## **Location, Direction, Environment**

Copyright Chris Nevison, Computer Science Department, Colgate University, Hamilton, NY, 13346, November, 2003.  This material may be reproduced with permission for face-to-face teaching purposes only.   This material may not be reproduced or used for any commercial purpose. This copyright notice must appear on any copies made.

We provide several assignments for working with the Location, Direction and Environment classes. These use three display classes that provide graphical results. We first provide a series of exercises to do simple manipulations of Location and Direction. Then we suggest more extensive programming assignments that make use of these classes. You should first read the description of these classes provided in Unit 1B. Access these from The blackboad Assignments Unit 1 folder.   
Basic Exercises for Location and Direction   
Exercises using Environment   
Programming Assignments

### **1. Introduction**

In this second part of unit 1 we review the three key "black-box" classes that are used extensively in the MBS case study, Location, Direction, Environment. Students are responsible for understanding how to use each of these, but not for knowing anything about their implementation. So the key is for students to become familiar with all the methods for these classes and how to use them with interacting objects. We will examine each in turn, and then consider some ways in which these classes can be used independent of the Marine Biology Simulation itself. These can be a source of assignments of various sorts and can be used for teaching different programming concepts. We also provide some display classes so that these classes can be used with a graphical display.

### **2. Location**

The class Location encapsulates the row and column of a location. A Location object is immutable, meaning that once it has been constructed it cannot be changed, similar to the standard Java classes Integer, Double, and String. Of course, a variable of type Location is a reference to a Location and can be reassigned to a different, possibly new Location. Here is a summary of the constructor and methods for Location.

**class Location implements Comparable**

Constructors  
  
Location(int row, int col)         constructs a Location with the given row and col  
  
Accessor methods

int row()                                           returns the row for this location  
int col()                                           returns the column for this location  
boolean equals(Object other)     returns true if other is a Location with the same row and  
                                                             column as this Location, otherwise returns false.  
int hashcode()                                 returns a hash value for this Location.  
int compareTo(Object other)       comparison is made in row-major order.  
String toString()                           returns a string representation of this Location.

A Location object simply contains integer row and column values that are accessible with the corresponding methods. These are set when the object is created with the constructor and cannot be changed.

The Location class defines the method equals so that we can check whether two locations have the same row and column. The hashCode method is used for advanced data structures and can safely be ignored at this point. For technical reasons, it is good practice to always define hashCode when equals is defined. Students in the AP CS AB course will learn more about this when they work with hash tables.

The class Location implements the standard Java interface Comparable by defining a compareTo method. This means that Location objects can be compared according to row-major order. (A location with a smaller row than another is smaller, if the rows are the same the one with the smaller column is smaller.)

It also overrides the toString method so that Location objects can be printed directly.

### 

### **3. Direction**

The Direction class encapsulates one of the directions north, east, south, west, northeast, southeast, northwest, or southwest. (In fact, any whole degree directiion on the compass, from 0 to 359, can be represented.) Only the first four cardinal directions are used in the MBS case study. Like the class Location, Direction is immutable. Here is a summary of the constructors and methods for Direction.

**class Direction**

Constructors  
  
Direction()                                                                constructs an instance of North  
Direction(int degrees)                                          constructs an instance of a direction at the indicated  
                                                                                      degrees of the compass (0 degrees is North, 90 degrees is  
                                                                                      East,180 degrees is South, 270 degrees is West)  
Direction(String str)                                            constructs an instance of a direction as indicated by str,  
                                                                                      with the following possible values: "North", "East", "West",  
                                                                                      "South", "Northeast", "Northwest", "Southeast",  
                                                                                      "Southwest".  
Accessors

int inDegrees()                                                        returns the number of degrees for this direction  
boolean equals(Object other)                              check if this direction is equal to another  
int hashcode()                                                          returns a hash value for this direction.  
Direction toRight()                                                returns the direction to the right of this one  
Direction toRight(int degrees)                          returns the direction degrees to the right of this one  
Direction toLeft()                                                  returns the direction to the left of this one  
Direction toLeft(int degrees)                            returns the direction degrees to the left of this one  
Direction reverse()                                                returns the direction that is the reverse of this one  
String toString()                                                    returns a string representation of this direction  
  
static Direction randomDirection()                  returns a random direction

Although a Direction object can be constructed using the constructor that takes an integer representing degrees or an appropriate string, the Direction class also provides constants, so that a direction variable can be given a value by assigning it a constant. Since a Direction object is immutable, there is no difficulty about aliasing with this style of coding. Eight constant directions are provided: NORTH, SOUTH, EAST, WEST, NORTHEAST, NORTHWEST, SOUTHEAST, SOUTHWEST.

Although a Direction object is immutable, it can return associated directions with the methods toRight, toLeft, and reverse. The versions of these methods that take degrees are not needed in our applications, since we will work with the four cardinal directions. We will not have occasion to use the method inDegrees, since we can compare a given Direction object either with another or with one of the constants for the cardinal compass directions using the equals method. As with Location, the hashcode method is also defined, although it is not used in the MBS. The toString method returns a string representation of the direction, using "north", northeast", etc for the eight named directions.

The static method randomDirection is used to generate a random direction of any degree value from 0 to 359. Because it is not limited to the cardinal directions, this method is not used in the MBS case study.

This Direction class is more general than is needed for the MBS case study. It allows directions in whole degrees, even though the case study only uses the four cardinal compass directions. (and later variations, in chapter five, use the eight named directions). The determination of which directions are appropriate and the geometry of the model are placed in the Environment classes. Consequently, the inDegrees, randomDirection, and the toRight and toLeft methods that take integer parameters are not used in this case study.

### **4. Environment**

Environment is really an interface rather than a class. It specifies the methods that any class that implements an Environment must have. By making it an interface, the MBS case study can have both bounded and unbounded environments, and possibly other variations. The first part of the case study only uses the class UnboundedEnv. Other implementations are considered in chapter five, which is only required for the AB curriculum. Consequently, we will only discuss the methods as described by the Environment interface and will interpret them in terms of a bounded environment.

The Environment has two types of responsibility. The first is to define the geometry of the environment, the relationship among locations and directions.

The second type of responsibility is to store and manage objects in the environment. Any object stored in an environment must have a Location that is returned by a method location. To enforce this requirement, only objects that implement the interface Locatable can be added to an Environment. This is a good example of the usefulness of interfaces. It allows Environment to be defined in such a way that it can contain objects of many different classes, so long as they satisfy this simple interface. Although in the MBS case study the only object stored in an Environment are Fish, we can use it for quite a variety of other things, which we will explore in this unit.

We will consider the methods specified by the interface Environment in two groups, the geometry and the object management.

#### **4.1 Geometry**

Here is a summary of the Environment methods that specify geometric relationships.

int numrows()                                                                                                 returns the number of rows in the environment  
                                                                                                                           returns -1 if environment is unbounded  
int numcols()                                                                                                 returns the number of columns in the environment  
                                                                                                                           returns -1 if environment is unbounded  
int numAdjacentNeighbors()                                                                       returns the number of immediate neighbors,  
                                                                                                                           this is four for the MBS, would be 8 if diagonals were included  
Direction randomDirection()                                                                     returns a random Direction among the immediate neighbors  
                                                                                                                           this is north, east, south, or west for the MBS

Direction getDirection(Location fromLoc, Location toLoc)           returns the direction from one Location to another  
Location getNeighbor(Location fromLoc, Direction compassDir)   returns the immediate neighbor from a given Location in  
                                                                                                                           in a given Direction  
ArrayList neighborsOf(Location ofLoc)                                                 returns a list of all Locations neighboring the given Location

Among these methods, numAdjacentNeighbors is really for internal use with the other methods getNeighbor and neighborsOf. It returns four for the MBS environments. It would return eight if diagonally adjacent locations were considered immediate neighbors. It would return six for a hexagonal environment. It is not normally called by other classes that are a client of an Environment.

The methods numrows and numcols simply return the number of rows or columns in the environment. For unbounded environments, introduced in chapter five, they return -1.

The method randomDirection returns a random direction among the possible directions to immediate neighbors. Thus, for the MBS this method will return a direction equal to Direction.NORTH, Direction.EAST, Direction.WEST, or Direction.SOUTH. This differs from the randomDirection method in the class Direction.

The methods getDirection and getNeighbor express the relationship among locations and directions, in terms of the immediate neighbors. The method neighborsOf returns all the neighboring locations of a given location. This takes into account locations in corners and on the sides of a bounded environment -- neighborsOf would return a list of two neighbors for a actual parameter that is in a corner and three for one on the side.

These methods can be used to manipulate Location and Direction objects according to the geometry specified by a given Environment, even if they are not embedded in objects that are stored in the Environment.

#### **4.2 Managing Locatable objects**

The second main responsibility for an Environment is to store Locatable objects and manage the stored objects. The following is a summary of the methods that carry out this responsibility.

void add(Locatable obj)                                                    adds the given object to this environment  
void remove(Locatable obj)                                              removes the given object from this environment

int numObjects()                                                                  returns the number of objects currently in this environment  
Locatable[] allObjects()                                                  returns an array of all objects in this environment  
boolean isEmpty(Location loc)                                        indicates whether a given location is empty or not  
Locatable objectAt(Location loc)                                  returns the object at the given location, null if no object is at the location

void recordMove(Locatable obj, Location oldLoc)    records a move of the given object from the old location to its current location

All environments for the MBS case study enforce the requirement that at most one object be at any location in the environment. This is enforced by having the add and recordMove methods throwan IllegalArgumentException if there is an attempt to put an object at a location already occupied by another object in the environment. This is specified in the documentation for the Environment interface.

The methods add and remove carry out these operations. Note that any object added to an Environment must be Locatable, meaning it implements that interface and therefore has a location method. The recordMove method has the responsibility of maintaining consistency between a particular Locatable object's own record of its location and the location that the Environment has recorded for it. (Because the environment classes are"black-box" code, we do not know how it is implemented, but it may use the location of an object to organize its storage method. This is discussed further in chapter five, where this black-box is opened.) Consequently, whenever a Locatable object stored in a particular environment changes its location, it should always call that environment's recordMove method.

The four methods numObjects, allObjects, isEmpty, and objectAt all provide information about the current state of the environment. Method numObjects tells us how many objects are currently in the environment and allObjects returns an array of those objects (in no specified order). Method isEmpty lets us determine is a given location is empty and objectAt returns the object at a given location. The object returned by objectAt and the array returned by allObjects are of type Locatable, so if in a particular application (such as the MBS) we know that they have a specific type (such as Fish), we will need to downcast to that type to use its methods.

The Environment classes provide a means of storing a collection of objects with location, and perhaps with other characteristics, according to the rules outlined above that are embedded ion the Environment methods. This can be used in a variety of different setting in addition to the case study, and we will consider some of these in this unit.

### **5. Random Numbers**

The MBS case study uses random numbers for determining several aspects of Fish behavior. It users the standard Java library class Random to generate random numbers. When Random is instantiated several within a single program, as it might be in versions of the MBS, a subtle problem arises. When the default seed for generating random numbers is used, the seed is based on reading the system clock. Since the typical program runs much faster than the system clock, several instances of Random may have the same seed, causing non-random behavior. One way to prevent this is to use only a single instance of Random for all random numbers. This is inconvenient, snce random numbers may be generated in different classes. The case study achieves the convenience of declaring an instance of Random anywhere it is needed while still really having only one instance by using the class RandNumGenerator to create any instance of random with theclass method getInstance. This method returns a reference to the same instance of Random everytime it is called, creating the one instance the first time it is called, a technique called the Singleton Pattern. Whenever an instance of random is needed, it is "created" by the following statement:  
Random rand = RandNumGenerator.getInstance();

Once a Random object such as rand has been declared, we use it by calling one of the following methods.

double nextDouble                     returns a random double in the range 0.0 to 1.0, not including 1.0  
int nextInt(int n)                   returns a random integer from 0 to n-1 inclusive

We will use random numbers for some examples for Location, Direction, and Environment.

### **6. Displays**

We have created a few classes for use in displaying location, direction, and environment information. These are relatively simple and can be adapted to other uses. Some of these displays create an icon from a graphics file such as a gif or jpg file, so that it would be easy to substitute your own graphic image for display. These files are stored under the Unit 1 folder in Course Documents.

#### **6.1 Location and Direction**

The first display enables us to set the color of a block in a grid by specifying the corresponding Location. We can also mark a spot with an arrow indicating a Direction. We use this display to test and experiment with the Location and Direction classes.

**class LocationDisplay**

constructor  
LocationDisplay(int numrows, int numcols)

methods  
setCellColor(Location loc, Color col)  
clearCellColor(Location loc)  
setCellDir(Location loc, Direction dir)  
clearCellDir(Location loc)

The program LocDirTest demonstrates the use of this display class. We will use it further in the exercises. LocationDisplay and LocDirTest and the required gif files are in the LocDir.zip file.

#### **6.2 Environment of Color Blocks**

The second display takes an environment that contains instances of the class ColorBlock, which implements Locatable. This display will show all the ColorBlocks in the environment.

**class ColorBlock**

constructor  
ColorBlock(Location loc, Color col)

methods  
Location location()  
Color color()  
String toString()

**class CBEnvDisplay**

constructor  
CBEnvDisplay(Environment env)

methods  
showEnv()

The program CBTest is a simple test of these classes. Additional exercises make use of this display. These classes are in the file ColorBlockEnv.zip.

#### **6.3 Environment of Location-Direction Objects**

We define a class that holds a Location and Direction, LocDir. The display class LDEnvDisplay will display an Environment containing these objects, each at its location with an arrow showing its direction. This display will also display ColorBlock objects, so the two types of object can be both be placed in the environment.

**class LocDir**

constructor  
LocDir(Location loc, Direction dir))

methods  
Location location()  
public Direction direction()  
String toString()

**class CBEnvDisplay**

constructor  
LDEnvDisplay(Environment env)

methods  
showEnv()

The program LDTest is a simple test of these classes. Additional exercises make use of this display. These classes are in the file LocDirEnv.zip.

Note that the display of a LocDir object uses gif formated graphics files to define the graphics display. By substituting your own gif files, you can display any graphic image that you want.