Quick Reference
AP® Computer Science AB

Content of Appendixes

Appendix A — AB Exam Java Quick Reference

Accessible Methods from the Java Library That May Be Included on the Exam

class java.lang.Object
- boolean equals(Object other)
- String toString()
- int hashCode()

interface java.lang.Comparable[*]
- int compareTo(Object other) // returns a value < 0 if this is less than other
                              // returns a value = 0 if this is equal to other
                              // returns a value > 0 if this is greater than other

class java.lang.Integer implements java.lang.Comparable[*]
- Integer(int value)
- int intValue()

class java.lang.Double implements java.lang.Comparable[*]
- Double(double value)
- double doubleValue()

class java.lang.String implements java.lang.Comparable[*]
- int length()
- String substring(int from, int to) // returns the substring beginning at from
                                     // and ending at to-1
- String substring(int from)   // returns substring(from, length())
- int indexOf(String str)   // returns the index of the first occurrence of str;
                                     // returns -1 if not found

class java.lang.Math
- static int abs(int x)
- static double abs(double x)
- static double pow(double base, double exponent)
- static double sqrt(double x)
- static double random()   // returns a double in the range [0.0, 1.0)

[*]The AP Java subset uses the "raw" Comparable interface, not the generic Comparable<T> interface.

interface java.util.List<E>
- int size()
- boolean add(E obj)   // appends obj to end of list; returns true
- void add(int index, E obj)   // inserts obj at position index (0 ≤ index ≤ size),
                                        // moving elements at position index and higher
                                        // to the right (adds 1 to their indices) and adjusts size
- E get(int index)
- E set(int index, E obj)   // replaces the element at position index with obj
                                        // returns the element formerly at the specified position
- E remove(int index)   // removes element from position index, moving elements
                                        // at position index + 1 and higher to the left
                                        // (subtracts 1 from their indices) and adjusts size
                                        // returns the element formerly at the specified position
- Iterator<E> iterator()
- ListIterator<E> listIterator()

class java.util.ArrayList<E> implements java.util.List<E>

class java.util.LinkedList<E> implements java.util.List<E>, java.util.Queue<E>
- void addFirst(E obj)
- void addLast(E obj)
- E getFirst()
- E getLast()
- E removeFirst()
- E removeLast()

interface java.util.Queue<E>   // implemented by LinkedList<E>
- boolean add(E obj)   // enqueues obj at the end of the queue; returns true
- E remove()   // dequeues and returns the element at the front of the queue
- E peek()   // returns the element at the front of the queue;
                                        // null if the queue is empty
- boolean isEmpty()

class java.util.PriorityQueue<E>   // E should implement Comparable[*]
- boolean add(E obj)   // adds obj to the priority queue; returns true
- E remove()   // removes and returns the minimal element from the priority queue
- E peek()   // returns the minimal element from the priority queue;
                                        // null if the priority queue is empty
- boolean isEmpty()

class java.util.Stack<E>
- E push(E item)   // pushes item onto the top of the stack; returns item
- E pop()   // removes and returns the element at the top of the stack
- E peek()   // returns the element at the top of the stack;
                                        // throws an exception if the stack is empty
- boolean isEmpty()

[*] The AP Java subset uses the "raw" Comparable interface, not the generic Comparable<T> interface.

interface java.util.Iterator\<E\>
- boolean hasNext()
- E next()
- void remove()   // removes the last element that was returned by next

interface java.util.ListIterator\<E\> extends java.util.Iterator\<E\>
- void add(E obj)   // adds obj before the element that will be returned by next
- void set(E obj)   // replaces the last element returned by next with obj

interface java.util.Set\<E\>
- int size()
- boolean contains(Object obj)
- boolean add(E obj)   // if obj is not present in this set, adds obj and returns true;
                                             // otherwise, returns false
- boolean remove(Object obj)   // if obj is present in this set, removes obj and returns true;
                                             // otherwise, returns false
- Iterator\<E\> iterator()

class java.util.HashSet\<E\> implements java.util.Set\<E\>
class java.util.TreeSet\<E\> implements java.util.Set\<E\>

interface java.util.Map\<K,V\>
- int size()
- boolean containsKey(Object key)
- V put(K key, V value)   // associates key with value
                                             // returns the value formerly associated with key
                                             // or null if key was not present
- V get(Object key)   // returns the value associated with key
                                             // or null if there is no associated value
- V remove(Object key)   // removes and returns the value associated with key;
                                             // returns null if there is no associated value
- Set\<K\> keySet()

class java.util.HashMap\<K,V\> implements java.util.Map\<K,V\>
class java.util.TreeMap\<K,V\> implements java.util.Map\<K,V\>

Implementation classes for linked list and tree nodes

Unless otherwise noted, assume that a linked list implemented from the `ListNode` class
does not have a dummy header node.

```java
public class ListNode
{
  private Object value;
  private ListNode next;

  public ListNode(Object initValue, ListNode initNext)
    { value = initValue; next = initNext; }

  public Object getValue() { return value; }
  public ListNode getNext() { return next; }

  public void setValue(Object theNewValue) { value = theNewValue; }
  public void setNext(ListNode theNewNext) { next = theNewNext; }
}
```

Unless otherwise noted, assume that a tree implemented from the `TreeNode` class
does not have a dummy root node.

```java
public class TreeNode
{
  private Object value;
  private TreeNode left;
  private TreeNode right;

  public TreeNode(Object initValue)
    { value = initValue; left = null; right = null; }

  public TreeNode(Object initValue, TreeNode initLeft, TreeNode initRight)
    { value = initValue; left = initLeft; right = initRight; }

  public Object getValue() { return value; }
  public TreeNode getLeft() { return left; }
  public TreeNode getRight() { return right; }

  public void setValue(Object theNewValue) { value = theNewValue; }
  public void setLeft(TreeNode theNewLeft) { left = theNewLeft; }
  public void setRight(TreeNode theNewRight) { right = theNewRight; }
}
```

Appendix B — Testable API

info.gridworld.grid.Location class (implements Comparable)

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

```
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 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` with the same row and column as this location; `false` otherwise

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

```
public int compareTo(Object other)
```
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` from this grid and returns the object that was removed (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 an array list of all occupied locations in this grid

```
ArrayList<Location> getValidAdjacentLocations(Location loc)
```
      returns an array list of the valid locations adjacent to `loc` in this grid
      Precondition: `loc` is valid in this grid

```
ArrayList<Location> getEmptyAdjacentLocations(Location loc)
```
      returns an array list of the valid empty locations adjacent to `loc` in this grid
      Precondition: `loc` is valid in this grid

```
ArrayList<Location> getOccupiedAdjacentLocations(Location loc)
```
      returns an array list of the valid occupied locations adjacent to `loc` in this grid
      Precondition: `loc` is valid in this grid

```
ArrayList<E> getNeighbors(Location loc)
```
      returns an array list of the 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, or `null` if this actor is not contained in a grid

```
public void putSelfInGrid(Grid<Actor> gr, Location loc)
```
puts this actor into location `loc` of 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 (`extends Actor`)

```
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 (`extends Actor`)

```
public Flower()
```
      constructs a pink flower

```
public Flower(Color initialColor)
```
      constructs a flower with 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);
  }


  /**
   *  Moves if it can move, turns otherwise.
   */
  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

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

/**
 * 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;
  }


  /**
   * Moves to the next location of the square.
   */
  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 moving to a new location.

 *  Define your own critters by extending this class and overriding any methods of this class except for  act.
 *  When you override these methods, be sure to preserve the postconditions.
 *  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 other actors, processing that list, 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. Implemented to return the actors that occupy neighboring grid locations.
   *  Override this method in subclasses to look elsewhere for actors to process.
   *  Postcondition: The state of all actors is unchanged.
   *  @return  a list of actors that this critter wishes to process.
   */
  public ArrayList<Actor> getActors()
  {
    return getGrid().getNeighbors(getLocation());
  }
```

```
/**
 * Processes the elements of actors. New actors may be added to empty locations.
 * Implemented to "eat" (i.e., remove) selected actors that are not rocks or critters.
 * Override this method in subclasses to process actors in a different way.
 * Postcondition: (1) The state of all actors in the grid other than this critter and the
 * elements of actors is unchanged. (2) The location of this critter is unchanged.
 * @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 a list of possible locations for the next move. These locations must be valid in the grid of this critter.
 * Implemented to return the empty neighboring locations. Override this method in subclasses to look
 * elsewhere for move locations.
 * Postcondition: The state of all actors is unchanged.
 * @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.
 * 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.
 * @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  loc,  or removes this critter from its grid if  loc  is  null.
     * An actor may be added to the old location. If there is a different actor at location  loc, that actor is
     * removed from the grid. Override this method in subclasses that want to carry out other actions
     * (for example, turning this critter or adding an occupant in its previous location).
     * Postcondition: (1) getLocation() == loc.
     * (2) The state of all actors other than those at the old and new locations is unchanged.
     * @param loc  the location to move to
     */
    public void makeMove(Location loc)
    {
      if (loc == null)
        removeSelfFromGrid();
      else
        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

```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`

```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.
    E r = get(loc);
    occupantArray[loc.getRow()][loc.getCol()] = null;
    return r;
  }
}
```

UnboundedGrid.java

```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 other)
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 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)
```

Appendix G: Index for Source Code

This appendix provides an index for the Java source code found in Appendix C and Appendix D.

Bug.java

BoxBug.java

Critter.java

ChameleonCritter.java

AbstractGrid.java

`getNeighbors(Location loc)`	**D1**
`getValidAdjacentLocations(Location loc)`	**D1**
`getEmptyAdjacentLocations(Location loc)`	**D1**
`getOccupiedAdjacentLocations(Location loc)`	**D2**
`toString()`	**D2**

BoundedGrid.java

`BoundedGrid(int rows, int cols)`	**D3**
`getNumRows()`	**D3**
`getNumCols()`	**D3**
`isValid(Location loc)`	**D3**
`getOccupiedLocations()`	**D4**
`get(Location loc)`	**D4**
`put(Location loc, E obj)`	**D4**
`remove(Location loc)`	**D4**

UnboundedGrid.java

`UnboundedGrid()`	**D5**
`getNumRows()`	**D5**
`getNumCols()`	**D5**
`isValid(Location loc)`	**D5**
`getOccupiedLocations()`	**D5**
`get(Location loc)`	**D5**
`put(Location loc, E obj)`	**D6**
`remove(Location loc)`	**D6**