Singleton vs utility class

- at first glance, the singleton pattern does not seem to offer any advantages to using a utility class
  - i.e., a utility class with non-final static fields looks a lot like a single object with non-static fields
- there is a fundamental difference between a singleton and a utility class:
  - a singleton represents an object whereas a utility is a class

Singleton vs utility class

- suppose that you want your singleton/utility class to implement an interface
  - up to and including Java 7, a utility class could not implement an interface
  - a singleton can freely implement interfaces

- Java 8 now allows static methods in interfaces
  - a utility class can now implement an interface that has all static methods
    - but a utility class still cannot implement an interface having non-static methods (such as `Iterable`)

Singleton vs utility class

- suppose that you decide later on that you need multiple instances rather than a singleton/utility class
  - a utility class cannot be used to create objects of the utility class type
  - a singleton can be converted to a non-singleton

Singleton vs utility class

- can you create a method that has a parameter whose type is a utility class?
  - no, a parameter is a variable that stores a reference to an object and there are no utility class objects
- can you create a method that has a parameter whose type is a singleton?
  - yes, a parameter is a variable that stores a reference to an object and there is one singleton object

Immutable classes

Immutable Classes

- `String` is an example of an immutable class
- 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
  - other Java examples: `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

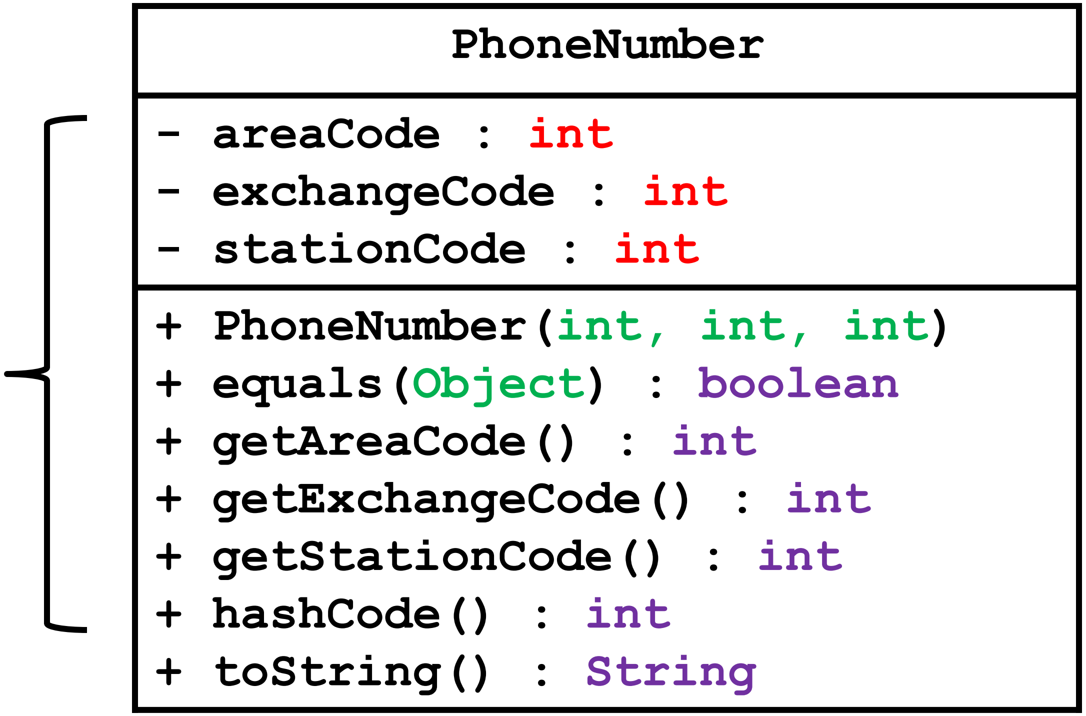
area code

exchange code

station code

Designing a Simple Immutable Class

- **`PhoneNumber`** API



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

none of these
features are static;
there are no
mutator methods

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

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

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

```
public int getAreaCode() {
  return this.areaCode;
}

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

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

```java
@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 (this.areaCode != other.areaCode ||
      this.exchangeCode != other.exchangeCode ||
      this.stationCode != other.stationCode) {
    return false;
  }
  return true;
}
```

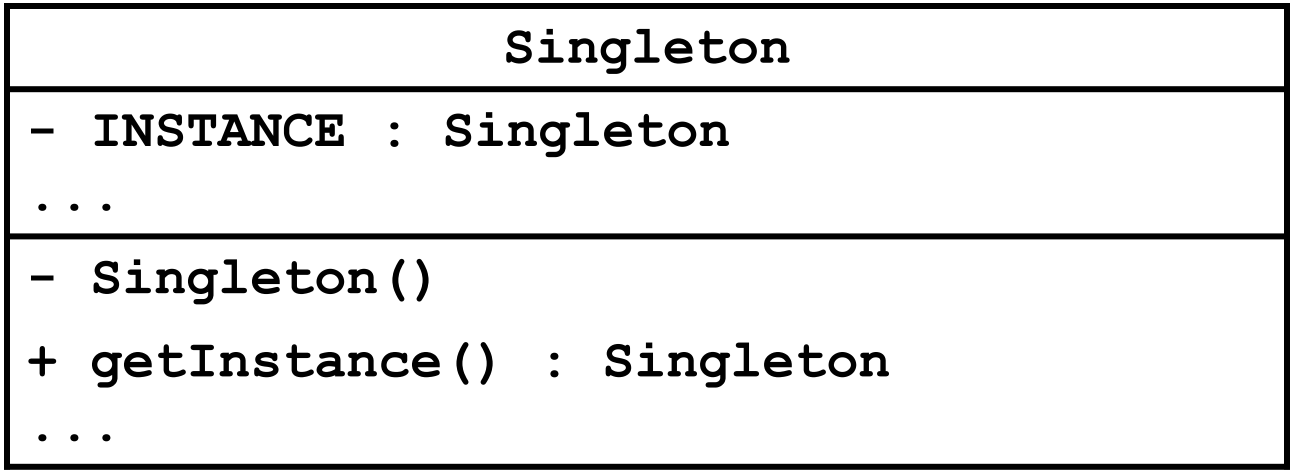
Mixing Static and Non-static

Multiton

Goals for Today

- Multiton
- review maps
- static factory methods

Singleton UML Class Diagram



```
Singleton
- INSTANCE : Singleton
...
- Singleton()
+ getInstance() : Singleton
...
```

One Instance per State

- the Java language specification guarantees that identical **`String`** literals are not duplicated

```
// client code somewhere

String s1 = "xyz";
String s2 = "xyz";

// how many String instances are there?
System.out.println("same object? " + (s1 == s2) );
```

  - prints: **`same object? true`**
- the compiler ensures that identical **`String`** literals all refer to the same object
  - a single instance per unique state

[notes 4.5]

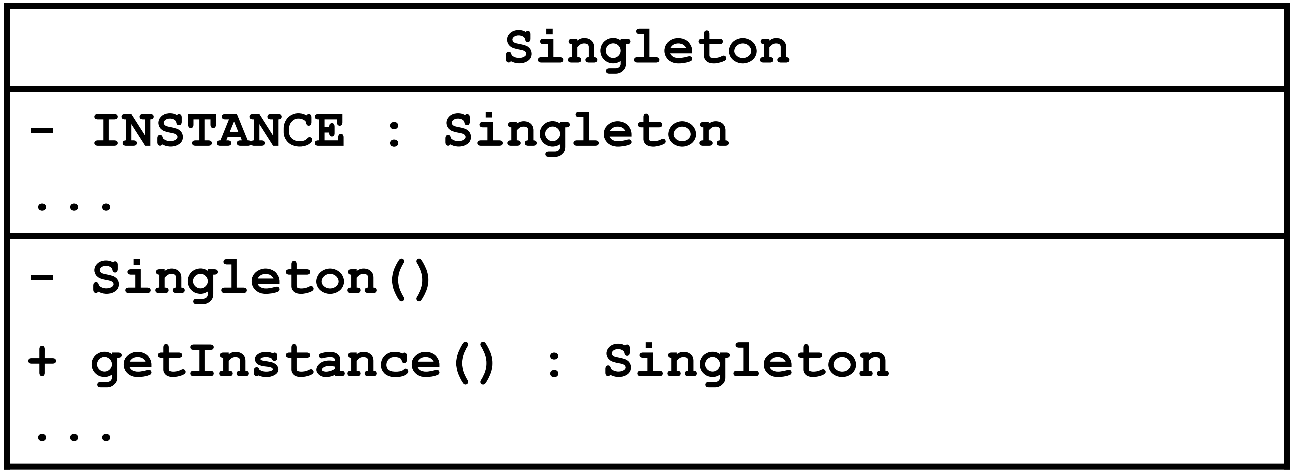
Multiton

- a *singleton* class manages a single instance of the class
- a *multiton* class manages multiple instances of the class

- what do you need to manage multiple instances?
  - a collection of some sort

- how does the client request an instance with a particular state?
  - it needs to pass the desired state as arguments to a method

Singleton vs Multiton UML Diagram



Singleton

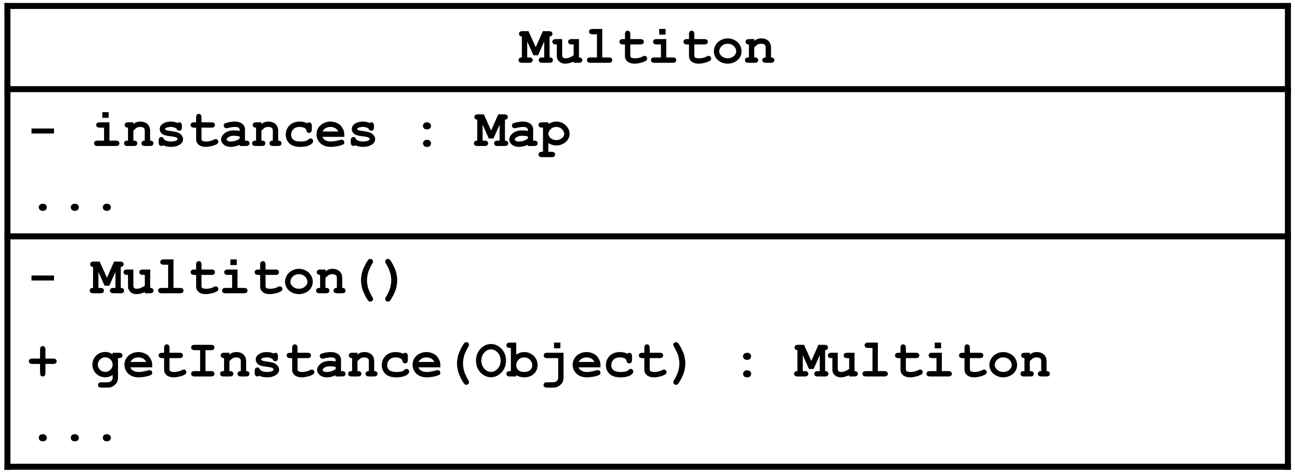
- INSTANCE : Singleton

...

- Singleton()

+ getInstance() : Singleton

...



Multiton

- instances : Map

...

- Multiton()

+ getInstance(Object) : Multiton

...

Singleton vs Multiton

- Singleton
  - one instance

```
private static final Santa INSTANCE = new Santa();
```

  - zero-parameter accessor

```
public static Santa getInstance()
```

Singleton vs Multiton

- Multiton
  - multiple instances (each with unique state)

```
private static final Map<String, PhoneNumber>
  instances = new TreeMap<String, PhoneNumber>();
```

  - accessor needs to provide state information

```
public static PhoneNumber getInstance(int areaCode,
                                     int exchangeCode,
                                     int stationCode)
```

Map

- a map stores key-value pairs

Map<String, PhoneNumber>

key type value type

- values are put into the map using the key

```
// client code somewhere
Map<String, PhoneNumber> m =
                      new TreeMap<String, PhoneNumber>;

PhoneNumber ago = new PhoneNumber(416, 979, 6648);
String key = "4169796648"

m.put(key, ago);
```

[AJ 16.2]

Mutable Keys

- from http://docs.oracle.com/javase/7/docs/api/java/util/Map.html
  - Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map.

```
public class MutableKey
{
  public static void main(String[] args)
  {
    Map<Date, String> m = new TreeMap<Date, String>();
    Date d1 = new Date(100, 0, 1);
    Date d2 = new Date(100, 0, 2);
    Date d3 = new Date(100, 0, 3);
    m.put(d1, "Jan 1, 2000");
    m.put(d2, "Jan 2, 2000");
    m.put(d3, "Jan 3, 2000");
    d2.setYear(101);           // mutator
    System.out.println("d1 " + m.get(d1));  // d1 Jan 1, 2000
    System.out.println("d2 " + m.get(d2));  // d2 Jan 2, 2000
    System.out.println("d3 " + m.get(d3));  // d3 null
  }
}
```

don't mutate keys;
bad things will happen

change TreeMap to HashMap and see what happens

Making **PhoneNumber** a Multiton

1. multiple instances (each with unique state)

```
private static final Map<String, PhoneNumber>
    instances = new TreeMap<String, PhoneNumber>();
```

2. accessor needs to provide state information

```
public static PhoneNumber getInstance(int areaCode,
                                      int exchangeCode,
                                      int stationCode)
```

- **getInstance()** will get an instance from **instances** if the instance is in the map; otherwise, it will create the new instance and put it in the map

Making **`PhoneNumber`** a Multiton

3. require private constructors
   - to prevent clients from creating instances on their own
     - clients should use **`getInstance()`**

4. require immutability of **`PhoneNumber`**s
   - to prevent clients from modifying state, thus making the keys inconsistent with the **`PhoneNumber`**s stored in the map
   - recall the recipe for immutability…

```
public class PhoneNumber
{
  private static final Map<String, PhoneNumber> instances =
                         new TreeMap<String, PhoneNumber>();

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

  private PhoneNumber(int areaCode,
                      int exchangeCode,
                      int stationCode)
  { // validate and set the
    // areaCode, exchangeCode, and stationCode
  }
```

why is validation not needed?

```
public static PhoneNumber getInstance(int areaCode,
                                      int exchangeCode,
                                      int stationCode)
{
  String key = "" + areaCode + exchangeCode + stationCode;
  PhoneNumber n = PhoneNumber.instances.get(key);
  if (n == null)
  {
    n = new PhoneNumber(areaCode, exchangeCode, stationCode);
    PhoneNumber.instances.put(key, n);
  }
  return n;
}
// remainder of PhoneNumber class ...
```

```java
public class PhoneNumberClient {

  public static void main(String[] args)
  {
    PhoneNumber x = PhoneNumber.getInstance(416, 736, 2100);
    PhoneNumber y = PhoneNumber.getInstance(416, 736, 2100);
    PhoneNumber z = PhoneNumber.getInstance(905, 867, 5309);

    System.out.println("x equals y: " + x.equals(y) +
                      " and x == y: " + (x == y));

    System.out.println("x equals z: " + x.equals(z) +
                      " and x == z: " + (x == z));
  }
}
```

```
x equals y: true and x == y: true
x equals z: false and x == z: false
```

A Singleton Puzzle: What is Printed?

```
public class Elvis {
  public static final Elvis INSTANCE = new Elvis();
  private final int beltSize;
  private static final int CURRENT_YEAR =
    Calendar.getInstance().get(Calendar.YEAR);

  private Elvis() { this.beltSize = CURRENT_YEAR - 1930; }

  public int getBeltSize() { return this.beltSize; }

  public static void main(String[] args) {
    System.out.println("Elvis has a belt size of " +
                       INSTANCE.getBeltSize());
  }
}
```

from Java Puzzlers by Joshua Bloch and Neal Gafter