CS 2100: Data Structures & Algorithms 1

Introduction to Vectors

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

A Bit More Polymorphism

With declaring/instantiating/initializing objects

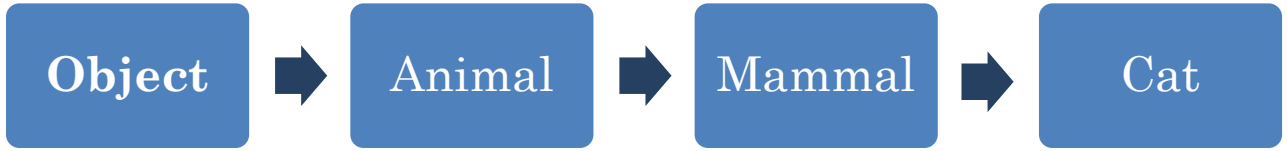
With method parameters

More Polymorphism Examples

- `List myList = new Vector(); //why does this work?`
- `Object something = new String(); //a string IS an object`
- `List myList2 = new List(); //does NOT work, why?`
- `Vector myList3 = new List(); //does NOT work, why?`

Quick: Inheritance

- Inheritance: **is-a** relationship (superclass/subclass)



  - `public class Mammal extends Animal { }`

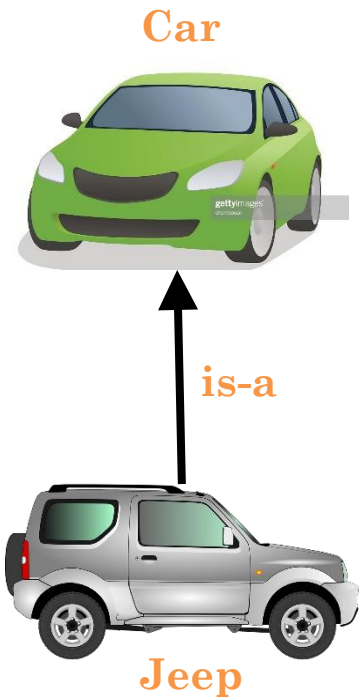
- Inheritance applies to Interfaces
  - The interface is the superclass, the class that implements the Inteface is the subclass
  - A **subclass** IS-A kind of **superclass**
  - E.g.,
    - a Vector IS-A kind of List (where **List** in an Interface)
    - a GrandfatherClock IS-A kind of TimeKeeper (where **TimeKeeper** is an Interface)

Substitutability Principle

- We say: any subclass object (e.g., Jeep) **is-a** instance of a superclass object (e.g., Car), and **inherits** its states and behaviors

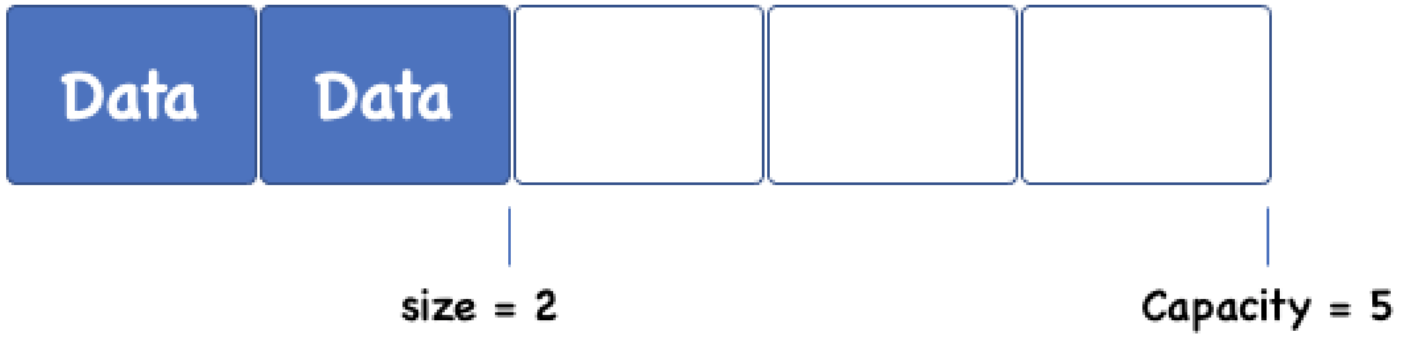
- Wherever we see a reference to a Car (superclass) object in our code, we can legally replace that with a reference to Jeep (any subclass object)

- Implies that we can **substitute** the subclass object in any way that's legal for the superclass



Given that background, … back to Polymorphism

- General rule about this in Java is:
  - if the variable being assigned (or parameterized) is a MORE general version of the original, **then it is allowed**.
  - Likewise, you CANNOT refer to a general object as a more specific type.
    - e.g., If I say give me an animal, you can give me a cat (no problem). but if I say give me a cat, you cannot give me any Animal.



What is a Vector?

A specific type of List

What is a Vector?

- Our first **specific type of List**
- Motivation for creating a Vector?
  - Make arrays a bit better
  - Arrays have a fixed size
    - Vector: Would be nice if I could just add elements at will and the array would grow automatically
  - Arrays need a specific size
    - Vector: No need to specify a size when creating the list (I may not know yet)

- Vector:
  - A resizable array
  - Automatically grows and shrinks as you add or remove items
  - In reality: simulates this using fixed size arrays

Imports

- Java has two primary built in vector classes you can use:
  - **Vector** (`import java.util.Vector`) and
  - **ArrayList** (`import java.util.ArrayList`)
  - Can use: import java.util.* (This means you import ALL of java.util)

  - See the Java API for list of methods!

Java Built-in Vectors

- Notice the data types are different
- We'll discuss this on Friday ("Generics")

```
// Vector
Vector<Integer> list1 = new Vector<Integer>();
list1.add(5);
System.out.println(list1.get(0)); // first element

// ArrayList
ArrayList<Double> list2 = new ArrayList<Double>();
list2.add(3.45);
System.out.println(list2.size()); // how many elements
```

Vector Basics

- **size**: *an attribute (simple variable)*
  - The number of elements that have been added to the Vector
  - [Used when simulating a Vector using an array as the underlying data structure]
- **capacity**: *an attribute (simple variable; can be a constant)*
  - The size of the underlying array (maximum number of elements it can contain)
  - Note: `size <= capacity`
  - [Used when simulating a Vector using an array as the underlying data structure]
- **resize**(): *a method*
  - A **private method** that doubles the size of the underlying array
    - This allows the Vector to grow automatically (when needed)
    - Automatically invoked when underlying array fills up

Vectors in General

- If building your own Vector, you would have to build the following (from List interface):
- For now, let's suppose this Vector stores doubles only *(will change in a bit)*
- **find()** – finds the index of value in the Vector (represented by an array "theList")

```
private double theList[];   // length of this IS the capacity
private int size;           // the ACTUAL size of the vector

/* Finds the index of value in theList, if present */
public int find(double value) {
    for(int i = 0; i < this.size; i++) {
        if(theList[i] == value) // they match
            return i; // return the index at which the element was found
    }
    return -1; // sentinel value (I didn't find it!)
}
```

Vectors in General

- **setAt()** – write a value to a particular index location in the Vector
- **getAt()** – get the item at the specified index in the Vector

```
/* Overwrites the item at a specified index */
public void setAt(int index, double value) {
    if(index >= 0 && index < this.size) // index within range
        theList[index] = value; // overwrite the item in this position with value
}
/* Returns the item at the specified index */
public double getAt(int index) {
    if(index >= 0 && index < this.size) // index within range
        return theList[index];
    else
        return 0.0; // Uh oh, we went off the bounds of the Vector!  (Return something)
}
```

Vectors in General

- **resize()** – doubles the size of the underlying array (has to make a new one!)

```
/* Doubles the size of the underlying array by making a new one */
/* Why double? We will see later! */
public void resize() {
    // Make a new array of size theList.length*2
    // Copy everything over from theList to the new one
    // Make theList equal to the newly created array instead
}
```

Vectors in General

```java
/* Inserts value into this list at the end */
public void insert(double value) {
    // If the list is full, call resize() to make it bigger
    // Then, add the element at index size
    // size++ (increment size by 1)
}

/* Inserts at the specified index */
public void insert(double value, int index) {
    // If index out of bounds, do nothing (just return;)
    // Call resize() if necessary
    // Move everything from index onward up by one position
    // Add the element at the index
    // size++ (increment size by 1)
}

/* Finds the value and removes it from the list */
public void remove(double value) {
    // Call find() to get the index of the value
    //      if it exists
    // Move everything from index+1 onward down one spot
    // size-- (decrement size by 1)
}
```

- **insert(double value)** – insert at the end
- **insert(double value, int index)** – insert at the specified index
- **remove(double value)** – finds and removes the value from the list

Vector Strengths

- Programmer does NOT need to worry about size of list. The list grows and shrinks automatically
- Still very fast (constant time) to **access** a specific element of the list because array get (e.g., theList[i]) is a fast operation
- VERY fast (constant time) if **inserting** / **removing** from the back of the list
- Works well with cache because arrays stored **contiguously** in memory

Vector Weaknesses

- Takes up more space than is actually being used *(most of the time)*.
  - *Remember, size is not the same as capacity, but the space taken is always the capacity*
  - i.e., the size is almost always less than the capacity
- Slow (linear time) if **inserting** or **removing** from indices NOT at the back of the list because the vector has to shift everything else one spot to account for the change
- Slow (linear time) every once in a while when the **vector needs to grow**.