Problème…

- Concevoir une classe (un ensemble d'objets) dont il ne doit exister qu'une seule et unique instance, accessible globalement (publique)
  - Par exemple, pour un gestionnaire de ressource (objet de connexion à une base de données, serveur d'impression…)
- Solution
  - Spécialiser ce qui concerne la création et la gestion de l'instance
    - Cacher le constructeur de la classe
    - En contrepartie, offrir une méthode qui gère l'instanciation et l'accès à l'instance
      - Une méthode et un attribut de classe
      - Il faut décider du moment de l'instanciation

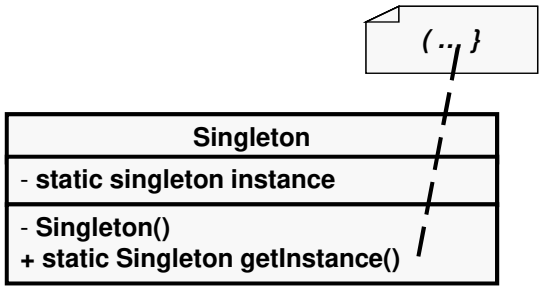
Le modèle « Singleton » (1/7)

- Nom
  - Singleton (modèle créateur)
- Alias
  - *Néant*
- Intention
  - Assurer qu'une classe n'ait qu'une instance et fournir un point d'accès global à celle-ci
- Motivation
  - Pour certaines classes, il est important de n'avoir qu'une seule instance : par exemple d'un serveur d'impression…
  - Pour cela, la classe assure l'unicité de l'instance et fournit un moyen pour y accéder
- Indications d'utilisation
  - S'il ne doit y avoir qu'une instance au plus de la classe

Le modèle « Singleton » (2/7)

- Constituants (ou participants)
  - Une seule classe
    - La classe elle-même et elle seule
  - Cas d'exception (c'est le seul design pattern parmi ceux du GoF)

- Structure



- Collaborations
  - Les clients accèdent à l'instance par le seul intermédiaire de la méthode (possiblement « synchronisée ») *getInstance()*

Le modèle « Singleton » (3/7)

```
public class MonSingleton {
   // l'unique instance
   private static MonSingleton instance = null;
   // le constructeur privé
   private MonSingleton(){
   }
   // méthode (de classe) pour la création d'instance
   public static MonSingleton getInstance() {
       // création « paresseuse de l'instance »
       if (instance == null) {
             instance = new MonSingleton();
       }
       return instance;
   }
}
```

Le modèle « Singleton » (4/7)

- Conséquences
  - La classe elle-même contrôle précisément comment et quand les clients accèdent à l'instance
    - Si l'instance doit être crée systématiquement, on peut la créer au chargement de la classe (simplification !)
  - Le modèle peut être adapté pour contrôler un nombre fixé d'instances
    - Allocation/gestion de « pools » d'objets
  - On peut sous-classer la classe Singleton et préserver le polymorphisme
  - …

Le modèle « Singleton » (5/7)

- Pour aller un peu plus loin…
  - Si on se place dans le contexte de la programmation concurrente (c'est-à-dire « multi-thread ») ?
    - C'est-à-dire si plusieurs accès différents et simultanés (concurrents) sont possibles ?
  - Il faut contrôler l'accès à la méthode getInstance()
    - En Java, la méthode getInstance() doit être « synchronized »

$\Longrightarrow$ Il faut adapter la solution (d'où l'importance du contexte !)

Le modèle « Singleton » (6/7)

```
public class MonSingleton {
   // l'unique instance
   private static MonSingleton instance = null;
   // le constructeur privé
   private MonSingleton(){
   }
   // méthode (de classe) pour la création d'instance
   public static synchronized MonSingleton getInstance() {
       // création « paresseuse de l'instance »
       if (instance == null) {
             instance = new MonSingleton();
       }
       return instance;
   }
}
```


Le modèle « Singleton » (7/7)

- Mise en œuvre dans un contexte *multi-thread*
  - Utilisation de « *synchronized* »
    - Mais surcoût à l'exécution !
  - Au besoin, on peut déplacer le verrouillage dans le corps de la méthode
    - Verrouillage seulement quand l'instance n'existe pas
    - Double vérification (mais, en pratique, ça ne suffit pas, cf. « Tête La première, Design Patterns » p. 182)

```
if (instance == null) {
      synchronized (MonSingleton.class) {
            if (instance == null) {
                  instance = new MonSingleton();
            }
      }
}
```

Tous ces éléments documentent le pattern