Singleton

Design Pattern

Giselle Quarto

WHAT

Design patterns are typical solutions to commonly occurring problems in software design.

TYPES

- Creational
- Structural
- Behavioral

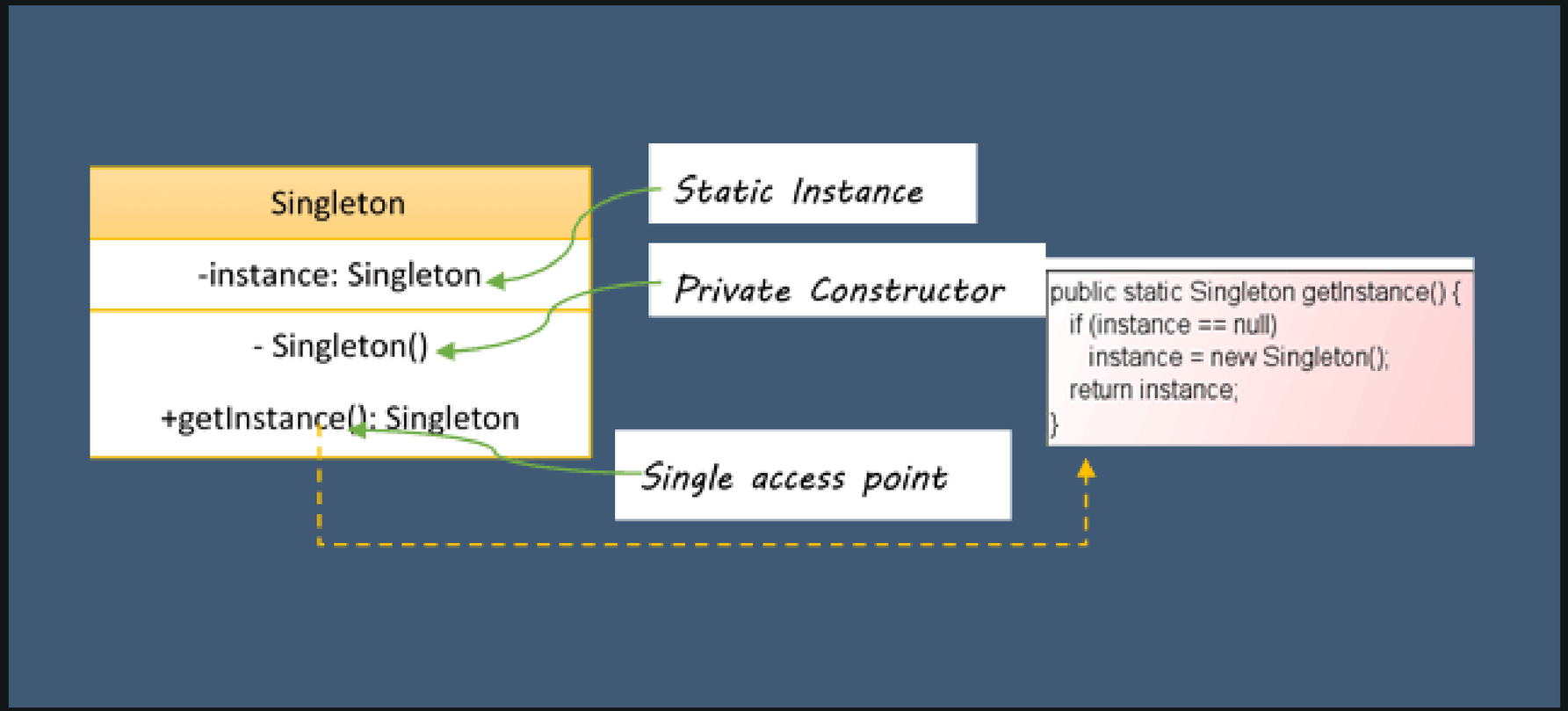
Problem:

- How to ensure that a class has just a single instance
- How to provide a global access point to that instance.

Implementation check list:

- Default constructor private, to prevent other objects from using the new operator with the Singleton class
- Static creation method that acts as a constructor. All following calls to this method return the cached object.
- Static attribute of the class instance itself

Class Diagram



THE PROBLEM:

Our FRIDGE is empty!!!

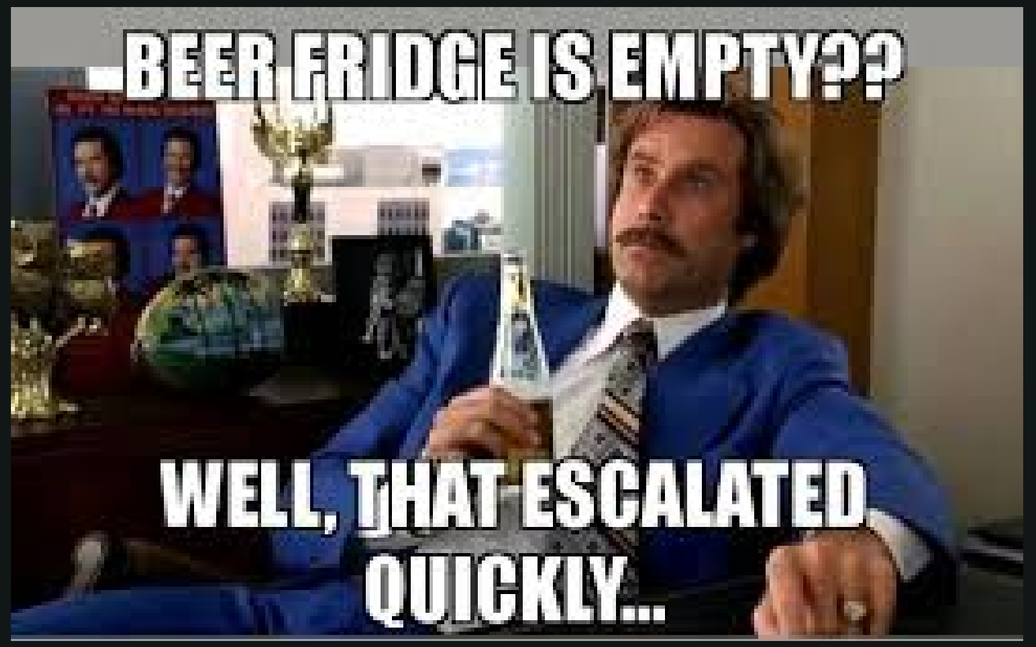
```java
public class Fridge {

    private int beerQuantity;

    public Fridge() {
    }

    public void fillFridge(int beer) {
        beerQuantity += beer;
    }

    public int getBeerQuantity() {
        return beerQuantity;
    }
}
```



Trying Parameter:

```java
public class CodeCadetParameter {

    //With Singleton you don't need to pass Fridge as parameter here
    public void buyBeer (Fridge fridge, int beer) {
        fridge.fillFridge(beer);
    }

}
```

```java
public class CodeBreakParameter {

    public static void main(String[] args) {

        //Random variable's name
        CodeCadetParameter edgard = new CodeCadetParameter();
        CodeCadetParameter louis = new CodeCadetParameter();

        Fridge smallFridge = new Fridge();

        edgard.buyBeer(smallFridge, beer: 10);

        // TO avoid this : SINGLETON
        Fridge otherFridge = new Fridge();

        louis.buyBeer(otherFridge, beer: 5);

        System.out.println(smallFridge.getBeerQuantity()); // 10
        System.out.println(otherFridge.getBeerQuantity()); // 5
```

Trying Reference:

```java
public class CodeCadetReference {

    //You can try with Reference
    private final Fridge fridge;

    public CodeCadetReference(Fridge fridge) {
        this.fridge = fridge;
    }

    public void buyBeer(int beer) {
        fridge.fillFridge(beer);
    }
}
```

```java
public class CodeBreakReference {

    public static void main(String[] args) {


        Fridge smallFridge = new Fridge();
        //Same Problem here!
        Fridge otherFridge = new Fridge();

        // random variable's name
        CodeCadetReference mark = new CodeCadetReference(smallFridge);
        CodeCadetReference andrew = new CodeCadetReference(otherFridge);

        mark.buyBeer(15);
        andrew.buyBeer(10);

        System.out.println(smallFridge.getBeerQuantity()); //15
        System.out.println(otherFridge.getBeerQuantity()); //10
```

Singleton Solution:

```java
public class Fridge {

    private static final Fridge fridge = new Fridge();
    private int beerQuantity;

    private Fridge() {
    }

    public static Fridge getInstance() {
        return fridge;
    }

    public void fillFridge(int beer) {
        beerQuantity += beer;
    }

    public int getBeerQuantity() {
        return beerQuantity;
    }
}
```

```java
public class CodeBreak {

    public static void main(String[] args) {

        CodeCadet erick = new CodeCadet();
        CodeCadet eloise = new CodeCadet();

        erick.buyBeer(10);
        eloise.buyBeer(15);

        //Even if you try
        Fridge smallFridge = Fridge.getInstance();
        Fridge otherFridge = Fridge.getInstance();

        System.out.println(Fridge.getInstance().getBeerQuantity()); // 25

    }
}
```

EAGER INITIALIZATION

```
public class Canvas {

    private static Canvas canvas = new Canvas();

    private Canvas(){

    }

    public static Canvas getInstance() {
        return canvas;
    }
```

LAZY INITIALIZATION

```
public class Canvas {

    private static Canvas canvas;

    private Canvas(){

    }

    public static Canvas getInstance() {
        if(canvas == null) {
            canvas = new Canvas();
        }
        return canvas;
    }
```

THE END