CSCI1913 Midterm 2 Cheat Sheet

- How *this* work

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

but it could have been written like this:

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 1, 1);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

- How to throw Java Exception

```
public Object pop() {
    Object obj;

    if (size == 0) {
        throw new IllegalStateException(“,,”);
    }

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

- What the Java pointer ***null*** is
- How the Java equals methods work.
- How the ‘==’ work
- When to use *equals* and ‘==’
- How Java ***toString*** method works
- How inheritance works with Java classes

<Lab5>

```
class Polygon {
           private int[] sideLengths;

             public Polygon(int sides, int ... lengths)
             {
               int index = 0;
               sideLengths = new int[sides];
               for (int length: lengths)
               {
                 sideLengths[index] = length;
                 index += 1;
               }
             }

             public int side(int number)
             {
               return sideLengths[number];
             }

             public int perimeter()
             {
               int total = 0;
               for (int index = 0; index < sideLengths.length;
index += 1)
               {
                 total += side(index);
               }
               return total;
             }
}

class Rectangle extends Polygon {

            public Rectangle(int side1, int side2) {
                        super(4, side1, side2, side1,
side2);
            }

            public int area() {
                        return side(0)*side(1);
            }
}

class Square extends Rectangle {

            public Square(int side1) {
                        super(side1, side1);
            }

}

class Shapes {

            public static void main(String[] args) {

                        Rectangle wreck = new Rectangle(3, 5);

                   System.out.println(wreck.side(0));      //  3
1 point.
                   System.out.println(wreck.side(1));      //  5
1 point.
                   System.out.println(wreck.side(2));      //  3
1 point.
                   System.out.println(wreck.side(3));      //  5
1 point.
                   System.out.println(wreck.area());       //  15
1 point.
                   System.out.println(wreck.perimeter());  //  16
1 point.

                   Square nerd = new Square(7);

                   System.out.println(nerd.side(0));       //  7
1 point.
                   System.out.println(nerd.side(1));       //  7
1 point.
                   System.out.println(nerd.side(2));       //  7
1 point.
                   System.out.println(nerd.side(3));       //  7
1 point.
                   System.out.println(nerd.area());        //  49
1 point.
                   System.out.println(nerd.perimeter());   //  28
1 point.

            }
}
```

```
<Lab6>
class BinaryVsLinear
{

    private static int linearSearch(int key, int[] array)
    {
        for (int i=0; i < array.length; i+=1)
        {
            if (array[i] == key)
            {
                                                    return i + 1;
                                        }
        }
        return array.length;
```

```
    }

    private static int binarySearch(int key, int[] array)
    {
        int left = 0;
        int right = array.length -1;
        int mid;
        int count = 0;

        while (left <= right)
        {
            mid = (left + right) / 2;

            if (array[mid] < key)
            {
                count += 1;
                left = mid + 1;
            }
            else if (array[mid] > key)
            {
                count += 2;
                right = mid - 1;
            }
            else
            {
                count += 2;
                return count;
            }
        }

        return count;
    }

    public static void main(String[] args)
    {
        for (int length = 1; length <= 30; length += 1)
        {
            int[] array = new int[length];
            for (int index = 0; index < length; index += 1)
            {
                array[index] = index;
            }

            double linearTotal = 0.0;
            double binaryTotal = 0.0;
            for (int element = 0; element < length; element +=
1)
            {
                linearTotal += linearSearch(element, array);
                binaryTotal += binarySearch(element, array);
            }

            double linearAverage = linearTotal / length;
            double binaryAverage = binaryTotal / length;
            System.out.println(length + " " + linearAverage + "
" + binaryAverage);
        }
    }
}

<Lab7>
class Map<Key,Value>
{
            private Key[] keys;
            private Value[] values;
            private int count;
            private int increment;



            public Map(int length)
            {
                        count = 0;
                        if (length <= 0)
                        {
                                    throw new
IllegalArgumentException("Length must be larger than 0");
                        }

                        this.keys = (Key[]) new Object[length];
                        this.values = (Value[]) new
Object[length];
            }

            private int where(Key key)
            {
                        for (int i=0; i < count; i++)
                        {
                                    if (isEqual(key, keys[i]))
                                    {
                                                return i;
                                    }
                        }
                        return -1;
            }

            private boolean isEqual(Key leftKey, Key rightKey)
            {
                                    if(leftKey == null &&
rightKey == null)
                                    {
                                                return true;
                                    }
                                    else if (leftKey == null ||
rightKey == null)
                                    {
                                                return false;
                                    }
                                    else
                                    {
                                                return
leftKey.equals(rightKey);
                                    }

                                    //~ if(leftKey != null &&
rightKey != null)
                                    //~ {
                                                //~ return
leftKey.equals(rightKey);
                                    //~ }
                                    //~ else
                                    //~ {
                                                //~ return
leftKey == rightKey;
                                    //~ }
            }

            public Value get(Key key)
            {
                        int index = where(key);
                        if (index >= 0)
                        {
                                    return values[index];
                        }
                        else
                        {
                                    throw new
IllegalArgumentException("No key" + key + "found in Map");
                        }
            }

            public boolean isIn(Key key)
            {
                        return where(key) != -1;
            }

            public void put(Key key, Value value)
            {
                        int index = where(key);
                        if (index >= 0)
                        {
                                    values[index] = value;
                                    return;
                        }

                        if (count < keys.length)
                        {
                                    keys[count] = key;
                                    values[count] = value;
                                    count++;
                        }
                        else
                        {
                                    throw new
IllegalStateException("The dictionary is full");
                        }
            }
}

class Hogwarts
{

//  MAIN. Make an instance of MAP and test it.

  public static void main(String [] args)
  {
    Map<String, String> map;

    try
    {
      map = new Map<String, String>(-5);
    }
    catch (IllegalArgumentException ignore)
    {
      System.out.println("No negatives");       //  No negatives
2 points.
    }

    map = new Map<String, String>(5);

    map.put("Harry",     "Ginny");
    map.put("Ron",       "Lavender");
    map.put("Voldemort", null);
    map.put(null,        "Wormtail");
```

```
    System.out.println(map.isIn("Harry"));      //  true
2 points.
    System.out.println(map.isIn("Ginny"));      //  false
2 points.
    System.out.println(map.isIn("Ron"));        //  true
2 points.
    System.out.println(map.isIn("Voldemort"));  //  true
2 points.
    System.out.println(map.isIn(null));         //  true
2 points.
    System.out.println(map.isIn("Joanne"));     //  false
2 points.

    System.out.println(map.get("Harry"));       //  Ginny
2 points.
    System.out.println(map.get("Ron"));         //  Lavender
2 points.
    System.out.println(map.get("Voldemort"));   //  null
2 points.
    System.out.println(map.get(null));          //  Wormtail
2 points.

    try
    {
      System.out.println(map.get("Joanne"));
    }
    catch (IllegalArgumentException ignore)
    {
      System.out.println("No Joanne");          //  No Joanne
2 points.
    }

    map.put("Ron",   "Hermione");
    map.put("Albus", "Gellert");
    map.put(null,    null);

    System.out.println(map.isIn(null));         //  true
2 points.
    System.out.println(map.isIn("Albus"));      //  true
2 points.

    System.out.println(map.get("Albus"));       //  Gellert
2 points.
    System.out.println(map.get("Harry"));       //  Ginny
2 points.
    System.out.println(map.get("Ron"));         //  Hermione
2 points.
    System.out.println(map.get("Voldemort"));   //  null
2 points.
    System.out.println(map.get(null));          //  null
2 points.

    try
    {
      map.put("Draco", "Pansy");
    }
    catch (IllegalStateException minnesota)
    {
      System.out.println("No Draco");           //  No Draco
2 points.
    }
  }
}

<Lab8>
class RunnyStack<Base>
{
            private Run top;
            private int totalRuns;
            private int totalDepth;

            private class Run
            {
                        private Base object;
                        private int length;
                        private Run next;

                        private Run(Base object, Run next)
                        {
                                    this.object = object;
                                    this.length = 1;
                                    this.next = next;
                        }
            }

            public RunnyStack()
            {
                        totalDepth = 0;
                        totalRuns = 0;
                        top = null;
            }

            public int depth()
            {
                        return totalDepth;
            }

            public boolean isEmpty()
            {
                        return top == null;
            }


            public Base peek()
            {
                        if (isEmpty())
                        {
                                    throw new
IllegalStateException("Stack is Empty");
                        }
                        else
                        {
                                    return top.object;
                        }
            }

            public void pop()
            {
                        if (isEmpty())
                        {
                                    throw new
IllegalStateException("Stack is Empty");
                        }
                        else if (top.length > 1)
                        {
                                    totalDepth--;
                                    top.length--;
                        }
                        else
                        {
                                    totalDepth--;
                                    top.length--;
                                    totalRuns--;
                                    top = top.next;
                        }
            }

            public void push (Base object)
            {
                        if (isEmpty())
                        {
                                    top = new Run(object, null);
                                    totalDepth++;
                                    totalRuns++;
                        }
                        else if (IsEqual(object,top.object))
                        {
                                    totalDepth++;
                                    top.length++;
                        }
                        else
                        {
                                    top = new Run(object,top);
                                    totalDepth++;
                                    totalRuns++;
                        }
            }

            public int runs()
            {
                        return totalRuns;
            }

            private boolean IsEqual(Base a, Base b)
            {
                        if(a != null && b != null)
                        {
                                    return a.equals(b);
                        }
                        else
                        {
                                    return a == b;
                        }
            }

}

class Camembert
{
  public static void main(String [] args)
  {
    RunnyStack<String> s = new RunnyStack<String>();

    System.out.println(s.isEmpty());         //  true       1
point
    System.out.println(s.depth());           //  0          1
point
    System.out.println(s.runs());            //  0          1
point

    try
    {
      s.pop();
```

```
    }
    catch (IllegalStateException ignore)
    {
      System.out.println("No pop");          //  No pop     1
point
    }

    try
    {
      System.out.println(s.peek());
    }
    catch (IllegalStateException ignore)
    {
      System.out.println("No peek");         //  No peek    1
point
    }

    s.push("A");
    System.out.println(s.peek());            //  A          1
point
    System.out.println(s.depth());           //  1          1
point
    System.out.println(s.runs());            //  1          1
point

    System.out.println(s.isEmpty());         //  false      1
point

    s.push("B");
    System.out.println(s.peek());            //  B          1
point
    System.out.println(s.depth());           //  2          1
point
    System.out.println(s.runs());            //  2          1
point

    s.push("B");
    System.out.println(s.peek());            //  B          1
point
    System.out.println(s.depth());           //  3          1
point
    System.out.println(s.runs());            //  2          1
point

    s.push("B");
    System.out.println(s.peek());            //  B          1
point
    System.out.println(s.depth());           //  4          1
point
    System.out.println(s.runs());            //  2          1
point

    s.push("C");
    System.out.println(s.peek());            //  C          1
point
    System.out.println(s.depth());           //  5          1
point
    System.out.println(s.runs());            //  3          1
point

    s.push("C");
    System.out.println(s.peek());            //  C          1
point
    System.out.println(s.depth());           //  6          1
point
    System.out.println(s.runs());            //  3          1
point

    s.pop();
    System.out.println(s.peek());            //  C          1
point
    System.out.println(s.depth());           //  5          1
point
    System.out.println(s.runs());            //  3          1
point

    s.pop();
    System.out.println(s.peek());            //  B          1
point
    System.out.println(s.depth());           //  4          1
point
    System.out.println(s.runs());            //  2          1
point

    s.pop();
    System.out.println(s.peek());            //  B          1
point
    System.out.println(s.depth());           //  3          1
point
    System.out.println(s.runs());            //  2          1
point

    s.pop();
    s.pop();
    System.out.println(s.peek());            //  A          1
point
    System.out.println(s.depth());           //  1          1
point
    System.out.println(s.runs());            //  1          1
point

    s.pop();
    System.out.println(s.isEmpty());         //  true       1
point
    System.out.println(s.depth());           //  0          1
point
    System.out.println(s.runs());            //  0          1
point

    try
    {
      System.out.println(s.peek());
    }
    catch (IllegalStateException ignore)
    {
      System.out.println("No peek");         //  No peek    1
point
    }
  }
}
```

- Big O types
  - O(1)>O(logn)>O(n)>O(nlogn)>O(n2)>O(Nk)

1) O(route(n))

```
A()
{
i=1,s=1;
while(s<=n)
{
i++;
s=s+i;
}
}

A()
{
i=1;
for(i=1,i<=route(n);i++)
{
print(ss)
}
}
```

2) O(n(square))

```
A()
{
int i,j,k,n;
for(i=1;i<=n;i++)
{
for(j=1;j<=I;j++)
{
for(k=1;k<=100;k++)
{
print(“dad”)
}
}
}
}
```

3) O(logn)

```
A()
{
for(i=1;i<n;i=i*2)
{
print(“dad”)
}
}


A()
{
while(n>1)
{
n=n/2;
}
}
```

4) O(n(logn)2)

```
A()
{
int i,j,k;
for(i=n/2;i<=n;i++) : n/2
for(j=1;j<=n;j=2*j) : logn
for(k=1;k<=n;k=k*2) : logn
}
```

5) O(nlogn)

```
A()
{
for(i=1;i<=n;i++)
for(j=1;j<=n;j+j+i)
}
```

Answers to CSci 1913 Midterm Two
James Moen
16 Nov 15

For questions 1 and 2: Students don't need to explain their reasoning. Just
a correct answer gets full credit. If they have a right answer but the wrong
explanation, then they get full credit. If they have the wrong answer but the
right explanation, then they should get partial credit.

---

1. (5 points.) Method B, which performs O(n log n) comparisons, does fewer
comparisons than method A, which performs O(n²) comparisons. Students need not
explain their reasoning. However, they might have noted that log n grows more
slowly than n, so n × log n should grow more slowly than n × n. They might also
look this up on a graph like the one in the textbook, or even drawn a graph
themselves.

Saying method B is more efficient gets 5 points. However please try to give
partial credit for partially correct explanations that accompany wrong answers.

---

2. (5 points.) The worst case occurs when there are no duplicate elements, so
both the inner and outer loops run to completion. If the array has n elements,
then the equality comparison is executed O(n²) times.

In the following table, the first column is the iteration of the outer loop
(so 1 means the first time it is executed, 2 means the second time, etc.) The
second column is the number of times the inner loop is executed each time.

outer	inner
1	n - 1
2	n - 2
3	n - 3
4	n - 4
⋮	⋮
n - 2	2
n - 1	1
n	0

Adding up the entries in the second column gives (n - 1) n / 2, by the usual
formula for the sum of the first n natural numbers, which is O(n²).

Saying O(n²) gets 5 points. However please try to give partial credit for
partially correct explanations that accompany wrong answers.

3a. (5 points.) Note that the method swap must NOT change the stack if it
throws an exception. I can think of three different ways to do this:

```
public void swap()
{
  if (count < 2)
  {
    throw new IllegalStateException("Not enough elements.");
  }
  else
  {
    Base b1 = peek();  //  b1 was on top.
    pop();
    Base b2 = peek();  //  b2 was under b1.
    pop();
    push(b1);          //  Now b1 is under b2.
    push(b2);          //  And b2 is on top.
  }
}
```

```
public void swap()
{
  if (count < 2)
  {
    throw new IllegalStateException("Not enough elements.");
  }
  else
  {
    Base b1 = objects[count - 1];  //  b1 was on top.
    count -= 1;
    Base b2 = objects[count - 1];  //  b2 was under b1.
    count -= 1;
    objects[count] = b1;           //  Now b1 is under b2.
    count += 1;
    objects[count] = b2;           //  And b2 is on top.
    count += 1;
  }
}
```

```
public void swap()
{
  if (isEmpty())
  {
    throw new IllegalStateException("Not enough elements.");
  }
  else
  {
    Base b1 = peek();    //  b1 was on top.
    pop();
    if (isEmpty())       //  Was b1 the only element?
    {
      push(p1);          //  Put b1 back.
      throw new IllegalStateException("Not enough elements.");
    }
    else
    {
      Base b2 = peek();  //  b2 was under b1.
      pop();
      push(b1);          //  Now b1 is under b2.
      push(b2);          //  And b2 is on top.
    }
  }
}
```

1 point for testing count < 2 somehow.
1 point for correctly throwing the exception if count < 2.
1 point for popping b1 and b2 off the stack.
1 point for pushing b1 back first.
1 point for pushing b2 back after b1, so b2 is on top.

3b. (10 points.) Here the trick is to make the new stack the same length as
this one, and to push elements on the new stack in reverse order from how they
appear in objects. The first element on the old stack is at count - 1, if it
exists. Here are two possible answers. Note that Stack can see private members
of other instances of Stack.

```
public Stack<Base> duplicate()
{
  Stack<Base> newStack = new Stack<Base>(objects.length);
  for (int index = 0; index < count; index += 1)
  {
    newStack.push(objects[index]);
  }
  return newStack;
}
```

```
public Stack<Base> duplicate()
{
  Stack<Base> newStack = new Stack<Base>(objects.length);
  newStack.count = count;
  for (int index = 0; index < count; index -= 1)
  {
    newStack.objects[index] = objects[index];
  }
  return newStack;
}
```

1 point for a local variable newStack whose type is Stack<Base> (not Stack).
1 point for initializing it to a new instance of Stack<Base> (not Stack).
1 point for initializing index correctly in a loop.
1 point for testing index correctly in a loop.
1 point for incrementing or decrementing index correctly in a loop.

2 point for correctly copying elements from objects into the new stack.
2 point for correctly copying count into the new stack.
1 point for returning the local variable newStack.

The loop must visit elements of objects in the correct order, but
that "correct" will vary depending on how each student's duplicate method
works.

4. (5 points.) The method wuhDuzitDu traverses the list first and returns a
copy of that list, but the elements of the copy are in reverse order. It leaves
first set to null. Here are some possible answers:

They get all 5 points if they say it makes a copy of first, with the copy in
reverse order. Other things that give them partial credit, and the number of
points they get for saying each thing, are:

4 points. It makes a copy of first (without mentioning reverse order).
1 point. It traverses or visits the Nodes in first.
1 point. It leaves first set to null.
1 point. It returns a list.
1 point. It pushes a Node on temp.
2 points. It makes a linked list of Nodes.

Students may draw a picture of the copy, or that shows first set to null, or
both. If the picture indicates they understand the method, then give them full
credit. Try hard to award partial credit where possible.

```
// Sequence
class Sequence
{
		private static final int defaultsize = 10;
		private Object[] objects;
		private int count;

		public Sequence (int size)
		{
				objects = new Object[size];
				count = 0;
		}
		public Sequence ()
		{
				this (defaultsize);
		}

		public void add(Object object)
		{
				if (count >= object.length)
				{
						Object[] newObejct = new Object[objects.length + k];
						for (int index=0; index < object.length; index +=1)
						{
								newObject[index] = object[index];
						}
						Object = newObject;
				}
				objects[count] = object;
				count += 1;
		}

		public Object get(int index)
		{
				if( 0<=index && index < count)
				{
						return objects[index];
				}
				else
				{
						throw new IllegalArgumentException("No Found");
				}
		}

		public boolean IsEmpty()
		{
				return count == 0;
		}

		public boolean IsFull()
		{
				return count == object.length;
		}

}

// Class 를 <Base>로 제너릭하게 만드는경우

class Sequence <Base>
{
		private Base[] objects;
		private int count;

		public sequence (int size)
		{
				objects = new Base[size]; // 이거는 틀린 방!
				objects = (Base[]) new object [size];
				count = 0;
		}

		public void add(Object object)
		{
				if (count >= object.length)
				{
						Object[] newObejct = (Base[]) new Object[objects.length + k];
						for (int index=0; index < object.length; index +=1)
						{
								newObject[index] = Object[index];
						}
						Object = newObject;
				}
				objects[count] = object;
				count += 1;
		}

		public int find(Base object)
		{
				if (object==null)
				{
						for (int index=0; index<count; index+=1)
						{
								if (obejct == null)
								{
										return index;
								}
						}
				}
				else
				{
						for (int index=0; index<count; index+=1)
						{

		if(object.equals(objects[index])
								{
										return index;
								}
						}
						return -1;
		}

// Best = O(1)
// Worst = O(n)
// index 를 주고 Remove 하는 경
		public Base remove (int index)
		{
				if(0<=index && index<count)
				{
						Base object = objects[index];
						while (index < count-1)
						{
								objects[index] = objects[index+1];
								index += 1;
						}
						count -= 1;
						object[count] = null;
						return object;
				}
				else
				{
						throw new IllegalArgumentException("Bad index");
				}
		}

// object 를 주고 Remove 하는 경우
// Worst: O(n)
		public void remove(Base object)
		{
				int index = find(object);
				if (index>0)
				{
						remove(index);
				}
		}
}

//tostring

  public String toString()
  {
    StringBuilder builder = new StringBuilder();
    builder.append('[');
    if (count > 0)
    {
      builder.append(objects[0]);
      for (int index = 1; index < count; index += 1)
      {
        builder.append(", ");
        builder.append(objects[index]);
      }
    }
    builder.append(']');
    return builder.toString();
  }
}


// Stack using Array
class Stack <Base>
{
		private Base[] objects;
		private int count;

		public Stack(int size)
		{
				objects = (Base[]) new object[size];
				count = 0;
		}

		public boolean isEmpty()
		{
				return count == 0;
		}

		public boolean isFull()
		{
				return count >= objects.length;
		}
```

```
	public void push(Base object)
	{
		if (isFull())
		{
			throw new IllegalStateException
("Stack is Full");
		}
		else
		{
			objects[count]=object;
			count += 1;
		}
	}

	public void pop()
	{
		if (isEmpty())
		{
			throw new IllegalStateException("Stack
is Empty");
		}
		else
		{
			count -= 1;
			objects[count] = null;
		}
	}

	public Base peek()
	{
		if(isEmpty())
		{
			throw new IllegalStateException("Stack
is Empty");
		}
		else
		{
			return objects[count-1];
		}
	}
}

// Stack using Linkedlist
// Worst: O(1)

class Stack <Base>
{
	private class Node
	{
		private Base value;
		private Node next;
		private Node(Base value, Node next)
		{
			this.value = value;
			this.next = next;
		}
	}

	private Node top;

	public Stack()
	{
		top = null;
	}

	public boolean isEmpty()
	{
		return top == null;
	}

	public boolean isFull()
	{
		return false;
	}

	public void push (Base base)
	{
		top = new Node(base,top);
	}

	public void pop()
	{
		if(isEmpty())
		{
			throw new IllegalStateException("Empty
Stack");
		}
		else
		{
			top = top.next;
		}
	}

	public Base peek()
	{
		if(isEmpty())
		{
			throw new IllegalStateException("Empty
Stack");
		}
		else
		{
			return top.next;
		}
	}
}

// #1 Queue using Linked-list
// Worst time: O(1)
class Queue <Base>
{
	private class Node
	{
		private Base object;
		private Node next;
		private Node (Base object, Node next)
		{
			this.object=object;
			this.next=next;
		}
	}

	private Node front;
	private Node rear;

	public Queue()
	{
		front = null;
		rear = null;
	}

	public boolean isEmpty()
	{
		return front == null && rear == null;
	}

	public void enqueue(Base object)
	{
		if(isEmpty())
		{
			front = new Node(object,null);
			rear = new Node(object,null);
		}
		else
		{
			rear.next = new Node(object,null);
			rear = rear.next;
		}
	}

	public Base dequeue() // combine "pop" and "peek"
	{
		if (isEmpty())
		{
			throw new
IllegalStateException("MTQ");
		}
		else // Make temp to store A
		{
			Base temp= front.object;
			front = front.next;
			if (front == null) // 이것이 Special
case
			{
				rear = null;
			}
			return temp;
		}
	}
}

// #2 Queue using Array (Doughnut Queue)
// Worst Time: O(1)

class Queue <Base>
{
	if (size > 0)
	{
		front = 0;
		rear = 0;
		objects = (Base[]) new Object[size];
	}
	else
	{
		throw new IllegalArgumentException("Bad size");
	}

	public boolean isEmpty()
	{
		return front == rear;
	}

	public void enqueue(Base object)
	{
		int nextRear = (rear+1)%objects.length;
		if(nextRear == front)
		{
			throw new IllegalStateException("Queue
full");
		}
		else
		{
			rear = nextRear;
			objects[rear]=object;
		}
	}

	public boolean isFull()
	{
		return front == (rear+1)%objects.length;
	}

	public Base dequeue()
	{
		if(isEmpty()) // front == rear
		{
			throw new
IllegalStateException("MTQ");
		}
		else
		{
			front = (front+1)%objects.length;
			Base temp = objects[front];
			objects[front] = null;
			return temp;
		}
	}
}


// Polish Expression Evaluator  코딩 시험에 나올 수 있다
1. Start with an empty stack
2. Move left to right across the expression, visiting each symbol
3. If we visit an argument, then push it
4. If we visit an operator, then pop its argument, do operation and push the result
5. Continue, until al symbols are visited
6. Worst time: O(n)


public class Solution {
    public int evalRPN(String[] tokens) {

        int returnValue = 0;

        String operators = "+-*/";

        Stack<String> stack = new Stack<String>();

        for(String t : tokens){
            if(!operators.contains(t)){
                stack.push(t);
            }else{
                int a = Integer.valueOf(stack.pop());
```

```
                int b = Integer.valueOf(stack.pop());
                int index = operators.indexOf(t);
                switch(index){
                    case 0:
                        stack.push(String.valueOf(a+b));
                        break;
                    case 1:
                        stack.push(String.valueOf(b-a));
                        break;
                    case 2:
                        stack.push(String.valueOf(a*b));
                        break;
                    case 3:
                        stack.push(String.valueOf(b/a));
                        break;
                }
            }
        }

        returnValue = Integer.valueOf(stack.pop());

        return returnValue;

    }
}
```