Creating an Immutable Class

Based on slides by Prof. Burton Ma

Value Type Classes

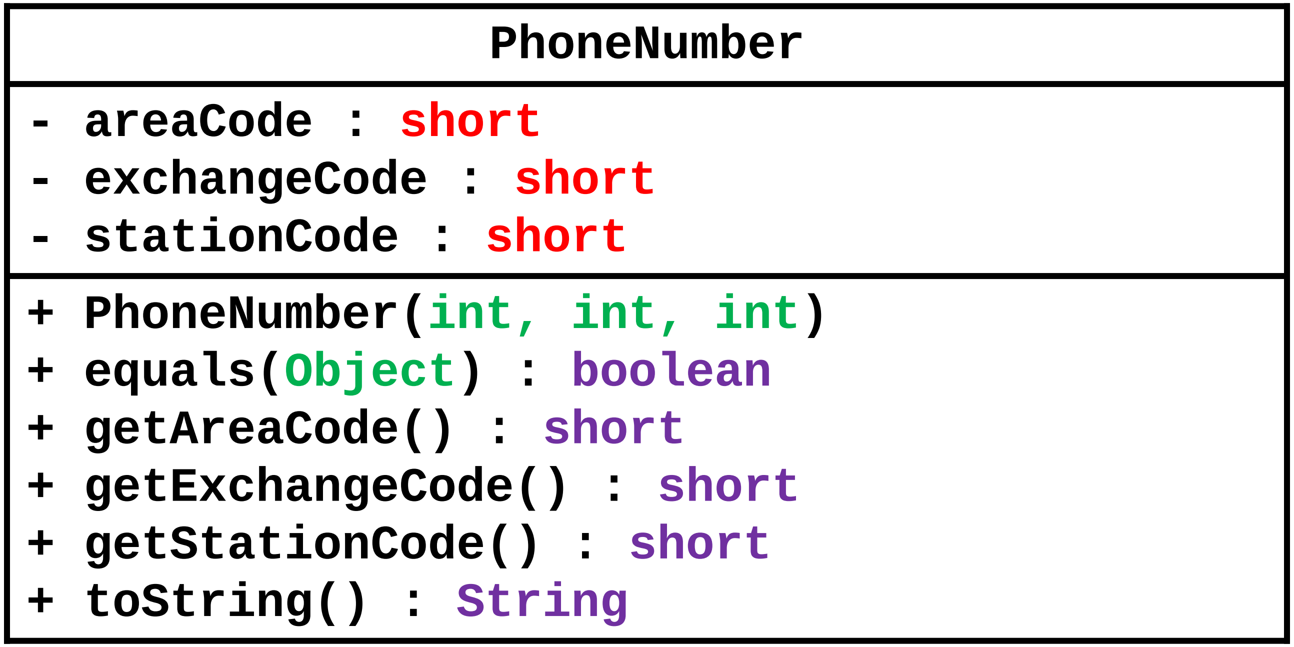
- A *value type* is a class that represents a value
  - Examples of values: name, date, colour, mathematical vector
  - Java examples: **`String`**, **`Date`**, **`Integer`**, **`List`**

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*

```
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

this

- Every non-static method of a class has an implicit parameter called **this**
- Recall that a non-static method requires an

```
// client of PhoneNumber

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

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

```
public final class PhoneNumber
{ // version 2; see version 1 for attributes

  public short getAreaCode()
  { return this.areaCode; }

  public short getExchangeCode()
  { return this.exchangeCode; }

  public short getStationCode()
  { return this.stationCode; }
}
```

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**

```
// client of PhoneNumber

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

  - This prints something like
    phonenumber.PhoneNumber@19821f

- `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`**

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

```
public final class PhoneNumber
{ // version 3; see versions 1 and 2 for attributes and methods

  @Override public String toString()
  {
    return String.format("(%1$03d) %2$03d-%3$04d",
                         this.areaCode,
                         this.exchangeCode,
                         this.stationCode);
  }
}
```

Constructors

- Constructors are responsible for initializing instances of a class
- 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)
- Every constructor has an implicit **`this`** parameter
- A constructor will often need to validate its arguments
  - Because you generally should avoid creating objects with invalid state

[notes 2.2.3], [AJ 4.4]

No Parameter Validation

```
public final class PhoneNumber
{ // version 4; see versions 1, 2, and 3 for attributes and methods

  private final short areaCode;
  private final short exchangeCode;
  private final short stationCode;

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

parameter names shadow attribute names

With Parameter Validation

```
public final class PhoneNumber
{ // version 4; see versions 1, 2, and 3 for attributes and methods

        public PhoneNumber(int areaCode,
            int exchangeCode,
            int stationCode)
 {
  rangeCheck(areaCode, 999, "area code");
  rangeCheck(exchangeCode, 999, "exchange code");
  rangeCheck(stationCode, 9999, "station code");
  this.areaCode = (short) areaCode;
  this.exchangeCode = (short) exchangeCode;
  this.stationCode = (short) stationCode;
 }
```

parameter names shadow attribute names

```
 private static void rangeCheck(int num,
                                int max,
                                String name)
 {
  if (num < 0 || num > max)
  {
   throw
    new IllegalArgumentException(name + " : " + num);
  }
 }

}
```

Constructor Overloading

- Note that you can overload constructors

```
// in PhoneNumber class; exercises for the student

public PhoneNumber(String areaCode,
                   String exchangeCode,
                   String stationCode)
{

}

public PhoneNumber(String phoneNum)
{
  // assume phoneNum looks like (ABC) XYZ-IJKL
}
```

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()**?

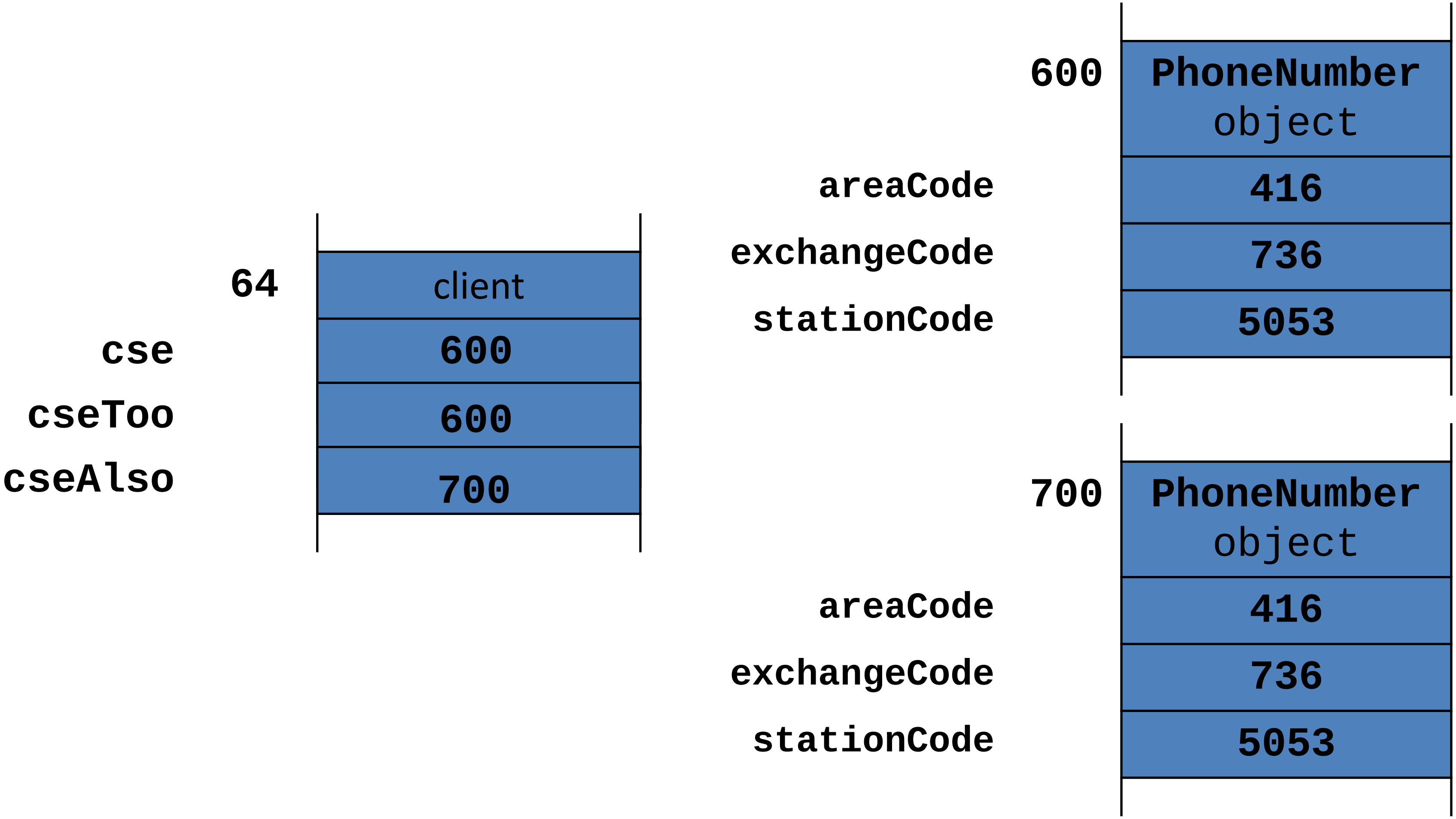
```
// PhoneNumber client

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], [AJ p 450-455]



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()`

- Implements an identity check
  - An instance is equal only to itself
  - **`x.equals(y)`** is true if and only if **`x`** and **`y`** are references to the same object
- 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()`**."

    – 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

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 **`==`**

PhoneNumber.equals(): Part 1

```
// inside class PhoneNumber

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





  return eq;
}
```

An Instance is Never Equal to **`null`**

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

`PhoneNumber.equals()`: Part 2

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




  return eq;
}
```

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.

PhoneNumber.equals(): Part 3

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




  return eq;
}
```

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

PhoneNumber.equals(): Part 4

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

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