Java 2 Stacks and Queues

- Queue is a First-In-First-Out (FIFO) data structure.
  - Dequeue (remove); Enqueue (add)

- Stack is a Last in, first out (LIFO) abstract data type and data structure.
  - Push (add); Pop (remove)

```
public class ArrayStack<T> implements StackInterface<T> {
	private T[] stack;
	private int top;

	public ArrayStack(int capacity) {
		stack = (T[]) new Object[capacity];
		top = 0;
	}

	//These will all be O(1)//
	pubic void push(T element) {
		if (size() == stack.length)
			resize(stack.length*2);
		stack[top] = element;
		top++;
	}
	public T pop() {
		if (isEmpty()) return null;

		top—; T result = stack[top];
		return result;
	}
	public T peek() {…}
	public int size() {…}
	public boolean isEmpty() {…}
	public Iterator<T> iterator() {…}
}
```

- In an ArrayQueue shifting is required in order to Dequeue elements
  - Bad because of constant time that cannot be amortized.
  - If we utilize a Circular Queue we can reduce this time complexity.
  - You will need to increment rear on every enqueue and increment front on every dequeue
  - This allows the elements of the array to circle back on itself.

Applications

- Stack Machines
  - We would **need** to PARSE the entire expression
  - We can circumvent this by using the concept of a Shunting Yard algorithm.
  - This can be further simplified by using Postfix notation.

- Maze Search
  - Solve 2D mazes
  - We will represent a maze as a 2D grid of positions
  - Restrict motion to 4 degrees of motion

Let's assume this is our goal: Find the finish position

- Strategy:

```
while (the current position is not the finish and the stack is not empty) {
```

choose an adjacent open position and move there
}

- This is simple but not usable
- New Strategy

while (the current position is not the finish) {
if (there is an adjacent place I haven't been)

- move to the position
- mark it visited

else
backtrack
}

- We will solve the maze, albeit slowly
- This is a **depth-first** search strategy
  - We explore paths in the amaze as deeply as possible. When we reach a dead end, only then do we backtrack to the closest branch.
- **How to implement forward and backward moves?**
  - Use a stack:
    - top = where we are
    - push = move forward
    - pop = move backward
  - What does the stack contain while the algorithm is running?
    - The most recent place you've been
  - What does the stack contain when the algorithm?
    - The path with no dead ends
  - What is the maze has no solution?
    - We need to modify the loop bound to take this into account.
    - We find the finish position.
    - We have no more viable moves
- Alternative to DFS? A **Breadth-first** search strategy
  - We explore the immediate neighborhood first, then branch out.
  - That is, explore the current position's neighbors, then their neighbors, the their neighbors…
- DFS
  - If we want just any solution.
- BFS
  - If we want the quickest solution possible.