Chapter 13

Other Collections

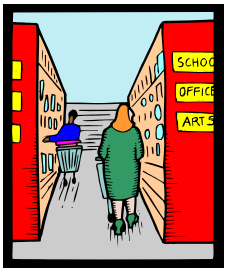
What is in This Chapter ?

We have already looked collections called **ArrayLists** and **Arrays**. There are actually other collections in JAVA which are useful for certain purposes. We will look here at the organization of some of the collection-related classes as well as explain the differences between the various **List** classes. We will look at two examples of how to use **Stacks** as well as how to **Sort** objects in a collection easily using the tools provided in JAVA's **Collections** class. Finally, we will show how we can remove duplicate objects by using **Sets**.

13.1 Collection Organization

Collections are a vital part of any programming language. As we have already seen with **ArrayLists** and **Arrays**, they allow many objects to be collected together, stored and passed around as one object (i.e., the collection itself). Just about every useful application of any kind requires collections for situations such as:

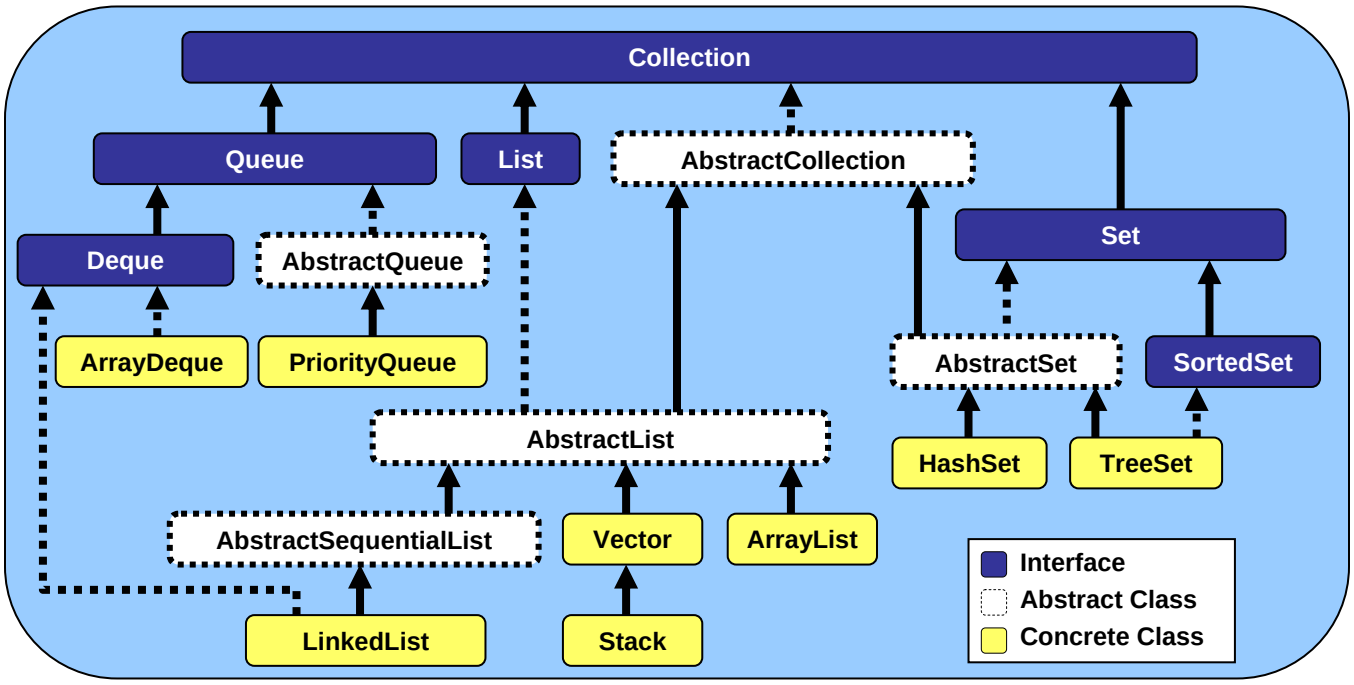
- storing products on shelves in a store
- maintaining information on customers
- keeping track of cars for sale, for rental or for servicing
- a personal collection of books, CDs, DVDs, cards, etc...
- maintaining a shopping cart of items to be purchased from a website





In JAVA, there is a small variety of *collection-related* classes that can be used to represent these various programming needs. These collections are located in the **java.util.Collection** package, along with some other useful tools. In this set of notes, we investigate (very briefly) some of these JAVA collections in a way that will help a programmer understand which collection is best for their particular programming application.

Collections are organized into a "seemingly complicated" hierarchy of JAVA interfaces and classes. Here is a diagram showing part of this hierarchy:



Notice that there are 8 concrete classes, and that all of them indirectly implement the **Collection** interface. Recall that an *interface* just specifies a list of method signatures ... not any code. That means, all of the concrete collection classes have something in common and that they all implement a common set of methods. The main commonality between the collection classes is that they all store objects called their ***elements***, which may be heterogeneous objects (i.e., the elements may be a mix of various (possibly unrelated) objects). Storing mixed kinds of objects in a **Collection** is allowed, but not often done unless there is something in common with the objects (i.e., they extend a common **Abstract** class or implement a common **Interface**).

The **Collection** interface defines common methods for querying (i.e., getting information from) and modifying (i.e., changing) the collection in some way. Here is a list of some of the common functionality available from all collections:

Querying methods (returns some kind of value):

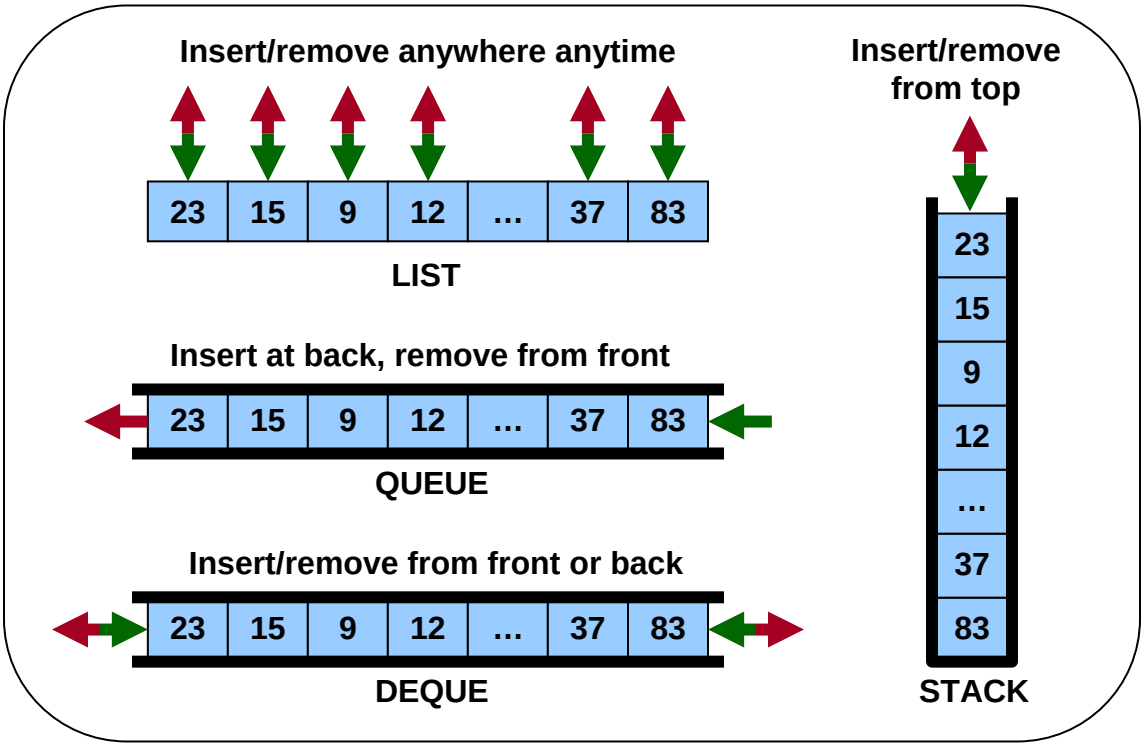
`size()`	returns the number of elements in the collection
`isEmpty()`	returns whether or not `size() == 0`
`contains(Object obj)`	returns whether or not given object **obj** is in the collection (uses `equals()` method for comparison)
`containsAll(Collection c)`	same as above but looks for ALL elements specified in the given collection **c**.

Modifying methods (changes the collection in some way):

`add(Object obj)`	adds the given object as an element to the collection (its location is not specified)
`addAll(Collection c)`	same as above but adds ALL elements specified in the given collection **c**.
`remove(Object obj)`	removes the given object from the collection (uses `equals()` method for comparison)
`removeAll(Collection c)`	same as above but removes ALL elements specified in the given collection **c**.
`retainAll(Collection c)`	same as above but removes all elements EXCEPT THOSE specified in the given collection **c**.
`clear()`	empties out the collection by removing all elements.

Now although the various collection types implement these methods, there are various restrictions for each of the collection classes that when followed, produce more efficient code.

For example, lets consider the differences between a **List**, a **Queue**, a **Deque** and a **Stack**:



A **List** (e.g., an **ArrayList**) allows us to access any of its elements at any time as well as insert or remove elements anywhere in the list at any time. The list will automatically shift the elements around in memory to make the needed room or reduce the unused space. The general **List** is the most flexible kind of list in terms of its capabilities. We use a general **List** whenever we have elements coming in and being removed in a random order. For example, when we have a shelf of library books, we may need to remove a book from anywhere on the shelf and we may insert books back into their proper location when they are returned. Typical methods for **Lists** are:

- add(**int** index, Object x)
- remove(**int** index)
- get(**int** index)
- set(**int** index, Object x)

Consider the **Queue**. A queue is used to store elements in a ***First-in-First-out*** *(FIFO)* order. In other words, the first element to be added to the queue is the first element to be taken out of the queue. This is analogous to a line-up that we see every day. The first person in line is the first person served (i.e., first-come-first-served). When people come, they go to the back of the line. People get served

from the front of the line first. Therefore, with a queue, we add to the back and remove from the front. We are not allowed to insert or remove elements from the middle of the queue. Why is this restriction a good idea ? Well, depending on how the queue is implemented, it can be more efficient (i.e., faster) to insert and remove elements since we know that all such changes will occur at the front or back of the queue. Removing from the front may then simply require moving the "front-of-the-line pointer" instead of shifting elements over. Also, adding to the back may require extending the "back-of-the-line pointer". Typical methods for **Queues** are:

- add(Object x)
- remove()
- peek()

What about a **Deque** ? A deque is a "***D**ouble-**E**nded-**QUE**ue*". It allows us to add/remove from the front or the back of the queue at any time, but no modifications to the middle. It has the same advantages of a regular single-ended Queue, but is a little more flexible in that it allows removal from the back of the queue and insertion at the front. An example of where we might use a deque is when we implement "Undo" operations in a piece of software. Each time we do an operation, we add it to the front of the deque. When we do an undo, we remove it from the front of the deque. Since undo operations usually have a fixed limit defined somewhere in the options (i.e., maximum 20 levels of undo), we remove from the back of the deque when the limit is reached. Typical methods for **Deques** are:

- addFirst(Object x)
- addLast(Object x)
- removeFirst()
- removeLast()
- getFirst()
- getLast()

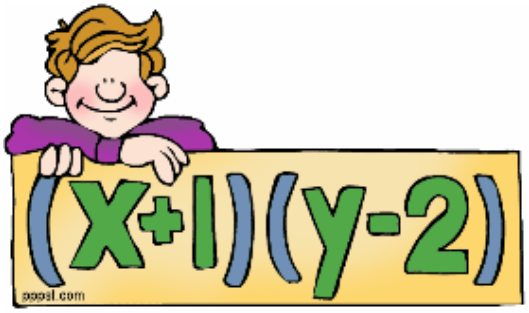
Finally, a **Stack** is used to store elements in a ***Last-in-First-out*** *(LIFO)* order. That is, we store information like a stack of items one on top of the other. When a new item comes in, we place it on the top of the stack and when we want to remove an item, we take the top one from the stack. Stacks are used for many applications in computer science such as syntax parsing, memory management, reversing data, backtracking, etc.. Typical methods for **Stacks** are:

- push(Object x)
- pop()
- isEmpty()
- peek()

It is not the purpose of this course to describe in detail various kinds of collections and data structures. You will gain a deeper understanding of the advantages and disadvantages between the various data structures in your second year COMP2402 course. For fun however, we will do a couple of examples to help you understand how to use a couple of these JAVA classes.

13.2 Example: Bracket Matching

Lets look at an example of using a data structure to solve a simple problem of matching brackets. Consider a math expression that contains numbers, operators and parentheses (i.e., round brackets). How could we write a program that takes a **String** representing a math expression and then determines whether or not the brackets match properly (i.e., each opening bracket has a matching closing bracket in the right order) ?




```
"((23 + 4 * 5) - 34) + (34 - 5))"  // no match
"((23 + 4 * 5) - 34) + ((34 - 5)"  // no match
"((23 + 4 * 5) - 34) + (34 - 5)"   // match
```

How would we approach solving this problem? Well, we need to understand the process. I'm sure that you realize that we need to look at all the String's characters. Perhaps from start to end with a loop, but then what do we do ?

Lets assume that we are not interested in determining whether the formula makes sense but rather that each opening bracket is matched by a closing bracket. Therefore, we are interested in the bracket characters **(** and **)**, but not the other characters. When encountering an open bracket as we go through the characters of the string, we need to do something. We might think right away of trying to find the matching closing bracket for each open bracket, but that is not as easy as it sounds. There are many special cases that can be tricky.

A simpler approach would be to make sure that whenever we find a closing bracket, we just need to make sure that we already encountered an open bracket to match with it. This can be done by keeping a count of the number of open brackets. When encountering an opening bracket we increment the counter and when encountering a closing bracket we decrement the counter. If, when all done, the counter is not zero, there is no match. Otherwise the brackets match. Consider these cases:

```
"()"          // counter = 0, match
"()("         // counter = 1, no match
"(("          // counter = 3, no match
"((())())"    // counter = 0, match
"(()))"       // counter = -1, no match
""            // counter = 0, match
```

There is a special case that we did not consider. If the counter ever becomes negative before we are done, then we must have encountered a closing bracket before an open bracket … and there is no match:

```
")("      // counter = -1, no match
"())("    // counter = -1, no match
```

So, how do we write the code ? We can use a **FOR** loop and some **IF** statements to check for brackets as follows:

```
public static boolean bracketsMatch(String s) {
    int   count = 0;
    char  c;

    for (int i=0; i<s.length(); i++) {
        c = s.charAt(i);
        if (c == '(') count++;
        if (c == ')') count--;
        if (count < 0)
            return false;
    }
    return count == 0;
}
```

Here is a test program to try it out:

```
import java.util.*;

public class BracketMatchingTestProgram {

    public static boolean bracketsMatch(String s) { /* code as above */ }

    public static void main(String args[]) {
        Scanner    keyboard = new Scanner(System.in);
        String     aString;

        do {
            System.out.println("Please enter expression: (<cr> to quit)");
            aString = keyboard.nextLine();

            if (bracketsMatch(aString))
                System.out.println("The brackets match");
            else
                System.out.println("The brackets do not match");
        } while (aString.length() > 0);
    }
}
```

Here are some testing results:

```
Please enter expression: (<cr> to quit)
((23 + 4 * 5) - 34) + (34 - 5))
The brackets do not match
Please enter expression: (<cr> to quit)
((23 + 4 * 5) - 34) + ((34 - 5)
The brackets do not match
Please enter expression: (<cr> to quit)
((23 + 4 * 5) - 34) + (34 - 5)
The brackets match
Please enter expression: (<cr> to quit)
()
The brackets match
Please enter expression: (<cr> to quit)
()(
The brackets do not match
Please enter expression: (<cr> to quit)
(((
The brackets do not match
Please enter expression: (<cr> to quit)
((()))())
The brackets match
Please enter expression: (<cr> to quit)
((()))
The brackets do not match
Please enter expression: (<cr> to quit)
)(
The brackets do not match
Please enter expression: (<cr> to quit)
())(
The brackets do not match
Please enter expression: (<cr> to quit)

The brackets match
```

I think that our solution works fine. The bracket-matching example above is not very difficult, but what if we have 3 kinds of brackets **(**, **[**, and **{** ? Consider this example:

```
"{2 +(3 *[4 - 5}])"   // not supposed to match
```

Maybe we can keep 3 counters ? If we just keep three counters separately, we cannot tell whether the brackets are well formed with respect to one another. We somehow need to know the ordering of each type of bracket so that we can ensure the reverse ordering when we find the closing brackets.

The need for backtracking may seem a little clearer if we consider a different application of the bracket matching program.

Suppose that we want to match the brackets in our JAVA code…

```
public class PrintWriterTestProgram {
        public static void main(String args[]) {
                try {
                        BankAccount    aBankAccount;
                        PrintWriter    out;

                        aBankAccount = new BankAccount("Rob Banks");
                        aBankAccount.deposit(100);
                        out = new PrintWriter(new FileWriter("myAccount2.dat"));
                        out.println(aBankAccount.getOwner());
                        out.println(aBankAccount.getAccountNumber());
                        out.println(aBankAccount.getBalance());
                        out.close();
                } catch (FileNotFoundException e) {
                        System.out.println("Error: Cannot open file for writing");
                } catch (IOException e) {
                        System.out.println("Error: Cannot write to file");
                }
        }
}
```

Here we see, for example, that the portion of code inside the **class** definition must have all of its brackets matching, and that involves matching the code inside the **main** method's body and then within the **try** block etc… The compiler does this kind of bracket matching to make sure that your code is well-formed.

The **stack** data structure is designed for this purpose. It allows us to back-track … which is essentially what we need to do when finding a closing bracket. Here is how we can use a stack. When we find an open bracket, we put it on the top of the stack, regardless of its type. When we find a closing bracket, we take the top opening bracket from the stack and check to see if it is the same type as the closing bracket. If not, the brackets are in the wrong order, otherwise all is fine and we continue onwards. If, when encountering a closing bracket, we find that the stack is empty, then there is no match either.

Lets look at the code. In JAVA, we make a Stack by simply calling its constructor:

```
Stack aStack = new Stack();
```

However, in our case, we are going to be placing bracket character on the Stack. Therefore we should specify this as follows:

```
Stack<Character> aStack = new Stack<Character>();
```

Then, we need to use the following **Stack** methods:

- **`push(Object obj)`** – to place a given object on the top of the stack
- **`pop()`** – to remove and return the top element of the stack.
- **`empty()`** – to determine whether or not there are any elements in the stack.

Here is the resulting code:

```
public static boolean bracketsMatch(String s) {
    Stack<Character>   aStack;
    char               c, top;

    aStack = new Stack<Character>();
    for (int i=0; i<s.length(); i++) {
        c = s.charAt(i);

        if ((c == '(') || (c == '[') || (c == '{'))  // got open bracket
            aStack.push(c);

        if ((c == ')') || (c == ']') || (c == '}')) { // got closed bracket
            if (aStack.empty())
                return false;    // no open bracket for this closed one

            top = aStack.pop();  // get the last opening bracket found
            if (((c == ')') && (top != '(')) ||
                ((c == ']') && (top != '[')) ||
                ((c == '}') && (top != '{')))
                return false;  // wrong closing bracket for last opened one
        }
    }
    return aStack.empty();  // No match if brackets are left over
}
```

Notice in the above code that it never has **return true** anywhere. In fact, it is only at the very end, once we have checked all characters that there is a chance for the method to return true. This will happen if the stack is empty (i.e., all open brackets have been matched with closing ones). If desired, you can simplify the above code by replacing the **IF** statements with a **SWITCH** statement as follows:

```
switch(c) {
    case '(':
    case '[':
    case '{':
        aStack.push(c); // got open bracket
        break;
    case ')':
        if (aStack.empty() || (aStack.pop() != '('))
            return false;
        break;
    case ']':
        if (aStack.empty() || (aStack.pop() != '['))
            return false;
        break;
    case '}':
        if (aStack.empty() || (aStack.pop() != '{'))
            return false;
        break;
}
```

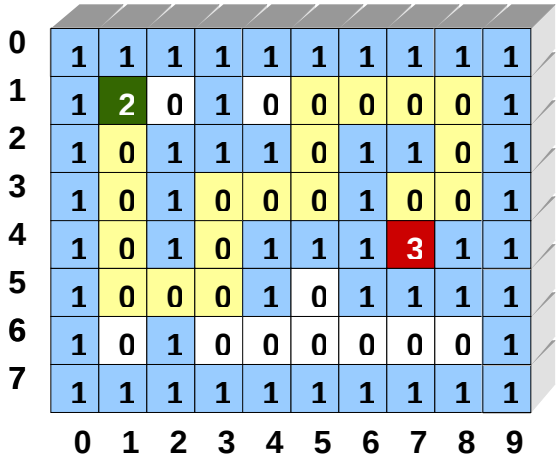
Here are some testing results that we would obtain if we replaced our previous **bracketsMatch()** method with this new method:

```
Please enter the expression: (just <cr> to quit)
](){}[
The brackets do not match
Please enter the expression: (just <cr> to quit)
()[]{}
The brackets match
Please enter the expression: (just <cr> to quit)
{{{{[[]]}}}}
The brackets match
Please enter the expression: (just <cr> to quit)
{{{{{{
The brackets do not match
Please enter the expression: (just <cr> to quit)
}}}}}}
The brackets do not match
Please enter the expression: (just <cr> to quit)
(((){}){}[()[]{])
The brackets do not match
Please enter the expression: (just <cr> to quit)
The brackets match
```

Challenge: Could you adjust the code above to read in a JAVA file instead of using a fixed string and have it ensure that the brackets match ?

13.3 Example: Maze Search

Recall the practice question from our discussion of 2D arrays. We had a maze, represented as an array of integers (where 0 meant an open space and 1 meant a wall was there). We wanted to know if we could find a path from a start location in the maze to an end location:



I'm sure you will agree if you were in the maze searching for the goal location, that you would occasionally encounter a dead-end and have to back track a little bit, trying another direction.

Since we are doing backtracking, could we use a **stack** to find a solution ? What would we even put in the stack ?

Well, the stack could keep track of the locations that we've been at. So, we could put locations on the stack. A location could be stored as an **(x,y)** Point.

There is a Point class in the **java.awt** package that we could use. Therefore, the start location in the above picture (i.e., the green 2) would be represented as `(1,1)` and the end location (i.e., the red 3) would be `(7,4)`.

Here are our starting variables:

```
int[][]          maze = {{1,1,1,1,1,1,1,1,1,1},
                         {1,0,0,1,0,0,0,0,0,1},
                         {1,0,1,1,1,0,1,1,0,1},
                         {1,0,1,0,0,0,1,0,0,1},
                         {1,0,1,0,1,1,1,0,1,1},
                         {1,0,0,0,1,0,1,1,1,1},
                         {1,0,1,0,0,0,0,0,0,1},
                         {1,1,1,1,1,1,1,1,1,1}};
Point            currentLoc = new Point(1,1);
Point            goalLoc = new Point(4,7);  // Here x=rows, y=cols
Stack<Point>     possibilities = new Stack<Point>();
```

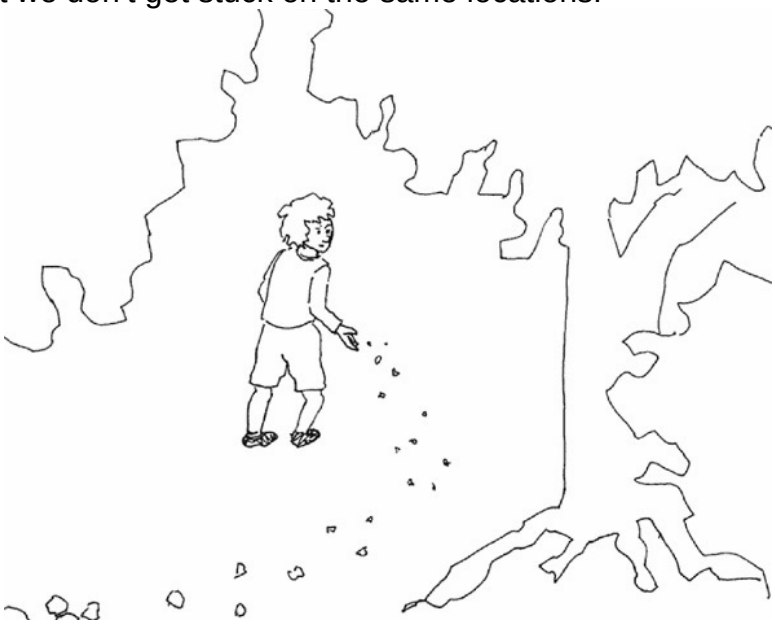
The **maze** variable maintains information about open space and walls. The **currentLoc** represents our current location in the maze. The **goalLoc** represents the location that we are trying to reach.

The **possibilities** stack will contain all the possible locations that we have been close to, but have not yet explored. Here is how it should work:

We start by placing the starting location **currentLoc** onto the stack. Then we go through a continuous loop that does the following:

```
REPEAT {
    IF (the stack is empty) THEN
        We cannot reach the goal, quit.
    Get a location from the top of the stack and move to it.
    IF (the location is the goal location) THEN
        We have reached the goal, quit.
    OTHERWISE
        Check all "open" locations around the current location
         and add them to the stack.
}
```

The above code will keep adding potential paths to the stack, but there is a problem. It is possible that we will add the same location to the stack multiple times. That is, we might end up circling back to the same locations over and over again. We need to add some breadcrumbs so that we don't get stuck on the same locations.



To do this, we need to change the values in the maze array so that we can distinguish between an "open" location that we **have never been** to and an "open" spot that we **have been** to before.

Perhaps we can use **-1** to *mark* a location as having been visited.
So, here are the changes to the algorithm:

```
REPEAT {
    IF (the stack is empty) THEN
        We cannot reach the goal, quit.
    Get a location from the top of the stack and move to it.
    Mark current location as having been visited (breadcrumb)
    IF (the location is the goal location) THEN
        We have reached the goal, quit.
    OTHERWISE
        Check all "open" locations around the current location
         that have not been visited and add them to the stack.
}
```

How do we leave a breadcrumb in the maze ?

We mark the current location in the array with a -1 as follows:

```
maze[currentLoc.x][currentLoc.y] = -1;
```

How do we add potential paths (i.e., the OTHERWISE part of the algorithm) ?

We just check the locations above, below, left and right of the current location to see if there is an unvisited location (i.e., with a value of **0** in that location of the maze) . Here is how we would handle checking the location above the current location:

```
if (maze[currentLoc.x-1][currentLoc.y] == 0)
    possibilities.push(new Point(currentLoc.x-1, currentLoc.y));
```

Can you come up with the other 3 cases ?

Be aware that the rows of the array correspond to the **x** values of the locations …and the columns correspond to the **y** values of the locations.

Here is the resulting code:

```
possibilities.push(currentLoc);
while(true) {
    if (possibilities.isEmpty()) {
        System.out.println("Cannot Reach Goal");
        break;
    }
    currentLoc = possibilities.pop();           // get the next location
    printMaze(maze, currentLoc, goalLoc);
    maze[currentLoc.x][currentLoc.y] = -1;      // place the breadcrumb
    if (currentLoc.equals(goalLoc)) {
        System.out.println("Reached Goal");
        break;
    }
    if (maze[currentLoc.x-1][currentLoc.y] == 0) // above
        possibilities.push(new Point(currentLoc.x-1, currentLoc.y));
    if (maze[currentLoc.x+1][currentLoc.y] == 0) // below
        possibilities.push(new Point(currentLoc.x+1, currentLoc.y));
    if (maze[currentLoc.x][currentLoc.y-1] == 0) // left
        possibilities.push(new Point(currentLoc.x, currentLoc.y-1));
    if (maze[currentLoc.x][currentLoc.y+1] == 0) // right
        possibilities.push(new Point(currentLoc.x, currentLoc.y+1));
}
```

You should be able to understand this code as it directly follows from the algorithm. For debugging purposes, a call to **printMaze()** has been inserted. This method will display the maze so that we can see how the program runs each time through the loop. It is similar to the code we used when we discussed 2D arrays.

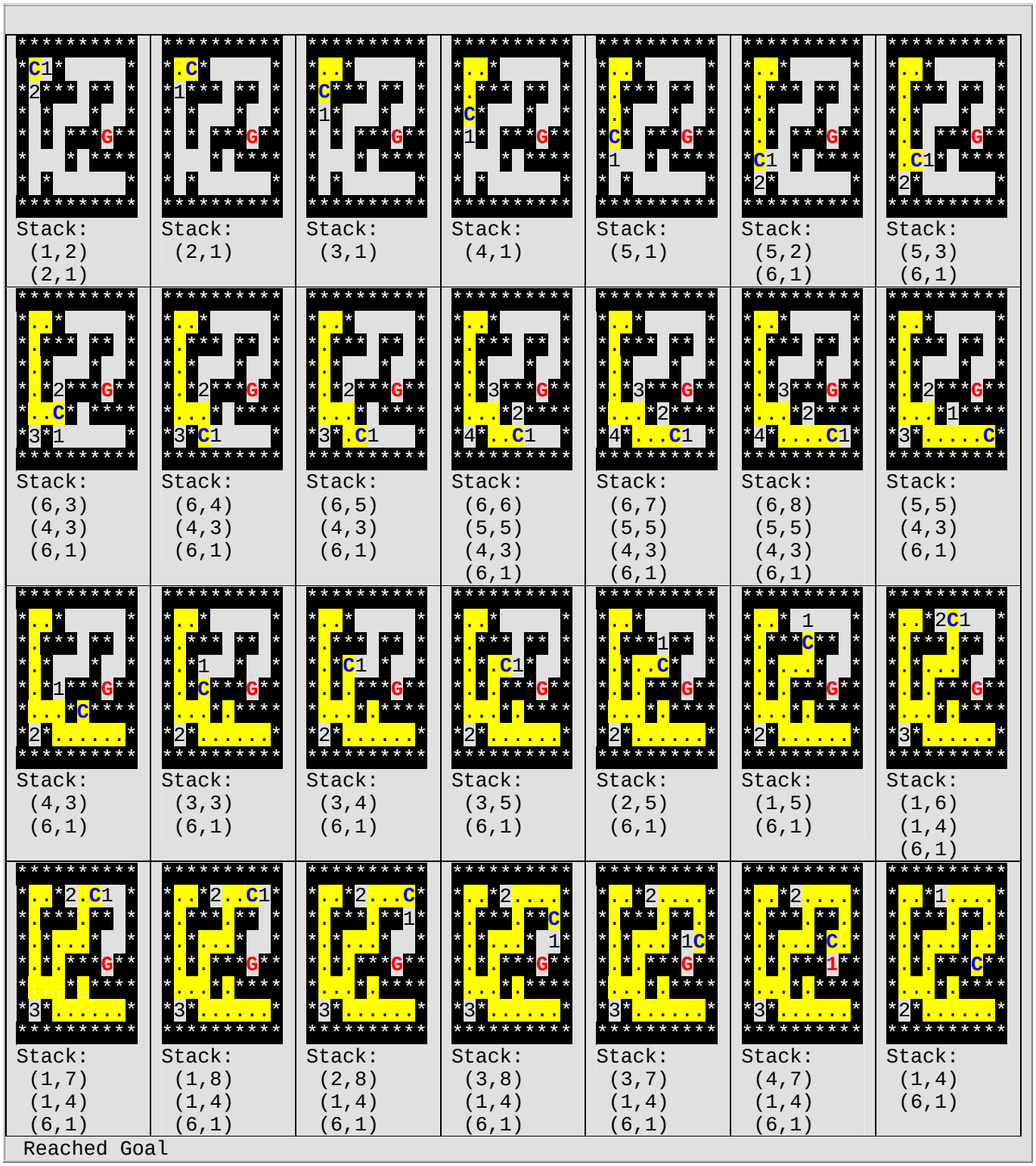
Here is all of the code as it appears in a test program…

```
import java.awt.Point;
import java.util.Stack;

public class MazeTestProgram {
```

```
   static void printMaze(int[][] maze, Point currentLoc, Point goalLoc) {
      for (int row=0; row<maze.length; row++) {
         for (int col=0; col<maze[0].length; col++) {
            if ((currentLoc.x == row) && (currentLoc.y == col))
               System.out.print('C');
            else if ((goalLoc.x == row) && (goalLoc.y == col))
               System.out.print('G');
            else if (maze[row][col] == 1)
               System.out.print('*');
            else if (maze[row][col] == 0)
               System.out.print(' ');
            else
               System.out.print('.');
         }
         System.out.println();
      }
      System.out.println();
   }
   public static void main(String args[]) {
      int[][]         maze = {{1,1,1,1,1,1,1,1,1,1}, {1,0,0,1,0,0,0,0,0,1},
                              {1,0,1,1,1,0,1,1,0,1}, {1,0,1,0,0,0,1,0,0,1},
                              {1,0,1,0,1,1,1,0,1,1}, {1,0,0,0,1,0,1,1,1,1},
                              {1,0,1,0,0,0,0,0,0,1}, {1,1,1,1,1,1,1,1,1,1}};
      Point           currentLoc = new Point(1,1);
      Point           goalLoc = new Point(4,7);  // Here x=rows, y=cols
      Stack<Point>    possibilities = new Stack<Point>();

      possibilities.push(currentLoc);
      while(true) {
         if (possibilities.isEmpty()) {
            System.out.println("Cannot Reach Goal");
            break;
         }
         currentLoc = possibilities.pop();       // get next location
         printMaze(maze, currentLoc, goalLoc);
         maze[currentLoc.x][currentLoc.y] = -1;  // place breadcrumb
         if (currentLoc.equals(goalLoc)) {
            System.out.println("Reached Goal");
            break;
         }
         if (maze[currentLoc.x-1][currentLoc.y] == 0) // above
            possibilities.push(new Point(currentLoc.x-1, currentLoc.y));
         if (maze[currentLoc.x+1][currentLoc.y] == 0) // below
            possibilities.push(new Point(currentLoc.x+1, currentLoc.y));
         if (maze[currentLoc.x][currentLoc.y-1] == 0) // left
            possibilities.push(new Point(currentLoc.x, currentLoc.y-1));
         if (maze[currentLoc.x][currentLoc.y+1] == 0) // right
            possibilities.push(new Point(currentLoc.x, currentLoc.y+1));
      }
   }
}
```

Here is the output when we run the code (placed in a table to save space, read left to right):

The * represent the walls, the small dots . represent the "breadcrumbs" (i.e., visited locations). The C represents the current location and the G represents the goal location. The numbers in the maze from 1 through 4 indicate the locations that are on the stack after each round through the loop. A 1 indicates the top of the stack (i.e., the next to be popped off), while the largest number indicates the bottom of the stack. You will notice how the stack grows and shrinks, with 2 items left on it (i.e., 2 unexplored path portions) after the program completes.



13.4 Sorting Objects

So far, we have examined some **List** classes and we did a couple of examples that used a **Stack**. In each of these cases, we added objects to a collection in some manner that was predictable according to the order that the items were added in. For some data structures, this is not the case. One example of this is the ***PriorityQueue***.

A **PriorityQueue** is a **Queue** in which the elements also maintain a priority. That is, we still add to the back of the queue and remove from the front, but elements with higher priority are automatically shifted closer to the front before lower priority elements.

As a real life example, when we go to the hospital for an "emergency", we wait in line (6 to 8 hours typically). We normally get served in the order that we came in at. However, if someone comes in after us who is bleeding or unconscious, they automatically get bumped up ahead of us since their injuries are likely more serious and demand immediate attention. We may think of a **PriorityQueue** as a ***sorted queue***.

Typical methods for **PriorityQueues** are:

- add(**int** priority, Object x) `// just add(Object x) in JAVA`
- remove()
- isEmpty()

In a **PriorityQueue**, when we add items, they usually get shuffled around inside according to their priority. Therefore, we may not necessarily know the order of the items afterwards … except that they will be in some sort of prioritized order. Consider adding some **Person** objects to an **ArrayList**:

```
import java.util.*;

public class SortTestProgram1 {
      public static void main(String args[]) {
            ArrayList<Person>  people = new ArrayList<Person>();
            people.add(new Person("Pete", "Zaria", 12, 'M', false));
            people.add(new Person("Rita", "Book", 20, 'F', false));
            people.add(new Person("Willie", "Maykit",65, 'M', true));
            people.add(new Person("Patty", "O'Furniture", 41, 'M', false));
            people.add(new Person("Sue", "Permann", 73, 'F', true));
            people.add(new Person("Sid", "Down", 19, 'M', false));
            people.add(new Person("Jack", "Pot", 4, 'M', false));

            for (Person p:  people)
                System.out.println(p);
      }
}
```

The people are displayed in the order that they are added:

```
12 year old non-retired person named Pete Zaria
20 year old non-retired person named Rita Book
65 year old retired person named Willie Maykit
41 year old non-retired person named Patty O'Furniture
73 year old retired person named Sue Permann
19 year old non-retired person named Sid Down
4 year old non-retired person named Jack Pot
```

What would happen if we added them to a priority queue instead ? Assume that we changed the variable type from **ArrayList** to **PriorityQueue** as follows:

```
PriorityQueue <Person>  people = new PriorityQueue<Person>();
```

If we tried running the code, we would get the following **Exception**:

```
java.lang.ClassCastException: Person cannot be cast to java.lang.Comparable
```

The problem is that JAVA does not know how to compare **Person** objects in order to be able to sort them. It is telling us that **Person** must implement the **Comparable** interface. Instead of supplying a priority when we add the objects to the queue, the items are sorted by means of a **Comparator** interface. That means, each object that we store in the **PriorityQueue**, must implement methods **compare()** and **equals()** … which are used determine the sort order (i.e., priority).

So, we should add **implements Comparable<Person>** to the **Person** class definition:

```
public class Person implements Comparable<Person> {
    ...
}
```

Interestingly, the additional **<Person>** at the end of **Comparable** indicates to JAVA that we will only be comparing **Person** objects, not **Person** objects with other types of objects.

Then we can easily write an **equals()** method … we did this previously when we discussed "Code Efficiency". But how do we write a **compare()** method ? It is similar to the **equals()** method in that it takes a single object parameter:

```
public int compareTo(Person  p) { ... }
```

However, you will notice that the return type is not a **boolean**, but an **int** instead. This integer reflects the ordering between the receiver and t he parameter. If a negative value is returned from the method, this informs JAVA that the receiver has higher priority (i.e., comes before in the ordering) than the incoming parameter object.

Likewise, a positive value indicates lower priority and a zero value indicates that they are equal priority.

Lets now give it a try for **Person** objects. If we want to sort by means of their increasing ages (i.e., younger first), this would be the **compareTo()** method:

```
public int compareTo(Person  p) {
    return (this.age – p.age);
}
```

Assume now that we ran the following program:

```
import java.util.*;

public class SortTestProgram2 {
    public static void main(String args[]) {
        PriorityQueue<Person>  people = new PriorityQueue<Person>();
        people.add(new Person("Pete", "Zaria", 12, 'M', false));
        people.add(new Person("Rita", "Book", 20, 'F', false));
        people.add(new Person("Willie", "Maykit",65, 'M', true));
        people.add(new Person("Patty", "O'Furniture", 41, 'M', false));
        people.add(new Person("Sue", "Permann", 73, 'F', true));
        people.add(new Person("Sid", "Down", 19, 'M', false));
        people.add(new Person("Jack", "Pot", 4, 'M', false));

        // Display the queue
        System.out.println("Here is what the queue looks like:");
        for(Person p:  people)
            System.out.println(p);

        // Extract them from the queue
        System.out.println("\nHere are items as extracted from queue:");
        while(!people.isEmpty())
            System.out.println(people.remove());
    }
}
```

Interestingly, the output from the **for** loop is as follows:

```
Here is what the queue looks like:
4 year old non-retired person named Jack Pot
20 year old non-retired person named Rita Book
12 year old non-retired person named Pete Zaria
41 year old non-retired person named Patty O'Furniture
73 year old retired person named Sue Permann
65 year old retired person named Willie Maykit
19 year old non-retired person named Sid Down
```

Notice that the items do not seem sorted at all ! That is because a **PriorityQueue** does not actually sort the items, it simple makes sure that the item at the front of the queue is the one with highest priority. In this case, that is the youngest person … which is indeed at the front of the queue.

To get the items in sorted order, we simply extract them from the queue one at a time as shown in the **while** loop from the code above. Here is the output from the **while** loop:

```
Here are items as extracted from queue:
4 year old non-retired person named Jack Pot
12 year old non-retired person named Pete Zaria
19 year old non-retired person named Sid Down
20 year old non-retired person named Rita Book
41 year old non-retired person named Patty O'Furniture
65 year old retired person named Willie Maykit
73 year old retired person named Sue Permann
```

Indeed, you can see that as we extract the items one at a time, they are sorted. More about these kinds of queue will be discussed in a later course.

What if we wanted to sort the people by their last names ? To do this, we would need to alter the **compareTo()** method to compare names, not ages. Lets make a subclass of **Person** called **AlphaPerson** that has a different **compareTo()** method:

```
public class AlphaPerson extends Person {
    // This is a 5-parameter constructor
    public AlphaPerson(String fn, String ln, int a, char g, boolean r) {
        super(fn, ln, a, g, r);
    }

    // Used to compare Persons by alphabetical order of last names
    public int compareTo(Person  p) {
        return this.lastName.compareTo(p.lastName); // assumes that lastName
    }                                               // is declared protected
}                                                   // in the Person class
```

Notice that the parameter is **Person**, not **AlphaPerson**. This is because **Person** implements the **Comparable<Person>** interface, which specifies type **Person** and **AlphaPerson** inherits from **Person**.

Consider the output then from the following program …

```
import java.util.*;

public class SortTestProgram3 {
    public static void main(String args[]) {
        PriorityQueue<AlphaPerson> people;
        people = new PriorityQueue<AlphaPerson>();

        people.add(new AlphaPerson("Pete", "Zaria", 12, 'M', false));
        people.add(new AlphaPerson("Rita", "Book", 20, 'F', false));
        people.add(new AlphaPerson("Willie", "Maykit",65, 'M', true));
        people.add(new AlphaPerson("Patty", "O'Furniture", 41, 'M', false));
        people.add(new AlphaPerson("Sue", "Permann", 73, 'F', true));
        people.add(new AlphaPerson("Sid", "Down", 19, 'M', false));
        people.add(new AlphaPerson("Jack", "Pot", 4, 'M', false));

        System.out.println("\nHere are items as extracted from queue:");
        while (!people.isEmpty())
            System.out.println(people.remove());
    }
}
```

Here is the output now … notice that they are sorted by their last names:

```
Here are items as extracted from queue:
20 year old non-retired person named Rita Book
19 year old non-retired person named Sid Down
65 year old retired person named Willie Maykit
41 year old non-retired person named Patty O'Furniture
73 year old retired person named Sue Permann
4 year old non-retired person named Jack Pot
12 year old non-retired person named Pete Zaria
```

So, we can decide how to sort the items and make the appropriate **compareTo()** method.

As you will learn in a later course, the **PriorityQueue** is very efficient. However, the above code required us to extract all the items from the queue in order to get them in sorted order … thereby leaving the queue empty. This is often undesirable. Sometimes we just want to leave the elements in the collection and sort them so that we can then use the collection as we normally would, but maintain all items in sorted order.

JAVA provides a nice tool-kit class called **Collections** that contains a bunch of useful methods that we can take advantage of. One of these is a **sort()** method which will sort an arbitrary collection.

Examine the following code to see how easy it is to sort our **ArrayList** of **Person** objects using this **sort()** method …

```
import java.util.*;

public class SortTestProgram4 {
    public static void main(String args[]) {
        ArrayList<Person> people = new ArrayList<Person>();

        people.add(new Person("Pete", "Zaria", 12, 'M', false));
        people.add(new Person("Rita", "Book", 20, 'F', false));
        people.add(new Person("Willie", "Maykit",65, 'M', true));
        people.add(new Person("Patty", "O'Furniture", 41, 'M', false));
        people.add(new Person("Sue", "Permann", 73, 'F', true));
        people.add(new Person("Sid", "Down", 19, 'M', false));
        people.add(new Person("Jack", "Pot", 4, 'M', false));

        Collections.sort(people);     // do the sorting

        for (Person p:  people)
            System.out.println(p);
    }
}
```

The output is as expected with all people sorted by their age.

Of course, we could use **AlphaPerson** to sort them alphabetical instead, if so desired. For the above code to work, we still need to have the **compareTo()** methods written. Hopefully you noticed how easy this **sort()** method is to use.

There is also a class called **Arrays** which has some useful methods for manipulating arrays. For example, if our code had arrays of **Person** objects instead of **ArrayLists**, here is what the code would look like to sort:

```
import java.util.*;

public class SortTestProgram4b {
    public static void main(String args[]) {
        Person[] people = {new Person("Pete", "Zaria", 12, 'M', false),
                  new Person("Rita", "Book", 20, 'F', false),
                  new Person("Willie", "Maykit",65, 'M', true),
                  new Person("Patty", "O'Furniture", 41, 'M', false),
                  new Person("Sue", "Permann", 73, 'F', true),
                  new Person("Sid", "Down", 19, 'M', false),
                  new Person("Jack", "Pot", 4, 'M', false)};

        Arrays.sort(people);     // do the sorting

        for (Person p:  people)
            System.out.println(p);
    }
}
```

There are similar sort methods for the primitive data types, so you can sort simple arrays of numbers such as this:

```
int[] nums = {23, 54, 76, 1, 29, 89, 45, 76};

Arrays.sort(nums);    // do the sorting
```

Interestingly, there are other useful methods in the **Collections** class such as **reverse()**, **shuffle()**, **max()** and **min()**. Can you guess what they do by looking at the output of the following program ?

```
import java.util.*;

public class SortTestProgram5 {
    public static void main(String args[]) {
        ArrayList<Person> people = new ArrayList<Person>();

        people.add(new Person("Pete", "Zaria", 12, 'M', false));
        people.add(new Person("Rita", "Book", 20, 'F', false));
        people.add(new Person("Willie", "Maykit",65, 'M', true));
        people.add(new Person("Patty", "O'Furniture", 41, 'M', false));
        people.add(new Person("Sue", "Permann", 73, 'F', true));
        people.add(new Person("Sid", "Down", 19, 'M', false));
        people.add(new Person("Jack", "Pot", 4, 'M', false));

        System.out.println("The list reversed:");
        Collections.reverse(people);
        for(Person p:  people)
            System.out.println(p);

        System.out.println("\nThe list shuffled:");
        Collections.shuffle(people);
        for(Person p:  people)
            System.out.println(p);

        System.out.println("\nThe list shuffled again:");
        Collections.shuffle(people);
        for(Person p:  people)
            System.out.println(p);

        System.out.println("\nOldest person: " + Collections.max(people));
        System.out.println("Youngest person:" + Collections.min(people));
    }
}
```

Here is the output … was it as you expected? …

```
The list reversed:
4 year old non-retired person named Jack Pot
19 year old non-retired person named Sid Down
73 year old retired person named Sue Permann
41 year old non-retired person named Patty O'Furniture
65 year old retired person named Willie Maykit
20 year old non-retired person named Rita Book
12 year old non-retired person named Pete Zaria

The list shuffled:
12 year old non-retired person named Pete Zaria
4 year old non-retired person named Jack Pot
19 year old non-retired person named Sid Down
20 year old non-retired person named Rita Book
41 year old non-retired person named Patty O'Furniture
73 year old retired person named Sue Permann
65 year old retired person named Willie Maykit

The list shuffled again:
4 year old non-retired person named Jack Pot
73 year old retired person named Sue Permann
19 year old non-retired person named Sid Down
12 year old non-retired person named Pete Zaria
41 year old non-retired person named Patty O'Furniture
20 year old non-retired person named Rita Book
65 year old retired person named Willie Maykit

Oldest person: 73 year old retired person named Sue Permann
Youngest person:4 year old non-retired person named Jack Pot
```

There are additional methods in the **Collections** class. Have a look at the API and see if you find anything useful.

13.5 Removing Duplicates

One more interesting tool for collections that we will look at is one that will allow us to remove duplicates. In the real world, it is often necessary to reduce the amount of data you have by eliminating the *duplicate* information. We have already had some practice at preventing duplicate information when we created the **differentMakes()** method in the **Autoshow** class when we discussed **ArrayLists**:

```
ArrayList<String> differentMakes() {
      ArrayList<String>  answer = new ArrayList<String>();

      for (Car c:  this.cars) {
            if (!answer.contains(c.make))
                  answer.add(c.make);
      }
      return answer;
}
```

Recall that we simply checked (with the **contains()** method) to see whether or not the make of the car was already in the answer collection before we added it. If already there, we did not add it. The code was straight forward. However, sometimes we want to have duplicates in our collections, we just may not want to show them all of the time. For example, a video store might be interested in producing a list of all its unique moves … nothing is gained by listing a move 15 times if the store, for example, has 15 copies of that movie.

Consider the following code which simulates some inventory at a video store. The code makes use of a simple **Movie** object, which contains only one attribute corresponding to its title. The code adds **10** movies from among **5** unique titles … hence many duplicates. The code makes us of **Math.random()** so that the inventory is different each time we run the program.

```
import java.util.*;

public class SetTestProgram1 {
   public static void main(String args[]) {
      Movie[] dvds = {new Movie("Bolt"),
                      new Movie("Monsters Vs. Aliens"),
                      new Movie("Marley & Me"),
                      new Movie("Hotel For Dogs"),
                      new Movie("The Day the Earth Stood Still")};

      ArrayList<Movie>   inventory = new ArrayList<Movie>();

      // Add 20 random movies from the list of dvds
      for (int i=0; i<10; i++) {
         inventory.add(dvds[(int)(Math.random()*5)]);
      }

      for (Movie m:  inventory)
         System.out.println(m);
   }
}
```

```
public class Movie {
    private String  title;

    public Movie(String t) { title = t; }
    public String getTitle() { return title; }
    public String toString() { return "Movie: \"" + title + "\""; }
}
```

Here is an example of the output (will differ each time you run though) …

```
Movie: "Hotel For Dogs"
Movie: "Hotel For Dogs"
Movie: "Hotel For Dogs"
Movie: "The Day the Earth Stood Still"
Movie: "Bolt"
Movie: "Marley & Me"
Movie: "Hotel For Dogs"
Movie: "Marley & Me"
Movie: "Hotel For Dogs"
Movie: "Bolt"
```

Can we adjust the **for** loop so that it only displays unique movies ? No. We would have to do some extra work of making a new list with the duplicates removed. So, we could replace the for **loop** with the following:

```
ArrayList<Movie>  uniqueList = new ArrayList<Movie>();

for (Movie m:  inventory) {
   if (!uniqueList.contains(m))
      uniqueList.add(m);
}
for (Movie m:  uniqueList) {
   System.out.println(m);
}
```

This would produce the following output (according to the earlier results):

```
Movie: "Hotel For Dogs"
Movie: "The Day the Earth Stood Still"
Movie: "Bolt"
Movie: "Marley & Me"
```

However, there is an easier way to do this … by making use of the **Set** classes in JAVA.

A **Set** is a collection that does not allow duplicates. That is, there cannot be two elements $\mathbf{e_1}$ and $\mathbf{e_2}$ such that $\mathbf{e_1.equals(e_2)}$. Any attempt to add duplicate elements is ignored. Sets differ from Lists in that the elements are not kept in the same order as when they were added. Sets are generally unordered, which means that the particular location of an element may change according to the particular set implementation. Typical methods for **Sets** are:

- add(Object x)
- remove(Object x)

There are two Set classes in JAVA. A **HashSet** is a set in which the order of the items is arbitrary, whereas a **TreeSet** keeps the items in sorted order (according to how the **compareTo()** method is written).

Consider what would happen if we changed the inventory from an **ArrayList** to a **HashSet** as follows:

```
HashSet<Movie>   inventory = new HashSet<Movie>();
```

Our code would produce the following output (according to the earlier results):

```
Movie: "Hotel For Dogs"
Movie: "The Day the Earth Stood Still"
Movie: "Bolt"
Movie: "Marley & Me"
```

Notice that the duplicates were removed. The **HashSet** prevented any duplicates from being added. Therefore, we have lost all duplicate copies from our inventory, which can be bad. Perhaps it would be better to only use a **HashSet** when displaying the inventory, so that we don't destroy the duplicate movies. This is easily done by creating an extra **HashSet** variable (**displayList** in this case) and using the **HashSet** constructor that takes a **Collection** parameter:

```
import java.util.*;

public class SetTestProgram2 {
   public static void main(String args[]) {
      Movie[] dvds = {new Movie("Bolt"),
                      new Movie("Monsters Vs. Aliens"),
                      new Movie("Marley & Me"),
                      new Movie("Hotel For Dogs"),
                      new Movie("The Day the Earth Stood Still")};

      ArrayList<Movie>   inventory = new ArrayList<Movie>();

      // Add 10 random movies from the list of dvds
      for (int i=0; i<10; i++) {
         inventory.add(dvds[(int)(Math.random()*5)]);
      }

      System.out.println("Here are the unique movies:");
      HashSet<Movie>   displayList = new HashSet<Movie>(inventory);
      for (Movie m:  displayList)
         System.out.println(m);

      System.out.println("\nHere is the whole inventory:");
      for (Movie m:  inventory)
         System.out.println(m);
   }
}
```

Notice the parameter to the **HashSet** constructor. This constructor will ensure to add all the elements from the inventory collection to the newly create **HashSet**. Then, in the **for** loop, we use this new **HashSet** for display purposes, while the original inventory remains unaltered. Here is the output:

```
Here are the unique movies:
Movie: "The Day the Earth Stood Still"
Movie: "Bolt"
Movie: "Monsters Vs. Aliens"
Movie: "Hotel For Dogs"

Here is the whole inventory:
Movie: "Bolt"
Movie: "Bolt"
Movie: "Bolt"
Movie: "Monsters Vs. Aliens"
Movie: "The Day the Earth Stood Still"
Movie: "Hotel For Dogs"
Movie: "Bolt"
Movie: "Monsters Vs. Aliens"
Movie: "Monsters Vs. Aliens"
Movie: "Bolt"
```

So, it is easy to remove duplicates from any collection … we simply create a new **HashSet** from the collection and it removes the duplicates for us.

However, there is one point that should be mentioned. In the above code, the duplicates movies all represented the same exact object in memory … that is … all duplicates were identical to one another. However, it is more common to have two equal movies which are not identical.

So, consider this code … notice the equal (but not identical) movies:

```
import java.util.*;

public class SetTestProgram3 {
   public static void main(String args[]) {
      Movie[] dvds = {new Movie("Bolt"),
                      new Movie("Monsters Vs. Aliens"),
                      new Movie("Marley & Me"),
                      new Movie("Monsters Vs. Aliens"),
                      new Movie("Hotel For Dogs"),
                      new Movie("Hotel For Dogs"),
                      new Movie("Monsters Vs. Aliens"),
                      new Movie("The Day the Earth Stood Still"),
                      new Movie("The Day the Earth Stood Still"),
                      new Movie("The Day the Earth Stood Still"),
                      new Movie("The Day the Earth Stood Still")};

      ArrayList<Movie>   inventory = new ArrayList<Movie>();
```

```
      // Add 10 random movies from the list of dvds
      for (int i=0; i<10; i++) {
         inventory.add(dvds[(int)(Math.random()*11)]);
      }

      System.out.println("Here are the unique movies:");
      HashSet<Movie>   displayList = new HashSet<Movie>(inventory);
      for (Movie m:  displayList)
         System.out.println(m);

      System.out.println("\nHere is the whole inventory:");
      for (Movie m:  inventory)
         System.out.println(m);
   }
}
```

Here is the result:

```
Here are the unique movies:
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "Bolt"
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
Movie: "Monsters Vs. Aliens"

Here is the whole inventory:
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
Movie: "The Day the Earth Stood Still"
Movie: "Monsters Vs. Aliens"
Movie: "Bolt"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
```

Notice that there are many duplicates still in the **Set**. What if we were to add the following **equals()** method to the **Movie** class:

```
public boolean equals(Object obj) {
      if (!(obj instanceof Movie)) return false;
      return title.equals(((Movie)obj).title);
}
```

Logically, that should solve the problem. However, it does not quite.

As it turns out, in JAVA, **Sets** make use of a programming technique called ***hashing***. Hashing is used as a way of quickly comparing and sorting objects because it quickly identifies objects that cannot be equal to one another, without needing to go deep down inside the object to make comparisons. For example, if you had an apple and a pineapple, they are clearly not equal. You need not compare them closely because a simple quick glance tells you that they are not the same.

In real life, hashing is used by post offices when sorting mail at various levels. First, they look at the destination country and make two piles … domestic mail vs. international mail. That is a quick "hash" in that the postmen do not need to examine any further details at that time … such as street names and recipient names etc… Then they hash again later by using the *postal code* to determine "roughly" and "quickly" the area of a city that your mail needs to be delivered to. This allows them to make a pile of mail for all people living in the same area. At each level of "sorting" the mail (i.e., country, city, postal code, street), the postmen must make a quick decision as to which pile to place the mail item into. This quick decision is based on something called a ***hash function*** (or ***hash code***).

In JAVA, for **Sets** to work properly, we must also write a **hashCode()** method for our objects. These methods return an **int** which represents the "pile" that the object belongs to. Similar objects will have similar hash codes, and therefore end up in the same "pile". Here is the **hashCode()** method for our **Movie** object:

```
public int hashCode() {
    return title.hashCode();
}
```

It must be **public**, return an **int** and have no parameters. The code simply returns the hash code of the title string for the movie. We do not wish to go into details here as to "how" to produce a proper hash code. Instead, let us simply use this "rule of thumb": the hash code for our objects should return a sum of all the hash codes of its attributes. If an attribute is a primitive, just convert it to an integer in some way and use that value in the **hashCode()** method's total value.

By adding the above method, therefore, the **HashSet** will work properly to eliminate the duplicates. You will notice however, that the **HashSet** does not sort the items. Also, the items don't even appear in the order that they were added. Instead, the order seems somewhat random and arbitrary.

If you want the items in sorted order, you can use a **TreeSet** instead of a **HashSet**:

```
TreeSet<Movie>   displayList = new TreeSet<Movie>(inventory);
```

Of course, as we did with **PriorityQueues**, we will need to make sure that our **Movie** class implements the **Comparable<Movie>** interface and thus has a **compareTo()** method. Here is the completed **Movie** class that will work with both **HashSet** and **TreeSet**:

```
public class Movie implements Comparable<Movie> {
    private String  title;

    public Movie(String t) { title = t; }
    public String getTitle() { return title; }
    public String toString() { return "Movie: \"" + title + "\""; }

    public boolean equals(Object obj) {
        if (!(obj instanceof Movie)) return false;
        return title.equals(((Movie)obj).title);
    }

    public int hashCode() {
        return title.hashCode();
    }

    public int compareTo(Movie m) {
        return title.compareTo(m.title);
    }
}
```

Here is the output from our **SetTestProgram3** when using **TreeSet** instead of **HashSet**:

```
Here are the unique movies:
Movie: "Bolt"
Movie: "Hotel For Dogs"
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
Movie: "The Day the Earth Stood Still"

Here is the whole inventory:
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
Movie: "Hotel For Dogs"
Movie: "Monsters Vs. Aliens"
Movie: "Marley & Me"
Movie: "Bolt"
```

Notice the sorted order of the movies from the **TreeSet**.

TO BE CONTINUED...