First, more on FrequencyCounter: Setting up a WordCounter service

- FrequencyCounter uses a HashMap<String, Integer> to store words and counts for them right in the one program.
- Let's split this up ("refactor it") into two parts:
  - Top-level code (the main()), now FrequencyCounter1.java
  - A WordCounter service that does the details work.
- Project 0 note: WordCounter, like LineUsage, has an instance variable of type HashMap<String, Integer>.

Note on example

- Admittedly, this is not needed for such a simple program, but it shows how to do such a refactoring.
- It shows to use a Map<String, Integer> as an instance variable in a class to allow its object to hold significant information, easily.
- Note that Patrick's LineUsage uses such a map to hold, for one terminal line, all the usernames and counts for them.

What is the API between the parts?

- WordCounter will hold the HashMap<String, Integer> with all the details.
- What does the top-level code need?
  - It needs the Set of words and the count of each.
  - So we have an API to invent…
What is the API between the parts?

- We need a constructor that accepts an InputStream and uses that input to fill a Map.
  - public WordCounter(InputStream in)
- We need a method to return the Set<String> for all the words
  - public Set<String> getWords()
- We need a method to return a count for a word.
  - public int getCount(String word)

Simpler top-level code

private Map<String, Integer> wordCounts;
// read from in, fill map
public WordCounter(InputStream in) {
  // what goes here? Needs to load up wordCounts map
  // return the set of words
  public Set<String> getWords() {
    // what goes here?
    return wordCounts.keySet();
  }
  // return a count for a word
  public int getCount(String word) {
    // what goes here?
    return wordCounts.get(word);
  }
}

WordCounter API in code skeleton

private Map<String, Integer> wordCounts;
// read from in, fill map
public WordCounter(InputStream in) {
  // what goes here? Needs to load up wordCounts map
  // return the set of words
  public Set<String> getWords() {
    // what goes here?
    return wordCounts.keySet();
  }
  // return a count for a word
  public int getCount(String word) {
    // what goes here?
    return wordCounts.get(word);
  }
}

WordCounter API code
WordCounter Constructor

```java
private Map<String, Integer> wordCounts;
// read from in, fill map
public WordCounter(InputStream in) {
    wordCounts = new HashMap<String, Integer>();
    Scanner scan = new Scanner(in); // uses whitespace delim.
    while (scan.hasNext()) { // Build symbol table and count frequencies
        String word = scan.next();
        // if (word.length() < minLen) continue;
        if (!wordCounts.containsKey(word))
            wordCounts.put(word, 1);
        else
            wordCounts.put(word, wordCounts.get(word) + 1);
    }
    scan.close();
}
```

Same as original input loop in FrequencyCounter, except it's dropping word-length filtering (see commented-out line above), and the map variable has a different name.

Hashing – quick intro

- Hash function properties:
  1. Map key elements to integers.
  2. Fast to calculate.
  3. Try to minimize collisions – when different keys hash to the same value.

Hashing: uses array O(1) lookup

- Hashing is a technique for fast lookup by key.
- It's related to the idea of keeping an array (lookup table) with a subscript for every possible value we might want to look up.
- Say we have a Map with 2000 integers in the domain, with values 0-1999. We can create a 2000 element array a[] and look up the range entry for value i in a single reference to the array, a[i], itself a pointer or reference.
- Array lookup is done by computed address: addr = start_address + size_of_entry * index, so O(1) performance.
- This is a lookup in O(1) time, put is also in time O(1), as is remove (set a[i] to null).

Simple Example of hashing.

- We have a map of int to int with 4 -> 100, 55 -> 44, 10 -> 12
- Here 4, 55, and 10 are the keys.
- The hash function is h(x) = x/10, for hashing the keys.
- So h(4) = 0, h(55) = 5, h(10) = 1 -- bucket numbers 0, 5 and 1
- 4 hashes to bucket 0, 55 hashes to 5, and 10 hashes to 1.
- Hash table: Set up array of 10 spots, put the (key, value) pairs in the array by hash bucket:
  a[0] = (4, 100) --ref to object containing 4, 100
  a[1] = (10, 12)
  a[2] = null
  ...  
  a[5] = (55, 44)

Hashing – less trivial examples

- For large, sparse domains, this plain-array approach is impractical.
- With a larger domain, like 1..1000000 with only 100 values in use we can still set up an array.
- Wastes memory but gives us O(1) lookup, Insert, and Delete.
- What if the domain is not integers at all?
- Solution: We map the domain objects to integers with a more complicated function called the hash function.
- The hash function computes the “bucket number”, itself an array index, and the compiled code will find the array element by calculating: addr = start_address + index*size_of_entry
Example of hashing (cont.)

- Look up 55: \( h(55) = 55/10 = 5 \), \[5][\] = ref to \((55, 44)\), 55 matches, so value = 44
- Look up 56: \( h(56) = 56/10 = 5 \), \[5][\] = ref to \((55, 44)\), no match so value not there (value = null)
- Luckily, the quick example has no “collisions” (two keys hashing to the same bucket).
- The above example is “hashing integers”. Similarly we can hash strings by coming up with a function that maps strings into bucket numbers.

Implementing maps using hashing

- Example: Given a string, count the occurrence of the 5 English vowels, using map from chars to ints.
  - 'a' \( \rightarrow \) count of a's
  - 'e' \( \rightarrow \) count of e's
  - 'i' \( \rightarrow \) count of i's
  - 'o' \( \rightarrow \) count of o's
  - 'u' \( \rightarrow \) count of u's

Recall FrequencyCounter's similar key \( \rightarrow \) count Map

Hashing terminology

- **Keys**: each value of type keytype can be called a key. It just means that we’re going to do a look-up using this value.
- **Hash table**: the array in use, of some size \( M \).
- **Hash bucket or hash slot**: a subscript in the hash table array, these are numbered from 0 to \( M - 1 \). \( M \) is the number of buckets.
- **Hash function**: a function from the keytype to a bucket-number: \( b = h(x) \), where \( x \) is of type keytype and \( 0 \leq b < M \) is the bucket number. We say “\( x \) hashes to \( b \)”. \( h(x) \) is a computable expression (some formula) and is expected to take \( O(1) \) computation time.
- **Collision**: when two keys \( x \) and \( y \) hash to the same bucket: \( b = h(x) = h(y) \).

Vowel example

```java
String s = “this is a test”; // string to count vowels in
// set up HashMap for stats
Map<Character, Integer> stats = new HashMap<Character, Integer>();
// “write your code here”
for (int i = 0; i < s.length(); i++) {
    c = s.charAt(i);
    Integer count = stats.get(c);
    // get Integer object, so can test if null
    if (count != null) // if vowel – found in map
        stats.put(c, count + 1); // count is auto-boxed, then
        // 1 is added, result boxed up
    }
    // print “w’s” + stats.get(‘w’) + “’s: “ + stats.get(‘e’) + “’s: “ + stats.get(‘i’) + “’s: “ + stats.get(‘o’) + “’s: “ + stats.get(‘u’);
// Rough picture of
// HashMap holding Map of Characters to Integers: each bucket has a list of pairs
```

Maps using hashing – vowel example

- How does HashMap work in this case?
  - The Character class has hashCode() already implemented and in use for the code on the last slide. But how does it work?
  - In homework 1, we saw that x.hashCode() = \( x \) for Integer
  - x.hashCode() = \( x \) for Character too, and that value is the ASCII code.
  - 'a' = 97 = 0x61, 'e' = 101 = 0x65, 'i' = 0x69, 'o' = 0x6f, 'u' = 0x75
- Use \( M = 16 \), the default table size of HashMap and \( h(x) = x \mod M \), as done in HashMap. Then \( x \mod M = x \mod 16 \) is just the last hex digit of \( x \).
  - So, made HashMap: \( h(2) = 2, h(5) = 5, h(7) = 7, h(8) = 8, h(10) = 10 \).
- One collision! Not wonderful, but not too bad.
Maps using hashing

- The hash table itself is hidden inside the HashMap implementation.
- Note: there might be collisions in the HashMap. It's OK, though, because HashMap takes appropriate action.
- hashCode() for key type. Only needs to provide an int. HashMap will scale it to the right array size.
- hashCode() vs. h(): hashCode yields an int, h() yields a bucket number, so h(x) = x.hashCode()%M for HashMap/HashSet with hash table of size M.
- Rule of Thumb: Try for only half-full (or less) hash tables to minimize collision on one hand and save space on the other hand. HashMap automatically grows its table as needed.

Somewhat bad hashing function

```java
public static int hash(String key)
{
  int hashVal = 0;
  for(int i=0;i<key.length();i++)
    hashVal += key.charAt(i);
  return hashVal;
}
```

Advantages:
1. Uses all the available information.
2. Simple to calculate.

Problems:
1. Returns same value for words like "bat" and "tab".

Hashing Strings

- It's very important to let all parts of the string contribute to the result.
- Think of hashing URLs, for ex., "http://www..." Better not be using just the first 12 chars!

Hashing Strings: Java's way

```java
int hash(String key)
{
  int hashVal = 0;
  for(int i=0;i<key.length();i++)
    hashVal += key.charAt(i);
  return hashVal;
}
```

- Then inside HashMap:
  ```java
  hashVal %= tableSize; // make positive
  ```

- Note on S&W pg. 479 says we can't use Math.abs here because it can return negative!
- Code on pg. 460 uses "tablesize during loop, but Java doesn't (in hashCode) for String". It's HashMap itself that does the "tableSize; modifying the hashCode it gets from the key.

Hashing Strings: Java's way

```java
int hash(String key)
{
  int hashVal = 0;
  for(int i=0;i<key.length();i++)
    hashVal += key.charAt(i);
  return hashVal;
}
```

- Then inside HashMap:
  ```java
  hashVal %= tableSize; // make positive
  ```

- Note on S&W pg. 479 says we can't use Math.abs here because it can return negative!
- Code on pg. 460 uses "tablesize during loop, but Java doesn't (in hashCode) for String". It's HashMap itself that does the "tableSize; modifying the hashCode it gets from the key.

FYI: Hashing Special Strings

How to get O(1) lookup performance by almost magical means

Consider id's A1, A2, ..., A9, B1, B2, ..., B9, C1, ..., C9, D1, ..., D9
Easy to map: X' = 0, 'Y' = 1, 'Z' = 2, 'D' = 3 by num1 = ch1 - 'A'.
Also 'A' = 1, 'B' = 2, ..., 'Y' = 9 by num2 = ch2 - 'A'.
Here num1 goes from 0 to 9, num2 goes from 1 to 9, so 36 cases in all.
How can we combine them into one number that goes from 0 to 35, or 0 to 39?
Answer, use code = num1*10 + num2 (or num1 + 4*num2)
   i.e., code = 10*(ch1 - 'A') + ch2 - 'A'
Then each id maps to a unique number: that's a "perfect hash":
   "A1" = 10*(9 - 'A') + 1 - 'A' = 1, "A2" = 2, ..., "A9" = 9,
   "B1" = 10*(8 - 'A') + 2 - 'A' = 11, "B2" = 12, ..., "B9" = 19, ...
   "D1" = 31, ..., "D9" = 39
FYI: Finding the perfect hash

• We have mapped all these patterns into \([1, 39]\), with all different hash function values. Some hash function values aren’t used, but that’s OK, since there aren’t a lot of them. We can use these range values as unique ids of these strings.
• We have a perfect hash, so a simple array can be used as a mapping table, since no collisions occur.
• For example, to map “B2” to 100, set \(a[2] = 100\)
• Note: in Java applications, we usually don’t worry about finding a perfect hash, since the non-perfect ones work so well, and we have HashSet and HashMap to do all the collision handling for us.

We can use these range values as unique ids of these strings.

Hashing more complex or larger objects

• Graphics bitmaps are sometimes hashed to identify and classify them—think of them as strings with binary codes.
• Complex Java objects often have an identifier in them, and that is what is hashed.
• Hashing implements fast look-up, so we only want to hash the things we want to look up by.
• Note that graphics bitmaps often don’t have a natural identifier, so we use their contents to id them for want of a better method.

Hashing more complex or larger objects

• Example: Employee record containing first name, last name, SSN (or empid, assigned), address, dept...
• We hash by SSN. They have max \(999,999,999 = 999,999,999 \times 1G\), so they fit nicely in 32-bit numbers.
• For hashCode() of Employee, just return the int SSN, and for equals, compare int SSNs
• object with a String id—just use String’s hashCode().

Hashing more complex or larger objects - Example

• Another source of unique ids, if the data is coming from a database, are the database “primary key” values, since they are guaranteed unique by the database.
• If using firstName, lastName as id, with equals requiring both to match, we could concatenate the two Strings and then do hashCode(), or add up the two hashCodes or XOR them.
• Java7 new feature: Objects.hash(firstName, lastName) computes a hashCode from instance variables firstName and lastName. See S&W pg. 478.