

# Objet et développement d'applications

## Cours 8 - Pattern Singleton

### Patrons de conception pour POO

Florian Richoux

2014-2015

# Définir le pattern Singleton

## Définition

**Le pattern Singleton** garantit qu'une classe n'a qu'**une seule instance** et fournit un point d'accès de type global à cette classe.

# Définir le pattern Singleton

## Définition

Le pattern **Singleton** garantit qu'une classe n'a qu'**une seule instance** et fournit un point d'accès de type global à cette classe.



# Définir le pattern Singleton

## Définition

**Le pattern Singleton** garantit qu'une classe n'a qu'**une seule instance** et fournit un point d'accès de type global à cette classe.



# Programmer le pattern Singleton



# Singleton

# Singleton

=

Singleton  
=  
boss final  
des patterns

## Gestion de la mémoire

Les difficultés de ce pattern sont intimement liés à la gestion de la mémoire.

## En Java

Le programmeur n'a pas de contrôle sur la gestion de la mémoire. On ne peut faire que des Singletons grossiers.

## C'est pourquoi...

...on va principalement parler en C++ aujourd'hui !

## Sceptique

À quoi ça sert d'écrire une classe pour instancier qu'un seul objet ?

Vous l'avez déjà fait plein de fois !

Pensez au nombre de classes que vous avez écrit alors qu'elles n'étaient instancier qu'une seule fois dans vos programmes.

Il y a une notion de **sécurité** dans le Singleton, par exemple pour l'accès à des ressources critiques.

## Singleton dans les patterns

On peut retrouver le Singleton dans :

- ▶ Les Factory,
- ▶ Façade,
- ▶ State,
- ▶ ...

# First shot

(Donnée statiques + méthodes statiques)  $\neq$  Singleton

```
class SingletonPrinterPool
{
    static private Stack<PrintJob> printQueue_ ;
    static private Font defaultFont_ ;
    static private Port printingPort_ ;

    static public void addPrintJob(PrintJob pj)
    {
        printQueue_.push( pj );
    }
}

class User
{
    public PrintJob pj( "document.txt" );
    SingletonPrinterPool.addPrintJob( pj );
}
```

# Problème avec cette approche

(Donnée statiques + méthodes statiques)  $\neq$  Singleton

- ▶ Les méthodes statiques ne sont plus virtuelles : **plus de polymorphisme**.
- ▶ Pas de contrôle sur l'instanciation des attributs. Static = instancié au chargement en mémoire du programme.

# En Java

# Approche classique

## Singleton classique en Java

```
class Singleton
{
    static private Singleton instance_ ;
    ... // autres attributs

    private Singleton() {} // constructeur privé !

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

    ... // autres méthodes
}
```

# Diagramme UML habituel

```
Singleton
-<<static>> instance: Singleton
-Singleton()
+<<static>> getInstance(): Singleton
```

# Vraiment difficile ?

## Deux cas où faire attention

- ▶ Gérer les singltons en **programmation concurrente**.
- ▶ Gérer la **plage de vie** des singltons.

## Deux cas où faire attention

- ▶ Gérer les singltons en **programmation concurrente**.
- ▶ Gérer la **plage de vie** des singltons. **Impossible en Java**

## Ajouter de la synchronisation

```
class Singleton
{
    static private Singleton instance_ ;
    ...
    private Singleton() {}

    static public synchronized Singleton getInstance()
    {
        if( instance_ == null )
            instance_ = new Singleton();
        return instance_;
    }
    ...
}
```

Ok, mais...

`synchronized` est une instruction coûteuse. Si on appelle souvent `getInstance()`, ça va pas le faire.

## Deux choix

- ▶ Soit vous savez que `getInstance()` ne va pas souvent être appelée, dans ce cas laissez le `synchronized`.
- ▶ Soit on s'en débarrasse et on trouve une astuce.

# Astuce bof

Astuce pas très astucieuse

```
class Singleton {  
    static private Singleton instance_ = new Singleton();  
  
    private Singleton() {}  
  
    static public Singleton getInstance() {  
        return instance_;  
    }  
}
```

Pour

Plus de problème d'instanciation simultanée.

Contre

Création systématique d'un singleton, même si on en a pas besoin lors d'une exécution.

# Astuce double verrou

## Astuce plus astucieuse

```
class Singleton
{
    static private volatile Singleton instance_;

    private Singleton() {}

    static public Singleton getInstance()
    {
        if( instance_ == null ) // 1er verrou
        {
            synchronized (Singleton.class)
            {
                if( instance_ == null ) // 2eme verrou
                    instance_ = new Singleton();
            }
        }
        return instance_;
    }
}
```

# Astuce double verrou

## Mise en cache

Quand un thread lit la valeur d'une variable, il écrit cette valeur dans le cache. Entre deux lectures, il sait si il a modifié cette variable ou pas. Si c'est pas le cas, il ira chercher la valeur dans le cache, beaucoup plus rapide d'accès.

## Variable volatile

`volatile` empêche le compilateur de faire certaines optimisations concernant la lecture/écriture de la variable, notamment en ne mettant pas sa valeur en cache (pour simplifier). Utile quand on a plusieurs threads susceptibles de modifier une même valeur.

## Double verrou

- ▶ **1er verrou** : on y entre **uniquement** si le singleton n'a jamais été instancié. C'est là qu'on gagne beaucoup !
- ▶ **Synchronization** : on entre ici dans une section critique.
- ▶ **2ème verrou** : Si effectivement le singleton n'a toujours pas été instancier, on l'instancie.

## Que du bonheur

Plus de problème d'instanciations simultanées, code efficace et singleton créé uniquement si quelqu'un appelle `getInstance()`.

# En C++

insérer un rire diabolique ici

# Approche classique (mais incomplète)

## Singleton classique en C++

```
// Singleton.h
class Singleton
{
public:
    static Singleton* getInstance()
    {
        if( !pInstance_ )
            pInstance_ = new Singleton();
        return pInstance_;
    }

private:
    Singleton();
    static Singleton *pInstance_;
};

// Singleton.cpp
Singleton* Singleton::pInstance_ = nullptr;
```

# Erreur de simplification

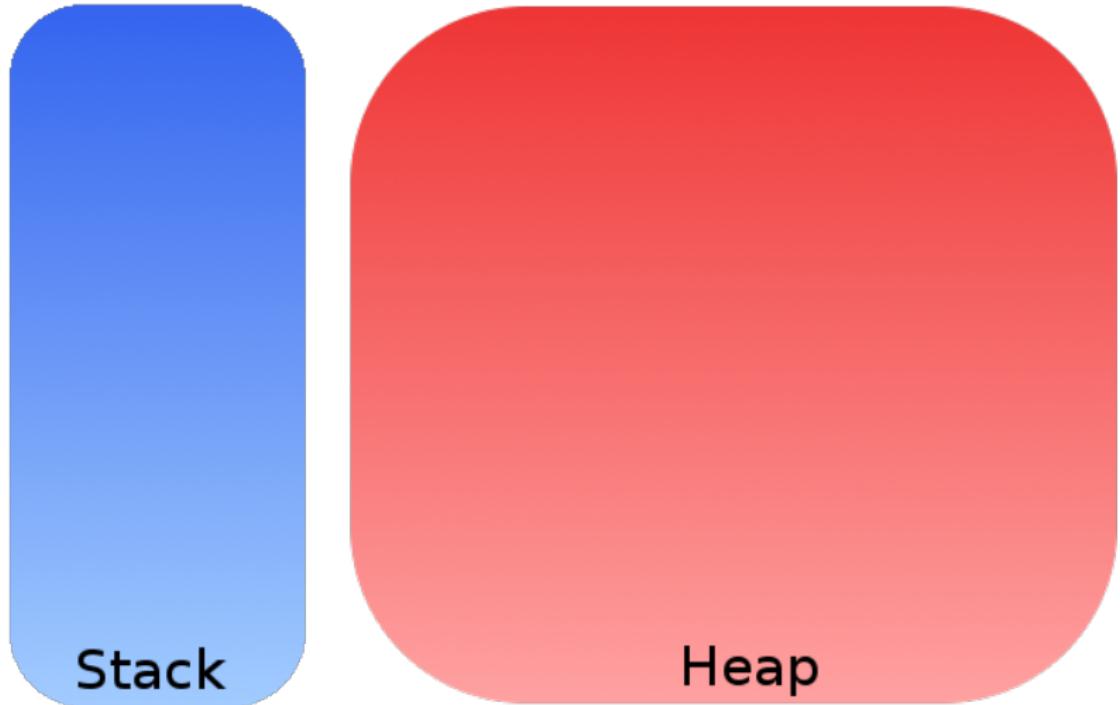
Pointeur ⇒ objet (ça marche pas)

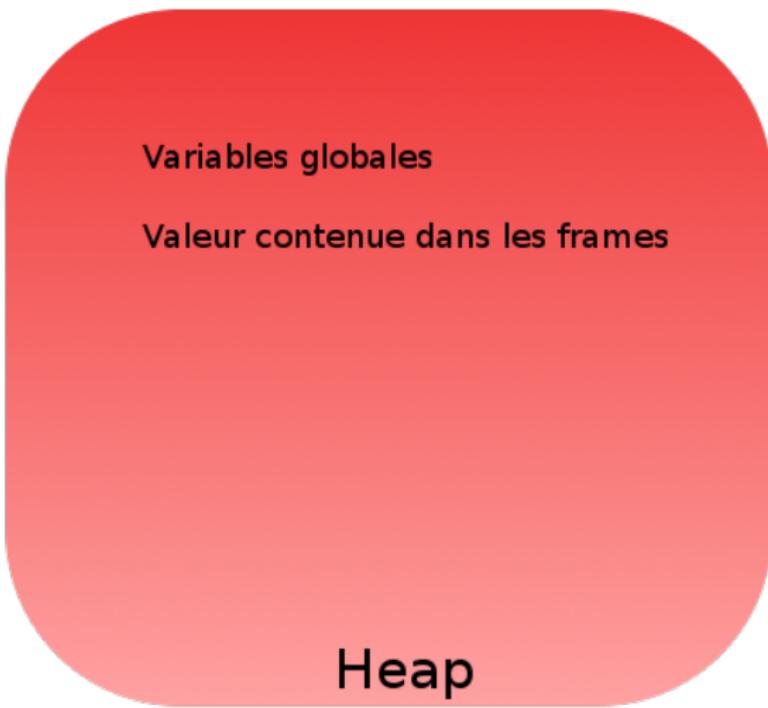
```
// Singleton.h
class Singleton
{
public:
    static Singleton* getInstance()
    {
        return &instance_;
    }

    int fonction();

private:
    Singleton();
    static Singleton instance_;
};

// Singleton.cpp
Singleton Singleton::instance_;
```





## Stack (la pile)

- ▶ En RAM (normalement).
- ▶ Gérée comme une pile, comme son nom l'indique.
- ▶ Une stack par thread.
- ▶ Taille définie au lancement du programme.
- ▶ Accès plus rapide (gestion de pile), et valeurs souvent mises en cache.

## Heap (le tas)

- ▶ En RAM (normalement).
- ▶ Une heap générale.
- ▶ Taille définie au lancement du programme mais extensible.

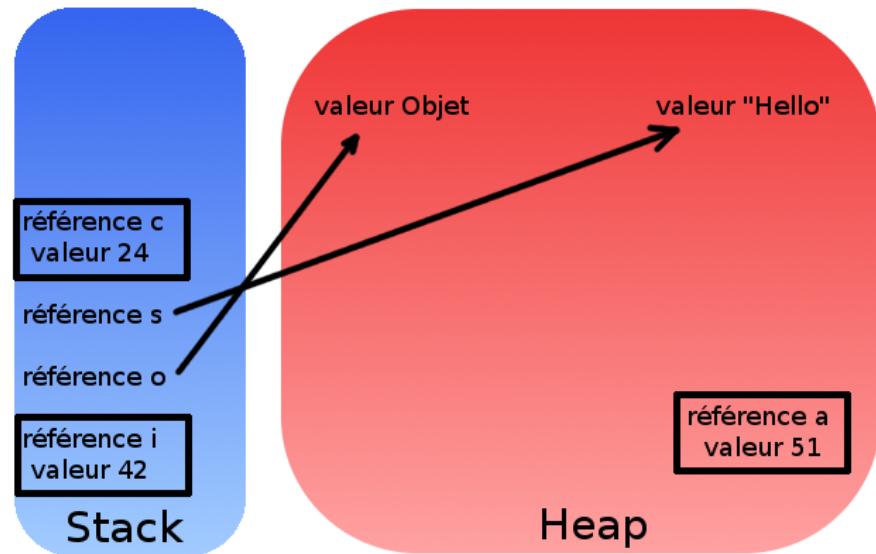
## Stack (la pile)

- ▶ Variables **locales** (final ou pas).
- ▶ **Frames** (valeur d'un pointeur)

## Heap (le tas)

- ▶ Valeurs pointées par les frames (allouées **dynamiquement**).
- ▶ Variables **statiques** (dans PermGen).

# Gestion de la mémoire en Java



```
public void method()
{
    int i;
    i = 42;

    Objet o = new Objet();

    String s = "Hello";

    final int c = 24;

    static int a;
    a = 51;
}
```

# Gestion de la mémoire en C++

## Stack (la pile)

- ▶ Variables **locales** (ie. automatique).
- ▶ **Frames** (valeur d'un pointeur)

## Heap (le tas)

- ▶ Variables **globales**.
- ▶ Valeurs de variables allouées **dynamiquement** par `malloc`, `calloc` et `realloc`.

## Free store

- ▶ Valeurs de variables allouées **dynamiquement** par `new`.

## Data

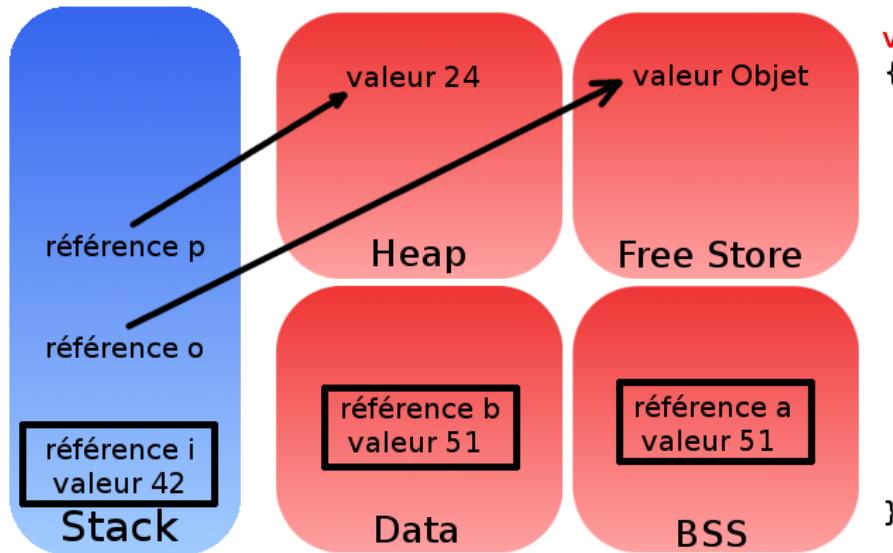
- ▶ Variables **statiques** initialisées à autre que 0.

## BSS (Block Started by Symbol)

- ▶ Autres variables **statiques** (init. à 0/NULL/nullptr ou pas init.).



# Gestion de la mémoire en C++



```
void function()
{
    int i;
    i = 42;

    Objet *o = new Objet();
    int *p =
        malloc(sizeof(int));
    *p = 24;

    static int a;
    a = 51;

    static int b = 51;
}
```

## Initialisation statique ou dynamique

Initialisation des variables **statiques** :

- ▶ Initialisation **statique** : variable initialisée lors du chargement en mémoire du programme. À lieu quand on initialise la variable avec autre chose qu'un constructeur.
- ▶ Initialisation **dynamique** : variable initialisée dynamiquement avec un constructeur.

## Cas d'une initialisation dynamique au namespace-level

En C++, l'ordre d'initialisation dynamique des variables au namespace-level se trouvant dans différents fichiers **est indéfinie**.

# Approche classique (mais incomplète)

## Singleton classique en C++

```
// Singleton.h
class Singleton
{
public:
    static Singleton* getInstance()
    {
        if( !pInstance_ )
            pInstance_ = new Singleton();
        return pInstance_;
    }

private:
    Singleton();
    static Singleton *pInstance_;
};

// Singleton.cpp
Singleton* Singleton::pInstance_ = nullptr;
```

## Initialisation de pInstance\_

pInstance\_ est initialisée sans constructeur avec une constante connue à la compilation : c'est une initialisation statique. Elle sera donc initialisée au chargement du programme.

# Erreur de simplification

Qu'est ce qui n'est pas sûr dans ce code ?

```
// Singleton.h
class Singleton
{
public:
    static Singleton* getInstance()
    {
        return &instance_;
    }

    int fonction();

private:
    Singleton();
    static Singleton instance_;
};

// Singleton.cpp
Singleton Singleton::instance_;
```

# Erreur de simplification

Initialisation dans différents fichiers au namespace-level.

```
// Singleton.cpp
#include "Singleton.h"
Singleton Singleton::instance_;

// Fichier.cpp
#include "Singleton.h"
int global = Singleton::getInstance()->fonction();
```

## Initialisation pas sûre

instance\_ sera initialisée dynamiquement par le constructeur par défaut Singleton().

## Ordre d'initialisation

On sait pas qui de global ou de instance\_ sera initialisé en premier.  
La fonction getInstance() peut potentiellement retourner un objet qui n'a pas encore été instancié.

# Assurer une instance unique

Ajout des *big three*

```
class Singleton
{
public:
    static Singleton* getInstance()
    {
        if( !pInstance_ )
            pInstance_ = new Singleton();
        return pInstance_;
    }

private:
    Singleton();
    ~Singleton(); // destructeur
    Singleton( const Singleton& ); // ctor de recopie
    Singleton& operator=( const Singleton& ); // =
    static Singleton *pInstance_;
};
```

# Programmation concurrente en C++

## Double verrou

```
class Singleton
{
public:
    static Singleton* getInstance()
    {
        if( !pInstance_ ) // 1er verrou
        {
            mutex.lock(); // pseudo-code
            if( !pInstance_ ) // 2eme verrou
                pInstance_ = new Singleton();
            mutex.unlock(); // pseudo-code
        }
        return pInstance_;
    }
private:
    ...
    static volatile Singleton *pInstance_;
};
```

Gérer la plage de vie d'un singleton

# Le problème de référence morte (dead reference)

## CEL

- ▶ Soit trois singletons : Clavier , Ecran et Log .
- ▶ On ne souhaite créer un Log que s'il y a un problème d'E/S.

## Mauvais scénario

- ➊ On instancie Clavier . Tout va bien.
- ➋ Il y a un problème avec Ecran . On crée donc Log .
- ➌ À la fin du programme, C++ désalloue dans l'ordre inverse d'instanciation (LIFO).
- ➍ Log est supprimé, mais il y a un problème lors de la fermeture de Clavier .
- ➎ Clavier souhaite écrire ce problème dans le Log , mais l'instance Log n'existe plus. Log::getInstance() est indéfini.

# Repérer une référence morte

## Dead reference check 1/3

```
// Singleton.h
class Singleton {
public:
    static Singleton* getInstance() {
        if ( destroyed_ )
            OnDeadReference();
        else if ( !pInstance_ ) {
            Create();
        }
        return pInstance_;
    }

private:
    static void Create() {
        static Singleton instance;
        pInstance_ = &instance;
    }
    ...
}
```

# Repérer une référence morte

## Dead reference check 2/3

```
private:  
    ...  
  
    static void OnDeadReference()  
    {  
        throw std::runtime_error("Dead Ref");  
    }  
  
    virtual ~Singleton()  
    {  
        pInstance_ = nullptr;  
        destroyed_ = true;  
    }  
  
    static Singleton *pInstance_;  
    static bool destroyed_;  
  
    // ctors, op d'affection  
};
```

# Repérer une référence morte

## Dead reference check 3/3

```
// Singleton.cpp
Singleton* Singleton::pInstance_ = nullptr;
bool Singleton::destroyed_ = false;
```

## Un bon début

On repère les dead references et on n'a plus un comportement indéfini.

## Vie et mort d'un singleton

- ▶ Le singleton **Phoenix**, qui renaît de ses cendres quand on l'appelle alors qu'il était mort.
- ▶ Le singleton avec une **longévité programmée**.

## Singleton Phoenix

```
class Singleton {
    ...
    static void KillPhoenixSingleton();
};

void Singleton::OnDeadReference() {
    Create(); // on choppe la carcasse du Phoenix
    new(pInstance_) Singleton; // on le ressuscite au
                                // même endroit en RAM

    atexit(KillPhoenixSingleton); // appelé à la fin
                                // du programme
    destroyed_ = false; // en vie
}

void Singleton::KillPhoenixSingleton() {
    pInstance_ ->~Singleton();
}
```

## Programmer une plage de vie

- ▶ Le singleton Phoenix est bien mais pas pour tout : les données seront perdues après une résurrection.
- ▶ Des fois, on sait jusqu'à quand on aimerait qu'un singleton dure. Par exemple, on veut un Log qui se termine **après** Clavier et Ecran .

On veut quelque chose comme ça

```
int main()
{
    ...
    SetLongevity(Clavier, 1);
    SetLongevity(Ecran, 1);
    SetLongevity(Log, 2); // Log > Clavier, Ecran
    ...
}
```

## Trop complexe pour ce cours

Nous ne ferons pas les détails d'implémentation du singleton avec longévité. Le pseudo-code sera sur Madoc pour les curieux.

## Pour ce cours

Il faut juste savoir que ça existe.

## Le dernier

Singleton est le dernier pattern de construction que nous verrons, et le dernier étudié en détails.

## La semaine prochaine

Des patterns en vrac.

- ▶ Vous déposez vos rapports dans ma boîte aux lettres (RDC bât 11, LINA, boîte rouge) au plus tard le **jeudi 27 novembre**.  
Aucun retard accepté.  
Pas de rapport = pas de présentation de projet.
- ▶ Vous déposez vos projets sur le serveur de dépôt dans la section "Projet" de ce cours sur madoc, au plus tard le **lundi 01 décembre à 23h55**.  
Aucun retard accepté (serveur de dépôt bloqué à 23h55 et 1 seconde).  
Pas de code = pas de présentation de projet.