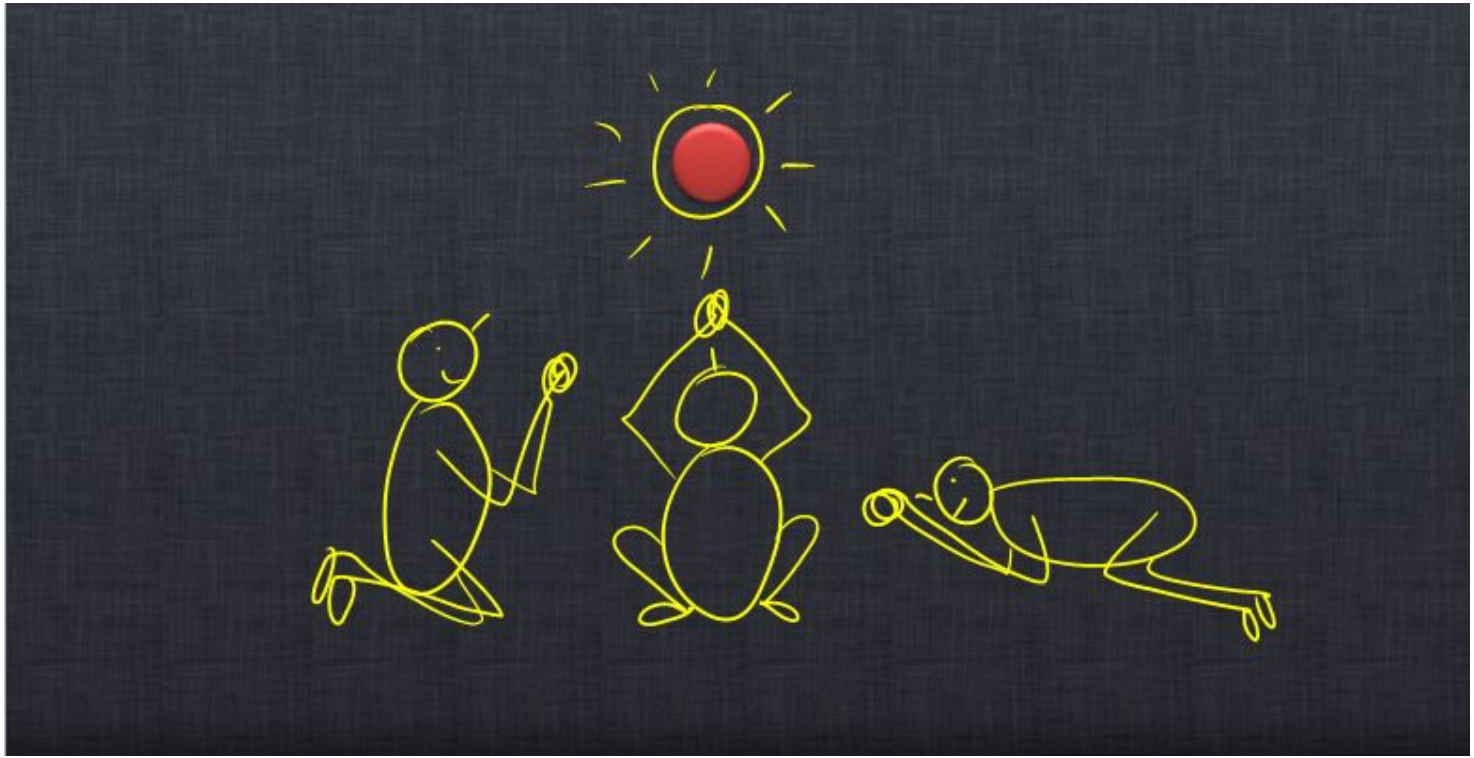
Design Patterns

Singleton , Dependency Injection

Singleton



Singleton

Definition:

a design pattern used to implement the mathematical concept of a singleton, by restricting the instantiation of a class to one object.

Usability:

- coordinate actions across the system
- systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects.

Singleton

Basic principle:

- private constructor

- special instance accessor

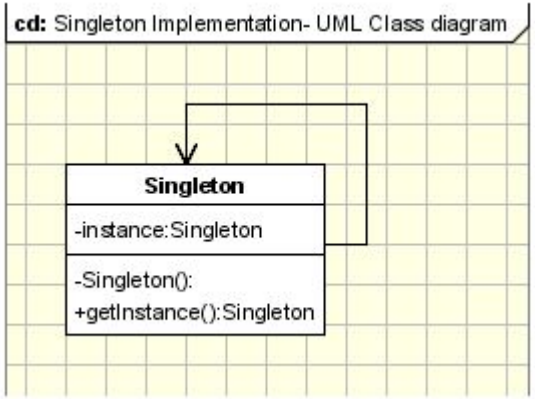
Γιατί όχι static instance;

(a) lazily constructed, requiring no memory or resources until needed

(b) static methods cannot be overriden

(b) static member classes cannot implement an interface (e.g in C#) or they can do it under special circumstances

Singleton


cd: Singleton Implementation- UML Class diagram
Singleton
-instance:Singleton
-Singleton():
+getInstance():Singleton

PHP (i): Basic structure

```
class MySingleton {

    protected static $instance;

    protected function __construct() { }

    public static function getInstance() {
        if (!isset(self::$instance)) {
            self::$instance = new self;
        }
            return self::$instance;
    }

}
```

PHP (ii): Prevent indirect instantiation

```
class MySingleton {
    protected static $instance;
    protected function __construct() { }
    protected function __clone() { }
    private function __wakeup() { }

    public static function getInstance() {
        if (!isset(self::$instance)) {
            self::$instance = new self;
        }
        return self::$instance;
    }
}
```

Java (i): Lazy initialization

```
public class Singleton {
        private static Singleton _instance;

        private Singleton() {  }

        public static synchronized Singleton getInstance() {
                if (null == _instance) {
                        _instance = new Singleton();
                }
                return _instance;
        }
}
```

Java (ii): Early initialization

```
public class Singleton {
        private static final Singleton instance = new Singleton();

        private Singleton() { }

        public static Singleton getInstance() {
                return instance;
        }
}
```

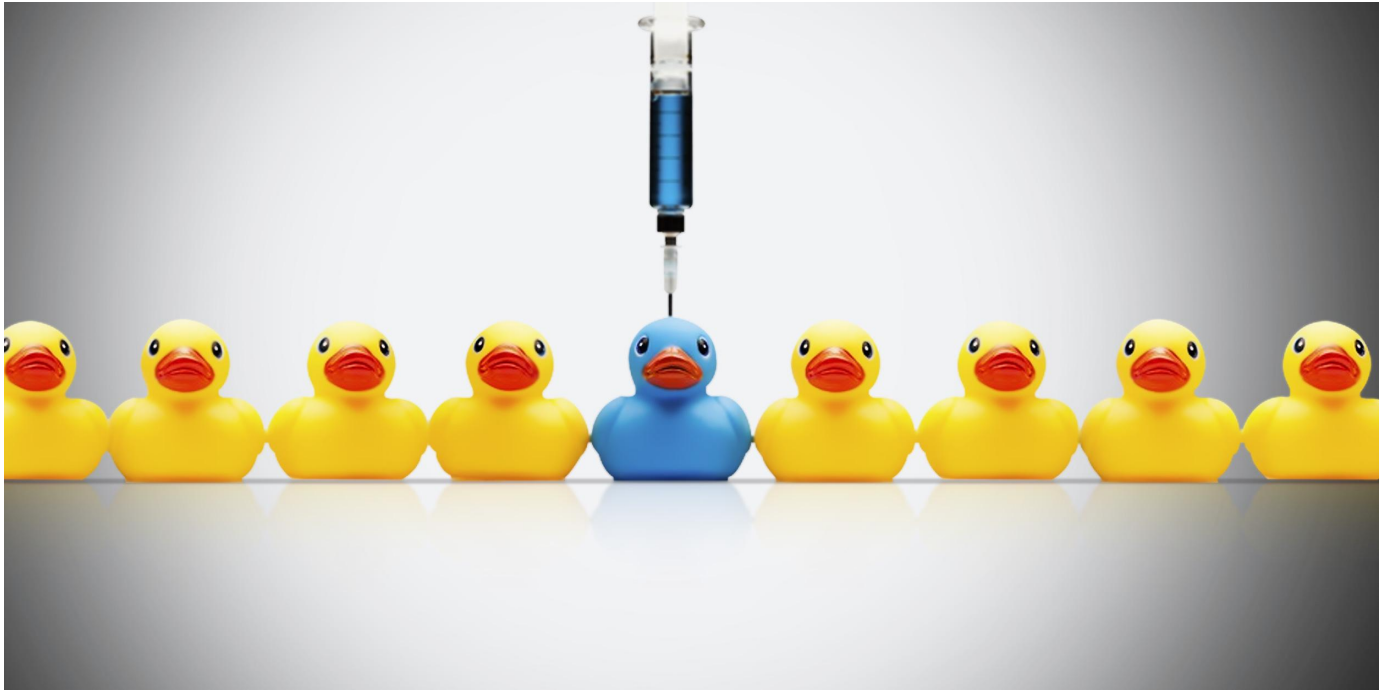
Java (iii): Bill Pugh's method

```
public class Singleton {
        // Private constructor prevents instantiation from other classes
        private Singleton() { }

        // SingletonHolder is loaded on the first execution of Singleton.getInstance()
        // or the first access to SingletonHolder.INSTANCE, not before.
        private static class SingletonHolder {
                public static final Singleton instance = new Singleton();
        }

        public static Singleton getInstance() {
                return SingletonHolder.instance;
        }
}
```

Java (iv): What is more?

- `Singleton as Enum`
- `Cloning ?`

Dependency Injection



Dependency Injection - Definition

DI is a Design Pattern commonly used to implement
the Inversion of Control Principle in software development.

Example (a) - Dependency

```
class Car {
    public function run() {
        echo 'Vroooaaammmm!';
    }
}

class Driver {
    private $car;
    public function __construct() {
        $this->car = new Car();
    }
    public function drive() {
        $this->car->run();
    }
}
```

Example (b) - Dependency Injection

```
class Car {
    public function run() {
        echo 'Vroooaaammmm!';
    }
}

class Driver {
    private $car;
    public function __construct(Car $car) {
        $this->car = $car;
    }
    public function drive() {
        $this->car->run();
    }
}
```

Example (c) - Dependency Inversion

```php
interface Car {
    public function run();
}

class Ferrari implements Car {
    public function run() {
        echo 'Vroooaaammmm!';
    }
}

class Driver {
    private $car;
    public function __construct(Car $car) {
        $this->car = $car;
    }
    public function drive() {
        $this->car->run();
    }
}
```

Dependency Injection (DI) Container

DI Container (key,value)

```
class ObjectContainer {

    protected $components = array();

    protected static $instance;
    protected function __construct() { }
    protected function __clone() { }
    private function __wakeup() { }


    public static function getInstance() {
        if (!isset(self::$instance)) {
                self::$instance = new self;
        }
        return self::$instance;
    }

    public function register($offset, $value) {
        if (is_null($offset)) {
            $this->components[] = $value;
        } else {
            $this->components[$offset] = $value;
        }
    }

    public function exists($offset) {
            return isset($this->components[$offset]);
    }

    public function unregister($offset) {
            unset($this->components[$offset]);
    }

    public function make($offset) {
            return isset($this->components[$offset]) ?
                    $this->components[$offset] : null;
    }

}
```

DI Container (key,value)

```
$container = ObjectContainer::getInstance();
$container->register('logger',new MyLogger(new EchoLogger));
$logger = $container->make('logger');
```

Closures

Anonymous/Lamda functions:

- no identifier
- first-class value types

Closures:

- with/without identifier
- persistent scope

Closures - Javascript Example

```
function makeCounter () {
  var count = 0;
  return function () {
    count += 1;
    return count;
  }
}

var x = makeCounter();

x(); returns 1

x(); returns 2

...etc...
```

DI Container - (key,closure)

```
class Container {

    protected $components = array();

    public static function register($offset, $value) {
        if ( ! $value instanceof \Closure) {
            $value = function() use ($value) {
                return $value;
            };
        }
        self::$components[$offset] = $value;
    }

    public static function exists($offset) {
        return isset(self::$components[$offset]);
    }

    public static function unregister($offset) {
        unset(self::$components[$offset]);
    }

    public static function make($offset) {
        return isset(self::$components[$offset]) ? self::$components[$offset]->__invoke() : null;
    }

}
```

DI Container - (key,closure)

```
Container::register('logger',new MyLogger(new EchoLogger));
Container::register('logger', function(){
                            $innerLogger = new EchoLogger();
                            $outerLogger = new MyLogger($innerLogger);
                            return $outerLogger;
                        }
                    );

$logger = Container::make('logger');
```

DI Container - (key,closure) with caching

```
class Container implements {

    protected $components = array();
    protected $cache;

    public function register($offset, $value) {
        if ( ! $value instanceof \Closure) {
            $value = function() use ($value)  {
                return $value;
            };
        }
        $this->components[$offset] = $value;
    }

    public function exists($offset) {
        return isset($this->components[$offset]);
    }

    public function unregister($offset) {
        unset($this->components[$offset]);
    }
```

```
    public function make($offset) {

            if(isset($this->components[$offset])){
                if(!isset($this->cache[$offset])){
                    $this->cache[$offset] = $this->components[$offset]-
>__invoke();
                }
                return $this->cache[$offset];
            } else {
                return null;
            }

        }

    }
```

Why use closures ?

Automatic Dependency Injection

```
$controller = “DomainController”;
$action = “registerDomain”;
$params = array();

$reflector = new \ReflectionClass($controller);
$dependencies = array();
foreach($reflector->getConstructor()->getParameters()  as $parameter){
    $className = $parameter->getClass()->getName();
    $dependencies[] = new $className;
}

return (new $controller($dependencies))->$action($params);
```