Patrones de diseño

Sesión 06:

PATRONES CREACIONALES: INTRODUCCIÓN A PATRONES CREACIONALES, PATRÓN SINGLETON Y PATRÓN PROTOTYPE

Universidad Tecnológica del Perú

Logro de la sesión

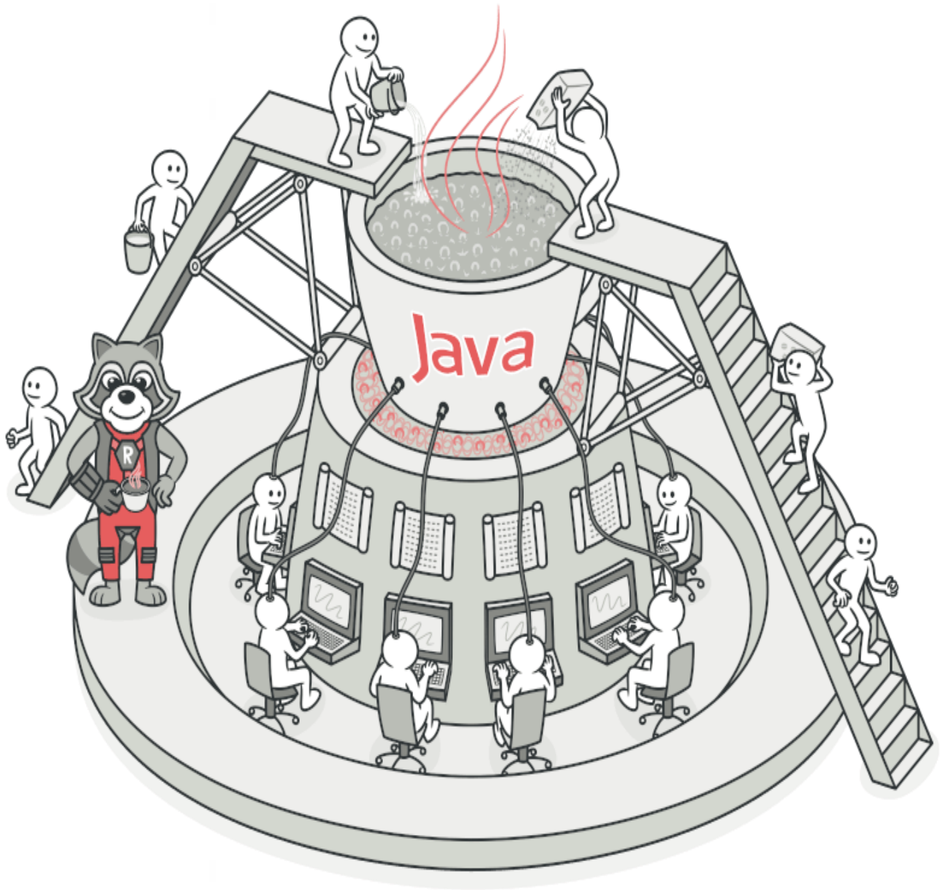
Al finalizar la sesión, el estudiante elabora su programación usando patrones creacionales como Patrón Prototype y Patrón Singleton empleando los diversos patrones de diseño aplicados a casos matemáticos, financieros y físicos.

Universidad Tecnológica del Perú

Patrones de Diseño

El catálogo de Patrones de Diseño está compuesto por 23 patrones:

1. Abstract Factory
2. Adapter
3. Bridge
4. Builder
5. Chain of Responsibility
6. Command
7. Composite
8. Decorator
9. Facade
10. Factory Method
11. Flyweight
12. Interpreter
13. Iterator
14. Mediator
15. Memento
16. Observer
17. Prototype
18. Proxy
19. Singleton
20. State
21. Strategy
22. Template Method
23. Visitor

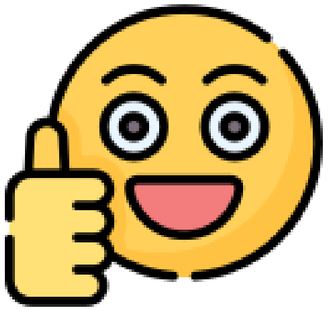


https://refactoring.guru/es/design-patterns/java

Universidad Tecnológica del Perú

Patrones de Diseño Los patrones de diseño varían en su granularidad y nivel de abstracción.

Como hay muchos patrones de diseño, necesitamos una forma de organizarlos. Los patrones se clasifican en base a dos criterios:



• Propósito: refleja qué hace el patrón

Creacional: conciernen al proceso de creación de objetos.
Estructural: tratan la composición de clases u objetos.
De Comportamiento: caracterizan las formas en que las clases u objetos interactúan y distribuyen la responsabilidad

• Alcance: especifica si el patrón se aplica primariamente a clases o a objetos.

Universidad Tecnológica del Perú

Patrones de Diseño

		PROPÓSITO		
		CREACIONAL	ESTRUCTURAL	COMPORTAMIENTO
ALCANCE	CLASE	Factory Method	Adapter	Interpreter Template Method
	OBJETO	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Universidad Tecnológica del Perú

PATRONES CREACIONALES (Creational Patterns)

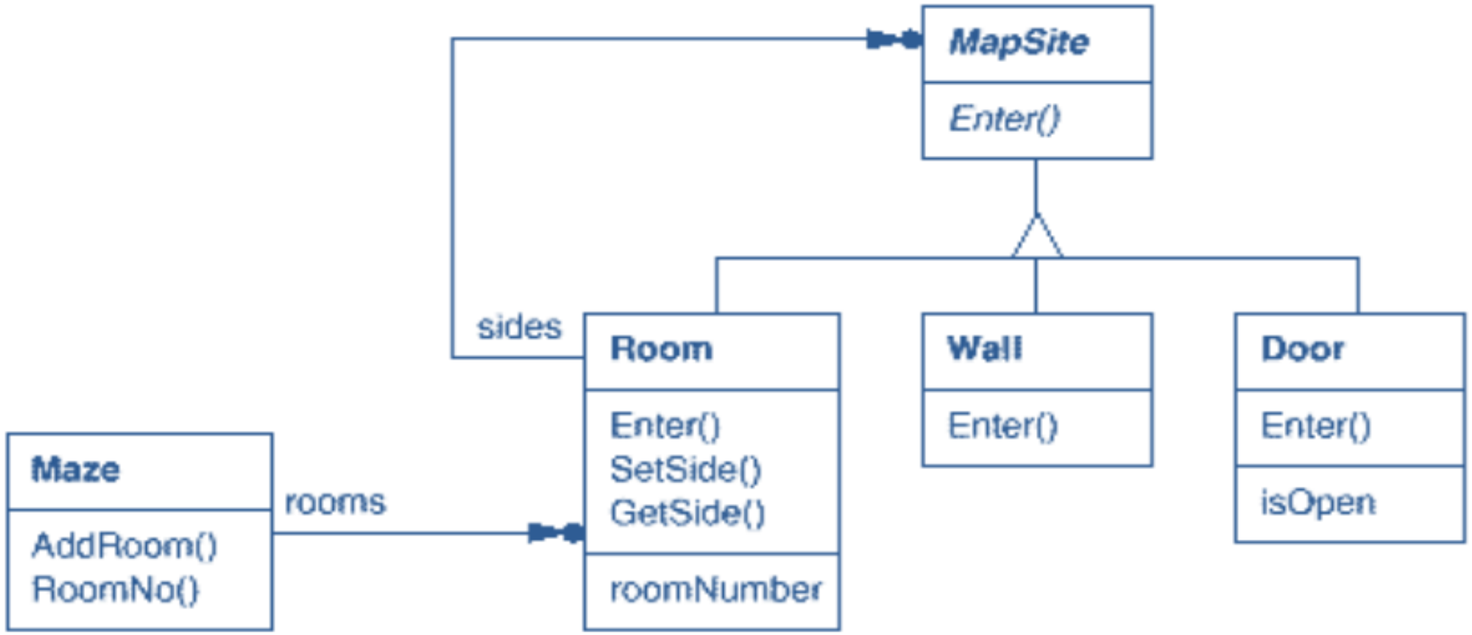
Los patrones creacionales son patrones que abstraen el proceso de instanciación.

Procuran independizar el sistema de cómo sus objetos son **creados, compuestos y representados.**

Los patrones indican soluciones de diseño para **encapsular** el conocimiento acerca de las clases que el sistema usa y **ocultar** cómo se crean instancias de estas clases.

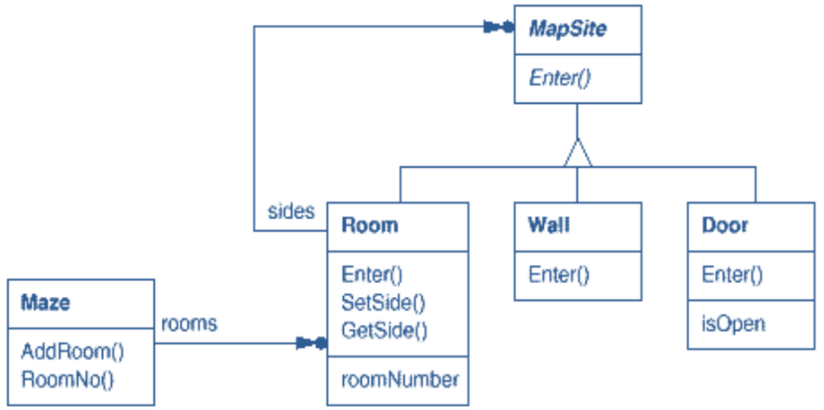
Patrones Creacionales

Un laberinto (maze) es un conjunto de habitaciones (rooms). Una habitación conoce a sus vecinos, los cuales pueden ser otra habitación, un muro o una puerta a otra habitación.



Universidad Tecnológica del Perú

Patrones Creacionales



- Cada habitación tiene cuatro lados.

  En C++ podemos declarar un enumerado: enum Direction {North, South, East, West};

- La clase MapSite es una clase abstracta para todos los componentes del laberinto. Posee una sola operación Enter() para simplificar.
- El significado de Enter() depende del componente: Si es una habitación, cambiamos de locación, si es una puerta y está abierta, pasamos a la siguiente habitación.

Universidad Tecnológica del Perú

Patrones Creacionales

```
class Room : public MapSite {
public:
        Room(int roomNo);
        MapSite* GetSide(Direction) const;
        void SetSide(Direction, MapSite*);
        virtual void Enter();
private:
        MapSite* _sides[4];
        int _roomNumber;
};
```

```
class Door : public MapSite {
public:
  Door(Room* = 0, Room* = 0);
  virtual void Enter();
  Room* OtherSideFrom(Room*);
private:
  Room* _room1; Room* _room2;
  bool _isOpen; };
```

```
class Wall:public MapSite
{
public:
  Wall();
  virtual void Enter();
};
```

```
class Maze {
public:
    Maze();
    void AddRoom(Room*);
    Room* RoomNo(int)const;
private:
        // ...
};
```

Universidad Tecnológica del Perú

Patrones Creacionales

En la clase **MazeGame** necesitamos crear un laberinto para el juego....

```
Maze* MazeGame::CreateMaze () {
        Maze* aMaze = new Maze;
        Room* r1 = new Room(1);
        Room* r2 = new Room(2);
        Door* theDoor = new Door(r1, r2);
        aMaze->AddRoom(r1);
        aMaze->AddRoom(r2);
        r1->SetSide(North, new Wall());
        r1->SetSide(East, theDoor);
        r1->SetSide(South, new Wall());
        r1->SetSide(West, new Wall());
        r2->SetSide(North, new Wall());
        r2->SetSide(East, new Wall());
        r2->SetSide(South, new Wall());
        r2->SetSide(West, theDoor);
        return aMaze;
}
```

Universidad Tecnológica del Perú

Patrones Creacionales

En la clase **MazeGame** necesitamos crear un laberinto para el juego....

```
Maze* MazeGame::CreateMaze () {
        Maze* aMaze = new Maze;
        Room* r1 = new Room(1);
        Room* r2 = new Room(2);
        Door* theDoor = new Door(r1, r2);
        aMaze->AddRoom(r1);
        aMaze->AddRoom(r2);
        r1->SetSide(North, new Wall());
        r1->SetSide(East, theDoor);
        r1->SetSide(South, new Wall());
        r1->SetSide(West, new Wall());
        r2->SetSide(North, new Wall());
        r2->SetSide(East, new Wall());
        r2->SetSide(South, new Wall());
        r2->SetSide(West, theDoor);
        return aMaze;
}
```

En la clase **MazeGame** necesitamos crear un laberinto para el juego....

```
Maze* MazeGame::CreateMaze () {
        Maze* aMaze = new Maze;
        Room* r1 = new Room(1);
        Room* r2 = new Room(2);
        Door* theDoor = new Door(r1, r2);
        aMaze->AddRoom(r1);
        aMaze->AddRoom(r2);
        r1->SetSide(North, new Wall());
        r1->SetSide(East, theDoor);
        r1->SetSide(South, new Wall());
        r1->SetSide(West, new Wall());
        r2->SetSide(North, new Wall());
        r2->SetSide(East, new Wall());
        r2->SetSide(South, new Wall());
        r2->SetSide(West, theDoor);
        return aMaze;
}
```

Universidad Tecnológica del Perú

En la clase **MazeGame** necesitamos crear un laberinto para el juego....

```
Maze* MazeGame::CreateMaze () {
        Maze* aMaze = new Maze;
        Room* r1 = new Room(1);
        Room* r2 = new Room(2);
        Door* theDoor = new Door(r1, r2);
        aMaze->AddRoom(r1);
        aMaze->AddRoom(r2);
        r1->SetSide(North, new Wall());
        r1->SetSide(East, theDoor);
        r1->SetSide(South, new Wall());
        r1->SetSide(West, new Wall());
        r2->SetSide(North, new Wall());
        r2->SetSide(East, new Wall());
        r2->SetSide(South, new Wall());
        r2->SetSide(West, theDoor);
        return aMaze;
}
```

Esta operación crea un laberinto con dos habitaciones

Puede simplificarse. Por ejemplo, las habitaciones podrían crear las paredes.

Pero esto sólo mueve código de un lugar a otro ☹

En la clase **MazeGame** necesitamos crear un laberinto para el juego....

```
Maze* MazeGame::CreateMaze () {
        Maze* aMaze = new Maze;
        Room* r1 = new Room(1);
        Room* r2 = new Room(2);
        Door* theDoor = new Door(r1, r2);
        aMaze->AddRoom(r1);
        aMaze->AddRoom(r2);
        r1->SetSide(North, new Wall());
        r1->SetSide(East, theDoor);
        r1->SetSide(South, new Wall());
        r1->SetSide(West, new Wall();
        r2->SetSide(North, new Wall());
        r2->SetSide(East, new Wall());
        r2->SetSide(South, new Wall());
        r2->SetSide(West, theDoor);
        return aMaze;
}
```

Además **¿Qué pasa si queremos agregar otro elemento al laberinto o modificar uno existente?**

Los patrones creacionales procuran hacer este diseño más flexible, quitando las referencias explícitas a clases concretas.

Universidad Tecnológica del Perú

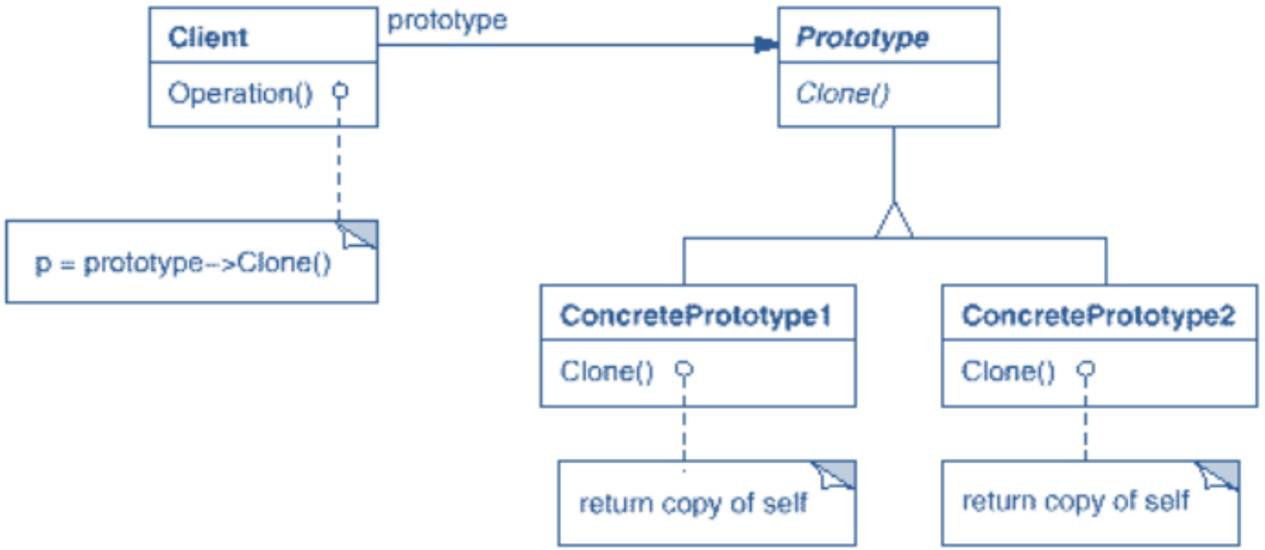
Patrones Creacionales: Prototype

- Intención: especifica los tipos de objetos a crear utilizando una instancia prototipo, y crea nuevos objetos copiando esta instancia.
- **Aplicabilidad:** Usamos este patrón

1. cuando las clases a instanciar son especificadas en tiempo de ejecución, por ejemplo, por carga dinámica.
2. cuando instancias de una clase pueden tener sólo una de muchas combinaciones de estados, lo que puede obtenerse clonando prototipos en lugar de hacerlo manualmente.

Este patrón utiliza “prototipos” de los objetos a crear, los cuales obtiene por clonación.

Estructura:



Participantes:

- **Prototype** declara una interfaz para la autoclonación.
- **ConcretePrototype** implementa una operación para clonarse a sí mismo.
- **Client** crea un nuevo objeto solicitándole al prototipo que se clone a sí mismo.

Universidad Tecnológica del Perú

Apliquemos este patrón al escenario de los laberintos.

```
class MazePrototypeFactory : public MazeFactory {
    public:
        MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

        virtual Maze* MakeMaze() const;
        virtual Room* MakeRoom(int) const;
        virtual Wall* MakeWall() const;
        virtual Door* MakeDoor(Room*, Room*) const;

    private:
        Maze* _prototypeMaze;
        Room* _prototypeRoom;
        Wall* _prototypeWall;
        Door* _prototypeDoor; };
```

El constructor inicializa los prototipos con los parámetros.

Las funciones miembro para crear habitaciones, puertas y paredes simplemente **solicitan clones a los prototipos**.

```
Wall* MazePrototypeFactory::MakeWall () const {
        return _prototypeWall->Clone();
}
Door* MazePrototypeFactory::MakeDoor (Room* r1, Room *r2) const {
        Door* door = _prototypeDoor->Clone();
        door->Initialize(r1, r2);
        return door;
}
```

Para crear laberintos utilizamos la operación **MakeMaze()**

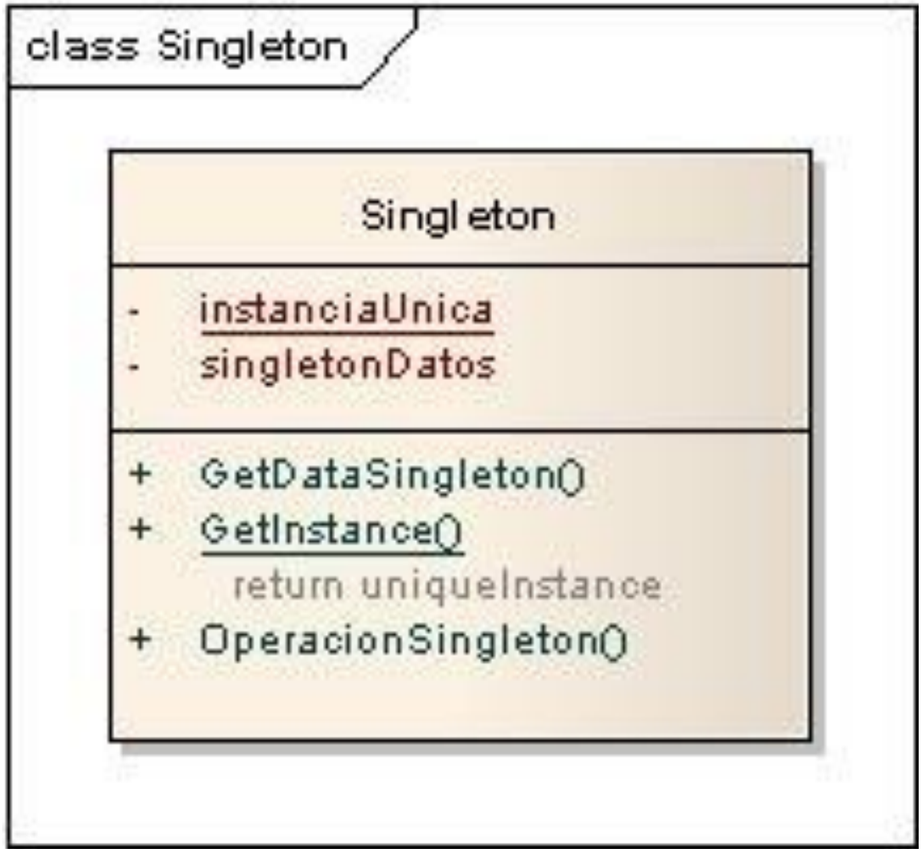
Para crear laberintos con otras características (por ejemplo, habitaciones con ciertas propiedades adicionales), simplemente inicializamos **MazePrototypeFactory** con otros prototipos.

Patrón Singleton

El patrón de diseño Singleton (instancia única) está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella. No se encarga de la creación de objetos en sí, sino que se enfoca en la restricción en la creación de un objeto.

Este patrón es aplicable en sistemas en los que se desea poder garantizar que solo existe una instancia de una clase.

Patrón Singleton



Implementación

Siempre que se crea un objeto nuevo (en Java, con la palabra reservada new) se invoca al constructor del objeto para que cree una instancia. Por lo general, los constructores son públicos. El Singleton lo que hace es convertir el constructor en privado, de manera que nadie lo pueda instanciar. Entonces, si el constructor es privado, ¿cómo se instancia el objeto? Pues a través de un método público y estático de la clase. En este método se revisa si el objeto ha sido instanciado antes. Si no ha sido instanciado, llama al constructor y guarda el objeto creado en una variable estática del objeto. Si el objeto ya fue instanciado anteriormente, lo que hace este método es devolver la referencia al estado creado anteriormente.

Hay que tener especial cuidado cuando el *Singleton* se utiliza en un ambiente multihilos, porque puede crear problemas si no se implementa de la manera adecuada. En Java es posible que tengamos que realizar una "**incialización bajo demanda"** para evitar problemas en este sentido. Concluyendo, la idea central del *Singleton* es esa: asegurar de que exista tan solo una instancia del objeto en toda la aplicación.

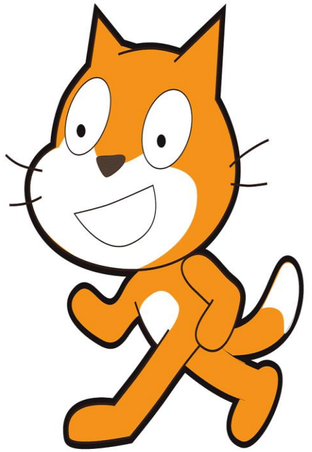
Es un patrón muy aplicado en Java, aunque, como todos los patrones, se puede implementar en cualquier lenguaje orientado a objetos.
Para realizar una buena implementación del patrón *Singleton* se recomienda.

•Ocultar el constructor.
•Declarar como estático el método getInstance.
•En ambiente concurrentes es necesario usar mecanismos que aseguren la atomicidad del método getInstance.

Universidad Tecnológica del Perú

Implementación del patrón bajo demanda. Tenemos el código siguiente:

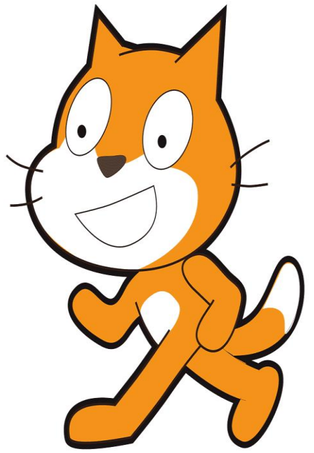
```
public class Singleton {

     static private Singleton singleton = null;

     private Singleton() { }

     static public Singleton getSingleton() {

         if (singleton == null) {
             singleton = new Singleton();
         }
         return singleton;\\
     }

     public String metodo() {\\
         return "Singleton instanciado bajo demanda";
     }

 }
```



Universidad Tecnológica del Perú

Esta implementación se caracteriza porque nuestra instancia única se crea cuando se usa por primera vez – bajo demanda – gracias a una sentencia condicional. En el a continuación eliminaremos esto, no por ser incorrecto, sino porque en general no es necesario. La condición del método singleton() puede ser eliminada. Dicha condición se va a evaluar a cierto siempre salvo la primera vez. Además, el cargador de clases de Java, cuando referenciamos una clase no cargada anteriormente, se ocupa de cogerla de disco y cargarla en memoria inicializando sus miembros static.Por tanto el siguiente código actúa exactamente igual que el caso 1, creando la instancia en la primera invocación pero eliminando la condición.

```
public class Singleton {

     static private Singleton singleton = new Singleton();\\

     private Singleton() { }

     static public Singleton getSingleton() {
         return singleton;
     }
     public String metodo() {\\
         return "Singleton ya instanciado";\\
     }

 }
```



tenemos que solicitar la instancia única del mismo. Habitualmente se hace de dos formas diferentes, la primera se usa en llamadas puntuales.

```
Singleton.getsingleton().metodo();
```

La segunda se usa cuando vamos a invocar nuestro Singleton varias veces, evitando llamadas innecesarias a métodos. Este tipo de singleton es un objeto como otro cualquiera, así que puede ser referenciado sin problemas.

```
Singleton s = Singleton.getSingleton();
 s.metodo();
```

Ejercicio grupal

Al Implementar el modelo de patrón Singleton y Prototype del Patron de diseño creacional usar como referencia los siguientes repositorios en JAVA y describir el comportamiento que observan en el código a diferencia de un modelo tradicional de POO.

https://github.com/mcupo/Patron-Singleton-Java/blob/master/src/ar/edu/ort/SingletonConfiguracion.java

https://github.com/Angel-Raa/Design-patterns-java

Universidad Tecnológica del Perú

¿Consultas o dudas?

Universidad Tecnológica del Perú

Actividad

Resolver la actividad planteada en la plataforma.

Universidad Tecnológica del Perú

Cierre

¿Qué hemos aprendido hoy?

Elaboramos nuestras conclusiones sobre el tema tratado

Universidad Tecnológica del Perú

Universidad
Tecnológica
del Perú