Vectors

Limitations of arrays

Limitations of arrays

- Arrays can be annoying to work with!

Limitations of arrays

- Arrays can be annoying to work with!
- Fixed size---can’t even create an array unless we know what size it will be!

Limitations of arrays

- Arrays can be annoying to work with!
- Fixed size---can’t even create an array unless we know what size it will be!
- No useful methods
  - Can only use [] to access specific items
  - And get the size of the array with .length

Vectors

Vectors

- Like an OOP version of arrays

Vectors

- Like an OOP version of arrays
- Don’t need to know the size up front

Vectors

- Like an OOP version of arrays
- Don’t need to know the size up front
- Come with other useful methods:
    - Check if an item exists in the Vector
    - “Insert” an item in the middle of the Vector

Vectors

- Like an OOP version of arrays
- Don’t need to know the size up front
- Come with other useful methods:
  - Check if an item exists in the Vector
  - “Insert” an item in the middle of the Vector
- Implemented with a Java class that we can all read

Java and structure5

- We’ll be talking about the `structure5` version of Vectors in this class

- Java has its own version of vectors
    - `java.util.vector`
    - We won’t use in this class, but it works in a very very similar way

Vector<E> API (select methods overview)

Vector<E> API (select methods overview)

- `get(int), set(int, E)`

Vector<E> API (select methods overview)

- `get(int), set(int, E)`
- `firstElement(), lastElement()`

Vector<E> API (select methods overview)

- `get(int), set(int, E)`
- `firstElement(),`
  `lastElement()`
- `contains(E), indexOf(E)`

Vector<E> API (select methods overview)

- `get(int), set(int, E)`
- `firstElement(), lastElement()`
- `contains(E), indexOf(E)`
- `add(E), addElement(E), add(int,E)`

Vector<E> API (select methods overview)

- `get(int), set(int, E)`
- `firstElement(), lastElement()`
- `contains(E), indexOf(E)`
- `add(E), addElement(E), add(int,E)`
- `remove(E)`

Vector<E> API (select methods overview)

- `get(int), set(int, E)`
- `firstElement(), lastElement()`
- `contains(E), indexOf(E)`
- `add(E), addElement(E), add(int,E)`
- `remove(E)`
- `clear()`

Vector<E> API (select methods overview)

- `get(int), set(int, E)`
- `firstElement(), lastElement()`
- `contains(E), indexOf(E)`
- `add(E), addElement(E), add(int,E)`
- `remove(E)`
- `clear()`
- `capacity()`

Vector<E> API (select methods overview)

- `get(int), set(int, E)`
- `firstElement(), lastElement()`
- `contains(E), indexOf(E)`
- `add(E), addElement(E), add(int,E)`
- `remove(E)`
- `clear()`
- `capacity()`
- `ensureCapacity()`

Vector<E> API (select methods overview)

- `get(int), set(int, E)`
- `firstElement(), lastElement()`
- `contains(E), indexOf(E)`
- `add(E), addElement(E), add(int,E)`
- `remove(E)`
- `clear()`
- `capacity()`
- `ensureCapacity()`
- `toString()`

Vector Details: Storing Data

Vector Details: Storing Data

Internally, the `Vector` class stores an array:

Vector Details: Storing Data

Internally, the `Vector` class stores an array:

```
        Object[] elementData;
```

Vector Details: Storing Data

Internally, the `Vector` class stores an array:

```
        Object[] elementData;
```

- The array is not necessarily filled

Vector Details: Storing Data

Internally, the `Vector` class stores an array:

```
        Object[] elementData;
```

- The array is not necessarily filled
- We keep track of the number of current elements in the array using an explicit `elementCount` variable

Vector Details: Storing Data

Internally, the `Vector` class stores an array:

```
        Object[] elementData;
```

- The array is not necessarily filled
- We keep track of the number of current elements in the array using an explicit `elementCount` variable
    - How do we ensure that `elementCount` stays in sync with our actual count?

Vector Details: Storing Data

Internally, the `Vector` class stores an array:

```
        Object[] elementData;
```

- The array is not necessarily filled
- We keep track of the number of current elements in the array using an explicit `elementCount` variable
    - How do we ensure that `elementCount` stays in sync with our actual count?
    - What happens if we try to add an element but the array is full?

Vector Details: Storing Data

Internally, the `Vector` class stores an array:

```
        Object[] elementData;
```

- The array is not necessarily filled
- We keep track of the number of current elements in the array using an explicit `elementCount` variable
  - How do we ensure that `elementCount` stays in sync with our actual count?
  - What happens if we try to add an element but the array is full?
- Overloaded constructor(s) allow us to specify an initial array size (we'll call this the Vector's capacity)

Vector Details: Storing Data

Internally, the `Vector` class stores an array:

```
Object[] elementData;
```

- The array is not necessarily filled
- We keep track of the number of current elements in the array using an explicit `elementCount` variable
    - How do we ensure that `elementCount` stays in sync with our actual count?
    - What happens if we try to add an element but the array is full?
- Overloaded constructor(s) allow us to specify an initial array size (we'll call this the Vector's capacity)
    - Default capacity used if none is provided

Vector Details: get(int)/set(int, E)

Arrays use bracket notation to access and update elements at a given index. Vectors use methods.

Vector Details: get(int)/set(int, E)

Arrays use bracket notation to access and update elements at a given index. Vectors use methods.

- We can't use bracket notation for non-array objects. We must call methods. But *internally* :
  - `v.get(int)` uses bracket notation to access `elementData[i]`
  - `v.set(int, E)` uses bracket notation to update `elementData[i]`

Vector Details: get(int)/set(int, E)

Arrays use bracket notation to access and update elements at a given index. Vectors use methods.

- We can't use bracket notation for non-array objects. We must call methods. But *internally* :
    - `v.get(int)` uses bracket notation to access `elementData[i]`
    - `v.set(int, E)` uses bracket notation to update `elementData[i]`

Get/set cost is the same as the cost of accessing/updating an array.

Vector Details: add(E)

Arrays don't have any notion of "appending”. add(E) is “Vector append”

Vector Details: add(E)

Arrays don't have any notion of "appending”. add(E) is “Vector append”

- What does it mean to “append” to a Vector?

Vector Details: add(E)

Arrays don't have any notion of "appending”. add(E) is “Vector append”

- What does it mean to “append” to a Vector?

When we think about performance, we often care most about the "worst case"

Vector Details: add(E)

Arrays don't have any notion of "appending”. add(E) is “Vector append”

- What does it mean to “append” to a Vector?

When we think about performance, we often care most about the "worst case"

- What are the "worst cases" that we need to consider when appending to a Vector?
    - If the Vector’s internal array has room, we can just place the element at the first free index, and increment the count
    - If the Vector’s internal array is full, we need to GROW! This means creating a larger array, copying everything into it, then adding the new element.
        - How big should we make the new array?

Vector Details: add(int, E)

Arrays don't have any notion of "inserting”. add(int, E) inserts at index i

Vector Details: add(int, E)

Arrays don't have any notion of "inserting”. add(int, E) inserts at index i

- What does it mean to insert into the middle of a Vector?

Vector Details: add(int, E)

Arrays don't have any notion of "inserting”. add(int, E) inserts at index i

- What does it mean to insert into the middle of a Vector?

Unlike an array that overwrites the element at a given index, a Vector “creates room”, then adds the element in that newly emptied space

Vector Details: add(int, E)

Arrays don't have any notion of "inserting”. add(int, E) inserts at index i

- What does it mean to insert into the middle of a Vector?

Unlike an array that overwrites the element at a given index, a Vector “creates room”, then adds the element in that newly emptied space

- How do we create room in the Vector’s internal array?
    - Shift all elements *after* the insertion point one space to the right

Vector Details: size()

Vector Details: size()

Vector size is different than Vector capacity.

Vector Details: size()

Vector size is different than Vector capacity.

- Size is how many elements are *currently* in the underlying Object array

Vector Details: size()

Vector size is different than Vector capacity.

- Size is how many elements are *currently* in the underlying Object array
- Capacity is the length of the underlying array

Vector Details: size()

Vector size is different than Vector capacity.

- Size is how many elements are *currently* in the underlying Object array
- Capacity is the length of the underlying array
- How do they differ?
  - The array may not be full! (Note: size <= capacity)
  - As we add and delete elements, size will fluctuate, but array size cannot change.
  - We may “grow” or “shrink” our array by creating a new array and copying items
    - When/how we do this has huge implications on performance! We’ll dive into this in another video

More Vector methods

More Vector methods

- contains(E)

More Vector methods

- contains(E)
- indexOf(E)

More Vector methods

- contains(E)
- indexOf(E)
- remove(E)

More Vector methods

- contains(E)
- indexOf(E)
- remove(E)

You *could* implement methods to do these on arrays,
but vectors come with them built in.

More Vector methods

- contains(E)
- indexOf(E)
- remove(E)

You *could* implement methods to do these on arrays, but vectors come with them built in.

What should they do? How can we implement them in our Vector class?

Vector Details: contains(E)

contains(E) determines if a value appears in the Vector

Vector Details: contains(E)

contains(E) determines if a value appears in the Vector

- What does it mean for a value to "appear in" a Vector?

Vector Details: contains(E)

contains(E) determines if a value appears in the Vector

- What does it mean for a value to "appear in" a Vector?
  - elementData[i].equals(obj) == true   (for some index i)
  - Note: indexOf(E) is similar, except it returns the index i, or -1 if not found

Vector Details: contains(E)

contains(E) determines if a value appears in the Vector

- What does it mean for a value to "appear in" a Vector?
  - elementData[i].equals(obj) == true (for some index i)
  - Note: indexOf(E) is similar, except it returns the index i, or -1 if not found

- What if there are multiple copies of the target value?

  - No worries! We just return true as soon as we find the first occurrence

Vector Details: contains(E)

contains(E) determines if a value appears in the Vector

- What does it mean for a value to "appear in" a Vector?
  - elementData[i].equals(obj) == true   (for some index i)
  - Note: indexOf(E) is similar, except it returns the index i, or -1 if not found

- What if there are multiple copies of the target value?

  - No worries! We just return true as soon as we find the first occurrence

- Note that contains uses .equals, and we can only call .equals on Objects.

  - We can’t store primitive values in a vector!

Vectors, generic types, and `equals()`

- We store generic object types in our internal array
- We use `equals()` (for whatever type E we are storing in our vector) to see if two things are the same
- Therefore, cannot store primitive types in a vector!

- Need to use something like `Vector<Integer>` instead

Vector Details: remove(E)

remove(E) removes the first occurrence of a value from the Vector

Vector Details: remove(E)

remove(E) removes the first occurrence of a value from the Vector

- Similar to contains: search using equals() to find a match

Vector Details: remove(E)

remove(E) removes the first occurrence of a value from the Vector

- Similar to contains: search using equals() to find a match
- What if there are multiple copies of the target value?
  - Delete the first. We stop as soon as we remove the first occurrence

Summary

Summary

Vectors are random-access data structures, like an array, but they add new functionality

Summary

Vectors are random-access data structures, like an array, but they add new functionality

- Inserting/Removing

Summary

Vectors are random-access data structures, like an array, but they add new functionality

- Inserting/Removing
- Resizing

Summary

Vectors are random-access data structures, like an array, but they add new functionality

- Inserting/Removing
- Resizing
- Searching

Summary

Vectors are random-access data structures, like an array, but they add new functionality

- Inserting/Removing
- Resizing
- Searching
- Support for “generic” types

Summary

Vectors are random-access data structures, like an array, but they add new functionality

- Inserting/Removing
- Resizing
- Searching
- Support for “generic” types

Summary

Vectors are random-access data structures, like an array, but they add new functionality

- Inserting/Removing
- Resizing
- Searching
- Support for “generic” types

The Vector class implements many functions that we will revisit when we discuss the abstract concept of a “List”