Instance-Control

Αθανάσιος Ανδρούτσος

Instance-Control

- Σε κάποιες εφαρμογές θέλουμε να δημιουργούμε κλάσεις που να μπορούν να ελέγχουν το πλήθος των instances που υπάρχει ανά πάσα στιγμή (*instance-controlled* classes)
- Θα θέλαμε για παράδειγμα να μπορούμε να δημιουργούμε κλάσεις που να παρέχουν **μηδέν instances (utility Classes)**, **ένα instance μόνο (Singletons)**, **συγκεκριμένα μοναδικά instances (flyweight pattern)**

Utility Classes

- Σε κάποιες περιπτώσεις θέλουμε απλές Utility κλάσεις ή Helper κλάσεις που να περιέχουν **μόνο public λειτουργίες** (όχι data) και να είναι **non-instantiable**
- Κάνουμε enforce noninstantiability με **private Constructor**
- Τέτοιες κλάσεις συνήθως θέλουμε όταν παρέχουν **οριζόντια λειτουργικότητα** και υπηρεσίες σε όλες τις άλλες κλάσεις
- Για παράδειγμα όταν θέλουμε να παρέχουμε ένα API για επικοινωνία με μια Βάση Δεδομένων ή ένα Validation API

HelloUtil Utility class


```java
package gr.aueb.cf.testbed.util;

public class HelloUtil {

    /**
     * No instances of this Utility class is available.
     */
    private HelloUtil() {}

    public static void sayHello() {
        System.out.println("Hello World!");
    }
}
```

- Οι **utility κλάσεις** είναι non-instantiable (private constructor) και περιέχουν **μόνο public static μεθόδους**

Χρήση της HelloUtil


```
1  package gr.aueb.cf.testbed.util;
2
3  public class Hello {
4
5      public static void main(String[] args) {
6          HelloUtil.sayHello();
7      }
8  }
```

- Οι utility methods (public static) καλούνται με το όνομα της Utility class και τον τελεστή τελεία

Static factory μέθοδοι (1)

- Ο παραδοσιακός τρόπος για μια κλάση να **επιτρέψει σε ένα client να δημιουργήσει ένα instance της κλάσης** είναι να παρέχει ένα public constructor
- **Υπάρχει και μία άλλη τεχνική** που πρέπει να έχουμε στην εργαλειοθήκη μας ως προγραμματιστές για να δίνουμε τη δυνατότητα δημιουργίας instances
- Να παρέχει η κλάση μία **static factory method** που είναι απλά μία **static μέθοδος που επιστρέφει ένα instance της κλάσης**

Static factory μέθοδοι (2)

- Γενικά οι factory τεχνικές είναι **creative τεχνικές**, δηλαδή τεχνικές που δημιουργούν νέα instances κλάσεων
- Είναι μία σημαντική τεχνική
- Ας δούμε ένα παράδειγμα…

Static factory μέθοδοι (3)


```java
14      /**
15       * Initializes a newly created point,
16       * so that it represents a (0,0) point.
17       */
18      public Point() {}
19
20      /**
21       * Constructs a new Point with specific
22       * (x,y) coordinates.
23       *
24       * @param x     the x coordinate
25       * @param y     the y coordinate
26       */
27      public Point(int x, int y) {
28          this.x = x;
29          this.y = y;
30      }
31
32      /**
33       * Static factory method.
34       * @return  a new Point instance.
35       */
36  @   public static Point getInstance() {
37          return new Point();
38      }
```

- Βλέπουμε ένα απλό παράδειγμα.
- Η μέθοδος ***getInstance*** επιστρέφει ένα νέο object reference σε Point
- Οι static factory μέθοδοι μπορούν να χρησιμοποιηθούν αντί των Constructors ή μαζί με τους Constructors

Πλεονεκτήματα – Όνομα (1)

- 1. Οι constructors έχουν **μόνο ένα** signature ενώ οι static factories έχουν **όνομα**
- Αν θέλουμε να επιστρέφουμε διαφορετικά instances με τον ίδιο όνομα του constructor είτε θα έπρεπε να διαφοροποιήσουμε τη σειρά των παραμέτρων που είναι όμως error prone ή αν δεν έχει παραμέτρους ή έχει μία παράμετρο δεν μπορούμε καθόλου να διαφοροποιήσουμε το επιστρεφόμενο instance
- Με τις static factories μπορούμε απλά να διαφοροποιήσουμε το όνομα

Πλεονεκτήματα – Όνομα (2)

- Πιο αναλυτικά, στις περιπτώσεις που **μία κλάση μπορεί να χρειάζεται να παρέχει περισσότερους από ένας constructors** με την ίδια υπογραφή, τότε ένας τρόπος είναι να παρέχουμε constructors με λίστες παραμέτρων που να διαφέρουν μόνο ως προς το ordering
- Κάτι τέτοιο είναι πολύ κακή ιδέα, γιατί κανείς δεν θα θυμάται ποιος constructor είναι ποιος με αποτέλεσμα να πρέπει να ανατρέχει στο documentation ή απλά να καλεί τον λάθος constructor

Πλεονεκτήματα – Όνομα (3)

- Επειδή **οι static factory μέθοδοι έχουν ονόματα** μου μπορούμε εμείς να επιλέξουμε, **θα μπορούσαμε να διαλέξουμε προσεκτικά επιλεγμένα ονόματα,** ώστε να είναι εύκολα readable οι διαφορές
- Έστω ότι θέλουμε η κλάση Point να παρέχει δύο default constructors που ο ένας θα επιστρέφει ένα σημείο (0,0) και ο άλλος ένα τυχαίο σημείο

Πλεονεκτήματα – Όνομα (4)

- Κάτι τέτοιο δεν μπορεί να γίνει με τους κλασσικούς constructors γιατί δεν μπορούμε να έχουμε δύο default constructors
- Με static factory methods θα μπορούσαμε να ορίσουμε δύο static factory methods με διαφορετικό όνομα, για παράδειγμα, *Point.getZeroPoint()* και *Point.getRandomPoint()*

Κλάση Random

- Η **κλάση Random** όπως και η μέθοδος Math.random() παράγει ψευδοτυχαίους αριθμούς
- Στην κλάση Random μπορούμε να θέσουμε ένα seed με **new Random(long seed)** ώστε αν το seed είναι διαφορετικό να πάρουμε διαφορετικούς ψευδοτυχαίους αριθμούς
- Με την μέθοδο **nextInt()** μπορούμε να πάρουμε οποιονδήποτε int. Με **nextInt(int bound)** μπορούμε να πάρουμε από 0 inclusive έως bound, exclusive
- Με **nextInt(max-min+1) + min** μπορούμε να πάρουμε από min έως max

Ονοματοδοσία

- Μερικά κοινά ονόματα για static factories είναι:
  - of()
  - valueOf()
  - instance() ή getInstance()
  - create() ή newInstance()
  - *type*()
  - get*Type*() ή new*Type*()

Παράδειγμα


```
23      /**
24       * A static factory that returns
25       * a (0,0) new point.
26       *
27       * @return
28       *      a (0, 0) new point
29       */
30  @   public static Point getZeroPoint() { return new Point(); }
33
34      /**
35       * A static factory that returns a random point
36       * between (0..100, 0..100).
37       *
38       * @return
39       *      a random point between (0..100, 0..100)
40       */
41  @   public static Point getRandomPoint() {
42          Random rnd = new Random(System.currentTimeMillis());
43          return new Point(rnd.nextInt(101), rnd.nextInt(101));
44      }
```

- Δύο static factory methods με προσεκτικά επιλεγμένα ονόματα ώστε να είναι εύκολα κατανοητή η λειτουργία τους

main


```
13          Point p1 = Point.getZeroPoint();
14          Point p2 = Point.getRandomPoint();
15          Point p3 = Point.getRandomPoint();
16
17          System.out.println(p1.pointToString());
18          System.out.println(p2.pointToString());
19          System.out.println(p3.pointToString());
20
```

- Παρατηρούμε ότι στην πλευρά του client ο κώδικας με τις static factory μεθόδους είναι σαν να 'τεκμηριώνει' τη λειτουργία των constructors

Instance-Controlled Classes

- Οι static factory μέθοδοι δίνουν τη δυνατότητα **να μην επιστρέφουμε πάντα new objects** αλλά και **preconstructed instances**
- Αυτή η ιδιότητα **μας επιτρέπει να ελέγχουμε το πλήθος των instances** που θέλουμε να υπάρχει ανά πάσα στιγμή (*instance-controlled* classes)
  - Μπορούμε για παράδειγμα να περιορίσουμε **σε μόνο ένα** το instance μιας κλάσης (*Singleton* pattern)
  - Επίσης, επιτρέπει να μην δημιουργούμε duplicate objects καθώς επίσης και να εγγυηθούν ότι δεν υπάρχουν two διαφορετικά instances με το ίδιο state, δηλαδή a.equals(b) μόνο αν a==b (*Flyweight* pattern)

Singleton Design Pattern (1)

- Ένα πολύ ενδιαφέρον design pattern είναι το **Singleton pattern** που είναι κατά βάση ένα instance-controlled pattern, που επιτρέπει τη δημιουργία από τους clients **μόνο ενός instance** της κλάσης
- Ένας τρόπος για να γίνει αυτό είναι να κάνουμε private τον default constructor και να χρησιμοποιήσουμε μία static factory method που να επιστρέφει ένα pre-constructed static final instance

Singleton Design Pattern (2)


```java
package testbed.ch13.staticfactory;

/**
 * Defines a Singleton Coding Factory
 * that says Hello!
 */
public class CodingFactory {
    private static final CodingFactory INSTANCE = new CodingFactory();

    private CodingFactory() {}

    public static CodingFactory getInstance() {
        return INSTANCE;
    }

    public void sayHello() {
        System.out.println("Hello!");
    }
}
```

- Eager Instantiation
- Το instance δημιουργείται όταν φορτώνεται η κλάση
- Στη συνέχεια η static factory επιστρέφει το preconstructed instance

Lazy Instantiation


```java
package testbed.ch13.staticfactory;

/**
 * Defines a Singleton Coding Factory
 * with lazy instantiation.
 */
public class CodingFactoryLazy {
    private static CodingFactoryLazy SINGLETON = null;

    private CodingFactoryLazy() {}

    public static CodingFactoryLazy getInstance() {
        if (SINGLETON == null) {
            SINGLETON = new CodingFactoryLazy();
        }
        return SINGLETON;
    }

    public void sayHello() {
        System.out.println("Hello!");
    }
}
```

- Το instantiation γίνεται μόλις καλείται η static factory την πρώτη φορά

Singleton Pattern

- Ο private constructor καλείται μία φορά και στη συνέχεια επιστρέφεται το ίδιο instance
- Η μη-παροχή public ή protected constructor εγγυάται ένα "mono-coding-factory" universe!
- Τα singletons τυπικά χρησιμοποιούνται σε **stateless objects** ή σε system components που είναι μοναδικά

Main


```java
package testbed.ch13.staticfactory;

public class Main {

    public static void main(String[] args) {
        CodingFactory codingFactory = CodingFactory.getInstance();
        codingFactory.sayHello();
    }
}
```

- Singleton instance (μόνο ένα) της κλάσης Coding Factory

Flyweight pattern (1)

- Το Flyweight pattern επιτρέπει σε immutable value classes να εγγυώνται ότι **δεν υπάρχουν δύο ή περισσότερα equal instances**
- Για παράδειγμα αν έχουμε ένα ImmutablePoint, τότε δεν μπορεί δύο διαφορετικά immutable points a, b να είναι equal παρά μόνο αν οι αναφορές είναι ίσες, δηλαδή a == b (Flyweight pattern)

Immutable Point


```java
/**
 * Defines an immutable point.
 */
final class ImmutablePoint {
    private final int x;
    private final int y;

    ImmutablePoint() {
        x = 0;
        y = 0;
    }

    ImmutablePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    public String convertToString() {
        return "(" + x + ", " + y +")";
    }
}
```

- Immutable κλάσεις και instances έχουμε με private **final πεδία**, και όχι setters

Flyweight Pattern (1)


```java
package gr.aueb.cf.ch14.flyweight;

import java.util.ArrayList;
import java.util.List;

/**
 * {@link FlyweightFactory} implements the <b>Flyweight design
 *  pattern<b/>. That is it guarantees that two {@link ImmutablePoint}
 *  instances a, b are equal (states are equal) iff (if and only if)
 * a == b (references are equal).
 *
 * This is also a <b>facade design pattern<b/> since {@link FlyweightFactory}
 * class provides an 'interface' to other packages for getting {@link ImmutablePoint}
 * instances since {@link ImmutablePoint} constructor is package private and
 * external packages can not directly get {@link ImmutablePoint} instances.
 */
public class FlyweightFactory {
    private static final List<ImmutablePoint> points = new ArrayList<>();

    private FlyweightFactory() {}
```

Flyweight Pattern (2)


```java
/**
 * Static factory that returns unique points
 * or a reference to an existing point.
 *
 * @param x
 * @param y
 * @return
 */
public static ImmutablePoint getFlyweightPoint(int x, int y) {
    ImmutablePoint point  = null;
    int position = getPointPosition(x, y);

    if (position == -1) {
        point = new ImmutablePoint(x, y);
        points.add(point);
    } else {
        point = points.get(position);
    }

    return point;
}
```

- Η getFlyweightPoint() είναι static factory και επιστρέφει points που είναι μοναδικά
- Αν υπάρχουν ήδη στην cache (η ArrayList points) τότε επιστρέφονται αναφορές σε αυτά

Flyweight Pattern (3)


```
44      private static int getPointPosition(int x, int y) {
45          int positionToReturn = -1;
46
47          for (int i = 0; i < points.size(); i++) {
48              if ((points.get(i).getX() == x) && (points.get(i).getY() == y)) {
49                  positionToReturn = i;
50                  break;
51              }
52          }
53          return positionToReturn;
54      }
55  }
```

- Αν το point υπάρχει στην λίστα, τότε επιστρέφεται η θέση του point. Αν δεν υπάρχει τότε επιστρέφεται -1.

Εργασίες (1)

- Υλοποιήστε μία Singleton class με όνομα **Logger** που κάνει log messages στο System.err. Θα πρέπει να περιέχει και μία μέθοδο void logMessage(String message) που εκτυπώνει ένα μήνυμα ακολουθούμενο από ένα timestamp
- Υλοποιήστε μία utility class με όνομα **MathHelper** με public static μεθόδους για κοινές μαθηματικές λειτουργίες όπως εύρεση του μέγιστου, ελάχιστου, και μέσου όρου ενός array από integers

Εργασίες (2)

- Υλοποιήστε μία utility class με όνομα **StringUtils** με public static μεθόδους για string manipulation, όπως να γίνεται reverse ένα string, να μετατρέπεται ένα string σε uppercase, και να ελέγχεται εάν ένα string είναι παλινδρομικό
- Αναπτύξτε ένα utility class με όνομα **ValidationUtils** με public static μεθόδους για data validation, όπως έλεγχος αν ένα string έχει μήκος 1-31 chars, και αν ένας αριθμός είναι ανάμεσα σε 1 και 10

Εργασίες (3)

- Αναπτύξτε ένα Singleton class με όνομα **StackManager** που υλοποιεί ένα LIFO μηχανισμό. Χρησιμοποιήστε πίνακα strings. Θα πρέπει να παρέχονται μέθοδοι για εισαγωγή στο τέλος (push) και εξαγωγή από το τέλος (pop) καθώς και βοηθητικές μέθοδοι (isFull(), isEmpty() ως private static μέθοδοι)
- Αναπτύξτε ένα Singleton class με όνομα **QueueManager** που υλοποιεί ένα FIFO μηχανισμό. Χρησιμοποιήστε πίνακα ακεραίων. Θα πρέπει να παρέχονται μέθοδοι για εισαγωγή στο τέλος (enqueue) και εξαγωγή από την αρχή (dequeue) καθώς και βοηθητικές μέθοδοι (isFull(), isEmpty() ως private static μέθοδοι)