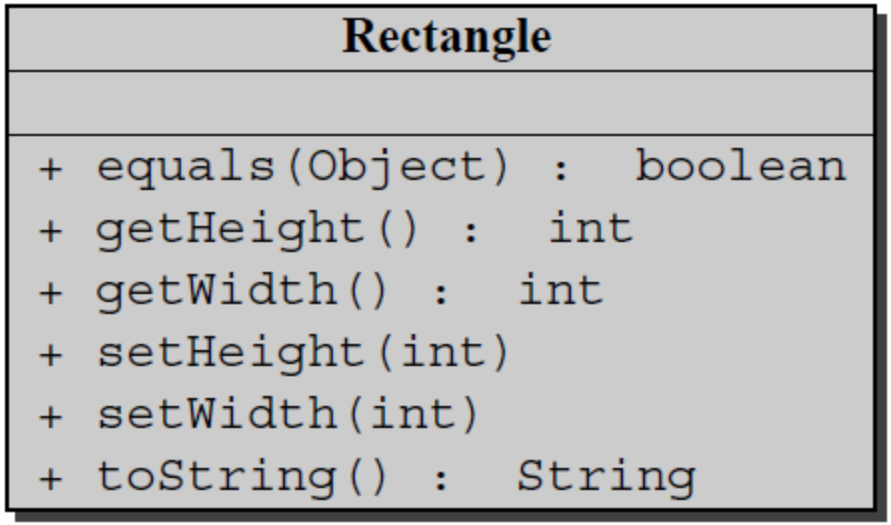
Implementing Non-Static Features

Rectangle Class

- For this section, we will implement a class with only non-static features, that represents a rectangle



Class Declaration

- As in the last lecture, the class declaration starts by specifying the class name

```
public class Rectangle
```

- However, we also want to compare Rectangle objects (e.g., to sort them)

```
public class Rectangle implements
    Comparable<Rectangle>
```

- Additional interfaces can be added in a comma-separated sequence

Attributes

- Remember that attributes are declared as

```
access [static] [final] type name [= value];
```

- We are choosing to declare private variables for the height and width, leaving them uninitialized

```
private int width;
private int height;
```

Attributes (2)

- Static attributes are typically initialized when they are declared, non-static are initialized in the constructor
- An attribute's scope is the entire class
- The fully qualified way to reference an attribute is with the `this` keyword

```
this.width or this.height
```

- However, `this` can be omitted if the result is unambiguous (e.g., attribute shadowing)

Constructors

- Constructors are defined as follows:

access ClassName(*anyParameters*)

- Like methods, they can be overloaded
- When a new object is created…
  - Memory is allocated for the new object
  - A constructor for the corresponding class is called
  - Attributes are initialized

Constructors (2)

```
public Rectangle(int width, int height)
{
    this.width = width;
    this.height = height
}


public Rectangle()
{
    this.width = 0;
    this.height = 0;
}
```

Copy Constructor

- Takes an object of the same class as a parameter

```
public ClassName(ClassName other)
```

- Creates a new object with attributes identical to the passed object
- Note that you do not need (and should not declare) a copy constructor for an immutable type, as the resulting object would be redundant

Constructor Chaining

- When a constructor invokes another constructor it is called *constructor chaining*
- To invoke a constructor in the same class you use the **`this`** keyword
  - If you do this then it must occur on the first line of the constructor body
- Used to reduce code duplication
- Not always feasible
  - E.g., if constructor can throw exceptions

Constructor Chaining (2)

```
public Rectangle()
{
    this(0, 0); // calls Rectangle(int, int)
}


public Rectangle(Rectangle r)
    // copy constructor
{
    this(r.width, r.height);
    // can access private attributes
}
```

Methods

- Methods are defined as follows:

  access returnType signature

- Methods can perform any operation on an object, but typically fall into categories:
  - Accessors – return an attribute value
  - Mutators – modify an attribute value
  - Obligatory – satisfy superclass or interface requirements
  - Class-specific – defined by purpose of class

Accessors

- Allows access to (otherwise private) attribute values
- Naming convention:
  - `getX()` for a non-Boolean attribute, named x
  - `isX()` for a Boolean attribute, named x

```
public int getWidth()
{
    return this.width;
}
// similar for getHeight()
```

Mutators

- Allows modification of (otherwise private) attribute values
- Naming convention:
  - `setX()` for an attribute, named `x`

```
public void setWidth(int width)
{
    this.width = width;
}
// similar for setHeight()
```

Validation using Mutators

- Instead of relying on preconditions, can use mutators to validate argument values
  - Throw and exception
  - Return a Boolean value

- For the Rectangle class, validate that any width argument is non-negative

Validation with Exception

```
/** Sets the width of this rectangle to the given width.
    @param width The new width of this rectangle.
    @throws IllegalArgumentException if width < 0.
*/
public void setWidth(int width)
            throws IllegalArgumentException
{
    if (width < 0)
    {
        throw new IllegalArgumentException(
            "Argument cannot be negative");
    }
    else
    {
        this.width = width;
    }
}
```

Validation with Boolean

```
/** Sets the width of this rectangle to the given width
    if the given width is greater than or equal to 0.
    Returns whether the width has been set.

    @param width The new width of this rectangle.
    @return true if width <= 0, false otherwise.
*/
public boolean setWidth(int width)
{
    boolean isSet = width >=0;
    if (isSet)
    {
        this.width = width;
    }
    return isSet;
}
```

toString Method

- Provides a textual representation of the Rectangle
  - E.g., “Rectangle of width 1 and height 2”
- Default returns class name and memory address

- For the Rectangle class:

```
public String toString()
{
    return "Rectangle of width " + this.width +
        " and height " + this.height;
}
```

`equals` Method

- Evaluate object equality using attribute values (i.e., object state)
- Default compares memory address (like ==)
- Implementing **`equals()`** is surprisingly hard
  - "One would expect that overriding **`equals()`**, since it is a fairly common task, should be a piece of cake. The reality is far from that. There is an amazing amount of disagreement in the Java community regarding correct implementation of **`equals()`**."
    - Angelika Langer, Secrets of equals() – Part 1
  - http://www.angelikalanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals.html
- Our approach is consistent with many texts

Instances of the Same Type can be Equal

- The implementation of **`equals()`** used in the notes and the textbook is based on the rule that an instance can only be equal to another instance of the same type
- At first glance, this sounds reasonable and is easy to implement using **`Object.getClass()`**

`public final Class<? extends Object> getClass()`

- Returns the runtime class of an object.

Instances with Same State are Equal

- Recall that the value of the attributes of an object define the state of the object
  - Two instances are equal if all of their attributes are equal
- Recipe for checking equality of attributes
  1. If the attribute type is a primitive type other than float or double use **==**
  2. If the attribute type is **`float`** use **`Float.compare()`**
  3. If the attribute type is **`double`** use **`Double.compare()`**
  4. If the attribute is an array consider **`Arrays.equals()`**
  5. If the attribute is a reference type use **`equals()`**, but beware of attributes that might be null

The `equals()` Contract Part 1

- For reference values **`equals()`** is
  1. Reflexive :
     - An object is equal to itself
     - **`x.equals(x)`** is **`true`**
  2. Symmetric :
     - Two objects must agree on whether they are equal
     - **`x.equals(y)`** is **`true`** if and only if **`y.equals(x)`** is **`true`**
  3. Transitive :
     - If a first object is equal to a second, and the second object is equal to a third, then the first object must be equal to the third
     - If **`x.equals(y)`** is **`true`**, and **`y.equals(z)`** is **`true`**, then **`x.equals(z)`** must be **`true`**

The `equals()` Contract Part 2

4. Consistent :
   - Repeatedly comparing two objects yields the same result (assuming the state of the objects does not change)
5. **`x.equals(null)`** is always false

equals Method (con't)

```
public boolean equals(Object object)
{
    boolean equal;
    if (object != null && this.getClass() ==
            object.getClass())
    {
        Rectangle other = (Rectangle) object;
        equal = (this.width == other.width) &&
            (this.height == other.height);
    }
    else
    {
        equal = false;
    }
    return equal;
}
```

compareTo Method

- Required if your class implements the Comparable interface (i.e., its object can be compared, order, or sorted)
- Compares this object with the specified object for order
  - Returns a negative integer if this object is less than
  - Returns a positive integer if this object is greater than
  - Returns zero if this object is equal to the passed one
- Throws a ClassCastException if the specified object type cannot be compared to this object

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`

Implementing `compareTo`

- Implementing **`compareTo`** is similar to implementing equals
- Typically compare all of the attributes
  - Starting with the attribute that is most significant for ordering purposes and working your way down
- For the Rectangle class, the API states that the width is used form comparison; if the widths are equal, the heights are used

compareTo Method (con't)

```
public int compareTo(Rectangle r)
{
     int difference;
     if (this.width != r.width)
     {
          difference = this.width - r.width;
     }
     else
     {
          difference = this.height - r.height;
     }
     return difference;
}
```

`hashCode` Method

- Hash codes used to uniquely (ideal) correspond to an object's state (i.e., like a fingerprint or signature)
- Default uses memory address of the object
- Two objects with the same state should have the same hash code
- Hash-based containers use hash codes to organize and access elements efficiently
  - Performance increases with distinct hash codes

hashCode Method (2)

- Poor, but legal implementation:

```
public int hashCode()
{
    return 1;
}
```

- Better implementation:

```
public int hashCode()
{
    return this.getWidth() + this.getHeight();
}
```

Eclipse Demo in Lecture

- Demonstrate how Eclipse can help generate obligatory methods

Class-Specific Methods

- Exist to fulfill the purpose of the class
- For the Rectangle class:

```
public int getArea()
{
    return this.width * this.height;
}
public void scale(int factor)
{
    this.width = this.width * factor;
    this.height = this.height * factor;
}
```

Testing

- Similar to testing a utility class, but…
  - Must call a constructor to create an object
  - Test all constructors to ensure objects are correctly initialized
  - Test all mutators and accessors to ensure attributes are correctly modified and returned
  - Test the equals and compareTo methods adhere to conventional rules for equality and comparison

Documenting

- Similar to documenting a utility class
- Use Javadoc comments to describe the class and all public features
- The fully implemented and documented version of the Rectangle class is available here:

  www.eecs.yorku.ca/~buildIt/code//2/Rectangle.java.txt

Avoiding Code Duplication

- Easier to program, debug, and maintain
- Already seen with constructor chaining
- Other techniques:
  - Delegating to mutators
  - Delegating to accessors

Delegating to Mutators

- Use mutator whenever an attribute needs changing
  - E.g., can re-write the Rectangle constructor's body

```
this.width = width;
this.height = height;
```

becomes

```
this.setWidth(width);
this.setHeight(height);
```

- Single source of attribute modification
  - Can change how attributes are represented with minimal code modification (e.g., as doubles, in array, as a String)

Delegating to Accessors

- Use accessor whenever reading an attribute
  - E.g., can re-write the getArea method's body

```
this.width * this.height;
```

becomes

```
this.getWidth() * this.getHeight();
```

- Single source of attribute access
  - Can change how attributes are represented with minimal code modification (e.g., as doubles, in array, as a String)

Privacy Leaks

- A mutable object that is passed to or returned from a method can be changed

- Problems:
  - Private attributes become publicly accessible
  - Objects can be put into an inconsistent state

- Solution:
  - Make a copy of the object and save the copy
    - Use copy constructors

Avoiding Privacy Leaks

- Bad

```
public Date getDueDate()
{
        return dueDate; // Unsafe
}
```

- Good

```
public Date getDueDate()
{
        return new Date(dueDate.getTime()); // Avoid leak
}
```

Avoiding Privacy Leaks (con't)

- Bad

```
public void setDueDate(Date newDate)
{
        dueDate = newDate; // Unsafe
}
```

- Good

```
public void setDueDate(Date newDate)
{
        dueDate = new Date(newDate.getTime()); // Avoid leak
}
```

Immutable Classes

- A class defines an immutable type if an instance of the class cannot be modified after it is created
  - Each instance has its own constant state
    - More precisely, the externally visible state of each object appears to be constant
  - Java examples: **`String`**, **`Integer`** (and all of the other primitive wrapper classes)
- Advantages of immutability versus mutability
  - Easier to design, implement, and use
  - Can never be put into an inconsistent state after creation

Designing a Simple Immutable Class

- **`PhoneNumber`** API

PhoneNumber
- areaCode : short - exchangeCode : short - stationCode : short
+ PhoneNumber(int, int, int) + equals(Object) : boolean + getAreaCode() : short + getExchangeCode() : short + getStationCode() : short + toString() : String

Recipe for Immutability 1

1. Do not provide any methods that can alter the state of the object
   - Methods that modify state are called *mutators*
   - Java example of a mutator:

```
import java.util.Calendar;

public class CalendarClient {
  public static void main(String[] args)
  {
    Calendar now = Calendar.getInstance();
    // set hour to 5am
    now.set(Calendar.HOUR_OF_DAY, 5);
  }
}
```

Recipe for Immutability 2

2. Prevent the class from being extended.
   - Note that all classes **`extend java.lang.Object`**
   - One way to do this is to mark the class as **`final`**

```
public final class PhoneNumber
{
  // version 0
}
```

   - A **`final`** class cannot be extended
     - Don't confuse **`final`** variable and **`final`** classes
   - The reason for this step will become clear in a couple of weeks

Recipe for Immutability 3

3. Make all attributes **`final`**
   - Recall that Java will not allow a **`final`** attribute to be assigned to more than once
   - **`final`** attributes make your intent clear that the class is immutable

```
public final class PhoneNumber
{  // version 1
  private final short areaCode;
  private final short exchangeCode;
  private final short stationCode;
}
```

   - Notice that the attributes are not initialized here
     - That task belongs to the class constructors

Recipe for Immutability 4

4. Make all attributes **`private`**
   - This applies to all **`public`** classes (including mutable classes)
   - In **`public`** classes, strongly prefer **`private`** attributes
     - Avoid using **`public`** attributes
   - **`private`** attributes support encapsulation
     - Because they are not part of the API, you can change them (even remove them) without affecting any clients
     - The class controls what happens to **`private`** attributes
       - It can prevent the attributes from being modified to an inconsistent state

Recipe for Immutability 5

5. Prevent clients from obtaining a reference to any mutable attributes

   - Recall that **`final`** attributes have constant state only if the type of the attribute is a primitive or is immutable
   - If you allow a client to get a reference to a mutable attribute, the client can change the state of the attribute, and hence, the state of your immutable class

Attribute Caching

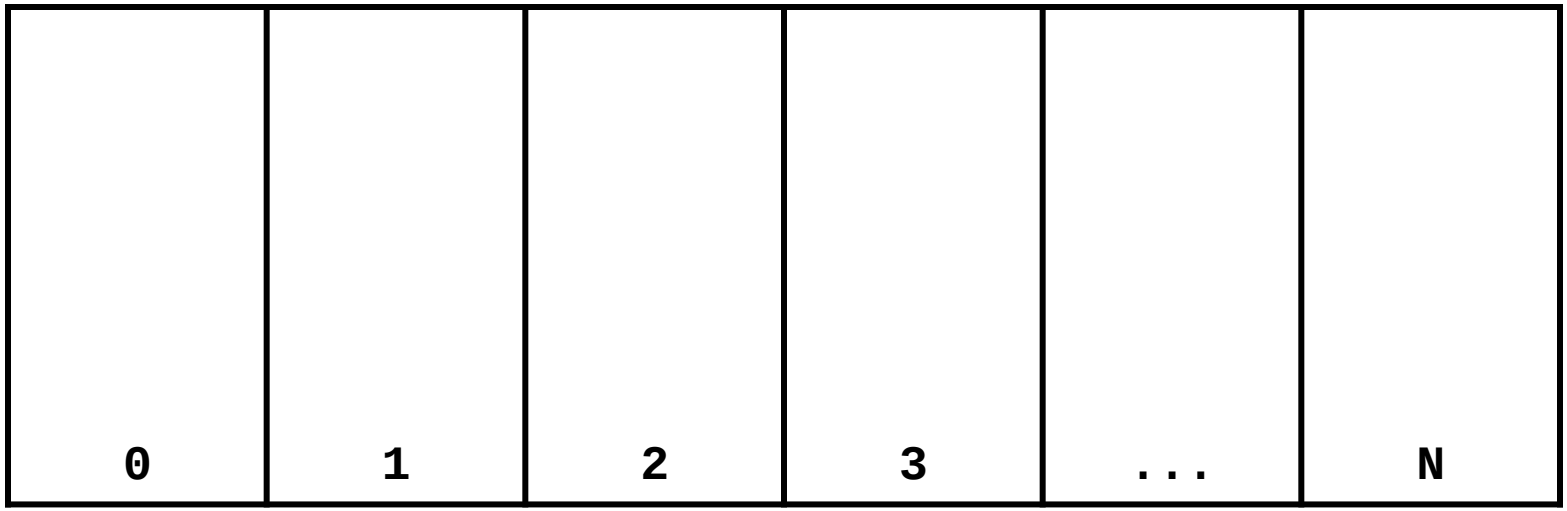
- Involves saving an object's characteristic, rather than recalculating it
- Must balance added redundancy (bad) with increased performance (good)
- Attribute is recalculated only when needed (i.e., when its components are modified) and returned when its accessor method is called

Class Invariants

- A property that objects of a particular class always hold
  - E.g., a Rectangle's width and height are always ≥ 0
- Must satisfy two constraints:
  1. The class invariant has to be true after each public constructor invocation (provided the constructor's precondition is also met)
  2. The class invariant has to be maintained by each public method invocation (provided the method's precondition is also met)

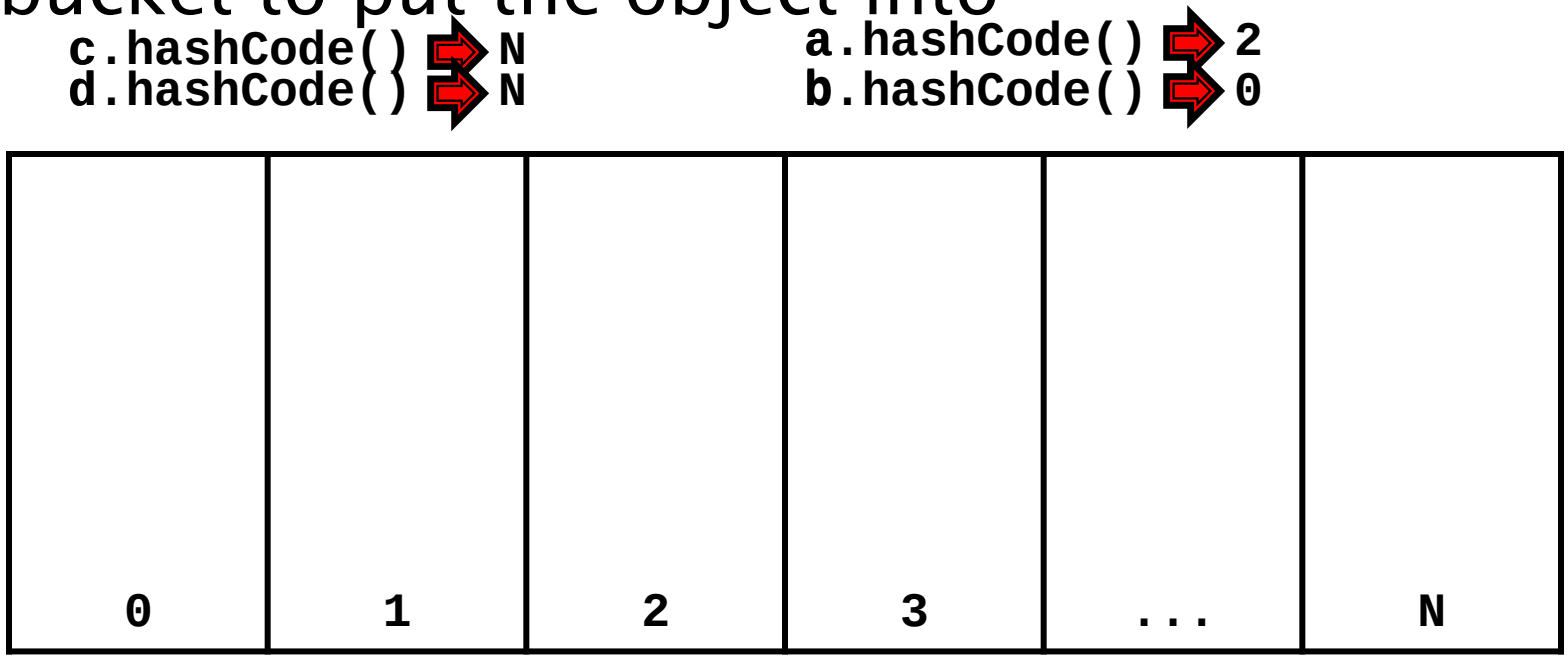
Hash Tables

- 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

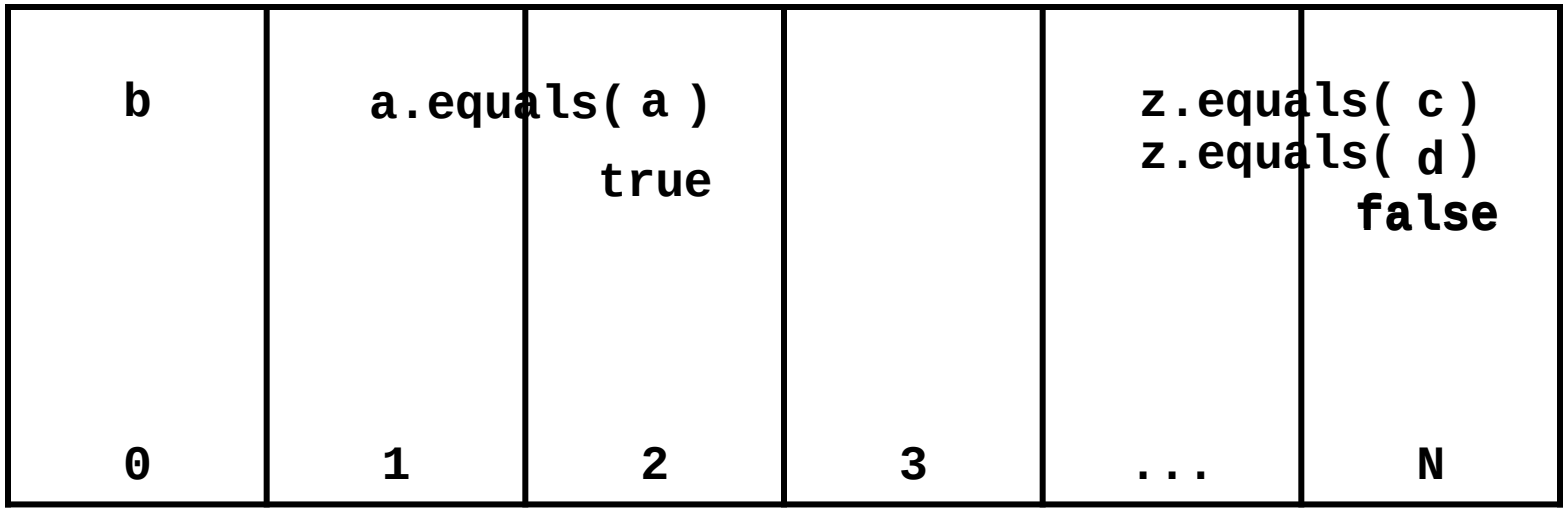
- To insert an object **a**, the hash table calls **a.hashCode()** method to compute which bucket to put the object into



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**



- 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()**

Something to Think About

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