



Lecture 08: Singleton Pattern

IN710: Object-Oriented Systems Development

Semester One, 2020

Kaiako: Grayson Orr

Te Kura Matatini ki Otago, Ōtepoti, Aotearoa

Thursday, 12 March

LECTURE 07: FACTORY PATTERN RECAP

- ▶ Design pattern 03: factory pattern
 - ▶ Definition
 - ▶ Problem/solution
 - ▶ UML & implementation
 - ▶ Applicability
 - ▶ Pros & cons

LECTURE 08: SINGLETON PATTERN TOPICS

- ▶ Design pattern 04: singleton pattern
 - ▶ Definition
 - ▶ Problem/solution
 - ▶ Real world analogy
 - ▶ UML & implementation
 - ▶ Pros & cons

SINGLETON PATTERN: GoF

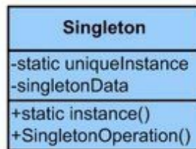
► GoF definition & UML

Singleton

Type: Creational

What it is:

Ensure a class only has one instance and provide a global point of access to it.



SINGLETON PATTERN: DEFINITION

- ▶ Creational pattern
- ▶ Restricts the instantiation of a class to a single instance
- ▶ Useful when exactly one object is needed to coordinate actions within a system
- ▶ The term comes from the mathematical concept of a singleton
- ▶ Considered to be an anti-pattern
 - ▶ Used in situations where is not beneficial
 - ▶ Introduces unnecessary restrictions where a single instance is not actually required
 - ▶ Introduces global state into an application

SINGLETON PATTERN: PROBLEM

- ▶ Ensure that a class has just a single instance
- ▶ Provide a global access point to that instance

SINGLETON PATTERN: SOLUTION

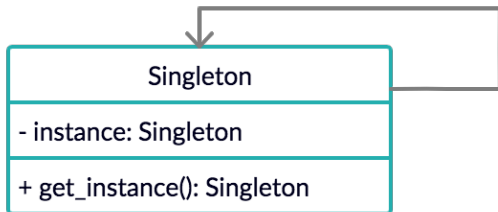
- ▶ Make a private default constructor
- ▶ Prevents other objects from using the "new" keyword
- ▶ Create a static method that acts as a constructor
- ▶ To create an object, call the private default constructor & save it in a static variable

SINGLETON PATTERN: REAL WORLD ANALOGY

- ▶ The government
- ▶ A country can only have one official government

SINGLETON PATTERN: UML

- Consider the following UML diagram:



SINGLETON PATTERN: IMPLEMENTATION

► Naïve singleton

```
class SingletonMeta(type):
    __instance = None

    def __call__(self):
        if not self.__instance:
            self.__instance = super().__call__()
        return self.__instance

class Singleton(metaclass=SingletonMeta):
    pass

def main():
    singleton_1 = Singleton()
    singleton_2 = Singleton()

    if id(singleton_1) == id(singleton_2):
        print('singleton_1 & singleton_2 contain the same instance')
    else:
        print('singleton_1 & singleton_2 contain different instances')

if __name__ == "__main__":
    main() # singleton_1 & singleton_2 contain the same instance
```

SINGLETON PATTERN: IMPLEMENTATION

► Thread-safe singleton

```
from threading import Lock, Thread

class SingletonMeta(type):
    __instance = None
    __lock = Lock()

    def __call__(cls, *args, **kwargs):
        with cls.__lock:
            if not cls.__instance:
                cls.__instance = super().__call__(*args, **kwargs)
            return cls.__instance

class Singleton(metaclass=SingletonMeta):
    def __init__(self, val):
        self.val = val

def test_singleton(val):
    singleton = Singleton(val)
    print(singleton.val)

def main():
    process_1 = Thread(target=test_singleton, args=('One singleton instance',))
    process_2 = Thread(target=test_singleton, args=('Two singleton instances',))
    process_1.start()
    process_2.start()

if __name__ == '__main__':
    main() # One singleton instance
          # One singleton instance
```

SINGLETON PATTERN: IMPLEMENTATION

► Two singleton instances

```
from threading import Lock, Thread

class SingletonMeta(type):
    __instance = None
    __lock = Lock()

    def __call__(cls, *args, **kwargs):
        with cls.__lock:
            if not cls.__instance:
                cls.__instance = super().__call__(*args, **kwargs)
            return cls.__instance

class Singleton:
    def __init__(self, val):
        self.val = val

def test_singleton(val):
    singleton = Singleton(val)
    print(singleton.val)

def main():
    process_1 = Thread(target=test_singleton, args=('One_singleton_instance',))
    process_2 = Thread(target=test_singleton, args=('Two_singleton_instances',))
    process_1.start()
    process_2.start()

if __name__ == '__main__':
    main() # One singleton instance
          # Two singleton instances
```

SINGLETON PATTERN: PROS

- ▶ Ensures a class has only one instance
- ▶ Object is only initialised when it's requested for the first time
- ▶ Global access point to the singleton instance

SINGLETON PATTERN: CONS

- ▶ Violates the single responsibility principle
- ▶ Requires a different implementation in a multi-threaded environment

PRACTICAL

- ▶ Series of tasks covering today's lecture
- ▶ Worth 1% of your final mark for the Object-Oriented Systems Development course
- ▶ Deadline: Tuesday, 24 March at 5pm

EXAM 02

- ▶ Series of tasks covering lectures 05-08
- ▶ Worth 6% of your final mark for the Object-Oriented Systems Development course
- ▶ Deadline: Thursday, 19 March at 5pm

LECTURE 09: ADAPTER PATTERN TOPICS

- ▶ Design pattern 05: adapter pattern
 - ▶ Definition
 - ▶ Problem/solution
 - ▶ Real world analogy
 - ▶ UML & implementation
 - ▶ Pros & cons