```java
/* ////////////////////////////////////////////////
 * Command
 */

public interface Command {
   void execute();
}

public class Interrupteur {

   private List<Command> history = new ArrayList<Command>();

   public Interrupteur() {
   }

   public void storeAndExecute(Command cmd) {
      this.history.add(cmd); // optional
      cmd.execute();
   }
}

public class Light {

   public Light() {
   }

   public void turnOn() {
      System.out.println("The light is on");
   }

   public void turnOff() {
      System.out.println("The light is off");
   }
}

public class Allumer implements Command {

   private Light theLight;

   public Allumer(Light light) {
      this.theLight = light;
   }

   public void execute(){
      theLight.turnOn();
   }
}

public class Eteindre implements Command {

   private Light theLight;

   public Eteindre(Light light) {
      this.theLight = light;
   }

   public void execute() {
```

```
58          theLight.turnOff();
59       }
60    }
61
62    public class PressSwitch {
63
64       public static void main(String[] args){
65          Light lamp = new Light();
66          Command allumer = new Allumer(lamp);
67          Command eteindre = new Eteindre(lamp);
68
69          Interrupteur s = new Interrupteur();
70
71          try {
72             if (args[0].equalsIgnoreCase("ON")) {
73                s.storeAndExecute(allumer);
74                System.exit(0);
75             }
76             if (args[0].equalsIgnoreCase("OFF")) {
77                s.storeAndExecute(eteindre);
78                System.exit(0);
79             }
80             System.out.println("Argument \"ON\" or \"OFF\" is required.");
81          } catch (Exception e) {
82             System.out.println("Argument's required.");
83          }
84       }
85    }
86
87    /* ////////////////////////////////////////////////
88     * Iterator
89     */
90
91    public interface Iterator {
92       public boolean hasNext();
93       public Object next();
94    }
95
96    public interface Container {
97       public Iterator getIterator();
98    }
99
100   public class NameRepository implements Container {
101      public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};
102
103      @Override
104      public Iterator getIterator() {
105         return new NameIterator();
106      }
107
108      private class NameIterator implements Iterator {
109         int index;
110
111         @Override
112         public boolean hasNext() {
113            if(index < names.length){
114               return true;
```

```
115             }
116             return false;
117         }
118 
119         @Override
120         public Object next() {
121             if(this.hasNext()){
122                 return names[index++];
123             }
124             return null;
125         }
126     }
127 }
128 
129 public class Main {
130 
131     public static void main(String[] args) {
132         NameRepository namesRepository = new NameRepository();
133 
134         for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
135             String name = (String)iter.next();
136             System.out.println("Name : " + name);
137         }
138 
139         /* Name : Robert, Name : John, Name : Julie, Name : Lora */
140     }
141 }
142 
143 
144 /* ////////////////////////////////////////////////
145  * Observateur
146  */
147 
148 /** Classe représentant un GPS (appareil permettant de connaître sa position). */
149 public class Gps extends Observable{
150         private String position;// Position du GPS.
151         private int precision;// Précision accordé à cette position
152 
153         // Constructeur.
154         public Gps(){
155                 position="inconnue";
156                 precision=0;
157         }
158 
159         // Méthode permettant de notifier tous les observateurs lors d'un
        changement d'état du GPS.
160         public void notifierObservateurs(){
161                 setChanged();// Méthode de l'API.
162                 notifyObservers();// Egalement une méthode de l'API.
163         }
164 
165         // Méthode qui permet de mettre à jour de façon artificielle le GPS.
166         // Dans un cas réel, on utiliserait les valeurs retournées par les capteurs.
167         public void setMesures(String position, int precision){
168                 this.position=position;
169                 this.precision=precision;
170                 notifierObservateurs();
```

```
171           }
172
173           public String getPosition() return position;
174           public int getPrecision() return precision;
175   }
176
177   /** Affiche un résumé en console des informations (position) du GPS. */
178   public class AfficheResume implements Observer {
179         // Méthode appelée automatiquement lors d'un changement d'état du GPS.
180         public void update(Observable o, Object obj){
181                 if(o instanceof Gps){
182                         Gps g = (Gps) o;
183                         System.out.println("Position : "+g.getPosition());
184                 }
185         }
186   }
187
188   public class Main{
189         // Méthode principale.
190         public static void main(String[] args){
191                 // Création de l'objet Gps observable.
192                 Gps g = new Gps();
193                 // Création de deux observeurs AfficheResume et AfficheComplet
194                 AfficheResume ar = new AfficheResume();
195                 // On ajoute AfficheResume comme observeur de Gps.
196                 g.addObserver(ar);
197                 // On simule l'arrivée de nouvelles valeurs via des capteurs.
198                 g.setMesures("N 39°59°993 / W 123°00°000", 4);
199                 // Nouvelle simulation d'arrivée de nouvelles valeurs via des capteurs.
200                 g.setMesures("N 37°48°898 / W 124°12°011", 5);
201         }
202   }
203
204   /* ////////////////////////////////////////////////
205    * Template method permet de deleguer des methodes aux sous classes
206    */
207
208    public abstract class Game {
209      abstract void initialize();
210      abstract void startPlay();
211      abstract void endPlay();
212
213      //template method
214      public final void play(){
215
216         //initialize the game
217         initialize();
218
219         //start game
220         startPlay();
221
222         //end game
223         endPlay();
224      }
225   }
226
227   public class Cricket extends Game {
```

```java

    @Override
    void endPlay() {
        System.out.println("Cricket Game Finished!");
    }

    @Override
    void initialize() {
        System.out.println("Cricket Game Initialized! Start playing.");
    }

    @Override
    void startPlay() {
        System.out.println("Cricket Game Started. Enjoy the game!");
    }
}

public class Football extends Game {

    @Override
    void endPlay() {
        System.out.println("Football Game Finished!");
    }

    @Override
    void initialize() {
        System.out.println("Football Game Initialized! Start playing.");
    }

    @Override
    void startPlay() {
        System.out.println("Football Game Started. Enjoy the game!");
    }
}

public class TemplatePatternMain {
    public static void main(String[] args) {

        Game game = new Cricket();
        game.play();
        System.out.println();
        game = new Football();
        game.play();
    }
}

```