Java Arrays

- **Fixed size, once created**
  - Can hold primitive types
  - Can hold objects (references)
- **Example: Creating an array of doubles**
  ```java
double[] times;
times = new double[30]; // or could combine w prev
  ```
- **Example: Creating an array of DLicenses**
  ```java
DLicense[] dls;
dls = new DLicense[50]; // create array (or combine)
for (int k; k < dls.length; k++) {
    dls[k] = new DLicense(); // create objects in dls
}
```

Java ArrayList Class

- **Flexible Arrays**
  - Grows in size as needed!
  - Many different methods to improved array processing
- **Create with:**
  ```java
ArrayList vect = new ArrayList();
```
- **Uses:** (assume dl, sl, are DLicense objects)
  ```java
vect.add(dl); // add to “end”
vect.add(k, dl); // insert at position k (shifts)
sl = (DLicense) vect.get(m); // retrieve from
// position m – note cast to DLicense
```
- **Note that [] brackets don’t work**
  - Also see: remove(), indexOf(), toArray(), contains(), size(), … Look them up!

Java Arrays

- **Can also create arrays by specifying initial values**
  - Avoids need for new
  - Avoids need to count the number of values
- **Example: Creating an array of ints**
  ```java
int[] counts = { 3, 12, 0, 8, 10}; // can use counts.length to get size of array
  ```
- **Example: Creating an array of Strings**
  ```java
String[] aHotel = {"Hilton", "Swans", "Astoria"};
String[] bHotel = {"Kwik8", "SleepyT", "TuckUIn"};
String[] cHotel = {"DiveX", "RRXing", "Swampys"};
  ```
- **Example: Creating an array of arrays (matrix)**
  ```java
String[][] hotelChoice = {aHotel, bHotel, cHotel};
```

Solving Problems: Anagrams/Jumbles

- **How do humans solve puzzles like that at [www.jumble.com](http://www.jumble.com)**
  - Is it important to get computers to solve similar puzzles? Reasons?
  - Should computers mimic humans in puzzle-solving, game playing, etc.? Lessons from chess?
  - nelir, nelri, nieir, nipri, nipriil, nleri, nlier, niler, nilri, nlie, nieir, nieir, niler, nilre, nirel, ... lenir, lenri, leinr, leinr
  - What’s the problem here?
Brute force? SillyAnagrams.java

```java
public String[] allAnagrams(String s) {
    int anaCount = factorial(s.length());
    Set anagrams = new TreeSet();
    ArrayList list = new ArrayList();
    for(int k=0; k < s.length(); k++) {
        list.add(s.substring(k, k+1));
    }
    while (anagrams.size() != anaCount) {
        Collections.shuffle(list);
        anagrams.add(listToString(list));
    }
    return (String[]) anagrams.toArray(new String[0]);
}
```

Quantifying brute force for anagrams

- All anagrams of "compute" takes average of 1 second over 20 trials. How long will "computer" take? Why?
  - What is worst case time?
  - What is best case time?

- We’re willing to do some pre-processing to make the time to find anagrams quicker
  - Often find that some initialization/up-front time or cost (investment?) saves in the long run
  - What properties do words share that are anagrams?

Toward a faster anagram finder

- Words that are anagrams have the same letters; use a letter fingerprint or signature/histogram to help find anagrams
  - Count how many times each letter occurs:
    - “teacher” 1 0 1 0 2 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0
    - “cheater” 1 0 1 0 2 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0

- Store words, but use fingerprint for comparison when searching for an anagram
  - How to compare fingerprints using .equals()
  - How to compare fingerprints using .compareTo()

- How do we make client programmers unaware of fingerprints? Should we do this?

Another anagram method

- Instead of fingerprint/histogram idea, use sorted form of word
  - “gable” and “bagel” both yield “abegl”
  - Anagrams share same sorted form

- Similarities/differences to histogram/fingerprint idea?
  - Both use canonical or normal/normalized form
  - Normalized form used for comparison, not for printing
  - When should this normal form be created?

- When is one method preferred over the other?
  - Big words, little words? Different alphabets? DNA vs English?
**OO and Java**

- We’ll use an *adapter* or *wrapper* class called *Anaword* instead of *String*
  - Clients can treat Anaword objects like strings, but the objects are better suited for finding anagrams than strings
  - The Anaword for “bear” prints as “bear” but compares to other Anaword objects as `11001000000000000100000000`

- In Java change behavior with `.toString()` and `.equals()`
  - No overloaded operators as in C++
    - Exception is +, this works for strings, but can’t change it
  - When string needed, automatically call `toString()`

**Understandable, extensible?**

- The code does things simply, but isn’t very OO. Why is simple sometimes better? Why is it worse?

```java
void printAll(Anaword[] list, Anaword target) {
    System.out.print("anagrams of "+target+": ");
    for(int k=0; k < list.length; k++){
        if (target.equals(list[k])) {
            System.out.print(list[k]);
        }
    }
    System.out.println();
}
```

**Find all anagrams in dictionary**

- If we sort the dictionary what will happen to the anagrams?
  - capitol optical topical
danger gander garden ranged
  - lameness maleness nameless salesmen

- How can we overload `.equals()`?
  - Look at "danger" or `1001101000000100010…`
- How can we sort with Collections.sort or Arrays.sort
  - Elements sorted must be comparable sortable
  - Must implement the java.lang.Comparable interface
    - Return negative, zero, positive number depending on less than, equal to, or greater than
    - What is method signature?

**Anaword objects with options**

- Can we use different canonical forms in different contexts?
  - Could have Anaword, FingerPrintAnaword, SortAnaword
  - What possible issues arise? What behavior is different in subclasses?
    - If there’s no difference in behavior, don’t have subclasses

- Alternative, make canonical/normalize method a class
  - Turn a function/idea into a class, then let the class vary to encapsulate different methods
  - Normalization done at construction time or later
  - Where is normalizer object created? When?
Anagram: Using Normalizers

- How can we normalize an Anaword object differently?
  - Call normalize explicitly on all Anaword objects
  - Have Anaword objects normalize themselves
  - Advantages? Disadvantages?

- If Anaword objects normalize themselves, how can we experiment with different normalization techniques?
  - Cut and paste. Problems? Versions? Saved code?
  - What about using save-as and several .java files?
  - What about deciding at runtime on normalization?

- We need inheritance!

Normalizer hierarchy

- Anaword objects normalize themselves
  - Where does the normalizer come from?
    - Passed in at construction time
    - Obtained from normalizer factory
    - Other approaches?
  - How is Normalizer used?

- Normalizer is conceptually an interface
  - Different implementations of the interface have different behavior (guts) but same skin (sort of)