The singleton pattern::the good, the bad, and the ugly

Published July 05, 2015

We use the **singleton pattern** in order to restrict the number of instances that can be created from a resource consuming class to only one.

Resource consuming classes are classes that might slow down our website or cost money. For example:

* Some external service providers (APIs) charge money per each use.
* Some classes that detect mobile devices might slow down our website.
* Establishing a connection with a database is time consuming and slows down our app.

So, in all of these cases, it is a good idea to restrict the number of objects that we create from the expensive class to only one.



Use a **singleton pattern** to restrict the number of objects to only one!

The anatomy of a singleton pattern

Let's start by understanding the structural characteristics of a class that obeys the **singleton pattern**:

1. A **private constructor** is used to prevent the direct creation of objects from the class.
2. The expensive process is performed within the private constructor.
3. The only way to create an instance from the class is by using a static method that creates the object only if it wasn't already created.

// General singleton class.

class Singleton {

  // Hold the class instance.

  private static $instance = null;

  // The constructor is private

  // to prevent initiation with outer code.

  private function \_\_construct()

  {

    // The expensive process (e.g.,db connection) goes here.

  }

  // The object is created from within the class itself

  // only if the class has no instance.

  public static function getInstance()

  {

    if (self::$instance == null)

    {

      self::$instance = new Singleton();

    }

    return self::$instance;

  }

}

Why is it a singleton?

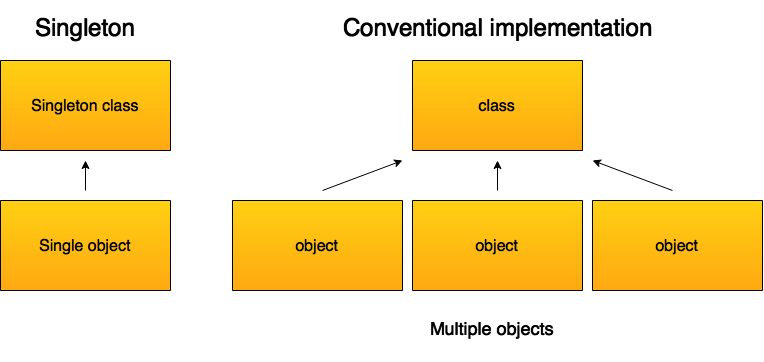
Since we restrict the number of objects that can be created from a class to only one, we end up with all the variables pointing to the same, single object.

// All the variables point to the same object.

$object1 = Singleton::getInstance();

$object2 = Singleton::getInstance();

$object3 = Singleton::getInstance();



Practical example::database class

Let's demonstrate the **singleton pattern** with a class that establishes a database connection, and restricts the number of instances to only one.

// Singleton to connect db.

class ConnectDb {

  // Hold the class instance.

  private static $instance = null;

  private $conn;

  private $host = 'localhost';

  private $user = 'db user-name';

  private $pass = 'db password';

  private $name = 'db name';

  // The db connection is established in the private constructor.

  private function \_\_construct()

  {

    $this->conn = new PDO("mysql:host={$this->host};

    dbname={$this->name}", $this->user,$this->pass,

    array(PDO::MYSQL\_ATTR\_INIT\_COMMAND => "SET NAMES 'utf8'"));

  }

  public static function getInstance()

  {

    if(!self::$instance)

    {

      self::$instance = new ConnectDb();

    }

    return self::$instance;

  }

  public function getConnection()

  {

    return $this->conn;

  }

}

Since we use a class that checks if a connection already exists before it establishes a new one, it really doesn't matter how many times we create a new object out of the class, we still get the same connection. To prove the point, let's create three instances out of the class and var dump them.

$instance = ConnectDb::getInstance();

$conn = $instance->getConnection();

var\_dump($conn);

$instance = ConnectDb::getInstance();

$conn = $instance->getConnection();

var\_dump($conn);

$instance = ConnectDb::getInstance();

$conn = $instance->getConnection();

var\_dump($conn);

The result is the same connection for the three instances.

Class that doesn't use a singleton to contact the database

To understand the problem that the singleton pattern solves, let's consider the following class that has no mechanism to check if a connection already exists before it establishes a new connection.

// Connect db without a singleton.

class ConnectDbWOSingleton {

  private $conn;

  private $host = 'localhost';

  private $user = 'db user-name';

  private $pass = 'db password';

  private $name = 'db name';

  // Public constructor.

  public function \_\_construct()

  {

    $this->conn = new PDO("mysql:host={$this->host};

    dbname={$this->name}", $this->user,$this->pass,

    array(PDO::MYSQL\_ATTR\_INIT\_COMMAND => "SET NAMES 'utf8'"));

  }

  public function getConnection()

  {

    return $this->conn;

  }

}

Now, each time we create a new object, we also establish a new database connection.

$instance = new ConnectDbWOSingleton();

$conn = $instance->getConnection();

var\_dump($conn);

$instance = new ConnectDbWOSingleton();

$conn = $instance->getConnection();

var\_dump($conn);

$instance = new ConnectDbWOSingleton();

$conn = $instance->getConnection();

var\_dump($conn);

This has implications for slowing down the system because each new connection with the database costs time.

I encourage you to run the code to see the result for yourself.

The singleton pattern::the good, the bad, and the ugly

The **singleton pattern** is probably the most infamous pattern to exist, and is considered an **anti-pattern** because it creates global variables that can be accessed and changed from anywhere in the code.

Yet, The use of the **singleton pattern** is justified in those cases where we want to restrict the number of instances that we create from a class in order to save the system resources. Such cases include data base connections as well as external APIs that devour our system resources.