Announcements

- Midterm 2 Grades have been released
  - We will go over it shortly
- Program 6 has been released
  - We will go over it shortly
- Last lecture on new material is on 12/9
- The last class will be on 12/11 and will serve strictly as a review session
  - I highly recommend you go...

Midterm 2

Class average was: 80%

No Curve…

Similar to midterm 1, the score on paper is out of 70 but the score on blackboard is out of 150, this is because each midterm is worth 15% of you grade…

I will cover the most missed questions and then open the floor to any other questions concerning midterm 2

Midterm 2 (cont.)

1) Which of the following determines which program behavior to execute depending on data types
   a) Generics
   b) Encapsulation
   c) Polymorphism
   d) Inheritance

Answer is in Weeks 7 to 11 Lecture 10/7

Generics helps us deal with the case where we do not know which data type we are working with, but does NOT define behavior that is contingent on any data type

Midterm 2 (cont.)

Questions 44 - 47

Question 47 was the most missed question… and undoubtedly the hardest question on this exam.

No i am not throwing it out

Midterm 2 (cont.)

Question 48

You should have already seen this one...

Midterm 2 (cont.)

29) Which of the following category would IOException fall under?

a) Checked
b) Unchecked
c) None of the above

41) Which of the following category would IndexOutOfBoundsException fall under?

d) Checked
e) Unchecked
f) None of the above

Midterm 2 (cont.)

Questions?

Program 6

Due Wednesday 12/11 at 11:59 PM

There is extra credit, but this time the criteria is less specific…

There will be NO late turn in on this one, so start soon.

In this program you will be coming up with your own program. It can do whatever you want, as long as it meets my specifications

WARNING: Due to the nature of this assignment it will be very easy to tell if you copy someone else's program...

Final Quiz Time ;)

Hash Table

A ***hash table*** is a data structure that stores unordered items by mapping (or hashing) each item to a location in an array.

In a hash table, an item's ***key*** is the value used to map to an index.

For all items that might possibly be stored in the hash table, every key is ideally unique, so that the hash table's algorithms can search for a specific item by that key.

Hash Table (cont.)

The **key** does not necessarily need to be a number, it can also (and very commonly) be a string.

Elements are inserted into a hash table as part of a pairing called a: **Key / Value pair**.

Let’s look at an example on the board...

Hash Table (cont.)

Each hash table array element is called a ***bucket***.

A ***hash function*** computes a bucket index from the item's key.

A common easy hash function uses the ***modulo operator %***, which computes the integer remainder when dividing two numbers.

Ex: For a 20 element hash table, a hash function of key % 20 will map keys to bucket indices 0 to 19. This is because the remainder will never exceed 20.

Let’s look at an example...

Hash Table (cont.)

The issue with using modulo as a hashing function is that it does not really produce unique values.

For example:

5 % 20 ---> This results in 5

25 % 20 ---> This also results in 5

When two keys map to the same location, we call this a **collision**.

Hash Table (cont.)

Various techniques are used to handle collisions during insertions, such as chaining or open addressing.

Chaining is a collision resolution technique where each bucket has a list of items (so bucket 5's list would become 55, 75). In other words an array of lists.

Open addressing (also called probing) is a collision resolution technique where collisions are resolved by looking for an empty bucket elsewhere in the table (so 75 might be stored in bucket 6).

Hash Table (cont.)

Chaining

Chaining handles hash table collisions by using a list for each bucket, where each list may store multiple items that map to the same bucket.

The insert operation first uses the item's key to determine the bucket, and then inserts the item in that bucket's list.

Searching also first determines the bucket, and then searches the bucket's list.

Let’s look at an example...

Hash Table (cont.)

Linear Probing

A hash table with ***linear probing*** handles a collision by starting at the key's mapped bucket, and then linearly searches subsequent buckets until an empty bucket is found.

Let’s look at an example...

Hash Table (cont.)

Linear probing distinguishes two types of empty buckets.

An ***empty-since-start*** bucket has been empty since the hash table was created.

An ***empty-after-removal*** bucket had an item removed that caused the bucket to now be empty.

The distinction will be important during searches, since searching only stops for empty-since-start, not for empty-after-removal.