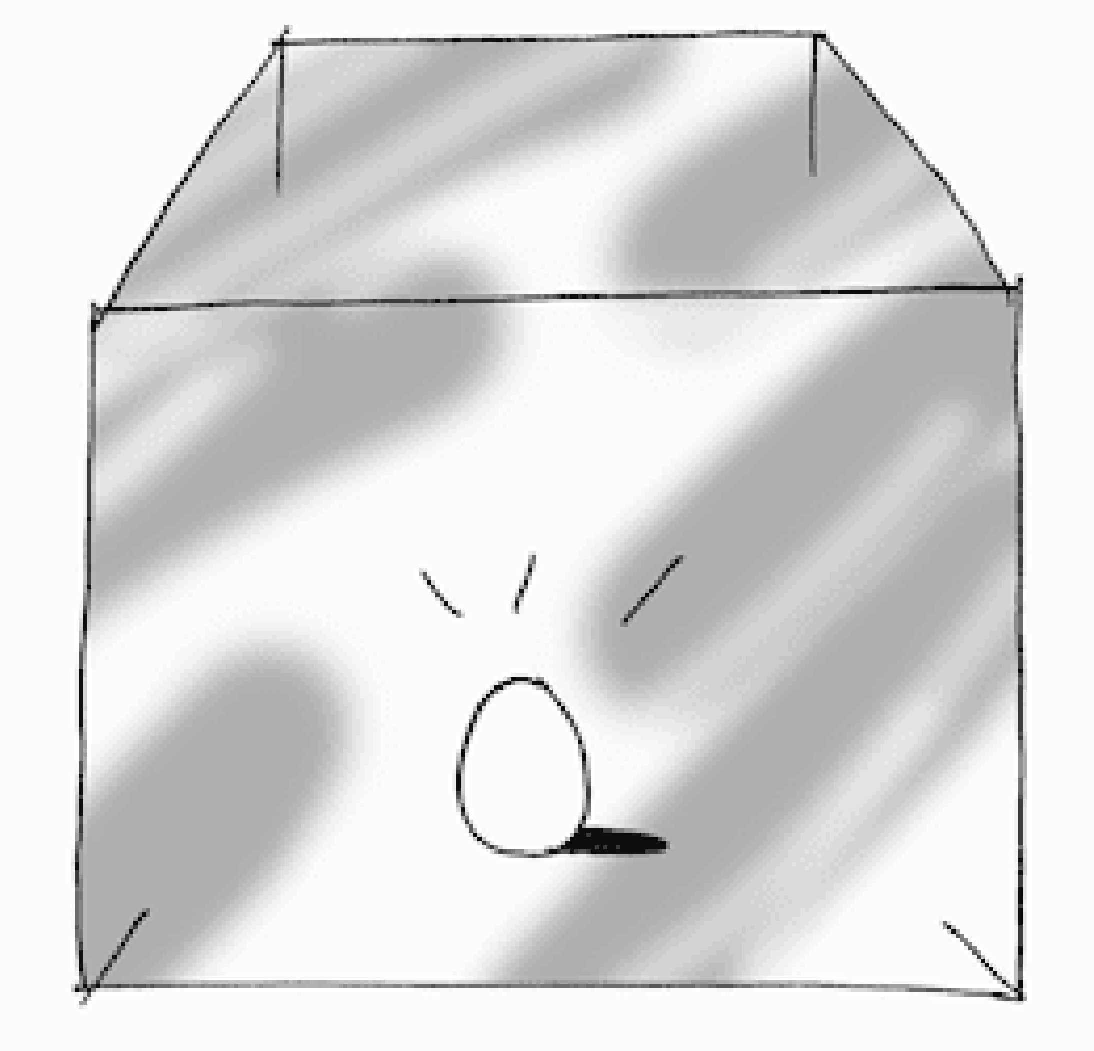
Singleton Pattern

Only One Instance



Singleton Pattern

When every object holds the same data, creating multiple copies wastes memory and effort.

Why not keep a single shared instance and reuse it everywhere?

This saves resources and ensures consistency.

The Problem

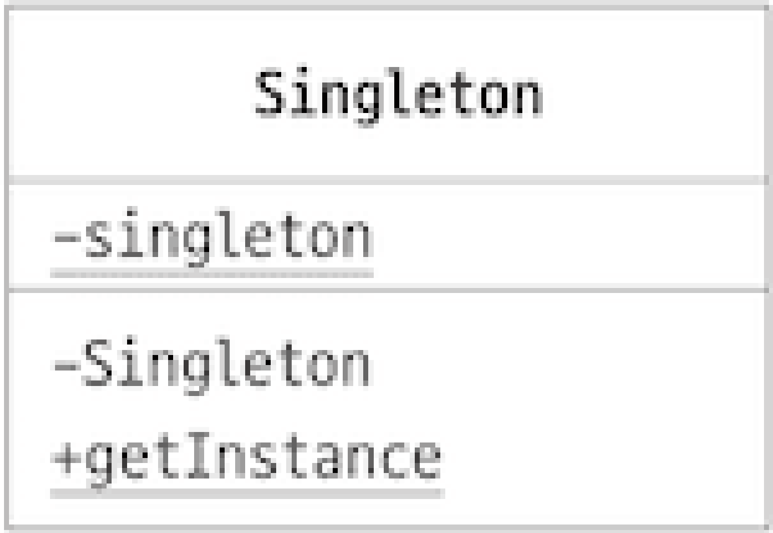
- Sometimes we need exactly one instance of a class in the entire application.
- Creating multiple instances could cause problems or waste resources.
- Database connection, logger, configuration settings needs only one instance.

The challenge: how to ensure a class has only one instance and provide global access to it?

The *Singleton* as the Solution

- Ensure a class has only one instance and provide a global point of access to it.
- The class itself is responsible for keeping track of its sole instance.
- The class can ensure that no other instances can be created.

The Design



Step 1: Understand the Players

In this design, we have only one player:

- **Singleton** (ensures only one instance exists)

Step 2: Key Characteristics

- **Private Constructor**: Prevents external instantiation
- **Static Instance**: Holds the single instance
- **Static Method**: Provides global access point
- **Lazy Initialization**: Instance created when first needed

Code

- Main Method
- Singleton Implementation

Main Method

```python
from singleton import Singleton

def main():
    print("=== Singleton Pattern Example ===\n")

    obj1 = Singleton.get_instance(100)
    obj2 = Singleton.get_instance(200)  # 200 is ignored

    if obj1 is obj2:
        print("obj1, obj2 are the same instances.")

    print(f"obj1: {obj1.value}")  # 100
    print(f"obj2: {obj2.value}")  # 100
```

Step 1: Create multiple "instances"

```
obj1 = Singleton.get_instance(100)
obj2 = Singleton.get_instance(200)
```

- All variables point to the same instance.
- Only the first value (100) is used for initialization.

Step 2: Verify they are the same

```python
if obj1 is obj2:
    print("obj1, obj2 are the same instances.")
```

- The `is` operator checks object identity.
- All three variables refer to the same object in memory.

Output Example

```
An instance is created.
obj1 and obj2 are the same instances.
obj1 id: 140234567890 100
obj2 id: 140234567890 100
Business method: Singleton instance 140234567890 with value=100
```

Singleton Implementation (Example)

There can be many ways to implement the Singleton.

```python
class Singleton:
    _instance = None
    _initialized = False

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            print("An instance is created.")
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self, value=None):
        if not self._initialized:
            self.value = value
            Singleton._initialized = True
```

```python
@classmethod
def get_instance(cls, value=None):
    return cls(value)

def some_business_method(self):
    return f"Singleton instance {id(self)} with value={self.value}"
```

- `get_instance()` provides an alternative access method
- Business methods work normally on the singleton instance

Python-Specific Implementation

Using `__new__()` Method

- `__new__()` is called before `__init__()`
- Controls object creation at the class level
- Returns the existing instance if already created

Initialization Control

- Use `_initialized` flag to prevent re-initialization
- Only initialize once, even if the constructor is called multiple times

Discussion

Consider DIP

A Good alternative to Singleton pattern in modern programming is Dependency Injection:

Dependency Injection - instead of global access, explicitly pass dependencies through constructors or methods. This makes dependencies visible and testing easier.

Use static method

```
# Wrong way
s1 = Singleton()
s2 = Singleton()
print(s1 is s2)  # Should print True

# Right way
s1 = Singleton.??()
s2 = Singleton.??()
print(s1 is s2)  # Should print True
```

Singleton Benefits and Drawbacks

Benefits:

- **Controlled Access**: Single point of access
- **Reduced Memory**: Only one instance
- **Global State**: Shared across application

Drawbacks:

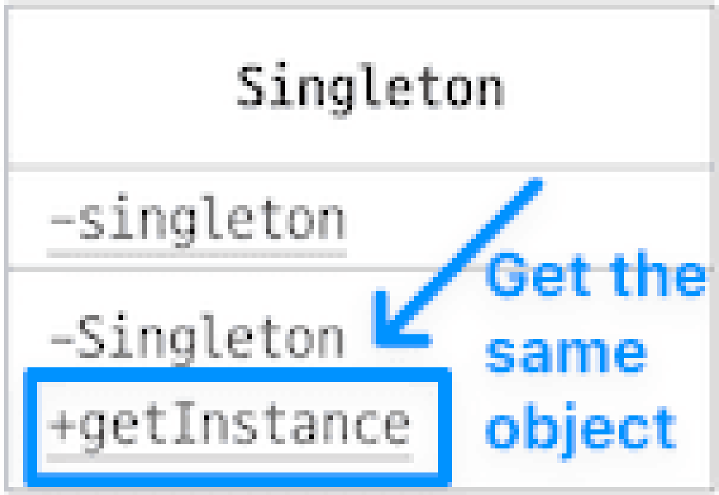
- **Testing Difficulty**: Hard to mock or reset
- **Hidden Dependencies**: Global state can cause issues
- **Thread Safety**: Need careful implementation in multithreaded environments

Related Patterns

Factory Method: Factory can be implemented as a Singleton

For example, instead of multiple factories creating DatabaseConnection objects, you enforce a single factory object (singleton) that ensures all connections come from a single source.

UML



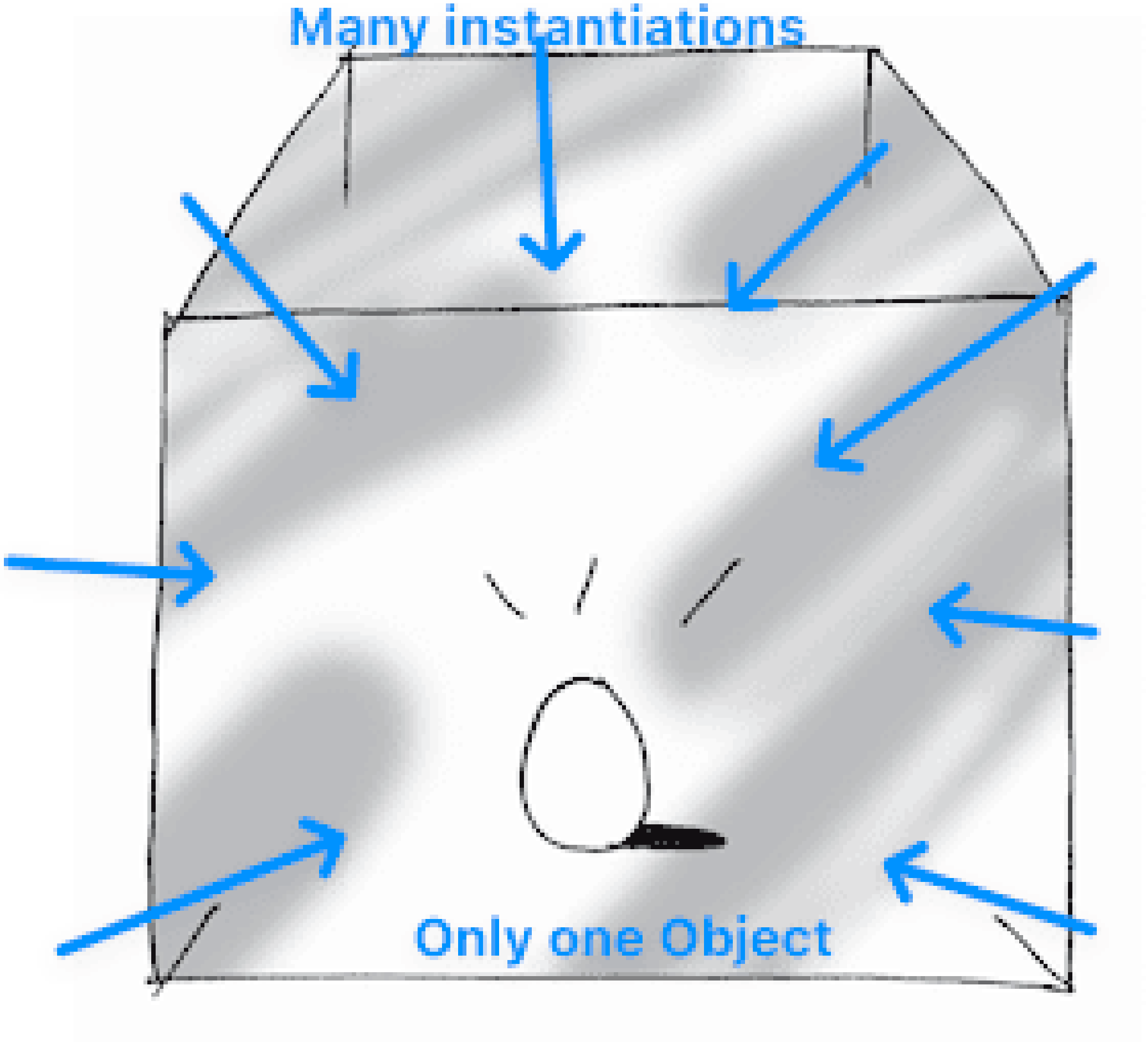
Singleton

-singleton

-Singleton

+getInstance

Get the same object


Many instantiations
Only one Object