Singleton Pattern

Pattern Name: Singleton Pattern

Context

We want to ensure there is only **one instance of a class**.
All parts of the application should share this single instance.

Motivation (Forces)

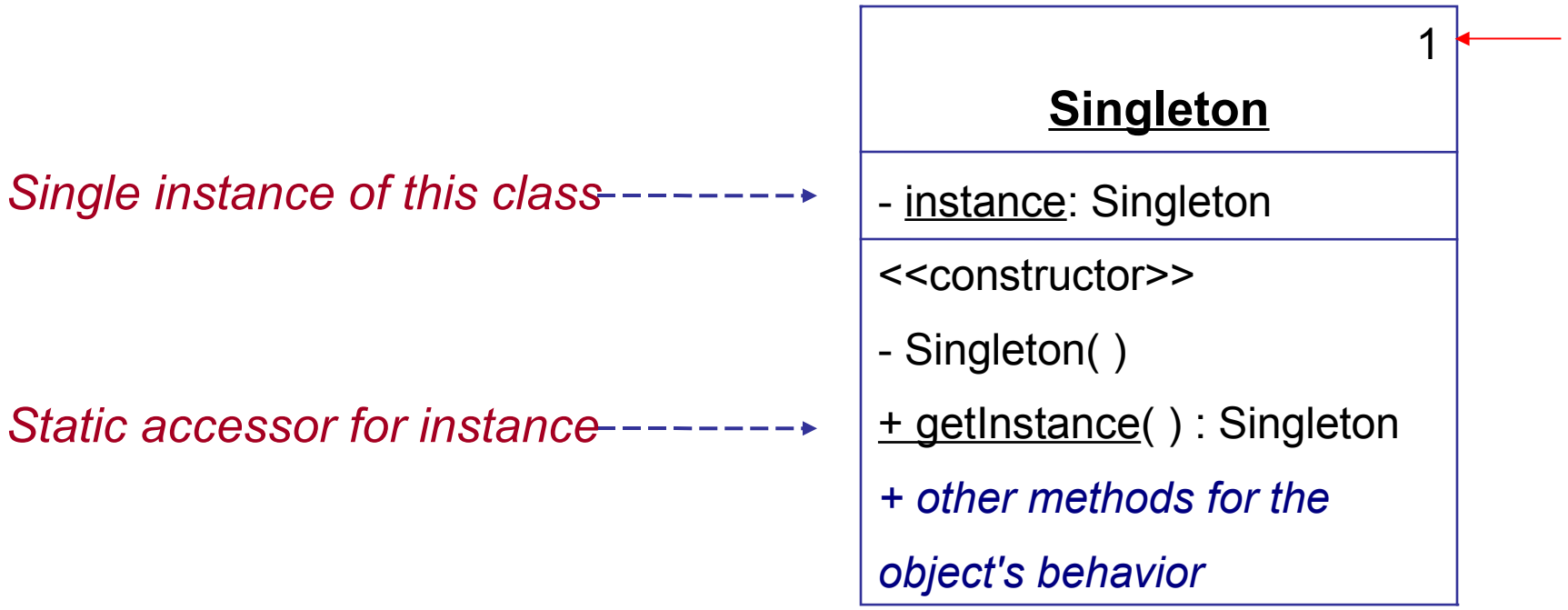
Several objects need to access the same resource, or we want objects to share a resource that is "expensive". Many parts of the program need to access this shared resource.

Solution

Prevent direct instantiation by making the constructor private.

Provide a static accessor method that always returns the same instance of this class (same object).

Singleton Pattern



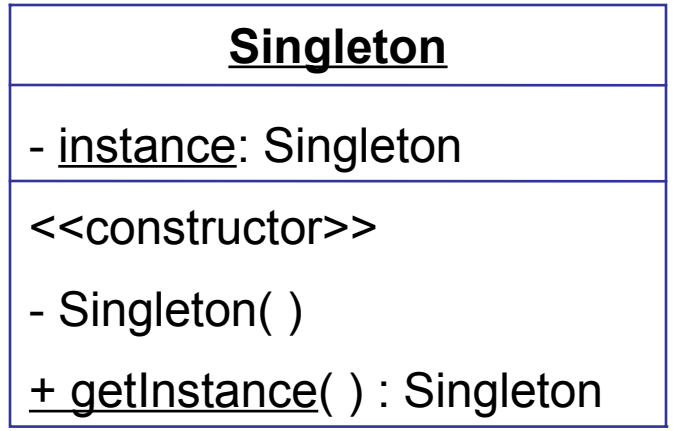
Singleton Pattern

Singleton has 3 elements:

(1) private static attribute that is the only instance of this class

(2) constructor is private to prevent other classes from creating objects

(3) public static accessor returns the single instance of this class.



Example of Singleton Pattern

A Store that has only one instance.

```
public class Store {
// (1) the single static instance
   private static Store theStore = null;
   private List<Transaction> transactions;
// (2) private constructor
   private Store( ) {
      transactions = new ArrayList<Transaction>( );
   }
// (3) static accessor method also creates singleton
public static Store getInstance() {
      if ( theStore == null ) theStore = new Store( );
      return theStore;
   }
```

lazy instantiation

Lazy Instantiation

Means that you create a resource only when it is needed.

This avoids creating something that may never be used.

```
// (3) static accessor method creates the singleton
public static Store getInstance() {
   if ( theStore == null ) theStore = new Store();
   return theStore;
}
```

The store instance is created the first time that getInstance( ) is called, but not before.

If getInstance is never called, no Store is created.

Getting the Singleton object

How do other objects get the Store?

```
// in your application use:


Store store = Store.getInstance( );
```

Lazy Instantiation of Loggers

Using Log4J you will see a lot of code like this:

```
// Create the logger for this class
private static Logger log = Logger.getLogger(...);
```

What if this class never logs any messages?

We wasted time and memory creating the logger.

So many apps use *lazy* instantiation:

```
// Don't create logger yet
private static Logger log = null;

private static Logger getLogger() {
   if (log == null) log = Logger.getLogger(...);
   return log;
}
```

Eager Instantiation

Eager instantiation means to create the object as early as possible.

Eager instantiation is used in cases such as:

- you want objects created during start-up, either so the application will "fail early" if object can't be created, or to avoid delay while app is running (e.g. a game needs to create a bunch of sprites while running).

```
public class Store {
   // eager instantiation: create instance when
   // the class is loaded into memory.
   private static final Store theStore =
                        new Store();
```

Consequences of Using Singleton

Benefits

- control access to a single instance
- reduce name space pollution - better than using a global variable (in languages with global variables)
- permits a variable number of instances - you can modify the singleton to produce more than one instance, w/o changing other parts of application

Disadvantages

- Singleton cannot be subclassed, since the constructor is private and static getInstance() is not polymorphic.

Related patterns

- Factory Method