



[🏠](#) / [Patrones de diseño](#) / [Singleton](#) / [PHP](#)



Singleton en PHP

Singleton es un patrón de diseño creacional que garantiza que tan solo exista un objeto de su tipo y proporciona un único punto de acceso a él para cualquier otro código.

El patrón tiene prácticamente los mismos pros y contras que las variables globales. Aunque son muy útiles, rompen la modularidad de tu código.

No se puede utilizar una clase que dependa del Singleton en otro contexto. Tendrás que llevar también la clase Singleton. La mayoría de las veces, esta limitación aparece durante la creación de pruebas de unidad.

 [Aprende más sobre el patrón Singleton →](#)

Navegación

 [Intro](#)

 [Ejemplo conceptual](#)

 [index](#)

 [Output](#)

 [Ejemplo del mundo real](#)

 [index](#)

 [Output](#)

Complejidad: ★☆☆

Popularidad: ★★☆☆



Identificación: El patrón Singleton se puede reconocer por un método de creación estático, que devuelve el mismo objeto guardado en caché.

Ejemplo conceptual

Este ejemplo ilustra la estructura del patrón de diseño **Singleton** y se centra en las siguientes preguntas:

- ¿De qué clases se compone?
- ¿Qué papeles juegan esas clases?
- ¿De qué forma se relacionan los elementos del patrón?

Después de conocer la estructura del patrón, será más fácil comprender el siguiente ejemplo basado en un caso de uso real de PHP.

index.php: Ejemplo conceptual

```
<?php
```

```
namespace RefactoringGuru\Singleton\Conceptual;
```

```
/**
```

```
 * The Singleton class defines the `GetInstance` method that serves as an
 * alternative to constructor and lets clients access the same instance of this
 * class over and over.
```

```
*/
```

```
class Singleton
```

```
{
```

```
    /**
```

```
     * The Singleton's instance is stored in a static field. This field is an
     * array, because we'll allow our Singleton to have subclasses. Each item in
     * this array will be an instance of a specific Singleton's subclass. You'll
     * see how this works in a moment.
```

```
    */
```

```
    private static $instances = [];
```

```
    /**
```

```
     * The Singleton's constructor should always be private to prevent direct
     * construction calls with the `new` operator.
```

```
    */
```



```
/**
 * Singletons should not be cloneable.
 */
protected function __clone() { }

/**
 * Singletons should not be restorable from strings.
 */
public function __wakeup()
{
    throw new \Exception("Cannot unserialize a singleton.");
}

/**
 * This is the static method that controls the access to the singleton
 * instance. On the first run, it creates a singleton object and places it
 * into the static field. On subsequent runs, it returns the client existing
 * object stored in the static field.
 *
 * This implementation lets you subclass the Singleton class while keeping
 * just one instance of each subclass around.
 */
public static function getInstance(): Singleton
{
    $cls = static::class;
    if (!isset(self::$instances[$cls])) {
        self::$instances[$cls] = new static();
    }

    return self::$instances[$cls];
}

/**
 * Finally, any singleton should define some business logic, which can be
 * executed on its instance.
 */
public function someBusinessLogic()
{
    // ...
}
}

/**
 * The client code.
 */
function clientCode()
{
    $s1 = Singleton::getInstance();
    $s2 = Singleton::getInstance();
    if ($s1 === $s2) {
```



```
        echo "Singleton failed, variables contain different instances.";
    }
}

clientCode();
```

Output.txt: Resultado de la ejecución

Singleton works, both variables contain the same instance.

Ejemplo del mundo real

El patrón **Singleton** es famoso por limitar la reutilización de código y complicar las pruebas de unidad. No obstante, sigue resultando muy útil en algunos casos. En particular, viene bien cuando debes controlar recursos compartidos. Por ejemplo, un objeto de registro global debe controlar el acceso a un archivo de registro. Otro buen ejemplo: un almacenamiento compartido de la configuración de tiempo de ejecución.

index.php: Ejemplo del mundo real

```
<?php

namespace RefactoringGuru\Singleton\RealWorld;

/**
 * If you need to support several types of Singletons in your app, you can
 * define the basic features of the Singleton in a base class, while moving the
 * actual business logic (like logging) to subclasses.
 */
class Singleton
{
    /**
     * The actual singleton's instance almost always resides inside a static
     * field. In this case, the static field is an array, where each subclass of
     * the Singleton stores its own instance.
     */
    private static $instances = [];

    /**
     * Singleton's constructor should not be public. However, it can't be
     * private either if we want to allow subclassing.
```



```
/**
 * Cloning and unserialization are not permitted for singletons.
 */
protected function __clone() { }

public function __wakeup()
{
    throw new \Exception("Cannot unserialize singleton");
}

/**
 * The method you use to get the Singleton's instance.
 */
public static function getInstance()
{
    $subclass = static::class;
    if (!isset(self::$instances[$subclass])) {
        // Note that here we use the "static" keyword instead of the actual
        // class name. In this context, the "static" keyword means "the name
        // of the current class". That detail is important because when the
        // method is called on the subclass, we want an instance of that
        // subclass to be created here.

        self::$instances[$subclass] = new static();
    }
    return self::$instances[$subclass];
}

/**
 * The logging class is the most known and praised use of the Singleton pattern.
 * In most cases, you need a single logging object that writes to a single log
 * file (control over shared resource). You also need a convenient way to access
 * that instance from any context of your app (global access point).
 */
class Logger extends Singleton
{
    /**
     * A file pointer resource of the log file.
     */
    private $fileHandle;

    /**
     * Since the Singleton's constructor is called only once, just a single file
     * resource is opened at all times.
     *
     * Note, for the sake of simplicity, we open the console stream instead of
     * the actual file here.
     */
}
```



```
$this->fileHandle = fopen('php://stdout', 'w');
}

/**
 * Write a log entry to the opened file resource.
 */
public function writeLog(string $message): void
{
    $date = date('Y-m-d');
    fwrite($this->fileHandle, "$date: $message\n");
}

/**
 * Just a handy shortcut to reduce the amount of code needed to log messages
 * from the client code.
 */
public static function log(string $message): void
{
    $logger = static::getInstance();
    $logger->writeLog($message);
}
}

/**
 * Applying the Singleton pattern to the configuration storage is also a common
 * practice. Often you need to access application configurations from a lot of
 * different places of the program. Singleton gives you that comfort.
 */
class Config extends Singleton
{
    private $hashmap = [];

    public function getValue(string $key): string
    {
        return $this->hashmap[$key];
    }

    public function setValue(string $key, string $value): void
    {
        $this->hashmap[$key] = $value;
    }
}

/**
 * The client code.
 */
Logger::log("Started!");

// Compare values of Logger singleton.
$l1 = Logger::getInstance();
```



```
    Logger::log("Logger has a single instance.");
} else {
    Logger::log("Loggers are different.");
}

// Check how Config singleton saves data...
$config1 = Config::getInstance();
$login = "test_login";
$password = "test_password";
$config1->setValue("login", $login);
$config1->setValue("password", $password);
// ...and restores it.
$config2 = Config::getInstance();
if ($login == $config2->getValue("login") &&
    $password == $config2->getValue("password"))
{
    Logger::log("Config singleton also works fine.");
}

Logger::log("Finished!");
```

Output.txt: Resultado de la ejecución

```
2018-06-04: Started!
2018-06-04: Logger has a single instance.
2018-06-04: Config singleton also works fine.
2018-06-04: Finished!
```

Ejemplo conceptual

Ejemplo del mundo real

LEER SIGUIENTE

Inicio

Refactorización

Patrones de diseño

Contenido Premium

Foro

Contáctanos







REBAJA DE VERANO




© 2014-2023 Refactoring.Guru. Todos los derechos reservados

 Ilustraciones por Dmitry Zhart

Ukrainian office:


 FOP Olga Skobeleva


 Abolmasova 7

Kyiv, Ukraine, 02002

 Email: support@refactoring.guru

Spanish office:

 Oleksandr Shvets

 Avda Pamplona 63, 4b

Pamplona, Spain, 31010

 Email: spain@refactoring.guru

[Términos y condiciones](#)

[Política de privacidad](#)

[Política de uso de contenido](#)

[About us](#)

