Not overriding `equals`

- what happens if you do not override `equals` for a value type class?
  - all of the Java collections will fail in confusing ways

Not overriding `equals`

```
Complex y = new Complex(1, -2);
Complex z = new Complex(1, -2);

List<Complex> list = new ArrayList<Complex>();
list.add(y);
System.out.println("contains (1 - 2i)? " + list.contains(z));
```

Output:

```
contains (1 - 2i)? false
```

`contains` uses `equals` to search the elements of the list

Not overriding `equals`

```
Complex y = new Complex(1, -2);
Complex z = new Complex(1, -2);

Set<Complex> set = new HashSet<Complex>();
set.add(y);
System.out.println("add (1 - 2i)? " + set.add(z));
```

Output:

```
add (1 - 2i)? true
```

`add` uses `equals` to search the elements of the set

Not overriding `equals`

```
Complex y = new Complex(1, -2);
Complex z = new Complex(1, -2);

Map<Complex, String> map = new TreeMap<Complex, String>();
map.put(y, y.toString());
System.out.println("contains (1 - 2i)? " + map.put(z, z.toString()));
```

Output:

```
contains (1 - 2i)? null
```

`put` uses **`equals`** to search the elements of the map

hashCode

hashCode

- if you override **`equals`** you must override **`hashCode`**
  - otherwise, the hashed containers won't work properly
    - recall that we did not override **`hashCode`** for **`Complex`**

```
// client code somewhere
Complex y = new Complex(1, -2);

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

Complex z = new Complex(1, -2);
System.out.println( h.contains(z) );       // false
```


[notes 3.3.5]

Arrays as Containers

- suppose you have an array of unique **`Complex`** numbers
  - how do you compute whether or not the array contains a particular **`Complex`** number?
    - write a loop to examine every element of the array

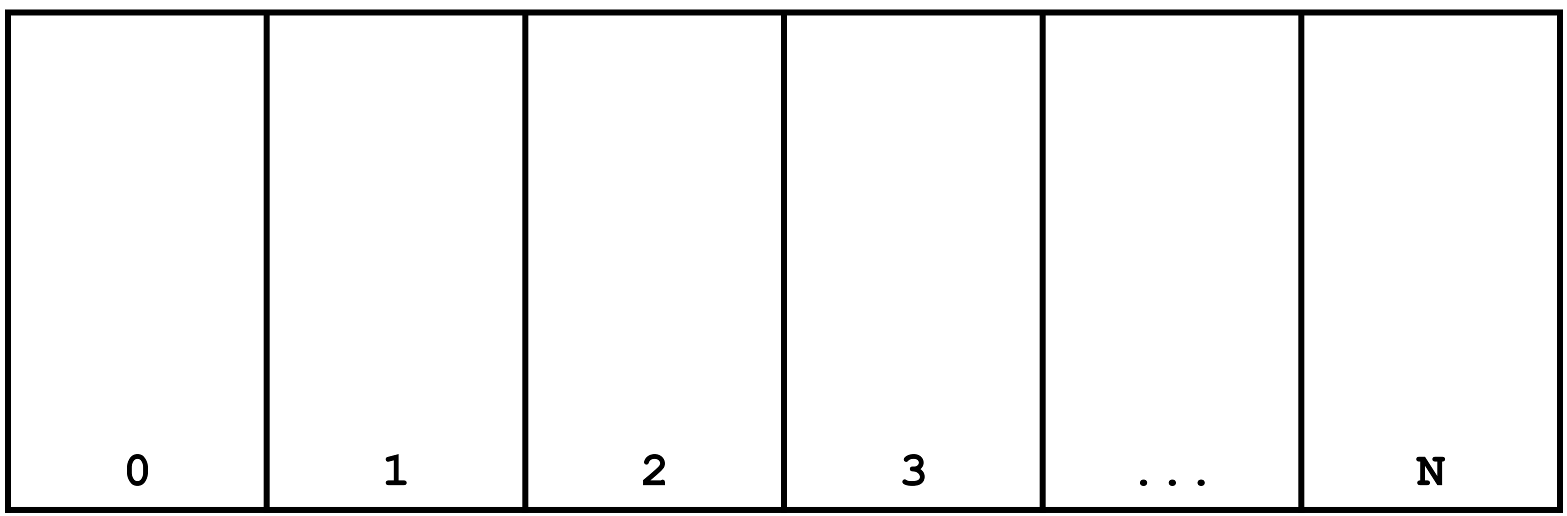
```
public static boolean
       hasNumber(Complex z, Complex[] numbers) {

  for( Complex num : numbers ) {
    if (num.equals(z)) {
      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 Complex`** numbers in the array:
  - best case
    - the first **`Complex`** number is the one we are searching for
      - 1 call to **`equals()`**
  - worst case
    - the **`Complex`** number is not in the array
      - n calls to **`equals()`**
  - average case
    - the **`Complex`** number is somewhere in the middle of the array
      - approximately (n/2) calls to **`equals()`**

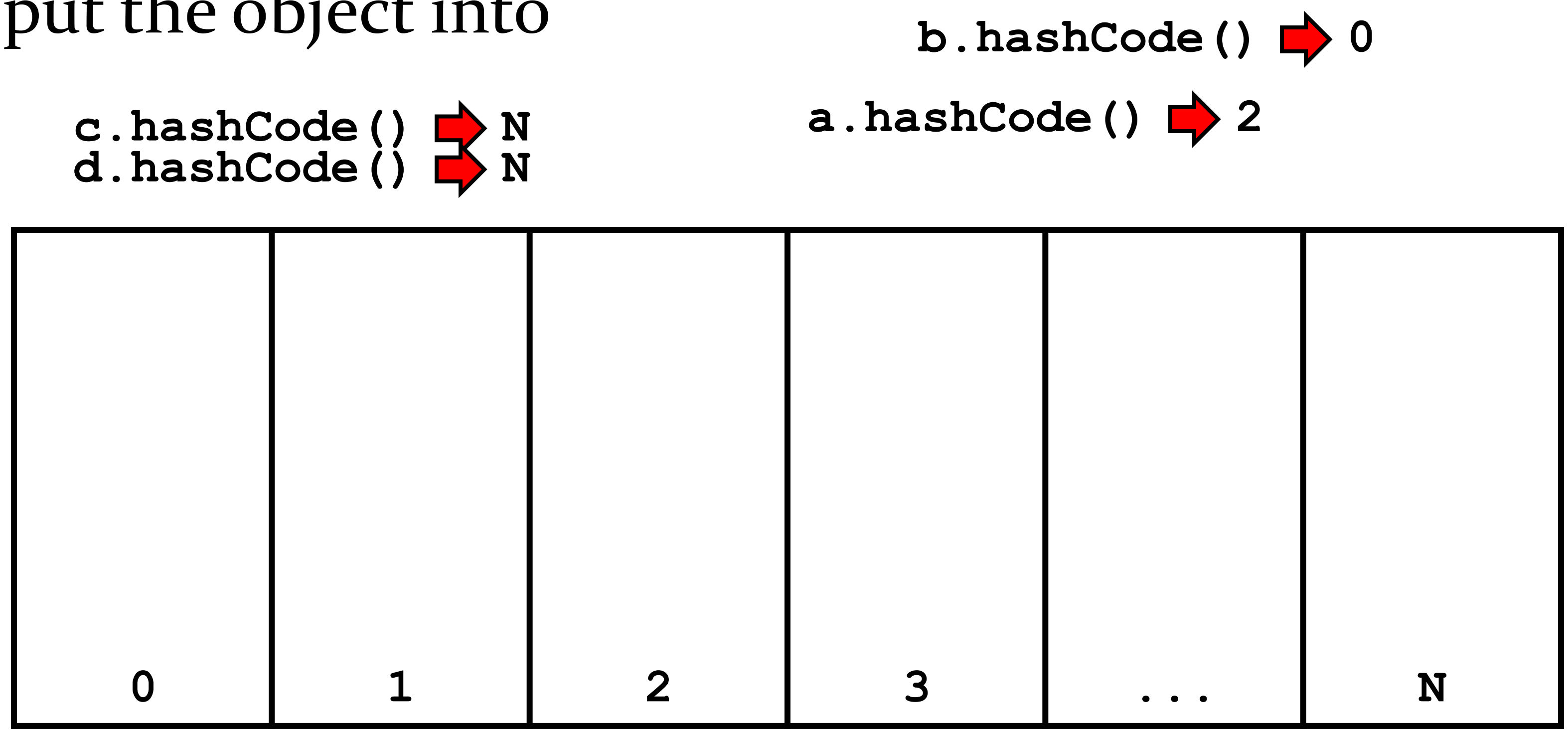
Hash Tables

- 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



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

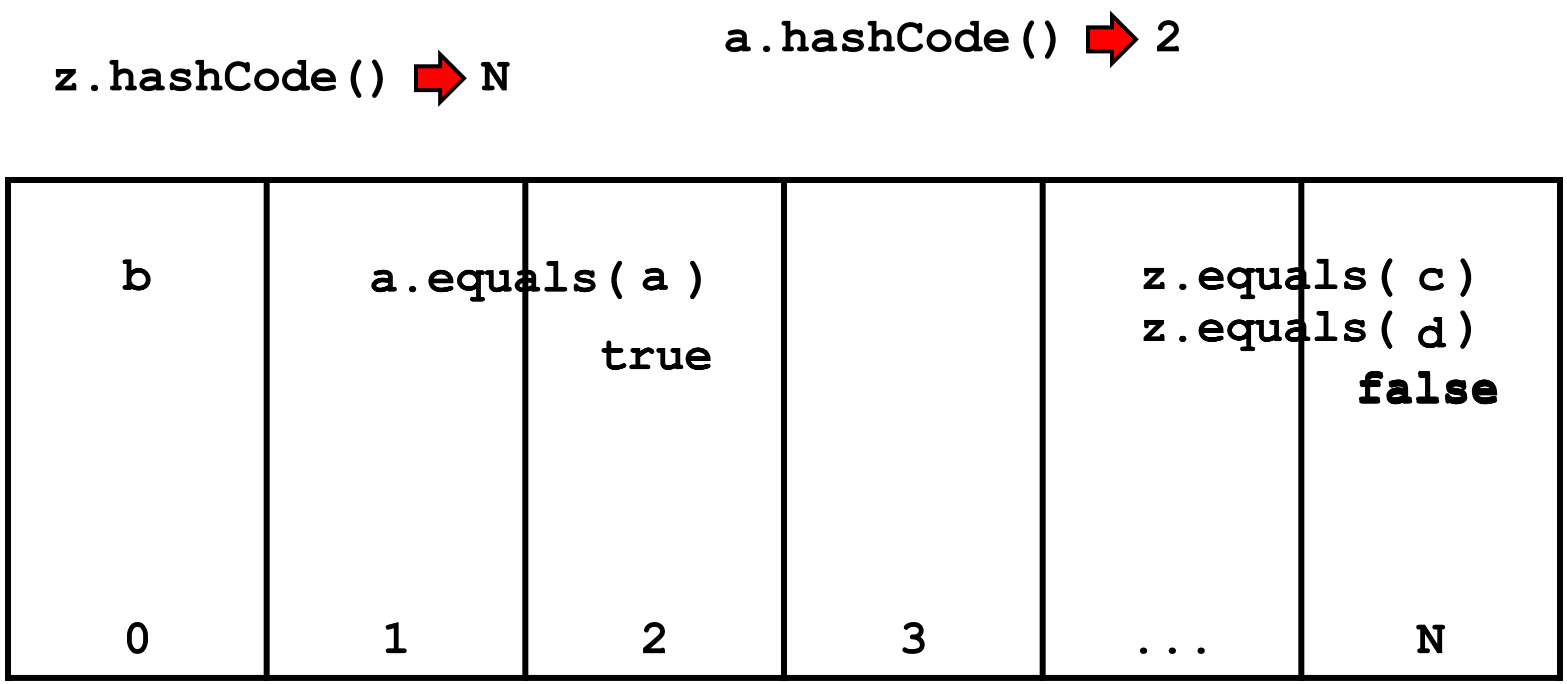
Insertion into a Hash Table

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

b		a			c d
0	1	2	3	...	N

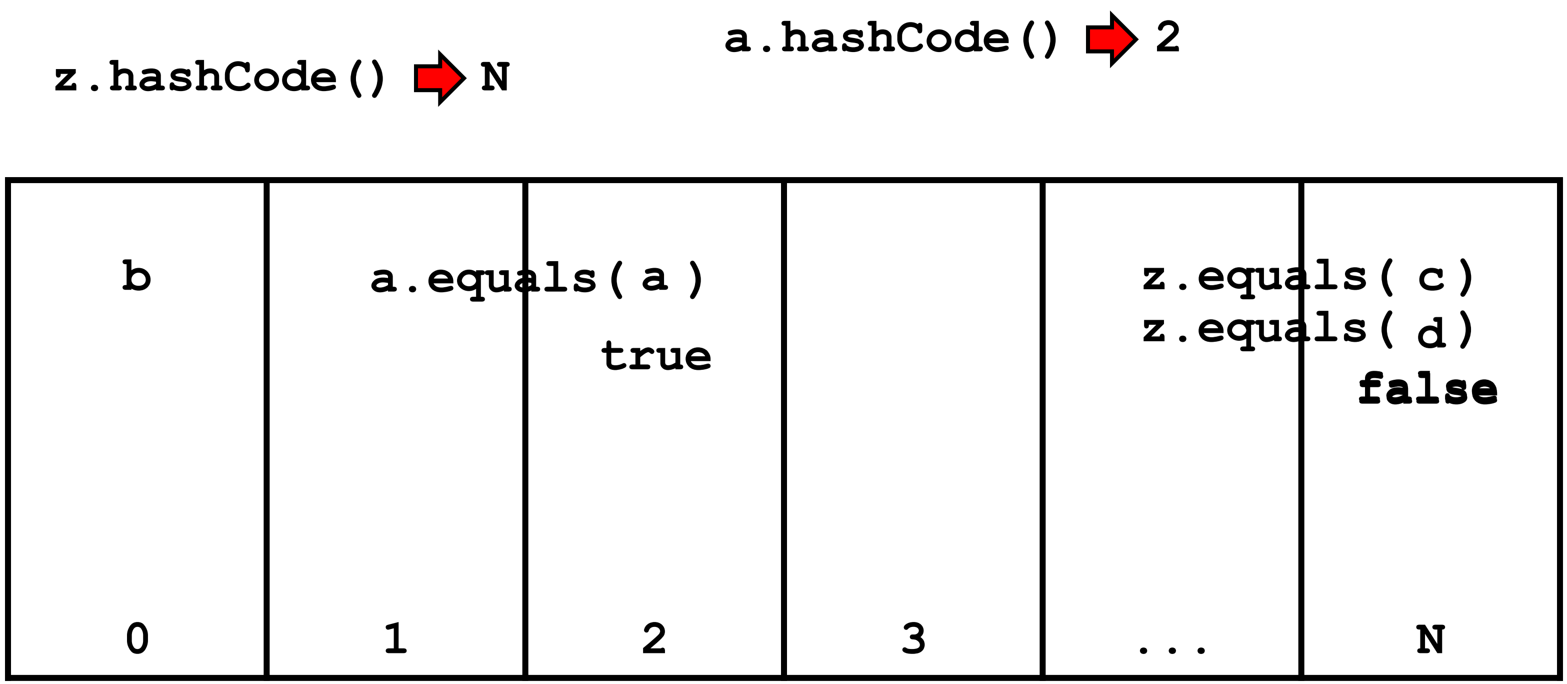
Search on a Hash Table

- 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



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



- 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 Complex`** numbers in the hash table:
  - best case
    - the bucket is empty, or the first **`Complex`** in the bucket is the one we are searching for
      - 0 or 1 call to **`equals()`**
  - worst case
    - all **`n`** of the **`Complex`** numbers are in the same bucket
      - n calls to **`equals()`**
  - average case
    - the **`Complex`** number is in a bucket with a small number of other **`Complex`** numbers
      - 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

```java
// client code somewhere
Complex y = new Complex(1, -2);

HashSet<Complex> h = new HashSet<Complex>();
h.add(y);

Complex z = new Complex(1, -2);
System.out.println( h.contains(z) );        // false
```

- note that **`y`** and **`z`** refer to 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 **`y.equals(z)`**

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

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

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

- this will cause a hashed container to put all **`Complex`** numbers into the same bucket

A Slightly Better `hashCode`

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

  @Override public int hashCode()
  {
    return (int)(this.getReal() + this.getImag());
  }
}
```

eclipse `hashCode`

- eclipse will generate a hashCode method for you
  - Source → Generate hashCode() and equals()…

- it uses an algorithm that
  - "… yields reasonably good hash functions, [but] does not yield state-of-the-art hash functions, nor do the Java platform libraries provide such hash functions as of release 1.6. Writing such hash functions is a research topic, best left to mathematicians and theoretical computer scientists."
    - Joshua Bloch, *Effective Java 2nd Edition*

- the basic idea is generate a hash code using the fields 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`**?
  - can you avoid the problem by using immutable objects?