# Creational Patterns

Singleton

Factory Method

Abstract Factory

# Creational Patterns

- **Goal:** make a system independent of how its objects are created, composed, and represented
  - isolating the details of object creation so your code is not dependent on what types of objects there are

- Becomes important as emphasis moves towards dynamically composing smaller objects to achieve complex behaviors.
  - need more than just instantiating a class
  - need consistent ways of creating related objects.

# Recurring Themes in Creational P.

- Hide the secret about which concrete classes the system uses.
- Hide the secret about how instances are created and put together.
- All the system knows is the interface

- Gives flexibility in
  - **what** gets created
  - **who** creates it
  - **how** it gets created
  - **when** it get gets created

# Creational Patterns

- They rarely use constructors directly.
  - often hides the constructors in the classes being created,
  - and provides alternate methods to return instances of the desired class.

- Useful when taking advantage of **polymorphism** and need to **choose** between different classes at **runtime** rather than compile time

# Creational Patterns

- Singleton
  - To make the class has only 1 instance
- Abstract factory
  - To create an instance from a family of related classes without specifying the concrete name
- Factory Method
  - To define a virtual constructor in a creator class but defer object creation to subclasses
- Builder
  - To separate construction of an object from its representation.
- Prototype
  - To make complex objects create like themselves

# What is common?

- Device drivers
- Thread pools
  - or other pool of resources
- Objects for logging
- Printer spooler
- Window manager
- Objects that contain registry settings
- ….

1. We only need **one** of them
   - Having multiples of them will cause *chaos*
2. They are accessed globally

# Solution attempts

| Attempt | Why Not |
| --- | --- |
| ■ Make it a global variable<br>Read: https://wiki.c2.com/?GlobalVariablesConsideredHarmful | ■ How to prevent any code not to create more than once object?<br>■ Eager initiation, always |
| ■ Have all methods static | ■ Looses polymorphism<br>■ No ‘virtual static’ in C++<br>■ Static method belongs to class, Java polymorphism does not work<br>■ Hard to refactor and subclass |

# Singleton

- **Intent**: Ensure the class has only 1 instance and provide global visibility

- Applicability:
  - The class should have *exactly one* instance which should be *accessible* to clients
  - The class can be *subclassed*, and clients should be able to use the subclass instance without having to change any of their code.
  - Simplify access to global resources without using global variables

# Enforcing singularity

- Need to control how class instances are created and then ensure that only one gets created at any given time.

1. The users of the class should always be *free* from having to monitor and control the number of running instances of the class.

2. *The responsibility of having only one instance of the class should fall on the class itself and not on the user of the class.*

- Let's try ourselves how to do it…

# Example

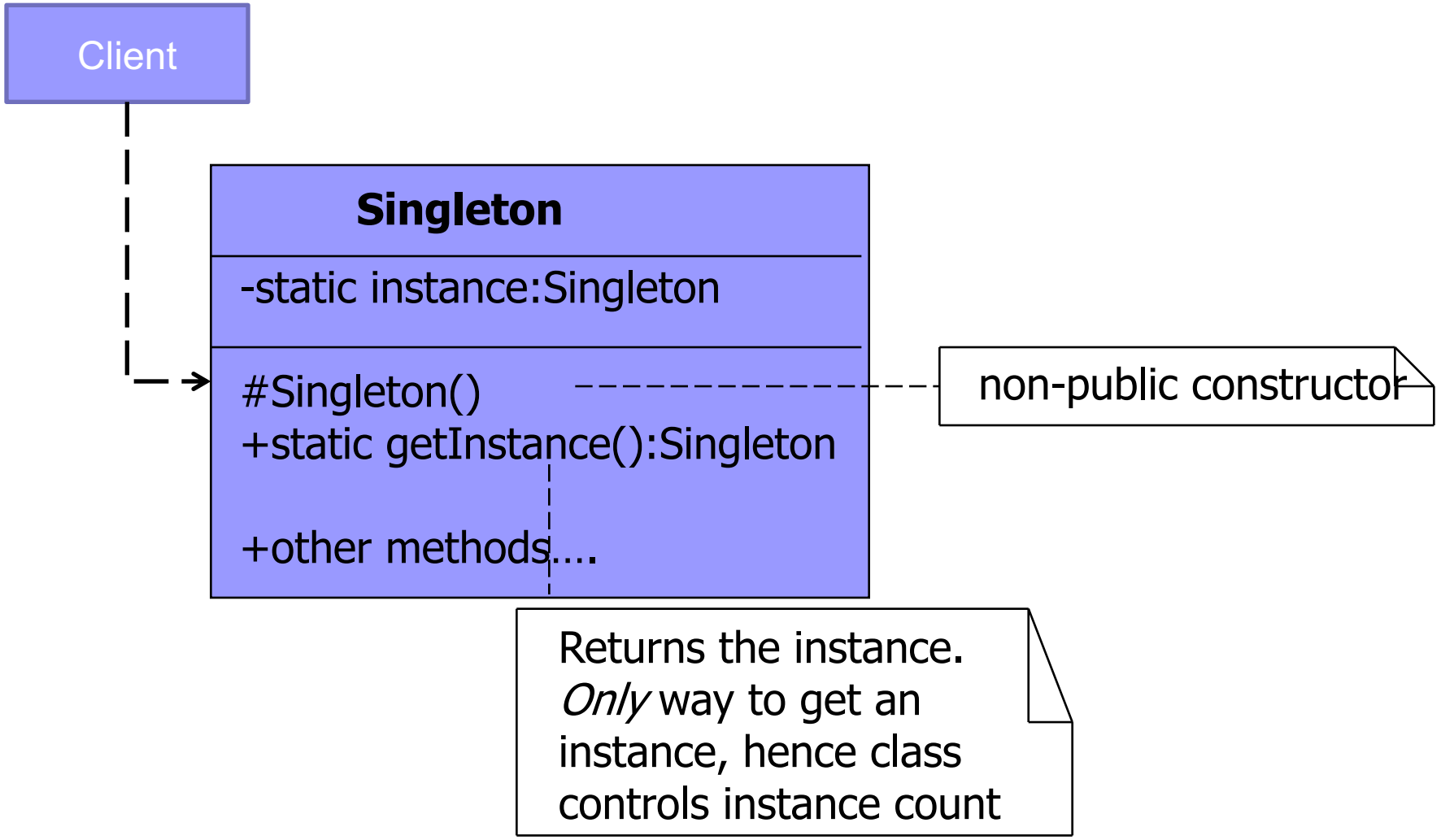
- We need to maintain an incremental counter,
- the simple counter class needs to keep track of an integer value that is being used in multiple areas of an application.
- The class needs to be able to increment this counter as well as return the current value.
- the desired class behavior would be to have exactly one instance of a class that maintains the integer and nothing more.
  - Problem: Global visibility, enforce only 1 instance

```
class  SingleCounter{
  public int getCurrentValue(){ return cnt;}
  public void increment(){cnt++;}
  private int cnt=0;
                                    ///restrict that no one will create instance
                                    // access/return the instance
}
                                    //save the only object
}
Main(){
   SingleCounter counter;
                                    //client usage. How to get the single instance
   counter.increment();
   counter.getCurrentValue();
…}
```

# Eager creation

```
class  SingleCounter{
  public synchronized int getCurrentValue(){ return cnt;}
  public synchronized void increment(){cnt++;}
  private int cnt=0;
                                    ///restrict that no one will create instance
                                    //return the instance
 private static SingleCounter theOne=new SingleCounter(); //eager
}
Main(){
                                    
   counter=SingleCounter. getInstance(); //client usage
   counter.increment();
   counter.getCurrentValue();
…}
```

# Singleton - Structure



# Singleton -Implementation

- Java implementation

```
public class Singleton{
   private static Singleton instance=null ;
   protected Singleton(){…} //nonpublic constructor

   public static Singleton getInstance(){
         //lazy init
      if(instance==null) instance=new Singleton();
      return instance;
   }
   /*other methods and attributes*/
}
```

# Singleton -Implementation

- C++ implementation

```
class Singleton{
   private: static Singleton* instance=0;
   protected: Singleton(){…}
   public:
       static Singleton* getInstance();

      Singleton(const Singleton& other)=delete;
      void operator= (const Singleton&)=delete;
      /*other methods and attributes*/
};
Singleton* Singleton::getInstance(){
     if (instance==0) instance=new Singleton();
     return instance;}
```

# Implementation issue

- Thread safety

These are not thread safe:

- Java

```
public static Singleton getInstance(){
      if(instance==null) instance=new Singleton();
      return instance;
}
```

- C++

```
Singleton* Singleton::getInstance(){
     if (instance==0) instance=new Singleton();
     return instance;
}
```

# Singleton –Java thread safe

```
public class Singleton{
   private final static Singleton instance=new Singleton();
                    //thread safe and eager init
   private Singleton(){…}
   public static Singleton getInstance(){
                    return instance;}
}

public class Singleton{
   private static Singleton instance=null ;
   private Singleton(){…}
   //lazy init and thread safe
  public static synchronized Singleton getInstance(){
          if(instance==null) instance=new Singleton();
          return instance;}
}
```

# Java singleton with enum

```
public enum Singleton {
   UNIQUE_INSTANCE(“default”);//init with default

   public void setValue(String str){ value=str;}
   public String getValue(){return value;}
   private String value;
   private Singleton(String val){value=val;}
}
//Client code
public class SingletonClient {
  public static void main(String[] args) {
      Singleton singleton = Singleton.UNIQUE_INSTANCE;
      singleton.setValue(“the one”);//using the singleton
    }
}
```

# Singleton- thread safe C++11

```
class Singleton {
public:
    static Singleton& getInstance(const string val) {
        static Singleton instance(val);
        // Initialized once, thread-safe, lazy init
        return instance;
    }
    string value() const{return value_;}
private:
    Singleton(const string& val):value_(val) {}
        // Private constructor
    /*delete copy and operator=  constructors*/
    string value_;
};
```

C++11: initializer for a local static variable is only run once, even in the presence of concurrency.

# Notes..

- Nothing, except for the Singleton class itself, can replace the cached instance.

- Java: each class loader will have its own singleton instance
  - Multiple class loader → multiple instance

- C++ does not have garbage collection, so..

Reading: *"To Kill a Singleton"*

https://sourcemaking.com/design_patterns/to_kill_a_singleton

# Singleton-Consequences

- Controlled access to the sole instance
- Reduced name space
  - Avoids polluting the namespace with global variables
- More flexible than class operations (static)
  - Still in the object oriented paradigm
- Permits subclassing
  - client applications can be configured at runtime by selecting a different subclass
    - Registry of singletons, getInstance() makes a look up
- Be careful in using singleton

https://wiki.c2.com/?SingletonsAreEvil

# Why not global?

- Temptation: create an instance of a counter class as a static global variable.
- Really solves only a part of the problem;
  - the problem of global accessibility,
  - but does nothing to **ensure** that there is only one instance of the class running at any given time.
- The responsibility of having only one instance of the class should fall on the class itself and not on the user of the class.

# Why not static

- Static breaks encapsulation
- Thread safety
- Initialization of static members
  - The order of init is not determined
  - Cannot do Lazy initialization – memory fill up
  - Proper initialization
- Refactoring is hard due to coupling
  - In later iterations you realized more than 1 instance is needed
- Have static global variable of the class?
  - Statics are initialized before main(), could not use the info main creates

# Implementation issue: Subclass

- The main issue is not so much defining the subclass but installing its unique instance so that clients will be able to use it.
  - How to make `instance` refer to an object of subclass?
  - If you know the subclasses, choose to create one of them in the super class getInstance() method

# Example: File System

```
FileSystem&  FileSystem::instance() {
  #if PLATFORM == PLAYSTATION3
    static FileSystem *instance = new PS3FileSystem();
  #elif PLATFORM == WII
    static FileSystem *instance = new WiiFileSystem();
  #endif
  return *instance;
}
```

- FileSystem is a Singleton with `readFile` and `writeFile` operations

```
class PS3FileSystem : public FileSystem{/* use sony IO */}
class WiiFileSystem : public FileSystem {/*use Nintendo IO*/
```

Source: https://gameprogrammingpatterns.com/singleton.html

# Implementation issue: Subclass

- The main issue is not so much defining the subclass but installing its unique instance so that clients will be able to use it.
  - How to make `instance` refer to an object of subclass?
  - Soln1:If you know the subclasses, choose to create one of them in the super class getInstance() method
- Registry of singletons: another solution
  - Also works when we need limited number of instances instead of one.
    - E.g. only 3 registers

# Registry of Singletons

- Registry: map of names to singletons
  - Java:  Map<String, Singleton> //use common interface
  - C++:  map<string, Singleton*>
- getInstance(String name) makes a name lookup
- How to put subclass instances?
  - Java reflection –careful, it violates encapsulation
  - Constructor of the subclass may register itself
  - Problem: how to activate the constructor in the subclass?

# Registry of Singletons: only 1

```
public class Singleton{
  private Map<String, Singleton> registry=new HashMap<String, Singleton>();
  private static Singleton theInstance=null;
  protected Singleton(){
        registry.put(this.getClass().getSimpleName(), this);
        //When a subclass is instantiated, it registers itself using its class name.
       //…other inits
  }
  public static Singleton getInstance(){
      if(theInstance==null){
            String name=System.getProperty("mypackg.singletonname");
            theInstance=registry.get(name);
     } return theInstance;
  }
  private final static Singleton sole=new Singleton();
  public static void activate(){}
}
```

https://stackoverflow.com/a/54859005/25503790

# Registry of Singletons

```
public class Sub extends Singleton{
  protected Sub(){
        super(); //makes the registry,i.e. registers itself in the base class registry.
       //…other inits
  }
  private final static Sub sole=new Sub(); //created at load time
  public static void activate(){}   //this is a hack to force class loading.
  /* other methods*/
}
```

- Too many instances are created, reflection would save space
- Best use registry for limited number of singletons
  - Register limited number of instances
  - Lookup inside the getInstance()

https://stackoverflow.com/a/54859005/25503790

# Summary

- Singleton ensures one instance with global access
  - Configuration managers, loggers, asset managers.
- Thread safety is achievable via DCL, enum, or eager initialization.
  - Use enum for simple Java Singletons.
  - Use DCL or static initialization for complex cases.
- Registries (like Service) extend Singleton for flexible subsystem management.

# Known uses

- Java.lang.Runtime
  - getRuntime() is the getInstance method
- Java.awt.Desktop
  - getDesktop
- Log4j library Logger

Beware: Do not overuse Singletons

- They are a kind of Global variable
- Evaluate your design, maybe you don’t need one

https://gameprogrammingpatterns.com/singleton.html

# Related patterns

- A **Facade** class could be a **Singleton** since a single facade object is sufficient in most cases.
- **Flyweight** would resemble **Singleton** if you somehow managed to reduce all shared states of the objects to just one flyweight object.
- But..
  - There should be only one Singleton instance, whereas a *Flyweight* class can have multiple instances with different intrinsic states.