6.170 Recitation 3

```
6.170 - Recitation 3 Topics Covered
```

- Abstraction Functions
- Representation Invariants

First Implementation

Recall the Command Line History example from Lecture 7 (last Wednesday). In this example we implemented a class, `CommandHistory` which allows a program to execute commands, and then perform **undo** and **redo** operations on them. The behaviors we would like a `CommandHistory` object to perform would be to `issue()` a `Command` for execution. To allow us to `undo()` and `redo()` any `Command`s we issued, and to let us know if we `canUndo()` or if we `canRedo()`.

```
public class CommandHistory {
  public void issue(Command c) {...}
  public void undo() {...}
  public void redo() {...}
  public boolean canUndo() {...}
  public boolean canRedo () {...}
}
```

Abstractly, a `CommandHistory` object must maintain two sequential lists of `Command`s that refer to the set of `Command`s you can undo, and the set of `Commands`s you can redo.

In lecture, Prof. Jackson discussed two ways to implement this class, either using `java.util.Stack`s or using a single circular array. While the `Stack` based implementation was relatively straightforward, the array based implementation was more complicated, and had skeleton code of the form:

```
public class CommandHistory {
  int MAX_HISTORY = 200;
  Command[] commands = new Commands[MAX_HISTORY];
  int undoLimit = ?;
  int nextCommand = ?;
  int redoLimit = ?;

  public void issue(Command c) {
  commands[nextCommand] = c;
  // ... adjust indices
  c.execute();
  }
```

```
  public void undo() {
  if (!canUndo()) return;
  // ... adjust indices
  commands[nextCommand].unexecute();
  }

  public void redo() {
  if (!canRedo()) return;
  commands[nextCommand].execute();
  // ... adjust indices
  }

  public boolean canUndo() {...}

  public boolean canRedo () {...}
}
```

In this method we use a circular array to improve efficiency and resource usage of our `CommandHistory`, however, we do this at a cost to the complexity of writing the implementation. Determining how to use three indices (in this case `undoLimit`, `nextCommand`, and `redoLimit`) can be tricky. Once we determine what the three indices are, we can use abstraction functions to help us implement the `CommandHistory`. Here, an ***Abstraction Function*** will map the circular array-based implementation to the two abstract sequential lists holding the set of `Commands`s we can undo and redo.

Defining Indices (First Implementation)

To begin, we must decide upon exactly what the three indices `undoLimit`, `nextCommand`, and `redoLimit` really mean.

```
nextCommand - the index of the next open space available to hold
issued Commands.

undoLimit - index to oldest undo-able Command if undo is possible
    How do we tell when there's nothing to undo? : undoLimit =
nextCommand

redoLimit - index to final redo-able Command if redo is possible
    How do we tell when there's nothing to redo? : decr(nextCommand)
= redoLimit
```

Note here, we already introduce `incr()` as a quick notation that will allow us to walk around a circular array more easily. This serves as a kind of utility function that makes it easier to think about the array as if it were circular, even though the underlying implementation is not (somewhat like an abstraction function). Along with `decr()` these can be defined as:

```
int incr(int val) {
  return (val == MAX_HISTORY) ? 0 : val + 1;
}
int decr(int val) {
  return (val == 0) ? MAX_HISTORY : val - 1;
}
```

Representation Invariants

Now, how would we like to define the `CommandHistory`'s representation invariant. We should think about "what assumptions do our underlying methods rely upon." This will translate into checks that we will make on fields in the `CommandHistory` class.
For Example:

- We should always check if any of these array indices are between 0 and the size of the array.
- The index `nextCommand` should always be "in between" (in a circular sense) of `undoLimit` and `redoLimit`; **EXCEPT** when we cannot `redo()`. In that case `nextCommand` is one index greater than `redoLimit`.
- Every element in the array, between the `undoLimit` and the `redoLimit` (in a circular sense) should not be **null**.
- ...

So lets look at some example scenarios when we set **MAX_HISTORY=3**.

- What are possible assignments of the three indices when both the abstract undo-sequence is empty and the abstract redo-sequence is empty?
  One possible setting for these indices is:

```
nextCommand = 0;
undoLimit = 0;
redoLimit = 2;
```

  So the decision that we "**canRedo**" when **decr(nextCommand) = redoLimit** requires us to initially set `redoLimit` to be **MAX_HISTORY-1**. Note, the decision of what to set `nextCommand` to does not matter, however, only its relation with `undoLimit` and `redoLimit`.

- What are the possible underlying index assignments for the case when the abstract undo-sequence has two elements [1,2] and the redo-sequence is empty? For example, when two `Command`s have been `issue()`'d.
  The `nextCommand` index could have been initially set to **0**, **1**, or **2**. Therefore the underlying array could look like: **[1,2,-]**, **[-,1,2]**, or **[2,-,1]**.
- Now consider the case of when the undo-sequence has three elements [1,2,3] and the redo-sequence is empty. What would the assignments of the three indices be?

> Would any of these invalidate any of our representation invariants?
> In fact, we cannot hold three elements in a circular array of size **MAX_HISTORY=3** without breaking our representation invariants.

So, the case where three elements cannot be held within our array implementation leads us to believe this is a **poorly designed** representation.

Optional: First (Poorly Designed) Implementation's Representation Invariant

We **could** write up a representation invariant if we'd like, even though this is a poor representation.

```
if (nextCommand >= commands.length || nextCommand < 0) { return
false; }
if (undoLimit >= commands.length || undoLimit < 0) { return false; }
if (redoLimit >= commands.length || redoLimit < 0) { return false; }
if (canRedo || canUndo) {
  bool foundN = false
  int i = undoLimit;
  while (i != redoLimit) {
    if (command[i] == null) { return false; }
    if (i == nextCommand) { foundN = true; }
    incr(i);
  }
  if ((!foundN) && (nextCommand != incr(redoLimit))) {
    return false;
  }
}
```

As we can see, this is a complicated method to check for an ill-formed object. This may stem from our initial choice to use the three indices in the way we did. There are certainly simpler ways to use an underlying circular array to implement a `CommandHistory`.

Second Implementation

Lets try to come up with a **different** implementation of `CommandHistory` that still uses three array management variables.

One such choice of fields could be:

```
next - the index of the next space where an issued command is placed

undoLen - the number of undo-able Commands

redoLen - the number of redo-able Commands
```

Again, looking at an array of size **MAX_HISTORY=3**, lets answer some of the questions we raised before about the management of these variables.

- What are possible assignments of the three indices when both the abstract undo-sequence is empty and the abstract redo-sequence is empty?
  One possible assignment is:

```
next = 0;
undoLen = 0;
redoLen = 0;
```

  Again, `next` can actually point to any index (0, 1, or 2), however `undoLen` and `redoLen` are always **0** to begin with.

- What are the possible underlying index assignments for the case when the abstract undo-sequence has two elements [1,2] and the redo-sequence is empty?
  Similar to before, the `next` index can be initially set to **0**, **1**, or **2**. Therefore the underlying array could look like: **[1,2,-]**, **[-,1,2]**, or **[2,-,1]**. However, for all of these cases `undoLen`=**2** and `redoLen`=**0**.
- Now consider the case of when the undo-sequence has three elements [1,2,3] and the redo-sequence is empty. What would the assignments of the three indices be? Would any of these invalidate any of our representation invariants?
  In this second implementation we **can** hold three undo `Command`s, even with our array sized **MAX_HISTORY=3**.

Now that we have a feel for how this implementation works, lets determine its representation invariant. Again, we should think about what assumptions do our underlying methods rely upon.

For example:

- The index `next` must be between 0 and the size of the array.
- The sum of the `undoLen` and `redoLen` must be less than or equal to the size of the array.
- None of the objects between the start of the undo sequence and the end of the redo sequence can be `null`
-
- ...

Writing up pseudocode to perform these checks, we get:

```
public boolean checkRep() {
  if ( next < 0 || next >= MAX_HISTORY) { return false; }
  if ( undoLen + redoLen > MAX_HISTORY) { return false; }
  int i = next;
```

```
  for (int c=0; c < undoLen; c++) {
    i = decr(i);
    if (commands[i] == null) { return false; }
  }
  for (int c=0; c < redoLen; c++) {
    if (commands[i] == null) { return false; }
    i = incr(i);
  }
}
```

Given the representation invariant, we can determine an ***Abstraction Function*** that shows us how to implement the abstract representation of a `CommandHistory`. Recall, this abstract representation is a combination of two sequential lists of `Command`s that tell what `Command`s you can undo and redo. Lets now write a `.toString()` abstraction function that will print out all of the elements in both of these abstract sequential lists.

```
public String toString() {
  String outString = "";
  int i = (next-undoLen+MAX_HISTORY) % MAX_HISTORY;
  for (int c=0; c < undoLen; c++) {
    outString += commands[i];
    i = incr(i);
  }
  outString += " : ";
  i = next;
  for (int c=0; c < redoLen; c++) {
    outString += commands[i];
    i = incr(i);
  }
  return outString;
}
```

Now that we have figured out an abstraction function, we can use what we learned to implement the rest of the `CommandHistory` class. The methods `issue()`, `undo()`, and `redo()` can be implemented as follows.

```
public void issue (Command c) {
  commands [next] = c;
  next = incr(next);
  if (undoLen < MAX_HISTORY)
    undoLen = incr(undoLen);
  redoLen = 0;
  c.execute ();
}
public void undo () {
  if (undoLen > 0) {
    next = decr(next);
    undoLen--;
    redoLen++;
    commands[next].unexecute();
```

```
  }
}
public void redo () {
  if (redoLen > 0) {
    commands[next].execute();
    next = incr(next);
    redoLen--;
    redoLen++;
  }
}
```

Java Puzzler : Null and Void

What does this program do?

```
public class Null {
  public static void greet() {
    System.out.println("Hello, world!");
  }

  public static void main(String[] args) {
    ((Null) null).greet();
  }
}
```

This program looks as though it ought to throw a NullPointerException. The main method invokes the greet method on null, and you can't invoke a method on null, can you? But if you ran the program, you found that it prints "Hello, world!"

The key is that Null.greet is a static method. It is a bad idea to use an expression as the qualifier in a static method invocation. Not only does the run-time type of the object referenced by the expression's value play no role in determining which method gets invoked, but also the identity of the object, if any, plays no role. In this case, there is no object, but that makes no difference. A qualifying expression for a static method invocation is evaluated, but its value is ignored. here is no requirement that the value be non-null.

To fix the problem, use the class as the qualifier: Null.greet(); Or better yet, eliminate the qualifier entirely: greet();

Lesson: Qualify static method invocations with a type, or don't qualify them at all.