

# Classes (Part 2)

Implementing non-static features

# Goals

---

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

# Overriding `equals()`

---

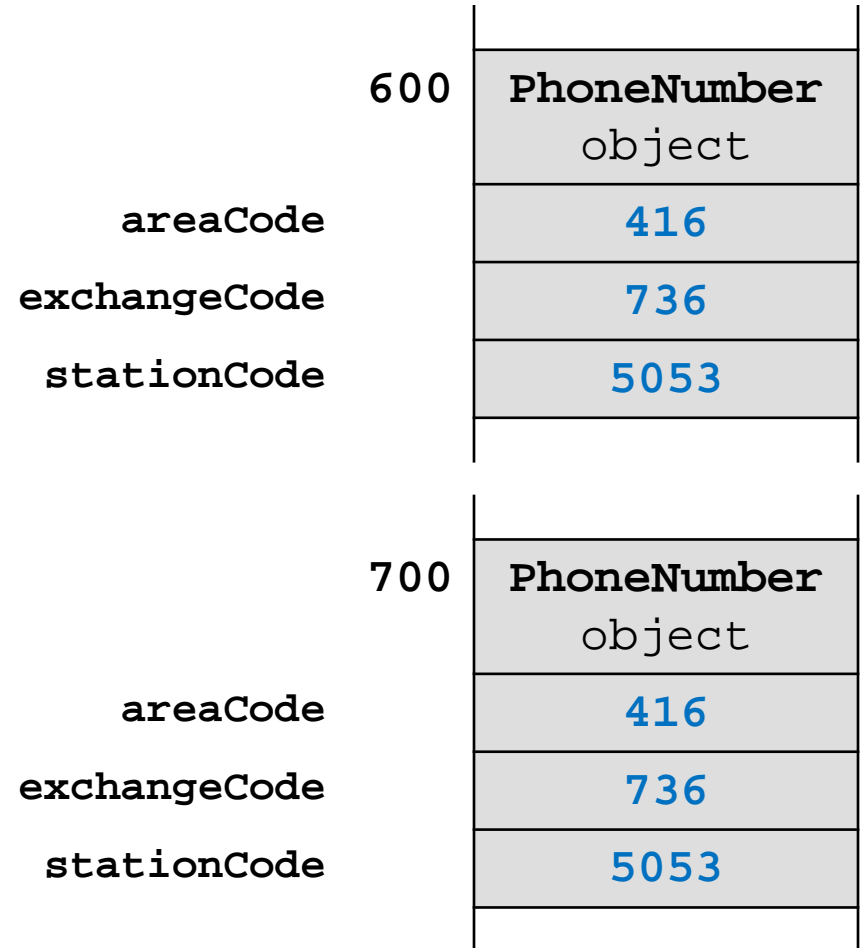
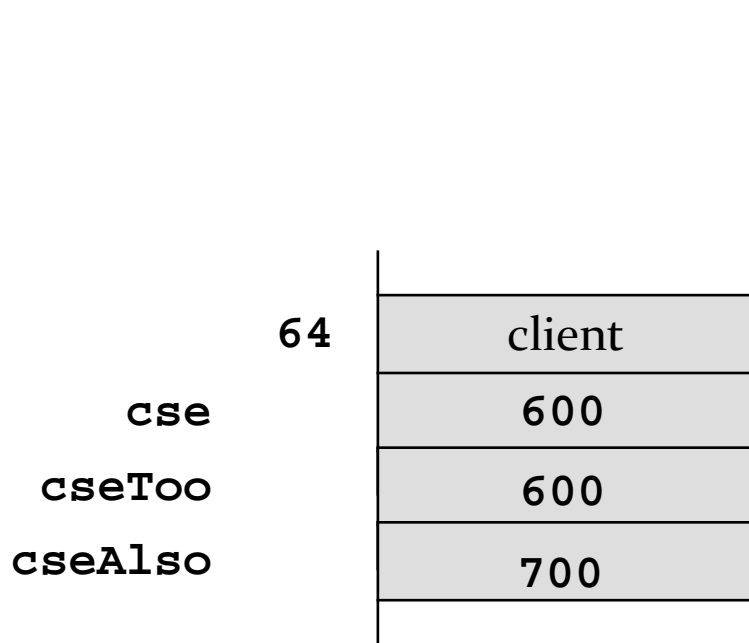
- ▶ suppose you write a value class that extends `Object` but you do not override `equals()`
  - ▶ what happens when a client tries to use `equals()`?
    - ▶ `Object.equals()` is called

```
// 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!
```



# 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

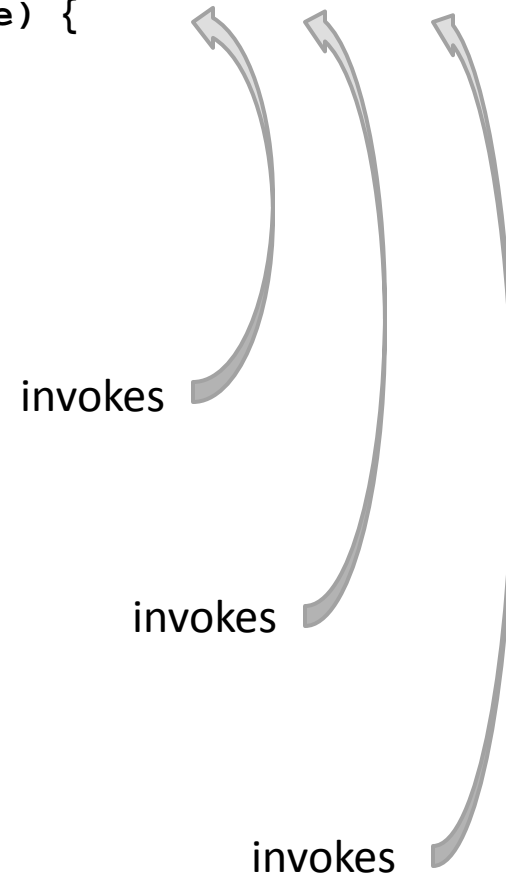
---

```
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);  
}
```



# 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?