###### Chapter XI

**Dynamic Arrays with the ArrayList Class**

**Chapter XI Topics**

11.1 Introduction

11.2 ArrayList Methods

11.3 ArrayList and Primitive Data Types

11.4 ArrayList and Generics

11.5 ArrayList and the <For..Each> Loop

11.6 Two-Dimensional Dynamic Arrays

11.7 Actor, the SuperClass for

Most GridWorld Classes

11.8 Summary

**11.1 Introduction**

Most programming languages have one type of array. Frequently the array data structure is - like the static array of the previous chapter - both one-dimensional and multi-dimensional. Java has two types of arrays. Java has the static array, shown in the last chapter, which is constructed with a fixed size that cannot be altered during program execution. Additionally, Java has a dynamic array, which does not require a predetermined size at instantiation and can be altered during program execution. The focus of this chapter is on the dynamic array, which is implemented with the **ArrayList** class.

You will see that the dynamic array is very different from the static array. The dynamic array is a modern array. The **ArrayList** class is a class with a constructor and lots of methods. Such is not the case with the static array. You do see the **new** operator as a reminder that some object is allocated space in memory, but there are no methods and access is only provided with some bracket index **[ ] [ ]** operators. In truth the static array dates back to the early days of programming and in many languages the array data structure works pretty in the manner that you saw in the last chapter.

This chapter teaches you the modern array with many methods and there should be a question coming up. If we have a modern array, why bother with an old one. Pick the better one of the two arrays and reject the other one. This is a reasonable argument, but the truth is that both arrays have something to offer. By the time you have seen program examples with both types of arrays you should know the advantages of using each type.

|  |
| --- |
| **Java Arrays** |
| **•** Java has a *static array* capable of multi-dimensions.  • Java has a *dynamic array*, which is also capable of multi- dimensions.  • The **ArrayList** class is used for the *dynamic array*.  • *Java static arrays* have no methods.  • **ArrayList** is a class with several methods. |

**11.2 ArrayList Methods**

Java has an answer to the static array shortcomings, which does not allow any change in size during program execution. It is the **ArrayList** class. With an **ArrayList** object the quantity of elements can be altered on the fly during program execution, which is officially known as *dynamic resizing*. Dynamic resizing it great, but there are other features available. Static arrays have a conspicuous absence of methods to enhance the data processing needs. The **ArrayList** class not only handles resizing, but it also has a convenient set of methods to manipulate data. Program **Java1101.java**, in figure 11.1, demonstrates how to construct an **ArrayList** object and it shows how to use the **add** method to add new elements to the array. Note how new elements are added to the end of the **ArrayList** object.

**Figure 11.1**

|  |
| --- |
| // Java1101.java  // This program demonstrates the <add> method of the <ArrayList> class.  // Note that each name is added to the end of the list.  import java.util.ArrayList;  public class Java1101  {  public static void main(String args[])  {  System.out.println();  System.out.println("Java1101.java\n");    ArrayList names = new ArrayList();  names.add("Isolde");  names.add("John");  names.add("Greg");  names.add("Maria");  names.add("Heidi");    System.out.println("names contains " + names);  System.out.println();  }  } |

|  |
| --- |
| **Java1101.java Output**  Java1101.java  names contains [Isolde, John, Greg, Maria, Heidi] |

|  |
| --- |
| **ArrayList Method add** |
| **names.add("Tom");**  The **add** method allocates space for the newly enlarged array and then stores the new array element at the end of the **ArrayList** object. |

You should have observed a feature of the **ArrayList** class that does not exist with Java static arrays. All the elements in the array are displayed rather casually with the use of a simple statement like **System.out.println(names)**.This is a serious *nono* with a static array. Actually, it is not a *nono* in the compile sense. The result is not real practical. All you get is some memory address of the array storage location.

|  |
| --- |
| **Displaying ArrayList Elements** |
| **ArrayList** elements can be accessed with various methods. It is possible to display all the elements inside square brackets, separated by commas by using the **println** method.  **System.out.println(names);**  [Isolde, John, Greg, Maria, Heidi] |

Java static arrays use the **length** field to store the number of elements in an array object. A static array can use a field, because it can be made final and access is possible without the possibility of altering the field value. A dynamic array, like **ArrayList** uses a method, which alters a private field value. Program **Java1102.java**, in figure 11.2, demonstrates the use of the **size** method.

**Figure 11.2**

|  |
| --- |
| // Java1102.java  // This program uses the <size> method to determine the number of elements  // in an <ArrayList> object.  // Note that the value returned by the <size> method changes when more names  // are added to the <ArrayList> object.  import java.util.ArrayList;  public class Java1102  {  public static void main(String args[])  {  System.out.println();  System.out.println("Java1102.java\n");    ArrayList names = new ArrayList();  names.add("Isolde");  names.add("John");  names.add("Greg");  System.out.println("names contains " + names);;  System.out.println("There are " + names.size() + " elements in the names object.");    names.add("Maria");  names.add("Heidi");  System.out.println("names contains " + names);  System.out.println("There are " + names.size() + " elements in the names object.");  System.out.println();  }  } |

|  |
| --- |
| **Java1102.java Output**  Java1102.java  names contains [Isolde, John, Greg]  There are 3 elements in the names object.  names contains [Isolde, John, Greg, Maria, Heidi]  There are 5 elements in the names object. |

|  |
| --- |
| **ArrayList Method size** |
| **int count = names.size();**  The **size** method returns the number of elements of the **ArrayList** object **names**. |

Program **Java1103.java**, in figure 11.3, shows how individual elements of the **names** object are accessed with the **get** method. Note how the **size** method controls the loop. Any attempt to access an array element at a location that does not exist, results in an IndexOutOfBoundsException error message, which is precisely what happens with static arrays.

**Figure 11.3**

|  |
| --- |
| // Java1103.java  // This program shows how to access specified elements in an <ArrayList> object  // with the <get> method.  import java.util.ArrayList;  public class Java1103  {  public static void main(String args[])  {  System.out.println();  System.out.println("Java1103.java\n");    ArrayList names = new ArrayList();  names.add("Isolde");  names.add("John");  names.add("Greg");  names.add("Maria");  names.add("Heidi");    System.out.println();  for (int k = 0; k < names.size(); k++)  System.out.println(names.get(k));  System.out.println();    for (int k = names.size()-1; k >= 0; k--)  System.out.println(names.get(k));  System.out.println();  }  } |

|  |
| --- |
| **Java1103.java Output**  Java1103.java  Isolde  John  Greg  Maria  Heidi  Heidi  Maria  Greg  John  Isolde |

|  |
| --- |
| **ArrayList Method get** |
| **System.out.println(names.get(3));**  The **get** method accesses a specified array element. The parameter of the **get** method is the *index* of the **ArrayList** object and starts at **0**. |

The static array index operator is very versatile. It can be used to access array elements for display and it can also be used to change the value of an array element. **ArrayList** methods are more specialized. The **get** method is fine for displaying individual elements, but another method is required to alter any values. This job is performed by the **set** method. Method **set** requires two parameters, one for the array element index and a second parameter for the new array element value. Program **Java1104.java**, in figure 11.4, starts with five initial names and then changes four of the names.

**Figure 11.4**

|  |
| --- |
| // Java1104.java  // This program demonstrates the <set> method of the <ArrayList> class, which  // replaces existing elements with a new object.  import java.util.ArrayList;  public class Java1104  {  public static void main(String args[])  {  System.out.println();  System.out.println("Java1104.java\n");    ArrayList names = new ArrayList();  names.add("Isolde");  names.add("John");  names.add("Greg");  names.add("Maria");  names.add("Heidi");    System.out.println("names contains " + names);  System.out.println();    names.set(1,"Jessica");  names.set(2,"Anthony");  names.set(3,"Haley");  names.set(4,"Alec");    System.out.println("names contains " + names);  System.out.println();  }  } |

**Figure 11.4 Continued**

|  |
| --- |
| **Java1104.java Output**  Java1104.java  names contains [Isolde, John, Greg, Maria, Heidi]  names contains [Isolde, Jessica, Anthony, Haley, Alec] |

|  |
| --- |
| **ArrayList Method set** |
| **names.set(4,"Tom");**  The **set** method uses the first parameter as the index location to find an array element and then replaces it with the value of the second **set** parameter. You will get an error if you try to access an index location, which has not been allocated yet. |

Earlier in this chapter, **ArrayList** was advertised as a dynamic array. Dynamic in the sense that resizing is possible during program execution. You have seen objects grow in size by adding additional elements. Resizing also can be used to make an array data structure smaller. Program **Java1105.java**, in figure 11.5, resizes the array with the **remove** method. Method **remove** requires a single parameter, which is the index of the array element to be removed.

**Figure 11.5**

|  |
| --- |
| // Java1105.java  // This program demonstrates the <remove> method of the <ArrayList> class to  // delete a specified list element.  import java.util.ArrayList;  public class Java1105  {  public static void main(String args[])  {  System.out.println();  System.out.println("Java1105.java\n");  ArrayList names = new ArrayList();  names.add("Isolde");  names.add("John");  names.add("Greg");  names.add("Maria");  names.add("Heidi");  System.out.println("names contains " + names);  System.out.println();  names.remove(2);  System.out.println("names contains " + names);  System.out.println();  names.remove(3);  System.out.println("names contains " + names);  System.out.println();  }  } |

**Figure 11.5 Continued**

|  |
| --- |
| **Java1105.java Output**  Java1105.java  names contains [Isolde, John, Greg, Maria, Heidi]  names contains [Isolde, John, Maria, Heidi]  names contains [Isolde, John, Maria] |

|  |
| --- |
| **ArrayList Method remove** |
| **names.remove(3);**  The **remove** method removes the array element at the index location of its parameter and decreases the object size by one array element. |

The **add** method is overloaded. An earlier program example introduced the **add** method with a single parameter. This single parameter provides the value of a new array element, which is added as the last element of the array. It is also possible to add a new element at a specified location. Be careful and do not confuse this second **add** method with the **set** method. Method **set** alters the value at a specified index. The overloaded **add** method *inserts* a new array element at a specified index and in the process bumps elements to the next index value. Program **Java1106.java**, in figure 11.6, will appear similar to the previous **set** program example. The key difference is that this time the array grows in size and none of the original values are lost.

**Figure 11.6**

|  |
| --- |
| // Java1106.java  // This program demonstrates how to use the <add> method of the <ArrayList> class to  // insert new elements at a specified location.  import java.util.ArrayList;  public class Java1106  {  public static void main(String args[])  {  System.out.println();  System.out.println("Java1106.java\n");    ArrayList names = new ArrayList();  names.add("Isolde");  names.add("John");  names.add("Greg");  names.add("Maria");  names.add("Heidi");    System.out.println("names contains " + names);  System.out.println();    names.add(2,"Jessica");    System.out.println("names contains " + names);  System.out.println();  names.add(3,"Anthony");    System.out.println("names contains " + names);  System.out.println();  }  } |

|  |
| --- |
| **Java1106.java Output**  Java1106.java  names contains [Isolde, John, Greg, Maria, Heidi]  names contains [Isolde, John, Jessica, Greg, Maria, Heidi]  names contains [Isolde, John, Jessica, Anthony, Greg, Maria, Heidi] |

|  |
| --- |
| **ArrayList Method add (second overloaded add)** |
| **names.add(3,"Kathy");**  The overloaded **add(3,"Kathy")** method adds or rather *inserts* a new array element at the indicated index. |

There are more methods in the **ArrayList** class. The six methods in this section are AP tested methods, because they are the most common methods and allow a wide range of data processing with a dynamic array.

It may seem that the **ArrayList**,with all its methods and dynamic resizing, is so superior to the static array that there exists little justification to hang on to a type of array that goes back to the early days of programming.

The simple reality is that **ArrayList**, good and flexible as it may be, is primarily a one-dimensional array. There exists plenty of processing that calls for two or more dimensional processing in the real computing word. Multi-dimensional arrays are easier to access with the static array. Furthermore, the convenience of constructing a new array with a set of initial values is very easy with static arrays and does not exist with the **ArrayList** class.

|  |
| --- |
| **AP Computer Science Exam Alert** |
| The **ArrayList** class is tested on the AP exam with the following six methods:  **int size()**  **boolean add(E obj)**  **void add(int index, E obj)**  **E get(int index)**  **E set(int index, E obj)**  **E remove(int index)**  In the method headings above **E** is the data type of the **E**lement that is added or returned. |

**11.3 ArrayList and Primitive Data Types**

In Chapter VI the **Integer** class was introduced. The **Integer** class is a *wrapper* class, which stores **int** values in an object. This is very important for data structures that can only store object values. The static array is quite relaxed about such issues and can store both primitive types and objects. Many classes, like the **ArrayList**,only store objects.

This does not mean that primitive types are off limits to an **ArrayList** object. Courtesy of the **Integer** class program **Java1107.java**, in figure 11.7, does a fine job storing **int** values. A "commented-out" segment proves that **Integer** objects are not **int** values and cannot be treated as such, like trying arithmetic addition.

**Figure 11.7**

|  |
| --- |
| // Java1107.java  // This program demonstrates that <int> values stored into an <ArrayList> object  // must first be converted to <Integer> objects.  // <ArrayList> can only store objects members, not primitive data types.  // Initially, this program compiles, and executes. If you remove the comments  // from the program an attempt is made to add the values of the <numbers>  // object, which is not possible.  import java.util.ArrayList;  import java.util.Random;  public class Java1107  {  public static void main(String args[])  {  System.out.println();  System.out.println("Java1107.java\n");  Random rand = new Random(12345);  ArrayList numbers = new ArrayList();    for (int k = 1; k <= 48; k++)  {  int rndInt = (rand.nextInt(900) + 100);  numbers.add(new Integer(rndInt));  }    // int sum = 0;  // for (int k = 0; k < numbers.size(); k++)  // {  // sum += numbers.get(k);  // }  // System.out.println("Sum: " + sum);  System.out.println();  }  } |

|  |
| --- |
| **1107.java Output with comments in place**  Java1107.java |

**Figure 11.7 Continued**

|  |
| --- |
| **1107.java Output with comments removed**  C:\Users\JohnSchram\Documents\LearnAPCS\APCS-LearningUnits\  APCS-11-Dynamic Arrays\Programs11\Java1107.java:32: error:  bad operand types for binary operator '+'  sum += numbers.get(k);  ^  first type: int  second type: Object  Note: C:\Users\JohnSchram\Documents\LearnAPCS\APCS-LearningUnits\APCS-11-Dynamic Arrays\Programs11\Java1107.java uses unchecked or unsafe operations.  Note: Recompile with -Xlint:unchecked for details.  1 error |

NOTE: Do not be concerned with the “unchecked or unsafe operations” warning.

The same approach can be used for other primitive data types, such as **double** and **boolean**. These primitive types both have wrapper classes **Double** and **Boolean**, which can be used to store simple data types as objects in the **ArrayList** class.

|  |
| --- |
| **ArrayList and Primitive Data Types** |
| The **ArrayList** class can only store **Object** values.  Primitive data type values can be stored indirectly using wrapper classes.  The **Integer** class *wraps* **int** values.  The **Double** class *wraps* **double** values.  The **Boolean** class *wraps* **boolean** values. |

**11.4 ArrayList and Generics**

Java version 5.0 solved some problems with Java classes. Prior to Java 5.0 there was a problem with handling objects. An object stores a reference, which is a memory address. Now at this memory address actual practical data information can be stored of any type. However, this information can be any type and that can cause confusion and peculiarities.

Consider program **Java1108.java**, in figure 11.08. This program shows an **ArrayList** object which properly stores **Person** objects as its elements. Two **Person** objects are instantiated and then added to the **people** object. When an attempt is made to access individual **people** objects the program complains and causes syntax errors. The error message indicates *incompatible types*. This may seem to be a mystery, because only **Person** objects are used and how can they be incompatible? The problem is that the actual value stored for each **people** object is a reference and its data type is unknown.

**Figure 11.8**

|  |
| --- |
| // Java1108.java  // This program has no output, which does not matter, because it does not compile.  // You will see two "incompatible types" syntax errors. This may seem strange  // because the <ArrayList> object stores <Person> objects.  import java.util.ArrayList;  public class Java1108  {  public static void main(String args[])  {  System.out.println();  System.out.println("Java1108.java\n");    ArrayList people = new ArrayList();  people.add(new Person("Joe",21));  people.add(new Person("Sue",20));    Person student1 = people.get(0);  Person student2 = people.get(1);  System.out.println();  }  }  class Person  {  private String name;  private int age;    public Person (String n, int a)  {  name = n;  age = a;  }  } |

|  |
| --- |
| **Java1108.java Output**  C:\Users\JohnSchram\Documents\LearnAPCS\APCS-LearningUnits\APCS-11-Dynamic Arrays\Programs11\Java1108.java:21: error: incompatible types  Person student1 = people.get(0);  ^  required: Person  found: Object  C:\Users\JohnSchram\Documents\LearnAPCS\APCS-LearningUnits\APCS-11-Dynamic Arrays\Programs11\Java1108.java:22: error: incompatible types  Person student2 = people.get(1);  ^  required: Person  found: Object  Note: C:\Users\JohnSchram\Documents\LearnAPCS\APCS-LearningUnits\APCS-11-Dynamic Arrays\Programs11\Java1108.java uses unchecked or unsafe operations.  Note: Recompile with -Xlint:unchecked for details.  2 errors |

Prior to Java 5.0 there was, and still is, a solution to the *unknown data type* problem. Program **Java1109.java**, in figure 11.9, is almost identical to the previous program, but two very strategic castings are used. In each assignment statement, like **Person student1 = (Person) people.get(0);** the casting of the object to **(Person)** provides Java with the required information. The program compiles, executes properly and everybody is happy.

**Figure 11.9**

|  |
| --- |
| // Java1109.java  // This program compiles and there is still no output. Output is not the  // issue. Understanding the correct syntax involved does matter.  // In this case lines 22 and 23 cast to the <Person> class, which makes  // Java happy. Without casting the data types are unknown.  import java.util.ArrayList;  public class Java1109  {  public static void main(String args[])  {  System.out.println();  System.out.println("Java1109.java\n");    ArrayList people = new ArrayList();  people.add(new Person("Joe",21));  people.add(new Person("Sue",20));    **Person student1 = (Person) people.get(0);**  **Person student2 = (Person) people.get(1);**  System.out.println();  }  }  class Person  {  private String name;  private int age;    public Person (String n, int a)  {  name = n;  age = a;  }  } |

|  |
| --- |
| **Java1109.java Output**  Java1109.java |

Along comes Java 5.0 and now Java has a better solution, which is known as *generics*. The selection of this term will make more sense in a later chapter. At the instantiation of a new **ArrayList** object the data type to be stored is specified. Look at **Java1110.java**, in figure 11.10, and you will see the following statement:

**ArrayList<Person> people = new ArrayList<Person>();**

This statement tells Java that the new **people** object will only store objects of the **Person** class. Casting is no longer necessary. This approach also creates an excellent self-documenting appearance where the code makes it clear what is stored in the new array.

**Figure 11.10**

|  |
| --- |
| // Java1110.java  // Since Java Version 5.0 the "casting" solution of the last program is so  // "old Java version". It is now possible to specify - at the time that the  // <ArrayList> object is constructed - what kind of object is stored.  // This is called "generics" and in this case Java knows it is <Person>.  import java.util.ArrayList;  public class Java1110  {  public static void main(String args[])  {  System.out.println();  System.out.println("Java1110.java\n");    ArrayList<Person> people = new ArrayList<Person>();  people.add(new Person("Joe",21));  people.add(new Person("Sue",20));    Person student1 = people.get(0);  Person student2 = people.get(1);  System.out.println();  }  }  class Person  {  private String name;  private int age;    public Person (String n, int a)  {  name = n;  age = a;  }  } |

|  |
| --- |
| **Java1116.java Output**  Java1116.java |

Program **Java1111.java**, in figure 11.11, shows how nicely a Java program works with the *generic* feature. You now see a program, which starts with a **number1** object. This object is instantiated to store **Integer** objects and three numbers are stored in **numbers1**.

A second **ArrayList** object, **numbers2**, is instantiated with the same intention to store **int** values wrapped inside **Integer** objects. This is followed by accessing three elements from **numbers1** and assigning all three to **numbers2**. This works flawlessly without a hitch.

**Figure 11.11**

|  |
| --- |
| // Java1111.java  // This program shows another benefit of using generics.  // There are two <ArrayList> objects and both are constructed  // to store <Integer> values. After three values are entered  // in the <numbers1> object, those values are then assigned  // to <numbers2>, which works without problems.  import java.util.ArrayList;  public class Java1111  {  public static void main(String args[])  {  System.out.println();  System.out.println("Java1111.java\n");    ArrayList<Integer> numbers1 = new ArrayList<Integer>();  numbers1.add(new Integer(100));  numbers1.add(new Integer(200));  numbers1.add(new Integer(300));  System.out.println(numbers1);    ArrayList<Integer> numbers2 = new ArrayList<Integer>();  numbers2.add(numbers1.get(0));  numbers2.add(numbers1.get(1));  numbers2.add(numbers1.get(2));  System.out.println(numbers2);    System.out.println();  }  } |

**Figure 11.11 Continued**

|  |
| --- |
| **Java1111.java Output**  Java1111.java  [100, 200, 300]  [100, 200, 300] |

It was mentioned earlier that an array by definition is a data structure with elements of the same type. Consider what happens when *generics* are not used. Program **Java1112.java**, in figure 11.12, adds a **Double** object and an **Integer** object and finally a **String** object in the array. Java allows this practice and the resulting structure is hardly a proper array.

**Figure 11.12**

|  |
| --- |
| // Java1112.java  // Generics make sure that an array is in fact an array. An array  // is supposed to be a data structure with elements of the same type.  // This program example - which does not use generics - allows the  // list array to store three different data types.  import java.util.ArrayList;  public class Java1112  {  public static void main(String args[])  {  System.out.println();  System.out.println("Java1112.java\n");    ArrayList list = new ArrayList();  list.add(new Double(3.14159));  list.add(new Integer(200));  list.add(new String("Dubrovnik"));  System.out.println(list);  System.out.println();  }  } |

|  |
| --- |
| **Java1112.java Output**  Java1112.java  [3.14159, 200, Dubrovnik] |

Program **Java1113.java**, in figure 11.13, uses the proper approach to instantiating an **ArrayList** object. Like before, an attempt is made to enter three different values into the array. Java is not amused and rewards the confused programmer with two error messages.

**Figure 11.13**

|  |
| --- |
| // Java1113.java  // Once generics are used, Java becomes very picky. If you want to create  // an <ArrayList> object to store <Double> values, such as is shown below,  // then only <Double> values must be added.  // The attempt to add one <Double>, one <Integer> and one <String> object  // results in two errors.  import java.util.ArrayList;  public class Java1113  {  public static void main(String args[])  {  System.out.println();  System.out.println("Java1113.java\n");    ArrayList<Double> list = new ArrayList<Double>();  list.add(new Double(3.14159));  list.add(new Integer(200));  list.add(new String("Dubrovnik"));  System.out.println(list);  System.out.println();  }  } |

|  |
| --- |
| **Java1113.java Output**  C:\Users\JohnSchram\Documents\LearnAPCS\APCS-LearningUnits\  APCS-11-Dynamic Arrays\Programs11\Java1113.java:20: error:  no suitable method found for add(Integer)  list.add(new Integer(200));  ^  method ArrayList.add(int,Double) is not applicable  (actual and formal argument lists differ in length)  method ArrayList.add(Double) is not applicable  (actual argument Integer cannot be converted to Double by method invocation conversion)  C:\Users\JohnSchram\Documents\LearnAPCS\APCS-LearningUnits\  APCS-11-Dynamic Arrays\Programs11\Java1113.java:21: error:  no suitable method found for add(Integer)  list.add(new Integer(300));  ^  method ArrayList.add(int,Double) is not applicable  (actual and formal argument lists differ in length)  method ArrayList.add(Double) is not applicable  (actual argument Integer cannot be converted to Double by method invocation conversion)  2 errors |

**11.5 ArrayList & the Enhanced For Loop**

Since we are on the subject of Java 5.0 enhancements with generics, let us continue and check out the special **for..each** loop structure. This loop structure was first shown with the static arrays in the last chapter. It was necessary to wait for the array structure because the **for..each** loop cannot work with primitive data types. You need a data structure and since **ArrayList** is a data structure, let us check and see how the new loop stacks up.

Program **Java1114.java**, in figure 11.14, presents three displays. The first output is done directly with **println** and the **names** object. You have seen this feature before. It works fine with **ArrayList** objects, but not static arrays.

The second output uses the original **for** loop and visits every element of the **names** array. The loop counter variable is used to access each array element starting with index **0** and continue until the **names.size() - 1** index. This works fine for the static array and the dynamic array.

Finally, the **for..each** loop is used. Note the convention of selection variable **name** of type **String** to represent the single element. This convention provides readability. We can now read the loop statement as: *for each* ***String*** *element, called* ***name*** *in the object* ***names****,**display the value of* ***name****.*

**Figure 11.14**

|  |
| --- |
| // Java1114.java  // The final program in this chapter shows three ways to display the  // members of an <ArrayList> object. Note that the enhanced <for>  // loop works very well with <ArrayList> objects.  import java.util.ArrayList;  public class Java1114  {  public static void main(String args[])  {  System.out.println();  System.out.println("Java1114.java\n");  ArrayList<String> names = new ArrayList<String>();  names.add("Isolde");  names.add("John");  names.add("Greg");  names.add("Maria");  names.add("Heidi");  System.out.println(names);  System.out.println();  for (int index = 0; index < names.size(); index++)  System.out.println(names.get(index));  System.out.println();  for (String name: names)  System.out.println(name);  System.out.println();  }  } |

**Figure 11.14 Continued**

|  |
| --- |
| **Java1114.java Output**  Java1114.java  [Isolde, John, Greg, Maria, Heidi]  Isolde  John  Greg  Maria  Heidi  Isolde  John  Greg  Maria  Heidi |

**11.6 Two-Dimensional Dynamic Arrays**

Static arrays are very convenient to use for two-dimensional arrays and greater dimensions. It is temping to state that the **ArrayList** class is restricted to a single dimension and in direct use that is correct. You can with a little maneuvering use the **ArrayList** class to create multi-dimensional arrays. It is a little tedious process and more than two dimensions will really get entirely too convoluted.

Program **Java1115.java**, in figure 11.15, first constructs three **ArrayList** objects, **cats**, **swimmers** and **primates**. Each one of these objects is now stored in another object called **mammals**. The result is an **ArrayList** object of three **ArrayList** objects. We now have a two dimensional, dynamic array.

Pay attention how the generic declaration is handled. The output will also give evidence that this is an array of arrays. The output of brackets and commas, which is a **ArrayList** trademark is seen three times inside yet another set of brackets and commas.

**Figure 11.15**

|  |
| --- |
| // Java1115.java  // It is possible for each member of an <ArrayList> object to be an <ArrayList> object.  // This creates an array of arrays or two-dimensional dynamic array.  // In this example you get an <ArrayList> of <ArrayList> member of <String> members.  // Observe the nested generics syntax of ArrayList<ArrayList<String>>.  import java.util.ArrayList;  public class Java1115  {  public static void main (String args[])  {  System.out.println();  System.out.println("Java1115.java\n");  ArrayList<String> cats = new ArrayList<String>();  cats.add("Lions");  cats.add("Tigers");  ArrayList<String> swimmers = new ArrayList<String>();  swimmers.add("Whales");  swimmers.add("Dolphins");  ArrayList<String> primates = new ArrayList<String>();  primates.add("Gorillas");  primates.add("Chimpanzees");  ArrayList<ArrayList<String>> mammals = new ArrayList<ArrayList<String>>();  mammals.add(cats);  mammals.add(swimmers);  mammals.add(primates);  System.out.println(mammals);  System.out.println();  }  } |

|  |
| --- |
| **Java1115.java Output**  Java1115.java  [[Lions, Tigers], [Whales, Dolphins], [Gorillas, Chimpanzees]] |

This is also a good opportunity to see how access is possible with this type of data structure. Program **Java1116.java**, in Figure 11.16, uses the **for..each** loop structure in a nested loop to display every element of the **mammals** array. Be careful to use the correct identifier for the inner loop. The *right-identifier* of the inner loop (**mammal** in this case) is the *left-identifier* of the outer loop.

**Figure 11.16**

|  |
| --- |
| // Java1116.java  // This program demonstrates how to display the elements of a  // two-dimensional dynamic array and it also shows how to use  // a set of nested <for..each> loop structures.  import java.util.ArrayList;  public class Java1116  {  public static void main (String args[])  {  System.out.println();  System.out.println("Java1116.java\n");  ArrayList<String> cats = new ArrayList<String>();  cats.add("Lions");  cats.add("Tigers");  ArrayList<String> swimmers = new ArrayList<String>();  swimmers.add("Whales");  swimmers.add("Dolphins");  ArrayList<String> primates = new ArrayList<String>();  primates.add("Gorillas");  primates.add("Chimpanzees");  ArrayList<ArrayList<String>> mammals = new ArrayList<ArrayList<String>>();  mammals.add(cats);  mammals.add(swimmers);  mammals.add(primates);  for (ArrayList<String> mammal: mammals)  {  for (String animal: mammal)  System.out.println(animal);  System.out.println();  }  System.out.println();  }  } |

|  |
| --- |
| **Java1116.java Output**  Java1116.java  Lions  Tigers  Whales  Dolphins  Gorillas  Chimpanzees |

The final program example in Figure 11.17 displays the same output as the previous program. This time the original **for** loop structure is used. In comparison to the **for..each** loop, the original structure is more involved, but you will need to use this approach if you need to alter any array values.

**Figure 11.17**

|  |
| --- |
| // Java1117.java  // This program example demonstrates how to use the original <for>  // loop structure to display dynamic two-dimensional arrays.  import java.util.ArrayList;  public class Java1117  {  public static void main (String args[])  {  System.out.println();  System.out.println("Java1117.java\n");  ArrayList<String> cats = new ArrayList<String>();  cats.add("Lions");  cats.add("Tigers");  ArrayList<String> swimmers = new ArrayList<String>();  swimmers.add("Whales");  swimmers.add("Dolphins");  ArrayList<String> primates = new ArrayList<String>();  primates.add("Gorillas");  primates.add("Chimpanzees");  ArrayList<ArrayList<String>> mammals = new ArrayList<ArrayList<String>>();  mammals.add(cats);  mammals.add(swimmers);  mammals.add(primates);  for (int row = 0; row < mammals.size(); row++)  {  for (int col = 0; col < mammals.get(row).size(); col++)  System.out.println(mammals.get(row).get(col));  System.out.println();  }  System.out.println();  }  } |

|  |
| --- |
| **Java1117.java Output**  Java1117.java  Lions  Tigers  Whales  Dolphins  Gorillas  Chimpanzees |

**11.7 Actor, the SuperClass for Most**

**GridWorld Objects**

At this stage some fundamental reality about the GridWorld Case Study should be clear. There is a lot of program code created to make the program functional that is not your concern. Bunches of code exist that handle the graphical appearance of the program. How is this done? It is not your concern, and it says so in the file. At the same time there are other files, which are extremely important. They are important, because such files contain information that teaches important computer science concepts and furthermore the contents of the files are tested on the AP Computer Science Exam. All this business can rapidly become confusing. How does a student know what must be studied, what can be ignored and how can somebody tell the difference between these many files? The answers to those questions cannot be quickly answered, but be assured that when you are finished with this book, you will have learned all the required GridWorld Case Study information and some more.

You have learned some details about the **Bug** class and you have made some changes to the **Bug** class behavior, but there really has not been a detailed focus on any one class. In this chapter that will change as you put the **Actor** class and the **Location** class under the microscope. A thorough understanding of the **Actor** class and the **Location** class is the first priority in comprehending the framework of the GWCS.

Future chapters will in detail study the subclasses of the **Actor** class. Each one of the subclasses will alter the behavior of its superclass in different ways. A study of the **Actor** class and its many subclasses will be an excellent tool in understanding class interaction with inheritance.

|  |
| --- |
| **Actor Class and Location Class Significance** |
| The **Actor** class is the *super class* for all the GridWorld class objects displayed on the grid display.  The **Location** classworks with the **Actor** class to place objects on the grid at the desired *row* and *col* locations. |

**Using the Quick Reference Guide**

The last section may have caused alarm. The GWCS has many classes and how can a student make sense of all this business and expect to perform well on the APCS Exam with the GridWorld questions? The College Board did appreciate the complexity of the GWCS and created a *Quick Reference Guide.* The guide will be available during the APCS Exam and it will be available for you as you learn, practice and do GWCS labs and tests prior to the AP Exam. It is possible that you have already received a copy of this reference guide. Either way, this section will briefly explain its existence and help you to use this guide effectively. Figure 11.18 shows the cover page of the Quick Reference guide.

**Figure 11.18**

|  |
| --- |
|  |

The quick reference guide consists of a number of pages that is one appendix after another. It is meant to be part of the College Board GridWorld Case Study Manual. Figure 11.19 shows Appendix A, which is not a GWCS set of classes. Appendix A is a quick reference for the Java subset of the standard library classes and their methods that may be tested on the APCS Exam. This is a handy guide during the exam, if you are not sure of the required parameters that you use for certain methods.

**Figure 11.19**

|  |
| --- |
|  |

Starting with Figure 11.20 every Appendix is a reference for a GWCS class. Appendix B shows the **Location** class, which is an important class used by many other classes. Notice in the heading that is states **Testable API**.The Quick Reference Guide only includes classes that are testable. The heading for each appendix will specify **Testable API (Application Program Interface)** or it will specify **Testable Code**, as is the case with Appendix B, shown by Figure 11.21.

**Figure 11.20**

|  |
| --- |
|  |

**Figure 11.21**

|  |
| --- |
|  |

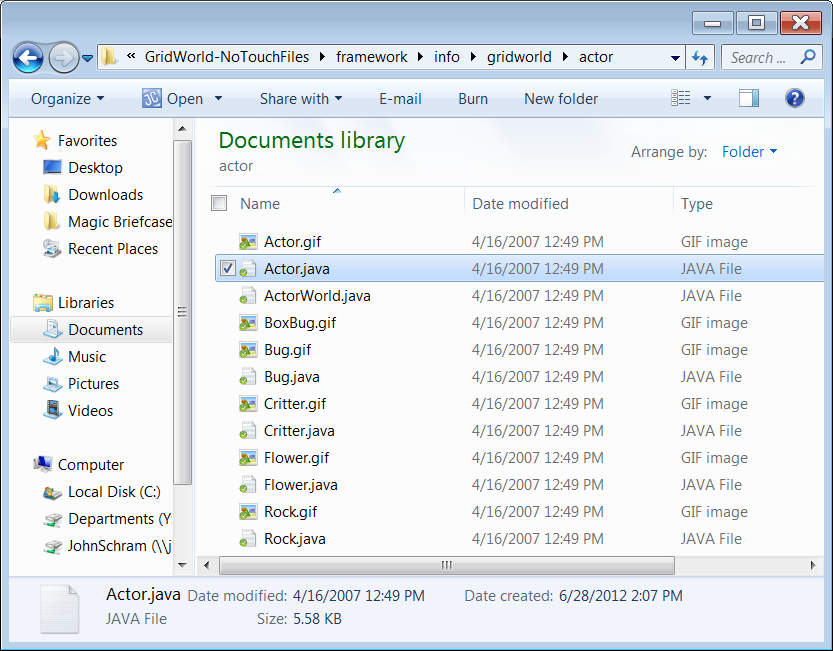
|  |
| --- |
| **GridWorld Case Study Quick Reference Note** |
| You are not expected to memorize all the testable classes and methods of the GWCS. A *Quick Reference Guide* is provided during the APCS Exam. You need to be very familiar with the guide, such that - even if you cannot remember everything - you know where to find the information quickly in your guide. |

**An Overview of the Actor Class**

Suppose you are bored and decide that investigating GridWorld classes is an excellent form of entertainment. Where would you go to find this information? If you guess the **GWCS-NoTouchFiles** then you are correct. Note it says *NoTouch*, not *NoRead*. Figure 11.22 shows the contents of the **actor** folder where you can find the **Actor** class and the sub classes of the **Actor** class. Observe the path, shown at the top of the window.

That folder path, **GWCS-NoTouchFiles\framework\info\gridworld\actor**, you need to use to find the folder shown in Figure 11.22. Observant students will see that the import statements used in the GWCS files are in fact the path to the desired location of a class.

**Figure 11.22**



Now let us take a peek at the actual **Actor** class. Figure 11.23 shows all the comments and all the code that is part of the **Actor** class. Scrolling through three pages of some small font, very detailed, code is quite intimidating.

**Figure 11.23**

|  |
| --- |
| /\*  \* AP(r) Computer Science GridWorld Case Study:  \* Copyright(c) 2005-2006 Cay S. Horstmann (http://horstmann.com)  \*  \* This code is free software; you can redistribute it and/or modify  \* it under the terms of the GNU General Public License as published by  \* the Free Software Foundation.  \*  \* This code is distributed in the hope that it will be useful,  \* but WITHOUT ANY WARRANTY; without even the implied warranty of  \* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  \* GNU General Public License for more details.  \*  \* @author Cay Horstmann  \*/  package info.gridworld.actor;  import info.gridworld.grid.Grid;  import info.gridworld.grid.Location;  import java.awt.Color;  /\*\*  \* An <code>Actor</code> is an entity with a color and direction that can act.  \* <br />  \* The API of this class is testable on the AP CS A and AB exams.  \*/  public class Actor  {  private Grid<Actor> grid;  private Location location;  private int direction;  private Color color;  /\*\*  \* Constructs a blue actor that is facing north.  \*/  public Actor()  {  color = Color.BLUE;  direction = Location.NORTH;  grid = null;  location = null;  }  /\*\*  \* Gets the color of this actor.  \* @return the color of this actor  \*/  public Color getColor()  {  return color;  }  /\*\*  \* Sets the color of this actor.  \* @param newColor the new color  \*/  public void setColor(Color newColor)  {  color = newColor;  }  /\*\*  \* Gets the current direction of this actor.  \* @return the direction of this actor, an angle between 0 and 359 degrees  \*/  public int getDirection()  {  return direction;  }  /\*\*  \* Sets the current direction of this actor.  \* @param newDirection the new direction. The direction of this actor is set  \* to the angle between 0 and 359 degrees that is equivalent to  \* <code>newDirection</code>.  \*/  public void setDirection(int newDirection)  {  direction = newDirection % Location.FULL\_CIRCLE;  if (direction < 0)  direction += Location.FULL\_CIRCLE;  }  /\*\*  \* Gets the grid in which this actor is located.  \* @return the grid of this actor, or <code>null</code> if this actor is  \* not contained in a grid  \*/  public Grid<Actor> getGrid()  {  return grid;  }  /\*\*  \* Gets the location of this actor.  \* @return the location of this actor, or <code>null</code> if this actor is  \* not contained in a grid  \*/  public Location getLocation()  {  return location;  }  /\*\*  \* Puts this actor into a grid. If there is another actor at the given  \* location, it is removed. <br />  \* Precondition: (1) This actor is not contained in a grid (2)  \* <code>loc</code> is valid in <code>gr</code>  \* @param gr the grid into which this actor should be placed  \* @param loc the location into which the actor should be placed  \*/  public void putSelfInGrid(Grid<Actor> gr, Location loc)  {  if (grid != null)  throw new IllegalStateException(  "This actor is already contained in a grid.");  Actor actor = gr.get(loc);  if (actor != null)  actor.removeSelfFromGrid();  gr.put(loc, this);  grid = gr;  location = loc;  }  /\*\*  \* Removes this actor from its grid. <br />  \* Precondition: This actor is contained in a grid  \*/  public void removeSelfFromGrid()  {  if (grid == null)  throw new IllegalStateException(  "This actor is not contained in a grid.");  if (grid.get(location) != this)  throw new IllegalStateException(  "The grid contains a different actor at location "  + location + ".");  grid.remove(location);  grid = null;  location = null;  }  /\*\*  \* Moves this actor to a new location. If there is another actor at the  \* given location, it is removed. <br />  \* Precondition: (1) This actor is contained in a grid (2)  \* <code>newLocation</code> is valid in the grid of this actor  \* @param newLocation the new location  \*/  public void moveTo(Location newLocation)  {  if (grid == null)  throw new IllegalStateException("This actor is not in a grid.");  if (grid.get(location) != this)  throw new IllegalStateException(  "The grid contains a different actor at location "  + location + ".");  if (!grid.isValid(newLocation))  throw new IllegalArgumentException("Location " + newLocation  + " is not valid.");  if (newLocation.equals(location))  return;  grid.remove(location);  Actor other = grid.get(newLocation);  if (other != null)  other.removeSelfFromGrid();  location = newLocation;  grid.put(location, this);  }  /\*\*  \* Reverses the direction of this actor. Override this method in subclasses  \* of <code>Actor</code> to define types of actors with different behavior  \*  \*/  public void act()  {  setDirection(getDirection() + Location.HALF\_CIRCLE);  }  /\*\*  \* Creates a string that describes this actor.  \* @return a string with the location, direction, and color of this actor  \*/  public String toString()  {  return getClass().getName() + "[location=" + location + ",direction="  + direction + ",color=" + color + "]";  }  } |

If the comments state that you must know the *API* of the **Actor** class then is all the code you just saw required for study and testing? Well that is not the case. You need to know the *API*only and that basically means you must know how to use the **Actor** class. This still requires studying, but only to the degree that you know, which methods are in a class and how to use them.

Figure 11.24 shows one page of the Quick Reference guide, which is the *API* of the **Actor** class. It is a lot less information, but it still requires time and practice to be comfortable with all the capabilities of the **Actor** class. Note that with each method heading you also get a comment that explains the method.

**Figure 11.24**

|  |
| --- |
|  |

**The Actor Class Attributes and Methods**

It helps to know what is expected of students when working with the GWCS. The GWCS is an example of a large, existing program. Programs of any consequence require debugging, because there are likely to be errors. Furthermore, programs are continuously updated with new enhancements.

Adding new features to a program requires creating new classes and new methods. The new features are created with existing tools, which can be found in the existing toolkits. Classes are toolkits and methods are tools. In other words, you cannot effectively create new functionality unless you have familiarity with the existing tools available to a program.

There will be many GWCS lab assignments where you are expected to create a new class and this new class will be a *subclass* of some existing GWCS class. The new class may newly-define some brand-new methods and then it may also re-define some existing methods. Either way, knowledge of existing GWCS classes and methods is a prerequisite to creating anything that is new.

We are getting ready to examine the methods of the **Actor** class. You do need to realize that study of this first class will not be completely clear. The **Actor** class uses methods of other classes that have not yet been explained. The reality is that all the GWCS classes interact, which means that the first few classes may appear a little fuzzy. By the time that you have investigated all the GWCS classes you will get a true understanding of the interaction of all the GWCS features.

The AP Computer Science Examination Quick Reference Guide only shows the method heading along with a description of the method. This book provides greater detail and shows the complete code for each method.

As you observe each one of the **Actor** methods that follow, notice the naming pattern. You will see that many of the *void* methods include **set** in the name and also many of the *return* methods include **get** in the name. *Void* methods frequently alter or set data attributes. *Return* methods get data attribute information.

**Actor Class Attributes**

The **Actor** class has four attributes that are shown in figure 11.25. The attribute identifiers do a good job explaining what information is stored.

**grid** stores the name of the **Actor**'s **Grid** or **null** if the object is not part of a **Grid**.

**location** stores the **Location** (row,col) coordinate of the **Actor** object.

**direction** stores the direction that the **Actor object** faces on the grid.

**color** stores the color of the **Actor** object.

**Figure 11.25**

|  |
| --- |
| **private Grid<Actor> grid;**  **private Location location;**  **private int direction;**  **private Color color;** |

**Actor Constructor Method**

The **Actor** class has only one constructor, shown in Figure 11.26. Every new **Actor** object is initialized as a blue object, which faces north. Facing north means that the object faces the top of the grid display.

It is very significant that both **grid** and **location** are initialized to **null.** The **Actor** class is the superclass for all the classes that you see on the GridWorld. All GridWorld classes will behave exactly like an **Actor**, unless changes are made. In other words, the constructor indicates that a new **Actor** object is not part of a GridWorld when it is constructed. This is precisely why you see program statements in the main method, like **world.add(new Bug());**

**Figure 11.26**

|  |
| --- |
| **public Actor()**  **{**  **color = Color.BLUE;**  **direction = Location.NORTH;**  **grid = null;**  **location = null;**  **}** |

**getColor return method**

The **getColor** method, shown in figure 11.27, returns the color of the **Actor** object. At first this method seems odd, because you may think that all **Actor** objects are blue. They are blue initially when they are constructed, but the color can be changed anywhere in a GridWorld program.

**Figure 11.27**

|  |
| --- |
| **public Color getColor()**  **{**  **return color;**  **}** |

**setColor void method**

Method **setColor**, in Figure 11.28,prevents the **Actor** objects from all being a boring blue color. This method has a single parameter, appropriately named **newColor**. You will see that the names or identifiers of methods and parameters in the GWCS are self-documenting. This means that the name is selected to indicate its meaning. It makes little sense to call a method **qwerty**, if it is supposed to change color.

**Figure 11.28**

|  |
| --- |
| **public void setColor(Color newColor)**  **{**  **color = newColor;**  **}** |

**getDirection return method**

Method **getDirection**, in figure 11.29, returns the direction of an **Actor** object. The **int** value returned is in the [0..359] degree range. Keep in mind that all of these **Actor** methods, shown here, are so significant, because these methods do not only apply to objects of the **Actor** class, but also to the many subclasses of the **Actor** class. In other words, method **getDirection** will also determine the direction that a **Bug** object is facing, since a **Bug** *is-an* **Actor**.

Now it is true that each one of the subclasses of **Actor** is different than an Actor in one or more ways, but you will find that all the subclasses use many methods that are completely unchanged from the original Actor methods.

**Figure 11.29**

|  |
| --- |
| **public int getDirection()**  **{**  **return direction;**  **}** |

**setDirection void method**

Method **setDirection**, in Figure 11.30, changes the direction of an object to face in a range between [0..359] degrees. Keep in mind that as you work with various **Actor** objects that the practical range for direction is really in multiples of **45** degrees. For instance, you have seen **Bug** objects move along the grid. Movement along the grid is possible only with direction changes that use multiples of **45** degrees.

The value of direction can change by adding or subtracting a numerical value. Method **setDirection** protects against the possibility that the direction can exceed 360 degrees or that the direction is negative.

**Figure 11.30**

|  |
| --- |
| **public void setDirection(int newDirection)**  **{**  **direction = newDirection % Location.FULL\_CIRCLE;**  **if (direction < 0)**  **direction += Location.FULL\_CIRCLE;**  **}** |

**getGrid return method**

Method **getGrid**, in Figure 11.31, is used for some special GWCS projects. It may seem odd to get the grid that the **Actor** object is contained in. It is possible to create multiple **Grid** objects and it is also possible that an **Actor** object is constructed, but not added to the current grid.

For instance, the statement **Actor tom = new Actor();**  creates a **tom** object. At this stage **tom** exists in an odd capacity without any home. A statement in the **main** method, like **world.add(tom);** takes care that object **tom** is contained in the grid defined by the **world** object. In the event that an **Actor** object is not part of any **Grid**, method **getGrid** returns **null**.

**Figure 11.31**

|  |
| --- |
| **public Grid<Actor> getGrid()**  **{**  **return grid;**  **}** |

**getLocation return method**

The **getLocation** method, in Figure 11.32, returns a **Location** object at the current location. A **Location** object stores a *row* and *col* value. Current location is the location of the object using **getLocation**. For example, suppose that a **Bug** object **bee** is used in the statement **bee.getLocation()**. In that case the *row* and *col* location of **bee** will be returned.

**Figure 11.32**

|  |
| --- |
| **public Location getLocation()**  **{**  **return location;**  **}** |

**putSelfInGrid void method**

Method **putSelfInGrid**, in Figure 11.33, serves double duty. First, it takes an **Actor** object, not currently contained in a grid, and adds it to the grid, provided by the first parameter. Second, the object will be placed at the location, provided by the second parameter. An object already at the desired location will be removed.

The method provides protection in the event the **Actor** object is already a member of a **Grid**. If the provided location is occupied then the occupant is removed from the **Grid**.

**Figure 11.33**

|  |
| --- |
| **public void putSelfInGrid(Grid<Actor> gr, Location loc)**  **{**  **if (grid != null)**  **throw new IllegalStateException("This actor is already contained in a grid.");**  **Actor actor = gr.get(loc);**  **if (actor != null)**  **actor.removeSelfFromGrid();**  **gr.put(loc, this);**  **grid = gr;**  **location = loc;**  **}** |

**removeSelfInGrid void method**

For various reasons sometimes it becomes necessary to remove an object from the GridWorld. Method **removeSelfInGrid**, in Figure 11.34, does the job, provided that the **Actor** object is first contained in a grid. It is difficult to explain this method thoroughly, because it uses methods from the **Grid** class that have not been explained yet.

Many classes in the GWCS use *exception handling*, a process whereby the programmer decides how to handle runtime error problems. The **throw** statements you see in various methods are exception handling statements.

**Figure 11.34**

|  |
| --- |
| **public void removeSelfFromGrid()**  **{**  **if (grid == null)**  **throw new IllegalStateException("This actor is not contained in a grid.");**  **if (grid.get(location) != this)**  **throw new IllegalStateException("The grid contains a different actor at location"**  **+ location + ".");**  **grid.remove(location);**  **grid = null;**  **location = null;**  **}** |

**act void method**

We now come to the single most important method of the GWCS, the **Act** method, shown in Figure 11.35. What makes each one of the **Actor** subclasses different from its **Actor** parent, is the redefinition of the **Act** method.

**Actor** objects are not very exciting. They make 180 turns in place. Method **act** is quite brief. First the current direction is found and then 180 degrees is added with the **HALF\_CIRCLE** constant value.

**Figure 11.35**

|  |
| --- |
| public void act()  {  setDirection(getDirection() + Location.HALF\_CIRCLE);  } |

**moveTo void method**

Method **moveTo**, in Figure 11.36, shows the method used to move an object to a specified new location. The object must be in a grid and the new location must be a valid location in the grid. If another object is at the new location, it will be removed.

**Figure 11.35**

|  |
| --- |
| public void moveTo(Location newLocation)  {  if (grid == null)  throw new IllegalStateException("This actor is not in a grid.");  if (grid.get(location) != this)  throw new IllegalStateException("The grid contains a different actor at location "  + location + ".");  if (!grid.isValid(newLocation))  throw new IllegalArgumentException("Location " + newLocation + " is not valid.");  if (newLocation.equals(location))  return;  grid.remove(location);  Actor other = grid.get(newLocation);  if (other != null)  other.removeSelfFromGrid();  location = newLocation;  grid.put(location, this);  } |

**toString return method**

The final method is a strange bird. At least **toString**, in Figure 11.36, is a strange bird to most students in the first semester of their AP Computer Science course. You will learn more about **toString** in later chapters. Right now understand this, method **toString** returns information about data attribute values of an object. These values are displayed using **System.out.print**.

Is that any use in a graphic GWCS program? Yes, it is for debugging purposes. In the process of working with a graphical program it can help to know attribute values. For instance, you might wonder why your new class is not moving correctly. With the help of **toString** you can see the values of location, direction and color. As you create your own classes, you can define your own **toString** method to display what is important to you and your class.

**Figure 11.36**

|  |
| --- |
| public String toString()  {  return getClass().getName() +  "[location=" + location + ",direction=" + direction + ",color=" + color + "]";  } |

**An Overview of the Location Class**

The Quick Reference Guide is a terrific aid when you work with the GWCS, but it is only a good reference, once you understand what is going on. For starters let us clear up some potential confusion. Do not get fooled. You did see **Location** business appear in the **Actor** section. The **getLocation** method is not a member of the **Location** class, as you might have expected. It is actually a member of the **Actor** class.

Perhaps it seems that everything connected with locations should be in the **Location** class, but that is not always logical. Consider **getLocation**, which returns a **Location** object that stores a *row* and *col* value. This method makes no sense in the **Location** class, which would mean that it returns a location of a location. It does make sense to return the location of some object on the grid and all these objects are **Actor** objects. The perfect place for **getLocation** is therefore in the **Actor** class.

Figure 11.37 shows the Quick Reference Guide portion of the **Location** class. It shows all the **Location** methods, but also a very large group of attributes. Something you did not see with the **Actor** class.

A tour of the **Actor** class started with a display of the entire class. This display was quite overwhelming and you learned much easier with the a step-by-step or method-by-method approach. You will not see the entire **Location** class in a single display now, but immediately start with the step-by-step approach.

**Figure 11.37**

|  |
| --- |
|  |

The data attributes, shown in Figure 11.38, are not part of the Quick Reference guide, but they are shown to illustrate a comparison. Keep in mind that the GWCS does not simply exist to play around a grid of objects. Our major goal is to use the GWCS to help understand computer science better. The **Location** class only has two **private** data attributes, show. The two attributes store the row and column values that are the main mission of the **Location** class. Only these two data attributes change during the execution of a GridWorld program.

Private data attributes are not part of a reference guide that shows only the *API*  of a class. The *API* can only consist of the **public** members of a class. Normally, data attributes are **private**, like **row** and **col** in Figure 11.38.

**Figure 11.38**

|  |
| --- |
| **private int row;** // row location in grid  **private int col;** // column location in grid |

Figure 11.39 shows additional data attributes, but these are **public**. Look on and you will see that each one of the **public** data attributes is also declared to be **final**. Yes, the data values can be accessed without a method in a public manner, but they are still *read-only*, because of the **final** keyword. Any data declared **final** cannot be changed. You will also note that the attributes are declared **static**, which means that they can be accessed for the duration of the program and they can be accessed with the **Location** class name. It is not necessary to create a **Location** object.

**Figure 11.39**

|  |
| --- |
| **public static final int NORTH = 0;**  **public static final int NORTHEAST = 45;**  **public static final int EAST = 90;**  **public static final int SOUTHEAST = 135;**  **public static final int SOUTH = 180;**  **public static final int SOUTHWEST = 225;**  **public static final int WEST = 270;**  **public static final int NORTHWEST = 315;** |

All the attributes in Figure 11.39 are compass directions. Technically, they are not necessary. A direction value of **180** for any **Actor** object will make that object face South. For readability, a set of constants or final attributes are declared that can be used anywhere in a GridWorld program when directions are needed.

The **Location** class has a second set of **public static final** attributes that are used to work with angles. Figure 11.40 shows these constant attributes and just like the directions they are technically not necessary. You can make any object turn **45** degrees clockwise by adding the value **45**.

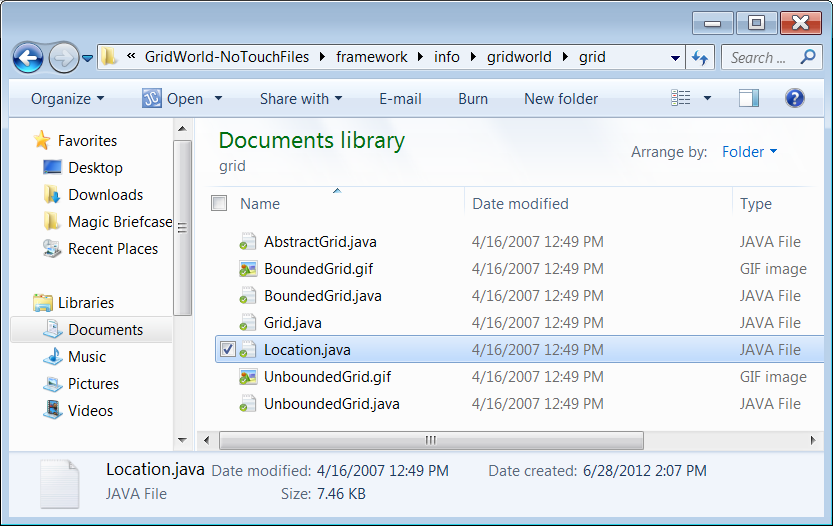
**Figure 11.40**

|  |
| --- |
| **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;** |

|  |
| --- |
| **Static Variables** |
| **Static** or **class** variables have recently been added to the AP Computer Science Examination.  Static variables can be accessed anywhere in the program without the construction of an object. Only the class name is required, as is shown inside the **turn** method of the **Bug** class below. This means that a **Bug** object makes a *45 degree,* *clockwise* turn whenever the **turn** method is called.  **public void turn()**  **{**  **setDirection(getDirection() + Location.HALF\_RIGHT);**  **}** |

The **Location** class is part of the **Grid** folder. Figure 11.41 shows the path you must follow to find the source code of **Location.java**. You will notice that **Location** is placed with all the *Grid* type classes.

**Figure 11.41**



**The Location Class Methods**

I know this GridWorld section is rapidly becoming quite boring. It cannot be avoided. It is more interesting to create some neat GridWorld project, but there comes the time to learn the basics that make everything work. The **Actor** class and the **Location** class are fundamental to the understanding of the GWCS. In the future there will not be a section with so many methods to absorb.

**Location constructor method**

There is only one, simple **Constructor**, shown in Figure 11.42, with the job to create a new object that stores row and column coordinate values.

**Figure 5.42**

|  |
| --- |
| **public Location(int r, int c)**  **{**  **row = r;**  **col = c;**  **}** |

**getRow and getCol return methods**

The short and simple methods in Figure 11.43 seem hardly necessary. You might wonder why they are not **public** attributes that can be accessed directly to get the values. This is precisely what was done before *Object Oriented Programming* came on the scene. Direct access of data variables is very unreliable and can unintentionally alter the data. Good program design makes data attributes **private** and uses local methods, like **getRow** and **getCol** to return the values.

**Figure 11.43**

|  |
| --- |
| **public int getRow()**  **{**  **return row;**  **}**  **public int getCol()**  **{**  **return col;**  **}** |

**equals return method**

The **equals** method is *redefined* for the **Location** class. Java has a super, super duper class, called the **Object** class. This class gives an initial definition for a variety of methods that are then available to any class in the Java language. Frequently these methods are altered in one of the many subclasses for a specific purpose. For instance, the **equals** method in Figure 11.44, is re-defined to test for equality if both the row and column values are equal.

**Figure 11.44**

|  |
| --- |
| **public boolean equals(Object other)**  **{**  **if (!(other instanceof Location))**  **return false;**  **Location otherLoc = (Location) other;**  **return getRow() == otherLoc.getRow() && getCol() == otherLoc.getCol();**  **}** |

**getAdjacentLocation return method**

Figure 11.45 shows the very practical **getAdjacentLocation** method. Objects on the grid move and they normally move in the direction that they are facing. This method returns the location of the next cell in the grid in any one of the eight compass directions. This simple statement requires a considerable amount of processing and many if statements.

**Figure 11.45**

|  |
| --- |
| **public Location getAdjacentLocation(int direction)**  **{**  **int adjustedDirection = (direction + HALF\_RIGHT / 2) % FULL\_CIRCLE;**  **if (adjustedDirection < 0)**  **adjustedDirection += FULL\_CIRCLE;**  **adjustedDirection = (adjustedDirection / HALF\_RIGHT) \* HALF\_RIGHT;**  **int dc = 0;**  **int dr = 0;**  **if (adjustedDirection == EAST)**  **dc = 1;**  **else if (adjustedDirection == SOUTHEAST)**  **{ dc = 1; dr = 1; }**  **else if (adjustedDirection == SOUTH)**  **dr = 1;**  **else if (adjustedDirection == SOUTHWEST)**  **{ dc = -1; dr = 1; }**  **else if (adjustedDirection == WEST)**  **dc = -1;**  **else if (adjustedDirection == NORTHWEST)**  **{ dc = -1; dr = -1; }**  **else if (adjustedDirection == NORTH)**  **dr = -1;**  **else if (adjustedDirection == NORTHEAST)**  **{ dc = 1; dr = -1; }**  **return new Location(getRow() + dr, getCol() + dc);**  **}** |

**getDirectionToward return method**

Suppose you want to visit another object on the grid for good or bad intentions. You are located in one place. The other object is in another location. You want to move to the other object in an efficient manner. Method **getDirectionToward**, in Figure 11.46, provides the closest compass direction to achieve your goal. This method actually involves calling a trigonometric function. Do keep in mind that you are expected to understand how to call the **Actor** and the **Location** classes with the proper parameters. You also need to understand what each method does, so you know which method to use. You are not expected to know the code.

**Figure 11.46**

|  |
| --- |
| **public int getDirectionToward(Location target)**  **{**  **int dx = target.getCol() - getCol();**  **int dy = target.getRow() - getRow();**  **int angle = (int) Math.toDegrees(Math.atan2(-dy, dx));**  **// mathematical angle is counterclockwise from x-axis,**  **// compass angle is clockwise from y-axis**  **int compassAngle = RIGHT - angle;**  **// prepare for truncating division by 45 degrees**  **compassAngle += HALF\_RIGHT / 2;**  **// wrap negative angles**  **if (compassAngle < 0)**  **compassAngle += FULL\_CIRCLE;**  **// round to nearest multiple of 45**  **return (compassAngle / HALF\_RIGHT) \* HALF\_RIGHT;**  **}** |

**hashCode return method**

You will see method **hashCode** in the Quick Reference Guide. A grid can be *bounded****,*** meaning that is has a fixed number of rows and columns. A grid can also be *unbounded*, meaning that row and column values can be positive, negative and have no maximum value. The former AP Computer Science AB course studied the design of an unbounded grid with the use of a **HashMap** object. The **hashCode** method, shown in Figure 11.47, was used - together with the **equals** method - to determine equal locations in an unbounded grid. This method is not used in AP Computer Science A.

**Figure 11.47**

|  |
| --- |
| **public int hashCode()**  **{**  **return getRow() \* 3737 + getCol();**  **}** |

**compareTo return method**

Method **compareTo**, in Figure 11.48, is implemented for the **Location** class. This makes it possible to determine order in grid locations. Order is compared numerically, which means that higher rows are lesser than lower rows and left columns are lesser than right columns.

**Figure 11.48**

|  |
| --- |
| **public int compareTo(Object other)**  **{**  **Location otherLoc = (Location) other;**  **if (getRow() < otherLoc.getRow())**  **return -1;**  **if (getRow() > otherLoc.getRow())**  **return 1;**  **if (getCol() < otherLoc.getCol())**  **return -1;**  **if (getCol() > otherLoc.getCol())**  **return 1;**  **return 0;**  **}** |

**toString return method**

Method **toString**, in Figure 11.49, showed up in the earlier **Actor** class and is redefined for many classes. It is a convenient method that can display practical value on the text screen with a **System.out.print** call. For the **Location** class the output is in the format (row,col).

**Figure11.49**

|  |
| --- |
| **public String toString()**  **{**  **return "(" + getRow() + ", " + getCol() + ")";**  **}** |

**11.8 Summary**

Java has two array data structures. There is the static array, shown in the last chapter, which cannot be resized during program execution. The static array also does not have any methods. However, on the plus side static arrays are very convenient for multi-dimensional arrays and they have initializer lists.

The second array, introduced in this chapter, is a dynamic array implemented with the **ArrayList** class. The dynamic array can be resized during program execution and there are many methods to process the data. The six most common methods are shown next and they are also tested on the AP Exam.

|  |
| --- |
| **ArrayList Method add** |
| **names.add("Tom");**  The **add** method allocates space for the newly enlarged array and then stores the new array element at the end of the **ArrayList** object. |

|  |
| --- |
| **ArrayList Method size** |
| **int count = names.size();**  The **size** method returns the number of elements of the **ArrayList** object **names**. |

|  |
| --- |
| **ArrayList Method get** |
| **System.out.println(names.get(3));**  The **get** method accesses a specified array element. The parameter of the **get** method is the *index* of the **ArrayList** object and starts at **0**.  Any attempt to access an element at a non-existing index results in an **IndexOutOfBoundsException** error. |

|  |
| --- |
| **ArrayList Method set** |
| **names.set(4,"Tom");**  The **set** method uses the first parameter as the index location to find an array element and then replaces it with the value of the second **set** parameter. You will get an error if you try to access an index location, which has not been allocated yet. |

|  |
| --- |
| **ArrayList Method remove** |
| **names.remove(3);**  The **remove** method removes the array element at the index location of its parameter and decreases the object size by one array element.  You will get an error if you try to access an index location, which does not exist. |

|  |
| --- |
| **ArrayList Method add (second overloaded add)** |
| **names.add(3,"Kathy");**  The overloaded **add(3,"Kathy")** method adds or rather *inserts* a new array element at the indicated index. |

**ArrayList** objects cannot store primitive data values directly. Some wrapper class, like **Integer**, **Double** or **Boolean** must be used to store simple data values.

Prior to Java 5.0 a program could not tell the type of value stored in an object, like an **ArrayList** object. Frequently casting was necessary to assign values. With Java 5.0 generics are possible, which means that at instantiation the data type of the **ArrayList** element is specified.

The enhanced **for..each** loop, introduced with static arrays also works very well with dynamic arrays.

It is possible to create a dynamic two-dimensional array with the **ArrayList** class. This becomes an **ArrayList** object of **ArrayList** objects. The generic declaration is nested as well.

The access of a two-dimensional array can be done with the newer **for..each** loop as well as the original index **for** loop. Access in this manner is more involved than using the index operator brackets of the static array.

The GridWorld Case Study uses the **Actor** class as the superclass for all classes that may occupy a GridWorld.

There are many methods in the **Actor** class and each one of the methods is available to any subclass of the **Actor** class.

A second important GridWorld class is the **Location** class. The **Location** class has many constant values defined, like **SOUTH** for compass directions and **HALF\_CIRCLE** for degree values. These class attributes are public and also static so that they can be called at any time in the program without the creation of an object.

Keep in mind that many methods have names that may seem *counter-intuitive*. A method like **getGrid** may appear to be a member of the **Grid** class and **getLocation** may seem to be a member of the **Location** class. Both methods are actually members of the **Actor** class.

Students are expected to understand how to use the GWCS classes and methods, but a reference guide of all the testable classes and methods is provided during the AP Computer Science Examination.