```java
/* ////////////////////////////////////////////////////////////////
   Composite permet de créer des structures hiérarchiques pour des relations
   tout-partie.
 */

// peut etre une classe abstraite
interface Element {
    public void print();
}

public abstract class Element{
    protected String nom;
    protected int taille;

    public Element(String nom, int taille) {
        this.nom=nom;
        this.taille=taille;
    }

    public abstract void print();
}

class Fichier implements Element {

    public void print(){
        System.out.Println("je suis un fichier");
    }
}

class Dossier implements Element {

    private List<Element> contenu = new ArrayList <Element>();

    public void print(){
        System.out.Println("je suis un dossier");
        for (Element e : contenu){
            e.print();
        }
    }

    public void add(Element e){
        contenu.add(e);
    }
}

/* ////////////////////////////////////////////////////////////////
  Adapteur permet à une classe d'être utilisée avec une interface qui n'est
  pas la sienne. Il permet d'utiliser des interfaces incompatibles.
 */

/**
 * Définit une interface qui est identifiée
 * comme standard dans la partie cliente.
 */
public interface Standard {

    /**
```

```
58          * L'opération doit multiplier les deux nombres,
59          * puis afficher le résultat de l'opération
60          */
61         public void operation(int pNombre1, int pNombre2);
62     }
63
64     /**
65      * Implémente l'interface "Standard".
66      */
67     public class ImplStandard implements Standard {
68
69         public void operation(int pNombre1, int pNombre2) {
70             System.out.println("Standard : Le nombre est : " + (pNombre1 * pNombre2));
71         }
72     }
73
74     /**
75      * Fournit les fonctionnalités définies dans l'interface "Standard",
76      * mais ne respecte pas l'interface.
77      */
78     public class ImplAdapte {
79
80         public int operationAdapte1(int pNombre1, int pNombre2) {
81             return pNombre1 * pNombre2;
82         }
83
84         /**
85          * Apporte la fonctionalité définie dans l'interface,
86          * mais la méthode n'a pas le bon nom
87          * et n'accepte pas le même paramètre.
88          */
89         public void operationAdapte2(int pNombre) {
90             System.out.println("Adapte : Le nombre est : " + pNombre);
91         }
92     }
93
94     /**
95      * Adapte l'implémentation non standard avec l'héritage.
96      */
97     public class Adaptateur extends ImplAdapte implements Standard {
98
99         /**
100          * Appelle les méthodes non standard
101          * depuis une méthode respectant l'interface.
102          * 1°) Appel de la méthode réalisant la multiplication
103          * 2°) Appel de la méthode d'affichage du résultat
104          * La classe adaptée est héritée, donc on appelle directement les méthodes
105          */
106         public void operation(int pNombre1, int pNombre2) {
107             int lNombre = operationAdapte1(pNombre1, pNombre2);
108             operationAdapte2(lNombre);
109         }
110     }
111
112     // OU BIEN
113     /**
114      * Adapte l'implémentation non standard avec la composition.
```

```
115   */
116  public class Adaptateur implements Standard {
117
118      private ImplAdapte adapte = new ImplAdapte();
119
120      /**
121       * Appelle les méthodes non standard
122       * depuis une méthode respectant l'interface.
123       * 1°) Appel de la méthode réalisant la multiplication
124       * 2°) Appel de la méthode d'affichage du résultat
125       * La classe adaptée compose l'adaptation,
126       * donc on appelle les méthodes de "ImplAdapte".
127       */
128      public void operation(int pNombre1, int pNombre2) {
129          int lNombre = adapte.operationAdapte1(pNombre1, pNombre2);
130          adapte.operationAdapte2(lNombre);
131      }
132  }
133
134  public class Main {
135
136      public static void main(String[] args) {
137          // Création d'un adaptateur
138          final Standard lImplAdapte = new Adaptateur();
139          // Création d'une implémentation standard
140          final Standard lImplStandard = new ImplStandard();
141
142          // Appel de la même méthode sur chaque instance
143          lImplAdapte.operation(2, 4);
144          lImplStandard.operation(2, 4);
145
146          // Affichage :
147          // Adapte : Le nombre est : 8
148          // Standard : Le nombre est : 8
149      }
150  }
151
152  /* ////////////////////////////////////////////////////////////
153   * Decorateur permet d'implémenter une classe puis de lui rajouter des
154   * fonctionnalités.
155   */
156
157  // Déclarations
158  public abstract class Voiture {
159
160      private String nom;
161      private String marque;
162
163      abstract int getPrix();
164      abstract int getPoids();
165  }
166
167  class DS extends Voiture{
168
169      public DS() {
170          this.nom = "DS"; this.marque = "Citroën";
171      }
```

```java
    int getPrix() {return 30000;}
    int getPoids() {return 1500;}
}

// Décorateurs
abstract class VoitureAvecOption extends Voiture{
    Voiture voiture;
}

class VoitureAvecToitOuvrant extends VoitureAvecOption{

    int getPrix() {return voiture.getPrix() + 10000;}
    int getPoids() {return voiture.getPoids() + 15;}
}

//On garde le nom du pattern Decorator pour savoir qu'on wrap un objet
class DSAvecToitOuvrantDecorator extends VoitureAvecToitOuvrant{
    public DSAvecToitOuvrantDecorator(DS ds) {
        this.voiture = ds;
    }
}

public class Main {
    // Implémentation
    public static void main(String[] args) {
        Voiture ds = new DS();
        Voiture dsOption = new DSAvecToitOuvrantDecorator((DS) ds);
    }
}

/* ////////////////////////////////////////////////////////////////
 * Facade permet d'utiliser les fonctionnalités de plusieurs classes à
 * partir d'une seule (Facade)
 */

/**
 * Classe implémentant diverses fonctionnalités.
 */
public class ClasseA {

    public void operation1() {
        System.out.println("Methode operation1() de la classe ClasseA");
    }

    public void operation2() {
        System.out.println("Methode operation2() de la classe ClasseA");
    }
}

/**
 * Classe implémentant d'autres fonctionnalités.
 */
public class ClasseB {

    public void operation3() {
        System.out.println("Methode operation3() de la classe ClasseB");
    }
```

```
229
230         public void operation4() {
231             System.out.println("Methode operation4() de la classe ClasseB");
232         }
233     }
234
235     /**
236      * Présente certaines fonctionnalités.
237      * Dans ce cas, ne présente que la méthode "operation2()" de "ClasseA"
238      * et la méthode "operation41()" utilisant "operation4()" de "ClasseB"
239      * et "operation1()" de "ClasseA".
240      */
241     public class Facade {
242
243         private ClasseA classeA = new ClasseA();
244         private ClasseB classeB = new ClasseB();
245
246         /**
247          * La méthode operation2() appelle simplement
248          * la même méthode de ClasseA
249          */
250         public void operation2() {
251             System.out.println("--> Méthode operation2() de la classe Facade : ");
252             classeA.operation2();
253         }
254
255         /**
256          * La méthode operation41() appelle
257          * operation4() de ClasseB
258          * et operation1() de ClasseA
259          */
260         public void operation41() {
261             System.out.println("--> Méthode operation41() de la classe Facade : ");
262             classeB.operation4();
263             classeA.operation1();
264         }
265     }
266
267     public class FacadePatternMain {
268
269         public static void main(String[] args) {
270             // Création de l'objet "Facade" puis appel des méthodes
271             Facade lFacade = new Facade();
272             lFacade.operation2();
273             lFacade.operation41();
274         }
275     }
276
277     /* ////////////////////////////////////////////////////////////
278      * BRIDGE
279      */
280
281     /**
282      * Définit l'interface de l'implémentation.
283      * L'implémentation fournit deux méthodes
284      */
285     public interface Implementation {
```

```
286
287         public void operationImpl1(String pMessage);
288         public void operationImpl2(Integer pNombre);
289     }
290
291     /**
292      * Sous-classe concrète de l'implémentation
293      */
294     public class ImplementationA implements Implementation {
295
296         public void operationImpl1(String pMessage) {
297             System.out.println("operationImpl1 de ImplementationA : " + pMessage);
298         }
299
300         public void operationImpl2(Integer pNombre) {
301             System.out.println("operationImpl2 de ImplementationA : " + pNombre);
302         }
303     }
304
305     /**
306      * Sous-classe concrète de l'implémentation
307      */
308     public class ImplementationB implements Implementation {
309
310         public void operationImpl1(String pMessage) {
311             System.out.println("operationImpl1 de ImplementationB : " + pMessage);
312         }
313
314         public void operationImpl2(Integer pNombre) {
315             System.out.println("operationImpl2 de ImplementationB : " + pNombre);
316         }
317     }
318
319     /**
320      * Définit l'interface de l'abstraction
321      */
322     public abstract class Abstraction {
323
324         // Référence vers l'implémentation
325         private Implementation implementation;
326
327         protected Abstraction(Implementation pImplementation) {
328             implementation = pImplementation;
329         }
330
331         public abstract void operation();
332
333         /**
334          * Lien vers la méthode operationImpl1() de l'implémentation
335          * @param pMessage
336          */
337         protected void operationImpl1(String pMessage) {
338             implementation.operationImpl1(pMessage);
339         }
340
341         /**
342          * Lien vers la méthode operationImpl2() de l'implémentation
```

```
343         * @param pMessage
344         */
345        protected void operationImpl2(Integer pNombre) {
346            implementation.operationImpl2(pNombre);
347        }
348    }
349
350    /**
351     * Sous-classe concrète de l'abstraction
352     */
353    public class AbstractionA extends Abstraction {
354
355        public AbstractionA(Implementation pImplementation) {
356            super(pImplementation);
357        }
358
359        public void operation() {
360            System.out.println("--> Méthode operation() de AbstractionA");
361            operationImpl1("A"); operationImpl2(1); operationImpl1("B");
362        }
363    }
364
365    /**
366     * Sous-classe concrète de l'abstraction
367     */
368    public class AbstractionB extends Abstraction {
369
370        public AbstractionB(Implementation pImplementation) {
371            super(pImplementation);
372        }
373
374        public void operation() {
375            System.out.println("--> Méthode operation() de AbstractionB");
376            operationImpl2(9); operationImpl2(8); operationImpl1("Z");
377        }
378    }
379
380    public class BridgePatternMain {
381
382        public static void main(String[] args) {
383            // Création des implémentations
384            Implementation lImplementationA = new ImplementationA();
385            Implementation lImplementationB = new ImplementationB();
386
387            // Création des abstractions
388            Abstraction lAbstractionAA = new AbstractionA(lImplementationA);
389            Abstraction lAbstractionAB = new AbstractionA(lImplementationB);
390            Abstraction lAbstractionBA = new AbstractionB(lImplementationA);
391            Abstraction lAbstractionBB = new AbstractionB(lImplementationB);
392
393            // Appels des méthodes des abstractions
394            lAbstractionAA.operation(); lAbstractionAB.operation();
395            lAbstractionBA.operation(); lAbstractionBB.operation();
396        }
397    }
398
```