Quick Reference
AP Computer Science A and AB

Content of Appendixes

Class Summary and Index

Class* **Package**	**Description**	**Page**	**Tested in AP CS Exam?**
AbstractGrid info.gridworld.grid	AbstractGrid contains the methods that are common to grid implementations.	D - 1, D - 2	Implementation (AB only)
Actor info.gridworld.actor	An Actor is an entity that can act. It has a color and direction.	B - 3	API only
ActorWorld info.gridworld.actor	An ActorWorld is occupied by actors.	Student Manual Part 3	Not tested
BoundedGrid info.gridworld.grid	A BoundedGrid is a rectangular grid with a finite number of rows and columns.	D - 3, D – 4	Implementation (AB only)
BoxBug (none)	A BoxBug traces out a square "box" of a given size.	C - 3	Implementation
BoxBugRunner (none)	This class runs a world that contains box bugs.	Student Manual Part 2	Not tested
Bug info.gridworld.actor	A Bug is an actor that can move and turn. It drops flowers as it moves.	C - 1, C - 2	Implementation
BugRunner (none)	This class runs a world that contains a bug and a rock, added at random locations.	In the projects/ firstProject folder of the code distribution	Not tested
ChameleonCritter (none)	A ChameleonCritter takes on the color of neighboring actors as it moves through the grid.	C - 6	Implementation
ChameleonRunner (none)	This class runs a world that contains chameleon critters.	In the projects/ critters folder of the code distribution	Not tested

***Bold formatting** of a class name indicates that students are responsible for the use of that class on the AP Computer Science Exams at the level indicated in this chart.

Class* ---------- **Package**	**Description**	**Page**	**Tested in AP CS Exam?**
`CrabCritter` ---------- (none)	A `CrabCritter` looks at a limited set of neighbors when it eats and moves.	Student Manual Part 4	Not tested
`CrabRunner` ---------- (none)	This class runs a world that contains crab critters.	In the `projects/critters` folder of the code distribution	Not tested
`Critter` ---------- `info.gridworld.actor`	A `Critter` is an actor that moves through its world, processing other actors in some way and then picking a new location.	C - 4, C – 5, C - 6	Implementation
`CritterRunner` ---------- (none)	This class runs a world that contains critters.	In the `projects/critters` folder of the code distribution	Not tested
`Flower` ---------- `info.gridworld.actor`	A `Flower` is an actor that darkens over time. Some actors drop flowers as they move.	B - 4	API only
`Grid` ---------- `info.gridworld.grid`	`Grid` provides an interface for a two-dimensional, grid-like environment containing arbitrary objects.	B - 2	API only
`Location` ---------- `info.gridworld.grid`	A `Location` object represents the row and column of a location in a two-dimensional grid.	B - 1	API only
`Rock` ---------- `info.gridworld.actor`	A `Rock` is an actor that does nothing. It is commonly used to block other actors from moving.	B - 4	API only
`UnboundedGrid` ---------- `info.gridworld.grid`	An `UnboundedGrid` is a rectangular grid with an unbounded number of rows and columns.	D - 5, D - 6	Implementation (AB only)

`World`
 ---------- 	
 `info.gridworld.world`
	A `World` is the mediator between a grid and the GridWorld GUI.	In the API documentation of the code distribution	Not tested

***Bold formatting** of a class name indicates that students are responsible for the use of that class on the AP Computer Science Exams at the level indicated in this chart.

Appendix B — Testable API

`info.gridworld.grid.Location` class

`public Location(int r, int c)`
constructs a location with row `r` and column `c`

`public int getRow()`
returns the row of this location

`public int getCol()`
returns the column of this location

`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`

`public boolean equals(Object other)`
returns `true` if `other` is a `Location` object with the same row and column values as this location, and `false` otherwise

`public int hashCode()`
returns a hash code for this location

`public int compareTo(Object otherObject)`
returns a negative integer if this location is less than `other`, zero if the two locations are equal, or a positive integer if this location is greater than `other`. Locations are ordered in row-major order.
Precondition: `other` is a `Location` object.

`public String toString()`
returns a string with the row and column of this location, in the format (row, col)

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;
```

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;
```

`info.gridworld.grid.Grid<E>` interface

`int getNumRows()`
returns the number of rows, or -1 if this grid is unbounded

`int getNumCols()`
returns the number of columns, or -1 if this grid is unbounded

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

`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

`ArrayList<Location> getOccupiedLocations()`
returns all occupied locations in this grid

`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

`info.gridworld.actor.Actor` class

`public Actor()`
constructs a blue actor that is facing north

`public Color getColor()`
returns the color of this actor

`public void setColor(Color newColor)`
sets the color of this actor to `newColor`

`public int getDirection()`
returns the direction of this actor, an angle between 0 and 359 degrees

`public void setDirection(int newDirection)`
sets the direction of this actor to the angle between 0 and 359 degrees that is equivalent to `newDirection`

`public Grid<Actor> getGrid()`
returns the grid of this actor, or `null` if this actor is not contained in a grid

`public Location getLocation()`
returns the location of this actor.
Precondition: this actor is contained in a grid

`public void putSelfInGrid(Grid<Actor> gr, Location loc)`
puts this actor into the location `loc` of the grid `gr`. If there is another actor at `loc`, it is removed.
Precondition: (1) This actor is not contained in a grid (2) `loc` is valid in `gr`

`public void removeSelfFromGrid()`
removes this actor from its grid.
Precondition: this actor is contained in a grid

`public void moveTo(Location newLocation)`
moves this actor to `newLocation`. If there is another actor at `newLocation`, it is removed.
Precondition: (1) This actor is contained in a grid (2) `newLocation` is valid in the grid of this actor

`public void act()`
reverses the direction of this actor. Override this method in subclasses of `Actor` to define types of actors with different behavior

`public String toString()`
returns a string with the location, direction, and color of this actor

`info.gridworld.actor.Rock` class

`public Rock()`
constructs a black rock

`public Rock(Color rockColor)`
constructs a rock with color `rockColor`

`public void act()`
overrides the `act` method in the `Actor` class to do nothing

`info.gridworld.actor.Flower` class

`public Flower()`
constructs a pink flower

`public Flower(Color initialColor)`
constructs a flower with initial color `initialColor`

`public void act()`
causes the color of this flower to darken

Appendix C — Testable Code for APCS A/AB

Bug.java

```
package info.gridworld.actor;

import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;

import java.awt.Color;

/**
 * A Bug is an actor that can move and turn. It drops flowers as
 * it moves.
 * The implementation of this class is testable on the AP CS A and AB Exams.
 */
public class Bug extends Actor
{
  /**
   * Constructs a red bug.
   */
  public Bug()
  {
    setColor(Color.RED);
  }

  /**
   * Constructs a bug of a given color.
   * @param bugColor the color for this bug
   */
  public Bug(Color bugColor)
  {
    setColor(bugColor);
  }

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

  /**
   * Turns the bug 45 degrees to the right without changing its location.
   */
  public void turn()
  {
    setDirection(getDirection() + Location.HALF_RIGHT);
  }
```

```
  /**
   * Moves the bug forward, putting a flower into the location it previously
   * occupied.
   */
  public void move()
  {
    Grid<Actor> gr = getGrid();
    if (gr == null)
      return;
    Location loc = getLocation();
    Location next = loc.getAdjacentLocation(getDirection());
    if (gr.isValid(next))
      moveTo(next);
    else
      removeSelfFromGrid();
    Flower flower = new Flower(getColor());
    flower.putSelfInGrid(gr, loc);
  }

  /**
   * Tests whether this bug can move forward into a location that is empty or
   * contains a flower.
   * @return true if this bug can move.
   */
  public boolean canMove()
  {
    Grid<Actor> gr = getGrid();
    if (gr == null)
      return false;
    Location loc = getLocation();
    Location next = loc.getAdjacentLocation(getDirection());
    if (!gr.isValid(next))
      return false;
    Actor neighbor = gr.get(next);
    return (neighbor == null) || (neighbor instanceof Flower);
    // ok to move into empty location or onto flower
    // not ok to move onto any other actor
  }
}
```

BoxBug.java

```
package info.gridworld.actor;

import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;

import java.awt.Color;

/**
 * A BoxBug traces out a square "box" of a given size.
 * The implementation of this class is testable on the AP CS A and AB Exams.
 */
public class BoxBug extends Bug
{
  private int steps;
  private int sideLength;

  /**
   * Constructs a box bug that traces a square of a given side length
   * @param length the side length
   */
  public BoxBug(int length)
  {
    steps = 0;
    sideLength = length;
  }

  public void act()
  {
    if (steps < sideLength && canMove())
    {
      move();
      steps++;
    }
    else
    {
      turn();
      turn();
      steps = 0;
    }
  }
}
```

Critter.java

```
package info.gridworld.actor;

import info.gridworld.grid.Location;
import java.util.ArrayList;

/**
 * A Critter is an actor that moves through its world, processing
 * other actors in some way and then picking a new location.
 * The implementation of this class is testable on the AP CS A and AB Exams.
 */
public class Critter extends Actor
{
  /**
   * A critter acts by getting a list of its neighbors, processing them,
   * getting locations to move to, selecting one of them, and moving to the
   * selected location.
   */
  public void act()
  {
    if (getGrid() == null)
      return;
    ArrayList<Actor> actors = getActors();
    processActors(actors);
    ArrayList<Location> moveLocs = getMoveLocations();
    Location loc = selectMoveLocation(moveLocs);
    makeMove(loc);
  }

  /**
   * Gets the actors for processing. The actors must be contained in the same
   * grid as this critter. Implemented to return the actors that occupy
   * neighboring grid locations. Override this method in subclasses to look
   * elsewhere for actors to process.
   * @return a list of actors that are neighbors of this critter
   */
  public ArrayList<Actor> getActors()
  {
    return getGrid().getNeighbors(getLocation());
  }
```

```
/**
 * Processes the actors. Implemented to "eat" (i.e. remove) all actors that
 * are not rocks or critters. Override this method in subclasses to process
 * neighbors in a different way.
 * Precondition: All objects in actors are contained in the
 * same grid as this critter.
 * @param actors the actors to be processed
 */
public void processActors(ArrayList<Actor> actors)
{
  for (Actor a : actors)
  {
    if (!(a instanceof Rock) && !(a instanceof Critter))
      a.removeSelfFromGrid();
  }
}

/**
 * Gets the possible locations for the next move. Implemented to return the
 * empty neighboring locations. Override this method in subclasses to look
 * elsewhere for move locations.
 * Postcondition: The locations must be valid in the grid of this critter.
 * @return a list of possible locations for the next move
 */
public ArrayList<Location> getMoveLocations()
{
  return getGrid().getEmptyAdjacentLocations(getLocation());
}

/**
 * Selects the location for the next move. Implemented to randomly pick one
 * of the possible locations, or to return the current location if locs has
 * size 0. Override this method in subclasses that have another mechanism
 * for selecting the next move location.
 * Precondition: All locations in locs are valid in the grid
 * of this critter
 * @param locs the possible locations for the next move
 * @return the location that was selected for the next move.
 */
public Location selectMoveLocation(ArrayList<Location> locs)
{
  int n = locs.size();
  if (n == 0)
    return getLocation();
  int r = (int) (Math.random()  * n);
  return locs.get(r);
}
```

```
  /**
   * Moves this critter to the given location. Implemented to call moveTo.
   * Override this method in subclasses that want to carry out other actions
   * for moving (for Example, turning or leaving traces).
   * Precondition: loc is valid in the grid of this critter
   * @param loc the location to move to (must be valid)
   */
  public void makeMove(Location loc)
  {
    moveTo(loc);
  }
}
```

ChameleonCritter.java

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

import java.util.ArrayList;

/**
 * A ChameleonCritter takes on the color of neighboring actors as
 * it moves through the grid.
 * The implementation of this class is testable on the AP CS A and AB Exams.
 */
public class ChameleonCritter extends Critter
{
  /**
   * Randomly selects a neighbor and changes this critter's color to be the
   * same as that neighbor's. If there are no neighbors, no action is taken.
   */
  public void processActors(ArrayList<Actor> actors)
  {
    int n = actors.size();
    if (n == 0)
      return;
    int r = (int) (Math.random()  * n);

    Actor other = actors.get(r);
    setColor(other.getColor());
  }

  /**
   * Turns towards the new location as it moves.
   */
  public void makeMove(Location loc)
  {
    setDirection(getLocation().getDirectionToward(loc));
    super.makeMove(loc);
  }
}
```

Appendix D — Testable Code for APCS AB

AbstractGrid.java

```
package info.gridworld.grid;

import java.util.ArrayList;

/**
 * AbstractGrid contains the methods that are common to grid
 * implementations.
 * The implementation of this class is testable on the AP CS AB Exam.
 */
public abstract class AbstractGrid<E> implements Grid<E>
{

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

  public ArrayList<Location> getValidAdjacentLocations(Location loc)
  {
    ArrayList<Location> locs = new ArrayList<Location>();

    int d = Location.NORTH;
    for (int i = 0; i < Location.FULL_CIRCLE / Location.HALF_RIGHT; i++)
    {
      Location neighborLoc = loc.getAdjacentLocation(d);
      if (isValid(neighborLoc))
        locs.add(neighborLoc);
      d = d + Location.HALF_RIGHT;
    }
    return locs;
  }

  public ArrayList<Location> getEmptyAdjacentLocations(Location loc)
  {
    ArrayList<Location> locs = new ArrayList<Location>();
    for (Location neighborLoc : getValidAdjacentLocations(loc))
    {
      if (get(neighborLoc) == null)
        locs.add(neighborLoc);
    }
    return locs;
  }
```

```
  public ArrayList<Location> getOccupiedAdjacentLocations(Location loc)
  {
    ArrayList<Location> locs = new ArrayList<Location>();
    for (Location neighborLoc : getValidAdjacentLocations(loc))
    {
      if (get(neighborLoc) != null)
        locs.add(neighborLoc);
    }
    return locs;
  }

  /**
   * Creates a string that describes this grid.
   * @return a string with descriptions of all objects in this grid (not
   * necessarily in any particular order), in the format {loc=obj, loc=obj, ...}
   */
  public String toString()
  {
    String s = "{";
    for (Location loc : getOccupiedLocations())
    {
      if (s.length() > 1)
        s += ", ";
      s += loc + "=" + get(loc);
    }
    return s + "}";
  }
}
```

BoundedGrid.java

```
package info.gridworld.grid;

import java.util.ArrayList;

/**
 * A BoundedGrid is a rectangular grid with a finite number of
 * rows and columns.
 * The implementation of this class is testable on the AP CS AB Exam.
 */
public class BoundedGrid<E> extends AbstractGrid<E>
{
  private Object[][] occupantArray; // the array storing the grid elements

  /**
   * Constructs an empty bounded grid with the given dimensions.
   * (Precondition: rows > 0 and cols > 0.)
   * @param rows number of rows in BoundedGrid
   * @param cols number of columns in BoundedGrid
   */

  public BoundedGrid(int rows, int cols)
  {
    if (rows <= 0)
      throw new IllegalArgumentException("rows <= 0");
    if (cols <= 0)
      throw new IllegalArgumentException("cols <= 0");
    occupantArray = new Object[rows][cols];
  }

  public int getNumRows()
  {
    return occupantArray.length;
  }

  public int getNumCols()
  {
    // Note: according to the constructor precondition, numRows() > 0, so
    // theGrid[0] is non-null.
    return occupantArray[0].length;
  }

  public boolean isValid(Location loc)
  {
    return 0 <= loc.getRow() && loc.getRow() < getNumRows() &&
                        0 <= loc.getCol() && loc.getCol() < getNumCols();
  }
```

```
  public ArrayList<Location> getOccupiedLocations()
  {
    ArrayList<Location> theLocations = new ArrayList<Location>();

    // Look at all grid locations.
    for (int r = 0; r < getNumRows(); r++)
    {
      for (int c = 0; c < getNumCols(); c++)
      {
        // If there's an object at this location, put it in the array.
        Location loc = new Location(r, c);
        if (get(loc) != null)
          theLocations.add(loc);
      }
    }

    return theLocations;
  }

  public E get(Location loc)
  {
    if (!isValid(loc))
      throw new IllegalArgumentException("Location " + loc + " is not valid");
    return (E) occupantArray[loc.getRow()][loc.getCol()]; // unavoidable warning
  }

  public E put(Location loc, E obj)
  {
    if (!isValid(loc))
      throw new IllegalArgumentException("Location " + loc+ " is not valid");
    if (obj == null)
      throw new NullPointerException("obj == null");

    // Add the object to the grid.
    E oldOccupant = get(loc);
    occupantArray[loc.getRow()][loc.getCol()] = obj;
    return oldOccupant;
  }

  public E remove(Location loc)
  {
    if (!isValid(loc))
      throw new IllegalArgumentException("Location " + loc + " is not valid");

    // Remove the object from the grid.
    occupantArray[loc.getRow()][loc.getCol()] = null;
    return r;
  }
}
```

UnboundedGrid.java

```
package info.gridworld.grid;

import java.util.ArrayList;

import java.util.*;

/**
 * An UnboundedGrid is a rectangular grid with an unbounded number of rows and
 * columns.
 * The implementation of this class is testable on the AP CS AB Exam.
 */
public class UnboundedGrid<E> extends AbstractGrid<E>
{
  private Map<Location, E> occupantMap;

  /**
   * Constructs an empty unbounded grid.
   */
  public UnboundedGrid()
  {
    occupantMap = new HashMap<Location, E>();
  }

  public int getNumRows()
  {
    return -1;
  }

  public int getNumCols()
  {
    return -1;
  }

  public boolean isValid(Location loc)
  {
    return true;
  }

  public ArrayList<Location> getOccupiedLocations()
  {
    ArrayList<Location> a = new ArrayList<Location>();
    for (Location loc : occupantMap.keySet())
      a.add(loc);
    return a;
  }

  public E get(Location loc)
  {
    if (loc == null)
      throw new NullPointerException("loc == null");
    return occupantMap.get(loc);
  }
```

```
  public E put(Location loc, E obj)
  {
    if (loc == null)
      throw new NullPointerException("loc == null");
    if (obj == null)
      throw new NullPointerException("obj == null");
    return occupantMap.put(loc, obj);
  }

  public E remove(Location loc)
  {
    if (loc == null)
      throw new NullPointerException("loc == null");
    return occupantMap.remove(loc);
  }
}
```

Quick Reference A/AB

Location Class (implements Comparable)

```
public Location(int r, int c)
public int getRow()
public int getCol()
public Location getAdjacentLocation(int direction)
public int getDirectionToward(Location target)
public boolean equals(Object other)
public int hashCode()
public int compareTo(Object otherObject)
public String toString()

NORTH, EAST, SOUTH, WEST, NORTHEAST, SOUTHEAST, NORTHWEST, SOUTHWEST
LEFT, RIGHT, HALF_LEFT, HALF_RIGHT, FULL_CIRCLE, HALF_CIRCLE, AHEAD
```

Grid<E> Interface

```
int getNumRows()
int getNumCols()
boolean isValid(Location loc)
E put(Location loc, E obj)
E remove(Location loc)
E get(Location loc)
ArrayList<Location> getOccupiedLocations()
ArrayList<Location> getValidAdjacentLocations(Location loc)
ArrayList<Location> getEmptyAdjacentLocations(Location loc)
ArrayList<Location> getOccupiedAdjacentLocations(Location loc)
ArrayList<E> getNeighbors(Location loc)
```

Actor Class

```
public Actor()
public Color getColor()
public void setColor(Color newColor)
public int getDirection()
public void setDirection(int newDirection)
public Grid<Actor> getGrid()
public Location getLocation()
public void putSelfInGrid(Grid<Actor> gr, Location loc)
public void removeSelfFromGrid()
public void moveTo(Location newLocation)
public void act()
public String toString()
```

Rock Class (extends Actor)

```
public Rock()
public Rock(Color rockColor)
public void act()
```

Flower Class (extends Actor)

```
public Flower()
public Flower(Color initialColor)
public void act()
```

Bug Class (extends Actor)

```
public Bug()
public Bug(Color bugColor)
public void act()
public void turn()
public void move()
public boolean canMove()
```

BoxBug Class (extends Bug)

```
public BoxBug(int n)
public void act()
```

Critter Class (extends Actor)

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

ChameleonCritter Class (extends Critter)

```
public void processActors(ArrayList<Actor> actors)
public void makeMove(Location loc)
```

Quick Reference AB Only

AbstractGrid Class (implements Grid)

```
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()
```

BoundedGrid Class (extends AbstractGrid)

```
public BoundedGrid(int rows, int cols) / public UnboundedGrid()
public int getNumRows()
public int getNumCols()
public boolean isValid(Location loc)
public ArrayList<Location> getOccupiedLocations()
public E get(Location loc)
public E put(Location loc, E obj)
public E remove(Location loc)
```

UnboundedGrid Class (extends AbstractGrid)

```
public UnboundedGrid()
public int getNumRows()
public int getNumCols()
public boolean isValid(Location loc)
public ArrayList<Location> getOccupiedLocations()
public E get(Location loc)
public E put(Location loc, E obj)
public E remove(Location loc)
```