Singleton（シングルトン）

❖ 概要

Singleton は，1 個しかインスタンスが存在し得ないクラスを定義するためのパターンです。

一般に，あるクラスのインスタンスは，生成のためのコンストラクタ呼び出しを行うことでその都度任意個数の生成が可能です。シングルトンでは，1 個だけインスタンスを生成して，その同一のインスタンスが常に使用されるようなパターンを提供します。

❖ 適用範囲

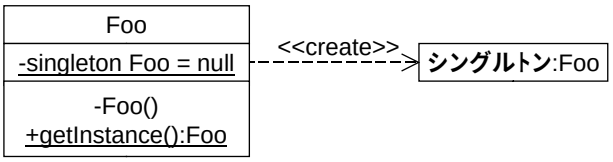
システムの環境変数保持オブジェクトなど，適用範囲は多岐にわたります。

記述がクラス 1 個の中で完結しているので，必要ならばどのようなクラスにでも適用することができます。

ただし，シングルトンパターンは static 変数を保持するクラスの領域が仮想マシン単位で単一領域であることを利用しているため，複数の仮想マシンを使用している場合には適用できません。

❖ 構造

「シングルトン」パターンにはクラスが 1 個しか登場しません。必要なことは，すべてそのクラス内で定義されます。



❖ 考えの進め方

① Foo クラスに, Foo クラスのインスタンス自身を格納できる static 変数(属性)を private で 1 個定義する(例では変数 singleton)。クラスは仮想マシン内で一意の存在になるので，static 変数 singleton が指し示すオブジェクトもシステム内で一意のものになる。

② 他から呼ばれないよう，Foo クラスのコンストラクタを private(または protected)に定義する。

③ Foo クラスに，static 変数を返すための static 関数(操作)を public で定義する(下の例では getInstance())。この関数は，static 変数が非ヌルであれば，それを返して終了する。ヌルであった場合には，インスタンスを生成して，それを static 変数に束縛したうえで返す。

④ Foo クラスを利用する側では，Foo.getInstance()を評価すれば，Foo の唯一のインスタンスが常に獲得できる。

❖ 同期化の必要性について

シングルトンの基本原理は前記の通りシンプルなものなのですが，厳密には上記の実装だけでは不十分です。Java はマルチスレッドで動作しているため，複数のスレッドが同時にメッセージ getInstance()でインスタンスを獲得しようとした場合に，別々のインスタンスを返してしまう可能性があるからです。すなわち，

```
public static Foo getInstance()
{
      if (singleton = = null)                  //(1)
      {
            singleton = new Singleton();       //(2)
      }
      return singleton;                        //(3)
}
```

のようなコードが同時に複数スレッドから呼ばれた場合，変数 singleton は(2)の評価後に初めて非ヌルになるため，同時に呼ばれた(1)は双方ともこの if 文を実行してしまいます。その結果として別々の(2)(3)が実行され，異なるインスタンスを返してしまう可能性があるからです。

これを排除するためには，getInstance()を同期化してください。すなわち，

```
public static synchronized Foo getInstance()
```

と synchronized 修飾を伴ったメソッドの宣言を行います(本文は同じです)。

これにより複数の getInstance()が同時に処理されることはなくなるので，インスタンス生成は最初の 1 回で確実に行われ，2 回目以降はそれが使われることになります。

❖ 同期化によるパフォーマンス低下を問題にする場合

同期化は「メソッドの実行が終了するまで同じメソッドは再度呼ばれないようにする」仕組みである以上，その結果として後のメソッドは単純に「待ち」が発生します。この点から，あちこちから頻繁に獲得する必要があるシングルトンのインスタンスを同期化メソッド getInstance()で呼ぶ場合に，パフォーマンス低下が問題になるかもしれません。

その場合には，メソッド getInstance()の同期化を止め，代わりにシングルトンのインスタンスは static 変数の初期化時に行うようにします。具体的には，次のようにすれば OK です。

```
public class Foo
{
      private static Foo singleton = new Foo();//ここで生成

      private Foo()
      {
          //インスタンス生成時の初期化処理
      }
```

```
    public static Foo getInstance()
    {
        return singleton;
    }
}
```

ただし static 変数の初期化時の処理は制御が困難で，またデバッグをいささか困難にする可能性があります。複数のシングルトンの生成順序を問題にするような場合には，この方法は選択できないかもしれません。

なお同期化を避ける他の方法として，double-checked locking や volatile を使用する方法というのもかつて考案されましたが，これらは適切ではないことが分かっています。

(参考)

http://www-06.ibm.com/jp/developerworks/java/020726/j_j-dcl.html

❖ 演習問題

1) 社内システムのサーバ構築をする際に，シングルトンで扱うべきオブジェクトを挙げよ。

2) 次のシングルトン生成コードのうち，直すべき点を 2 つ指摘せよ。

```
/**
 * シングルトンクラス
 */
public class Foo {

    /**
     * シングルトン変数
     */
    private static Foo singleton = null;

    /**
```

```
     * シングルトン獲得
     * @return シングルトン
     */
    public static Foo getInstancen() {
        if (singleton == null) {
            singleton = new Foo();
        }
        return singleton;
    }

    /**
     * コンストラクタ
     */
    pubic Foo() {
        super();
    }

}
```

◆ 解答例

1）

システム変数管理オブジェクト

マスタオブジェクトのリストを保持するオブジェクト

など

2）

1：コンストラクタは public ではなく，private(または protected)にする。

2：getInstance()に synchronized 修飾子を付ける。

※static 変数宣言時にインスタンス生成をする，でもよいが，その場合

メソッド getInstance()の if 節は全部不要になる。

Prototype（プロトタイプ）

❖ 概要

あるクラスのインスタンスを，コンストラクタ呼び出しではなく，すでに存在するインスタンスから複製して生成するパターンです。

「浅い複製」の問題を常に意識する必要があります。

❖ 適用範囲

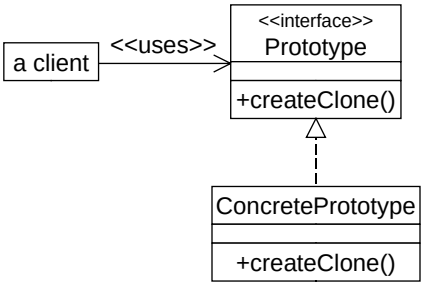
インスタンスを「複製」により生成することにメリットが生まれる場合。

例えば，あるインスタンスをユーザーが画面のエディタにより修正したものと同等のインスタンスを獲得したいような場合に，そのインスタンスから生成パラメータを獲得してコンストラクタに渡すよりも素直な実装になる可能性があります。

❖ 構造

「プロトタイプ」パターンは，複製メソッドを共通化できるという意味で interface を示してありますが, 基本的にはインスタンスを複製するクラス 1 個で機能します。

Proptotype インタフェースを省略してしまっても構わないでしょう。


<<interface>>
Prototype
+createClone()
a client
<<uses>>
ConcretePrototype
+createClone()