Creational Design Patterns

Instructor: Mehroze Khan

Creational Design Patterns

- Creational design patterns provide various **object creation** mechanisms, which increase **flexibility** and **reuse** of existing code.
- Creational design patterns **abstract** the instantiation process.
- They help make a system independent of how its objects are created, composed, and represented.
- A **class creational pattern** uses inheritance to vary the class that's instantiated.
- An **object creational pattern** will delegate instantiation to another object.

Creational Design Patterns

- First, they all encapsulate knowledge about which concrete classes the system uses.
- Second, they hide how instances of these classes are created and put together.
- Creational patterns give you a lot of flexibility in what gets created, who creates it, how it gets created, and when.

Catalog of Design Patterns

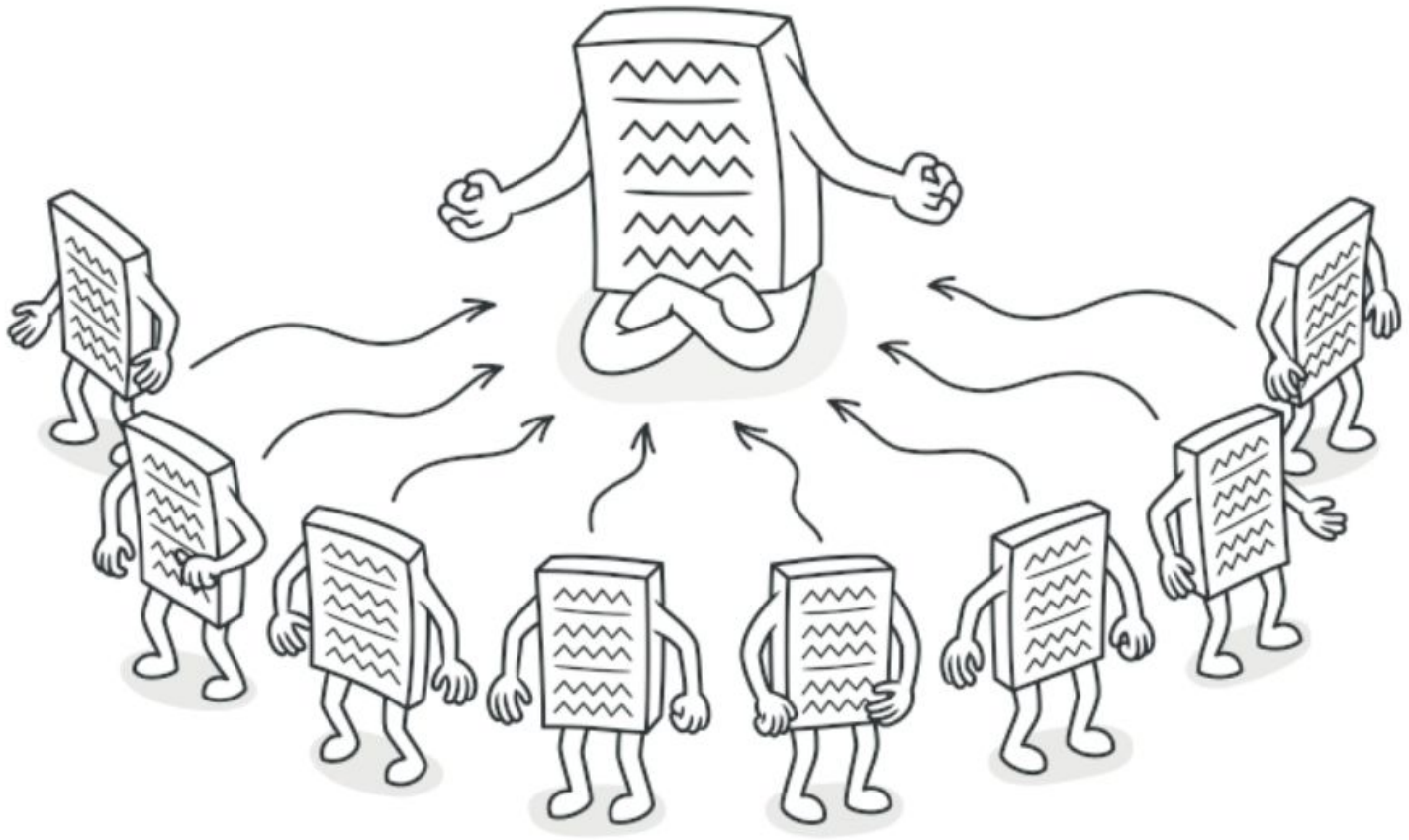
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Singleton Pattern

Singleton Pattern

Intent

- **Singleton** is a creational design pattern that lets you ensure that a class has only **one instance**, while providing a **global access point** to this instance.



Problem

- The Singleton pattern solves two problems at the same time:

1. **Ensure that a class has just a single instance**.

   Why would anyone want to control how many instances a class has?

   The most common reason for this is to **control access to some shared resource**—for example, a database or a file.

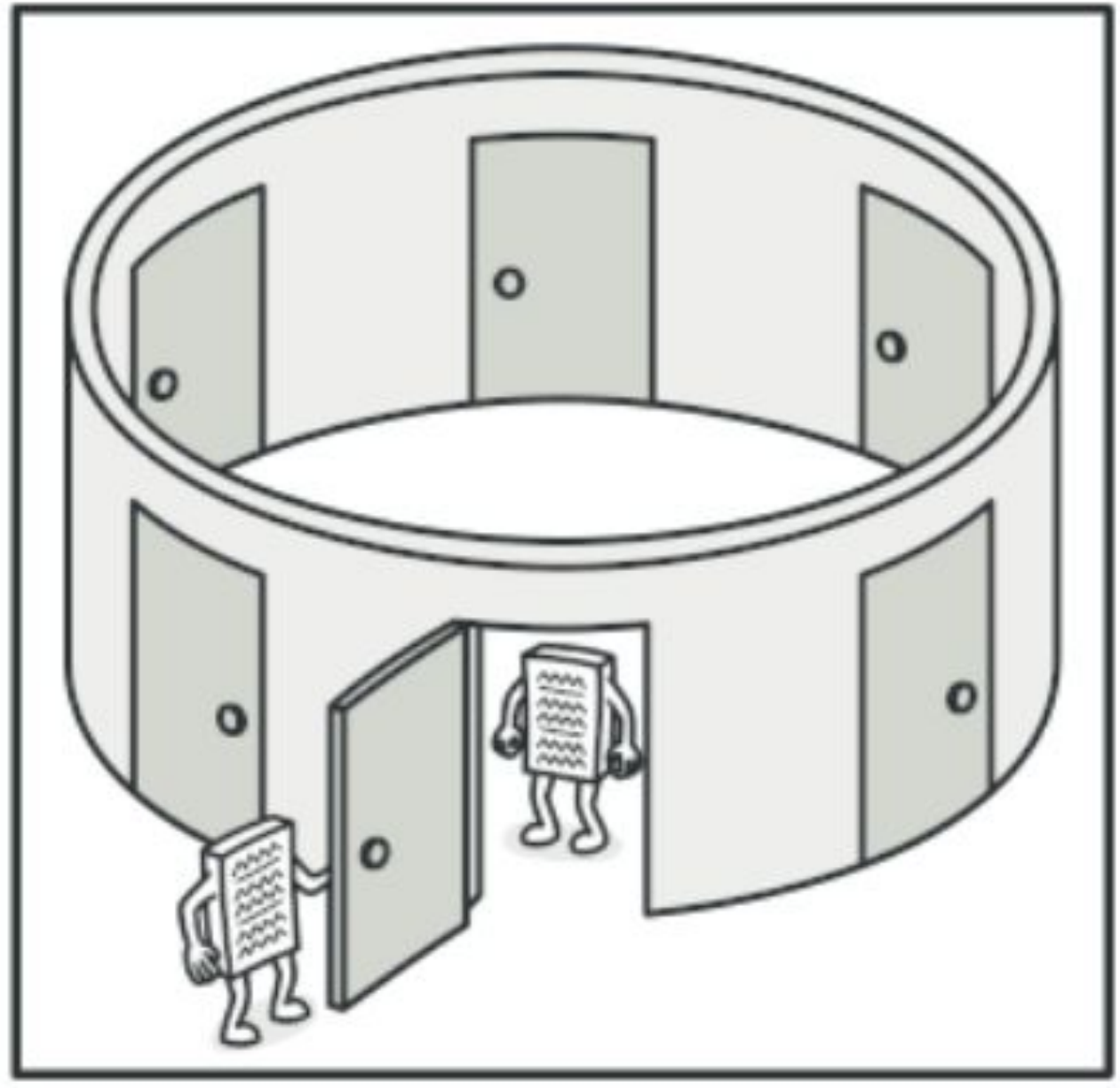
- Imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you'll get the one you already created.
- This behavior is impossible to implement with a regular constructor since a constructor call **must** always return a new object by design.

Problem

2. **Provide a global access point to that instance.**

- Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program.
- However, it also protects that instance from being overwritten by other code.
- There's another side to this problem:

  You don't want the code that solves problem #1 to be scattered all over your program. It's much better to have it within one class, especially if the rest of your code already depends on it.



Clients may not even realize that they're working with the same object all the time.

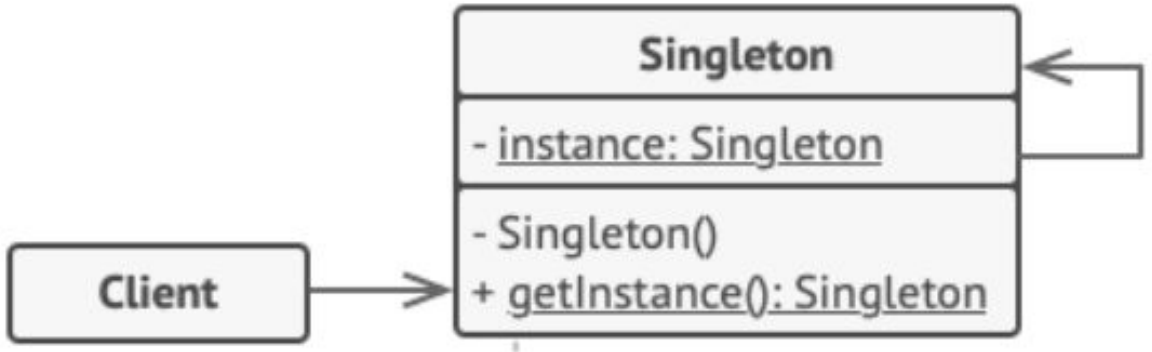
Solution

- All implementations of the Singleton have these two steps in common:

1. Make the **default constructor private**, to prevent other objects from using the new operator with the Singleton class.
2. Create a **static creation method** that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

- If your code has access to the Singleton class, then it's able to call the Singleton's static method. So, whenever that method is called, the same object is always returned.

Strcuture



1 The **Singleton** class declares the static method `getInstance` that returns the same instance of its own class.

The Singleton's constructor should be hidden from the client code. Calling the `getInstance` method should be the only way of getting the Singleton object.

```
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```

Implementation (Lazy Initialization)

```
class Singleton {
  private:
  static Singleton *instance;
  int data;

  // Private constructor so that no objects can be created.
  Singleton() {
    data = 0;
  }
  public:
  // Lazy Initialization
  static Singleton *getInstance() {
    if (instance == NULL) {
       instance = new Singleton();
    }
    return instance;
  }
```

```
 int getData() {
     return this -> data;
   }

   void setData(int data) {
     this -> data = data;
   }
};
```

Implementation

```
//Initialize pointer with NULL so that it can be initialized in first call to getInstance()
Singleton *Singleton::instance = NULL;


int main(){
   Singleton *s = Singleton::getInstance();
   s->setData(7);
   cout << s->getData() << endl;


   Singleton *p =  Singleton::getInstance();
   p->setData(100);
   cout << p->getData() << endl;
   return 0;
}
```

Output:

```
// 7
// Address of s : 0x1793010


// 100
// Address of p : 0x1793010
```

Implementation (Eager Initialization)

```
class Singleton {
  private:
  static Singleton *instance = new Singleton();
  int data;

  // Private constructor so that no objects can be created.
  Singleton() {
    data = 0;
  }
  public:
  // Eager Initialization
  static Singleton *getInstance() {
      return instance;
  }
```

```
 int getData() {
    return this -> data;
  }

  void setData(int data) {
    this -> data = data;
  }
};
```

References

- Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Pearson, 1995.
- Helping Links:
- https://refactoring.guru/design-patterns/
- https://www.geeksforgeeks.org/factory-method-for-designing-pattern/
- https://sourcemaking.com/design_patterns/