Collections 2

Sets and Hashing
Overview of Sets

- Recall from Collections 1 that a set is an unordered collection with no duplicates.

- Which of the following is sure to be a set?
  - Students in this class: Yes
  - Coins in my pocket: No, can have 2 of the same coin
  - Letters in the alphabet: Yes

- The Collections API provides a Set Interface for you.
The Set Interface

- The set interface provides the five basic operations to manipulate sets.
  - Add an element
  - Remove an element
  - Find the size of the set
  - List all the elements in the set
  - Check if an element is in the set
The Set Interface

- **add(x)**
  - Adds element x to the set. If x was already present, nothing changes.

- **remove(x)**
  - Removes x from the set if x was present, else leaves the set unchanged.

- **size()**
  - Returns an int indicating the current size of the set.
The Set Interface

- **iterator()**
  - Returns an Iterator object over the elements in the set.
  - This provides a way to ‘look’ at all the elements in the set in turn. *(Note: Remember that the set has no specified order.)*

- **contains(x)**
  - Returns a boolean representing whether or not the set contains x.
Hashing

- Hashing is a technique that can save on search time and storage space.
- Java Objects such as HashSet and HashMap are based on this technique.
- This section will teach you how these classes work and how to use them effectively.
Hashing Example

- Imagine each student at this school has a sheet with their info on it.
- Someone wants to know whether or not a person is a student here.
- If there is no order to the sheets we would have to look through all of them.
- Can we improve upon this to reduce the search time?
Hashing Example

- Let's say we have two bins, one for sheets of male students and one for female.

- If we are told the sex of the person we need only check in one bin.

- We could do the same for any characteristic such as age, weight, eye color.

- Some of these will be more useful than others depending on how many bins there are and how many sheets in each bin.
An important property is that each bin has approx the same number of student sheets.

Suppose we want 100 bins. Which of the following do you think has the above property?

- Bin for each age from 1 – 100 years
- Bin for each height 100 – 199 cm
- Bin for each last two digits of phone no.
A **hash code** is an integer associated with an object that can be derived from the object.

Every object in Java has a `hashCode()` method that returns an int.

Classes such as `HashSet` and `HashMap` use the number to store objects in bins just like our student example.

If we call `contains(x)` on a `HashSet`, instead of looking through the whole set for x, Java will only look in the bin corresponding to `x.hashCode()`.
Hash Code Problem

- As described, hashing divides elements into many bins for quick retrieval and lookup.

- What could happen if two equal elements do not have the same hash code?

- You could put an element in a bin and not be able to find it again because you look in the wrong bin!
hashCode() and equals()

- If two elements are equal we need them to map to the same bin!

- **This must be true of any hash code:**
  \[ x\.equals(y) \implies x\.hashCode() = y\.hashCode() \]

- When you override `hashCode` you *must* override `equals`
How to write `hashCode()`

- Return some computation on the hash codes of each of the fields:

```java
public class Student {
    String name;
    int year;

    public int hashCode() {
        return 7 * name.hashCode() + year + 33;
    }
}
```