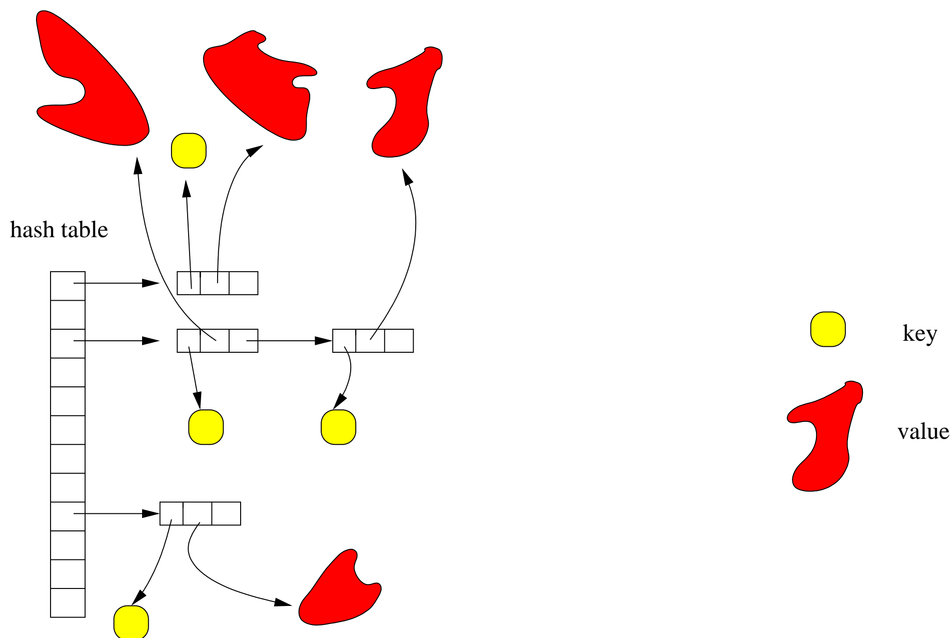
I began this lecture by reviewing some basic terms about maps and hashing including:

- *key*, *value*, and *hash function*, *hash code*, *hash value*, *collision*, *separate chaining*.
- the *load factor* of a hash table is the ratio of the number of (key,value) pairs in the table to the number of slots in the table (m).



hashing in Java

I then spent the rest of the lecture discussing some of the basic ideas of hash maps work in Java. Many details were mentioned. The important highlights are included here.

`HashMap` class

In lecture 28, I mentioned the `Map` interface. To make a hash table in java, you can use the `HashMap<K,V>` class which implements the `Map` interface.

The default constructor for `HashMap` constructs an empty hashtable array of size $m = 16$. The default (maximum) load factor is defined to be 0.75. Once an element is added so that the hash table's load factor increases beyond 0.75, the size m of the hashtable is doubled and all the (key,value) pairs in the table are re-hashed to this new hash table. Note: a maximum of 12 entries can be added to the original table before rehashing occurs (since $12/16 = 0.75$). When an attempt is made to add a 13th entry, a new hash table is created and the 13 entries are hashed to the new table. Notice that having a bigger hash table requires having a different hashing function. In particular, if a *modm* hash function can be used, and in this case the hash function just takes the low order $\lg m$ bits of the hash code. This hash function is not specified in the API, but it is an easy

implementation. (Like all standard Java methods, the implementation details are hidden from the user and are not part of the API.)

Some of the methods in class `HashMap<K,V>` are:

```
public  V             put(K key, V value)    // returns previous value (or null)
public  V             get(Object key)
public  V             remove(Object key)
```

I encourage you to check out the API and examine the other methods and try to understand what they do.

What do these methods do? The `put` method adds a $(key, value)$ pair to the hash table, whereas the `get` method takes a *key* and returns the *value* associated with that key (or null if there is no value associated with that key).

Here is a sketch of how the implementation could work. We would need a private method `hash()` which is a hash function. It has one argument (a key). We would also have an array `hashTable` whose elements are linked lists of (key,value) pairs.

For `put`, you could attempt to add a new (key,value) pair as follows. If there is already an entry with that key, then we would replace the previous value with the new one, and return the previous value.

```
put(key, value){     //  does nothing if (key,value) is already there
  index = hash(key)   // and replaces value if (key,othervalue) is there
  search entries in hashTable[index] for this key
  if (key is found)
    replace old value with new value   //   cannot have (k,v1) and k(v2)
    return old value
  else
    insert (key,value) into list
 }
```

The methods `remove` and `getValue` are similar.

```
remove(key){
  index = hash(key)
  if (hashTable[index] != null)
     search entries in list hashTable[index} for key
     if key is found
         remove entry (key,value)
  }

getValue(key){
  index = hash(key)
  search entries in list hashTable[index] for key
  if key is found
    return value
  else
    return null
}
```

`hashCode()` method

Recall that all classes extend the class `Object`. The class `Object` has a method `hashCode()` which returns a 32 bit number that is the location of the object in memory. Notice that this number has nothing to do with what class the object belongs to and what values are stored in the fields of the object. It merely assures that each object has a hash code, so that if you want to use this object as a key in a hash table, you can do so.

Recall also that the Object class has an `equals()` method, which checks if one object equals another. In particular, if `x` and `y` reference the same `Object` if and only if `x.equals(y)` returns true. Note that if you use both the `Object` class'es `hashcode()` and `equals()` method, then `x.equals(y)` returns true if and only if `x.hashCode() == y.hashCode()`. The reason is that both methods compare the addresses of the two objects!

Typically, when you write your own class, you want to override the `equals` method. You typically want `x.equals(y)` to return true if and only if the objects are of the same class and the fields of the objects are the same. It can easily happen that you wish to have two objects considered "equal", even though their addresses in memory are not the same. This is the case with `String` objects, for example, which is why the `String` object overrides the `equals` method.

If you going to override `equals` and allow `k1.equals(k2)` to return true even though **`k1`** and `k2` may refer to distinct object keys, then you also need to override the `hashCode()` method. Otherwise, you could could have two key objects which would have different addresses (and hence different hashcodes) and which might hash to different slots in the hashtable. This would be disasterous if used `k1` to `put` the entry and `k2` to `get` the entry.

The general rule to follow is this: if you are going to override either the `hashCode()` or `equals()` method, then you should:

- override *both* of them
- ensure that: if `x.equals(y)` returns true, then `x.hashCode() == y.hashCode()`

The converse of the latter is not required. It could happen that `x.hashCode() == y.hashCode()` but `x.equals(y)` is false. This just says that we are allowing collisions to occur.