## Singleton

Singleton – jeden ze wzorców konstrukcyjnych. Jego celem jest zapewnienie możliwości utworzenia tylko jednego obiektu danej klasy i zapewnienie do niego globalnego dostępu.

Przez wielu uważany za antywzorzec, głównie ze względu na ograniczoną możliwość testowania kodu. Pomijając kwestie słuszności jego wykorzystania przedstawię jego poprawną implementację, co nie jest trywialne jakby mogło się wydawać.

Klasa z implementacją singletona zawiera:

* statyczną zmienną przechowującą instancję tej klasy,
* prywatny „pusty” konstruktor (aby nie można było utworzyć nowego obiektu tej klasy z wykorzystaniem operatora „new”),
* publiczną statyczną metodę „getInstance” bez żadnych parametrów.

**Klasyczna implementacja (błędna, nieobsługiwana wielowątkowość).**

*public class Singleton {*

*private Singleton() {}*

*public static Singleton getInstance() {*

*return SingletonHolder.INSTANCE;*

*}*

*private static class SingletonHolder {*

*private static final Singleton INSTANCE = new Singleton();*

*}*

*}*

W przypadku wielowątkowego dostępu problem jest w liniach 10 i 11. Przykład:

wątek A sprawdza czy instance == null (prawda)

w tym samym czasie wątek B również sprawdza ten warunek przed utworzeniem obiektu przez wątek A w linii 11

wątek B tworzy drugi obiekt singletona.

**Implementacja z synchronizowanym dostępem**

*public class Singleton {*

*private static Singleton instance;*

*private Singleton() {}*

*public static synchronized Singleton getInstance() {*

*if (instance == null) {*

*instance = new Singleton();*

*}*

*return instance;*

*}*

*}*

Ta implementacja jest poprawna, ale kosztowna, a synchronizacja dostępu do metody konieczna jest tylko podczas pierwszego użycia tej metody. Dlatego lepszą wersją synchronizowanego dostępu będzie kolejna implementacja.

**Implementacja z ‚,podwójnym blokowaniem”**

*public class Singleton {*

*private volatile static Singleton instance;*

*private Singleton() {}*

*public static Singleton getInstance() {*

*if (instance == null) {*

*synchronized (Singleton.class) {*

*if (instance == null) {*

*instance = new Singleton();*

*}*

*}*

*}*

*return instance;*

*}*

*}*

W tej implementacji synchronizacja ma miejsce tylko przy pierwszym wywołaniu metody. Jeżeli dwa wątki równocześnie wywołają metodę getInstance i pierwszy warunek (instance == null) zostanie spełniony przez nie oba, to w kolejnej linii (synchronized (Singleton.class)) dostęp do ponownego sprawdzenia warunku (instance == null) zostanie zsynchronizowany. W efekcie czego jeden z wątków utworzy nową instancję, a drugi już go pobierze nie wchodząc w warunek z linii 12.  
Kluczową kwestią jest tu słowo kluczowe **volatile**. Powoduje ono, że zmienna instance będzie prawidłowo traktowana kiedy następuje przypisywanie do niej nowo utworzonej instancji obiektu (w programie wielowątkowym). Przykład:

wątek A zauważa niezainicjowaną zmienną, uzyskuje blokadę i rozpoczyna inicjalizować zmienną

środowisko może zaktualizować współdzieloną zmienną częściowo utworzonego obiektu zanim wątek A zakończy inicjalizację

wątek B zauważa że zmienna została zainicjowana i pomija synchronizowany blok

Jest to niuans, który może doprowadzić do poważnych błędów, a ich wykrycie jest bardzo trudne.

**Implementacja z prywatną klasą statyczną**

*public class Singleton {*

*private Singleton() {*

*}*

*public static Singleton getInstance() {*

*return SingletonHolder.INSTANCE;*

*}*

*private static class SingletonHolder {*

*private static final Singleton INSTANCE = new Singleton();*

*}*

*}*

Jest to najlepsza implementacja singletona. Należy zwrócić uwagę, że gdy klasa Singleton zostanie załadowana, klasa SingletonHolder nie zostanie wczytana do pamięci dopóki nie zostanie wywołana metoda getInstance(). Takie podejście nie wymaga synchronizacji, jest łatwe do zrozumienia i zaimplementowania.