Mixing Static and Non-Static Features

Based on slides by Prof. Burton Ma

`static` Attributes

- An attribute that is **`static`** is a per-class member
  - Only one copy of the attribute, and the attribute is associated with the class
    - Every object created from a class declaring a static attribute shares the same copy of the attribute
- Static attributes are used when you really want only one common instance of the attribute for the class

Example

- A common textbook example of a static attribute is a counter that counts the number

```
// adapted from Sun's Java Tutorial
public class Bicycle {
  // some attributes here...
  private static int numberOfBicycles = 0;

  public Bicycle() {
    // set some attributes here...
    Bicycle.numberOfBicycles++;
  }

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

note:
not this.numberOfBicycles++

[notes 3.2]

- 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;
  }
}
```

Mixing Static and Non-static Attributes

- A class can declare static (per class) and non-static (per instance) attributes
- A common textbook example is giving each instance a unique serial number
  - The serial number belongs to the instance

```
public class Bicycle {
  // some attributes here...
  private static int numberOfBicycles = 0;

  private int serialNumber;

  // ...
```

- 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 attributes
  - E.g. using the number of instances created as a

```
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++;
  }
}
```

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
- A **`static`** method can only use **`static`** attributes 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 attributes
- Non-static methods can use all of the attributes 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 attributes

non-static method
can use
non-static attributes

and static attributes

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 **no more than 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 3.4]

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

```
public class Santa
{
  // whatever attributes you want for santa...

  private static final Santa INSTANCE = new Santa();

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

  …

}
```

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)

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

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

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

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

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

Lazy Instantiation

- Notice that the previous singleton implementation always creates the singleton instance whenever the class is loaded
  - If no client uses the instance then it was created needlessly
- It is possible to delay creation of the singleton instance until it is needed by using lazy instantiation

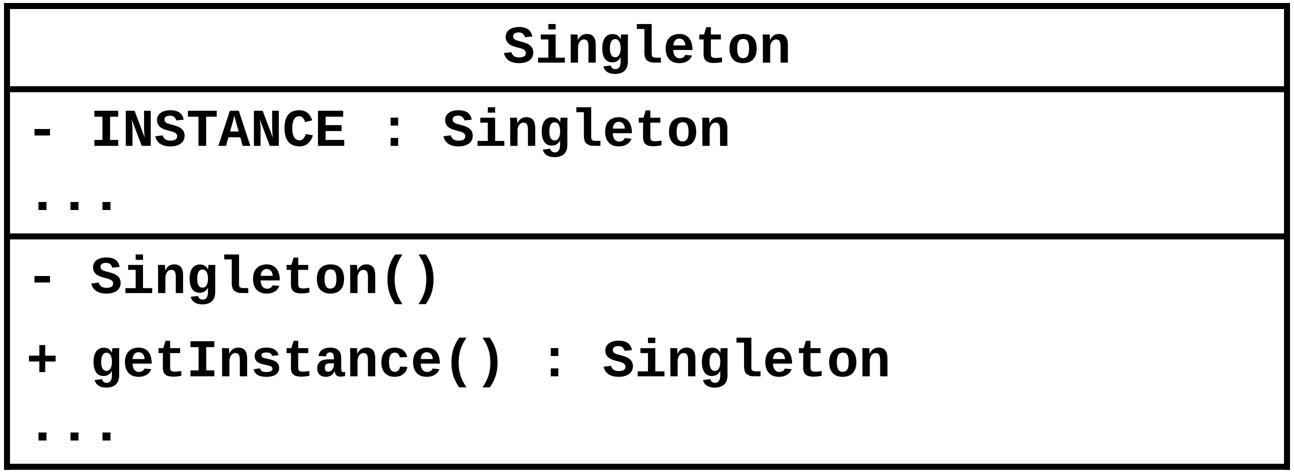
Lazy Instantiation as per Notes

```
public class Santa {
  private static Santa INSTANCE = null;

  private Santa()
  { // ... }

  public static Santa getInstance()
  {
    if (Santa.INSTANCE == null) {
      Santa.INSTANCE = new Santa();
    }
    return Santa.INSTANCE;
  }
}
```

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 3.5]

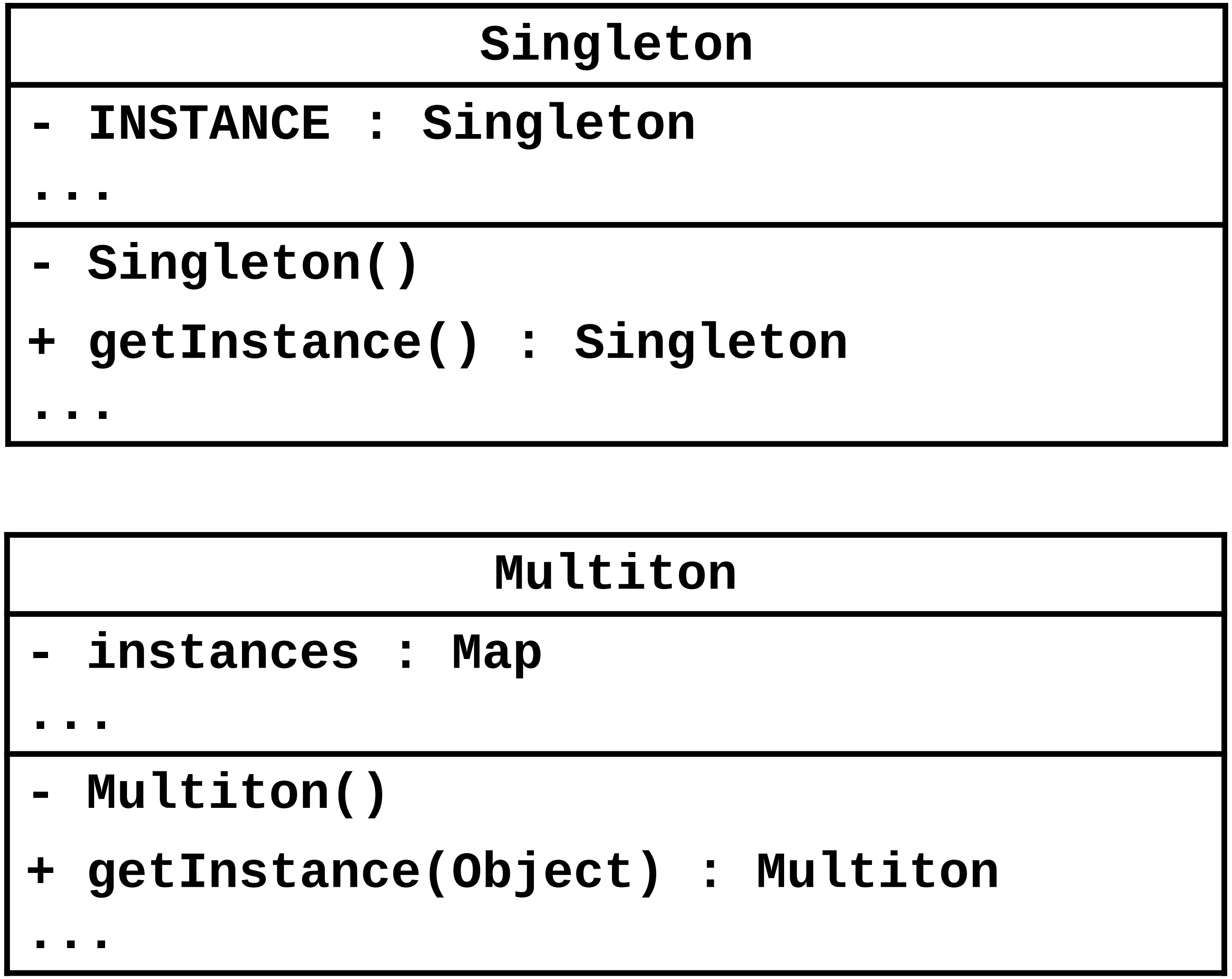
Multiton

- 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)
```

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)
  { // identical to previous versions }
```

```
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
```

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]

- Values can be retrieved from the map using only the key
  - If the key is not in the map the value returned is **`null`**

```
// 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);

PhoneNumber gallery = m.get(key);            // == ago
PhoneNumber art = m.get("4169796648");       // == ago

PhoneNumber pizza = m.get("4169671111");     // == null
```

- A map is not allowed to hold duplicate keys
  - If you re-use a key to insert a new object, the existing object corresponding to the key is removed and the new object inserted

```
// 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);                               // add ago
System.out.println(m);

m.put(key, new PhoneNumber(905, 760, 1911));   // replaces ago
System.out.println(m);
```

Prints

```
{4169796648=(416) 979-6648}
{4169796648=(905) 760-1911}
```

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");
    d3.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

Static Factory Method

- Notice that Singleton and Multiton use a static method to return an instance of a class
- A static method that returns an instance of a class is called a *static factory method*
  - Factory because, as far as the client is concerned, the method creates an instance
    - Similar to a constructor