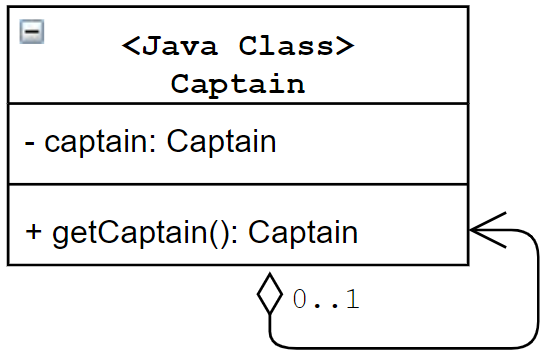
Design Patterns

Singleton Pattern

- It can keep having a single instance rather than multiple.

- It is suitable in a centralized system to restrict unnecessary object creations.



package singleton;

// We need a final class to prevent the loose hole 1

// final class Captain {

class Captain {

static int numberOfInstance=0;

private static Captain captain; // it will proceed Lazy initialization

// or

// private static final Captain captain = new Captain(); // Early initialization and thread safe

// private is to private multiple instance creation

private Captain() {

numberOfInstance++;

System.out.println("Number of instances at this moment=" + numberOfInstance);

}

public static synchronized Captain getCaptain() {

// Lazy initialization

if(captain == null) {

captain = new Captain();

System.out.println("New captain is elected for your team.");

}

else {

System.out.println("You already have a captain for your team.");

}

return captain;

}

// a static dummy method

public static void dummyMethod() {

System.out.println("It is a dummy method");

}

// inner a non-static nested class

public class CaptainDerived extends Captain {

//Some code

}

}

public class SingletonPatternExample {

public static void main(String[] args) {

System.out.println("\*\*\*Singleton Pattern Demo\*\*\*\n");

System.out.println("Trying to make a captain for your team:");

// Constructor is private. We cannot use "new" here.

// Captain c3 = new Captain(); // error

Captain captain1 = Captain.getCaptain();

System.out.println("Trying to make another captain for your team:");

Captain captain2 = Captain.getCaptain();

if (captain1 == captain2) {

System.out.println("captain1 and captain2 are same instance.");

}

// loose hole 1: it will trigger to the constructor method of parent class again

Captain.CaptainDerived derived = captain1.new CaptainDerived();

// loose hole 2: it will trigger to the constructor method without intention

Captain.dummyMethod();

}

}

Solutions of above loose hole 2:

Solution 1: Bill Pugh’s Solution

* It uses a static nested helper class
* It does not use synchronization technique and eager initialization

class Captain1 {

private Captain1() {

System.out.println("A captain is elected for your team.");

}

// Bill Pugh solution

private static class SingletonHelper {

/\*Nested class is referenced after getCaptain() is called\*/

private static final Captain1 captain = new Captain1();

}

public static Captain1 getCaptain() {

return SingletonHelper.captain;

}

public static void dummyMethod() {

System.out.println("It is a dummy method");

}

}

Solution 2: Double-Checked Locking

Double-checked locking is the practice of checking a lazy-initialized object's state both before and after a synchronized block is entered to determine whether or not.

final class Captain2 {

private static Captain2 captain;

//We make the constructor private to prevent the use of "new"

static int numberOfInstance=0;

private Captain2() {

numberOfInstance++;

System.out.println("Number of instances at this moment="+

numberOfInstance);

}

public static Captain2 getCaptain(){

if (captain == null) {

synchronized (Captain2.class) { // it is better for performance using inside if-then-else condition

// Lazy initialization

if (captain == null){

captain = new Captain2();

System.out.println("New captain is elected for your team.");

}

else {

System.out.print("You already have a captain for your team.");

System.out.println("Send him for the toss.");

}

}

}

return captain;

}

}