CSC7203 : Advanced Object Oriented Development

J Paul Gibson, D311

Singleton Pattern

…/~gibson/Teaching/CSC7203/CSC7203-AdvancedOO-L2-Singleton.pdf


LEARNS SOFTWARE DESIGN PATTERNS
USES NOTHING BUT SINGLETON
memegenerator.net

The Singleton Design Pattern

See - http://sourcemaking.com/design_patterns/singleton

- Intent
  - Ensure a class has only one instance, and provide a global point of access to it.
  - Encapsulated “just-in-time initialization” or “initialization on first use”.
- Problem
  - Application needs one, and only one, instance of an object.
  - Additionally, lazy initialization and global access are necessary.

The Singleton Design Pattern

See - http://sourcemaking.com/design_patterns/singleton

Relation to other patterns

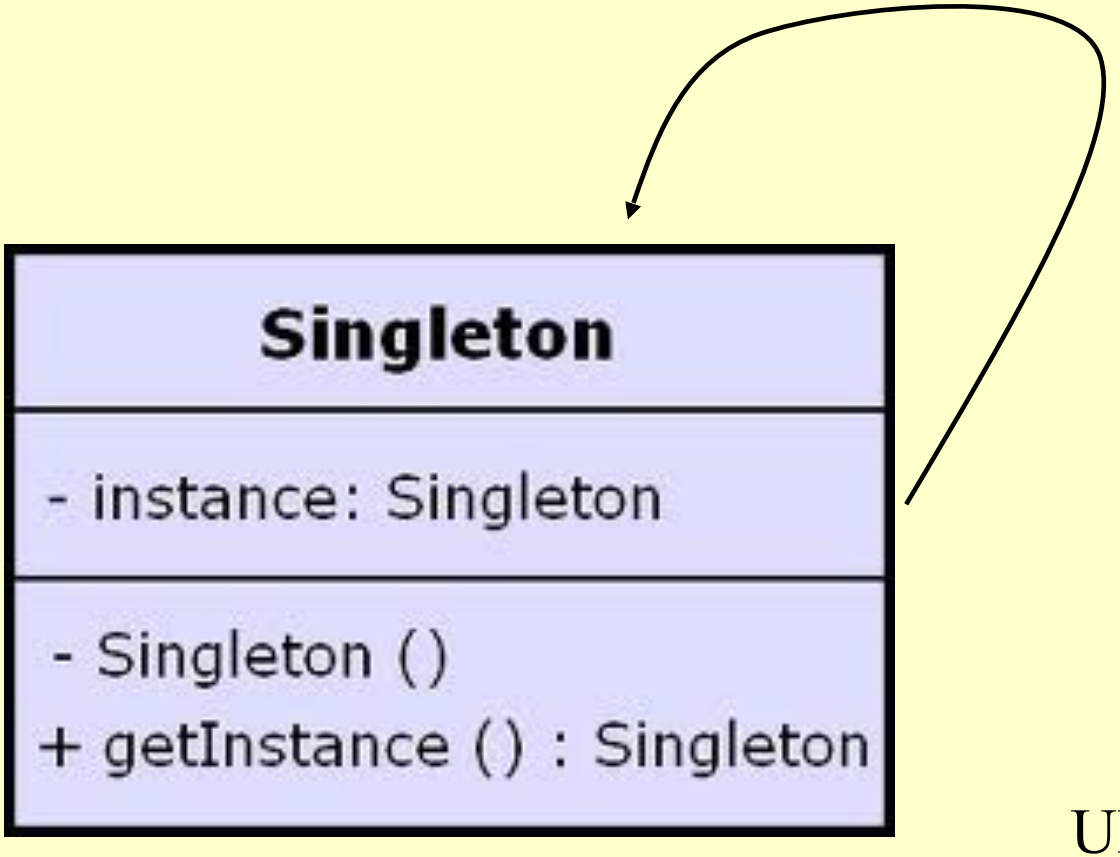
•Abstract Factory, Builder, and Prototype can use Singleton in their implementation.

•Facade objects are often Singletons because only one Facade object is required.

•The advantage of Singleton over global variables is that you are absolutely sure of the number of instances when you use Singleton.

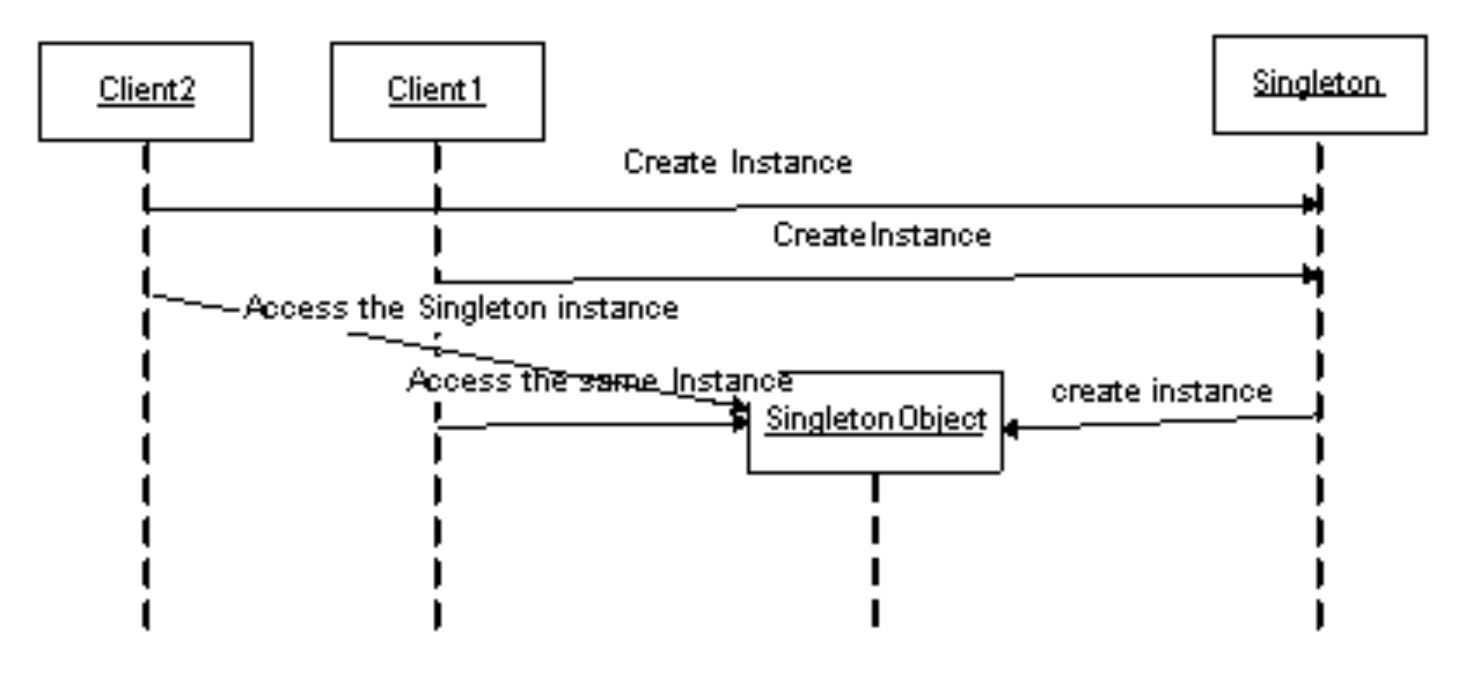
•The Singleton design pattern is one of the most inappropriately used patterns. Designers frequently use Singletons in a misguided attempt to replace global variables. A Singleton is, for intents and purposes, a global variable.

The Singleton Design Pattern



UML class diagram

The Singleton Design Pattern

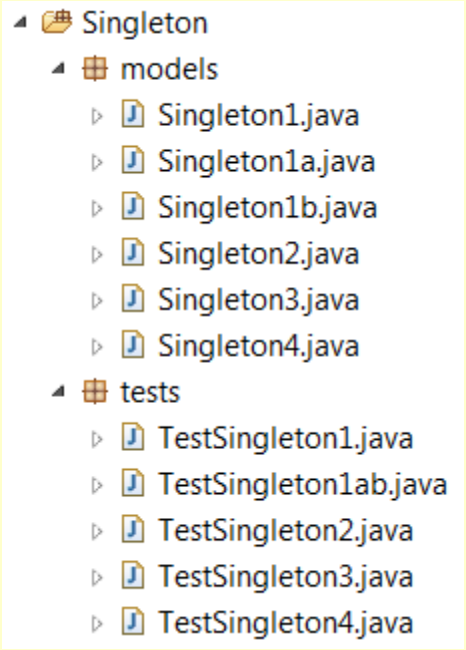


UML sequence diagram

The Singleton Design Pattern

Problem: Examine the 4 Singleton Java implementations in the folder Singleton (~gibson/Teaching/CSC7203/Code/Singleton.zip)

Question: what are the differences between the implementations: `Singleton1, Singleton2, Singleton3, Singleton4`; and which best corresponds to our requirements/design?



NOTE: `Singleton1a` and `Singleton1b` are for the next question on subclassing

The Singleton Design Pattern – example implementation (1)

```
public class Singleton1 {

protected static Singleton1 uniqueInstance = null;

private int data;

public synchronized static Singleton1 instance() {
if(uniqueInstance == null) uniqueInstance = new Singleton1();
return uniqueInstance;
}

protected Singleton1() {data=0;}
public int getData(){return data;}
public void setData(int d){data =d;}

}
```

The Singleton Design Pattern – example implementation (2)

```
public class Singleton2 {

public static final Singleton2 uniqueinstance = new Singleton2();

private int data;

private Singleton2() {data=0;}

public int getData(){return data;}

public void setData(int d){data =d;}

}
```

The Singleton Design Pattern – example implementation (3)

```
public class Singleton3 {

private static final Singleton3 instance = new Singleton3();

private int data;

private Singleton3() { data=0; }

public static Singleton3 instance() {return instance;}

public int getData(){return data;}

public void setData(int d){data =d;}

}
```

The Singleton Design Pattern – example implementation (4)

```
public class Singleton4 {

private int data;

private Singleton4() { data=0; }

private static class SingletonHolder {
    private static final Singleton4 INSTANCE = new Singleton4();
  }

public static Singleton4 getInstance() {
    return SingletonHolder.INSTANCE;
  }

public int getData(){return data;}

public void setData(int d){data =d;}

}
```

The Singleton Design Pattern – what about subclassing?

The Singleton Design Pattern is meant to give you control over access to the Singleton class. But subclassing allows other code to access your class without you having direct control

The uniqueness of the class cannot be imposed as a compile-time constraint on the subclass unless you use a private constructor (or declare the class to be final).

If you want to allow subclassing, for example, you might make the constructor protected, but then a subclass could provide a public constructor, allowing anyone to make instances.

QUESTION: which of these 2 *sub-classable singleton* designs do you prefer:?

Supposing we have a Singleton class A and a class B that is a sub-class of A:

1). You can have a single instance of A *OR* a single instance of B, but not both.

2). You can have exactly one instance of A *AND* exactly one instance of B.

TO DO: Can you implement and test one of these designs?

QUESTION: How can/should this design/code be extended to multiple subclasses?