# Programming Languages and Techniques (CIS120)

Lecture 36

December 3rd , 2015

## Hashing, HashSets

# Game project grading

- Final Program Due: (88 points)
  Tuesday December 8th at 11:59pm
  - Submit zipfile online, submission *only* checks if your code compiles
- Grade based on demo with your TA during reading days
  - Make sure that you test your program in Moore 100, especially if you use outside libraries
  - Grading rubric on the assignment website
  - Recommendation: don't be too ambitious.
- *NO LATE SUBMISSIONS PERMITTED*

How is the Game Project going so far?

1. not started
2. got an idea
3. submitted design proposal
4. started coding
5. it's somewhat working
6. it's mostly working
7. debugging / polishing
8. done!

# Hash Sets & Hash Maps

array-based implementation of sets and maps

# Hash Sets and Maps: The Big Idea

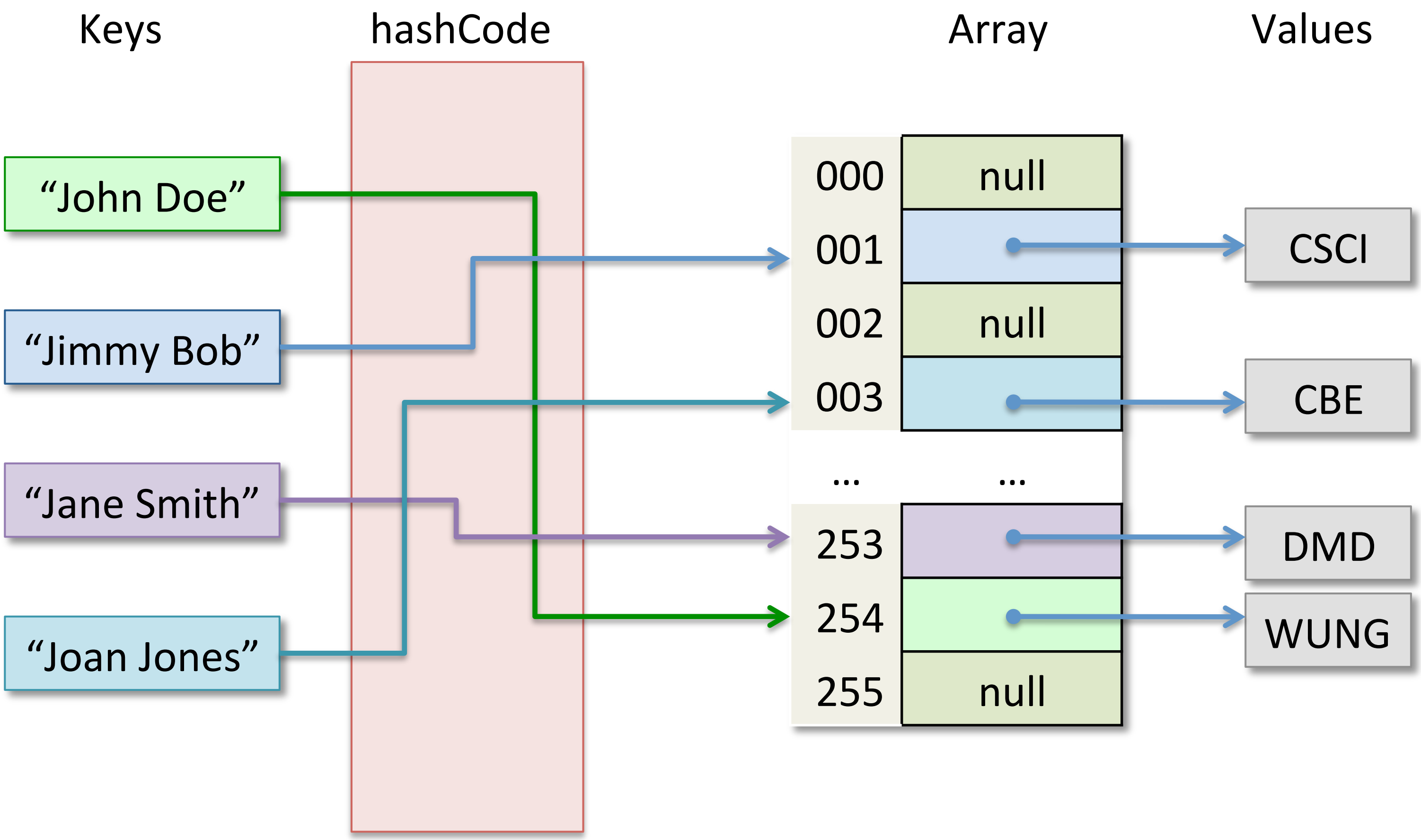
Combine:

- 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

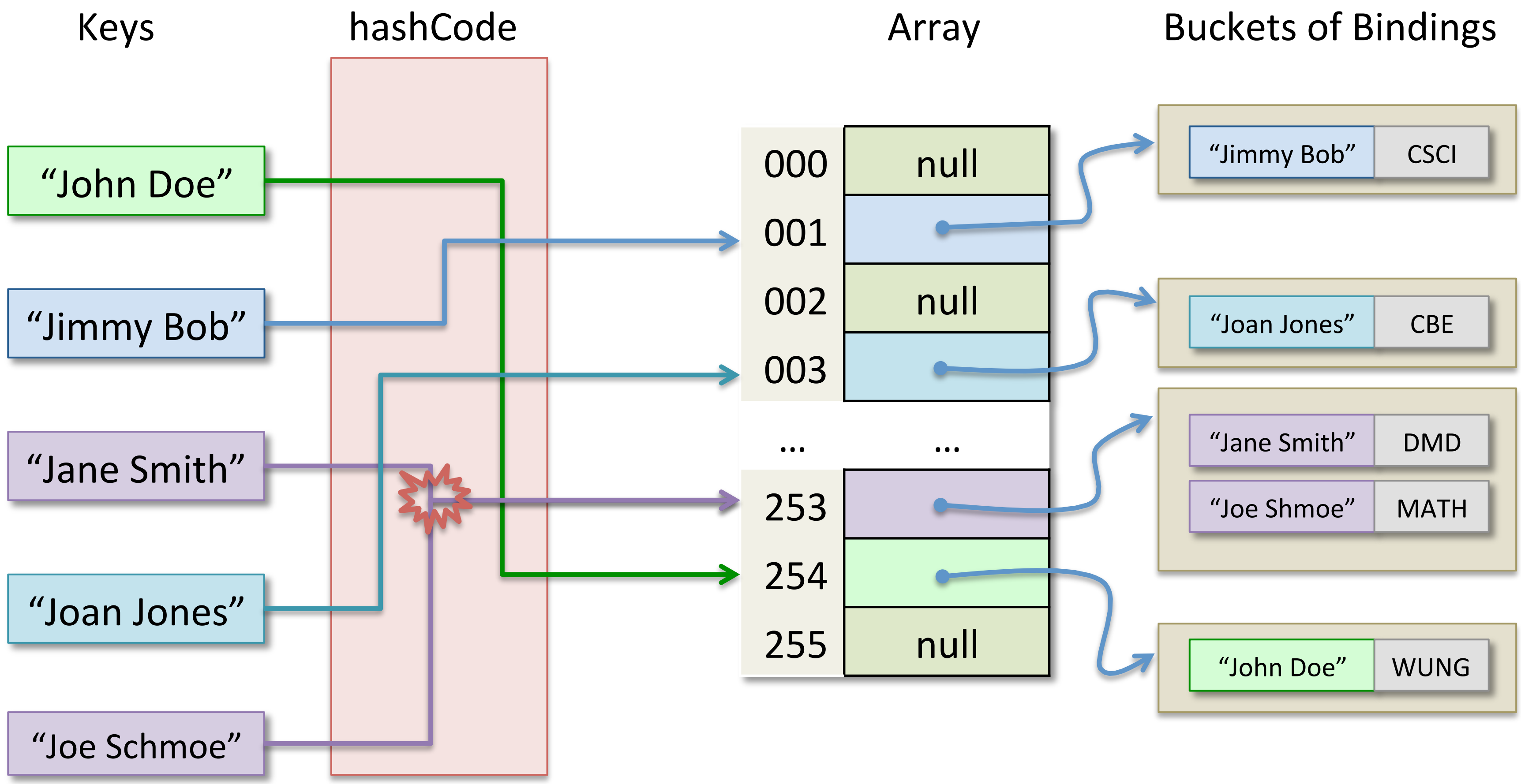
- Uh Oh: Indices derived via hashing may not be unique!

```
“Jane Smith”.hashCode() % 256 ➔ 253
“Joe Schmoe”.hashCode() % 256 ➔ 253
```

- Good hashCode functions make it *unlikely* that two keys will produce the same hash

- But, it can happen that two keys do produce the same index – that is, their hashes *collide*

# Bucketing and Collisions



Keys

“John Doe”

“Jimmy Bob”

“Jane Smith”

“Joan Jones”

“Joe Schmoe”

hashCode

Array

| | |
|---|---|
| 000 | null |
| 001 | |
| 002 | null |
| 003 | |
| … | … |
| 253 | |
| 254 | |
| 255 | null |

Buckets of Bindings

| | |
|---|---|
| “Jimmy Bob” | CSCI |

| | |
|---|---|
| “Joan Jones” | CBE |

| | |
|---|---|
| “Jane Smith” | DMD |
| “Joe Shmoe” | MATH |

| | |
|---|---|
| “John Doe” | WUNG |

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.

# Bucketing and Collisions

- Using an array of *buckets*
  - Each bucket stores the mappings for keys that have the same hash.
  - Each bucket is itself a map from keys to values (implemented by a linked list or binary search tree).
  - The buckets can't use hashing to index the values – instead they use key equality (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

- Not the only solution to the collision problem

# Hashing and User-defined Classes

```java
public class Point {
   private final int x;
   private final int y;
   public Point(int x, int y) { this.x = x; this.y = y; }
   public int getX() { return x; }
   public int getY() { return y; }
}

// somewhere in main…
Map<Point,String> m = new HashMap<Point,String>();
m.put(new Point(1,2), "House");
System.out.println(m.containsKey(new Point(1,2)));
```

What gets printed to the console?

1. true
2. false
3. I have no idea

# HashCode Requirements

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()`

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

- Note: the converse does not have to hold:
  - `o1.hashcode() == o2.hashCode()` does *not* necessarily mean that o1.equals(o2)

# Example for Point

```
public class Point {
   @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + x;
        result = prime * result + y;
        return result;
    }
}
```

- Examples:
  - (new Point(1,2)).hashCode()   yields  994
  - (new Point(2,1)).hashCode()   yields 1024
- Note that *equal* points have the same hashCode
- Why 31?  Prime chosen to create more uniform distribution
- Note: eclipse can generate this code

# Computing Hashes

- 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 = prime * result + c;`
4. return `result`

# 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)

- Performance depends on workload

# Terminological Clash

- The word "hash" is also used in cryptography
- SHA-1, SHA-2, SHA-3, MD5, etc.

- Cryptographic hashes are intended to reduce large byte sequences to short byte sequences
  - Very hard to invert
  - Should only rarely have collisions
  - Are considerably more expensive to compute than hashCode (so not suitable for hash tables)

- Never use hashCode when you need a cryptographic hash!
  - See CIS 331 for more details

# Collections: take away lessons

equals

hashCode

compareTo

# Collections Requirements

- All collections use `equals`
  - Defaults to == (reference equality)
  - Override `equals` to create structural equality
  - Should be: false for distinct instance classes
  - An equivalence relation: reflexive, symmetric, transitive

- HashSets/HashMaps use `hashCode`
  - Override when equals is overridden
  - Should be compatible with equals
  - Should try to "distribute" the values uniformly
  - Iterator not guaranteed to follow element order

- Ordered collections (`TreeSet`, `TreeMap`) need to implement `Comparable<Object>`
  - Override `compareTo`
  - Should implement a *total order*
  - Strongly recommended to be compatible with equals (i.e. o1.equals(o2) exactly when o1.compareTo(o2) == 0)

# Comparing Collection Performance

HashTest.java