Programming Languages and Techniques (CIS120)

Lecture 36

April 17, 2013

Hashing, Design Exercise

Hashing

Hash Maps: The Big Idea

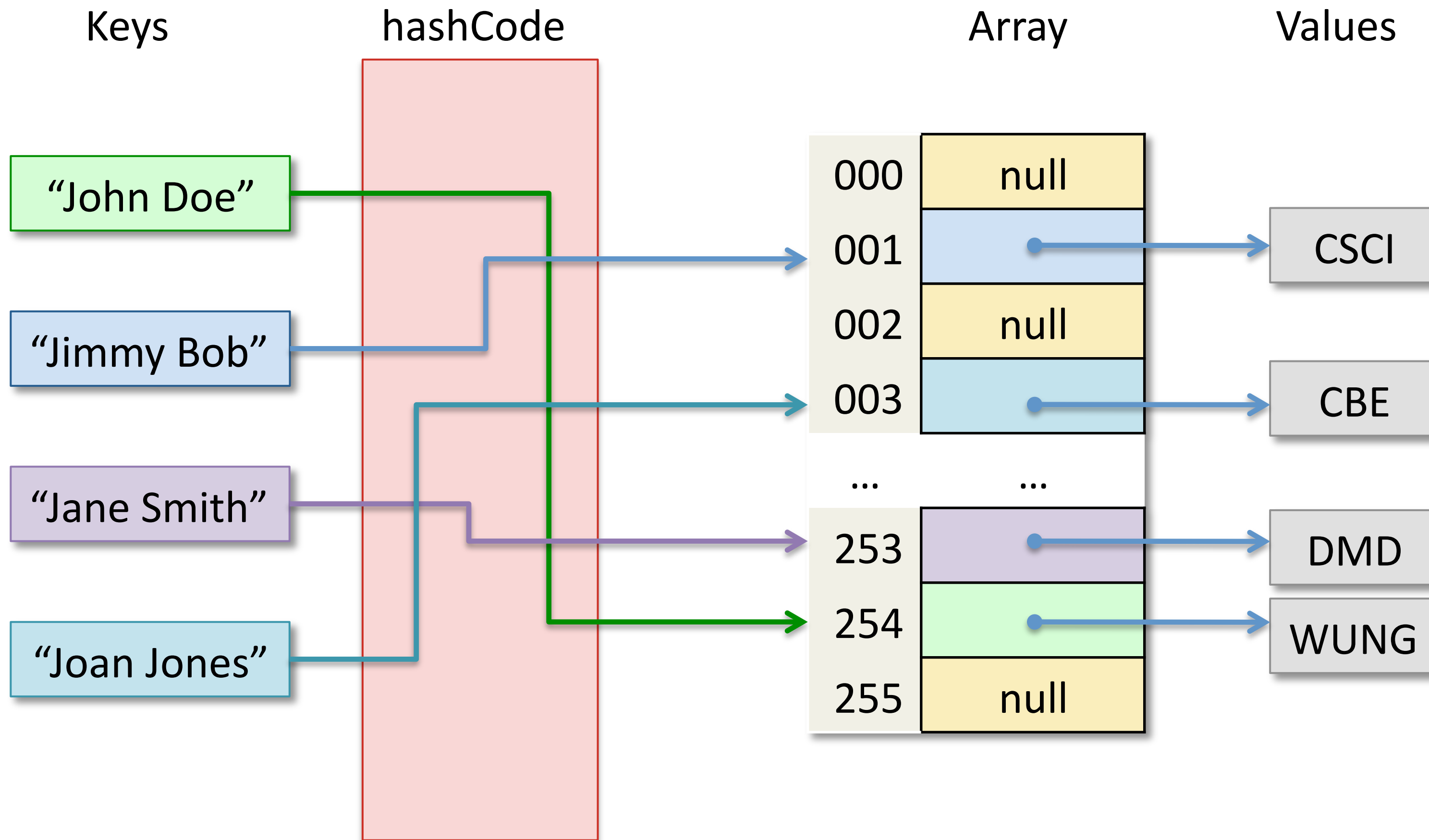
Combine:

- the advantage of arrays:
  - *efficient* random access to its elements
- with the advantage of a map datastructure
  - arbitrary keys (not just integer indices)

How?

- Create an index into an array by *hashing* the data in the key to turn it into an int
  - Java's hashCode method maps key data to ints
  - Generally, the space of keys is much larger than the space of hashes, so, unlike array indices, hashCodes might not be unique

Hash Maps, Pictorially



A schematic HashMap taking Strings (student names) to Undergraduate Majors. Here, `"John Doe".hashCode()` returns an integer n, its *hash*, such that n mod 256 is 254.

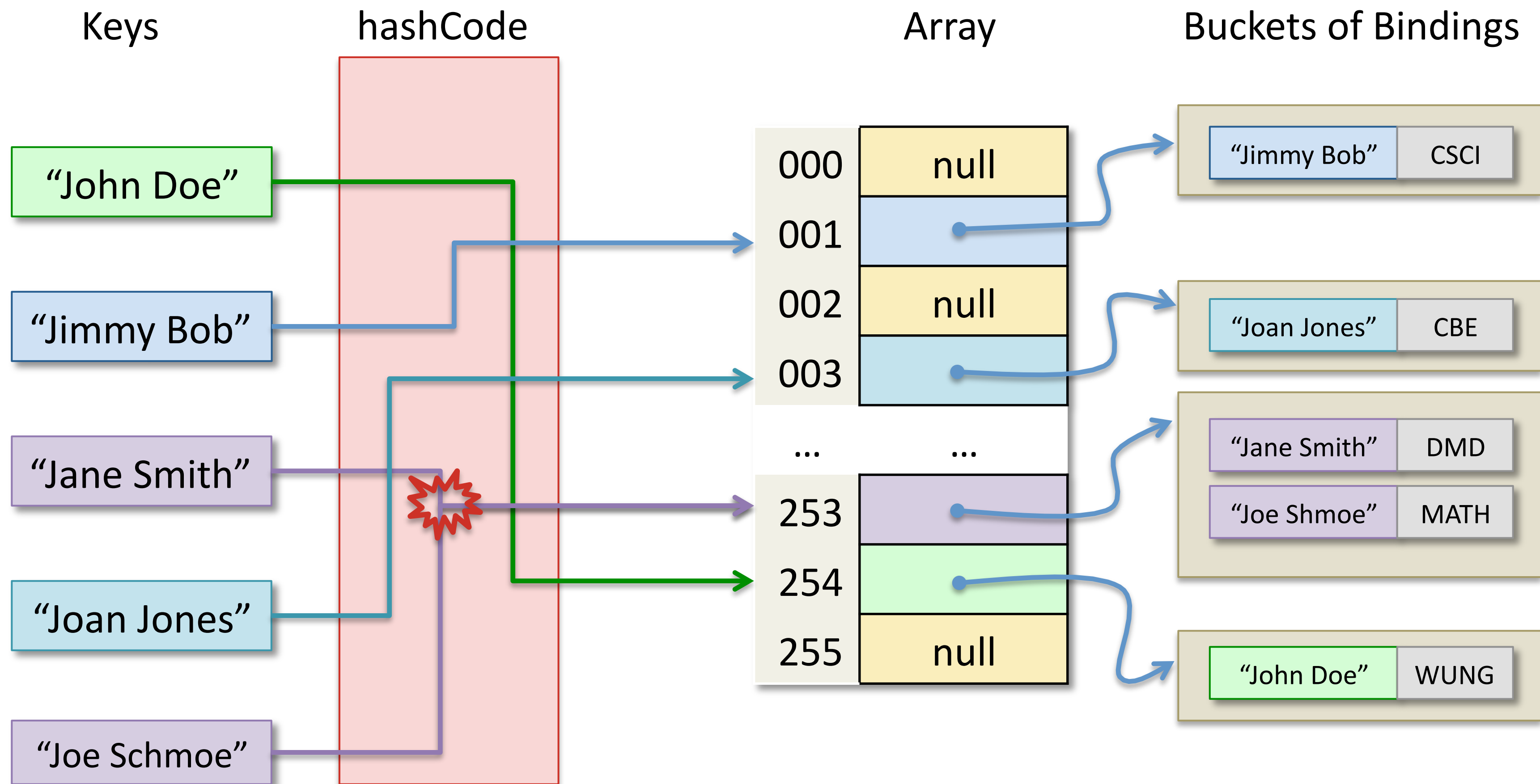
Hash Collisions

- The hashCode function should be chosen so that it is unlikely that two keys will produce the same hash.
  - However, it can happen that two keys do have the same hash value – that is, their hashes *collide*

- Hash Map data structure implementations must handle such collisions to preserve the "map" semantics… there are many possible solutions.
- One simple fix: array of *buckets*
  - Each bucket is itself a map from keys to values (implemented by a linked list).
  - Each bucket stores the mappings for keys that have the same hash.
  - The buckets can't use hashing to index the values – instead they use key equality (in Java, via the key's equals method)

- To lookup a key in the Hash Map:
  - First, find the right bucket by indexing the array through the key's hash
  - Second, search through the bucket to find the value associated with the key

Bucketing and Collisions



Here, "Jane Smith".hashCode() and "Joe Schmoe".hashCode() happen to collide. The bucket at the corresponding index of the Hash Map array stores the map data.

Hash Map Performance

- Hash Maps can be used to efficiently implement Maps and Sets
  - There are many different strategies for dealing with hash collisions with various time/space tradeoffs
  - Real implementations also dynamically rescale the size of the array (which might require re-computing the bucket contents)

- If the hashCode function gives a good (close to uniform) distribution of hashes the buckets are expected to be small (only one or two elements)

Whenever you override `equals` you must also override `hashCode` in a consistent way:

- whenever `o1.equals(o2)== true` you must ensure that `o1.hashCode() == o2.hashCode()`
- note: the converse does not have to hold:

Why? Because comparing hashes is supposed to be a quick approximation for equality.

Computing Hashes

- Java library classes come equipped with a good hashCode method
  - e.g. String
- What is a good recipe for computing hash values for your own classes?
  - intuition: "smear" the data throughout all the bits of the resulting integer

1. Start with some constant, arbitrary, non-zero int in `result`.
2. For each significant field f of the class (i.e. each field taken into account when computing equals), compute a "sub" hash code `c` for the field:
   - For boolean fields: `(f ? 1 : 0)`
   - For byte, char, int, short: `(int) f`
   - For long: `(int) (f ^ (f >>> 32))`
   - For references: 0 if the reference is null, otherwise use the `hashCode()` of the field.
3. Accumulate those subhashes into the result by doing (for each field's `c`):
   `result = 31 * result + c;`
4. return `result`

Example for Point

```
public class Point {
  …
  @Override public int hashCode() {
    int result = 17;
    result = result * 31 + getX();
    result = result * 31 + getY();
    return result;
  }
}
```

- Examples:
  - (new Point(1,2)).hashCode() yields 16370
  - (new Point(2,1)).hashCode() yields 16400
- Double check that equal points have the same hashCode
  - Trivial in this case.
- Why 17 and 31? Primes chosen to create more uniform distributions.

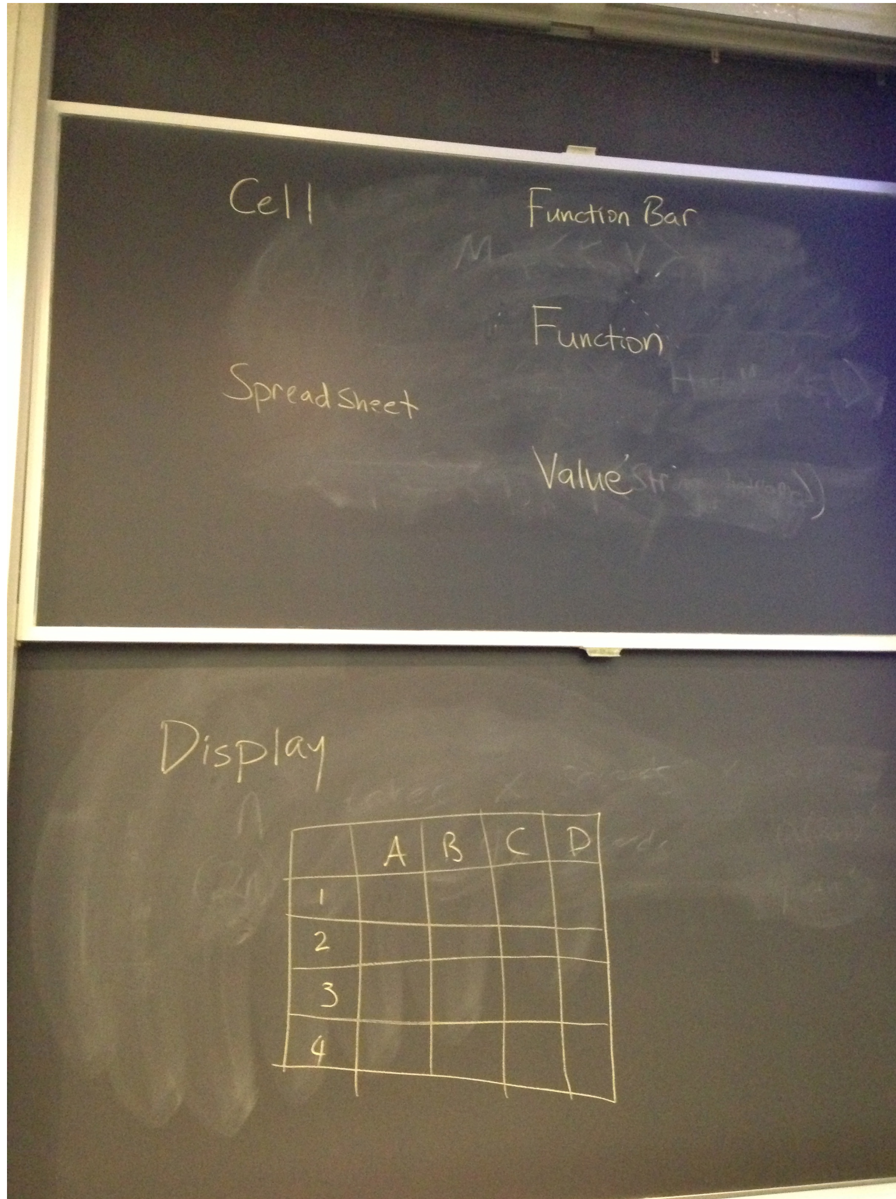
Spreadsheet design exercise

How does the design recipe scale to larger, graphical projects?

Design Recipe

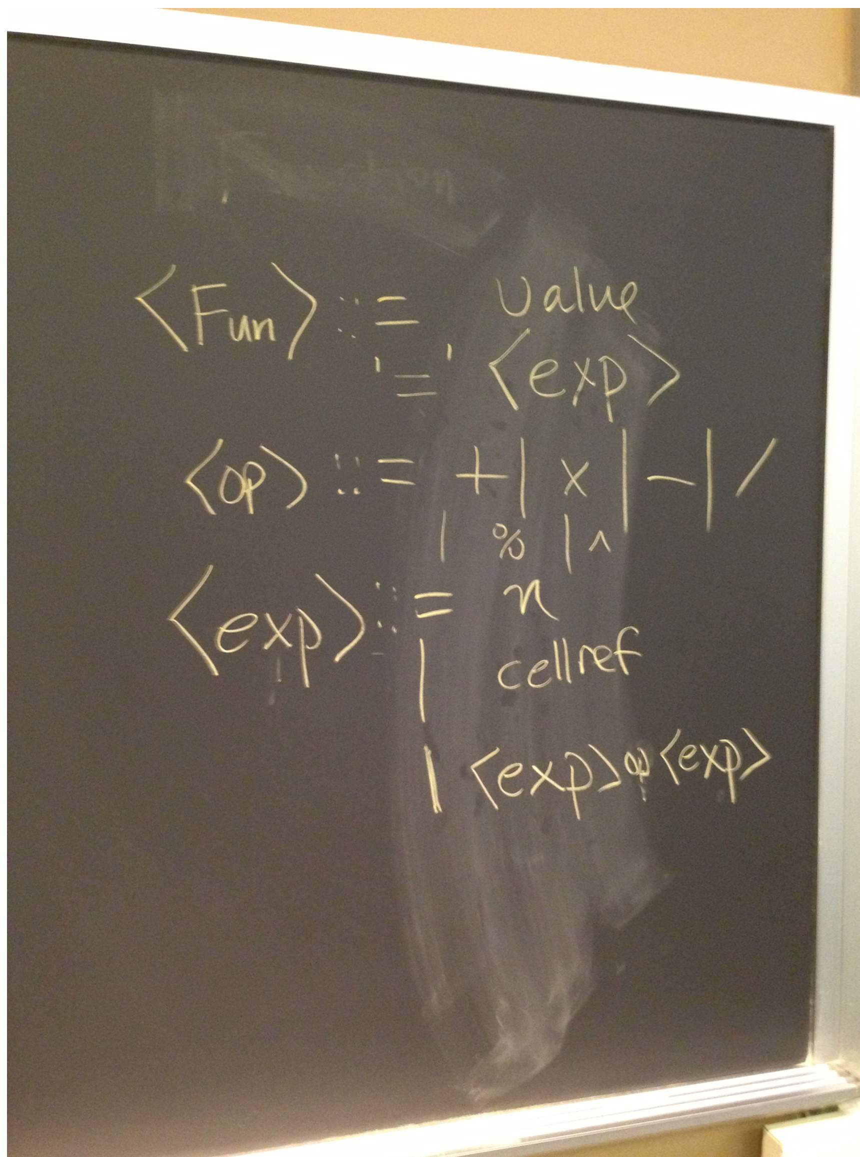
1. Understand the problem
   What are the relevant concepts and how do they relate?
2. Formalize the interface
   How should the program interact with its environment?
3. Write test cases
   How does the program behave on typical inputs? On unusual ones? On erroneous ones?
4. Implement the required behavior
   Often by decomposing the problem into simpler ones and applying the same recipe to each

In Class Design: Concepts



- Class Cell
  - stores a value and a function
- SpreadSheet
  - stores a 2D array of cells
  - displays them using a GridLayout
  - user input of functions in the function Bar
- Interface Value with implementations IntValue, StringValue, ErrorValue

Initial specification of functions



A function is either

- a value
- an '=' character followed by an expression

An expression is either

- a value
- a cell reference
- two expressions separated by an operator

An operator is one of '+', '-', '*', '/', '%', '^'