



# **Advanced Object-Oriented Programming**

CPT204 – Lecture 8  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大學

# CPT204 Advanced Object-Oriented Programming

## Lecture 8

### **Specification 2, Linked List 4**

# Welcome !

---

- Welcome to Lecture 8 !
- In this lecture we are going to
  - continue to learn about specification
    - discuss the properties of specs
    - learn what makes some specs better than others
  - construct a homemade list, this time using an array as the underlying data structure, while learning about
    - array copy
    - array resizing
    - generic arrays
    - object deletion

## Part 1: Three Dimensions for Comparing Specs

---

- In the first part of the lecture, we'll look at different specs for similar behaviors (e.g. finding an element in an array), and discuss the tradeoffs between them
- We'll look at three dimensions for comparing specs:
  - How **deterministic** it is: does the spec define only a single possible output for a given input, or does it allow the implementor to choose from a set of legal outputs?
  - How **declarative** it is: does the spec just characterize what the output should be, or does it explicitly say how to compute the output?
  - How **strong** it is: does the spec have a small set of legal implementations, or a large set?
- Not all specifications we might choose for a module are equally useful, and we'll explore what makes some specifications *better* than others

# Deterministic vs Underdetermined Specs (1)

---

- Recall the two example implementations of find we began with in previous lecture:

```
static int findFirst(int[] arr, int val) {  
    for (int i = 0; i < arr.length; i++) {  
        if (arr[i] == val) return i;  
    }  
    return arr.length;  
}  
  
static int findLast(int[] arr, int val) {  
    for (int i = arr.length - 1 ; i >= 0; i--) {  
        if (arr[i] == val) return i;  
    }  
    return -1;  
}
```

- The subscripts <sub>First</sub> and <sub>Last</sub> are not actual Java syntax: we're using them here to distinguish the two implementations for the sake of discussion  
In the actual code, both implementations should be Java methods called find

## Deterministic vs Underdetermined Specs (2)

---

- Here is one possible specification of find:

```
static int findExactlyOne(int[] arr, int val)
```

**requires:** val occurs exactly once in arr

**effects:** returns index i such that arr[i] == val

- This specification is **deterministic**: when presented with a state satisfying the precondition, the outcome is completely determined
  - Only one return value and one final state are possible
  - There are no valid inputs for which there is more than one valid output
- Both  $\text{find}_{\text{First}}$  and  $\text{find}_{\text{Last}}$  **satisfy** the specification, so if this is the specification on which the clients relied, the two implementations are equivalent and substitutable for one another

## Deterministic vs Underdetermined Specs (3)

---

- Here is a slightly different specification:

```
static int findOneOrMore,AnyIndex (int[] arr, int val)  
  requires: val occurs in arr  
  effects: returns index i such that arr[i] == val
```

- This specification is **not deterministic**: it *doesn't* say which index is returned if val occurs more than once
  - It simply says that if you look up the entry at the index given by the returned value, you'll find val
  - This specification allows *multiple valid outputs* for the *same input*
- Note that this is different from nondeterministic, for example, when the code's behavior depends on a random number
  - To avoid the confusion, we'll refer to specifications that are **not deterministic** as **underdetermined**

## Deterministic vs Underdetermined Specs (4)

---


- This underdetermined find spec is again **satisfied** by *both*  $\text{find}_{\text{First}}$  and  $\text{find}_{\text{Last}}$ , each resolving the underdeterminedness in its own way !
- A client of  $\text{find}^{\text{OneOrMore, AnyIndex}}$  spec **cannot** rely on which index will be returned if `val` appears more than once
- The spec would be satisfied by a nondeterministic/random implementation, too — for example, one that tosses a coin to decide whether to start searching from the beginning or the end of the array
- But in almost all cases we'll encounter, underdeterminism in specifications offers *a choice* that is made by the implementer at implementation time



# Declarative vs Operational Specs (1)

---

- **Operational** specifications give a series of steps that the method performs
  - pseudocode descriptions are operational
- **Declarative** specifications don't give details of intermediate steps
  - instead, they just give properties of the final outcome and how it's related to the initial state
- Almost always, *declarative* specifications are ***preferable***
  - they're usually shorter, easier to understand, and
  - most importantly, they don't inadvertently *expose implementation details* that a client may rely on (and then find no longer hold when the implementation is changed)



some of our specs do this,  
for example when I ask for recursive implementation.  
it's for educative purpose; generally, don't do that!

## Declarative vs Operational Specs (2)

---

- For example, if we want to allow either implementation of find, we would not want to say in the spec that the method “*goes down the array until it finds val*”
  - since aside from being rather vague, this spec suggests that the search proceeds from lower to higher indices and that the lowest will be returned, which perhaps the specifier did **not** intend
- One reason programmers sometimes lapse into operational specifications is because they're using the spec comment to explain the implementation for a maintainer
  - don't do that!
  - explain to the user first, not just to the maintainer
  - when it's necessary, use comments within the body of the method, **not** in the spec comment

## Declarative vs Operational Specs (3)

---

- For a given specification, there may be many ways to express it declaratively:
  - a. `static boolean startsWith(String str, String prefix)`  
**effects:** returns true if and only if there exists String suffix such that `prefix + suffix == str`
  - b. `static boolean startsWith(String str, String prefix)`  
**effects:** returns true if and only if there exists integer `i` such that `str.substring(0, i) == prefix`
  - c. `static boolean startsWith(String str, String prefix)`  
**effects:** returns true if the first `prefix.length()` characters of `str` are the characters of `prefix`, false otherwise
- It's up to us to choose the clearest specification for clients and maintainers of the code

# Stronger vs Weaker Specs (1)

---

- Suppose you want to change a method — either how its implementation behaves, or the specification itself
  - There are already clients that depend on the method's current specification
  - How do you compare the behaviors of two specifications to decide whether it's safe to replace the old spec with the new spec?
- A specification S2 is ***stronger than or equal*** to a specification S1 if
  - S2's precondition is *weaker than or equal* to S1's, and
  - S2's postcondition is *stronger than or equal* to S1's for the states that satisfy S1's precondition
- If this is the case, then an implementation that satisfies S2 can be used to satisfy S1 as well, and it's *safe* to *replace* S1 with S2 in your program

## Stronger vs Weaker Specs (2)

---

- These two rules embody several ideas: they tell you that you can *always weaken the precondition*; placing fewer demands on a client will never upset them; and you can always *strengthen the postcondition*, which means making more promises
- For example, this spec for find:

○ `static int findExactlyOne(int[] a, int val)`  
**requires:** val occurs *exactly once* in a  
**effects:** returns index i such that a[i] == val

the two rules can be hard to understand, but memorize these parts, these actually make sense !

can be replaced with:

○ `static int findOneOrMore, AnyIndex(int[] a, int val)`  
**requires:** val occurs *at least once* in a  
**effects:** returns index i such that a[i] == val

which has *a weaker precondition*

user already gave exactly once, automatically fulfil at least once

## Stronger vs Weaker Specs (3)

---

- In turn, that last spec:

- `static int findOneOrMore, AnyIndex(int[] a, int val)`  
**requires:** val occurs at least once in a  
**effects:** returns index i such that `a[i] == val`

can be replaced with:

- `static int findOneOrMore, FirstIndex(int[] a, int val)`  
**requires:** val occurs at least once in a  
**effects:** returns *lowest* index i such that `a[i] == val`

which has *a stronger postcondition*

user promised an index before,  
now also promised an index,  
in particular the lowest

# In-Class Quiz 1

---

- Consider the following specification

```
static int findCanBeMissing(int[] a, int val)  
  requires: nothing  
  effects: returns index i such that a[i] == val, or -1 if no such i
```

and compare with:

```
static int findOneOrMore,FirstIndex(int[] a, int val)  
  requires: val occurs at least once in a  
  effects: returns lowest index i such that a[i] == val
```

- which is true about their *precondition* and *postcondition* ?

# Incomparable

---

- The specification:

```
static int findCanBeMissing(int[] a, int val)  
  requires: nothing  
  effects: returns index i such that a[i] == val, or -1 if no such i
```

compared to **find**<sup>OneOrMore,FirstIndex</sup>

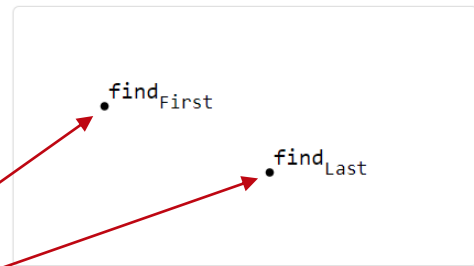
- again the precondition is weaker
- but for inputs that satisfy **find**<sup>OneOrMore,FirstIndex</sup>'s precondition, the postcondition is also weaker: the requirement for lowest index has been removed
- Neither of these two specifications is stronger than the other: they are **incomparable**



# Diagramming Specification (1)

---

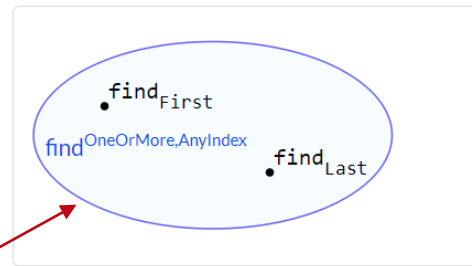
- Imagine (very abstractly) the space of all possible Java methods
- Each point in this space represents a method implementation
- First we'll diagram  $\text{find}_{\text{First}}$  and  $\text{find}_{\text{Last}}$  defined previously
- Look back at the code and see that  $\text{find}_{\text{First}}$  and  $\text{find}_{\text{Last}}$  are **not** specs
  - they are *implementations*, with method bodies that implement their actual behavior
  - so we denote them as points in the space



## Diagramming Specification (2)

---

- A specification defines a region in the space of all possible implementations
- A given implementation either behaves according to the spec, satisfying the precondition-implies-postcondition contract (it is inside the region), *or* it does not (outside the region)
- Both  $\text{find}_{\text{First}}$  and  $\text{find}_{\text{Last}}$  satisfy  **$\text{find}^{\text{OneOrMore, AnyIndex}}$** , so they are *inside* the region defined by that spec



## Diagramming Specification (3)

---

- We can imagine clients looking in on this space: the specification acts as a *firewall*
  - Implementors have the freedom to move around inside the spec, changing their code without fear of upsetting a client
    - This is crucial in order for the implementer to be able to improve the performance of their algorithm, the clarity of their code, or to change their approach when they discover a bug, etc
  - Clients don't know which implementation they will get
    - They must respect the spec, but also have the freedom to change how they're using the implementation without fear that it will suddenly break

## Diagramming Specification (4)

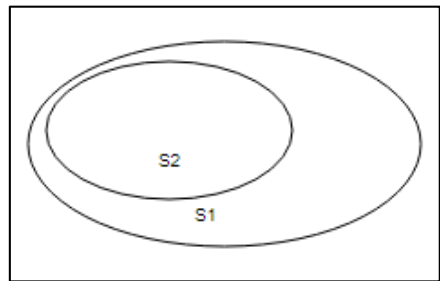
---

- How will similar specifications relate to one another?
- Suppose we start with specification S1 and adapt it to create a new specification S2
  - If S2 is stronger than S1, how will these specs appear in our diagram?
- Let's start by **strengthening the postcondition**
  - If S2's postcondition is now stronger than S1's postcondition, then S2 is the stronger specification
  - Think about what strengthening the postcondition means for implementers
  - It means they have less freedom, because the requirements on their output are stronger

## Diagramming Specification (5)

---

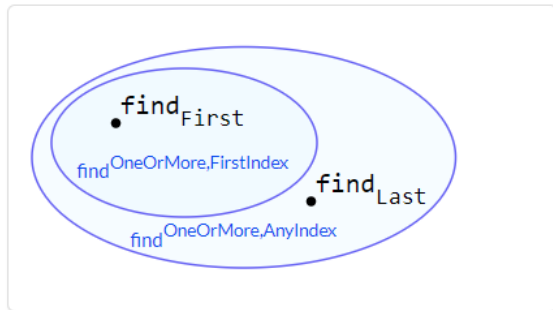
- If we **weaken the precondition**, which again makes S2 a stronger specification, then implementations will have to handle new inputs that were previously excluded by the spec
  - If they behaved badly on those inputs before, we wouldn't have noticed, but now their bad behavior is exposed
- So when S2 is stronger than S1, it defines a smaller region in this diagram
  - Fewer implementations satisfy S2 than S1
  - Every implementation that satisfies S2 also satisfies S1, though, so the smaller region S2 is nested inside S1



## Diagramming Specification (6)

---

- Now let's see how this works with our find specifications
- Where does **find**<sup>OneOrMore,FirstIndex</sup> fit on the diagram?
  - Is it stronger or weaker?
- Implementations can satisfy **find**<sup>OneOrMore,AnyIndex</sup> by returning any index  $i$ , but now the spec demands the lowest index  $i$
- So there are now implementations *inside* **find**<sup>OneOrMore,AnyIndex</sup> but *outside* **find**<sup>OneOrMore,FirstIndex</sup>



## Diagramming Specification (7)

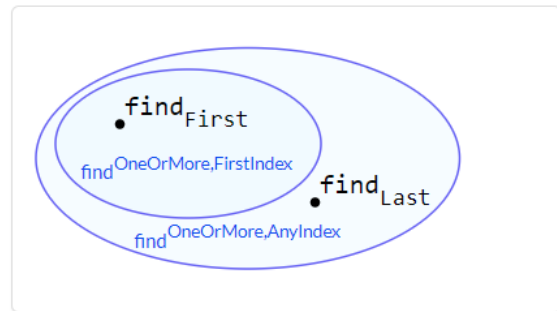
---

- Could there be implementations inside  $\text{find}^{\text{OneOrMore, FirstIndex}}$  but outside  $\text{find}^{\text{OneOrMore, AnyIndex}}$ ?

- No, all of those implementations satisfy a stronger postcondition than what  $\text{find}^{\text{OneOrMore, AnyIndex}}$  demands

- In our figure, since  $\text{find}_{\text{Last}}$  iterates from the end of the array `arr`, it does not satisfy  $\text{find}^{\text{OneOrMore, FirstIndex}}$  and is outside that region

- Another specification S3 that is neither stronger nor weaker than S1 ***might overlap*** (such that there exist implementations that satisfy only S1, only S3, and both S1 and S3) or ***might be disjoint***
  - In both cases, S1 and S3 are **incomparable**



# Designing Good Specifications

---

- What makes a good method?
  - Designing a method means primarily writing a specification
- About the form of the specification: it should obviously be succinct, clear, and well-structured, so that it's easy to read.
- The content of the specification, however, is harder to prescribe
  - There are no infallible rules, but there are some useful guidelines, as follows in the rest of this first part of the lecture



# The Specification should be Coherent (1)

---

- The spec shouldn't have lots of different cases
  - Long argument lists, deeply nested if-statements, and boolean flags are all signs of trouble
- Consider this specification:

```
static int sumFind(int[] a, int[] b, int val)
```

**effects:** returns the sum of all indices in arrays a and b at which val appears

- Is that a well-designed procedure?
  - Probably not: it's **incoherent**, since it does several things (finding in two arrays and summing the indexes) that are not really related
  - It would be better to use *two separate procedures*, one that finds the indexes, and the other that sums them

## The Specification should be Coherent (2)

---

- Here's another example:

```
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;

/**
 * Update longestWord to be the longest element of words, and print
 * the number of elements with length > LONG_WORD_LENGTH to the console.
 * @param words list to search for long words
 */
public static void countLongWords(List<String> words)
```

- In addition to *terrible use of global variables* and *printing instead of returning*, the specification is **not coherent** — it does two different things, counting words and finding the longest word
- Separating those two responsibilities into two different methods will make them simpler (easy to understand) and more useful in other contexts (ready for change)

# The Results of a Call should be Informative

---

- Consider the specification of a method that puts a value in a map, where keys are of some type K and values are of some type V:

```
static V put(Map<K,V> map, K key, V val)
```

**requires:** val may be null, and map may contain null values

**effects:** inserts (key, val) into the mapping, overriding any existing mapping for key, and returns old value for key, unless none, in which case it returns null

- Note that the precondition does **not** rule out null values so the map can store nulls
- But the postcondition **uses** null as a special return value for a missing key
  - This means that if null is returned, you *can't tell* whether the key was not bound previously, *or* whether it was in fact bound to null
  - This is not a very good design, because the return value is *useless* unless you know for sure that you didn't insert nulls

# The Specification should be Strong Enough

---

- Spec should give clients a strong enough guarantee in the general case — it needs to satisfy their basic requirements
  - we must use extra care when specifying *the special cases*, to make sure they don't undermine what would otherwise be a useful method
- For example, there's no point throwing an exception for a bad argument but allowing arbitrary mutations, because a client won't be able to determine what mutations have actually been made :

```
static void addAll(List<T> list1, List<T> list2)
```

**effects:** adds the elements of list2 to list1, unless it encounters a null element, at which point it throws a NullPointerException

- If a NullPointerException is thrown, the client is left to figure out on their own *which elements of list2 **actually made it** to list1*

# The Specification should be Weak Enough

---

- Consider this specification for a method that opens a file:

```
static File open(String filename)  
  effects: opens a file named filename
```

- This is a bad specification
- It lacks important details: is the file opened for reading or writing?
  - Does it already exist or is it created?
  - And it's ***too strong***, since there's no way it can guarantee to open a file
  - The process in which it runs may lack permission to open a file, or there might be some problem with the file system beyond the control of the program
  - Instead, the specification should say something much weaker: that it attempts to open a file, and if it succeeds, the file has certain properties

# The Specification should use Abstract Types where possible

---

- In Java, we can distinguish between more abstract notions like a `List` or `Set` and particular implementations like `ArrayList` or `HashSet` (will talk about this in future lecture in detail)
- Writing our specification with abstract types gives more freedom to both the client and the implementer
- In Java, this often means using an *interface* type, like `Map` or `Reader`, instead of specific implementation types like `HashMap` or `FileReader`, for example this specification:

```
static ArrayList<T> reverse(ArrayList<T> list)
```

**effects:** returns a new list which is the reversal of `list`, i.e.

`newList[i] == list[n-i-1]` for all  $0 \leq i < n$ , where `n = list.size()`

- This forces the client to pass in an `ArrayList`, and forces the implementer to return an `ArrayList`, even if there might be alternative `List` implementations that they would rather use
- Since the behavior of the specification doesn't depend on anything specific about `ArrayList`, it would be **better** to write this spec in terms of the more abstract `List`

## Precondition or Postcondition? (1)

---

- Another design issue is whether to use a precondition, and if so, whether the method code should attempt to make sure the precondition has been met before proceeding
- In fact, the most common use of preconditions is to demand a property precisely because it would be hard or expensive for the method to check it
- As mentioned above, a non-trivial precondition inconveniences clients, because they have to ensure that they don't call the method in a bad state (that violates the precondition); if they do, there is no predictable way to recover from the error
- That's why the Java API classes, for example, tend to specify (as a postcondition) that they throw unchecked exceptions when arguments are inappropriate
  - This approach makes it easier to find the bug or incorrect assumption in the caller code that led to passing bad arguments
  - In general, it's better to **fail fast**, as close as possible to the site of the bug, rather than let bad values propagate through a program far from their original cause

## Precondition or Postcondition? (2)

---

- Sometimes, it's not feasible to check a condition without making a method unacceptably slow, and a precondition is often necessary in this case
- If we wanted to implement the find method using binary search, we would have to require that the array be sorted
  - Forcing the method to actually check that the array is sorted would defeat the entire purpose of the binary search: to obtain a result in logarithmic and not linear time
- The decision of whether to use a precondition is an engineering judgment
- The key factors are the cost of the check (in writing and executing code), and the scope of the method
  - If it's only called locally in a class, the precondition can be discharged by carefully checking all the sites that call the method
  - But if the method is public, and used by other developers, it would be less wise to use a precondition — instead, like the Java API classes, you should throw an exception



# About Access Control (1)

---

- The decision to make a method **public** or **private** is actually a decision about the ***contract*** of the class
- Public methods are freely accessible to other parts of the program
  - Making a method public advertises it as a service that your class is willing to provide
- If you make *all* your methods public — including helper methods that are really meant only for local use within the class — then other parts of the program may come to depend on them, which will make it harder for you to change the internal implementation of the class in the future
  - Your code won't be as ready for change

## About Access Control (2)

---

- Making *internal helper methods* public will also add clutter to the visible interface your class offers
  - Keeping internal things private makes your class's public interface smaller and more coherent (meaning that it does one thing and does it well)
  - Your code will be easier to understand
- We will see even stronger reasons to use private in the next lectures, when we start to write classes with persistent internal state
  - Protecting this state will help keep the program safe from bugs

# About Static vs Instance Methods (1)

---

- Static methods are **not** associated with any particular instance of a class, while instance methods (declared without the static keyword) must be called on a particular object
- Specifications for instance methods are written just the same way as specifications for static methods, but they will often refer to properties of the instance on which they were called
- For example, by now we're very familiar with this specification:

```
static int find(int[] arr, int val)
```

```
  requires: val occurs in arr
```

```
  effects: returns index i such that arr[i] = val
```

## About Static vs Instance Methods (2)

---

- Instead of using an `int[]`, what if we had a class `IntArray` designed for storing arrays of integers?
- The `IntArray` class might provide an instance method with the specification:

```
int find(int val)
```

**requires:** `val` occurs in this array

**effects:** returns index `i` such that the value at index `i` in this array is `val`

## In-Class Quiz 2

---

- Which of the following is a sign of a good specification?
  - ☐ the specification is declarative
  - ☐ the specification is operational
  - ☐ the specification is as strong as possible
  - ☐ the specification is as weak as possible
  - ☐ the implementation is allowed to ignore invalid arguments in the specification
  - ☐ the implementation is allowed to use different algorithms depending on the arguments
  - ☐ the specification is making use of the reader's knowledge of the implementation

## Part 2: Array-based List

---

- For the last few weeks, we have built our homemade list using node and pointer to another node (link-based) as our underlying data structure
  - another popular underlying data structure is *array*
- In this second part of the lecture, as a running example, before we go into more sophisticated data structure, let us once again build a homemade list, but now *using array*
  - for completeness, let us briefly review array

# Review: Arrays (1)

---



- Arrays are fixed-length sequences of another type
- To declare an array variable and construct an array value to assign to it:
  - `int[] a = new int[100];`
  - it includes all possible int array values, but once an array is created, we **cannot** change its length
- Operations on array types include:
  - indexing      `a[2]` ← index starts from 0
  - assignment      `a[2] = 0`
  - length      `a.length` ← this is not a method call  
array does **not** have methods

## Review: Arrays (2)

- Initialize an array

- `arr1 = new int[5];`
- `arr2 = new int[]{1, 2, 3, 4, 5};`
- `int[] arr3 = {9, 10, 11, 12, 13};`

each entry will get a default value for int, the default value is 0

no need to specify the length

no need to use new if you also declare the variable

- Example:

```
int[] z = null;  
int[] x, y;
```

```
x = new int[]{1, 2, 3, 4, 5};  
y = x;  
x = new int[]{6, 7, 8, 9};  
y = new int[3];  
z = new int[0];
```

try running in Java Visualizer

aliasing, y points to x's array as well

at this point, no one points to {1, 2, 3, 4, 5} and it's deleted from memory by garbage collector

an empty array



# Arraycopy

---

- Two ways to copy the elements of an array:
  - Element by element using a loop
  - Using arraycopy
- Arraycopy takes 5 parameters, in order:
  - Source array
  - Start position in source
  - Target array
  - Start position in target
  - Number to copy
- Example: `System.arraycopy(a, 0, b, 3, 2);`
  - it will copy 2 elements from a (starting from 0) to b (starting from 3)

# Why Array-Based List?

---

- We have build a link-based list, why bother to build another one with array?
  - Well, we want to practice using it, for later when we build more complicated data structures
  - But there is also another reason specific to list: retrieval with **get**(int i) is slow!
    - with linked-list, you have to scan from beginning/last to get to the desired position i
    - one way to make this fast, is to change from link-based to array-based
- Retrieval from any position of an array is *very fast*
  - Independent of the array size

# ARList1

---

- Let us now start to build our homemade list using an array
  - start simple again, a list of *integers*
  - functionality includes:
    - empty constructor, size, get i-th item
    - addLast, getLast, delLast
- We will aim to give an alternate implementation of list using array
  - for the lab later, you will also give an alternate implementation of deque

we'll make it generic later

only add/delete from the back

# ARList1: Instance Variables, Empty Constructor, size

---

- As the underlying data structure of our list, we store the integers inside an int array

```
public class ARList1 {  
  
    private int[] items;  
    private int size;  
  
    /** Creates an empty list. */  
    public ARList1() {  
        items = new int[100];  
        size = 0;  
    }  
  
    /**  
     * Returns the number of items in the list.  
     * @return the size of the list.  
     */  
    public int size() {  
        return size;  
    }  
}
```

start simple with integer-only array

using an array, we have to pick a fixed size, let's pick an arbitrary size, say 100

as before, we cache the size

next, we are going to **addLast** an integer  
any idea how we implement that using an array?

## ARList1: addLast

---

- One idea is to put the integer starting in index 0, and then index 1, 2, and so on, while incrementing the size

```
/**
 * Adds item to the end of the list.
 * @param item is an integer.
 */
public void addLast(int item) {
    items[size] = item;
    size++;
}
```

notice that the next int goes into index size

next, we want to write **getLast** and **get(i)**  
how do we implement those using similar idea?

# ARList1: getLast, get(i)

---

- Remember that this is why we use array, we want get to be fast

```
/**
 * Returns the last item at the back of the list.
 * @return the last integer in the list.
 */
public int getLast() { return items[size - 1]; }

/**
 * Gets the item at the given index in the list.
 * @param i is an index where 0 is the front.
 * @return the ith item in the list.
 */
public int get(int i) { return items[i]; }
```

use the invariant that the last item is always in index size - 1

just return item in index i, in constant-time

next, we want to implement **delLast**  
how do you delete the item from the array?

# ARList1: delLast

---

- We don't have to actually delete the item from the array!

```
/**
 * Deletes and returns the item at the back of the list.
 * @return the last integer of the list to be deleted.
 */
public int delLast() {
    int x = getLast();
    size--;
    return x;
}
```

by decrementing the size and our invariant that the **last item is in size - 1**, we have effectively "delete" the item

the user of the list does **not** know that the item is actually still in the array, only we the implementer need to know this !

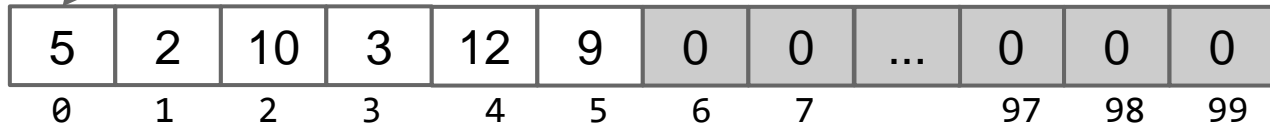
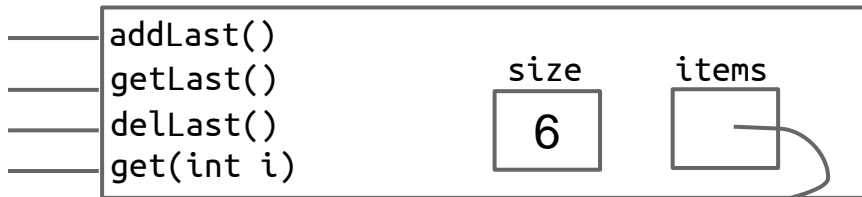
# ARList1: delLast

- We don't have to actually delete the item from the array!

```
/**  
 * Deletes and returns the item at the back of the List.  
 * @return the last integer of the list to be deleted.  
 */  
public int delLast() {  
    int x = getLast();  
    size--;  
    return x;  
}
```

by decrementing the size and our invariant that the **last item is in size - 1**, we have effectively "delete" the item

the user of the list does **not** know that the item is actually still in the array, only we the implementer need to know this !





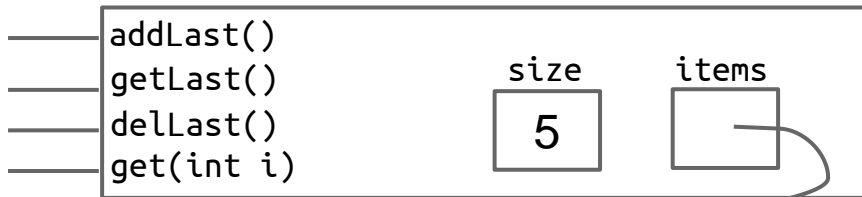
# ARList1: delLast

- We don't have to actually delete the item from the array!

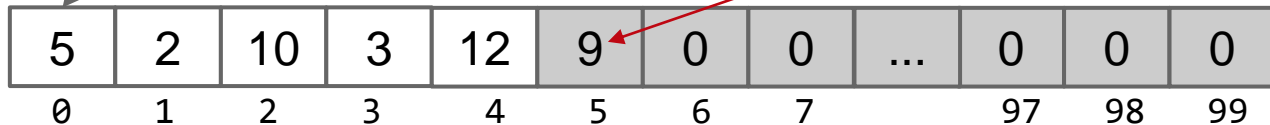
```
/**  
 * Deletes and returns the item at the back of the list.  
 * @return the last integer of the list to be deleted.  
 */  
public int delLast() {  
    int x = getLast();  
    size--;  
    return x;  
}
```

by decrementing the size and our invariant that the **last item is in size - 1**, we have effectively "delete" the item

the user of the list does **not** know that the item is actually still in the array, only we the implementer need to know this !



we don't really need to change this to, for example, zero, to preserve the invariant



## A Limitation

---

- There is an obvious limitation to our ARList implementation so far:
  - it cannot hold more than 100 items
  - and remember that the length of an array is fixed

what should we do if we want to store arbitrary number of items in our list ?

# Resizing Array

---

- One solution is called **resizing array**
  - we *don't* actually change the size of the array (because we cannot)
  - but we create a new array of bigger size, and copy the items

# Resizing Array

---

- One solution is called **resizing array**
  - we *don't* actually change the size of the array (because we cannot)
  - but we create a new array of bigger size, and copy the items

- So, when the array `items` gets too full


`size == items.length`

- we create a new array `a` with new **capacity**
- we copy using `arraycopy` to `a`
- we set `items` to be `a`

```
int[] a = new int[capacity];  
System.arraycopy(items, srcPos: 0, a, destPos: 0, size);  
items = a;
```

# Resizing Array

---

- One solution is called **resizing array**
  - we *don't* actually change the size of the array (because we cannot)
  - but we create a new array of bigger size, and copy the items
- So, when the array `items` gets too full 
  - we create a new array `a` with new **capacity**
  - we copy using `arraycopy` to `a`
  - we set `items` to be `a`
  - how large should the new capacity be?
    - `size + 1` ?

`size == items.length`

# Resizing Array

---

- One solution is called **resizing array**
  - we *don't* actually change the size of the array (because we cannot)
  - but we create a new array of bigger size, and copy the items

`size == items.length`

- So, when the array `items` gets too full
  - we create a new array `a` with new **capacity**
  - we copy using `arraycopy` to `a`
  - we set `items` to be `a`
  - how large should the new capacity be?
    - `size + 1` ?

even though there is now a place to store the new item in index `size`, `items` is full again and the next time another new item gets added we have to `arraycopy` (expensive)

# Resizing Array

---

- One solution is called **resizing array**
  - we *don't* actually change the size of the array (because we cannot)
  - but we create a new array of bigger size, and copy the items

`size == items.length`

- So, when the array `items` gets too full
  - we create a new array `a` with new **capacity**
  - we copy using `arraycopy` to `a`
  - we set `items` to be `a`
  - how large should the new capacity be?
    - `size + 1` ?
    - `size + 100` ?

additive factor is also turned out to be not good

# Resizing Array

---

- One solution is called **resizing array**
  - we *don't* actually change the size of the array (because we cannot)
  - but we create a new array of bigger size, and copy the items

`size == items.length`

- So, when the array `items` gets too full
  - we create a new array `a` with new **capacity**
  - we copy using `arraycopy` to `a`
  - we set `items` to be `a`
  - how large should the new capacity be?
    - `size + 1` ?
    - `size + 100` ?
    - `2 * size`

it can be shown\* that the best way is with multiplicative factor, say 2 times

\* interested students can look up running time analysis of "amortized analysis of table doubling"



# Implementing Resizing Array: Array Doubling

---

- Instead of writing all the code in addLast, it is better to create a separate method

```
/** Resizes the underlying array to the target capacity. */
private void resize(int capacity) {
    int[] a = new int[capacity];
    System.arraycopy(items, srcPos: 0, a, destPos: 0, size);
    items = a;
}

/**
 * Adds item to the end of the list.
 * @param item is an integer.
 */
public void addLast(int item) {
    if (size == items.length) {
        resize( capacity: 2 * size);
    }
    items[size] = item;
    size++;
}
```

if the array is full,  
we double the array size

so now we can add item as many as we want,  
just like what a list is supposed to do !

# Array Halving

---

- Suppose you addLast and double the array until it of size 1,000,000,000
  - and then delLast and remove 990,000,000 items
  - if you are not doing anything, you waste memory,
  - and this code is not efficient in space !




any idea how to fix this ?

# Array Halving

---

- Suppose you addLast and double the array until it of size 1,000,000,000
  - and then delLast and remove 990,000,000 items
  - if you are not doing anything, you waste memory,
  - and this code is not efficient in space !
- Opposite yet similar: once it is less full, reduce the size of the array
  - create a new array with smaller capacity
  - copy the elements to the new array



how much is less full?  
what is the new capacity?

# Array Halving

---

- Suppose you addLast and double the array until it of size 1,000,000,000
  - and then delLast and remove 990,000,000 items
  - if you are not doing anything, you waste memory,
  - and this code is not efficient in space !
- Opposite yet similar: once it is less full, reduce the size of the array
  - create a new array with smaller capacity
  - copy the elements to the new array
  - how much is less full? what is the new capacity?
    - if it is half full, halve the capacity ?




is this a good idea?

# Array Halving

---

- Suppose you addLast and double the array until it of size 1,000,000,000
  - and then delLast and remove 990,000,000 items
  - if you are not doing anything, you waste memory,
  - and this code is not efficient in space !
- Opposite yet similar: once it is less full, reduce the size of the array
  - create a new array with smaller capacity
  - copy the elements to the new array
  - how much is less full? what is the new capacity?
    - if it is half full, halve the capacity ?



this is a bad idea !  
it is full after delete and resize  
a sequence of add-deletes  
will result in copy all the time

# Array Halving

---

- Suppose you addLast and double the array until it of size 1,000,000,000
  - and then delLast and remove 990,000,000 items
  - if you are not doing anything, you waste memory,
  - and this code is not efficient in space !
- Opposite yet similar: once it is less full, reduce the size of the array
  - create a new array with smaller capacity
  - copy the elements to the new array
  - how much is less full? what is the new capacity?
    - if it is half full, halve the capacity ?
    - if it is one-quarter full, halve the capacity

it is half-full after resize

You will implement this idea in Lab !

## ARList2: Generic ARList (1)

---

- Finally, let us make our ARList generic, accepting any type T of items
  - as we previously did, we add <T> and change the int item into T item
  - with the exception of:

```
/** Creates an empty list. */
```

```
public ARList2() {  
    items = new T[100];  
    size = 0;  
}
```

Type parameter 'T' cannot be instantiated directly

this is because in Java,  
generic arrays are not allowed,  
causing generic array creation error

## ARList2: Generic ARList (2)

---

- Finally, let us make our ARList generic, accepting any type T of items
  - as we previously did, we add <T> and change the int item into T item
  - with the exception of:

```
/** Creates an empty list. */  
public ARList2() {  
    items = (T[]) new Object[100];  
    size = 0;  
}
```

to do this, we simply create  
an array of objects, and then  
**cast** it into array of T objects

```
/** Resizes the underlying array to the target capacity. */  
private void resize(int capacity) {  
    T[] a = (T[]) new Object[capacity];  
    System.arraycopy(items, srcPos: 0, a, destPos: 0, size);  
    items = a;  
}
```

However, this will trigger  
a compiler warning !



## ARList2: Generic ARList (3)

---

- Finally, let us make our ARList generic, accepting any type T of items
  - as we previously did, we add <T> and change the int item into T item
  - with the exception of:

```
/** Creates an empty list. */  
@SuppressWarnings("unchecked")  
public ARList2() {  
    items = (T[]) new Object[100];  
    size = 0;  
}
```

one solution is to add this **annotation** before method that casts generic array

```
/** Resizes the underlying array to the target capacity. */  
@SuppressWarnings("unchecked")  
private void resize(int capacity) {  
    T[] a = (T[]) new Object[capacity];  
    System.arraycopy(items, 0, a, 0, size);  
    items = a;  
}
```

we will discuss more about this in future lecture

## ARList2: delLast

---

- And on delLast:
  - unlike our integer-based ARList1, we actually want to **null out** deleted items

```
/**
 * Deletes and returns the item at the back of the list.
 * @return the last item of the list to be deleted.
 */
public T delLast() {
    T x = getLast();
    items[size - 1] = null;
    size--;
    return x;
}
```

if we don't do this,  
there still is reference to the deleted object  
and Java garbage collector won't delete it

# Thank you for your attention !

---

- In this lecture, you have learned to:
  - write declarative specs and to compare spec strength
  - write coherent, useful specifications of appropriate strength
  - use an array as an underlying dynamic data structure
  - resize an array efficiently in time and space : expand and shrink
  - create generic arrays
  - delete an object from an array completely
- Please continue to Lecture Quiz 8 and Lab 8:
  - to do Lab Exercise 8.1 - 8.4, and
  - to do Exercise 8.1 - 8.4