

CS231 Fall 2013 Lecture 12

Stephanie R Taylor

September 30, 2013

1 Administrative Topics

- Return the quiz?
- I will be away this week. No HW. No quiz. Bruce will lecture on Wed and Fri.
- Go over course timing for the spring.
- Be sure to read the notes on exceptions and generics.

2 Review Iterators

For our linked list classes (both the in-class and project classes), there are iterators. with this relationship to the class:

- The `LinkedList` implements `Iterable`, which means it has a method that creates an `Iterator` object.
- The `LinkedList` has an inner class that implements `Iterator`. This inner class has `hasNext`, `next`, and `remove` methods.

And because the linked lists have iterators, it is possible to use the `foreach` loop structure. Nice!

3 Abstract classes

- Suppose you have a linked list, like the Landscape in your project. In that project, you are storing Cell objects.
- But suppose you want to put several kinds of objects in each grid cell instead of just a Cell object. For example, you might want to include food in some cells, weapons in others, and deer, humans, or wolves on others and leave some empty.
- We could just use Strings in each cell to represent the kind of object, in which case we could declare the linked list to be `LinkedList<String>`.
- But we might want to store information about each object, such as what weapons and food the human is carrying, the type and strength of the weapon, the hunger level of the wolves.
- So a better solution is to use different classes of objects Weapon, Food, Wolf, Human, etc.
- But how can we store all those different kinds of objects in the list? When we declare the linked list, what type should it be? String won't work
- We could use `LinkedList<Object>`, which allows any kind of object to be passed in. However, there is a problem.
- Suppose we want to repeatedly update the grid each generation and suppose each object knows how to update itself. Then, whenever we wanted to the objects in the grid, we would like to do something like the following:

```
for (Object obj: cellList) {  
    obj.update();  
}
```
- The problem is that `obj` is of type `Object` and so the compiler won't let you call `update()` on it. Suggestions?
- Here's one solution:

```

for (Object obj: cellList) {
    if (obj instanceof Human) {
        Human h = (Human) obj;
        h.update();
    } else if (obj instanceof Wolf) {
        Wolf w = (Wolf) obj;
        w.update();
    } else if (...)
}

```

- This is really ugly. There is another related problem. We want objects of all these classes to know where they are (their row and column) and so all objects will have methods `getRow()`, `getCol()`, `setPosition(row, col)`. That's a lot of duplicated code and so an ugly design.
- Solution? Use superclasses! This is exactly what they are for.

```

public class SimObject {
    public int getRow() {...}
    public int getCol() {...}
    public void setPosition(int row, int col) {...}
    public void update() {...}
}
public class Wolf extends SimObject { ... }
public class Human extends SimObject { ... }
<SimObject> obj_list;

```

- Now the original for loops will work:

```

for (SimObject obj: cellList) {
    obj.update();
}

```
- Also, we don't have duplicated code. Instead, the code that would have been duplicated is now in the superclass instead.
- One last problem: the update method is not the same for each subclass. Each subclass has its own version of update. Solution? [have each subclass create an update method that overrides the inherited method.]
- Good, but now what should the superclass's update method do? Error message?

- More elegant solution: Declare `SimObject` and `update()` as abstract. This tells the compiler that the superclass is not implementing `update()` and instead the subclasses must implement `update()` or they are not legal subclasses.

4 Improving linked lists

4.1 Time complexities

- What is the time complexity of adding to the beginning of a linked list? It takes constant time, regardless of how many items are in the list. We can state this formally using *Big O* notation. If something takes constant time, then it is considered $O(1)$ (pronounced "order 1" or "O 1").
- What is the time complexity of adding to the end of a linked list? Since you have to traverse every node in the list, the amount of time is a function of the size N of the list. It is $O(N)$.
- How does that compare to adding something to the end of an `ArrayList`? When there is enough memory allocated, it is constant time. But if there isn't enough memory allocated, then all N elements must be copied, which is $O(N)$. Since Big O notation describes the worst-case scenario, we say this operation is $O(N)$.
- What about accessing a particular element in the list? (e.g. `list.get(i)`).
 - For arrays, it is $O(1)$ (we can just "index into it")
 - For `ArrayLists`, it is $O(1)$ (we can just "index into it")
 - For linked lists, it is $O(N)$.
- In general, if we can make an operation $O(1)$, then that is a very good thing.

[Show memory and how nodes are actually scattered all over the place this is why `insertAtBeginning` is $O(1)$.]

4.2 Adding a tail pointer

Consider the method `addLast()` (which adds an element to the end of the list). What might have made it easier? Is there a way to cache some value to make it faster? i.e. do you think we could add a field to our list that would allow us to jump to the end of the list? [tail pointer] We can easily add another instance variable `tail` that always points to the last node of the list.

```
public class HTLinkedList
{
    private ListNode head, tail;

    ...
}
```

Lets rewrite some of the methods in the `StringLinkedList` class for this class.