Documenting Code

Javadoc

- documenting code was not a new idea when Java was invented
  - however, Java was the first major language to embed documentation in the code and extract the documentation into readable electronic APIs

- the tool that generates API documents from comments embedded in the code is called Javadoc

Javadoc

- Javadoc processes *doc comments* that immediately precede a class, attribute, constructor or method declaration
  - doc comments delimited by `/**` and `*/`
  - doc comment written in HTML and made up of two parts
    1. a description
       - first sentence of description gets copied to the summary section
       - only one description block; can use **`<p>`** to create separate paragraphs
    2. block tags
       - begin with `@` (**`@param`**, **`@return`**, **`@exception`**)
       - **`@pre.`** is non-standard (custom tag used in CSE1030)

Javadoc Guidelines

- http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html
- [notes 1.5.1, 1.5.2]

- precede every exported class, interface, constructor, method, and attribute with a doc comment
- for methods the doc comment should describe the contract between the method and the client
  - preconditions ([notes 1.4], [JBA 2.3.3])
  - postconditions ([notes 1.4], [JBA 2.3.3])

Javadoc Examples

- short in-class demo here
- see any lab exercise

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

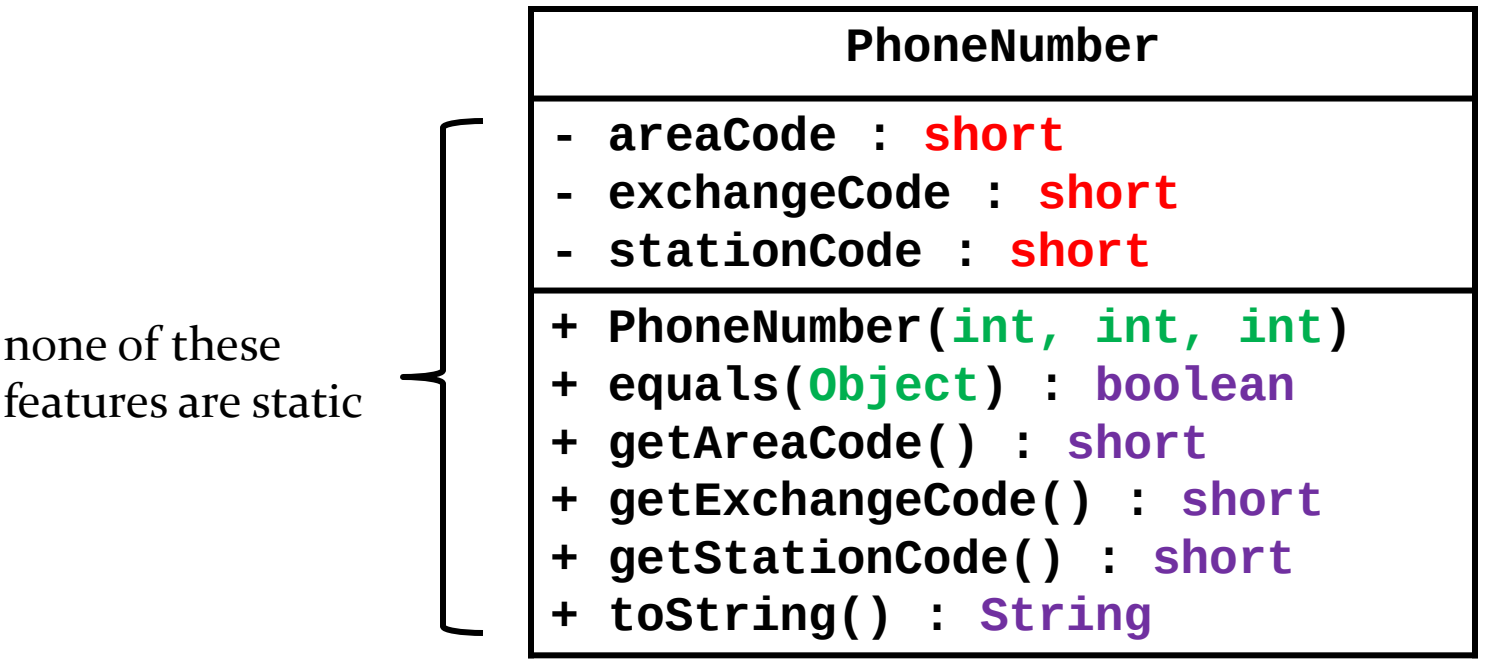
- North American Numbering Plan is the standard used in Canada and the USA for telephone numbers
- telephone numbers look like

416-736-2100

area code	exchange code	station code
416	736	2100

Designing a Simple Immutable Class

- **`PhoneNumber`** API



```
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

```java
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

```
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

```
/**
 * 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)

```
public PhoneNumber() {

}
```

the *default* constructor (has no parameters)

```
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