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]

- 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 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++;
  }
}
```

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`** 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 attributes

non-static method can use non-static attributes

and static attributes

Static factory methods

- a common use of static methods is to create a *static factory method*
  - a static factory method is a static method that returns an instance of the class
- you can use a static factory method to create methods that behave like constructors
  - they create and return a new instance
  - unlike a constructor, the method has a name

Static factory methods

- recall our complex number class
  - suppose that you want to provide a constructor that constructs a complex number given only the real part of the number
    - the imaginary part is zero

```
public class Complex {

  private double real;
  private double imag;

  public Complex(double real, double imag) {
    this.real = real;
    this.imag = imag;
  }
```

```
  public Complex(double real) {
    this(real, 0);
  }
```

Static factory methods

- suppose that you also want to provide a constructor that constructs a complex number given only the imaginary part of the number
  - the real part is zero

- if you try to add such a constructor you encounter a problem…

```
public class Complex {

  private double real;
  private double imag;

  public Complex(double real, double imag) {
    this.real = real;
    this.imag = imag;
  }
```

```
  public Complex(double real) {
    this(real, 0);
  }

  public Complex(double imag) {
    this(0, imag);
  }
```

Illegal overload; both constructors have the same signature.

Static factory methods

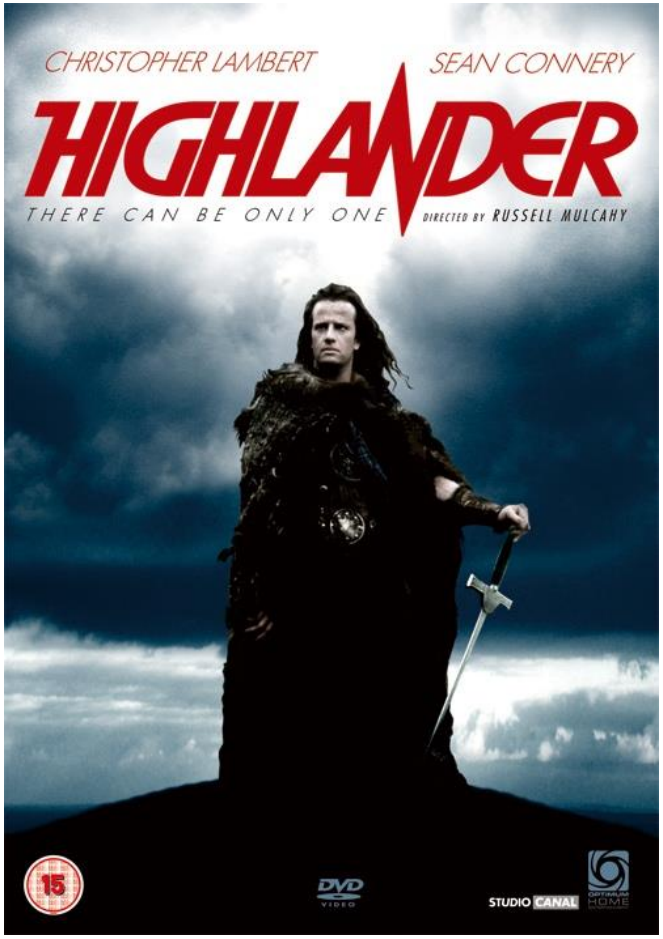
- we can eliminate the problem by replacing both constructors with named static factory methods

```
public class Complex {

  private double real;
  private double imag;

  public Complex(double real, double imag) {
    this.real = real;
    this.imag = imag;
  }

  public static Complex pureReal(double real) {
    return new Complex(real, 0);
  }

  public static Complex pureImag(double imag) {
    return new Complex(0, imag);
  }
```

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

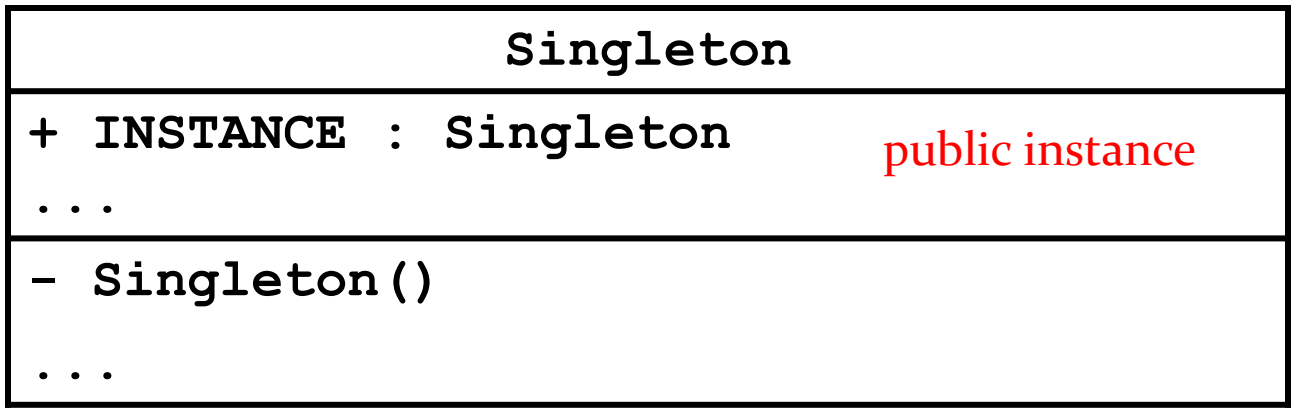
- how do you enforce this?
  - need to prevent clients from creating instances of the singleton class
    - **`private`** constructors
  - the singleton class should create the one instance of itself
    - note that the singleton class is allowed to call its own **`private`** constructors
    - need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



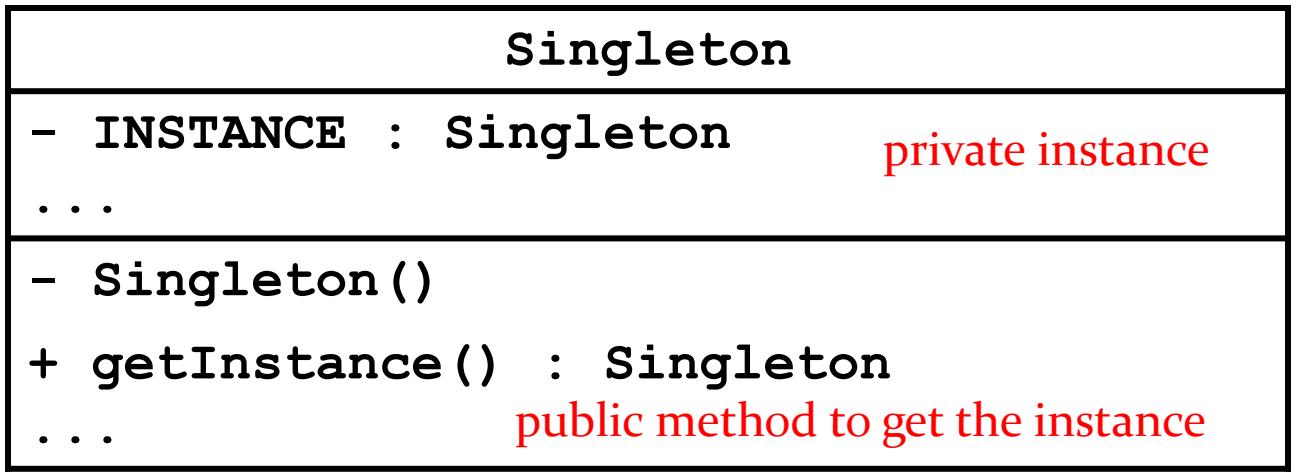
```
import xmas;

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

A Silly Example: Version 2

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

```
package xmas;

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

  private static final Santa INSTANCE = new Santa();

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

}
```

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`

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;
  }
}
```