

Mixing Static and Non-static

Singleton

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 3.4]

*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

```
package xmas;
```

uses a public field that
all clients can access

```
public class Santa
```

```
{
```

```
    // whatever fields you want for santa...
```

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

```
    private Santa()
```

```
    { // initialize attributes here... }
```

```
}
```

```
import xmas;

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

A Silly Example: Version 2

```
package xmas;
```

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

```
public class Santa
```

```
{
```

```
    // whatever fields 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)

```
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();
}
```



Enumerations

- ▶ an enumeration is a special data type that enables for a variable to be a set of predefined constants
- ▶ the variable must be equal to one of the values that have been predefined for it
 - ▶ e.g., compass directions
 - ▶ NORTH, SOUTH, EAST, and WEST
 - ▶ days of the week
 - ▶ MONDAY, TUESDAY, WEDNESDAY, etc.
 - ▶ playing card suits
 - ▶ CLUBS, DIAMONDS, HEARTS, SPADES
- ▶ useful when you have a fixed set of constants

A Silly Example: Version 3

```
package xmas;
```

singleton as an
enumeration

```
public enum Santa
```

```
{
```

```
    // whatever fields you want for santa...
```

```
    INSTANCE;
```

will call the private
default constructor

```
private Santa()
```

```
{ // initialize attributes here... }
```

```
}
```

same usage as public
field (Version 1)

```
import xmas;

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

Singleton as an enumeration

- ▶ considered the preferred approach for implementing a singleton
 - ▶ for reasons beyond the scope of CSE1030

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`

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
 - ▶ only works for version 2

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;
    }
}
```

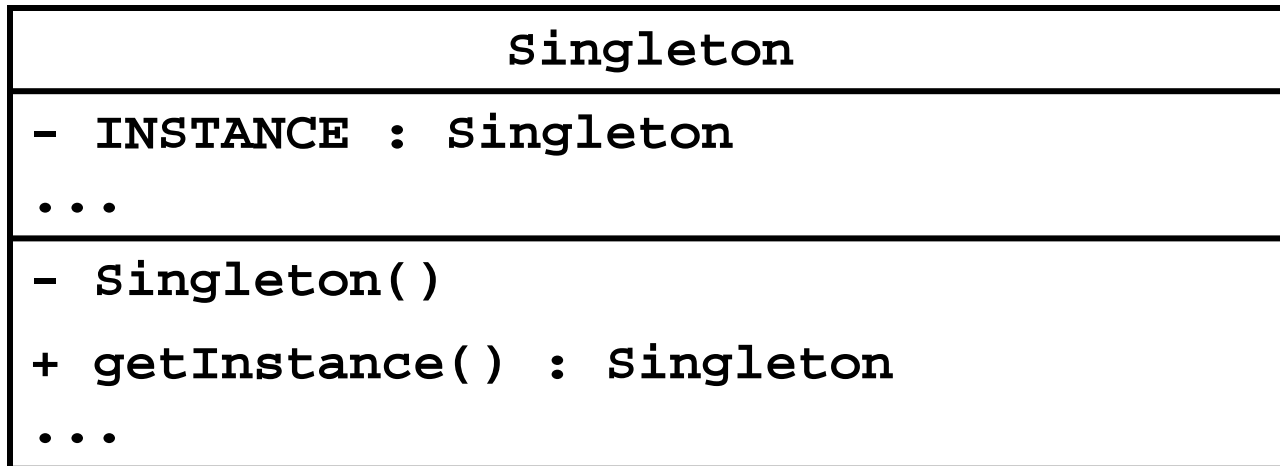
Mixing Static and Non-static

Multiton

Goals for Today

- ▶ Multiton
- ▶ review maps
- ▶ static factory methods

Singleton UML Class Diagram



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]

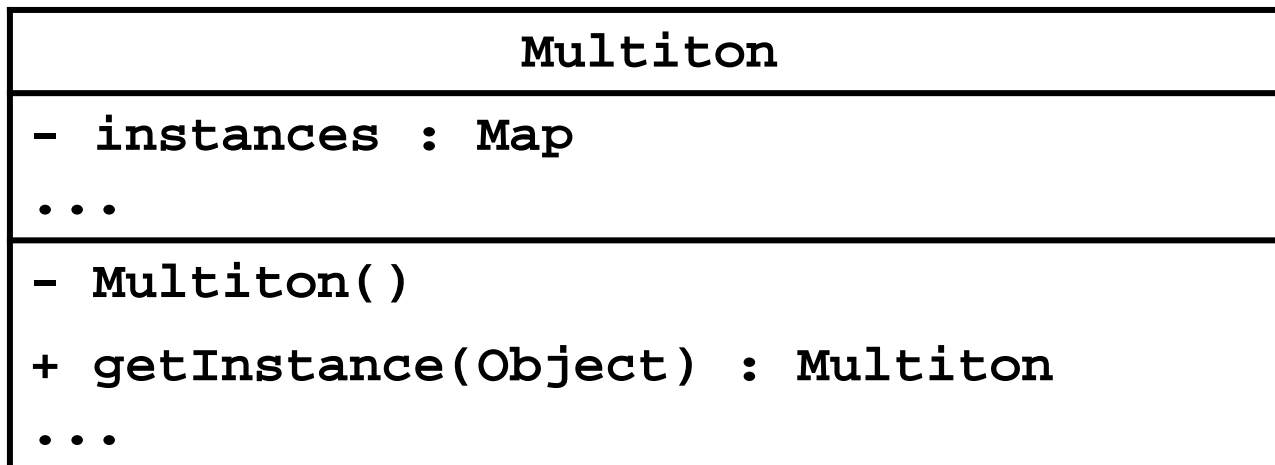
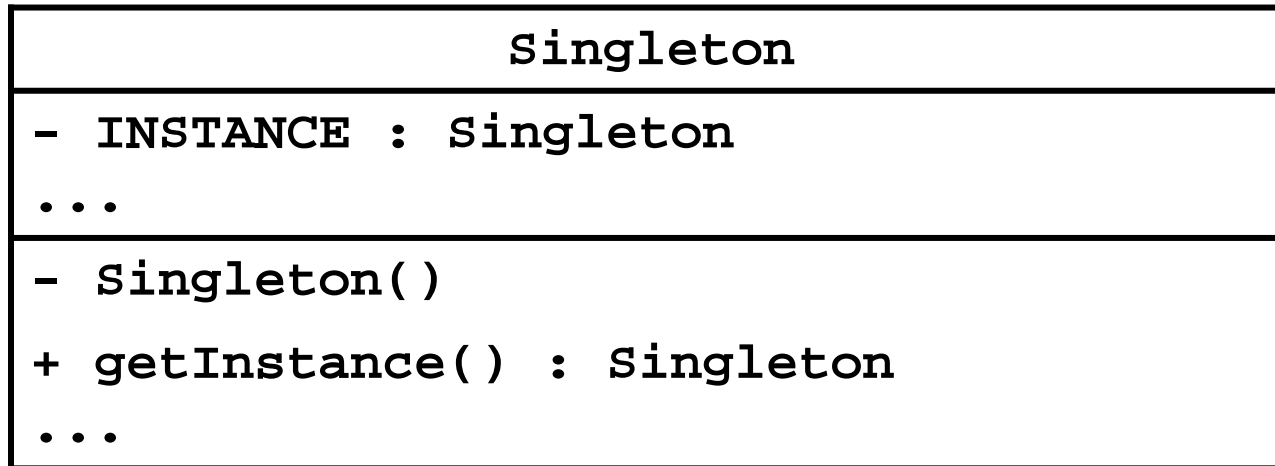
Multiton

- ▶ 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 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);
```

[A] 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");
        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 `PhoneNumbers`
 - ▶ to prevent clients from modifying state, thus making the keys inconsistent with the `PhoneNumbers` 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 ...
```

why is validation not needed?

```
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
```

Bonus Content

- ▶ 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

Static Factory Methods

- ▶ many examples

- ▶ `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 API Static Factory Methods

- ▶ `java.lang.String`

```
public static String format(String format, Object... args)
```

- ▶ Returns a formatted string using the specified format string and arguments.

- ▶ `cse1030.Complex`

```
public static Complex valueOf(String s)
```

- ▶ Returns a complex number holding the value represented by the given string.

-
- ▶ you can give meaningful names to static factory methods (unlike constructors)

```
public class Person {  
    private String name;  
    private int age;  
    private int weight;  
  
    public Person(String name, int age, int weight) { // ... }  
  
    public Person(String name, int age) { // ... }  
  
    public Person(String name, int weight) { // ... }  
    // ...          illegal overload: same signature  
}
```

```
public class Person { // modified from PEx's
    // attributes ...

    public Person(String name, int age, int weight) { // ... }

    public static Person withAge(String name, int age) {
        return new Person(name, age, DEFAULT_WEIGHT);
    }

    public static Person withWeight(String name, int weight) {
        return new Person(name, DEFAULT_AGE, weight);
    }
}
```

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
