The College Board: Connecting Students to College Success

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 association is composed of more than 5,000 schools, colleges, universities, and other educational organizations. Each year, the College Board serves seven million students and their parents, 23,000 high schools, and 3,500 colleges through major programs and services in college admissions, guidance, assessment, financial aid, enrollment, and teaching and learning. Among its best-known programs are the SAT®, the PSAT/NMSQT®, and the Advanced Placement Program® (AP®). 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.
GridWorld Case Study

(2008 AP® CS Exam)

The AP® Program wishes to acknowledge and to thank the following individuals for their contributions to the GridWorld Case Study.

Narrative by Chris Nevison and Barbara Cloud Wells, Colgate University

Framework design and implementation by Cay Horstmann, San Jose State University

Images created by Chris Renard, a student at the School for the Talented and Gifted, Dallas Independent School District

Introduction

The GridWorld Case Study provides a graphical environment where visual objects inhabit and interact in a two-dimensional grid. In this case study, you will design and create “actor” objects, add them to a grid, and determine whether the actors behave according to their specifications. A graphical user interface (GUI) is provided that displays the grid and the actors. In addition, the GUI has a facility for adding actors to the grid and for invoking methods on them.

This guide for GridWorld is organized into the following parts:

Part 1: Provides experiments to observe the attributes and behavior of the actors.

Part 2: Defines Bug variations.

Part 3: Explores the code that is needed to understand and create actors.

Part 4: Defines classes that extend the Critter class.

Part 5: (CS AB only) Explains grid data structures.
Part 1: Observing and Experimenting with GridWorld

Running the Demo

You need to download and install the case study files. Instructions are provided in a separate document.

Once the code is installed, simply compile and run the BugRunner.java application supplied with the case study. The GridWorld GUI will show a grid containing two actors, a “bug” and a “rock.” Clicking on the Step button runs one step, making each actor act once. Clicking on the Run button carries out a series of steps until the Stop button is clicked. The delay between steps during a run can be adjusted with the slider. Try it!
Clicking on an empty cell in the grid displays a drop-down menu that shows the constructors for different actor types.

The menu lists constructors for the classes of all objects that have ever been placed in the grid.

Selecting one of these constructors places an instance of that type in the grid. If the constructor has parameters, a dialog window appears, requesting parameter values. For example, after selecting the constructor `info.gridworld.actor.Bug(java.awt.Color)`, the following dialog window appears. Clicking in the color bar produces a drop-down menu from which to choose a color.
What’s Happening?

The grid uses directions as on a map: north is up on the screen, east is to the right, south is down, west is to the left. The diagonal directions are northeast, southeast, southwest, and northwest.

One attribute of the bug is its direction (indicated by its antennae). Initially, the bug faces north.

Make a bug, take several steps and observe its behavior. Then add more rocks and bugs by clicking on empty cells and selecting the actors of your choice. Answer the following questions based on your observations using the Step and Run buttons.

Do You Know?
Set 1

1. Does the bug always move to a new location? Explain.
2. In which direction does the bug move?
3. What does the bug do if it does not move?
4. What does a bug leave behind when it moves?
5. What happens when the bug is at an edge of the grid? (Consider whether the bug is facing the edge as well as whether the bug is facing some other direction when answering this question.)
6. What happens when a bug has a rock in the location immediately in front of it?
7. Does a flower move?
8. What behavior does a flower have?
9. Does a rock move or have any other behavior?
10. Can more than one actor (bug, flower, rock) be in the same location in the grid at the same time?
Exploring Actor State and Behavior

When you click on a cell containing an actor (bug, flower, or rock), a drop-down menu displays the methods that you can invoke on the actor. The methods that appear above the separator line are specified by the class that defines this actor; those that appear below the line are the methods inherited from the `Actor` class.

Experiment with the different methods to see how they work. Accessor methods will have their results displayed in a dialog window. Modifier methods will cause an appropriate change in the display of the actor in the grid. If parameters are needed for a method, you provide them through a dialog window such as the following.
If you select one of the methods in the menu, it will be invoked for this actor. As an example, click on a bug to see the menu of methods. Now observe what happens when you select the `void setDirection(int)` method. The bug will change its direction to the angle that you supply. Try selecting some of the other methods. What about the `act` method? If you invoked that method, the actor will behave as if you had clicked on the `Step` button, but no other actors will do anything. A bug will move forward or turn to the right, a flower’s color will darken, and a rock will do nothing. The `Step` button simply causes the `act` method to be invoked on all actors in the grid.

**Exercises**

By clicking on a cell containing a bug, flower, or rock, do the following.

1. Test the `setDirection` method with the following inputs and complete the table, giving the compass direction each input represents.

<table>
<thead>
<tr>
<th>Degrees</th>
<th>Compass Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>North</td>
</tr>
<tr>
<td>45</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td></td>
</tr>
<tr>
<td>135</td>
<td></td>
</tr>
<tr>
<td>180</td>
<td></td>
</tr>
<tr>
<td>225</td>
<td></td>
</tr>
<tr>
<td>270</td>
<td></td>
</tr>
<tr>
<td>315</td>
<td></td>
</tr>
<tr>
<td>360</td>
<td></td>
</tr>
</tbody>
</table>

2. Move a bug to a different location using the `moveTo` method. In which directions can you move it? How far can you move it? What happens if you try to move the bug outside the grid?

3. Change the color of a bug, a flower, and a rock. Which method did you use?

4. Move a rock on top of a bug and then move the rock again. What happened to the bug?
## GUI Summary

<table>
<thead>
<tr>
<th>Mouse Action</th>
<th>Keyboard Shortcut</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Click on an empty location</td>
<td>Select empty location with cursor keys and press the \texttt{Enter} key</td>
<td>Shows the constructor menu</td>
</tr>
<tr>
<td>Click on an occupied location</td>
<td>Select occupied location with cursor keys and press the \texttt{Enter} key</td>
<td>Shows the method menu</td>
</tr>
<tr>
<td>Select the Location -&gt; \texttt{Delete} menu item</td>
<td>Press the \texttt{Delete} key</td>
<td>Removes the occupant in the currently selected location from the grid</td>
</tr>
<tr>
<td>Click on the Step button</td>
<td></td>
<td>Calls \texttt{act} on each actor</td>
</tr>
<tr>
<td>Click on the Run button</td>
<td></td>
<td>Starts run mode (in run mode, the action of the Step button is carried out repeatedly)</td>
</tr>
<tr>
<td>Click on the Stop button</td>
<td></td>
<td>Stops run mode</td>
</tr>
<tr>
<td>Adjust the Slow/Fast slider</td>
<td></td>
<td>Changes speed of run mode</td>
</tr>
<tr>
<td>Select the Location -&gt; \texttt{Zoom in/Zoom out} menu item</td>
<td>Press the Ctrl+PgUp / Ctrl+PgDn keys</td>
<td>Zooms grid display in or out</td>
</tr>
<tr>
<td>Adjust the scroll bars next to grid</td>
<td>Move the location with the cursor keys</td>
<td>Scrolls to other parts of the grid (if the grid is too large to fit inside the window)</td>
</tr>
<tr>
<td>Select the World -&gt; Set grid menu item</td>
<td></td>
<td>Changes between bounded and unbounded grids</td>
</tr>
<tr>
<td>Select the World -&gt; Quit menu item</td>
<td>Press the Ctrl+Q keys</td>
<td>Quits GridWorld</td>
</tr>
</tbody>
</table>
GridWorld Case Study

Part 2: Bug Variations

Methods of the Bug Class

The Bug class provides three methods that specify how bugs move and turn.

- public boolean canMove()
  - tests whether the bug can move forward into a location that is empty or contains a flower

- public void move()
  - moves the bug forward, putting a flower into the location it previously occupied

- public void turn()
  - turns the bug 45 degrees to the right without changing its location

These methods are used in the bug’s act method.

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

The experiments in the previous section showed that the bug moves forward when it can. When the bug has a rock in front of it or is facing an edge of the grid, it cannot move, so it turns. However, it can step on a flower (which removes the flower from the grid). When the bug moves, it leaves a flower in its previous location. This behavior is determined by the act method and the three methods that the act method calls.
Extending the Bug Class

A new type of bug with different behavior can be created by extending the Bug class and overriding the act method. No new methods need to be added; the act method uses the three auxiliary methods from the Bug class listed above. A BoxBug moves in a square pattern. In order to keep track of its movement, the BoxBug class has two instance variables, sideLength and steps.
Do You Know?
Set 2

The source code for the BoxBug class is in Appendix C.

1. What is the role of the instance variable sideLength?
2. What is the role of the instance variable steps?
3. Why is the turn method called twice when steps becomes equal to sideLength?
4. Why can the move method be called in the BoxBug class when there is no move method in the BoxBug code?
5. After a BoxBug is constructed, will the size of its square pattern always be the same? Why or why not?
6. Can the path a BoxBug travels ever change? Why or why not?
7. When will the value of steps be zero?

Runner Classes

In order to observe the behavior of one or more actors, a “runner” class is required. That class constructs an ActorWorld object, places actors into it, and shows the world. For the bug, this class is BugRunner. For the box bug, it is BoxBugRunner. In each of these runner classes, the overloaded add method is used to place actors (instances of classes such as Bug, BoxBug, Rock) into the grid of the ActorWorld. The add method with an Actor parameter and a Location parameter places an actor at a specified location. The add method with an Actor parameter but no Location parameter places an actor at a random empty location. When you write your own classes that extend Bug, you also need to create a similar runner class.

The source code for the BoxBugRunner class is at the end of this part of GridWorld.
Exercises

In the following exercises, write a new class that extends the Bug class. Override the act method to define the new behavior.

1. Write a class CircleBug that is identical to BoxBug, except that in the act method the turn method is called once instead of twice. How is its behavior different from a BoxBug?
2. Write a class SpiralBug that drops flowers in a spiral pattern. Hint: Imitate BoxBug, but adjust the side length when the bug turns. You may want to change the world to an UnboundedGrid to see the spiral pattern more clearly.
3. Write a class `ZBug` to implement bugs that move in a “Z” pattern, starting in the top left corner. After completing one “Z” pattern, a `ZBug` should stop moving. In any step, if a `ZBug` can’t move and is still attempting to complete its “Z” pattern, the `ZBug` does not move and should not turn to start a new side. Supply the length of the “Z” as a parameter in the constructor. The following image shows a “Z” pattern of length 4. Hint: Notice that a `ZBug` needs to be facing east before beginning its “Z” pattern.
4. Write a class `DancingBug` that “dances” by making different turns before each move. The `DancingBug` constructor has an integer array as parameter. The integer entries in the array represent how many times the bug turns before it moves. For example, an array entry of 5 represents a turn of 225 degrees (recall one turn is 45 degrees). When a dancing bug acts, it should turn the number of times given by the current array entry, then act like a `Bug`. In the next move, it should use the next entry in the array. After carrying out the last turn in the array, it should start again with the initial array value so that the dancing bug continually repeats the same turning pattern.

The `DancingBugRunner` class should create an array and pass it as a parameter to the `DancingBug` constructor.

5. Study the code for the `BoxBugRunner` class. Summarize the steps you would use to add another `BoxBug` actor to the grid.

---

**BoxBugRunner.java**

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

/**
 * This class runs a world that contains box bugs.
 * This class is not tested on the AP CS A and AB exams.
 */
public class BoxBugRunner
{
    public static void main(String[] args)
    {
        ActorWorld world = new ActorWorld();
        BoxBug alice = new BoxBug(6);
        alice.setColor(Color.ORANGE);
        BoxBug bob = new BoxBug(3);
        world.add(new Location(7, 8), alice);
        world.add(new Location(5, 5), bob);
        world.show();
    }
}
```

---

© 2007 The College Board. All rights reserved. Visit apcentral.collegeboard.com (for AP professionals) and www.collegeboard.com/apstudents (for AP students and parents).
GridWorld Case Study

Part 3: GridWorld Classes and Interfaces

In our example programs, a grid contains actors that are instances of classes that extend the Actor class. There are two classes that implement the Grid interface: BoundedGrid and UnboundedGrid. Locations in a grid are represented by objects of the Location class. An actor knows the grid in which it is located as well as its current location in the grid. The relationships among these classes are shown in the following figure.

The Location Class

Every actor that appears has a location in the grid. The Location class encapsulates the coordinates for an actor’s position in a grid. It also provides methods that determine relationships between locations and compass directions.

Every actor in the grid also has a direction. Directions are represented by compass directions measured in degrees: 0 degrees is north, 45 degrees is northeast, 90 degrees is east, etc. The Location class provides eight constants that specify the compass directions.
public static final int NORTH = 0;
public static final int EAST = 90;
public static final int SOUTH = 180;
public static final int WEST = 270;
public static final int NORTHEAST = 45;
public static final int SOUTHEAST = 135;
public static final int SOUTHWEST = 225;
public static final int NORTHWEST = 315;

In addition, the Location class specifies constants for commonly used turn angles. For example, Location.HALF_RIGHT denotes a turn by 45 degrees. Here are the constants for the turn angles.

public static final int LEFT = -90;
public static final int RIGHT = 90;
public static final int HALF_LEFT = -45;
public static final int HALF_RIGHT = 45;
public static final int FULL_CIRCLE = 360;
public static final int HALF_CIRCLE = 180;
public static final int AHEAD = 0;

To make an actor turn by a given angle, set its direction to the sum of the current direction and the turn angle. For example, the turn method of the Bug class makes this call.

setDirection(getDirection() + Location.HALF_RIGHT);

A location in a grid has a row and a column. These values are parameters of the Location constructor.

public Location(int r, int c)

Two accessor methods are provided to return the row and column for a location.

public int getRow()
public int getCol()
Two other Location methods give information about the relationships between locations and directions.

```
public Location getAdjacentLocation(int direction)
    returns the adjacent location in the compass direction that is closest to
direction

public int getDirectionToward(Location target)
    returns the closest compass direction from this location toward target
```

For example, assume you have the statement below.

```
Location loc1 = new Location(5, 7);
```

The following statements will result in the values indicated by the comments.

```
Location loc2 = loc1.getAdjacentLocation(Location.WEST);
    // loc2 has row 5, column 6
Location loc3 = loc1.getAdjacentLocation(Location.NORTHEAST);
    // loc3 has row 4, column 8
int dir = loc1.getDirectionToward(new Location(6, 8));
    // dir has value 135 (degrees)
```

Note that the row values increase as you go south (down the screen) and the column values increase as you go east (to the right on the screen).

The Location class defines the equals method so that loc.equals(other) returns true if other is a Location object with the same row and column values as loc and returns false otherwise.

The Location class implements the Comparable interface. The compareTo method compares two Location objects. The method call loc.compareTo(other) returns a negative integer if loc has a smaller row coordinate than other, or if they have the same row and loc has a smaller column coordinate than other. The call returns 0 if loc and other have the same row and column values. Otherwise, the call returns a positive integer.
Do You Know?

Set 3

The API for the Location class is in Appendix B.

Assume the following statements when answering the following questions.

```java
Location loc1 = new Location(4, 3);
Location loc2 = new Location(3, 4);
```

1. How would you access the row value for `loc1`?
2. What is the value of `b` after the following statement is executed?
   ```java
   boolean b = loc1.equals(loc2);
   ```
3. What is the value of `loc3` after the following statement is executed?
   ```java
   Location loc3 = loc2.getAdjacentLocation(Location.SOUTH);
   ```
4. What is the value of `dir` after the following statement is executed?
   ```java
   int dir = loc1.getDirectionToward(new Location(6, 5));
   ```
5. How does the `getAdjacentLocation` method know which adjacent location to return?
The Grid Interface

The interface Grid<E> specifies the methods for any grid that contains objects of the type E. Two classes, BoundedGrid<E> and UnboundedGrid<E> implement the interface.

You can check whether a given location is within a grid with this method.

```java
boolean isValid(Location loc)
    returns true if loc is valid in this grid, false otherwise
    Precondition: loc is not null
```

All methods in this case study have the implied precondition that their parameters are not null. The precondition is emphasized in this method to eliminate any doubt whether null is a valid or invalid location. The null reference does not refer to any location, and you must not pass null to the isValid method.

The following three methods allow us to put objects into a grid, remove objects from a grid, and get a reference to an object in a grid.

```java
E put(Location loc, E obj)
    puts obj at location loc in this grid and returns the object previously at that location (or null if the location was previously unoccupied)
    Precondition: (1) loc is valid in this grid (2) obj is not null

E remove(Location loc)
    removes the object at location loc and returns it (or null if the location is unoccupied)
    Precondition: loc is valid in this grid

E get(Location loc)
    returns the object at location loc (or null if the location is unoccupied)
    Precondition: loc is valid in this grid
```

An additional method returns all occupied locations in a grid.

```java
ArrayList<Location> getOccupiedLocations()
```
Four methods are used to collect adjacent locations or neighbor elements of a given location in a grid. The use of these methods is demonstrated by the examples in Part 4.

`ArrayList<Location> getValidAdjacentLocations(Location loc)`
returns all valid locations adjacent to `loc` in this grid

Precondition: `loc` is valid in this grid

`ArrayList<Location> getEmptyAdjacentLocations(Location loc)`
returns all valid empty locations adjacent to `loc` in this grid

Precondition: `loc` is valid in this grid

`ArrayList<Location> getOccupiedAdjacentLocations(Location loc)`
returns all valid occupied locations adjacent to `loc` in this grid

Precondition: `loc` is valid in this grid

`ArrayList<E> getNeighbors(Location loc)`
returns all objects in the occupied locations adjacent to `loc` in this grid

Precondition: `loc` is valid in this grid

Finally, you can get the number of rows and columns of a grid.

```java
int getNumRows();
int getNumCols();
```

For unbounded grids, these methods return -1.

Do You Know?
Set 4

The API for the `Grid` interface is in Appendix B.

1. How can you obtain a count of the objects in a grid? How can you obtain a count of the empty locations in a bounded grid?
2. How can you check if location (10,10) is in a grid?
3. `Grid` contains method declarations, but no code is supplied in the methods. Why? Where can you find the implementations of these methods?
4. All methods that return multiple objects return them in an `ArrayList`. Do you think it would be a better design to return the objects in an array? Explain your answer.
The **Actor Class**

The following accessor methods of the **Actor** class provide information about the state of the actor.

```java
public Color getColor()
public int getDirection()
public Grid<Actor> getGrid()
public Location getLocation()
```

One method enables an actor to add itself to a grid; another enables the actor to remove itself from the grid.

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

The **putSelfInGrid** method establishes the actor’s location as well as the grid in which it is placed. The **removeSelfFromGrid** method removes the actor from its grid and makes the actor’s grid and location both null.

> When adding or removing actors, do **not** use the **put** and **remove** methods of the **Grid** interface. Those methods do not update the **location** and **grid** instance variables of the actor. That is a problem since most actors behave incorrectly if they do not know their location. To ensure correct actor behavior, always use the **putSelfInGrid** and **removeSelfFromGrid** methods of the **Actor** class.

To move an actor to a different location, use the following method.

```java
public void moveTo(Location loc)
```

The **moveTo** method allows the actor to move to any valid location. If the actor calls **moveTo** for a location that contains another actor, the other one removes itself from the grid and this actor moves into that location.
You can change the direction or color of an actor with the methods below.

```java
public void setColor(Color newColor)
public void setDirection(int newDirection)
```

These `Actor` methods provide the tools to implement behavior for an actor. Any class that extends `Actor` defines its behavior by overriding the `act` method.

```java
public void act()
```

The `act` method of the `Actor` class reverses the direction of the actor. You override this method in subclasses of `Actor` to define actors with different behavior. If you extend `Actor` without specifying an `act` method in the subclass, or if you add an `Actor` object to the grid, you can observe that the actor flips back and forth with every step.

The `Bug`, `Flower`, and `Rock` classes provide examples of overriding the `act` method.

The API for the `Grid` interface and the `Location` and `Actor` classes is provided in Appendix B. The following questions help analyze the code for `Actor`.

---

**Do You Know?**

**Set 5**

The API for the `Actor` class is in Appendix B.

1. Name three properties of every actor.
2. When an actor is constructed, what is its direction and color?
3. Why do you think that the `Actor` class was created as a class instead of an interface?
4. Can an actor put itself into a grid twice without first removing itself? Can an actor remove itself from a grid twice? Can an actor be placed into a grid, remove itself, and then put itself back? Try it out. What happens?
5. How can an actor turn 90 degrees to the right?
Extending the Actor Class

The Bug, Flower, and Rock classes extend Actor in different ways. Their behavior is specified by how they override the act method.

The Rock Class

A rock acts by doing nothing at all. The act method of the Rock class has an empty body.

The Flower Class

A flower acts by darkening its color, without moving. The act method of the Flower class reduces the values of the red, green, and blue components of the color by a constant factor.

The Bug Class

A bug acts by moving forward and leaving behind a flower. A bug cannot move into a location occupied by a rock, but it can move into a location that is occupied by a flower, which is then removed. If a bug cannot move forward because the location in front is occupied by a rock or is out of the grid, then it turns right 45 degrees.

In Part 2, exercises were given to extend the Bug class. All of the exercises required the act method of the Bug class to be overridden to implement the desired behavior. The act method uses the three auxiliary methods in the Bug class: canMove, move, and turn. These auxiliary methods call methods from Actor, the superclass.

The canMove method determines whether it is possible for this Bug to move. It uses a Java operator called instanceof (not part of the AP CS Java subset). This operator is used as in the following way.

```
expr instanceof Name
```

Here, expr is an expression whose value is an object and Name is the name of a class or interface type. The instanceof operator returns true if the object has the specified type. If Name is a class name, then the object must be an instance of that class itself or one of its subclasses. If Name is an interface name, then the object must belong to a class that implements the interface.
This statement in the `canMove` method checks whether the object in the adjacent location is `null` or if it is a `Flower`.

```java
return (neighbor == null) || (neighbor instanceof Flower);
```

The following statement in the `canMove` method checks that the bug is actually in a grid—it would be possible for the bug not to be in a grid if some other actor removed it.

```java
if (gr == null) return false;
```

The other code for `canMove` is explored in the questions at the end of this section.

The `move` method for `Bug` moves it to the location immediately in front and puts a flower in its previous location. The `turn` method for `Bug` turns it 45 degrees to the right. The code for these methods is explored in the following questions.

![Ladybug](image)

**Do You Know?**

**Set 6**

The source code for the `Bug` class is in Appendix C.

1. Which statement(s) in the `canMove` method ensures that a bug does not try to move out of its grid?
2. Which statement(s) in the `canMove` method determines that a bug will not walk into a rock?
3. Which methods of the `Grid` interface are invoked by the `canMove` method and why?
4. Which method of the `Location` class is invoked by the `canMove` method and why?
5. Which methods inherited from the `Actor` class are invoked in the `canMove` method?
6. What happens in the `move` method when the location immediately in front of the bug is out of the grid?
7. Is the variable `loc` needed in the `move` method, or could it be avoided by calling `getLocation()` multiple times?
8. Why do you think the flowers that are dropped by a bug have the same color as the bug?
9. When a bug removes itself from the grid, will it place a flower into its previous location?
10. Which statement(s) in the move method places the flower into the grid at the bug’s previous location?
11. If a bug needs to turn 180 degrees, how many times should it call the turn method?

Group Activity

Organize groups of 3–5 students.

1. Specify: Each group creates a class called Jumper. This actor can move forward two cells in each move. It “jumps” over rocks and flowers. It does not leave anything behind it when it jumps.

In the small groups, discuss and clarify the details of the problem:

   a. What will a jumper do if the location in front of it is empty, but the location two cells in front contains a flower or a rock?
   b. What will a jumper do if the location two cells in front of the jumper is out of the grid?
   c. What will a jumper do if it is facing an edge of the grid?
   d. What will a jumper do if another actor (not a flower or a rock) is in the cell that is two cells in front of the jumper?
   e. What will a jumper do if it encounters another jumper in its path?
   f. Are there any other tests the jumper needs to make?

2. Design: Groups address important design decisions to solve the problem:

   a. Which class should Jumper extend?
   b. Is there an existing class that is similar to the Jumper class?
   c. Should there be a constructor? If yes, what parameters should be specified for the constructor?
   d. Which methods should be overridden?
   e. What methods, if any, should be added?
   f. What is the plan for testing the class?

3. Code: Implement the Jumper and JumperRunner classes.
4. Test: Carry out the test plan to verify that the Jumper class meets the specification.
What Makes It Run? (Optional)

A graphical user interface (GUI) has been provided for running GridWorld programs. The World class makes the connection between the GUI and the classes already described. The GUI asks the world for its grid, locates the grid occupants, and draws them. The GUI allows the user to invoke the step method of the world, either taking one step at a time or running continuously. After each step, the GUI redisplays the grid.

For our actors, a subclass of the World class called ActorWorld is provided. ActorWorld defines a step method that invokes act on each actor.

The World and ActorWorld classes are not tested in the AP CS Exam.

Other worlds can be defined that contain occupants other than actors. By providing different implementations of the step method and other methods of the World class, one can produce simulations, games, and so on. This is not tested on the AP CS Exam.

In order to display the GUI, a runner program constructs a world, adds occupants to it, and invokes the show method on the world. That method causes the GUI to launch.

The ActorWorld class has a constructor with a Grid<Actor> parameter. Use that constructor to explore worlds with grids other than the default 10 x 10 grid.

The ActorWorld has two methods for adding an actor.

```java
public void add(Location loc, Actor occupant)
public void add(Actor occupant)
```

The add method without a Location parameter adds an actor at a random empty location.

When adding actors to a world, be sure to use the add method of the ActorWorld class and not the put method of the Grid interface. The add method calls the Actor method putSelfInGrid. As explained previously, the putSelfInGrid method sets the actor’s references to its grid and location and calls the grid method put, giving the grid a reference to the actor.
The `remove` method removes an actor from a given location and returns the `Actor` that has been removed.

```java
public Actor remove(Location loc)
```

The relationship between the GUI, world, and actor classes is shown in the following figure. Note that the GUI has no knowledge of actors. It can show occupants in any world. Conversely, actors have no knowledge of the GUI.
GridWorld Case Study

Part 4: Interacting Objects

The Critter Class

Critters are actors that share a common pattern of behavior, but the details may vary for each type of critter. When a critter acts, it first gets a list of actors to process. It processes those actors and then generates the set of locations to which it may move, selects one, and moves to that location.

Different types of critters may select move locations in different ways, may have different ways of selecting among them, and may vary the actions they take when they make the move. For example, one type of critter might get all the neighboring actors and process each one of them in some way (change their color, make them move, and so on). Another type of critter may get only the actors to its front, front-right, and front-left and randomly select one of them to eat. A simple critter may get all the empty neighboring locations, select one at random, and move there. A more complex critter may only move to the location in front or behind and make a turn if neither of these locations is empty.

Each of these behaviors fits a general pattern. This general pattern is defined in the act method for the Critter class, a subclass of Actor. This act method invokes the following five methods.

```java
ArrayList<Actor> getActors()
void processActors(ArrayList<Actor> actors)
ArrayList<Location> getMoveLocations()
Location selectMoveLocation(ArrayList<Location> locs)
void makeMove(Location loc)
```

These methods are implemented in the Critter class with simple default behavior—see the following section. Subclasses of Critter should override one or more of these methods.

It is usually not a good idea to override the act method in a Critter subclass. The Critter class was designed to represent actors that process other actors and then move. If you find the act method unsuitable for your actors, you should consider extending Actor, not Critter.
Do You Know?
Set 7

The source code for the Critter class is in Appendix C.

1. What methods are implemented in Critter?
2. What are the five basic actions common to all critters when they act?
4. Describe three ways that a critter could process actors.
5. What three methods must be invoked to make a critter move? Explain each of these methods.
6. Why is there no Critter constructor?

Default Critter Behavior

Before moving, critters process other actors in some way. They can examine them, move them, or even eat them.

There are two steps involved:

1. Determination of which actors should be processed
2. Determination of how they should be processed

The getActors method of the Critter class gets a list of all neighboring actors. This behavior can be inherited in subclasses of Critter. Alternatively, a subclass can decide to process a different set of actors, by overriding the getActors method.

The processActors method in the Critter class eats (that is, removes) actors that are not rocks or critters. This behavior is either inherited or overridden in subclasses.

When the critter has completed processing actors, it moves to a new location. This is a three-step process.

1. Determination of which locations are candidates for the move
2. Selection of one of the candidates
3. Making the move

Each of these steps is implemented in a separate method. This allows subclasses to change each behavior separately.
The Critter implementation of the getMoveLocations method returns all empty adjacent locations. In a subclass, you may want to compute a different set of locations. One of the examples in the case study is a CrabCritter that can only move sideways.

Once the candidate locations have been determined, the critter needs to select one of them. The Critter implementation of selectMoveLocation selects a location at random. However, other critters may want to work harder and pick the location they consider best, such as the one with the most food or the one closest to their friends.

Finally, when a location has been selected, it is passed to the makeMove method. The makeMove method of the Critter class simply calls moveTo, but you may want to override the makeMove method to make your critters turn, drop a trail of rocks, or take other actions.

Note that there are postconditions on the five Critter methods called by act. When you override these methods, you should maintain these postconditions.

The design philosophy behind the Critter class is that the behavior is carried out in separate phases, each of which can be overridden independently. The postconditions help to ensure that subclasses implement behavior that is consistent with the purpose of each phase.
Extending the Critter Class

The ChameleonCritter class defines a new type of critter that gets the same neighboring actors as a Critter. However, unlike a Critter, a ChameleonCritter doesn’t process actors by eating them. Instead, when a ChameleonCritter processes actors, it randomly selects one and changes its own color to the color of the selected actor.

The ChameleonCritter class also overrides the makeMove method of the Critter class. When a ChameleonCritter moves, it turns toward the new location.
The following figure shows the relationships among Actor, Critter, and ChameleonCritter, as well as the CrabCritter class that is discussed in the following section.

Do You Know?
Set 8

The source code for the ChameleonCritter class is in Appendix C.

1. Why does act cause a ChameleonCritter to act differently from a Critter even though ChameleonCritter does not override act?
2. Why does the makeMove method of ChameleonCritter call super.makeMove?
3. How would you make the ChameleonCritter drop flowers in its old location when it moves?
4. Why doesn’t ChameleonCritter override the getActors method?
5. Which class contains the `getLocation` method?
6. How can a `Critter` access its own grid?

**Another Critter**

A `CrabCritter` is a critter that eats whatever is found in the locations immediately in front, to the right-front, or to the left-front of it. It will not eat a rock or another critter (this restriction is inherited from the `Critter` class). A `CrabCritter` can move only to the right or to the left. If both locations are empty, it randomly selects one. If a `CrabCritter` cannot move, then it turns 90 degrees, randomly to the left or right.
Do You Know?
Set 9

The source code for the CrabCritter class is reproduced at the end of this part of GridWorld. This code is not required for the AP CS Exam, but working with the code is good practice as preparation for the exam.

1. Why doesn’t CrabCritter override the processActors method?
2. Describe the process a CrabCritter uses to find and eat other actors. Does it always eat all neighboring actors? Explain.
3. Why is the getLocationsInDirections method used in CrabCritter?
4. If a CrabCritter has location (3, 4) and faces south, what are the possible locations for actors that are returned by a call to the getActors method?
5. What are the similarities and differences between the movements of a CrabCritter and a Critter?
6. How does a CrabCritter determine when it turns instead of moving?
7. Why don’t the CrabCritter objects eat each other?

Exercises

1. Modify the processActors method in ChameleonCritter so that if the list of actors to process is empty, the color of the ChameleonCritter will darken (like a flower).

In the following exercises, your first step should be to decide which of the five methods—getActors, processActors, getMoveLocations, selectMoveLocation, and makeMove—should be changed to get the desired result.

2. Create a class called ChameleonKid that extends ChameleonCritter as modified in exercise 1. A ChameleonKid changes its color to the color of one of the actors immediately in front or behind. If there is no actor in either of these locations, then the ChameleonKid darkens like the modified ChameleonCritter.
3. Create a class called RockHound that extends Critter. A RockHound gets the actors to be processed in the same way as a Critter. It removes any rocks in that list from the grid. A RockHound moves like a Critter.

4. Create a class BlusterCritter that extends Critter. A BlusterCritter looks at all of the neighbors within two steps of its current location. (For a BlusterCritter not near an edge, this includes 24 locations). It counts the number of critters in those locations. If there are fewer than \( c \) critters, the BlusterCritter’s color gets brighter (color values increase). If there are \( c \) or more critters, the BlusterCritter’s color darkens (color values decrease). Here, \( c \) is a value that indicates the courage of the critter. It should be set in the constructor.

5. Create a class QuickCrab that extends CrabCritter. A QuickCrab processes actors the same way a CrabCritter does. A QuickCrab moves to one of the two locations, randomly selected, that are two spaces to its right or left, if that location and the intervening location are both empty. Otherwise, a QuickCrab moves like a CrabCritter.

6. Create a class KingCrab that extends CrabCritter. A KingCrab gets the actors to be processed in the same way a CrabCritter does. A KingCrab causes each actor that it processes to move one location further away from the KingCrab. If the actor cannot move away, the KingCrab removes it from the grid. When the KingCrab has completed processing the actors, it moves like a CrabCritter.

**Group Activity**

Organize groups of 2–4 students.

1. Specify: Each group specifies a new creature that extends Critter. The specifications must describe the properties and behavior of the new creature in detail.

2. Design: The groups exchange specifications. Each group reads the specification that it received and determines the needed variables and basic algorithms for the creature.

3. Code: Each group implements the code for the creature in the specification that it received.

4. Test: Each group writes test cases for the creature that it specified in step 1. Test cases are exchanged as in step 2. The recipients verify that the implementations meet the specifications.
CrabCritter.Java

import info.gridworld.actor.Actor;
import info.gridworld.actor.Critter;
import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;
import java.awt.Color;
import java.util.ArrayList;

/**
 * A CrabCritter looks at a limited set of neighbors when it eats and moves.
 * This class is not tested on the AP CS A and AB Exams.
 */
public class CrabCritter extends Critter {
    public CrabCritter() {
        setColor(Color.RED);
    }

    /**
     * A crab gets the actors in the three locations immediately in front, to its
     * front-right and to its front-left
     * @return a list of actors occupying these locations
     */
    public ArrayList<Actor> getActors() {
        ArrayList<Actor> actors = new ArrayList<Actor>();
        int[] dirs = {Location.AHEAD, Location.HALF_LEFT, Location.HALF_RIGHT};
        for (Location loc : getLocationsInDirections(dirs)) {
            Actor a = getGrid().get(loc);
            if (a != null) actors.add(a);
        }
        return actors;
    }

    /**
     * @return list of empty locations immediately to the right and to the left
     */
    public ArrayList<Location> getMoveLocations() {
        ArrayList<Location> locs = new ArrayList<Location>();
        int[] dirs = {Location.LEFT, Location.RIGHT};
        for (Location loc : getLocationsInDirections(dirs)) {
            if (getGrid().get(loc) == null) locs.add(loc);
        }
        return locs;
    }
}
/**
 * If the crab critter doesn't move, it randomly turns left or right.
 */
public void makeMove(Location loc)
{
    if (loc.equals(getLocation()))
    {
        double r = Math.random();
        int angle;
        if (r < 0.5)
            angle = Location.LEFT;
        else
            angle = Location.RIGHT;
        setDirection(getDirection() + angle);
    }
    else
        super.makeMove(loc);
}

/**
 * Finds the valid adjacent locations of this critter in different 
 * directions.
 * @param directions - an array of directions (which are relative to the 
 * current direction)
 * @return a set of valid locations that are neighbors of the current 
 * location in the given directions
 */
public ArrayList<Location> getLocationsInDirections(int[] directions)
{
    ArrayList<Location> locs = new ArrayList<Location>();
    Grid gr = getGrid();
    Location loc = getLocation();

    for (int d : directions)
    {
        Location neighborLoc = loc.getAdjacentLocation(getDirection() + d);
        if (gr.isValid(neighborLoc))
            locs.add(neighborLoc);
    }
    return locs;
}
GridWorld Case Study

Part 5: Grid Data Structures (AP® CS AB only)

The AbstractGrid Class

Two concrete implementations of the Grid interface are provided, one for a bounded grid that has a fixed number of rows and columns, and a second for an unbounded grid, for which any row and column values are valid. Rather than have repeated code in two classes, the AbstractGrid class defines five methods of the Grid interface that are common to both implementations.

```java
public ArrayList<E> getNeighbors(Location loc)
public ArrayList<Location> getValidAdjacentLocations(Location loc)
public ArrayList<Location> getEmptyAdjacentLocations(Location loc)
public ArrayList<Location> getOccupiedAdjacentLocations(Location loc)
public String toString()
```

The code for these methods uses other methods specified in the interface. Here is the definition of the getNeighbors method. Note that the same method works for bounded and unbounded grids.

```java
public ArrayList<E> getNeighbors(Location loc)
{
    ArrayList<E> neighbors = new ArrayList<E>();
    for (Location neighborLoc : getOccupiedAdjacentLocations(loc))
        neighbors.add(get(neighborLoc));
    return neighbors;
}
```

The AbstractGrid class is a superclass of the BoundedGrid and UnboundedGrid classes. Since AbstractGrid does not define all methods specified by the Grid interface, it is an abstract class. The concrete BoundedGrid and UnboundedGrid classes define the methods of the Grid interface that are not defined in AbstractGrid. This design is illustrated in the following figure.
Do You Know?
Set 10

The source code for the AbstractGrid class is in Appendix D.

1. Where is the isValid method specified? Which classes provide an implementation of this method?
2. Which AbstractGrid methods call the isValid method? Why don’t the other methods need to call it?
3. Which methods of the Grid interface are called in the getNeighbors method? Which classes provide implementations of these methods?
4. Why must the get method, which returns an object of type E, be used in the getEmptyAdjacentLocations method when this method returns locations, not objects of type E?
5. What would be the effect of replacing the constant Location.HALF_RIGHT with Location.RIGHT in the two places where it occurs in the getValidAdjacentLocations method?
The BoundedGrid Class

A bounded grid has a fixed number of rows and columns. You can access only locations that are within the bounds of the grid. If you try to access an invalid location, a run-time exception will be thrown.

The BoundedGrid<E> class stores grid occupants in a two-dimensional array.

```java
private Object[][] occupantArray;
```

Note that occupantArray is declared to hold references of type Object rather than the generic type E. (In the Java language, it is impossible to declare arrays of generic types.) Nevertheless, all elements of occupantArray must belong to the type E. Only the put method adds elements to the array, and it requires elements of type E.

Do You Know?

Set 11

The source code for the BoundedGrid class is in Appendix D.

1. What ensures that a grid has at least one valid location?
2. How is the number of columns in the grid determined by the getNumCols method? What assumption about the grid makes this possible?
3. What are the requirements for a Location to be valid in a BoundedGrid?

In the next four questions, let r = number of rows, c = number of columns, and n = number of occupied locations.

4. What type is returned by the getOccupiedLocations method? What is the time complexity (Big-Oh) for this method?
5. What type is returned by the get method? What parameter is needed? What is the time complexity (Big-Oh) for this method?
6. What conditions may cause an exception to be thrown by the put method? What is the time complexity (Big-Oh) for this method?
7. What type is returned by the remove method? What happens when an attempt is made to remove an item from an empty location? What is the time complexity (Big-Oh) for this method?

8. Based on the answers to questions 4, 5, 6, and 7, would you consider this an efficient implementation? Justify your answer.

The UnboundedGrid Class

In an unbounded grid, any location is valid, even when the row or column is negative or very large. Since there is no bound on the row or column of a location in this grid, the UnboundedGrid<E> class does not use a fixed size two-dimensional array. Instead, it stores occupants in a Map<Location, E>. The key type of the map is Location and the value type is E, the type of the grid occupants.

The numRows and numCols methods both return -1 to indicate that an unbounded grid does not have any specific number of rows or columns. The isValid method always returns true. The get, put, and remove methods simply invoke the corresponding Map methods. The getOccupiedLocations method returns the same locations that are contained in the key set for the map.

Do You Know?

Set 12

The source code for the UnboundedGrid class is in Appendix D.

1. Which method must the Location class implement so that an instance of HashMap can be used for the map? What would be required of the Location class if a TreeMap were used instead? Does Location satisfy these requirements?
2. Why are the checks for null included in the get, put, and remove methods? Why are no such checks included in the corresponding methods for the BoundedGrid?
3. What is the average time complexity (Big-Oh) for the three methods: get, put, and remove? What would it be if a TreeMap were used instead of a HashMap?
4. How would the behavior of this class differ, aside from time complexity, if a TreeMap were used instead of a HashMap?
5. Could a map implementation be used for a bounded grid? What advantage, if any, would the two-dimensional array implementation that is used by the BoundedGrid class have over a map implementation?
Exercises

1. Suppose that a program requires a very large bounded grid that contains very few objects and that the program frequently calls the `getOccupiedLocations` method (as, for example, `ActorWorld`). Create a class `SparseBoundedGrid` that uses a “sparse array” implementation. Your solution need not be a generic class; you may simply store occupants of type `Object`.

The “sparse array” is an array list of linked lists. Each linked list entry holds both a grid occupant and a column index. Each entry in the array list is a linked list or is `null` if that row is empty.

You may choose to implement the linked list in one of two ways. You can use raw list nodes.

```java
public class SparseGridNode
{
    private Object occupant;
    private int col;
    private SparseGridNode next;
    ... 
}
```

Or you can use a `LinkedList<OccupantInCol>` with a helper class.

```java
public class OccupantInCol
{
    private Object occupant;
    private int col;
    ... 
}
```

For a grid with \( r \) rows and \( c \) columns, the sparse array has length \( r \). Each of the linked lists has maximum length \( c \).

Implement the methods specified by the `Grid` interface using this data structure. Why is this a more time-efficient implementation than `BoundedGrid`?
The `World` has a public `addGridClass` method. Since the `ActorWorld` is a `World`, you can call this method in a runner. Here is the code to add a new grid to the GUI.

```java
import info.gridworld.actor.Actor;
import info.gridworld.actor.ActorWorld;
import info.gridworld.grid.Location;
import info.gridworld.actor.Critter;
import info.gridworld.actor.Rock;
import info.gridworld.actor.Flower;

/**
 * This class runs a world with additional grid choices.
 */
public class SparseGridRunner
{
    public static void main(String[] args)
    {
        ActorWorld world = new ActorWorld();
        world.addGridClass("SparseBoundedGrid");
        world.addGridClass("SparseBoundedGrid2");
        world.addGridClass("SparseBoundedGrid3");
        world.addGridClass("UnboundedGrid2");
        world.add(new Location(2, 2), new Critter());
        world.show();
    }
}
```

When you execute a runner class and choose the World menu->set grid, the new grid type will be available for you to choose.
2. Consider using a `HashMap` or `TreeMap` to implement the `SparseBoundedGrid`. How could you use the `UnboundedGrid` class to accomplish this task? Which methods of `UnboundedGrid` could be used without change?

Fill in the following chart to compare the expected Big-Oh efficiencies for each implementation of the `SparseBoundedGrid`.

Let \( r \) = number of rows, \( c \) = number of columns, and \( n \) = number of occupied locations

<table>
<thead>
<tr>
<th>Methods</th>
<th>SparseGridNode version</th>
<th>LinkedList&lt;OccupantInCol&gt; version</th>
<th>HashMap version</th>
<th>TreeMap version</th>
</tr>
</thead>
<tbody>
<tr>
<td>getNeighbors</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getEmptyAdjacentLocations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getOccupiedAdjacentLocations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getOccupiedLocations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>get</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>put</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. Consider an implementation of an unbounded grid in which all valid locations have non-negative row and column values. The constructor allocates a 16 x 16 array. When a call is made to the `put` method with a row or column index that is outside the current array bounds, double both array bounds until they are large enough, construct a new square array with those bounds, and place the existing occupants into the new array.

Implement the methods specified by the `Grid` interface using this data structure. What is the Big-Oh efficiency of the `get` method? What is the efficiency of the `put` method when the row and column index values are within the current array bounds? What is the efficiency when the array needs to be resized?