Creating a Class Beyond the Basics (pt 2)

Based on slides by Prof. Burton Ma

Arrays as Containers

- Suppose you have an array of unique **PhoneNumber**s
  - How do you compute whether or not the array

```
public static boolean
       hasPhoneNumber(PhoneNumber p,
                      PhoneNumber[] numbers)
{
  if (numbers != null) {
    for( PhoneNumber num : numbers ) {
      if (num.equals(p)) {
        return true;
      }
    }
  }
  return false;
}
```

- Called *linear search* or *sequential search*
  - Doubling the length of the array doubles the amount of searching we need to do
- If there are **`n PhoneNumber`**s in the array:
  - Best case: the first **`PhoneNumber`** is the one we are searching for → 1 call to **`equals()`**
  - Worst case: the **`PhoneNumber`** is not in the array → n calls to **`equals()`**
  - Average case: the **`PhoneNumber`** is somewhere in the middle of the array → approximately (n/2) calls to **`equals()`**

hashCode()

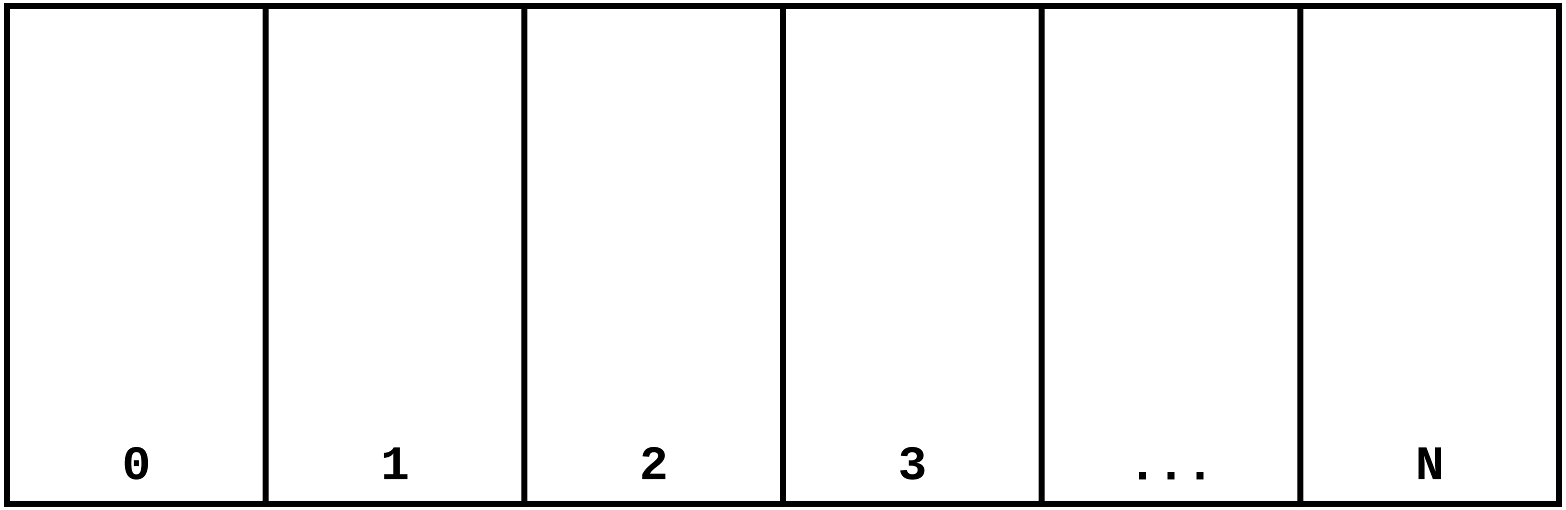
- If you override **equals()** you must override **hashCode()**
  - Otherwise, the hashed containers won't work

```
// client code somewhere
PhoneNumber pizza = new PhoneNumber(416, 967, 1111);

HashSet<PhoneNumber> h = new HashSet<PhoneNumber>();
h.add(pizza);
System.out.println( h.contains(pizza) );       // true

PhoneNumber pizzapizza =
                    new PhoneNumber(416, 967, 1111);
System.out.println( h.contains(pizzapizza) );  // false
```

[notes 2.3.5]

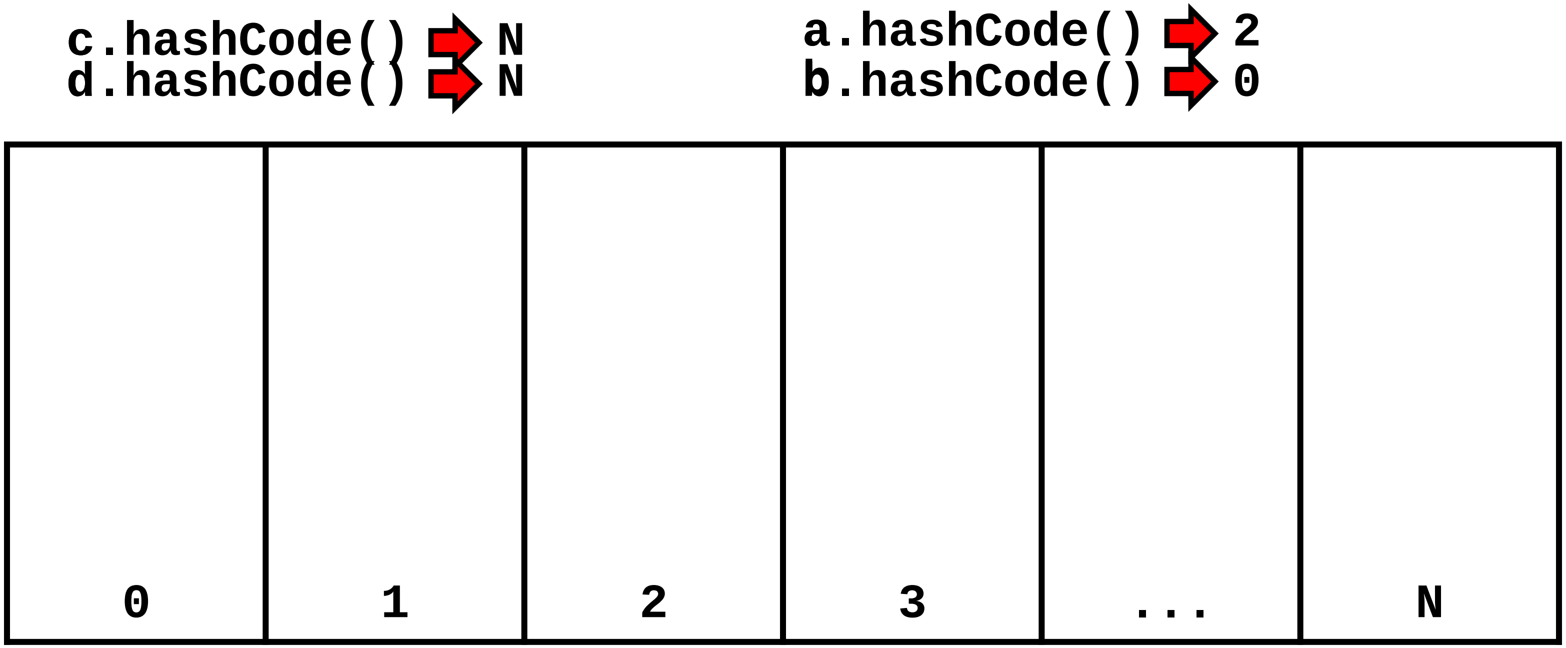
Hash Tables

- You can think of a hash table as being an array of buckets where each bucket holds the stored objects



Insertion into a Hash Table

- To insert an object **`a`**, the hash table calls **`a.hashCode()`** method to compute which bucket to put the object into



`c.hashCode()` ➡ `N`
`d.hashCode()` ➡ `N`

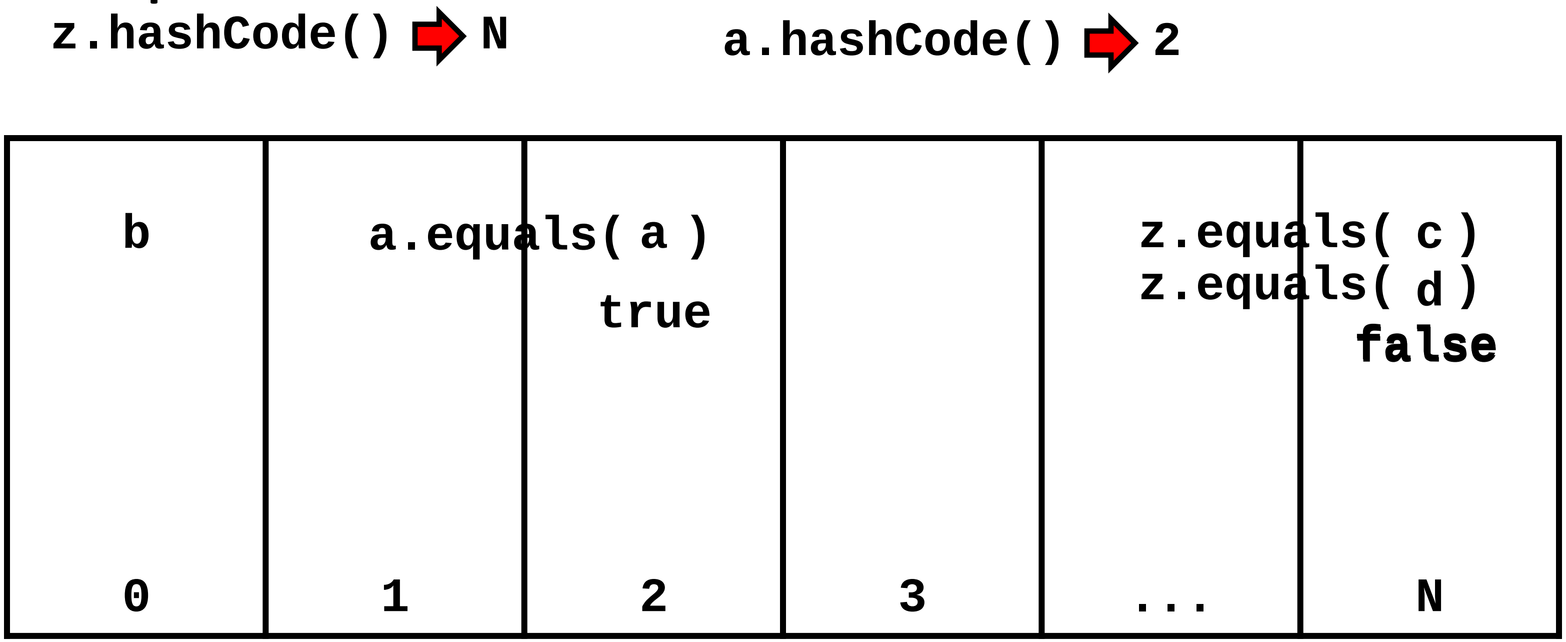
`a.hashCode()` ➡ `2`
`b.hashCode()` ➡ `0`

0	1	2	3	...	N

➡ means the hash table takes the hash code and does something to it to make it fit in the range **`0—N`**

Search on a Hash Table

- To see if a hash table contains an object **`a`**, the hash table calls **`a.hashCode()`** method to compute which bucket to look for **`a`** in



`z.hashCode()` → `N`

`a.hashCode()` → `2`

0	1	2	3	...	N
b					

`a.equals( a )`

`true`

`z.equals( c )`
`z.equals( d )`

`false`

- Searching a hash table is usually much faster than linear search
  - Doubling the number of elements in the hash table usually does not noticably increase the amount of search needed
- If there are **`n PhoneNumber`**s in the hash table:
  - Best case: the bucket is empty, or the first **`PhoneNumber`** in the bucket is the one we are searching for → 0 or 1 call to **`equals()`**
  - Worst case: all **`n`** of the **`PhoneNumber`**s are in the same bucket → N calls to **`equals()`**
  - Average case: the **`PhoneNumber`** is in a bucket with a small number of other **`PhoneNumbers`** → a small number of calls to **`equals()`**

`Object hashCode()`

- If you don't override **`hashCode()`**, you get the implementation from **`Object.hashCode()`**
  - **`Object.hashCode()`** uses the memory address of the object to compute the hash code

```
// client code somewhere
PhoneNumber pizza = new PhoneNumber(416, 967, 1111);

HashSet<PhoneNumber> h = new HashSet<PhoneNumber>();
h.add(pizza);

PhoneNumber pizzapizza = new PhoneNumber(416, 967, 1111);
System.out.println( h.contains(pizzapizza) );  // false
```

- Note that **`pizza`** and **`pizzapizza`** are distinct objects
  - Therefore, their memory locations must be different
    - Therefore, their hash codes are different (probably)
    - Therefore, the hash table looks in the wrong bucket (probably) and does not find the phone number even though **`pizzapizza.equals(pizza)`**

A Bad (but legal) **hashCode()**

```
public final class PhoneNumber {
  // attributes, constructors, methods ...

  @Override public int hashCode()
  {
    return 1;  // or any other constant int
  }
}
```

- This will cause a hashed container to put all **PhoneNumbers** in the same bucket

A Slightly Better **hashCode()**

```
public final class PhoneNumber {
  // attributes, constructors, methods ...

  @Override public int hashCode()
  {
    return (int)(this.getAreaCode() +
                 this.getExchangeCode() +
                 this.getStationCode());
  }
}
```

- The basic idea is generate a hash code using the attributes of the object
- It would be nice if two distinct objects had two distinct hash codes
  - But this is not required; two different objects can have the same hash code
- It is required that:
  1. If **`x.equals(y)`** then **`x.hashCode() == y.hashCode()`**
  2. **`x.hashCode()`** always returns the same value if **`x`** does not change its state

Something to Think About

- What do you need to be careful of when putting a mutable object into a **`HashSet`**?