**What is singleton design pattern?**

A singleton is a particular kind of class that, as you correctly said, can be instantiated only once.

First point: it isn't a PHP related concept but an OOP concept.

What "instantiated only once means?" It simply means that if an object of that class was already instantiated, the system will return it instead of creating new one. Why? Because, sometimes, you need a "common" instance (global one) or because instantiating a "copy" of an already existent object is useless.

Let's consider for first case a framework: on bootstrap operation you need to instantiate an object but you can (you have to) share it with other that request for a framework bootstrap.

For the second case let's consider a class that has only methods and no members (so basically no internal state). Maybe you could implement it as a static class, but if you want to follow design patterns, consider AbstractFactory) you should use objects. So, having some copy of the same object that has only methods isn't necessary and is also memory-wasting.

Those are two main reason to use singleton to me.

<?php

class DBConn {

private static $obj;

private final function \_\_construct() {

echo \_\_CLASS\_\_ . " initializes only once\n";

}

public static function getConn() {

if(!isset(self::$obj)) {

self::$obj = new DBConn();

}

return self::$obj;

}

}

$obj1 = DBConn::getConn();

$obj2 = DBConn::getConn();

var\_dump($obj1 == $obj2);

?>

**What is Inversion of Control?**

The Inversion of Control (IoC) and Dependency Injection (DI) patterns are all about removing dependencies from your code.

For example, say your application has a text editor component and you want to provide spell checking. Your standard code would look something like this:

public class TextEditor {

private SpellChecker checker;

public TextEditor() {

this.checker = new SpellChecker();

}

}

What we've done here creates a dependency between the TextEditor and the SpellChecker. In an IoC scenario we would instead do something like this:

public class TextEditor {

private IocSpellChecker checker;

public TextEditor(IocSpellChecker checker) {

this.checker = checker;

}

}

In the first code example we are instantiating SpellChecker (this.checker = new SpellChecker();), which means the TextEditor class directly depends on the SpellChecker class.

In the second code example we are creating an abstraction by having the SpellChecker dependency class in TextEditor constructor signature (not initializing dependency in class). This allows us to call the dependency then pass it to the TextEditor class like so:

SpellChecker sc = new SpellChecker; // dependency

TextEditor textEditor = new TextEditor(sc);

Now the client creating the TextEditor class has the control over which SpellChecker implementation to use because we're injecting the dependency to the TextEditor signature.

This is just a simple example, there's a good series of articles by Simone Busoli that explains it in greater detail.