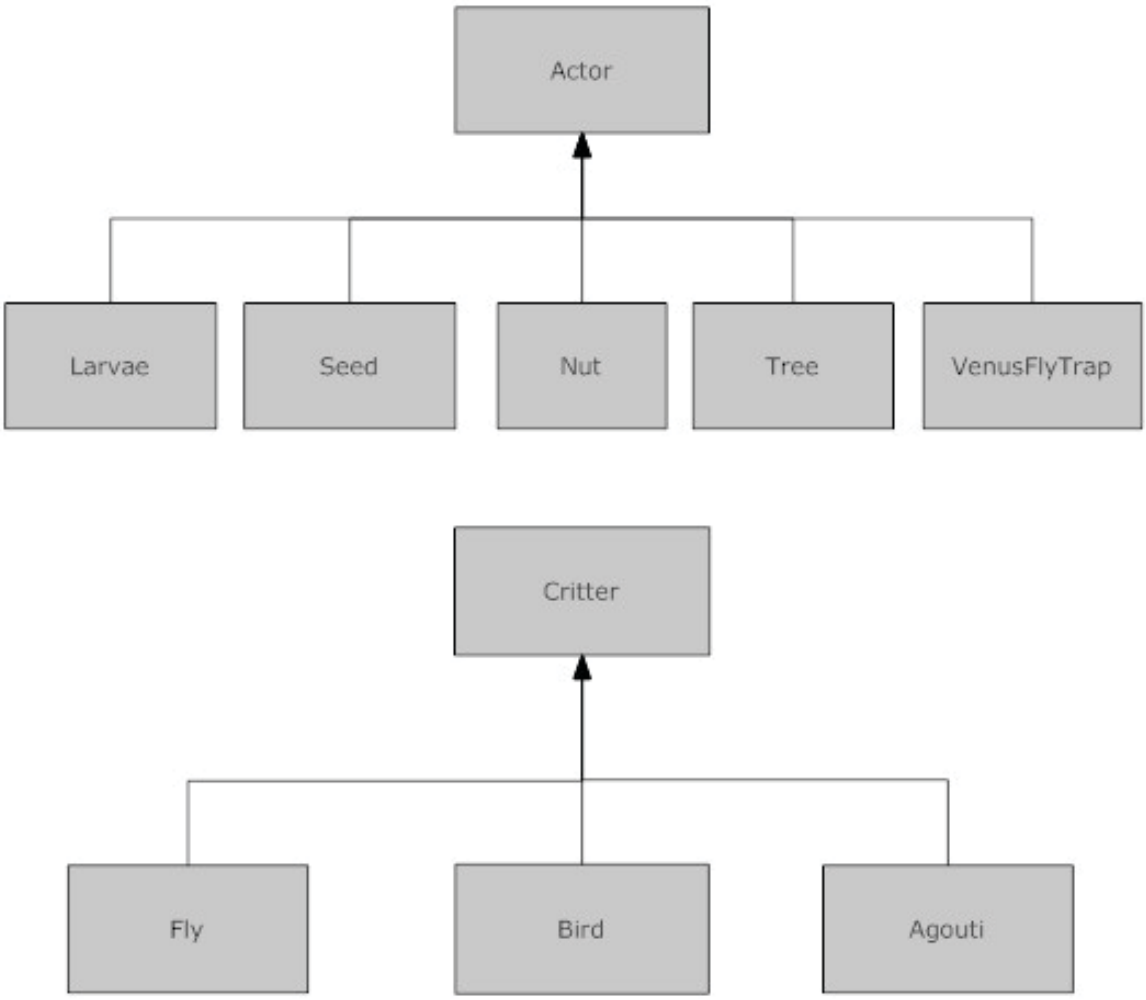
Student give copies of their Java files and images for creature(s) and environmental features to all other class members/groups.

The creature(s) and environmental feature are directly usable in a standard GridWorld project.

Sample Project

In this sample project, I created a simple ecosystem in the Amazon rainforest with a symbiotic relationship, a predator-prey relationship and an environmental feature. The project consists of a Bird, Agouti (a nut-eating rodent), Nut, Seed, Fly, Larvae, Venus Flytrap and Tree. An inheritance diagram is shown below.

Inheritance Hierarchy of Amazon Project



Symbiotic Relationship

In my model, the agouti rodent and the bird exist in a commensalistic relationship. Here the agouti feeds on nuts, leaving the seeds behind. The bird, which would otherwise be unable to eat the seeds, benefits from the agouti breaking the nut apart, while the agouti does not benefit from the bird. The `Agouti` and `Bird` classes both inherit from `Critter`, while the `Nut` and `Seed` classes inherit from `Actor`.

Predator-Prey Relationships

These types of relationships are easy to think about and implement. Here the predator is the `VenusFlytrap` and the fly is the prey. Flies can fly around as a `Critter` moves, as mentioned above. However, if a fly moves into a location in front of a `VenusFlytrap`, it will be eaten by the `VenusFlytrap`. The `VenusFlytrap` will then turn to the right or left 45 degrees and wait for the next fly. If it does not eat within five steps, then it will turn 45 degrees again.

However, to keep the fly population alive and well, flies will leave larvae behind after each move. The larvae will turn into a fly after a period of 10 iterations of its `act` method. `VenusFlytrap` and `Fly` both inherit from `Critter`, while `Larvae` inherits from `Actor`.

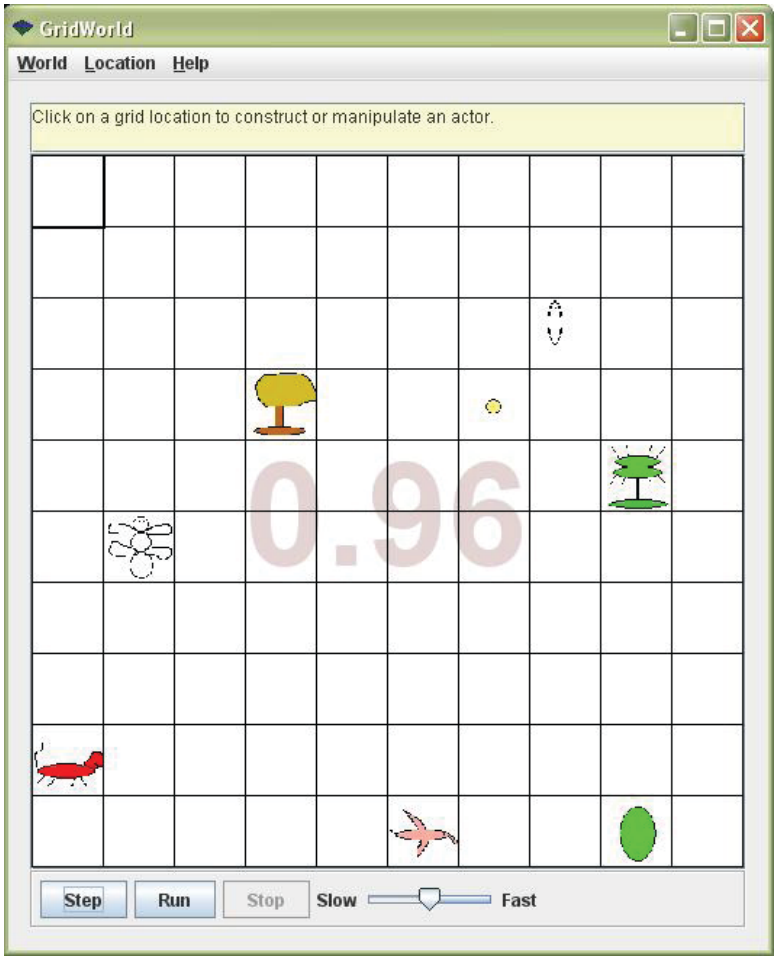
(Note that I created a third creature in this example [i.e., the Larvae creature], whereas the assignment calls for only two creatures. If students become enthusiastic about the project and ask to create more creatures than are required, I encourage them to proceed as long as the new creatures meet the requirements of their respective parts.)

Environmental Feature

In this project, a tree serving two purposes was implemented. First, it will drop nuts into adjacent locations for the agouti to eat. Second, it serves as a resting place for flies. If a fly flies into an adjacent location, the fly will "enter" the tree and be stored internally in the tree in an `ArrayList`. When four flies have entered the tree, they will, one by one, fly away from the tree.

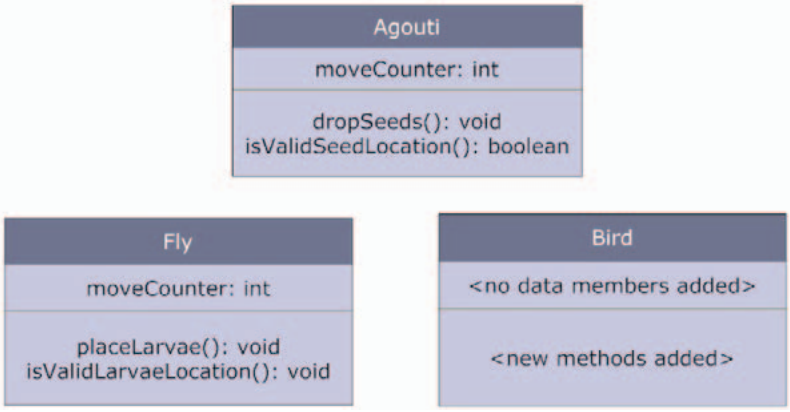
The following figure shows a complete Amazon Ecosystem, with each of the objects described above.

Complete Amazon Ecosystem



The following figure shows the new data members and methods added to the class derived from Critter.

Class Diagrams for Objects Derived from Critter



Select methods from the `Fly`, `Agouti` and `VenusFlytrap` classes are shown below.

`Fly placeLarvae` method

```
void placeLarvae(Location loc)

{
    if (moveCounter < 10)
    {
        moveCounter++;
    }
    else
    {
        moveCounter = 0;
        int currentDir = getDirection();
        Location currentLoc = getLocation();

        // find the direction BEHIND the fly
        // where the larvae will be left
        int behindDir = currentDir + Location. HALF_CIRCLE;

        // find the location behind the fly
        Location locBehind =
            currentLoc.getAdjacentLocation(behindDir);

        if (isValidLarvaeLocation(locBehind))
        {
            // place larvae behind fly
            Larvae newLarvae = new Larvae();
            newLarvae.putSelfInGrid(getGrid(),locBehind);
        }
    }

} // end method placeLarvae
```

Agouti dropSeeds method

```
public void dropSeeds()
{
    int currentDir = getDirection();
    Location currentLoc = getLocation();
    // finding the direction to the left and right
    // of the Aguouti
    int dirLeft = currentDir + Location.LEFT;
    int dirRight = currentDir + Location.RIGHT;

    // these are the locations left and right of the bird
    Location locLeft =
        currentLoc.getAdjacentLocation(dirLeft);
    Location locRight =
        currentLoc.getAdjacentLocation(dirRight);
    if( isValidSeedLocation(locLeft) )
    {
        Seed s = new Seed();
        s.putSelfInGrid(getGrid(), locLeft);
    }
    if( isValidSeedLocation(locRight) )
    {
        Seed s = new Seed();
        s.putSelfInGrid(getGrid(), locRight);
    }
} // end method dropSeeds
```

VenusFlytrap processActors method

```
public void processActors(ArrayList<Actor> creatures)

{
    int currentDirection = getDirection();
    Location inFront =
      getLocation().getAdjacentLocation(currentDirection);
    for (Actor thisCreature : creatures)
    {
      if (thisCreature.getLocation().equals(inFront)
          && thisCreature instanceof Fly)
      {
          thisCreature.removeSelfFromGrid();
          changeDirection();
          timeSinceEaten = 0;
      }

    } // end for

} // end processActors
```

Conclusion

This sample Amazon project can be used to demonstrate possibilities for each of the four parts of the project. After some discussion, students will undoubtedly think of even more relevant and sophisticated ecosystems to simulate, especially if they have taken or are taking AP Environmental Science. (Note that this project can be further modified to model systems consisting of "nonliving creatures," e.g., robots, traffic systems, air traffic controller systems, etc.) Allowing students to design their own creatures and integrating these creations into one project gives them ownership of the project and makes for a great "Grand Finale," both for the students and the teacher.

Sample Files for the Amazon Project can be downloaded from the GridWorld tag at http://www.thecubscientist.com/APCS/indexAPCS.html.

Worksheets on Part 4 of GridWorld, written by Joe Coglianese, are in the appendixes that follow.

Appendix A

GridWorld Case Study Part 4 — Critter Interacting Objects Reading Worksheet

The `Critter` Class

1. Read page 29 of the Student Manual.
2. What do all `Critters` share?
3. What does a `Critter` do first when it acts?
4. What are the next four things a `Critter` does?
5. How can different types of `Critters` move differently than `Critter`?
6. Which method identifies the behavior of a `Critter` ?
7. How many methods are invoked by the `act` method?
8. Which method(s) could be overridden in the subclasses of `Critter` to produce different behaviors?
9. Declare an `ArrayList<Actor>` variable named `actorList`.
10. Assign `actorList` to be the actors for `Critter someCritter` to process.
11. Write code to have `someCritter` process all the actors in `actorList` that are not a `Critter`. (Leave the contents of `actorList` unchanged.)
12. Declare an `ArrayList<Location>` variable named `locList`.
13. Assign `locList` to be the locations that `someCritter` could move into.
14. Write a segment of code to have `someCritter` move to a randomly selected location from the `locList`.
15. Which method should not be overridden in the subclasses of `Critter`?
16. What was the design intention of the `Critter` class?
17. When the `Critter class` is unsuitable for extending, what class should be extended?
18. Do questions **Do You Know?** Set 7.

GridWorld Case Study Part 4 — Critter Interacting Objects Reading Worksheet (Solutions)

The `Critter` Class

1. Read page 29 of the Student Manual.
2. What do all `Critters` share? A common pattern of behavior
3. What does a `Critter` do first when it acts? Gets a list of actors to process
4. What are the next four things a `Critter` does? Processes those actors, and then generates a set of possible move locations, selects one of them and moves to that location
5. How can different types of `Critters` move differently than `Critter`? They may get possible move locations differently, and they may select the actual move location differently.
6. Which method identifies the behavior of a `Critter`? `act method`
7. How many methods are invoked by the `act` method? Five
8. Which method(s) could be overridden in the subclasses of `Critter` to produce different behaviors? `getActors, processActors, getMoveLocations, selectMoveLocation, makeMove`
9. Declare an `ArrayList<Actor>` variable named `actorList`.
   ```
   ArrayList<Actor> actorList;
   ```
10. Assign `actorList` to be the actors for `Critter someCritter` to process.
    ```
    actorList = someCritter.getActors();
    ```
11. Write a segment of code to have `someCritter` store all the actors in `actorList` that are not `Critters` in a different `ArrayList`. (Leave the contents of `actorList` unchanged.)
    ```
    ArrayList<Actor> nonCritters = new ArrayList<Actor>();
    for (Actor actr : actorList)
        if (!(actr instanceof Critter))
            nonCritters.add(actr);
    ```
12. Declare an `ArrayList<Location>` variable named `locList`.
    ```
    ArrayList<Location> locList;
    ```

13. Assign `locList` to be the locations that `someCritter` could move into.
```
locList = someCritter.getMoveLocations();
```

14. Write a segment of code to have `someCritter` move to a randomly selected location from the `locList`.
```
int size = locList.size();
int ranNum = (int)(Math.random() * size);
someCritter.makeMove(locList.get(ranNum));
```

15. Which method should not be overridden in the subclasses of `Critter`? `act`

16. What was the design intention of the `Critter` class? `Critters` are actors that process other actors and then move.

17. When the `Critter` class is unsuitable for extending, what class should be extended? `Actor`

18. Do questions **Do You Know?** Set 7.

Appendix B

GridWorld Case Study Part 4 — Critter Behaviors Interacting Objects Reading Worksheet

Default `Critter` Behavior

1. Read pages 30–31 of the Student Manual.
2. What does a critter do before moving?
3. What two steps are involved in processing other actors?
4. How does the `processActors` method in the `Critter` class know which actors to process?
5. Why would a subclass of `Crittter` override the `getActors` method?
6. Write code to have a `Critter` variable named `someCritter` process all the actors in the grid.
7. What does the `processActors` method in the `Critter` remove?
8. What operator allows a `Critter` to tell if it is going to process another `Critter`?
9. If a critter did not eat `Actors`, what would it eat?
10. What is the three-step process for a critter to move to a new location?
11. Why are there three different methods implementing this single process?
12. What is returned by the `getMoveLocations` method for `Critter`?
13. If a critter were at (4, 3) facing East, what could be in the list returned by `getMoveLocations`?
14. How does a critter select which location to move to?
15. What does `getMoveLocations` return if a critter is unable to move?
16. What gets passed to the `makeMove` method by a critter?
17. What if `null` is passed in as an argument for the `makeMove` method?

GridWorld Case Study Part 4 — Critter Behaviors
Interacting Objects
Reading Worksheet (Solutions)

Default `Critter` Behavior

1. Read pages 30–31 of the Student Manual.
2. What does a critter do before moving? Process other actors in some way
3. What two steps are involved in processing other actors?
   a. Select which actors to process
   b. Process each selected actor
4. How does the `processActors` method in the `Critter` class know which actors to process? An `ArrayList<Actor>` containing all neighboring actors is passed in as an argument.
5. Why would a subclass of `Critter` override the `getActors` method? To choose a different set of actors that it will process
6. Write a segment of code to have a `Critter` variable named `someCritter` process all the actors in the grid.

```
Grid<Actor> grd = someCritter.getGrid();
ArrayList<Location> allOccpLoc = grd.getOccupiedLocations();
ArrayList<Actor> allActors = new ArrayList<Actor>();
for (Locations loc : allOccpLoc)
    allActors.add(grd.get(loc));
someCritter.processActors(allActors);
```

7. What does the `processActors` method in the `Critter` remove? All actors that are not rocks or critters
8. What operator allows a `Critter` to tell if it is going to process another `Critter`? `instanceof`
9. If a critter did not eat `Actors`, what would it eat? Nothing
10. What is the three-step process for a critter to move to a new location?
    a. Determine which locations are candidates for the move.
    b. Select one candidate
    c. Make the move
11. Why are there three different methods implementing this single process? Allows subclasses to change each behavior separately

12. What is returned by the `getMoveLocations` method for `Critter`? All the empty adjacent locations
13. If a critter were at (4, 3) facing East, what could be in the list returned by `getMoveLocations`? (3, 2), (3, 3), (3, 4), (4, 2), (4, 4), (5, 2), (5, 3) and (5, 4)
14. How does a `Critter` select which location to move to? Randomly
15. What does `getMoveLocations` return if a `Critter` is unable to move? Its current location
16. What gets passed to the `makeMove` method by a critter? The selected location
17. What if `null` is passed in as an argument for the `makeMove` method? The critter removes itself from the grid!

Appendix C

GridWorld Case Study Part 4 — ChameleonCritter Interacting Objects Reading Worksheet

Extending the `Critter` Class

`ChameleonCritter`

1. Read pages 32–33 of the Student Manual.
2. Which actors does `Critter` class send its list of actors for processing?
3. Which actors does `ChameleonCritter` class send its list of actors for processing?
4. What are the neighbors of the `ChameleonCritter` at (6, 3)?
5. How does `Critter` process actors?
6. How does `ChameleonCritter` process actors?
7. Which method(s) does the `ChameleonCritter` override?
8. What are the neighbors of the `ChameleonCritter` at (6, 3)?
9. If the `ChameleonCritter` at (4, 4) move to (3, 4), what direction would it be facing?

GridWorld Case Study Part 4 — ChameleonCritter Interacting Objects Reading Worksheet (Solutions)

Extending the `Critter` Class

`ChameleonCritter`

1. Read pages 32–33 of the Student Manual.
2. Which actors does `Critter` class send its list of actors for processing? All the neighboring actors (touching the critter)
3. Which actors does `ChameleonCritter` class send its list of actors for processing? The same as `Critter`, all the neighboring actors
4. What are the neighbors of the `ChameleonCritter` at (6, 3)? (5, 2), (5, 3), (5, 4), (6, 2), (6, 5), (7, 2), (7, 3) and (7, 5)
5. How does `Critter` process actors? Removes actors that are not rocks or critters
6. How does `ChameleonCritter` process actors? Randomly selects one and changes its own color to the color of the selected actor
7. Which method(s) does the `ChameleonCritter` override? `makeMove`, `processActors`
8. What are the neighbors of the `ChameleonCritter` at (6, 3)? (5, 2), (5, 3), (5, 4), (6, 2), (6, 5), (7, 2), (7, 3) and (7, 5)
9. If the `ChameleonCritter` at (4, 4) move to (3, 4), what direction would it be facing? `Location.NORTH`

Appendix D

GridWorld Case Study Part 4 — CrabCritter Interacting Objects Reading Worksheet

Another `Critter`

`CrabCritter`

1. Read page 34 of the Student Manual.
2. Which actors does `CrabCritter` class send its list of actors for processing?
3. Actors at which locations would be sent for processing by a `CrabCritter` at (6, 3) facing `Location.EAST`?
4. How does `CrabCritter` process actors?
5. Where is `CrabCritter` process actors defined?
6. If a `CrabCritter` were at (4, 3) facing `Location.EAST`, what locations would be returned by `getMoveLocations`?
7. What methods did the `CrabCritter` override?
8. From what location would a `CrabCritter` at (3, 4) facing `Location.NORTH` eat?
9. What will a `CrabCritter` not eat? Why?
10. How does a crab select which location to move to?
11. What does a crab do if it cannot move?

Suppose there is a new subclass of `CrabCritter` named `SkinnyCrabCritter`. It processes actors like `CrabCritter` but it only randomly selects one actor to eat. The `SkinnyCrabCritter` moves like `CrabCritter`, but if it has not eaten it dies.

12. Explain why `SkinnyCrabCritter` dying in `processActors` would violate the method's postconditions.
13. In what method could `SkinnyCrabCritter` die?
14. Propose a way to have `SkinnyCrabCritter` die based on not eating in any given turn.

15. Write the `processActors` method for `SkinnyCrabCritter`.
16. Complete the **Do You Know?** Set 9.

GridWorld Case Study Part 4 — CrabCritter Interacting Objects Reading Worksheet (Solutions)

Another `Critter`

`CrabCritter`

1. Read page 34 of the Student Manual.
2. Which actors does `CrabCritter` class send its list of actors for processing? Actors from the three cells in front of `CrabCritter`
3. Actors at which locations would be sent for processing by a `CrabCritter` at (6, 3) facing `Location.EAST`? (5, 4), (6, 4) and (7, 4)
4. How does `CrabCritter` process actors? Removes all actors from the three cells in front that are not rocks or critters
5. Where is `CrabCritter` process actors defined? `Critter`
6. If a `CrabCritter` were at (4, 3) facing `Location.EAST`, what locations would could be returned by `getMoveLocations`? [(4, 2), (4, 4)]
7. What methods did the `CrabCritter` override? `getActor`, `getMoveLocations` and `makeMove` .
8. From what location would a `CrabCritter` at (3, 4) facing `Location.NORTH` eat? (2, 3), (2, 4), (2, 5)
9. What will a `CrabCritter` not eat? Why? `Rock` or `Critter` because it was inherited from `Critter`
10. How does a crab select the location to move to? Randomly
11. What does a crab do if it can not move? Turns 90 degrees

Suppose there is a new subclass of `CrabCritter` named `SkinnyCrabCritter`. It processes actors like `CrabCritter` but it only randomly selects one actor to eat. The `SkinnyCrabCritter` moves like `CrabCritter`, but if it has not eaten it dies.

12. Explain why `SkinnyCrabCritter` dying in `processActors` would violate the method's postconditions. Postcondition (2) the location of the critter is unchanged.
13. In what method could `SkinnyCrabCritter` die? `makeMove`

14. Propose a way to have `SkinnyCrabCritter` die based on not eating in any given turn. Add a `Boolean` instance variable to store if it has eaten. Then check the value of the variable in the `makeMove` method and remove itself from the grid if it has not eaten.

15. Write the `processActors` method for `SkinnyCrabCritter`.

```
public void processActors(ArrayList<Actor> actors)
{
    int num = actors.size();
    if( num == 0)
    {
        hasEasten = false;
        return;
    }
    int ranNum = (int) (Math.random() * num);
    Actor other = actors.get(ranNum);
    other.removeSelfFromGrid();
}
```

16. Complete the **Do You Know?** Set 9.

Unit Plan: Part 5 — Grid

Leigh Ann Sudol

Introduction

This project is designed to help students learn about extending the `AbstractGrid` class as a part of the GridWorld case study (http://www.thecubscientist.com/APCS/indexAPCS.html). Included in this project are a number of supporting materials to help students become proficient at using the `Grid` class and also to become familiar with questions regarding different implementations of a grid. The program assignment, Airplane Scheduling, asks the students to extend the `AbstractGrid` class and then use the class in a context to answer questions about seating priority in an airplane. In addition to the program assignment, you will also find multiple-choice questions and a short-answer assessment question involving Big-Oh for different implementations of a grid.

Student Programming Assignment: Airplane Scheduling

Airlines face interesting problems every day. One common problem is to determine the best way to seat passengers to ensure that it takes the least possible time for them to board the airplane. Since there is only one door to the airplane, only one passenger can enter the plane at a time. The passenger then walks to the row that contains his or her assigned seat, stores his or her carry-on luggage, and then moves into his or her seat.

The question becomes whether it is better to board the passengers who have assigned seats in the rear of the plane first, board them in random order or board them in some combination of the two. This question is a perfect use for a computer simulation, because we can adjust variables and run a large number of tests without inconveniencing the consumers.

In order to create the simulation you will need to write the following classes:

`AirplaneTester:`	This class will contain a main method for testing your simulation.
`AirplaneWorld:`	This class will extend the `ActorWorld` class and handle the order in which the passengers board the plane.
`Passenger:`	This class will extend the `Actor` class and handle information that passengers need, such as what seat they will sit in during the flight.
`Airplane:`	This class will extend `AbstractWorld` and will be used to represent the seats in the airplane.

Passenger

The `Passenger` class should extend the `Actor` class and also provide functionality to store the location of the passenger's assigned seat on the airplane (separate from her or his current location). It should also provide functionality for a counter to be used to "pause" passengers as they are putting away their luggage upon reaching their rows, before leaving the aisle to sit in their assigned seats.

The `Passenger` constructor should take two parameters: a row and seat (column) value for the assigned seat of that passenger. It should assign values to the private variables, including a wait time of 10 time steps for stowing luggage.

The `Passenger` class should also have an implementation for an `act` method. The `Passenger` `act` method should deal with the following cases:

1. If the passenger is already in his or her seat, he or she should not do anything.
2. If the passenger is in the row where his or her seat is located and their luggage counter is 0, then he or she should move into their seat.
3. If the passenger is in the row where his or her seat is located and their luggage counter is greater than 0, he or she should continue storing their luggage (subtract one from the luggage counter).
4. If the passenger is not yet to his or her row and the aisle is not blocked in front of them, he or she should move one row closer to their assigned row.

`Passenger` class starter code

```
import info.gridworld.actor.Actor;
import info.gridworld.grid.Location;

public class Passenger extends Actor {
//declare instance variables here
public Passenger(int row, int seat){

}

public void act(){

}

}
```

Airplane

The `Airplane` class should extend the `AbstractGrid` class and provide data storage for the grid. The `Airplane`, in addition to maintaining the ability to store rows and columns of passengers, should also maintain a variable for the aisle row within the airplane. The aisle is a column without seats and will be the means by which passengers make their way to their seats.

The `Airplane` class needs a constructor that will take the number of rows, the number of seats across and the location of the aisle.
Other methods the `Airplane` class needs to implement will be:

1. `get(Location loc)`, which will retrieve a passenger stored at a particular row/seat location
2. `getNumCols()`, which will return the number of seats across, plus one for the aisle
3. `getNumRows()`, which will return the number of rows of seats in the airplane
4. `getOccupiedLocations()`, which will return an `ArrayList of Locations` for all of the seats that currently have passengers sitting in them, as well as all of the locations in the aisle where passengers are either loading baggage or waiting to continue on to their seats
5. `isValid(Location loc)`, which will return true if the given location is a valid seat or aisle location on the plane and false otherwise
6. `remove(Location loc)`, which will remove the passenger from the grid (plane)
7. `isAisleEmpty()`, which will return a Boolean value if the aisle of the airplane is empty (either the plane is completely empty or all passengers are seated)
8. `getCenterAisle()`, which will return the column number of the central aisle
9. `put(Location loc, Passenger obj)`, which will insert the passenger into the grid (Airplane) at the given location
10. `put(Location loc, Object obj)`, which is used only to satisfy the abstract class inheritance requirements

`Airplane` class starter code

```
import java.util.ArrayList;
import info.gridworld.actor.Actor;
import info.gridworld.grid.Grid;
import info.gridworld.grid.AbstractGrid;
import info.gridworld.grid.Location;

public class Airplane extends AbstractGrid{
//declare instance variables here
/**
 * Constructor for an Airplane
 * Creates a simulated airplane with seatsAcross-1 seats (leaving
 * an empty aisle in the middle for passengers to walk down)
 * @param numR the number of rows of seats in the airplane
 * @param seatsAcross the number of seats across plus the aisle
 * @param centerRow the location of the aisle in the airplane
 */
public Airplane(int numR, int seatsAcross, int centerRow){

}

/**
 * returns the passenger at the given location, null if there is
 * no one there
 */
public Passenger get(Location loc) {

}

/**
 * returns the number of columns in the grid - this is the
 * number of seats in any given row plus the aisle
 */
public int getNumCols() {

}

/**
 * returns the number of rows in the grid/Airplane
 */
public int getNumRows() {
```

```
}

/**
 * returns an ArrayList containing the occupied locations in the
 * grid
 */
public ArrayList<Location> getOccupiedLocations() {

}

/**
 * returns true if the given Location is valid, false otherwise
 */
public boolean isValid(Location loc) {

}

/**
 * removes the passenger at the given location from the grid
 */
public Passenger remove(Location loc) {

}

/**
 * Returns true if there are no passengers in the aisle
 * waiting to be seated
 */
public boolean isAisleEmpty(){

}

/**
 * returns the number corresponding to the column that serves
 * as the center aisle on the airplane
 */
public int getCenterAisle(){

}

/**
 * puts the passenger into the grid at the given location
 */
public Passenger put(Location loc, Passenger obj) {

}
```

```
/**
 * puts the indicated object into the grid.
 * This method exists only to satisfy inheritance requirements.
 * only the overloaded put method with a passenger
 * object should be called
 */
public Object put(Location loc, Object obj) {

}

}
```

AirplaneWorld

The `AirplaneWorld` class should extend the `ActorWorld` class. The `AirplaneWorld` class does not need any new data stored other than what `ActorWorld` provides for.

The `AirplaneWorld` constructor should take an `Airplane` parameter and call the `ActorWorld` constructor with that parameter.

The `Airplane` world has one additional method, `runSim()`, that is used to run the airplane simulation and watch how passengers seat themselves.

In a purely random seating arrangement, the `runSim` method will create a new `ArrayList` of `Passengers`, with each passenger having a targeted seat location within the aircraft. The easiest way to accomplish this is to write some loops to create a passenger for every seat; be careful not to seat anyone in the aisle.

Until the `ArrayList` is empty, if the "door" to the airplane is empty (location 0, aisle), randomly remove one passenger from the `ArrayList` and add him or her to the airplane. Don't forget to step and show the grid as you go. Also, make sure that all of your passengers get seated before you finish the simulation (i.e., make sure the aisle is empty).

AirplaneWorld class starter code

```
import info.gridworld.actor.ActorWorld;
import info.gridworld.grid.Location;
import java.util.ArrayList;

public class AirplaneWorld extends ActorWorld {
public AirplaneWorld(Airplane grid) {

    }

    public void runSim(){

    }

}
```

`AirplaneTester` class starter code

```
import info.gridworld.actor.ActorWorld;
import info.gridworld.grid.Location;

public class AirplaneTester {
public static void main(String args[]){
    Airplane toLoad = new Airplane(30, 7, 3);
    AirplaneWorld simulation = new AirplaneWorld(toLoad);
    simulation.runSim();
}

}
```

Notes to the Instructor for the Airplane Scheduler Assignment

Use and ordering of activities

It is assumed that students have already covered a number of data structures and Parts 1–4 of the GridWorld case study. If students have not addressed a number of data structures, the materials should be edited to cover only the data structures that have been covered in class. These materials can be revisited once the additional data structures have been studied. The suggested ordering and use of these materials within an AP Computer Science class are listed below.

1. Introduce the Grid classes.
   - Introduction to the Grid Classes (PowerPoint presentation) is available at http://www.virtualcompsci.net/gridworld/Gridworld.ppt.
   - Students should also read Part 5 of the case study narrative (http://apcentral.collegeboard.com/apc/members/repository/ap07_gridworld_casestudy_5.pdf).
2. Use the Airplane Scheduling program assignment with students.
   - Have students read the description of the assignment and complete the preassignment questions (see Appendix A). The discussion points can be used to help students construct answers to the preassignment questions that will give them insight into the program assignment.
   - Students should then complete the program assignment. Links to the starter code are available at http://www.virtualcompsci.net/gridworld/AirplaneStarterCode/Airplane.java, http://www.virtualcompsci.net/gridworld/AirplaneStarterCode/AirplaneTester.java, http://www.virtualcompsci.net/gridworld/AirplaneStarterCode/AirplaneWorld.java, which includes the starter

files `Airplane.java, AirplaneTester.java, AirplaneWorld.java` and `Passenger.java`.

- Students complete the post-assignment questions (see Appendix A).

3. Use the multiple-choice questions (see Appendix B) as assessments (as a part of a quiz or test) or as homework.
4. Use the short-answer assessment questions (see Appendix C) as a part of a formal assessment or as homework for the students.

General Description

This assignment was designed to not only familiarize students with writing code in order to implement a special type of `Grid` (an `Airplane`), but to also use that grid in context and to draw conclusions based upon what is observed in the simulation.

The `Airplane` class itself can be implemented by a wide variety of data structures. The solution is shown with a 2D matrix; however, since this is the implementation provided to students in the narrative (along with a `HashMap`), you can choose to either specify another data structure for the students or require them to select their own data structure and reflect upon their choice. Questions for discussion are provided in the appendixes that follow. These questions can either be used for group discussion; for individual reflection by each student; or, in the case where students select their own data structure, as prompts for a class presentation of the data structure that they used and an explanation of why they chose that structure. The `Airplane` class was chosen to extend `AbstractGrid` to allow you the freedom to implement any data structure you choose with your students and this assignment.

Also included in this module is starter code for the four classes of the assignment. It is left to the discretion of the instructor whether he or she chooses to provide the starter code to the students.

Appendix A

Discussion Questions

Preassignment Questions

These questions can be used before the assignment (after the description is read, but before any code is written), for a discussion part way through the assignment as a way to check for understanding, or after the assignment in a summary. Following the questions, you will find possible discussion points to use with each question in your class discussion.

1. What similarities are there between an `Airplane` and a `Grid`? What are the characteristics of each `Location` within an airplane as modeled by a `Grid`?
2. What information does an `Airplane` need to know that is separate from a `BoundedGrid`? (For example, what is the difference between the information you need to store for an `Airplane` and the information you store for a generic `BoundedGrid`?)
3. Without extending any `Grid` class or interface, what would the challenges be for implementation of the simulation?
4. Consider an `Airplane` implementation where the data structure used specifically stores `Passenger` objects. For this particular program, why is it feasible to store the objects in the grid in this manner, instead of using a data structure that can store any type of object the way that `BoundedGrid` does?
5. What are some possible data structures that could be used for the `Airplane`? What criteria would you use to choose one of these data structures? What information would you like to know about the flights in order to choose the most appropriate data structure?
6. Knowing that the planes are restricted to a relatively small size, but that the airline wants to run the simulation a very large number of times repeatedly in order to gather results, how does that affect your considerations?

Possible Discussion Points

1. Both an `Airplane` seat and a `Grid` cell can be occupied by only one actor/person at a time. You can divide an airplane into the places where people are able to stand and sit, just like you can divide a grid into locations. Each location in the airplane can be defined by a row number and a seat assignment (or classified as an aisle — a passenger standing between seats).

2. An airplane needs to know the location of the aisle and any informational methods that would be helpful in looking at seated pasengers versus standing passengers.
3. Without extending a `Grid` class, the airplane could not know where the aisle was; this information would have to be kept elsewhere, and it would make the code in other places more difficult to write and less abstracted.
4. For this simulation, we are programming it specifically to hold passengers. With the problem description given and the requirements of the client, it is not likely that other types of objects would take up seats on the airplane.
5. Students may name any data structure covered in the course. Answers will vary based on the material covered prior to this assignment. Knowledge about how full the flights are would be helpful in selecting the most appropriate data structure.
6. This indicates that speed is a more important concern than space. It would be acceptable to choose a data structure that required more memory in favor of a faster implementation (i.e., a two-dimensional array would be preferable to a linked list or even a sparse matrix).

Post-Assignment Questions

These questions can either be used for class discussion or as a part of an assessment.

1. For each of the following methods, specify the Big-Oh time for your implementation:
   `getCenterAisle`
   `aisleEmpty`
   `getOccupiedLocations`
   `get`
   `put`
2. Name a data structure, other than the one you used, that improves the time efficiency of at least one of the above listed methods (even if the other methods get slower). List the Big-Oh time for each of the above methods for the other data structure that you chose. Defend your choice over this other data structure.
3. How would you change the program if we allowed for very large aircraft that had two aisles (three seats to one side, an aisle, four seats in the middle, a second aisle and three seats to the other side)?
4. How would you change the program if there were two doors to enter the airplane?
5. Based on your observations of seating time, what would you recommend to the airline as a method of boarding passengers?
6. We only dealt with passengers as they entered the airplane. What other possibilities in an airport exist for the study of data structures?

Appendix B

Multiple-Choice Questions for Assessment GridWorld Module 5

Multiple-Choice Questions

Use the following information to answer questions 1–4.

Consider the following two implementation of a `Grid` that stores `Actors` in an array. You may assume that the average time for adding new items to the grid does not take resizing into account.

Implementation I: The array is maintained in sorted order by inserting each new actor into the array in order by location whenever an `Actor` is put into the `Grid`. The `Grid` is also set up to maintain the list in sorted order with no gaps (null elements between data) throughout the program. In order to find an `Actor` within the `Grid`, a private `getIndex` method is implemented that uses a binary search in order to locate either the index of the `Actor` itself or the index where the actor should be inserted.

Implementation II: The array is maintained in the order in which `Actors` were added to the environment. A linear search is performed in order to get the index of an `Actor` within the array any time it needs to be located.

1. What is the expected time for the `Grid` method `get` under Implementation I above?
   a. O(1)
   b. O(log n)
   c. O(n)
   d. O(n log n)
   e. $O(n^2)$

2. What is the expected time for the `Grid` method `get` under Implementation II above?
   a. O(1)
   b. O(log n)
   c. O(n)
   d. O(n log n)
   e. $O(n^2)$

3. Which of the following data structures would yield a comparable time for the `get` method as Implementation I above?
   a. `LinkedList`
   b. `HashMap`
   c. `TreeMap`
   d. `HashSet`
   e. None of the above

4. Which of the following methods would not need to be changed from their current implementation in `BoundedGrid` in order to implement a `Grid` as defined above (either Implementation I or II)?
   a. `put`
   b. `isValid`
   c. `remove`
   d. `getNeighbors`
   e. All methods would need to be reimplemented

Use the following information to answer questions 5–7.

A group of scientists want to study how different plants attract different types of insects. For this, they are going to write a computer simulation containing complex insect behavior programmed into actors. When creating the simulation, they decided it would be easier to divide the garden into sections based on the plants being grown and then record all the different types of insects in each section to see what plants are favored. Their current implementation of the `Grid` allows for only one `Actor` per location; however, they would like to change the implementation of the `Grid` to allow the actors to move freely throughout the grid, regardless of the number of actors already at that location. Consider the following two implementations for the `GardenGrid`.

Implementation I: The `GardenGrid` implementation contains a `HashMap` keyed by location. Each value stored in the `HashMap` is a `HashSet` of `IDActors` representing the collection of actors in any one grid in the garden. The `IDActor` class is an extension of `Actor` that contains a unique ID number for each actor that is used as the key for the `HashMap`.

Implementation II: The `GardenGrid` implementation contains a `HashMap` of `IDActors`. The unique ID number for each actor is used as the key for the `HashMap`.

5. With Implementation I above, if there are N locations inside the grid and S insects in the simulation, what is the worst case time for retrieving information about a single insect if we have its location and ID number?
   a. O(1)
   b. O(N)
   c. O(S*N)
   d. O(S)

e. O((log S)*N)

6. With Implementation II above, if there are N locations inside the grid and S insects in the simulation, what is the worst case time for retrieving information about a single insect if we have its location and ID number?
   a. O(1)
   b. O(N)
   c. O(S*N)
   d. O(S)
   e. O((log S)*N)

7. The scientists in the study anticipate having a very large number of actors spread relatively evenly over the different grids. They anticipate concentrations in particular areas; however, they do not anticipate all of the insects being in only one location on the grid. With this in mind, which of the above implementations will be more efficient if the scientists plan on focusing on one location in the grid at a time? This means that, during the simulation, at each time step they would retrieve all the actors stored at a given location in order to process such information as number of insects and variety of species.
   a. Implementation I would be better.
   b. Implementation II would be better.
   c. Implementation I and II would be the same.
   d. It is impossible to know the answer without more information about the behavior of the insects.
   e. It is impossible to know the information without more information about the plants.

8. When creating an implementation of the `Grid` interface, which of the following methods would most likely be affected by a change in data structure?
   a. `remove`
   b. `getValidAdjacentNeighbors`
   c. `getNeighbors`
   d. a and c only
   e. a, b, and c

Answers to Multiple-Choice Questions

1. b
2. c
3. c
4. b
5. d
6. a
7. a
8. d

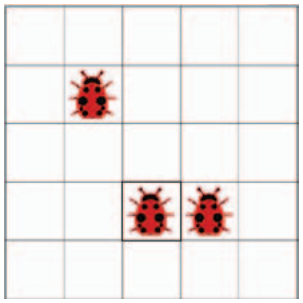
Appendix C

Short-Answer Assessment Questions Involving Big-Oh GridWorld Case Study Module 5

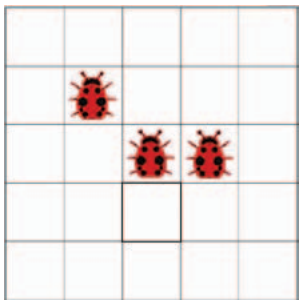
Consider a GridWorld environment used to study migration patterns of bugs. The migration of bugs is done in a very large bounded grid (think of a locust cloud 20 meters wide that is represented in a grid large enough to be comparable to the state of Texas). In addition to migrating based on their own internal control, the bugs are also migrating based upon wind patterns. The `BoundedGrid` is modified to include a method `applyWind`. The `applyWind` method takes a `Direction` (N, S, E or W) and also a start and end index for the wind. In this simplified model, the wind "pushes" each object within the start and end indices one space in the direction the wind is blowing. The indices are used opposite to the direction, so if the wind is blowing either north or south then the index is a row value; if the wind is blowing east or west, then the index is a column value.

Consider the following images of worlds after the call to applyWind:

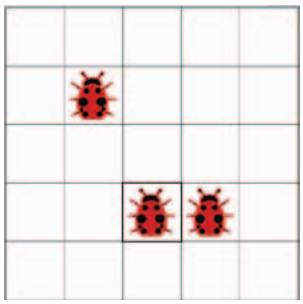
Starting Locations



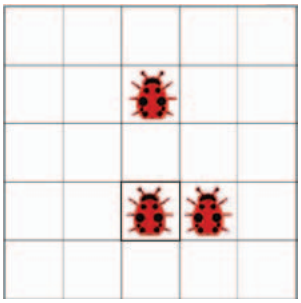
After applyWind(Direction.NORTH, 2,3)



Starting Locations



After applyWind(Direction.EAST, 1, 1)



Fill in the following table with Big-Oh notation, representing the time it would take for the `applyWind` method for each of the following implementations of the `GridWorld` environment. Use *r* for the number of rows in the grid, *c* for the number of columns in the grid and *n* for the width of the range to be blown. If another variable is needed for the analysis, be sure to define it (i.e., *b* = number of bugs in environment).

North/South Wind

Implementation	**Big-Oh time for** `applyWind`
2D `Arrayarray`	
`HashMap` keyed by `Location`	
`TreeMap` keyed by `Location`	
`Linked List` of all `Bugs` (no ordering to how they are stored)	
`HashSet` of all `Bugs`	

East/West Wind

Implementation	**Big-Oh time for** `applyWind`
2D `aArray`	
`HashMap` keyed by `Location`	
`TreeMap` keyed by `Location`	
`Linked List` of all `Bugs` (no ordering to how they are stored)	
`HashSet` of all `Bugs`	

Solutions

Use *r* for the number of rows, *c* for the number of columns in the grid and *n* for the width of the range to be blown. If another variable is needed for the analysis, be sure to define it (i.e., *b* = number of bugs in environment).

North/South Wind

Implementation	Big-Oh time for `applyWind`
2D Array	O(n*r)
`HashMap` keyed by Location	O(n*r)
`TreeMap` keyed by Location	O(n*r*log b)
`LinkedList` of all `BugBugs` (no ordering to how they are stored)	O(b)
`HashSet` of all `BugBugs`	O(b)

East/West Wind

Implementation	Big-Oh time for `applyWind`
2D Array	O(n*c)
`HashMap` keyed by Location	O(n*c)
`TreeMap` keyed by Location	O(n*c*log b)
`LinkedList` of all `BugBugs` (no ordering to how they are stored)	O(b)
`HashSet` of all `BugBugs`	O(b)

Discussion Question

How do the values of *n*, *r*, *c* and *b* affect which implementations are more efficient? (really large *n*, *r* and *c* with a really small *b* or vice versa)

Appendix D

GridWorld Case Study Part 5 — AbstractGrid Grid Data Structures Reading Worksheet

The `AbstractGrid` Class

1. Read pages 39–40 of the Student Manual.
2. What two concrete implementations of the `Grid` interface are provided?
3. What is the difference between bounded and unbounded grids?
4. Why was the `AbstractGrid` class defined?
5. How many methods of the `Grid` interface are defined by the `AbstractGrid` class?
6. How are methods in `AbstractGrid` different from methods in `Grid`?
7. What are the subclasses of `AbstractGrid`?
8. Why is `AbstractGrid` an abstract class?
9. What does the `toString` method return?
10. The concrete subclasses of the `AbstractGrid` must define which methods?
11. Does a concrete implementation of a `Grid` need to extend `AbstractGrid`? Why/Why not?
12. Complete **Do You Know?** Set 10.

GridWorld Case Study Part 5 — AbstractGrid
Grid Data Structures
Reading Worksheet (Solutions)

The `AbstractGrid` Class

1. Read pages 39–40 of the Student Manual.
2. What two concrete implementations of the `Grid` interface are provided? Bounded grid (`BoundedGrid<E>`) and unbounded grid (`UnboundedGrid<E>`)
3. What is the difference between bounded and unbounded grids? Bounded grids have a fixed number of rows and columns. Unbounded grids allow any row and column location to be a valid location in the grid.
4. Why was the `AbstractGrid` class defined? The `AbstractGrid` provides code that two concrete grid classes share.
5. How many methods of the `Grid` interface are defined by the `AbstractGrid` class? Five
6. How are methods in `AbstractGrid` different from methods in `Grid`? All methods in an interface (`Grid`) are undefined; some of the methods in an abstract class (`AbstractGrid`) can be defined .
7. What are the subclasses of `AbstractGrid`? `BoundedGrid` and `UnboundedGrid`
8. Why is `AbstractGrid` an abstract class? `AbstractGrid` does not define all the methods in `Grid` and it's not intended to be instantiated.
9. What does the `toString` method return? A `String` containing the locations of all the occupants
10. The concrete subclasses of the `AbstractGrid` must define which methods? `getNumRows`, `getNumCols`, `isValid`, `getOccupiedLocations`, `get`, `put`, `remove`
11. Does a concrete class that implements the `Grid` interface need to extend `AbstractGrid`? Why/Why not? No, but a concrete class that does not extend `AbstractGrid` needs to implement all `Grid` methods.
12. Complete **Do You Know?** Set 10.

Appendix E

GridWorld Case Study Part 5 — BoundedGrid
Grid Data Structures
Reading Worksheet

The `BoundedGrid` Class

1. Read page 41 of the Student Manual.
2. What does a `BoundedGrid` have a fixed number of ?
3. What arguments does the constructor for `BoundedGrid` require?
4. If a method attempts to access a location outside of a `BoundedGrid`, what results?
5. How does the `BoundedGrid` class store grid occupants?
6. What type of data is the `occupantArray` declared to hold?
7. When the array is constructed, what is contained in the array elements?
8. Why not declare: `private E[][] occupantArray`?
9. Why not just have the `Grid<Actor>` and avoid generics?
10. What advantage would it be to hold the occupants in an instance variable `ArrayList<ArrayList<E>> occupantList` instead of `Object[][] occupantArray`?
11. How does the `BoundedGrid` make sure that only type `E` objects are added to the array?
12. Complete **Do You Know?** Set 11.

Suppose a new class `BoundedArrayListGrid<E>` is created, which is a subclass of `AbstractGrid<E>`. The class stores the grid occupants in a two-dimensional list.

```
private ArrayList<ArrayList<E>> occupantList;
```

13. Write the `getNumRows` method for `BoundedArrayListGrid<E>`.
14. What is the time complexity (Big-Oh) for the `getNumRows` method?
15. Write the `getNumCols` method for `BoundedArrayListGrid<E>`.

16. What is the time complexity (Big-Oh) for the `getNumCols` method?
17. Write the constructor for `BoundedArrayListGrid<E>`. (Hint: use `BoundedGrid.java` as a model.)

GridWorld Case Study Part 5 — BoundedGrid
Grid Data Structures
Reading Worksheet (Solutions)

The `BoundedGrid` Class

1. Read page 41 of the Student Manual.
2. What does a `BoundedGrid` have a fixed number of? `rows and columns`
3. What arguments does the constructor for `BoundedGrid` require? `int rows, int cols`
4. If a method attempts to access a location outside of a `BoundedGrid`, what results? A run-time exception is thrown.
5. How does the `BoundedGrid` class store grid occupants? 2-D array.
6. What type of data is the `occupantArray` declared to hold? `Object`
7. When the array is constructed, what is contained in the array elements? `null`
8. Why not declare: `private E[][] occupantArray`? Java does not allow generic arrays.
9. Why not just have the `Grid<Actor>` and avoid generics? The `Grid<E>` is designed to work with any type of objects, not just actors. Limiting `Grid` to actors would prevent it from being used for making games, maps or other non `Actor` uses.
10. What advantage would it be to hold the occupants in an instance variable `ArrayList<ArrayList<E>> occupantList` instead of `Object[][] occupantArray`? Java allows generic ArrayLists.
11. How does the `BoundedGrid` make sure that only type `E` objects are added to the array? The `put` method requires elements to be of type `E`.
12. Complete **Do You Know?** Set 11.

Suppose a new class `BoundedArrayListGrid<E>` is created that is a subclass of `AbstractGrid<E>`. The class stores the grid occupants in a two-dimensional list.

```
private ArrayList<ArrayList<E>> occupantList;
```

13. Write the `getNumRows` method for `BoundedArrayListGrid<E>`.

```
public int getNumRows()
{
    return occupantList.size();
}
```

14. What is the time complexity (Big-Oh) for the `getNumRows` method? O(1)

15. Write the `getNumCols` method for `BoundedArrayListGrid<E>`.

```
public int getNumCols()
{
    return occupantList.get(0).size();
}
```

16. What is the time complexity (Big-Oh) for the getNumCols method? O(1)

17. Write the constructor for BoundedArrayListGrid<E> . (Hint: use BoundedGrid. java as a model.)

```
public BoundedArrayListGrid(int rows, int cols)
{
    if (rows <= 0)
         throw new IllegalArgumentException("rows <= 0");
    if (cols <= 0)
         throw new IllegalArgumentException("cols <= 0");

    occupantList = new ArrayList<ArrayList<E>>();

    for (int i = 0; i < rows; i++)
{

    ArrayList<E> row = new ArrayList<E>();

         occupantList.add(row);
         for (int j = 0; j < cols; j++)

          row.add(null);

}

}
```

Appendix F

GridWorld Case Study Part 5 — UnboundedGrid
Grid Data Structures
Reading Worksheet

The `UnboundedGrid` Class

1. Read page 42 of the Student Manual.
2. What arguments does the constructor for `BoundedGrid` require?
3. What locations are valid in an `UnboundedGrid`?
4. How are locations stored in an `UnboundedGrid`?
5. What is the key type for the map?
6. What is the value type for the map?
7. What is returned by the `getNumRows` and `getNumCols` methods for `UnboundedGrid`?
8. What is returned by the `isValid` method for `UnboundedGrid`?
9. What map method does `getOccupiedLocations` use to determine occupied location in the grid?
10. Complete **Do You Know?** Set 12.

Suppose a new class was constructed `UnboundedBinarySearchTreeGrid<E>`, which is a subclass of `AbstractGrid<E>`. The class stores the occupants in a binary search tree based on their `Location`. (Need to guarantee a `getLocation` method for this to work, so the constructor throws an exception if `E` is not an `Actor`.)

```
private TreeNode occupantBST;
```

11. What is returned by the `getNumRows` and `getNumCols` methods for `UnboundedGrid`?
12. Write the `get` method for `UnboundedBinarySearchTreeGrid<E>`.
13. What is the time complexity (Big-Oh) for `get` method?

GridWorld Case Study Part 5 — UnboundedGrid
Grid Data Structures
Reading Worksheet (Solutions)

The `UnboundedGrid` Class

1. Read page 42 of the Student Manual.
2. What arguments does the constructor for `BoundedGrid` require? None
3. What locations are valid in an `UnboundedGrid`? Any
4. How are locations stored in an `UnboundedGrid`? `Map<Location, E>`
5. What is the key type for the map? `Location`
6. What is the value type for the map? `E` — same type as grid occupants
7. What is returned by the `getNumRows` and `getNumCols` methods for `UnboundedGrid`? -1
8. What is returned by the `isValid` method for `UnboundedGrid`? true
9. What map method does `getOccupiedLocations` use to determine occupied location in the grid? `keyset()`
10. Complete **Do You Know?** Set 12.

Suppose a new class was constructed `UnboundedBinarySearchTreeGrid<E>`, which is a subclass of `AbstractGrid<E>`. The class stores the occupants in a binary search tree based on their `Location`. (Need to guarantee a `getLocation` method for this to work, so the constructor throws an exception if `E` is not an `Actor`.)

```
private TreeNode occupantBST;
```

11. What is returned by the `getNumRows` and `getNumCols` methods for `UnboundedGrid`? -1
12. Write the `get` method for `UnboundedBinarySearchTreeGrid<E>`.

```
public E get(Location loc)
 {
     if (loc == null)
          throw new NullPointerException("loc == null");
     return getHelper(root, loc);
 }

  private E getHelper(TreeNode node, Location loc)
```

```
{
    if (node == null)
        return null;
    Location nodeLoc = node.getLocation();
    if (loc.compareTo(nodeLoc) == 0)
        return node.getOccupant();
    else if (loc.compareTo(nodeLoc) < 0)
        return getHelper (node.getLeft(), loc);
    else
        return getHelper (node.getRight(), loc);
}
```

13. What is the time complexity (Big-Oh) for `get` method?

    O(n) //Unbalanced Tree

    O(Log n) //Balanced Tree

About the Editor and Authors

Joe Coglianese is an AP Computer Science teacher at Troy High School in Fullerton, Calif., and has been teaching since 2000. He has been an AP Reader for the AP Computer Science Exam since 2006. He has worked to develop programs to interest historically underrepresented students in technology.

Judy Hromcik is an AP Computer Science teacher at Arlington High School in Arlington, Texas. She was a member of the AP Computer Science Development Committee from 2001 to 2005, has been an AP reader and is currently a Question Leader for the AP Computer Science Exam. She has been a College Board consultant since 1997 and has conducted many AP Computer Science Summer Institutes. Hromcik wrote the solutions for the new case study GridWorld and has pilot tested GridWorld in her classes.

Kathleen A. Larson taught AP Computer Science at Kingston High School, Kingston, N.Y., from 1984 through 2005. She currently teaches Introduction to Java Programming at Ulster County Community College, part of the State University of New York System. She has served as an AP Reader, Table Leader and Question Leader for the AP Computer Science Exam; is a past member of the AP Computer Science Development Committee; and has taught numerous one-day, two-day and weeklong College Board–sponsored workshops for AP Computer Science teachers. Her publications include AP Computer Science syllabi and the *Teacher's Guide for the Marine Biology Simulation Case Study*.

Mike Lew is an AP Computer Science and AP Physics teacher at Loyola High School in Los Angeles, Calif., where he has taught since 1991. He has been teaching AP Computer Science since 1995. Lew was an AP Computer Science Exam Reader from 2001 to 2004 and has been presenting one-day AP workshops and weeklong AP Summer Institutes since 2004. He has also authored a teacher's guide for the textbook *Head First Java*.

Leigh Ann Sudol has been teaching computer science since 1996. During that time she has taught in public high schools, community colleges and universities. Her courses have been delivered online, in person and also in mixed settings. She is currently a visiting lecturer in the School of Computer Science at Carnegie Mellon University and a member of the board of directors of the Computer Science Teachers Association, where she serves as chair of the publications committee. Sudol has served as the high school liaison for the Special Interest Group on Computer Science Education (SIGCSE) 2007. She coauthored *Java Software Structures for AP Computer Science AB* and also the most recent edition of *Addison Wesley's Review for the AP Computer Science Exam in Java*. She has been a College Board consultant and an AP Computer Science Exam Reader since 2002 and a Question Leader for the AP Exam since 2005.

Fran Trees taught AP Computer Science at Westfield High School in Westfield, N.J., from 1983 to 2001. She presently teaches computer science and mathematics at Drew University in Madison, N.J. She has served as an AP Reader, Question Leader and Exam Leader for the AP Computer Science Exam. She has also served as a member of the AP Computer Science Test Development Committee and a member of various ad hoc committees for AP Computer Science. Trees has served as a College Board consultant in computer science since 1985 and is the primary author of the *Teacher's Guide for AP Computer Science* (C++) and the *Advanced Placement® Computer Science Study Guide to Accompany Java Concepts*.