Solutions du chapitre 15

Gestion du multithread

15.3 Établissez que chacune des propositions suivantes est soit vraie, soit fausse. Si elle est fausse, expliquez pourquoi.

a) La méthode **`sleep`** ne consomme pas de temps de processeur pendant qu'un thread sommeille.

RÉP.: Vrai.

b) La déclaration d'une méthode comme **`synchronized`** garantit qu'aucun interblocage ne peut se produire.

RÉP.: Faux. Des interblocages peuvent se produire si on ne déverrouille jamais un objet.

c) Java fournit une possibilité puissante appelée l'héritage multiple.

RÉP.: Faux. Java ne permet que l'héritage unique.

d) Les méthodes **`suspend`** et **`resume`** de **`Thread`** sont obsolètes.

RÉP.: Vrai.

15.4 Définissez chacun des termes suivants.

a) le thread

RÉP.: Un contexte d'exécution individuel d'un programme.

b) le fonctionnement en multithread

RÉP.: La faculté de permettre à plus d'un thread de fonctionner de manière simultannée.

c) l'état prêt

RÉP.: L'état dans lequel un thread est capable de fonctionner, dans la mesure de la disponiblilité du processeur, bien entendu.

d) l'état bloqué

RÉP.: L'état dans lequel le thread ne peut utiliser le processeur. L'état bloqué se manifeste par exemple lorsque le thread lance une requête d'entrée-sortie.

e) la planification avec préemption

RÉP.: Un threrad d'une priorité supérieure entre à l'état d'exécution et reçoit les services d'un processeur. Le thread ainsi perçu avec préemption par le processeur est placé à nouveau à l'état prêt, en fonction de sa priorité.

f) l'interface **`Runnable`**

RÉP.: Une interface qui fournit une méthode **`run`**. En implémentant l'interface **`Runnable`**, toute classe est exécutable dans un thread distinct.

g) le moniteur

RÉP.: Un moniteur surveille en quelque sorte les données partagées parmi des threads. Le moniteur est reponsable du verrouillage d'un objet, c'est-à-dire de n'autoriser qu'un seul thread à exécuter des méthodes synchronisées sur l'objet.

h) la méthode **`notify`**

RÉP.: Elle avertit un thread en attente que le verrou d'un objet a été levé et que ce thread peut dès lors tenter d'obtenir le verrou à son profit.

i) la relation producteur-consommateur

RÉP.: C'est une relation entre un producteur et un consommateur qui partagent des données en commun. Le producteur souhaite généralement produire (comprenez ajouter des informations) et le consommateur souhaite consommer (comprenez retirer ces informations).

15.8 Qu'est-ce que le découpage de temps? Donnez une différence fondamentale de fonctionnement de la planification sur des systèmes Java qui autorisent le découpage de temps, par rapport aux systèmes Java qui ne l'autorisent pas.

RÉP.: Le découpage de temps indique que chaque thread peut utiliser le processeur pendant un intervalle de temps limité.

Lorsque la période d'un thread expire, un thread d'égale priorité obtient une chance de s'exécuter. Les systèmes qui ne permettent pas le découpage de temps ne tentent pas de passer avant les autres threads de même priorité. Les threads en attente ne peuvent s'exécuter tant que le processeur n'a pas achevé sa tâche ou s'ils se retirent du processeur.

15.9 Pourquoi un thread aurait-t-il envie d'appeler **`yield`**?
RÉP.: Dans un environnement qui n'exploite pas le découpage de temps, l'utilisation de la méthode **`yield`** accorde à un thread de même priorité une chance de s'exécuter.

15.10 Quels aspects du développement d'applets Java destinées au Web encouragent-t-ils les concepteurs d'applets à utiliser en abondance **`yield`** et **`sleep`**?
RÉP.: Tous les systèmes n'appliquent pas le découpage de temps, de sorte que l'utilisation de **`yield`** et de **`sleep`** assure aux threads de priorités égales de partager le processeur de manière équivalente sur tous les systèmes.

15.13 Écrivez une instruction en Java qui teste si un thread est vivant.
RÉP.: **`t.isAlive()`**

15.14 a) Qu'est-ce que l'héritage multiple?
RÉP.: La faculté d'hériter de plus d'une classe à la fois.
b) Expliquez pourquoi Java n'offre pas l'héritage multiple.
RÉP.: L'héritage multiple a été jugé comme étant un concept beaucoup trop complexe dans le cadre de Java.
c) Quelle caractéristique Java offre-t-il à la place de l'héritage multiple?
RÉP.: La possibilité d'implémenter des interfaces.
d) Expliquez un usage type de cette caractéristique.
RÉP.: Une classe étend (mot clé **`extends`**) déjà une autre classe mais nécessite des fonctionnalités spéciales telles que le fonctionnement en thread. Puisque Java ne permet pas l'héritage multiple, les fonctionnalités supplémentaires sont distribuées par l'interface (mot clé **`implements`**).
e) En quoi cette caractéristique diffère-t-elle des classes abstraites (**`abstract`**)?
RÉP.: Une interface ne peut contenir que des méthodes abstraites (**`abstract`**) publiques et des variables d'instance **`final`**. Dans une classe abstraite, les variables d'instance ne doivent pas être nécessairement **`final`** et les méthodes peuvent être implémentées.

15.16 Commentez chacun des termes suivants, dans le contexte des moniteurs:
a) moniteur.
RÉP.: Détermine quand le producteur peut produire et le consommateur consommer.
b) producteur.
RÉP.: Un thread qui écrit des données dans une ressource de mémoire partagée.
c) consommateur.
RÉP.: Un thread qui lit des données dans une ressource de mémoire partagée.
d) **`wait`**.
RÉP.: Place un thread dans l'état d'attente jusqu'à ce qu'il reçoive la notification de libération du verrou.
e) **`notify`**.
RÉP.: Avertit un thread en attente qu'un verrou a été libéré.
f) **`InterruptedException`**.
RÉP.: La méthode **`wait`** a la possibilité de lancer une **`InterruptedException`**.
g) **`synchronized`**.
RÉP.: Quand une méthode est déclarée comme **`synchronized`** et quand elle s'exécute, l'objet correspondant qui porte la méthode est verrouillé. Les autres threads sont interdits d'accès aux autres méthodes **`synchronized`** de l'objet tant que le verrou n'est pas libéré.

15.17 *(Le lièvre et la tortue)* Aux exercices du chapitre 7 nous vous avions demandé de simuler la célèbre course du lièvre et de la tortue. Réalisez une nouvelle version de cette simulation qui place, cette fois, chacun des protagonistes dans des threads distincts. Au début de la course, appelez les méthodes **`start`** des deux threads. Exploitez les **`wait`**, **`notify`** et **`notifyAll`** pour synchroniser les activités des animaux.
RÉP.:

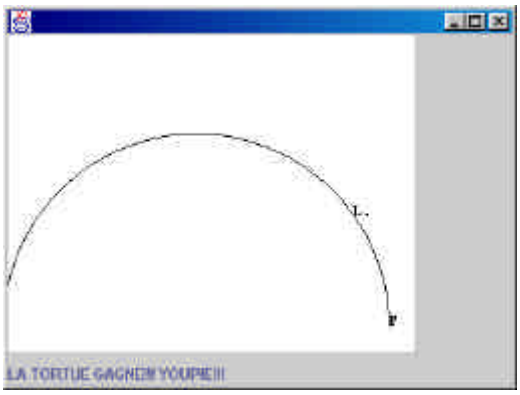
```
1  // Solution de l'exercice 15.17.
2  // Course3.java
3  // Ce programme simule la course du lièvre et de la tortue.
4  //
5  // Pour des raisons liées à l'affichage graphique, la course est étendue à 300.
6  // La montagne est représentée par un demi-cercle, grâce à l'équation du cercle:
```

```
// (x-h)^2 + (y-k)^2 = r^2
// Un léger réglage est apporté à l'équation pour compenser la différence
// de directions du système de coordonnées.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Course3 extends JFrame {
   public final static int FIN_COURSE = 300;
   private int challenger, lapin;
   private Lievre lievre;
   private Tortue tortue;
   private Font f;
   private boolean courseFinie;
   private JLabel etiquetteResultats;

   public Course3()
   {
      etiquetteResultats = new JLabel();
      getContentPane().setLayout( new BorderLayout() );
      getContentPane().add( etiquetteResultats, BorderLayout.SOUTH );
      etiquetteResultats.setText( "Prêt pour la course" );

      challenger = 1;
      lapin = 1;
      courseFinie = false;
      f = new Font( "Monospaced", Font.BOLD, 12 );
      setSize( 400, 300 );
      show();
   }

   public void start()
   {
      etiquetteResultats.setText("Ils sont partis!");

      if ( lievre == null ) {
         lievre = new Lievre( this );
         lievre.start();
      }

      if ( tortue == null ) {
         tortue = new Tortue( this );
         tortue.start();
      }
   }

   public void paint( Graphics g )
   {
      g.setFont( f );
      g.setColor( Color.white );
      g.fillRect( 0, 0, 320, 270 );  // Taille du html.
      g.setColor( Color.black );
      g.drawArc( 0, 100, 300, 300, 0, 180 );

      afficherPositionsEnCours( g );
   }

   public void afficherPositionsEnCours( Graphics g2 )
   {
      int yLievre = ( int ) ( 250 - Math.sqrt( 150 * 150 -
                              Math.pow( lapin - 150, 2 ) ) );

      int yTortue = ( int ) ( 250 - Math.sqrt( 150 * 150 -
                                 Math.pow( challenger - 150, 2 ) ) );

      if ( yLievre == yTortue && lapin == challenger && lapin != FIN_COURSE ) {
```

```
74            g2.drawString( "OUILLE!", lapin, yLievre - 60 );
75            g2.drawString( "L", lapin, yLievre - 20 );  // Faire bondir le lièvre.
76         }
77         else
78            g2.drawString( "L", lapin, yLievre );
79
80         g2.drawString( "T", challenger, yTortue );
81
82         if ( lapin == FIN_COURSE && courseFinie == false ) {
83            etiquetteResultats.setText( "Le lièvre gagne. Bof!" );
84            courseFinie = true;
85         }
86         else if ( challenger == FIN_COURSE && courseFinie == false ) {
87            etiquetteResultats.setText( "LA TORTUE GAGNE!!! YOUPIE!!!" );
88            courseFinie = true;
89         }
90         else if ( !courseFinie && ( lapin > ( challenger + 10 ) ) )
91            etiquetteResultats.setText( "Le lièvre est en tête!" );
92         else if ( !courseFinie && ( lapin < ( challenger - 10 ) ) )
93            etiquetteResultats.setText( "La tortue est en tête!" );
94         else if ( !courseFinie )
95            etiquetteResultats.setText( "Ils sont au coude à coude!" );
96      }
97
98      public synchronized void setPosition( int position, boolean isLievre )
99      {
100        if ( isLievre == true )
101           lapin = position;
102        else
103           challenger = position;
104
105        repaint();
106     }
107
108     public static void main( String args[] )
109     {
110        Course3 app = new Course3();
111
112        app.start();
113
114        app.addWindowListener(
115           new WindowAdapter() {
116              public void windowClosing( WindowEvent e )
117              {
118                 System.exit( 0 );
119              }
120           }
121        );
122     }
123  }
124
125  class Lievre extends Thread {
126     private int lapinPosition;
127     private Course3 montagne;
128
129     public Lievre( Course3 piste )
130     {
131        lapinPosition = 1;
132        montagne = piste;
133     }
134
135     public void run()
136     {
137        while ( lapinPosition != Course3.FIN_COURSE ) {
138           deplacerLievre();
139           montagne.setPosition( lapinPosition, true );
140
```

```
         try {
            sleep( 100 );
         } catch ( Exception e ) {
            System.err.println( "Exception: " + e.toString() );
         }
      }
   }

   public void deplacerLievre()
   {
      int y = ( int ) ( 1 + Math.random() * 10 );
      int deplacementsLievre[] = { 9, 12, 2 };

      if ( y == 3 || y == 4 )        // Grand saut.
         lapinPosition += deplacementsLievre[ 0 ];
      else if ( y == 5 )             // Grand dérapage.
         lapinPosition -= deplacementsLievre[ 1 ];
      else if ( y >= 6 && y <= 8 )   // Petit saut.
         ++lapinPosition;
      else if ( y > 8 )              // Petit dérapage.
         lapinPosition -= deplacementsLievre[ 2 ];

      if ( lapinPosition < 1 )
         lapinPosition = 1;
      else if ( lapinPosition > Course3.FIN_COURSE )
         lapinPosition = Course3.FIN_COURSE;
   }
}

class Tortue extends Thread {
   private int tortuePosition;
   private Course3 montagne;

   public Tortue( Course3 piste )
   {
      tortuePosition = 1;
      montagne = piste;
   }

   public void run()
   {
      while ( tortuePosition != Course3.FIN_COURSE ) {
         deplacerTortue();
         montagne.setPosition( tortuePosition, false );

         try {
            sleep( 100 );
         } catch ( Exception e ) {
            System.err.println( "Exception: " + e.toString() );
         }
      }
   }

   public void deplacerTortue()
   {
      int x = ( int ) ( 1 + Math.random() * 10 );
      int mouvementsTortue[] = { 3, 6 };

      if ( x >= 1 && x <= 5 )             // Avance rapide.
         tortuePosition += mouvementsTortue[ 0 ];
      else if ( x == 6 || x == 7 )        // Glissade.
         tortuePosition -= mouvementsTortue[ 1 ];
      else                                // Avance lente.
         ++tortuePosition;

      if ( tortuePosition < 1 )
         tortuePosition = 1;
```

```
208         else if ( tortuePosition > Course3.FIN_COURSE )
209            tortuePosition = Course3.FIN_COURSE;
210      }
211   }
```



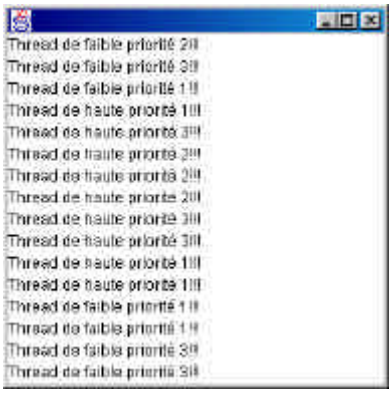
15.18 *(Applications en multithread, comparées aux applications en réseau et coopératrices)* Au chapitre 21 nous étudierons la mise en réseau en Java. Une application Java en multithread peut communiquer simultanément avec plusieurs ordinateurs hôtes. Ceci crée l'opportunité de créer quelques exemples intéressants d'applications collaborant entre elles. Pour anticiper quelque peu sur l'étude de la mise en réseau du chapitre 21, développez des propositions de plusieurs applications réseau possibles en multithread. Après l'assimilation du chapitre 21, réalisez quelques-unes de ces applications.

15.20 Si votre ordinateur accepte le découpage de temps, écrivez un programme Java qui montre le découpage de temps parmi plusieurs threads d'égales priorités. Montrez que l'exécution d'un thread de priorité plus faible est retardée par le découpage de temps des threads de priorités supérieures.

RÉP.:

```
1   // Solution de l'exercice 15.20.
2   // Demo2.java
3   // Ce programme montre l'action de threads de mêmes priorités
4   // et l'évolution du découpage de temps sous Windows 98.
5   import javax.swing.*;
6   import java.awt.*;
7   import java.awt.event.*;
8
9   public class Demo2 extends JFrame {
10     private ThreadHaut haut[];
11     private ThreadBas bas[];
12     private JTextArea sortie;
13
14     public Demo2()
15     {
16        sortie = new JTextArea( 10, 20 );
17        getContentPane().add( sortie );
18        setSize( 300, 300 );
19        show();
20        haut = new ThreadHaut[ 3 ];
21        bas = new ThreadBas[ 3 ];
22     }
23
24     public void start()
25     {
26        for ( int i = 0; i < haut.length; i++ )
27           if ( haut[ i ] == null ) {
28              haut[ i ] = new ThreadHaut( sortie, i + 1 );
29              haut[ i ].start();
30           }
31        for ( int i = 0; i < bas.length; i++ )
32           if ( bas[ i ] == null ) {
```

```
33              bas[ i ] = new ThreadBas( sortie, i + 1 );
34              bas[ i ].start();
35           }
36     }
37
38     public static void main( String args[] )
39     {
40        Demo2 app = new Demo2();
41
42        app.addWindowListener(
43           new WindowAdapter() {
44              public void windowClosing( WindowEvent e )
45              {
46                 System.exit( 0 );
47              }
48           }
49        );
50
51        app.start();
52     }
53  }
54
55  class ThreadHaut extends Thread {
56     private JTextArea affichage;
57     private int id;
58
59     public ThreadHaut( JTextArea a, int n )
60
61     {
62        affichage = a;
63        id = n;
64        setPriority( Thread.MAX_PRIORITY );
65     }
66
67     public void run()
68     {
69        for ( int x = 1; x <= 3; x++ )
70           affichage.append( "Thread de haute priorité " + id + "!!!\n" );
71     }
72  }
73
74  class ThreadBas extends Thread {
75     private JTextArea affichage;
76     private int id;
77
78     public ThreadBas( JTextArea a, int n )
79     {
80        affichage = a;
81        id = n;
82        setPriority( Thread.MIN_PRIORITY );
83     }
84
85     public void run()
86     {
87        for ( int y = 1; y <= 3; y++ )
88           affichage.append( "Thread de faible priorité " + id + "!!!\n" );
89     }
90  }
```



15.21 Écrivez un programme Java qui démontre l'utilisation de **`sleep`** par un thread de haute priorité pour donner aux threads de priorité plus faible une chance de s'exécuter.

RÉP.:

```
1  // Solution de l'exercice 15.21.
2  // Demo2.java
3  // Ce programme montre l'utilisation de sleep dans un thread de haute priorité.
4  import javax.swing.*;
5  import java.awt.*;
6  import java.awt.event.*;
7
8  public class Demo2 extends JFrame {
9     private ThreadHaut haut;
10    private ThreadBas bas;
11    private JTextArea sortie;
12
13    public Demo2()
14    {
15       super( "Demo2" );
16       sortie = new JTextArea( 10, 20 );
17       getContentPane().add( sortie );
18       setSize( 250, 200 );
19       setVisible( true );
20
21       haut = new ThreadHaut( sortie );
22       haut.start();
23
24       bas = new ThreadBas( sortie );
25       bas.start();
26    }
27
28    public static void main( String args[] )
29    {
30       Demo2 app = new Demo2();
31       app.addWindowListener(
32          new WindowAdapter() {
33             public void windowClosing( WindowEvent e )
34             {
35                System.exit( 0 );
36             }
37          }
38       );
39    }
40 }
41
42 class ThreadHaut extends Thread {
43    private JTextArea affichage;
44
45    public ThreadHaut( JTextArea a )
```

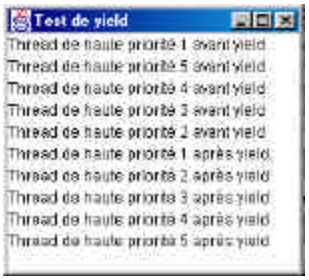
```
46      {
47         affichage = a;
48         setPriority( Thread.MAX_PRIORITY );
49      }
50
51      public void run()
52      {
53         for ( int x = 1; x <= 5; x++ ) {
54            try {
55               sleep( ( int ) ( Math.random() * 200 ) );
56            }
57            catch ( Exception e ) {
58               JOptionPane.showMessageDialog(
59                  null, e.toString(), "Exception",
60                  JOptionPane.ERROR_MESSAGE );
61            }
62
63            affichage.append( "Thread de haute priorité\n" );
64         }
65      }
66   }
67
68   class ThreadBas extends Thread {
69      private JTextArea affichage;
70
71      public ThreadBas( JTextArea a )
72      {
73         affichage = a;
74         setPriority( Thread.MIN_PRIORITY );
75      }
76
77      public void run()
78      {
79         for ( int y = 1; y <= 5; y++ )
80            affichage.append( "Thread de faible priorité!!!\n" );
81      }
82   }
```

15.22 Si votre ordinateur n'accepte pas le découpage de temps, écrivez un programme Java qui présente deux threads exploitant **`yield`** pour permettre l'un à l'autre de s'exécuter.

RÉP.:

```
1   // Solution de l'exercice 15.22.
2   // TestYield.java
3   // Ce programme montre l'utilisation de yield entre threads.
4   import javax.swing.*;
5   import java.awt.*;
6   import java.awt.event.*;
7
8   public class TestYield extends JFrame {
9      private ThreadHaut haut[];
10     private JTextArea sortie;
11     private static final int NBRE_THREADS = 5;
12
13     public TestYield()
```

```
14      {
15         super("Test de yield");
16         sortie = new JTextArea( 10, 30 );
17         getContentPane().add( sortie );
18         setSize( 300, 300 );
19         show();
20      }
21
22      public void start()
23      {
24         haut = new ThreadHaut[ NBRE_THREADS ];
25
26         for ( int i=0; i < haut.length; i++ )
27            haut[ i ] = new ThreadHaut( sortie, i + 1 );
28
29         for ( int i=0; i < haut.length; i++ )
30               haut[ i ].start();
31      }
32
33      public static void main( String args[] )
34      {
35         TestYield app = new TestYield();
36
37         app.addWindowListener(
38            new WindowAdapter() {
39               public void windowClosing( WindowEvent e )
40               {
41                  System.exit( 0 );
42               }
43            }
44         );
45
46         app.start();
47      }
48   }
49
50   class ThreadHaut extends Thread {
51      private JTextArea affichage;
52      private int id;
53
54      public ThreadHaut( JTextArea a, int n )
55      {
56         affichage = a;
57         id = n;
58         setPriority( Thread.MAX_PRIORITY );
59      }
60
61      public void run()
62      {
63         affichage.append( "Thread de haute priorité " + id + " avant yield\n" );
64         yield();
65        affichage.append( "Thread de haute priorité " + id + " après yield\n" );
66      }
67   }
```



15.23 Les deux problèmes qui peuvent se présenter dans des systèmes tels que Java, qui permettent à des threads d'attendre, sont l'interblocage, où un ou plusieurs threads attendent éternellement un événement qui ne peut se produire, et la postposition infinie, où un ou plusieurs threads sont retardés pendant une durée longue et imprévisible. Donnez un exemple concret de chacun de ces problèmes dans un programme Java en multithread.

15.24 *(Lecteurs et scripteurs)* Cet exercice vous demande de développer un moniteur Java qui résolve le célèbre problème du contrôle de simultanéité. Ce problème a été introduit pour la première fois par P. J. Courtois, F. Heymans et D. L. Parnas dans leur article consécutif à leurs recherches, «Concurrent Control with Readers and Writers,» *Communications of the ACM,* Vol. 14, No. 10, octobre 1971, pp. 667–668. L'étudiant curieux sera intéressé par la lecture de l'article relatif à une recherche qui a fait école, de C. A. R. Hoare, «Monitors: An Operating System Structuring Concept,» *Communications of the ACM,* Vol. 17, No. 10, octobre 1974, pp. 549–557. Corrigendum, *Communications of the ACM,* Vol. 18, No. 2, février 1975, p. 95. Le problème des lecteurs et scripteurs est exposé en détail au chapitre 5 du livre d'auteur: Deitel, H. M., *Operating Systems*, Reading, MA: Addison-Wesley, 1990.

Dans le multithread, de nombreux threads peuvent accéder à des données partagées; ainsi que nous l'avons vu, l'accès à des données partagées nécessite une synchronisation soignée pour éviter la corruption des données.

Prenez le cas d'un système de réservation de billets d'avion, par lequel de nombreux clients tentent de réserver des sièges sur des vols particuliers entre des villes données. Toutes les informations concernant les vols et les sièges sont stockées dans une base de données présente en mémoire. La base de données contient de nombreuses entrées qui représentent toutes un siège sur un vol donné pour un jour donné entre des villes données. Dans un scénario type de réservation, le client sonde la base de données, à la recherche du vol optimal qui correspond à ses besoins. Ainsi, le client risque de sonder la base de données quelques fois avant de se décider à tenter de réserver un vol précis. Bien entendu, un siège qui était libre pendant la phase de sondage a peut-être été réservé par quelqu'un d'autre, avant que le client n'ait eu une chance de le réserver, suite à sa décision. Dans ce cas précis, lorsque le client tente de marquer sa réservation, il découvre que les données ont changé et que le vol n'est plus disponible.

Le client qui sonde un peu partout dans la base de données est appelé un *lecteur*. Le client qui tente de réserver un vol est appelé un *scripteur*. En clair, n'importe quel nombre de lecteurs peuvent sonder immédiatement les données partagées, mais chaque scripteur a besoin d'un accès exclusif aux données partagées pour éviter que les données ne subissent de corruption.

Écrivez un programme Java en multithread qui lance plusieurs threads lecteurs et plusieurs threads scripteurs, tentant chacun d'accéder à un même enregistrement de réservation. Un thread scripteur peut effectuer deux types de transactions: **`placerReservation`** et **`annulerReservation`**. Un lecteur ne peut effectuer qu'un seul type de transaction possible: **`requeteReservation`**.

Réalisez une première version de ce programme qui permette l'accès à l'enregistrement de réservation sans synchronisation. Montrez combien l'intégrité de la base de données se dégrade. Réalisez ensuite une version de ce programme qui utilise la synchronisation par moniteur, et utilise **`wait`** et **`notify`** pour renforcer le protocole discipliné d'accès aux données de réservation par les lecteurs et les scripteurs. En particulier, le programme doit permettre à plusieurs lecteurs d'accéder aux données de manière simultanée, lorsqu'aucun scripteur n'est actif. Mais si un scripteur s'active, aucun lecteur n'est autorisé à accéder aux données partagées.

Soyez prudent. Ce problème met en évidence de nombreuses subtilités. Par exemple, que se passe-t-il lorsque plusieurs lecteurs sont actifs et qu'un scripteur souhaite écrire? Si vous autorisez un afflux continu de lecteurs qui se partagent les données, ils pourraient postposer indéfiniment le scripteur, qui risque d'ailleurs de se lasser d'attendre et de décider de conclure l'affaire ailleurs. Pour résoudre ce problème, vous pourriez décider de favoriser les scripteurs par rapport aux lecteurs. Mais, là aussi, il y a un piège. En effet, puisqu'un afflux permanent de scripteurs risque inversement de postposer les lecteurs et, eux aussi, risquent de décider de faire affaire ailleurs. Réalisez le moniteur sur base des méthodes suivantes: **`demarrerLecture`**, qui est appelée par tout lecteur qui souhaite accéder à une réservation, **`arreterLecture`**, qu'appelle tout lecteur qui a fini de lire une réservation, **`demarrerEcriture`**, qui permet à un scripteur de placer une réservation et **`arreterEcriture`**, qu'appelle le scripteur qui a terminé de placer sa réservation.

15.25 Écrivez un programme qui fasse rebondir une balle bleue au sein d'une applet. La balle est initialisée par un événement **`mousePressed`** de bouton de souris pressé. Lorsque la balle frappe un bord de l'applet, elle doit rebondir sur ce bord et continuer dans la direction opposée.

RÉP.:

```
1  // Solution de l'exercice 15.25.
2  // Balle.java
3  // Ce programme fait rebondir une balle autour d'une applet.
4  import javax.swing.*;
5  import java.awt.*;
6  import java.awt.event.*;
7
8  public class Balle extends JApplet
9            implements Runnable, MouseListener {
10    private Thread balleBleue;
11    private boolean xHaut, yHaut, rebond;
12    private int x, y, xDx, yDy;
13
```

```
14      public void init()
15      {
16         xHaut = false;
17         yHaut = false;
18         xDx = 1;
19         yDy = 1;
20         addMouseListener( this );
21         rebond = false;
22      }
23
24      public void mousePressed( MouseEvent e )
25      {
26         if ( balleBleue == null ) {
27            x = e.getX();
28            y = e.getY();
29            balleBleue = new Thread( this );
30            rebond = true;
31            balleBleue.start();
32         }
33      }
34
35      public void stop()
36      {
37         if ( balleBleue != null )
38            balleBleue = null;
39      }
40
41      public void paint( Graphics g )
42      {
43         super.paint( g );
44
45         if ( rebond ) {
46            g.setColor( Color.blue );
47            g.fillOval( x, y, 10, 10 );
48         }
49      }
50
51      public void run()
52      {
53         while ( true ) {
54
55            try {
56               balleBleue.sleep( 100 );
57            }
58            catch ( Exception e ) {
59               System.err.println( "Exception: " + e.toString() );
60            }
61
62            if ( xHaut == true )
63               x += xDx;
64            else
65               x -= xDx;
66
67            if ( yHaut == true )
68               y += yDy;
69            else
70               y -= yDy;
71
72            if ( y <= 0 ) {
73               yHaut = true;
74               yDy = ( int ) ( Math.random() * 5 + 2 );
75            }
76            else if ( y >= 190 ) {
77               yDy = ( int ) ( Math.random() * 5 + 2 );
78               yHaut = false;
79            }
80
```

```
81            if ( x <= 0 ) {
82               xHaut = true;
83               xDx = ( int ) ( Math.random() * 5 + 2 );
84            }
85            else if ( x >= 190 ) {
86               xHaut = false;
87               xDx = ( int ) ( Math.random() * 5 + 2 );
88            }
89
90            repaint();
91         }
92      }
93
94      public void mouseExited( MouseEvent e ) {}
95      public void mouseClicked( MouseEvent e ) {}
96      public void mouseReleased( MouseEvent e ) {}
97      public void mouseEntered( MouseEvent e ) {}
98   }
```

15.26 Modifiez le programme de l'exercice 15.25 pour ajouter une nouvelle balle chaque fois que l'utilisateur clique de la souris. Proposez ainsi au moins 20 balles. Choisissez au hasard la couleur de chaque nouvelle balle.