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 points by their distance from the origin
- 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

- suppose that we want to compare points by their distance from the origin

Point2 compareTo

```
public class Point2 implements Comparable<Point2> {
  // fields, constructors, methods...
```

```
  @Override
  public int compareTo(Point2 other) {
    double thisDist = Math.hypot(this.x, this.y);
    double otherDist = Math.hypot(other.x, other.y);
    if (thisDist > otherDist) {
      return 1;
    }
    else if (thisDist < otherDist) {
      return -1;
    }
    return 0;
  }
```

`Point2 compareTo`

- don't forget what you learned in previous courses
  - you should delegate work to well-tested components where possible
- for distances, we need to compare two **`double`** values
  - **`java.lang.Double`** has methods that do exactly this

Point2 compareTo

```
public class Point2 implements Comparable<Point2> {
  // fields, constructors, methods...

  @Override
  public int compareTo(Point2 other) {
    double thisDist = Math.hypot(this.x, this.y);
    double otherDist = Math.hypot(other.x, other.y);
    return Double.compare(thisDist, otherDist);
  }
```

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
- is **`Point2 compareTo`** consistent with equals?

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 Oracle's Java Tutorial
public class Bicycle {
  // some other fields here...
  private static int numberOfBicycles = 0;

  public Bicycle() {
    // set some non-static fields here...
    Bicycle.numberOfBicycles++;
  }

  public static int getNumberOfBicyclesCreated() {
    return Bicycle.numberOfBicycles;
  }
}
```

note: not `this.numberOfBicycles++`

[notes 4.3]

- why does **`numberOfBicycles`** have to be **`static`**?
  - because we really want one common value for all **`Bicycle`** instances

- what would happen if we made **`numberOfBicycles`** non-**`static`**?
  - every **`Bicycle`** would think that there was a different number of **`Bicycle`** instances

- 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;
  }
}
```

- is it useful to add the following to **`Point2`**?

```
public static final Point2 ORIGIN = new Point2(0.0, 0.0);
```

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;

  // ...
```


[notes 4.3.2]

- 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++;
  }
}
```

but you would need an implementation of this class

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

Static Methods

- a **`static`** method can use only **`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 fields

non-static method can use non-static fields

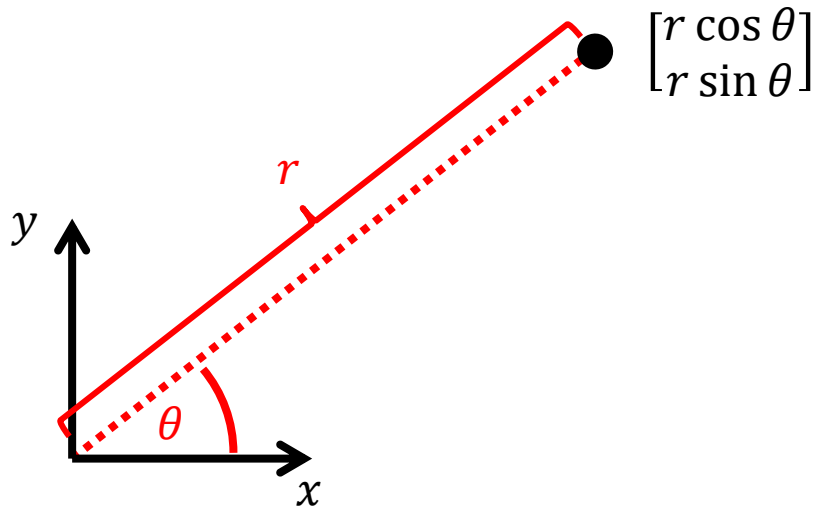
and static fields

Static factory methods

- a common use of static methods in non-utility classes is to create a *static factory method*
  - a static factory method is a static method that returns an instance of the class
  - called a factory method because it makes an object and returns a reference to the object
- you can use a static factory method to create methods that behave like constructors
  - they create and return a reference to a new instance
  - unlike a constructor, the method has a name

Static factory methods

- recall our point class
  - suppose that you want to provide a constructor that constructs a point given the polar form of the point



```
public class Point2 {

  private double x;
  private double y;

  public Point2(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public Point2(double r, double theta) {
    this(r * Math.cos(theta), r * Math.sin(theta));
  }
```

Illegal overload; both constructors have the same signature.

Static factory methods

- we can eliminate the problem by replacing the second constructor with a static factory method

```
public class Point2 {

  private double x;
  private double y;

  public Point2(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public static Point2 polar(double r, double theta) {
    double x = r * Math.cos(theta);
    double y = r * Math.sin(theta);
    return new Point2(x, y);
  }
```

Static Factory Methods

- many examples in Java API
  - **`java.lang.Integer`**

    `public static Integer valueOf(int i)`
    - Returns a **`Integer`** instance representing the specified **`int`** value.
  - **`java.util.Arrays`**

    `public static int[] copyOf(int[] original, int newLength)`
    - Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
  - **`java.lang.String`**

    `public static String format(String format, Object... args)`
    - Returns a formatted string using the specified format string and arguments.

UML class diagrams

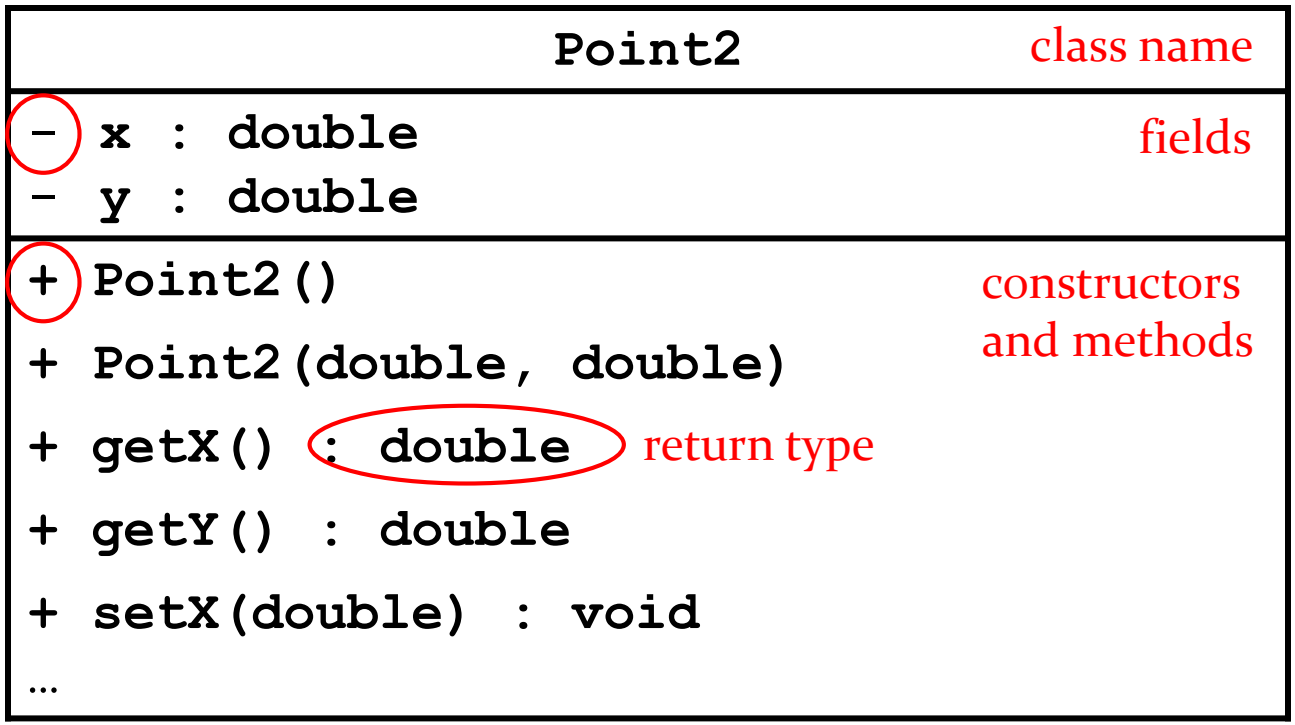
UML class diagram

- Unified Modeling Language
  - software engineering language used to visualize the design of a software system in a standardized way

- a UML class diagram describes the structure of a class and how that class is related to other classes

UML class diagram

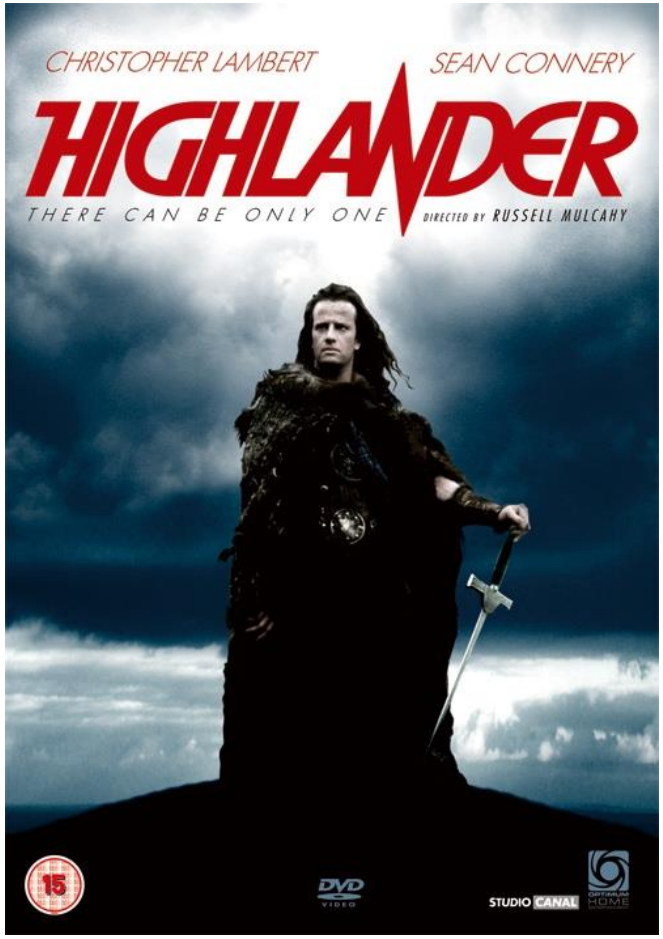
- class diagram for **`Point2`**



Singleton pattern

Singleton Pattern

- “There can be only one.”



- Connor MacLeod, Highlander

Singleton Pattern

- a singleton is a class that is instantiated exactly once
- singleton is a well-known design pattern that can be used when you need to:
  1. ensure that there is one, and only one*, instance of a class, and
  2. provide a global point of access to the instance
     - any client that imports the package containing the singleton class can access the instance

[notes 4.4]

*or possibly zero

One and Only One

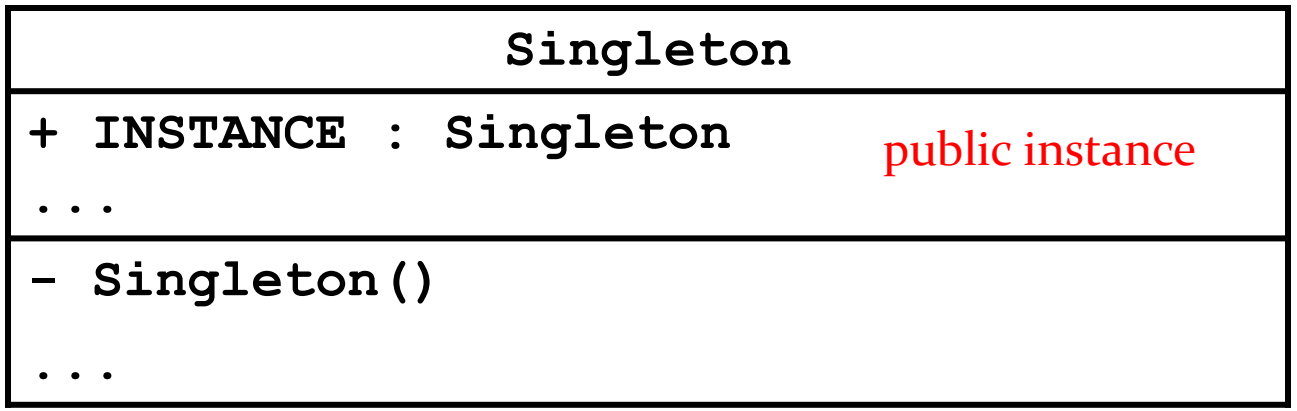
- how do you enforce this?
  - need to prevent clients from creating instances of the singleton class
    - **`private`** constructors
  - the singleton class should create the one instance of itself
    - note that the singleton class is allowed to call its own **`private`** constructors
    - need a **`static`** attribute to hold the instance

A Silly Example: Version 1

uses a public field that all clients can access

```
package xmas;

public class Santa
{
  // whatever fields you want for santa...

  public static final Santa INSTANCE = new Santa();

  private Santa()
  { // initialize non-static fields here... }

}
```

UML Class Diagram (Version 1)



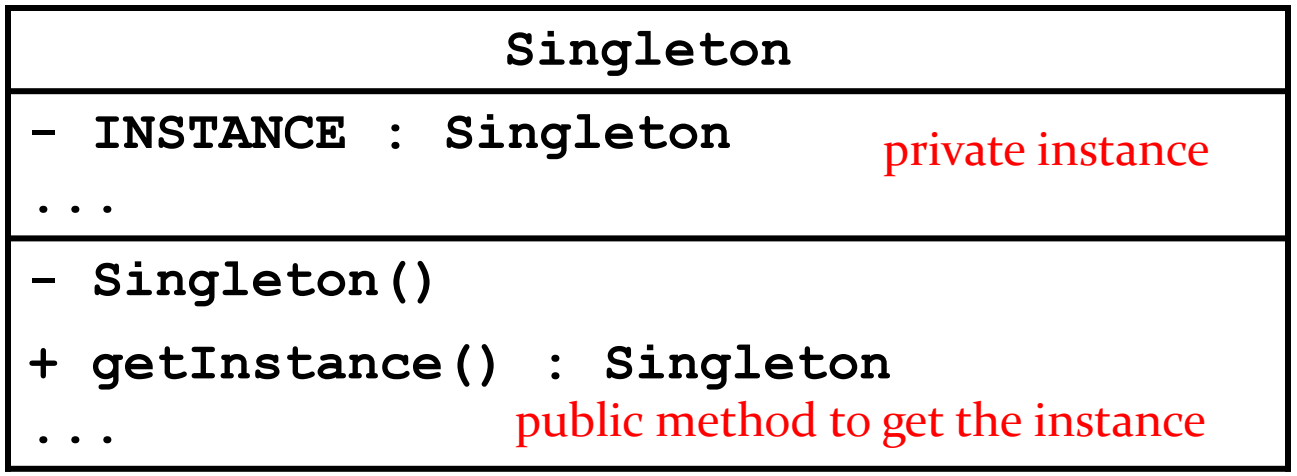
```
import xmas;

// client code in a method somewhere ...
public void gimme()
{
  Santa.INSTANCE.givePresent();
}
```

A Silly Example: Version 2

```
package xmas;

public class Santa
{
  // whatever fields you want for santa...

  private static final Santa INSTANCE = new Santa();

  private Santa()
  { // initialize fields here... }

}
```

uses a private field; how do clients access the field?

UML Class Diagram (Version 2)



Global Access

- how do clients access the singleton instance?
  - by using a static method

- note that clients only need to import the package containing the singleton class to get access to the singleton instance
  - any client method can use the singleton instance without mentioning the singleton in the parameter list

A Silly Example (cont)

```
package xmas;

public class Santa {
   private int numPresents;
  private static final Santa INSTANCE = new Santa();

  private Santa()
  { // initialize fields here... }

   public static Santa getInstance()
   { return Santa.INSTANCE; }

   public Present givePresent() {
      Present p = new Present();
      this.numPresents--;
      return p;
   }
}
```

uses a private field; how do clients access the field?

clients use a public static factory method

```
import xmas;

// client code in a method somewhere ...
public void gimme()
{
  Santa.getInstance().givePresent();
}
```

Applications

- singletons should be uncommon
- typically used to represent a system component that is intrinsically unique
  - window manager
  - file system
  - logging system

Logging

- when developing a software program it is often useful to log information about the runtime state of your program
  - similar to flight data recorder in an airplane
  - a good log can help you find out what went wrong in your program
- problem: your program may have many classes, each of which needs to know where the single logging object is
  - global point of access to a single object == singleton
- Java logging API is more sophisticated than this
  - but it still uses a singleton to manage logging
    - `java.util.logging.LogManager`

Multiton

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 3.5]

North American Phone Numbers

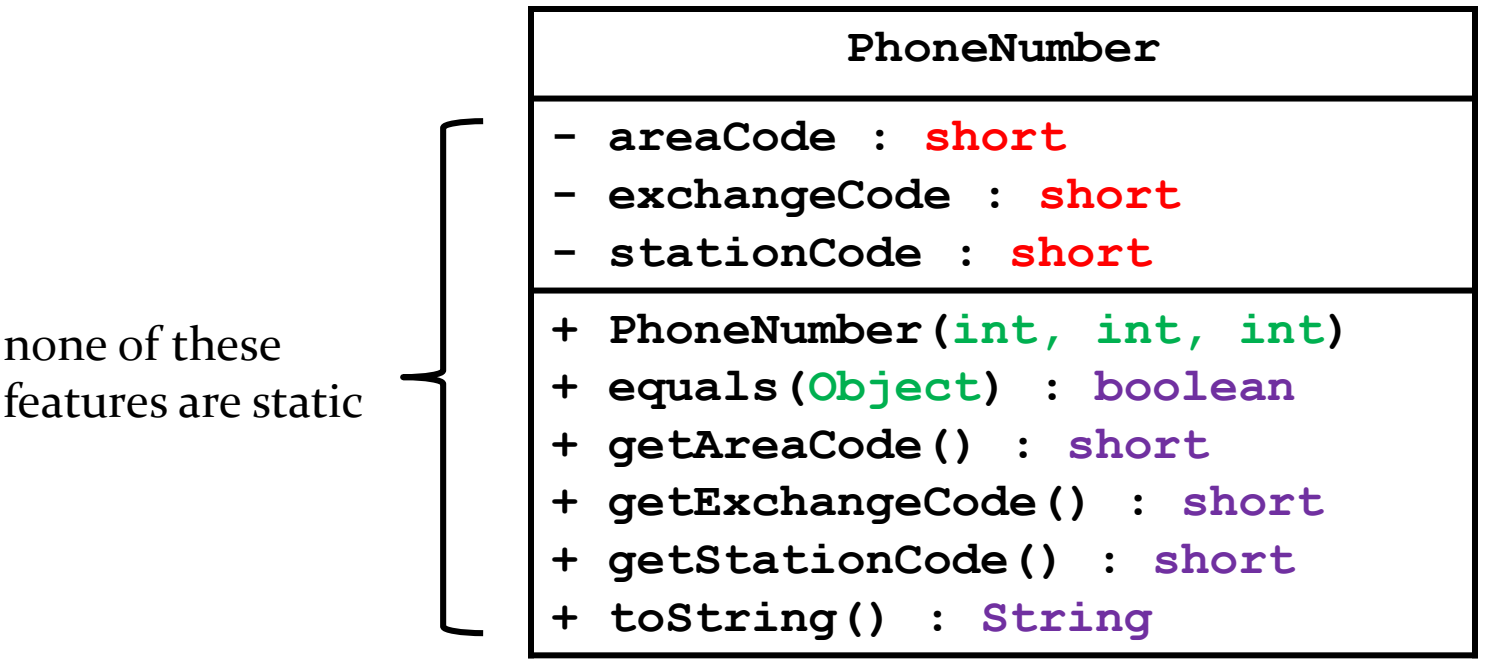
- North American Numbering Plan is the standard used in Canada and the USA for telephone numbers
- telephone numbers look like

416-736-2100

area code	exchange code	station code
416	736	2100

UML Class Diagram

- **`PhoneNumber`** API



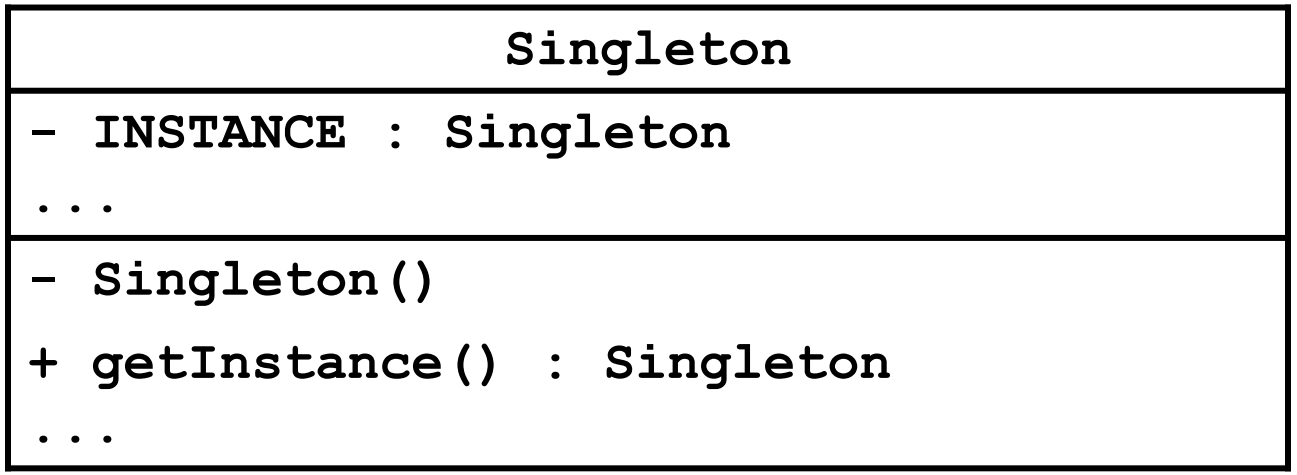
Multiton

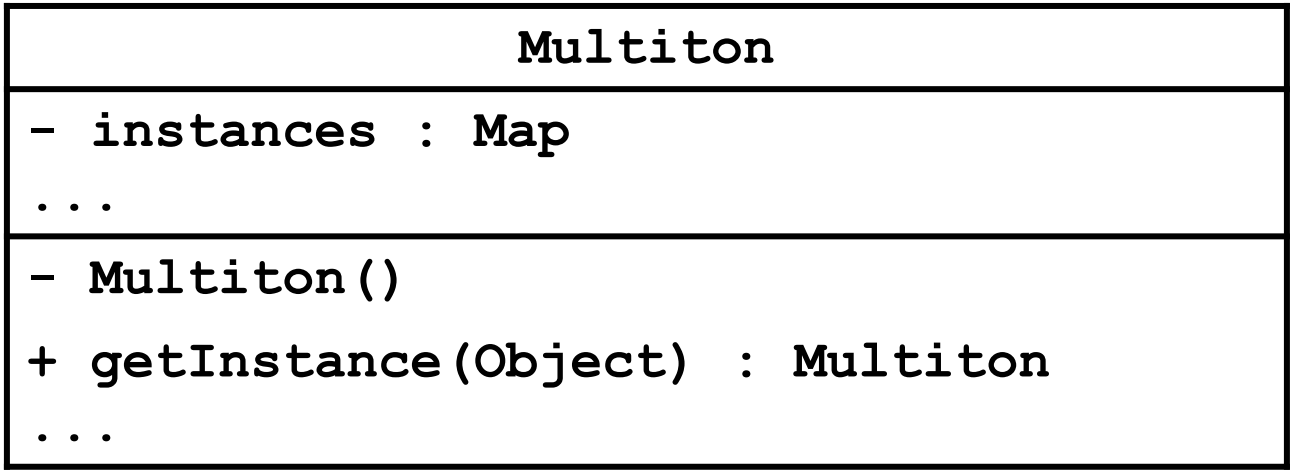
- a *singleton* class manages a single instance of the class
- a *multiton* class manages multiple instances of the class

- 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 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)
```

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 implements Comparable<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)
  { // initialize this.areaCode,
                    this.exchangeCode, and this.stationCode }
```

```
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 ...
```

```
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
```