

```
1
2  /* ////////////////////////////////////////
3     SINGLETON garantit qu'une seule instance d'une classe est créée.
4  */
5
6  // SOLUTION AVEC CLASSE
7
8  class Singleton {
9      private static Singleton INSTANCE; // instance unique
10
11     private Singleton(){ // constructeur privé
12         // ...
13     }
14
15     public static Singleton getInstance(){
16         if(INSTANCE == null) { // crée l'instance au premier appel
17             INSTANCE = new Singleton();
18         }
19         return INSTANCE;
20     }
21
22     public void run(String[] args){
23         // ...
24     }
25
26     public static void main(String[] args){
27         getInstance().run(args);
28     }
29 }
30
31 // SOLUTION AVEC ENUMERATION
32
33 enum Singleton {
34     ENVIRONNEMENT;
35
36     public void run(String[] args){
37         // ...
38     }
39
40     public static void main(String[] args){
41         ENVIRONNEMENT.run(args);
42     }
43 }
44
45 /* ////////////////////////////////////////
46     Builder permet de créer une classe avec un grand nombre d'attributs et
47     qui doit gérer un grand nombre de constructeurs.
48 */
49
50 public class StreetMap{
51     private final Point origin;
52     private final Point destination;
53
54     private final Color waterColor;
55     private final Color landColor;
56     private final Color highTrafficColor;
57     private final Color mediumTrafficColor;
```

```
58     private final Color lowTrafficColor;
59
60     public static class Builder{
61         //required parameters
62         private final Point origin;
63         private final Point destination;
64         // optional parameters initialize with default values
65         private Color waterColor = Color.BLUE;
66         private Color landColor = Color.RED;
67         private Color highTrafficColor = Color.YELLOW;
68         private Color mediumTrafficColor = Color.PURPLE;
69         private Color lowTrafficColor = Color.ORANGE;
70
71         public Builder(Point origin, Point destination){
72             this.origin=origin;
73             this.detination=destination;
74         }
75
76         //faire la meme chose pour chaque parametre
77         public Builder waterColor(Color color){
78             this.waterColor=color;
79             return this;
80         }
81
82         public StreetMap build(){
83             return new StreetMap(this);
84         }
85     }
86
87     private StreetMap(Builder builder){
88         //required parameters
89         origin = builder.origin;
90         destination = builder.destination;
91
92         //optional parameters
93         waterColor = builder.waterColor;
94         landColor = builder.landColor;
95         highTrafficColor = builder.highTrafficColor;
96         mediumTrafficColor = builder.mediumTrafficColor;
97         lowTrafficColor = builder.lowTrafficColor;
98     }
99 }
100
101 public static void main(String args[]){
102     StreetMap s = new StreetMap
103         .Builder(new Point(1,2), new Point(2,3))
104         .landColor(Color.GREY)
105         .waterColor(Color.BLACK)
106         .build();
107 }
108
109 /* ////////////////////////////////////////
110     Factory method permet la création d'objets sans préciser explicitement la
111     classe à utiliser. Les
112     objets sont créés en utilisant une méthode de fabrication redéfinie dans des
113     sous-classes.
114 */
```

```
113
114 // Exemple 1 :
115
116 public class Client {
117
118     public static void main(String[] args) {
119
120         ComplexeIndustriel usinePomme = new UsinePomme();
121         ComplexeIndustriel usinePoire = new UsinePoire();
122
123         Fruit fruit1 = null;
124         System.out.println("Utilisation de la premiere fabrique");
125         fruit1 = usinePomme.getFruit();
126         fruit1.afficheFruit(); // "Je suis une Pomme"
127
128         Fruit fruit2 = null;
129         System.out.println("Utilisation de la seconde fabrique");
130         fruit2 = usinePoire.getFruit();
131         fruit2.afficheFruit(); // "Je suis une Poire"
132     }
133 }
134
135 public abstract class ComplexeIndustriel {
136
137     public Fruit getFruit() {
138         return createFruit();
139     }
140
141     protected abstract Fruit createFruit();
142 }
143
144 public class UsinePomme extends ComplexeIndustriel {
145
146     @Override
147     protected Fruit createFruit() {
148         return new Pomme();
149     }
150 }
151
152 public class UsinePoire extends ComplexeIndustriel {
153
154     @Override
155     protected Fruit createFruit() {
156         return new Poire();
157     }
158 }
159
160 public abstract class Fruit {
161     public abstract void afficheFruit();
162 }
163
164 public class Pomme extends Fruit {
165
166     @Override
167     public void afficheFruit() {
168         System.out.println("Je suis une Pomme");
169     }
170 }
```

```
170
171 public class Poire extends Fruit {
172
173     @Override
174     public void afficheFruit() {
175         System.out.println("Je suis une Poire");
176     }
177 }
178
179 /* ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
180     Abstract Factory method permet la création d'objets sans préciser
181     explicitement la classe à utiliser. Les
182     objets sont créés en utilisant une méthode de fabrication redéfinie dans des
183     sous-classes.
184 */
185
186 public class Client {
187
188     public static void main(String[] args) {
189         ComplexeIndustriel usineCarottePomme = new UsineCarottePomme();
190         ComplexeIndustriel usineHaricotPoire = new UsineHaricotPoire();
191
192         Legume legume = null;
193         Fruit fruit = null;
194         System.out.println("Utilisation de la premiere fabrique");
195         legume = usineCarottePomme.getLegume();
196         fruit = usineCarottePomme.getFruit();
197         legume.afficheLegume(); // "Je suis une Carotte"
198         fruit.afficheFruit(); // "Je suis une Pomme"
199
200         System.out.println("Utilisation de la seconde fabrique");
201         legume = usineHaricotPoire.getLegume();
202         fruit = usineHaricotPoire.getFruit();
203         legume.afficheLegume(); // "Je suis un Haricot"
204         fruit.afficheFruit(); // "Je suis une Poire"
205     }
206 }
207
208 public interface ComplexeIndustriel {
209
210     public Legume getLegume();
211     public Fruit getFruit();
212 }
213
214 public class UsineCarottePomme implements ComplexeIndustriel {
215
216     public Legume getLegume() {
217         return new Carotte();
218     }
219
220     public Fruit getFruit() {
221         return new Pomme();
222     }
223 }
224
225 public class UsineHaricotPoire implements ComplexeIndustriel {
```

```
225     public Legume getLegume() {
226         return new Haricot();
227     }
228
229     public Fruit getFruit() {
230         return new Poire();
231     }
232 }
233
234 public abstract class Legume {
235     public abstract void afficheLegume();
236 }
237
238 public class Carotte extends Legume {
239
240     public void afficheLegume() {
241         System.out.println("Je suis une carotte");
242     }
243 }
244
245 public class Haricot extends Legume {
246
247     public void afficheLegume() {
248         System.out.println("Je suis un Haricot");
249     }
250 }
251
252 public abstract class Fruit {
253     public abstract void afficheFruit();
254 }
255
256 public class Pomme extends Fruit {
257
258     public void afficheFruit() {
259         System.out.println("Je suis une Pomme");
260     }
261 }
262
263 public class Poire extends Fruit {
264
265     public void afficheFruit() {
266         System.out.println("Je suis une Poire");
267     }
268 }
269
```