

# hashCode and compareTo

# 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 `PhoneNumber`

```
// 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
```

# Arrays as Containers

---

- ▶ suppose you have an array of unique **PhoneNumbers**
  - ▶ how do you compute whether or not the array contains a particular **PhoneNumber**?

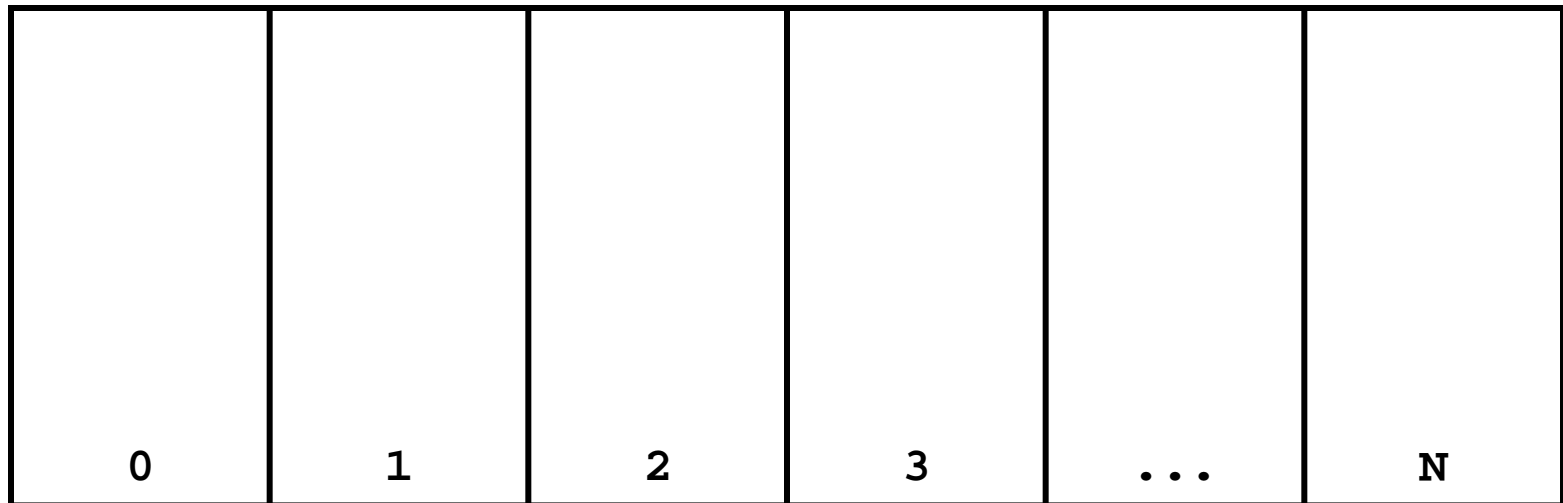
```
public static boolean
    hasPhoneNumber(PhoneNumber p,
                  PhoneNumber[] numbers)
{
    if (numbers != null) {
        for( PhoneNumber num : numbers ) {
            if (num.equals(p)) {
                return true;
            }
        }
    }
    return false;
}
```

- ▶ called *linear search* or *sequential search*
  - ▶ doubling the length of the array doubles the amount of searching we need to do
- ▶ if there are  $n$  **PhoneNumbers** 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()**

# Hash Tables

---

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



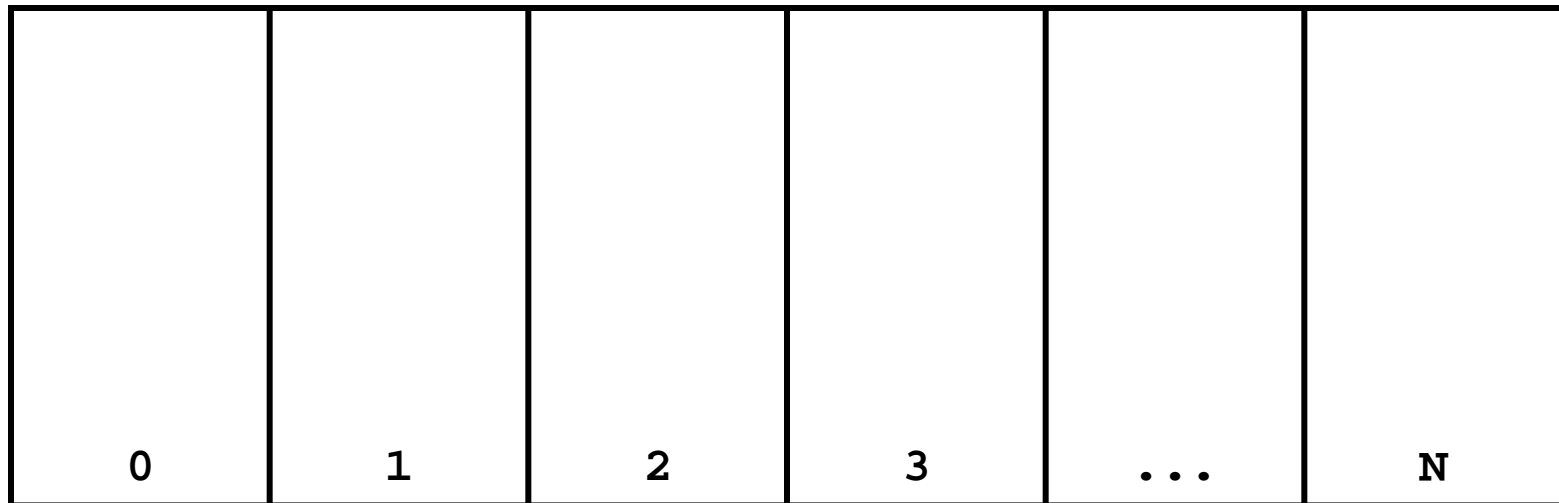
# Insertion into a Hash Table

---

- ▶ 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

b.hashCode() → 0  
a.hashCode() → 2

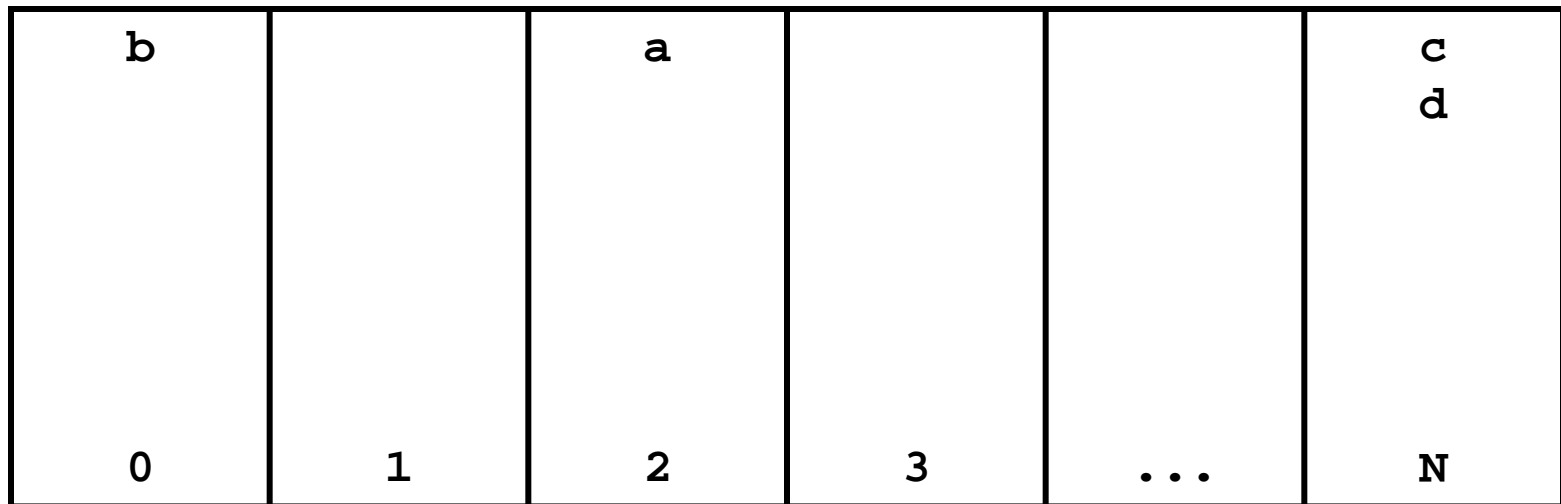


→ means the hash table takes the hash code and does something to it to make it fit in the range 0–N

# 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



# 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

b	<code>a.equals(a)</code>	true		<code>z.equals(c)</code> <code>z.equals(d)</code>	<b>false</b>
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

`z.hashCode()` → N                      `a.hashCode()` → 2

b	<code>a.equals(a)</code>	true		<code>z.equals(c)</code> <code>z.equals(d)</code>	false
0	1	2	3	...	N

- ▶ searching a hash table is usually much faster than linear search
  - ▶ doubling the number of elements in the hash table usually does not noticeably increase the amount of search needed
- ▶ if there are  $n$  **PhoneNumbers** 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 **PhoneNumbers** 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**?
- ▶ can you avoid the problem by using immutable objects?



`compareTo`

# Comparable Objects

---

- ▶ many value types have a natural ordering
  - ▶ that is, for two objects **x** and **y**, **x** is less than **y** is meaningful
    - ▶ **Short**, **Integer**, **Float**, **Double**, etc
    - ▶ **Strings** can be compared in dictionary order
    - ▶ **Dates** can be compared in chronological order
    - ▶ you might compare **Vector2ds** by their length
    - ▶ **Dies** can be compared by their face value
- ▶ if your class has a natural ordering, consider implementing the **Comparable** interface
  - ▶ doing so allows clients to sort arrays or **Collections** of your object

# Interfaces

---

- ▶ an interface is (usually) a group of related methods with empty bodies
  - ▶ the `Comparable` interface has just one method

```
public interface Comparable<T>
{
    int compareTo(T t);
}
```

- ▶ a class that implements an interfaces promises to provide an implementation for every method in the interface

# compareTo ( )

---

- ▶ Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
- ▶ Throws a **ClassCastException** if the specified object type cannot be compared to this object.

# Die compareTo()

---

```
public class Die implements Comparable<Die> {
    // attributes, constructors, methods ...

    public int compareTo(Die other) {
        int result = 0;
        if (this.getValue() < other.getValue()) {
            result = -1;
        }
        else if (this.getValue() > other.getValue()) {
            result = 1;
        }
        return result;
    }
}
```

# Die compareTo ( )

---

- ▶ the following also works for the Die class, but is dangerous in general:

```
public int compareTo(Die other) {  
    int result = this.getValue() - other.getValue();  
    return result;  
}
```

# Comparable Contract

---

1. the sign of the returned `int` must flip if the order of the two compared objects flip
  - ▶ if `x.compareTo(y) > 0` then `y.compareTo(x) < 0`
  - ▶ if `x.compareTo(y) < 0` then `y.compareTo(x) > 0`
  - ▶ if `x.compareTo(y) == 0` then `y.compareTo(x) == 0`

# Comparable Contract

---

2. `compareTo()` must be transitive

- ▶ if `x.compareTo(y) > 0` && `y.compareTo(z) > 0` then  
`x.compareTo(z) > 0`
- ▶ if `x.compareTo(y) < 0` && `y.compareTo(z) < 0` then  
`x.compareTo(z) < 0`
- ▶ if `x.compareTo(y) == 0` && `y.compareTo(z) == 0` then  
`x.compareTo(z) == 0`



# Comparable Contract

---

3. if `x.compareTo(y) == 0` then the signs of `x.compareTo(z)` and `y.compareTo(z)` must be the same

# Consistency with equals

---

- ▶ an implementation of `compareTo()` is said to be consistent with `equals()` when

```
if x.compareTo(y) == 0 then
    x.equals(y) == true
```

- ▶ and

```
if x.equals(y) == true then
    x.compareTo(y) == 0
```

# Not in the Comparable Contract

---

- ▶ it is not required that `compareTo()` be consistent with `equals()`
  - ▶ that is
    - if `x.compareTo(y) == 0` then  
`x.equals(y) == false` is acceptable
  - ▶ similarly
    - if `x.equals(y) == true` then  
`x.compareTo(y) != 0` is acceptable
- ▶ try to come up with examples for both cases above

# Implementing `compareTo`

---

- ▶ implementing `compareTo` is similar to implementing `equals`
- ▶ you need to compare all of the fields
  - ▶ starting with the field that is most significant for ordering purposes and working your way down

# PhoneNumber compareTo()

---

```
public class PhoneNumber implements Comparable<PhoneNumber> {
    // attributes, constructors, methods ...

    public int compareTo(PhoneNumber other) {
        int result = 0;
        result = this.getAreaCode() - other.getAreaCode();
        if (result == 0) {
            result = this.getExchangeCode() - other.getExchangeCode();
        }
        if (result == 0) {
            result = this.getStationCode() - other.getStationCode();
        }
        return result;
    }
}
```

# Implementing `compareTo`

---

- ▶ if you are comparing fields of type `float` or `double` you should use `Float.compareTo` or `Double.compareTo` instead of `<`, `>`, or `==`
- ▶ if your `compareTo` implementation is broken, then any classes or methods that rely on `compareTo` will behave erratically
  - ▶ `TreeSet`, `TreeMap`
  - ▶ many methods in the utility classes `Collections` and `Arrays`

# Mixing Static and Non-Static

# static Fields

---

- ▶ a field that is **static** is a per-class member
  - ▶ only one copy of the field, and the field is associated with the class
    - ▶ every object created from a class declaring a static field shares the same copy of the field
- ▶ static fields are used when you really want only one common instance of the field for the class
  - ▶ less common than non-static fields



# Example

---

- ▶ a textbook example of a static field is a counter that counts the number of created instances of your class

```
// adapted from Sun's Java Tutorial
public class Bicycle {
    // some other fields here...
    private static int numberOfBicycles = 0;

    public Bicycle() {
        // set some attributes here...
        Bicycle.numberOfBicycles++;
    }

    public static int getNumberOfBicyclesCreated() {
        return Bicycle.numberOfBicycles;
    }
}
```

**note:**  
not `this.numberOfBicycles++`

- 
- ▶ another common example is to count the number of times a method has been called

```
public class X {  
  
    private static int numTimesXCalled = 0;  
    private static int numTimesYCalled = 0;  
  
    public void xMethod() {  
        // do something... and then update counter  
        ++X.numTimesXCalled;  
    }  
  
    public void yMethod() {  
        // do something... and then update counter  
        ++X.numTimesYCalled;  
    }  
}
```

# Mixing Static and Non-static Fields

---

- ▶ a class can declare static (per class) and non-static (per instance) fields
- ▶ a common textbook example is giving each instance a unique serial number
  - ▶ the serial number belongs to the instance
    - ▶ therefore it must be a non-static field

```
public class Bicycle {  
    // some attributes here...  
    private static int numberOfBicycles = 0;  
  
    private int serialNumber;  
  
    // ...  
}
```

- 
- ▶ how do you assign each instance a unique serial number?
    - ▶ the instance cannot give itself a unique serial number because it would need to know all the currently used serial numbers
  - ▶ could require that the client provide a serial number using the constructor
    - ▶ instance has no guarantee that the client has provided a valid (unique) serial number

- 
- ▶ the class can provide unique serial numbers using static fields
    - ▶ e.g. using the number of instances created as a serial number

```
public class Bicycle {
    // some attributes here...

    private static int numberOfBicycles = 0;
    private int serialNumber;

    public Bicycle() {
        // set some attributes here...
        this.serialNumber = Bicycle.numberOfBicycles;
        Bicycle.numberOfBicycles++;
    }
}
```

- 
- ▶ a more sophisticated implementation might use an object to generate serial numbers

```
public class Bicycle {  
  
    // some attributes here...  
    private static int numberOfBicycles = 0;  
  
    private static final  
        SerialGenerator serialSource = new SerialGenerator();  
  
    private int serialNumber;  
  
    public Bicycle() {  
        // set some attributes here...  
        this.serialNumber = Bicycle.serialSource.getNext();  
        Bicycle.numberOfBicycles++;  
    }  
}
```

# Static Methods

---

- ▶ recall that a **static** method is a per-class method
  - ▶ client does not need an object to invoke the method
  - ▶ client uses the class name to access the method
- ▶ a **static** method can only use **static** fields of the class
  - ▶ **static** methods have no **this** parameter because a **static** method can be invoked without an object
  - ▶ without a **this** parameter, there is no way to access non-static fields
- ▶ non-static methods can use all of the fields of a class (including **static** ones)

```
public class Bicycle {  
    // some attributes, constructors, methods here...
```

```
    public static int getNumberCreated()  
    {  
        return Bicycle.numberOfBicycles;  
    }
```

static method  
can only use  
static attributes

```
    public int getSerialNumber()  
    {  
        return this.serialNumber;  
    }
```

non-static method  
can use  
non-static attributes

```
    public void setNewSerialNumber()  
    {  
        this.serialNumber = Bicycle.serialSource.getNext();  
    }  
}
```

and static attributes