Classes (Part 1)

Implementing non-static features

Goals

- implement a small immutable class with *non-static* attributes and methods
  - recipe for immutability
  - **`this`**
  - **`toString`** method
  - **`equals`** method

Value Type Classes

- a *value type* is a class that represents a value
  - examples of values: name, date, colour, mathematical vector
  - Java examples: **`String`**, **`Date`**, **`Integer`**

- the objects created from a value type class can be:
  - mutable: the state of the object can change
    - **`Date`**
  - immutable: the state of the object is constant once it is created
    - **`String`**, **`Integer`** (and all of the other primitive wrapper classes)

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

North American Phone Numbers

- North American Numbering Plan is the standard used in Canada and the USA for telephone numbers
- telephone numbers look like

416-736-2100

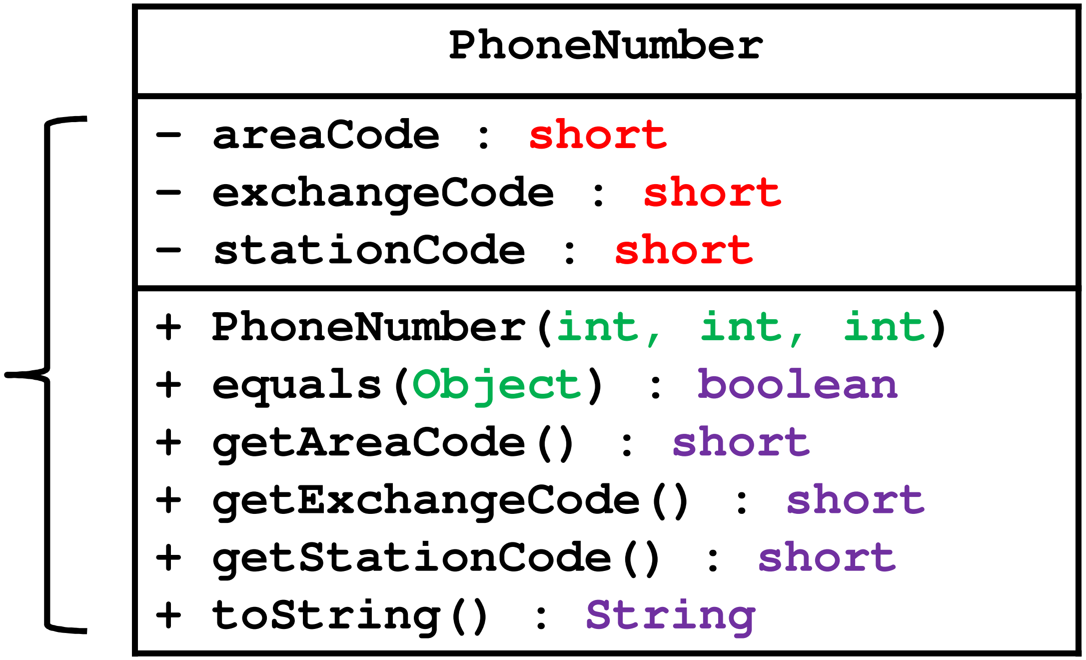
area
code

exchange
code

station
code

Designing a Simple Immutable Class

- **`PhoneNumber`** API



none of these features are static

PhoneNumber

```
- areaCode : short
- exchangeCode : short
- stationCode : short
```

```
+ PhoneNumber(int, int, int)
+ equals(Object) : boolean
+ getAreaCode() : short
+ getExchangeCode() : short
+ getStationCode() : short
+ toString() : String
```

```
package cse1030;

public class PhoneNumber {

}
```

Recipe for Immutability

- the recipe for immutability in Java is described by Joshua Bloch in the book *Effective Java**

1. Do not provide any methods that can alter the state of the object
2. Prevent the class from being extended
3. Make all fields **`final`**
4. Make all fields **`private`**
5. Prevent clients from obtaining a reference to any mutable fields

revisit when we talk about inheritance

revisit when we talk about composition

*highly recommended reading if you plan on becoming a Java programmer

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
   - one way to do this is to mark the class as **`final`**

- a **`final`** class cannot be extended using inheritance
  - don't confuse **`final`** variable and **`final`** classes

- the reason for this step will become clear in a couple of weeks

```
package cse1030;

public final class PhoneNumber {

}
```

Recipe for Immutability 3

3. Make all fields **`final`**
   - recall that **`final`** means that the field can only be assigned to once
   - **`final`** fields make your intent clear that the class is immutable

```
package cse1030;

public final class PhoneNumber {
  final int areaCode;
  final int exchangeCode;
  final int stationCode;

}
```

Recipe for Immutability 4

4. Make all fields **`private`**
   - this applies to all **`public`** classes (including mutable classes)
   - in **`public`** classes, strongly prefer **`private`** fields
     - and avoid using **`public`** fields
   - **`private`** fields 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`** fields
       - it can prevent the fields from being modified to an inconsistent state

```java
package cse1030;

public final class PhoneNumber {
  private final int areaCode;
  private final int exchangeCode;
  private final int stationCode;

}
```

Recipe for Immutability 5

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

- recall that **`final`** fields 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 field, the client can change the state of the field, and hence, the state of your immutable class
- revisit this point when we talk about composition
  - also, none of our fields are reference types so we don't have to worry about this point

`this`

- every non-static method of a class has an implicit parameter called **`this`**
- recall that a non-static method requires an object to call the method

```
// client of PhoneNumber

PhoneNumber num = new PhoneNumber(416, 736, 2100);
int areaCode = num.getAreaCode();    // get the
                                     // area code that
                                     // belongs to num
```

- inside **`getAreaCode`**, **`this`** is a reference to object used to invoke the method

getAreaCode

- how does the method **`getAreaCode()`** get the area code for the correct instance?
  - **`this`** is a reference to the calling object

```
/**
 * Get the area code of this phone number.
 *
 * @return the area code of this phone number
 */
public int getAreaCode() {
  return this.areaCode;
}
```

return the area code belonging to the **`PhoneNumber`** object that was used to invoke the method

getExchangeCode and getStationCode

- **`getExchangeCode()`** and **`getStationCode()`** are very similar

```
/**
 * Get the exchange code of this phone number.
 *
 * @return the exchange code of this phone number
 */
public int getExchangeCode() {
  return this.exchangeCode;
}
```

return the exchange code belonging to the **`PhoneNumber`** object that was used to invoke the method

getExchangeCode and getStationCode

- **`getExchangeCode()`** and **`getStationCode()`** are very similar

```
/**
 * Get the station code of this phone number.
 *
 * @return the station code of this phone number
 */
public int getStationCode() {
  return this.stationCode;
}
```

return the station code belonging to the **`PhoneNumber`** object that was used to invoke the method

`toString()`

- recall that every class extends **`java.lang.Object`**
- **`Object`** defines a method **`toString()`** that returns a **`String`** representation of the calling object
  - we can call **`toString()`** with our current **`PhoneNumber`** class

```
// client of PhoneNumber

PhoneNumber num = new PhoneNumber(416, 736, 2100);
System.out.println(num.toString());
```

  - this prints something like

    `phonenumber.PhoneNumber@19821f`

`toString()`

- **`toString()`** should return a concise but informative representation that is easy for a person to read
- it is recommended that all subclasses override this method
  - this means that any non-utility class you write should redefine the **`toString()`** method
    - in this case, our new **`toString()`** method has the same declaration as **`toString()`** in **`java.lang.Object`**

`toString()`

- it is "easy" to override **`toString()`** for our class

```java
/**
 * Returns a string representation of this phone number. The string starts
 * with the area code inside of parenthesis, followed by a space, followed by
 * the exchange code, followed by a hyphen, followed by the station code. The
 * area code and exchange code always have three digits (zero-padded), and the
 * station code always has four digits (zero-padded). For example, the string
 * representation of the phone number 416-736-2100 is:
 *
 * <p>
 * <code>(416) 736-2100</code>
 *
 * @return a string representation of this phone number
 * @see java.lang.Object#toString()
 */
@Override
public String toString() {
  return String.format("(%1$03d) %2$03d-%3$04d",
                                this.areaCode,
                                this.exchangeCode,
                                this.stationCode);
}
```

Constructors

Constructors

- constructors are responsible for initializing instances of a class
  - usually, a constructor will set the fields of the object to:
    - some reasonable default values, or
    - some client specified values,
    - or some combination of the two

[notes 2.2.3]

Constructors

- a constructor declaration looks a little bit like a method declaration:
  - the name of a constructor is the same as the class name
  - a constructor may have an access modifier (but no other modifiers)

```java
public PhoneNumber() {

}
```

the *default* constructor
(has no parameters)

```java
public PhoneNumber(int areaCode,
                   int exchangeCode,
                   int stationCode) {

}
```

a constructor with
three parameters

Constructors

- every constructor has an implicit **`this`** parameter
  - the **`this`** parameter is a reference to the object that is currently being constructed

```
public PhoneNumber() {
  this.areaCode = 800;
  this.exchangeCode = 555;
  this.stationCode = 1111;
}

public PhoneNumber(int areaCode,
                   int exchangeCode, int stationCode) {
  this.areaCode = areaCode;
  this.exchangeCode = exchangeCode;
  this.stationCode = stationCode;
}
```

Bell Canada operator phone number?

client specified phone number

Constructors

- a constructor will often need to validate its arguments
  - because you generally should avoid creating objects with invalid state

- what are valid area codes, exchange codes, and station codes?
  - we will assume:
    - must not be negative
    - area code and exchange codes < 1,000
    - station code < 10,000
  - reality is more complicated…

```
public PhoneNumber(int areaCode,
                   int exchangeCode, int stationCode) {

  if (areaCode < 0 || areaCode > 999) {
    throw new IllegalArgumentException("bad area code");
  }
  if (exchangeCode < 0 || exchangeCode > 999) {
    throw new IllegalArgumentException("bad exchange code");
  }
  if (stationCode < 0 || stationCode > 9999) {
    throw new IllegalArgumentException("bad station code");
  }
  this.areaCode = areaCode;
  this.exchangeCode = exchangeCode;
  this.stationCode = stationCode;
}
```

Comment on Immutability

- notice that our constructors make it impossible for a client to create an invalid phone number
- also recall that our class is immutable
  - i.e., the client cannot change a phone number once it is created
- the above two features guarantee that all **`PhoneNumber`** objects will be valid phone numbers

Classes (Part 2)

Implementing non-static features

Goals

- finish implementing the immutable class **`PhoneNumber`**
  - **`equals()`**
- implement a mutable class

Overriding `equals()`

- suppose you write a value class that extends **`Object`** but you do not override **`equals()`**
  - what happens when a client tries to use **`equals()`**?
    - **`Object.equals()`** is called

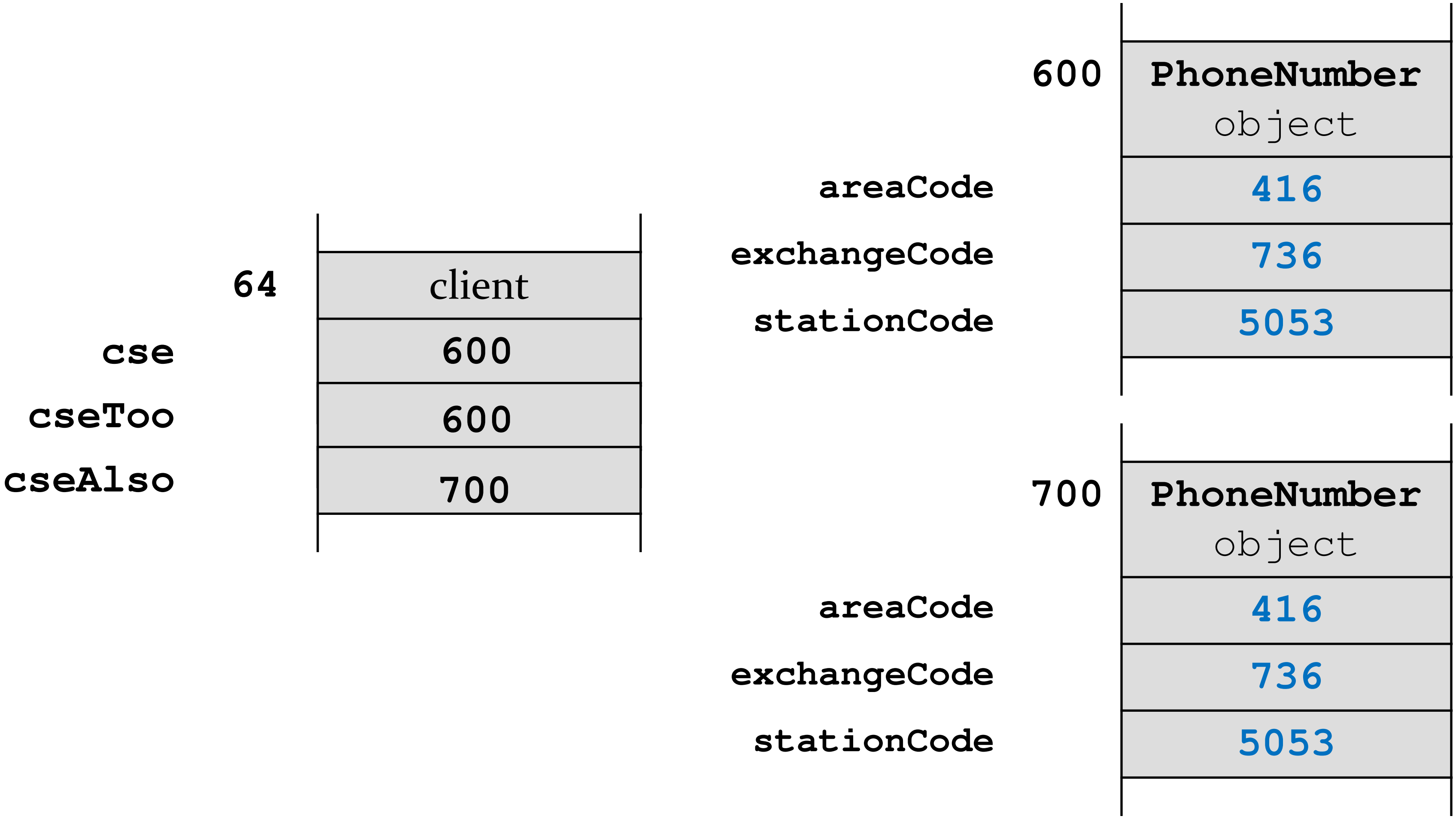
```
// PhoneNumber client

PhoneNumber cse = new PhoneNumber(416, 736, 5053);
System.out.println( cse.equals(cse) );       // true

PhoneNumber cseToo = cse;
System.out.println( cseToo.equals(cse) );    // true

PhoneNumber cseAlso = new PhoneNumber(416, 736, 5053);
System.out.println( cseAlso.equals(cse) );   // false!
```

[notes 2.2.4]



64	client
cse	600
cseToo	600
cseAlso	700

600	PhoneNumber object
areaCode	416
exchangeCode	736
stationCode	5053

700	PhoneNumber object
areaCode	416
exchangeCode	736
stationCode	5053

`Object.equals()`

- `Object.equals()` checks if two references refer to the same object
  - `x.equals(y)` is true if and only if `x` and `y` are references to the same object

`PhoneNumber.equals()`

- most value classes should support logical equality
  - an instance is equal to another instance if their states are equal
    - e.g. two **`PhoneNumbers`** are equal if their area, exchange, and station codes have the same values

- 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()`**. Look into the best Java source code or open an arbitrary Java textbook and take a look at what you find. Chances are good that you will find several different approaches and a variety of recommendations."
    - Angelika Langer, Secrets of equals() – Part 1
  - `http://www.angelikalanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals.html`

- what we are about to do does not always produce the result you might be looking for
    - but it is always satisfies the **`equals()`** contract
    - and it's what the notes and textbook do

CSE1030 Requirements for `equals`

1. an instance is equal to itself
2. an instance is never equal to `null`
3. only instances of the exact same type can be equal
4. instances with the same state are equal

1. An Instance is Equal to Itself

- **`x.equals(x)`** should always be **`true`**
- also, **`x.equals(y)`** should always be true if **`x`** and **`y`** are references to the same object
- you can check if two references are equal using **`==`**

```
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }

}
```

2. An Instance is Never Equal to `null`

- Java requires that **`x.equals(null)`** returns **`false`**
- and you must not throw an exception if the argument is **`null`**
  - so it looks like we have to check for a **`null`** argument…

```
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  if (obj == null) {
    return false;
  }


}
```

3. 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
- you can find the class of an object using **`Object.getClass()`**

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

- Returns the runtime class of an object.

```
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  if (obj == null) {
    return false;
  }
  if (this.getClass() != obj.getClass()) {
    return false;
  }


}
```

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

- unfortunately, we cannot yet retrieve the attributes of the parameter **`obj`** because it is declared to be an **`Object`** in the method signature
  - we need a cast

```
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  if (obj == null) {
    return false;
  }
  if (this.getClass() != obj.getClass()) {
    return false;
  }
  PhoneNumber other = (PhoneNumber) obj;


}
```

Instances with Same State are Equal

- there is a recipe for checking equality of fields

1. if the field 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

```
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  if (obj == null) {
    return false;
  }
  if (this.getClass() != obj.getClass()) {
    return false;
  }
  PhoneNumber other = (PhoneNumber) obj;
  if (areaCode != other.areaCode) {
    return false;
  }
  if (exchangeCode != other.exchangeCode) {
    return false;
  }
  if (stationCode != other.stationCode) {
    return false;
  }
  return true;
}
```

The `equals()` Contract

- for reference values `equals()` is
  1. reflexive
  2. symmetric
  3. transitive
  4. consistent
  5. must not throw an exception when passed `null`

The `equals()` contract: Reflexivity

1. reflexive :
    - an object is equal to itself
    - `x.equals(x)` is `true`

The `equals()` contract: Symmetry

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`**

The `equals()` contract: Transitivity

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: Consistency

4. consistent :
    - repeatedly comparing two objects yields the same result (assuming the state of the objects does not change)

The `equals()` contract: Non-nullity

5. `x.equals(null)` is always `false` and never does not throw an exception

The **`equals()`** contract and **`getClass()`**

- using **`getClass()`** makes it relatively easy to ensure that the **`equals()`** contract is obeyed
  - e.g., symmetry and transitivity are easy to ensure

- however, using **`getClass()`** means that your **`equals()`** method won't work as expected in inheritance hierarchies
  - more on this when we talk about inheritance

One more thing regarding `equals()`

- if you override `equals()` you must override `hashCode()`
  - otherwise, the hashed containers won't work properly

- we will see how to implement `hashCode()` in the next lecture or so
  - also a discussion about how the hashed containers actually work

Mutable Classes

Mutable Classes

- a mutable class can change how its state appears to clients
  - recall that immutable classes are generally easier to implement and use
  - so why would we want a mutable class?
    - because you need a separate immutable object for every value you need to represent
      - example is String concatenation

Reading a Text File into a String

```
BufferedReader in =
    new BufferedReader(new FileReader(file));
String contents = "";
while (in.ready()) {
  contents = contents + in.readLine();
}
```

creates a new String object
to perform the concatenation
each iteration of the loop

Reading a Text File into a StringBuilder

```
BufferedReader in =
    new BufferedReader(new FileReader(file));
StringBuilder contents = new StringBuilder();
while (in.ready()) {
  contents.append(in.readLine());
}
```

new String not created for each iteration

Example Mutable class

- we will create a class to represent 2-dimensional vectors

What Can Mathematical Vectors Do?

- add
- subtract
- multiply by scalar
- set coordinates
- get coordinates
- construct
- equals
- toString

Vector2D

`- x: double`
`- y: double`
`- name: String`

`+ Vector2D()`
`+ Vector2D(double, double)`
`+ Vector2D(String, double, double)`
`+ Vector2D(Vector2D)`
`+ add(Vector2D): void`
`+ equals(Object): boolean`
`+ getX(): double`
`+ getY(): double`
`+ length(): double`
`+ multiply(double): void`
`...`

Constructors

- recall that the role of the constructor is to initialize the attributes of a new object
  - for **`Vector2D`** we need to initialize **`x`**, **`y`**, and **`name`**
- we have 4 overloaded constructors

`Vector2D()` Create the vector (0, 0) with no name.
`Vector2D(double x, double y)` Create the vector (x, y) with no name.
`Vector2D(String name, double x, double y)` Create the vector (x, y) with the given name.
`Vector2D(Vector2D other)` Create a new vector that is equal to the given vector.

Constructors

```
public Vector2D() {
  this.x = 0;
  this.y = 0;
  this.name = null;
}

public Vector2D(double x, double y) {
  this.x = x;
  this.y = y;
  this.name = null;
}
```

Constructors

```
public Vector2D(String name, double x, double y) {
  this.x = x;
  this.y = y;
  this.name = name;
}

public Vector2D(Vector2D other) {
  this.x = other.x;
  this.y = other.y;
  this.name = other.name;
}
```

Avoiding Code Duplication

- notice that the constructor bodies are almost identical to each other
- whenever you see duplicated code you should consider moving the duplicated code into a method
- in this case, one of the constructors already does everything we need to implement the other constructors…

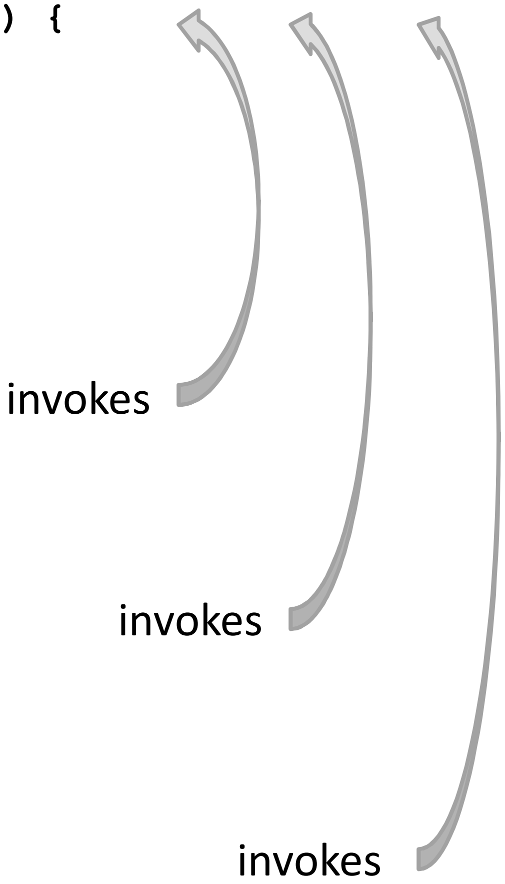
Constructors

```
public Vector2D(double x, double y, String name) {
  this.x = x;
  this.y = y;
  this.name = name;
}

public Vector2D() {
  this(0, 0, null);
}

public Vector2D(double x, double y) {
  this(x, y, null);
}

public Vector2D(Vector2D other) {
  this(other.x, other.y, other.name);
}
```



invokes

invokes

invokes

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

Accessor Methods

- recall that accessor methods return information about the state of the object
  - for **`Vector2D`** we need to return information about **`x`**, **`y`**, and **`name`**
- we have 3 accessor methods

`double getX()` Get the x coordinate of the vector.
`double getY()` Get the y coordinate of the vector.
`String getName()` Get the name of the vector.

Accessor Methods

```
public double getX() {
  return this.x;
}

public double getY() {
  return this.y;
}

public double getName() {
  return this.name;
}
```

Mutator Methods

- recall that mutator methods allow a client to manipulate the state of the object
  - for **`Vector2D`** we need to allow the client to manipulate **`x`**, **`y`**, and **`name`**

Mutator Methods

- we have 5 mutator methods

`void setX(double x)` Set the x coordinate of the vector.
`void setY(double y)` Set the y coordinate of the vector.
`void setName(String name)` Set the name of the vector.
`void set(double x, double y)` Set the x and y coordinate of the vector
`void set(String name, double x, double y)` Set the name, x, and y coordinate of the vector

`setX()`, `setY()`, and `set()`

```
public void setX(double x) {
  this.x = x;
}

public void setY(double y) {
  this.y = y;
}

public void setName(String name) {
  this.name = name;
}

public void set(double x, double y) {
  this.setX(x);
  this.setY(y);
}

public void set(String name, double x, double y) {
  this.setName(name);
  this.set(x, y);
}
```

Equals

- recall that most value type classes will want their own version of **`equals`**
  - we shall say that two vectors are equal if their **`x`**, and **`y`** coordinates are equal
    - i.e., two vectors might be equal even if their names are different

`boolean equals(Object obj)`
Compares two vectors for equality.

equals()

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }







  return eq;
}
```

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }
  else if (obj != null && this.getClass() == obj.getClass()) {



  }
  return eq;
}
```

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }
  else if (obj != null && this.getClass() == obj.getClass())  {
    Vector2d other = (Vector2d) obj;


  }
  return eq;
}
```

This version works most of the time (except when it doesn't!)

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }
  else if (obj != null && this.getClass() == obj.getClass())  {
    Vector2d other = (Vector2d) obj;
    eq = this.getX() == other.getX() &&
         this.getY() == other.getY();
  }
  return eq;
}
```

This version always works.

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if (obj == this) {
    eq = true;
  }
  else if (obj != null && this.getClass() == obj.getClass()) {
    Vector2d other = (Vector2d) obj;
    eq = Double.compare(this.getX(), other.getX()) == 0 &&
         Double.compare(this.getY(), other.getY()) == 0;
  }
  return eq;
}
```

== vs Double.compare

- the issue here is quite subtle
- if you use == to compare the coordinates then

```
Vector2D u = new Vector2D(0.0 / 0.0, 1.0);  // (NaN, 1.0)
Vector2D v = new Vector2D(u);                // (NaN, 1.0)
boolean eq = u.equals(v);
```

`eq` will be **`false`** because **`NaN == NaN`** is always **`false`**

  - NaN means "not a number" and is used to represent a mathematically undefined number
    - such as occurs when you divide zero by zero
    - the behavior of NaN is defined in the IEEE 754 standard for floating point arithmetic (i.e., this is not just a Java issue)

== vs Double.compare

- if you use == to compare the coordinates then all hash based collections and all sets will behave strangely with vectors having NaN as a component

```
Set<Vector2D> set = new HashSet<Vector2D>();
Vector2D u = new Vector2D(0.0 / 0.0, 1.0);  // (NaN, 1.0)
Vector2D v = new Vector2D(u);               // (NaN, 1.0)
set.add(u);
set.add(v);
System.out.println(set.size());             // prints 2
```

- sets are supposed to reject duplicate elements but there are 2 identical vectors in **`set`**
  - occurs because **`Set`** uses **`equals`** to check for duplicates

== vs Double.compare

- if you use **`Double.compare`** to compare the coordinates then

```
Vector2D u = new Vector2D(0.0 / 0.0, 1.0);  // (NaN, 1.0)
Vector2D v = new Vector2D(u);               // (NaN, 1.0)
boolean eq = u.equals(v);
```

`eq` will be **`true`** because **`Double.compare`** is implemented to allow for equality of **`NaN`**

- checking for equality of **`NaN`** can be useful when trying to track down errors in computations
- also the hash based collections and sets will work as expected

== vs Double.compare

- there is a side effect of using **`Double.compare`** to compare the coordinates

```
Vector2D u = new Vector2D(0.0, 1.0);    // (0.0, 1.0)
Vector2D v = new Vector2D(-0.0, 1.0);   // (-0.0, 1.0)
boolean eq = u.equals(v);
```

`eq` will be **`false`** because **`Double.compare`** considers 0.0 and -0.0 to be unequal

  - can you see how to implement equals to allow for equality of **`NaN`** and equality of 0.0 and -0.0?

== vs Double.compare

- the real issue here is that floating point arithmetic is tricky and affects every programming language
- a good starting point for learning more about some of the issues involved
  - http://floating-point-gui.de/

Observe That…

- instead of directly using the fields, we use accessor methods where possible
  - this reduces code duplication, especially if accessing an field requires a lot of code
  - this gives us the possibility to change the representation of the fields in the future
    - as long as we update the accessor methods (but we would have to do that anyway to preserve the API)
    - for example, instead of two attributes **`x`** and **`y`**, we might want to use an array or some sort of **`Collection`**

- the notes [notes 2.3.1] call this *delegating to accessors*

Observe That…

- instead of directly modifying the attributes, we use mutator methods where possible
    - this reduces code duplication, especially if modifying an attribute requires a lot of code
    - this gives us the possibility to change the representation of the attributes in the future
        - as long as we update the mutator methods (but we would have to do that anyway to preserve the API)
    - for example, instead of two attributes **`x`** and **`y`**, we might want to use an array or some sort of **`Collection`**

- the notes [notes 2.3.1] call this *delegating to mutators*

Things to Think About

- how do you implement **`Vector2D`** using an array to store the coordinates?
- how do you implement **`Vector2D`** using a **`Collection`** to store the coordinates?
- how do you implement **`VectorND`**, an N-dimensional vector?

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
```

[notes 2.3.5]

Arrays as Containers

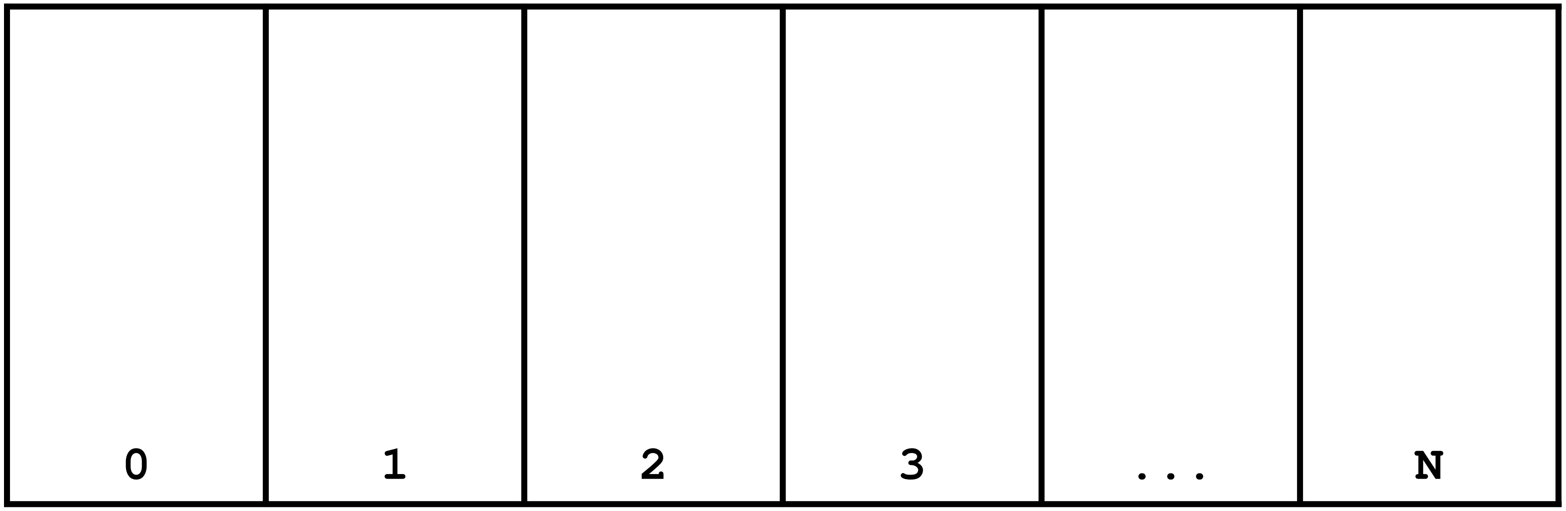
- suppose you have an array of unique **`PhoneNumbers`**
  - how do you compute whether or not the array contains a particular **`PhoneNumber`**?

```
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 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()`**

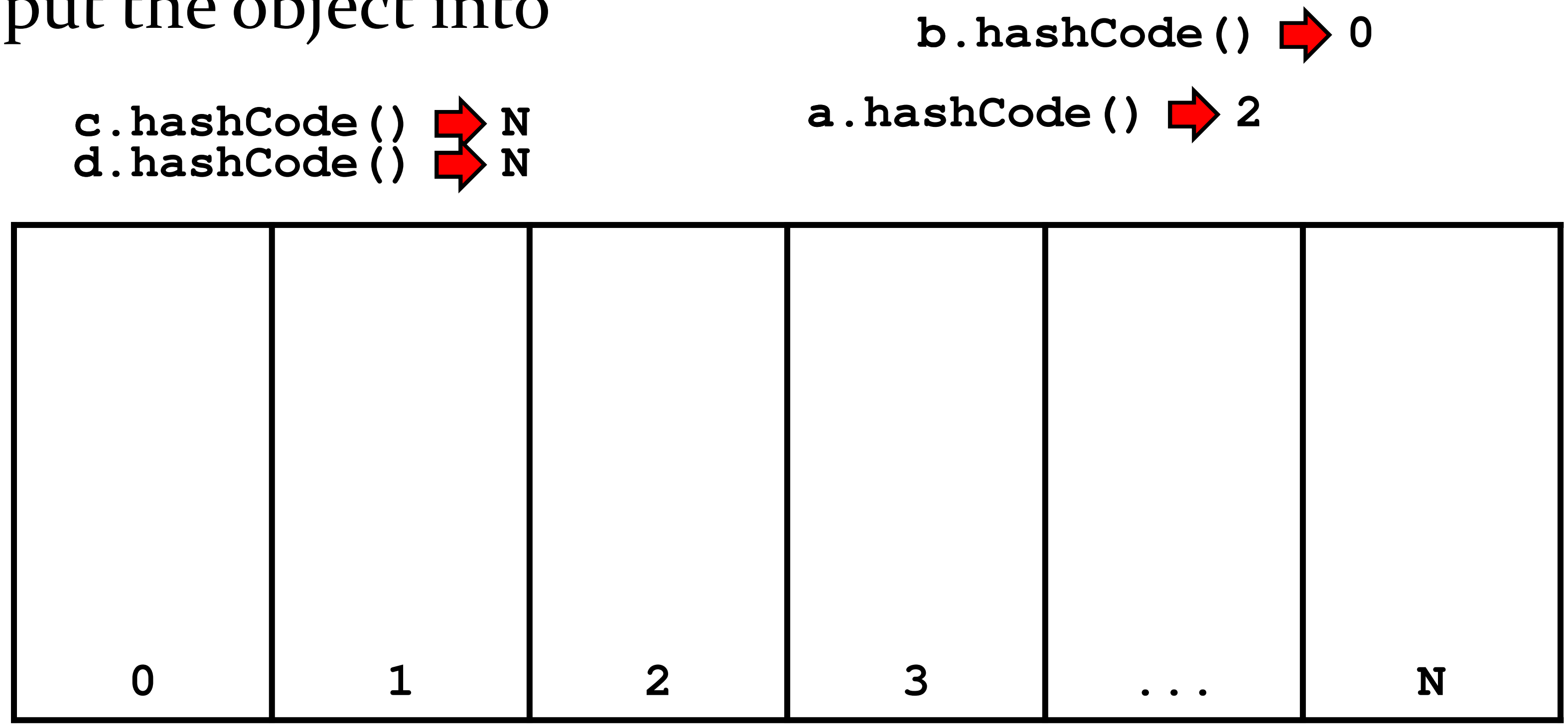
Hash Tables

- 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



`b.hashCode()` ➡ `0`

`a.hashCode()` ➡ `2`

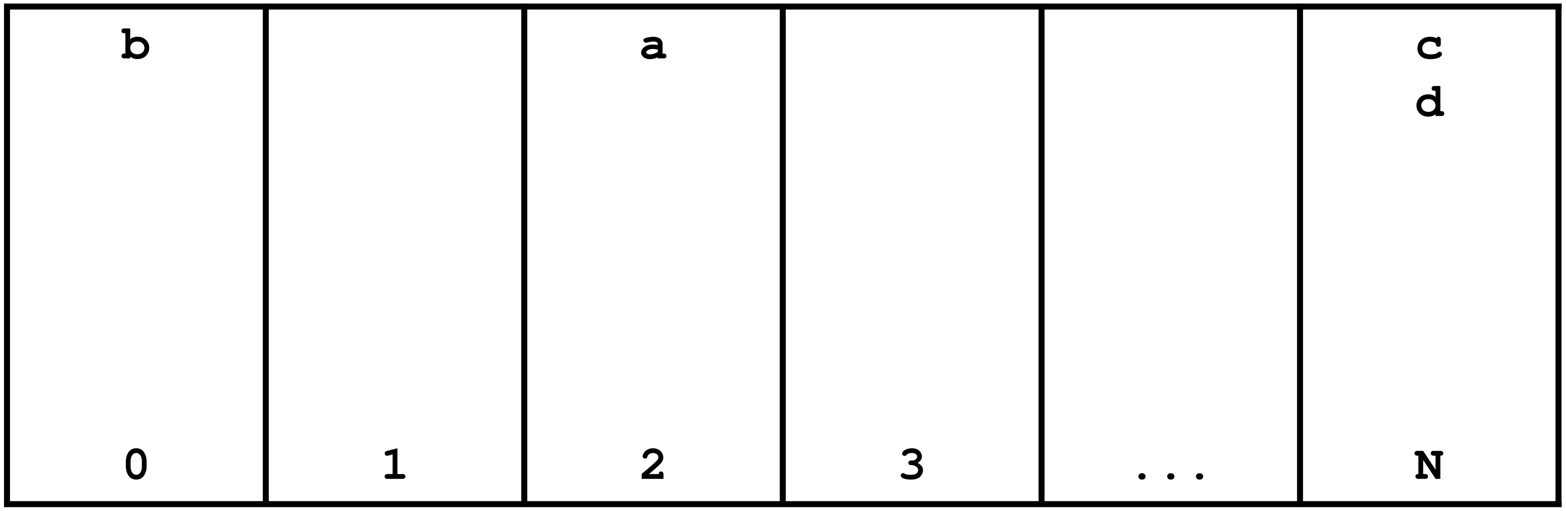
`c.hashCode()` ➡ **`N`**

`d.hashCode()` ➡ **`N`**

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

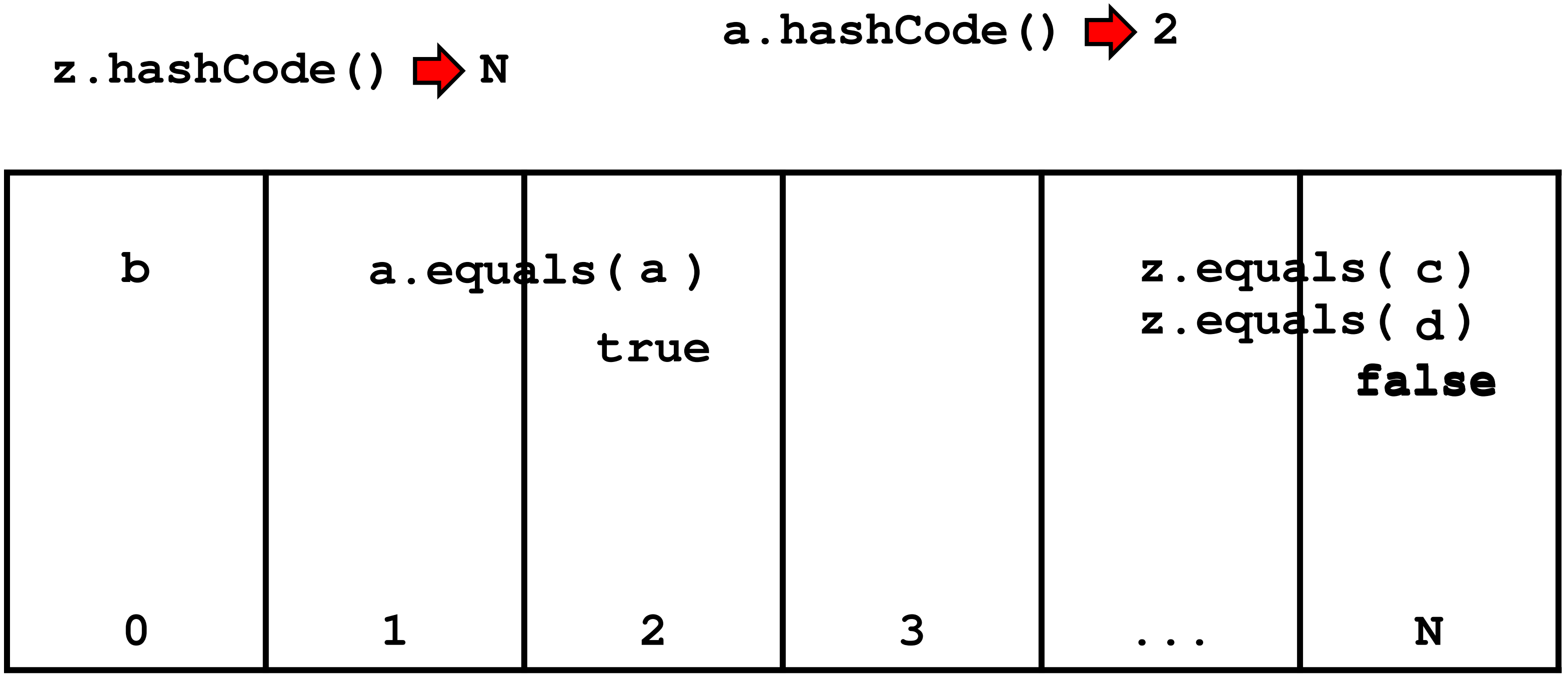
Insertion into a Hash Table

- 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



`a.hashCode()` ➡ **2**

`z.hashCode()` ➡ **N**

`b`

`a.equals( a )`

`true`

`z.equals( c )`

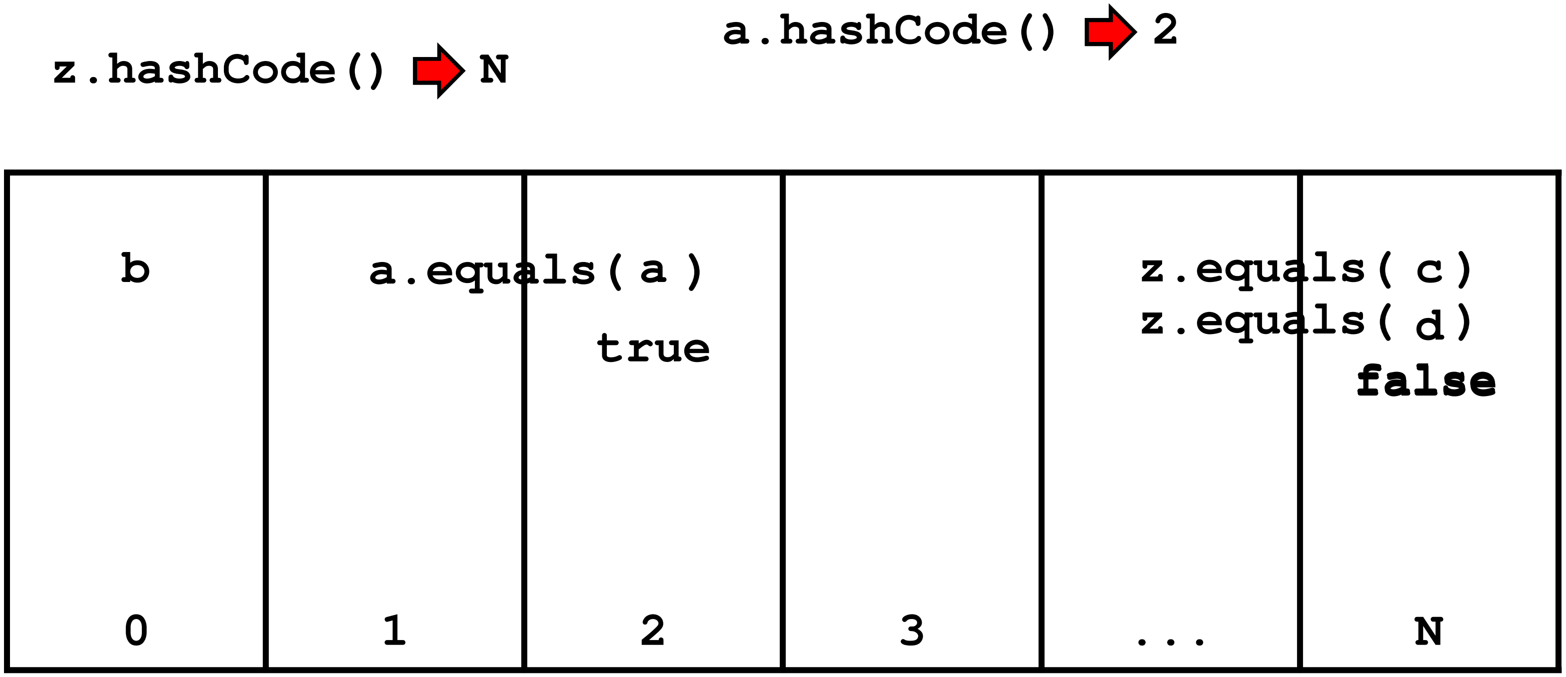
`z.equals( d )`

`false`

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



`a.hashCode()` → `2`

`z.hashCode()` → `N`

b		a.equals(a) true			z.equals(c) z.equals(d) 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 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)`** *

---

* unless you're from Naples

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 `PhoneNumber`s 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
    - `String`s can be compared in dictionary order
    - `Date`s can be compared in chronological order
    - you might compare `Vector2D`s by their length
    - `Die`s 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 `Collection`s 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.compare`** or **`Double.compare`** 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++

[notes 3.2]

- 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;
  }

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

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

static method
can only use
static attributes

non-static method
can use
non-static attributes

and static attributes