```java

/* ////////////////////////////////////////////////////////////////////
    SINGLETON garantit qu'une seule instance d'une classe est créée.
*/

// SOLUTION AVEC CLASSE

class Singleton {
    private static Singleton INSTANCE; // instance unique

    private Singleton(){ // constructeur privé
        // ...
    }

    public static Singleton getInstance(){
        if(INSTANCE == null) { // crée l'instance au premier appel
            INSTANCE = new Singleton();
        }
        return INSTANCE;
    }

    public void run(String[] args){
        // ...
    }

    public static void main(String[] args){
        getInstance().run(args);
    }
}

// SOLUTION AVEC ENUMERATION

enum Singleton {
    ENVIRONNEMENT;

    public void run(String[] args){
        // ...
    }

    public static void main(String[] args){
        ENVIRONNEMENT.run(args);
    }
}

/* ////////////////////////////////////////////////////////////////////
    Builder permet de créer une classe avec un grand nombre d'attributs et
    qui doit gérer un grand nombre de constructeurs.
*/

public class StreetMap{
    private final Point origin;
    private final Point destination;

    private final Color waterColor;
    private final Color landColor;
    private final Color highTrafficColor;
    private final Color mediumTrafficColor;
```

```java
    private final Color lowTrafficColor;

    public static class Builder{
        //required parameters
        private final Point origin;
        private final Point destination;
        // optional parameters initialize with default values
        private Color waterColor = Color.BLUE;
        private Color landColor = Color.RED;
        private Color highTrafficColor = Color.YELLOW;
        private Color mediumTrafficColor = Color.PURPLE;
        private Color lowTrafficColor = Color.ORANGE;

        public Builder(Point origin, Point destination){
            this.origin=origin;
            this.detination=destination;
        }

        //faire la meme chose pour chaque parametre
        public Builder waterColor(Color color){
            this.waterColor=color;
            return this;
        }

        public StreetMap build(){
            return new StreetMap(this);
        }
    }

    private StreetMap(Builder builder){
        //required parameters
        origin = builder.origin;
        destination = builder.destination;

        //optional parameters
        waterColor = builder.waterColor;
        landColor = builder.landColor;
        highTrafficColor = builder.highTrafficColor;
        mediumTrafficColor = builder.mediumTrafficColor;
        lowTrafficColor = builder.lowTrafficColor;
    }
}

public static void main(String args[]){
    StreetMap s = new StreetMap
        .Builder(new Point(1,2), new Point(2,3))
        .landColor(Color.GREY)
        .waterColor(Color.BLACK)
        .build();
}

/* ////////////////////////////////////////////////////////////////////
    Factory method permet la création d'objets sans préciser explicitement la classe à utiliser. Les
    objets sont créés en utilisant une méthode de fabrication redéfinie dans des sous-classes.
*/
```

```java

// Exemple 1 :

public class Client {

    public static void main(String[] args) {

        ComplexeIndustriel usinePomme = new UsinePomme();
        ComplexeIndustriel usinePoire = new UsinePoire();

        Fruit fruit1 = null;
        System.out.println("Utilisation de la premiere fabrique");
        fruit1 = usinePomme.getFruit();
        fruit1.afficheFruit(); // "Je suis une Pomme"

        Fruit fruit2 = null;
        System.out.println("Utilisation de la seconde fabrique");
        fruit2 = usinePoire.getFruit();
        fruit2.afficheFruit(); // "Je suis une Poire"
    }
}

public abstract class ComplexeIndustriel {

    public Fruit getFruit() {
        return createFruit();
    }

    protected abstract Fruit createFruit();
}

public class UsinePomme extends ComplexeIndustriel {

    @Override
    protected Fruit createFruit() {
        return new Pomme();
    }
}

public class UsinePoire extends ComplexeIndustriel {
    @Override
    protected Fruit createFruit() {
        return new Poire();
    }
}

public abstract class Fruit {
    public abstract void afficheFruit();
}

public class Pomme extends Fruit {

    @Override
    public void afficheFruit() {
        System.out.println("Je suis une Pomme");
    }
}
```

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

```java
    public Legume getLegume() {
        return new Haricot();
    }

    public Fruit getFruit() {
        return new Poire();
    }
}

public abstract class Legume {
    public abstract void afficheLegume();
}

public class Carotte extends Legume {

    public void afficheLegume() {
        System.out.println("Je suis une carotte");
    }
}

public class Haricot extends Legume {

    public void afficheLegume() {
        System.out.println("Je suis un Haricot");
    }
}

public abstract class Fruit {
    public abstract void afficheFruit();
}

public class Pomme extends Fruit {

    public void afficheFruit() {
        System.out.println("Je suis une Pomme");
    }
}

public class Poire extends Fruit {

    public void afficheFruit() {
        System.out.println("Je suis une Poire");
    }
}
```