BÀI 9.
ĐIỀU KHIỂN LUỒNG

Nội dung

- Tạo và điều khiển luồng trong Java
- Lập trình đa luồng trong Java
- Đa luồng trên giao diện chương trình
- Deadlock và Livelock

1. LUỒNG TRONG JAVA

3

Khái niệm cơ bản

- Tiến trình
- Luồng
- Trong Java: Luồng là đơn vị nhỏ nhất của đoạn mã có thể thực thi được để thực hiện một công việc riêng biệt
- Java hỗ trợ đa luồng,
  - Có khả năng làm việc với nhiều luồng.
  - Một ứng dụng có thể bao hàm nhiều luồng.
  - Mỗi luồng được đăng ký một công việc riêng biệt, mà chúng được thực thi đồng thời với các luồng khác.
- Đặc điểm đa luồng
  - Đa luồng giữ thời gian nhàn rỗi của hệ thống thành nhỏ nhất. (tận dụng tối đa CPU)
  - Trong đa nhiệm, nhiều chương chương trình chạy đồng thời, mỗi chương trình có ít nhất một luồng trong nó.

4

Tạo và quản lý luồng

- Khi chương trình Java thực thi hàm main() tức là luồng main được thực thi. luồng này được tạo ra một cách tự động. tại đây :
  - Các luồng con sẽ được tạo ra từ đó
  - Nó là luồng cuối cùng kết thúc việc thực hiện.
  - Khi luồng chính ngừng thực thi, chương trình bị chấm dứt
- Luồng có thể được tạo ra bằng 2 cách:
  - Kế thừa từ lớp Thread
  - Thực thi từ giao diện Runnable.

5

Lớp Thread

```
public class Thread extends Runable(){
   public static final int MAX_PRIORITY = 10;
   public static final int MIN_PRIORITY = 1;
   public static final int NORM_PRIORITY = 5;

   //Nested class
   static class State(){};//Trạng thái của luồng
   //Xử lý sự kiện luồng bị dừng do không bắt ngoại lệ
   static interface UncaughtExceptionHandler(){}

   //Constructor
   public Thread(){};
   public Thread(Runable target);
   public Thread(Runable target, String threadName);

   //public methods...
}
```

6

Một số phương thức chính

- `void start()`: bắt đầu thực thi luồng
- `void run()`: thực thi luồng. Mặc định được gọi trong phương thức `start()`
- `void setName(String name)`: đặt tên cho luồng
- `void setPriority(int priority)`: thiết lập độ ưu tiên
- `void interrupt()`: ngắt luồng đang thực thi
- `final void join()`: chờ luồng kết thúc
- `final void join(long milisecond)`: chờ luồng kết thúc
- `final void join(long milisecond, int nanosecond)`: chờ luồng kết thúc
- `final boolean isAlive()`: trả lại `true` nếu luồng còn đang thực thi
- `Thread.State getState()`: Trả lại trạng thái của luồng

7

Một số phương thức static

- `void yield()`: nhường các luồng có cùng mức ưu tiên thực thi trước
- `void sleep(long millisec)`: tạm dừng luồng trong khoảng thời gian tối thiểu nào đó, nhưng vẫn giữ quyền điều khiển
  - Ủy nhiệm xử lý ngoại lệ `InterruptException` cho phương thức gọi
- `void sleep(long millisec, int nanosecond)`
- `Thread currentThread()`

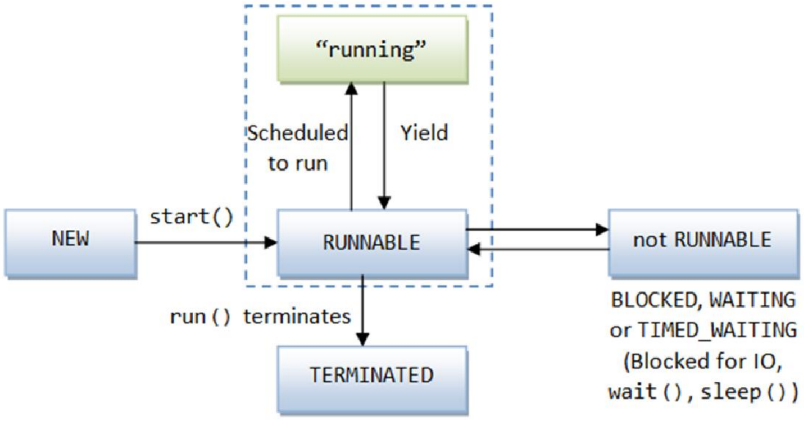
8

Trạng thái của luồng

- NEW: luồng được tạo, chưa thực thi
- RUNNABLE: có thể thực thi
- BLOCKED: luồng bị tạm khóa
- WAITING: chờ các luồng khác thực thi
- TIMED_WAITING: chờ với thời gian xác định
- TERMINATED: kết thúc luồng

9

Vòng đời của một luồng



10

Tạo thread(1) – Kế thừa lớp Thread

```
class MyThread extends Thread{
      //Ghi đè phương thức run() của lớp cha
      public void run()
      {
            //do something
      }

       //Định nghĩa các phương thức khác
}
```

11

Tạo thread(1) – Kế thừa lớp Thread

```
class OtherMyThread extends Thread{
   private Thread t;
   //Ghi đè phương thức run() của lớp cha
   public void run()
   {
      //do something
   }
   //Ghi đè phương thức start() của lớp cha
   public void start(){
      if (t == null)
         {
            t = new Thread (this);
            t.start ();
         }
   }
       //Định nghĩa các phương thức khác
}
```

12

Ví dụ

```
class PingPongThread extends Thread {
   private String word;
   private int delay;
   PingPongThread(String s, int d){
        this.word =s;
        this.delay=d;
   }
   public void run(){
        try{
           for(int i = 1; i <= 10; i++){
                System.out.print(word + " " + i);
                sleep(delay);}
                }
        catch(InterruptedException e){
                System.out.println("Thread " + word +
                                   "interrupted.");}
   }
   public static void main(String[] args){
        new PingPongThread("ping",500).start();
        new PingPongThread("PONG",1000).start();
}
```

13

Ví dụ - Cách viết khác

```
class PingPongThread extends Thread {
   private String word;
   private int delay;
   private Thread t;

   PingPongThread(String s, int d){
        this.word =s;
        this.delay=d;
   }
   public void run(){
        try{
           for(int i = 1; i <= 10; i++){
                System.out.print(word + " " + i);
                sleep(delay);}
                }
        catch(InterruptedException e){
                System.out.println("Thread " + word +
                                   "interrupted.");
        }
   }
```

14

Ví dụ - Cách viết khác(tiếp)

```
    public void start(){
        if(t==null){
            t = new Thread(this);
            t.start();
        }
    }
    public static void main(String[] args){
        new PingPongThread("ping",500).start();
        new PingPongThread("PONG",1000).start();
}
```

15

Giao diện Runable

```
public interface Runnable{
    public void run();
}
```

- Kế thừa từ Thread hay triển khai Runable?
  - Runable đơn giản hơn, phù hợp khi chúng ta chỉ quan tâm đến luồng thực thi những gì bằng cách ghi đè phương thức run()
  - Lớp triển khai từ Runable có thể kế thừa từ lớp khác
  - Thread cung cấp nhiều phương thức, cho phép điều khiển luồng, kiểm tra các trạng thái của luồng
  - Lớp kế thừa từ Thread không thể kế thừa thêm từ lớp khác

16

Tạo Thread(2)-Triển khai Runable

```
class MyThread implements Runnable{
      //Định nghĩa phương thức run()
      public void run()
      {
            //do something
      }

      //Định nghĩa các phương thức khác của lớp
}
```

17

Ví dụ

```
class PingPongRunable implements Runable{
   private String word;
   private int delay;
   PingPongRunable(String s, int d){
       this.word =s;
       this.delay=d;
   }
   public void run(){
       try{
          for(int i = 1; i <= 10; i++){
              System.out.print(word + " " + i);
              sleep(delay);}
              }
       catch(InterruptedException e){
          System.out.println("Thread " + word +
                                          "interrupted.");
       }
   }
```

18

Ví dụ

```
   public static void main(String[] args){
       Runable ping = new PingPongRunable("ping",500);
       Runable pong = new PingPongThread("PONG",1000);
       new Thread(ping).start();
       new Thread(pong).start();
}
```

19

Xử lý luồng trên giao diện

- Xem file UnresponsiveUI.java

```
      btnStart.addActionListener(new ActionListener() {
         @Override
         public void actionPerformed(ActionEvent evt) {
            stop = false;
            for (int i = 0; i < 100000; ++i) {
               if (stop) break;
               tfCount.setText(count + "");
               ++count;
            }
         }
      });

      btnStop.addActionListener(new ActionListener() {
         @Override
         public void actionPerformed(ActionEvent evt) {
            stop = true;
         }
      });
```

20

Tại sao đoạn mã trên thất bại?

- Các luồng được tạo ra bởi chương trình

1. Luồng main được tạo ra bởi phương thức `main()`
2. Lời gọi `SwingUtilities.invokeLater()` tạo ra 3 luồng AWT-Windows, AWT-Shutdown, AWT-EventQueue-0
   - Luồng AWT-EventQueue-0 là luồng duy nhất xử lý các sự kiện trên cửa sổ đồ họa
3. Khi phương thức `main()` hoàn thành, luồng main đóng lại, luồng DestroyJavaVM được tạo ra

→Khi nhấp nút Start, phương thức `actionPerformed()` thực thi trên luồng AWT-QeventQueue-0. Luồng này vào vòng lặp for và không thể xử lý các sự kiện khác

→Giải pháp: tạo luồng riêng cho phương thức `actionPerformed()`

21

Đa luồng xử lý sự kiện trên giao diện

```
btnStart.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent evt) {
       stop = false;
       // Create our own Thread to do the counting
       Thread t = new Thread() {
          @Override
          public void run() {
             for (int i = 0; i < 100000; ++i) {
                if (stop) break;
                tfCount.setText(count + "");
                ++count;
             }
          }
       };
       t.start();  // call back run()
    }
 });
```

Nhưng luồng mới thực hiện vòng lặp, không cho các luồng khác khởi động

22

Đa luồng trên giao diện (tiếp)

```
btnStart.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent evt) {
       stop = false;
       // Create our own Thread to do the counting
       Thread t = new Thread() {
          @Override
          public void run() {
             for (int i = 0; i < 100000; ++i) {
                if (stop) break;
                tfCount.setText(count + "");
                ++count;
                try{
                   sleep(10);
                }catch(InterruptedException ex) {}
             }
          }
       };
       t.start();  // call back run()
    }
 });
```

23

2. ĐỒNG BỘ LUỒNG

24

Đồng bộ luồng

- Khi có nhiều luồng cùng truy cập vào một tài nguyên, cần đồng bộ luồng để tránh các luồng "giẫm chân nhau", thậm chí gây hỏng tài nguyên
- Cơ chế đồng bộ luồng của Java:
  - Mỗi đối tượng trong Java có một khóa
  - Khi có một luồng truy cập vào đối tượng, khóa này mặc định được điều khiển bởi luồng đó
  - Khi có nhiều luồng đồng thời cùng truy cập, chỉ luồng nào có khóa mới được truy cập, các luồng khác phải chờ.

25

Ví dụ - Truy cập đa luồng không đồng bộ

```
public class NonSynchronizedCounter {
   private static int count = 0;

   public static void increment() {
      ++count;
      System.out.println("Count is " + count + " - "
                                    + System.nanoTime());
   }

   public static void decrement() {
      --count;
      System.out.println("Count is " + count + " - " +
                           System.nanoTime());
   }
}
```

26

Ví dụ - Truy cập đa luồng không đồng bộ

```
public class TestNonSynchronizedCounter {
   public static void main(String[] args) {
      Thread threadIncrement = new Thread() {
         @Override
         public void run() {
            for (int i = 0; i < 5; ++i)
               NonSynchronizedCounter.increment();
         }
      };
      Thread threadDecrement = new Thread() {
         @Override
         public void run() {
            for (int i = 0; i < 5; ++i)
               NonSynchronizedCounter.decrement();
          }
      };
      threadIncrement.start();
      threadDecrement.start();
   }
}
```

27

Kết quả thực thi

- Kết quả khác nhau ở những lần chạy khác nhau. Ví dụ

```
Count is 0 - 18075747816257
Count is 0 - 18075747816724
Count is 1 - 18075748264109
Count is 0 - 18075748363009
Count is 1 - 18075748436252
Count is 1 - 18075748588334
Count is 0 - 18075748533286
Count is 2 - 18075748677905
Count is 1 - 18075748779604
Count is 0 - 18075748965276
```

```
Count is 0 - 17995912995995
Count is 0 - 17995912995062
Count is -1 - 17995913534816
Count is 0 - 17995913652377
Count is -1 - 17995913769005
Count is 0 - 17995913874903
Count is -1 - 17995913991531
Count is 0 - 17995914105360
Count is -1 - 17995914182335
Count is 0 - 17995914272372
```

28

Giải thích

- Thực hiện lệnh `++count;` gồm 3 bước:
  - Bước 1: Lấy giá trị của `count` từ bộ nhớ
  - Bước 2: Cộng 1 vào giá trị
  - Bước 3: Cất kết quả vào bộ nhớ
- Thực hiện lệnh `--count` tương tự
- Hai luồng khác nhau cùng truy cập tới giá trị `count` ở những thời điểm khác nhau trên bộ nhớ. Ví dụ:
  - `count` có giá trị là 0
  - Luồng `threadIncrement` đang thực hiện bước 2 thì luồng `threadDecrement` truy cập vào bộ nhớ lấy ra giá trị của `count`
  - Luồng `threadIncrement` cất giá trị mới (1) vào bộ nhớ và chuẩn bị thực hiện phương thức hiển thị System.out.println(), luồng `threadDecrement` cất giá trị sau khi biến đổi (-1) vào bộ nhớ
  - Luồng `threadIncrement` hiển thị kết quả là -1

29

Từ khóa synchronized

- Khi một đối tượng, phương thức hoặc một đoạn mã được đánh dấu là `synchronized`, luồng nào truy cập tới phải chờ khóa → cho phép đồng bộ các luồng

```
// synchronized a method
public synchronized void methodA() { ..... }

public void methodB() {
// synchronized a block of codes
synchronized(this) {
      ......
   }
// synchronized a block of codes based on another object
synchronized(anObject) {
      ......
   }
   ......
}
```

30

Đồng bộ luồng – Cách tiếp cận 1

```
public class SynchronizedCounter {
   private static int count = 0;

   public synchronized static void increment() {
      ++count;
      System.out.println("Count is " + count + " - "
                                    + System.nanoTime());
   }

   public synchronized static void decrement() {
      --count;
      System.out.println("Count is " + count + " - " +
                            System.nanoTime());
   }
}
```

31

Đồng bộ luồng - Cách tiếp cận 2

```
public class NonSynchronizedCounter {
   private static int count = 0;
   public void increment() {
      synchronized(this){
          ++count;
          System.out.println("Count is " + count + " - "
                                    + System.nanoTime());
       }
   }
 public void decrement() {
      synchronized(this){
          --count;
          System.out.println("Count is " + count + " - "
                                    + System.nanoTime());
       }
   }
}
```

32

Đồng bộ luồng - Cách tiếp cận 3

```
public class NonSynchronizedCounter {
   private static int count = 0;

   public void increment() {
      ++count;
      System.out.println("Count is " + count + " - "
                                    + System.nanoTime());
   }

   public void decrement() {
      --count;
      System.out.println("Count is " + count + " - " +
                           System.nanoTime());
   }
}
```

33

Đồng bộ luồng - Cách tiếp cận 3

```
public class SynchronizedTestCounter {
   public static void main(String[] args) {
       NonSynchronizedCounter counter = new
                            NonSynchronizedCounter();
       Thread threadIncrement = new Thread() {
            @Override
            public void run() {
            synchronized(counter){
              for (int i = 0; i < 5; ++i)
                   counter.increment();
          }
         }
       };
```

34

Đồng bộ luồng - Cách tiếp cận 3

```
        Thread threadDecrement = new Thread() {
            @Override
            public void run() {
                synchronized(counter){
                        for (int i = 0; i < 5; ++i)
                            counter.decrement();
                }
            };
        threadIncrement.start();
        threadDecrement.start();
    }
}
```

35

Hạn chế của synchronized

- Không tận dụng triệt để tài nguyên (biến đa luồng thành đơn luồng)
- Các phương thức/đoạn mã synchronized có tốc độ thực hiện chậm

→Kỹ thuật "chờ-báo" (wait-notify): sử dụng các phương thức của lớp Object

- wait(): giúp một luồng chờ một sự kiện xảy ra
- notify(): thông báo cho ít nhất 1 luồng về sự kiện xảy ra
- notifyAll(): thống báo cho tất cả các luồng về sự kiện
- Chỉ được gọi trong các khối lệnh được chỉ định đồng bộ bằng synchronized

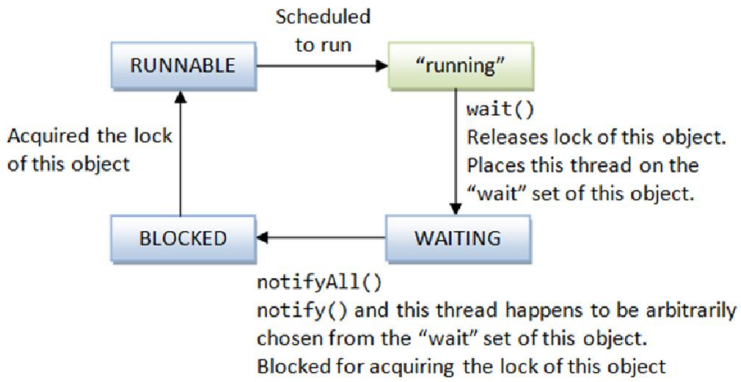
36

Các phương thức

- *public final void wait(long timeout) throws InterruptedException*
  - luồng hiện thời chờ cho tới khi được cảnh báo hoặc một khoảng thời gian timeout nhất định. Nếu timeout bằng 0 thì phương thức sẽ chỉ chờ cho tới khi có cảnh báo về sự kiện.
- public final void notify()
  - Cảnh báo ít nhất một luồng đang chờ một sự kiện
- public final void notifyAll()
  - Phương thức này thông báo báo tất cả các luồng đang chờ một sự kiện. Trong số các luồng đã được thông báo, luồng nào có độ ưu tiên cao nhất thì sẽ chạy trước tiên.

37

Vòng đời của luồng với wait-notify



38

Ví dụ

```
public class MessageBox {
   private String message;
   private boolean hasMessage;

   // producer phát ra một thông báo
   public synchronized void putMessage(String message) {
      while (hasMessage) {
         // có thông báo chưa được lấy
         try {
            wait();  // nhả khóa
         } catch (InterruptedException e) { }
      }
      // yêu cầu khóa và tiếp tục
      hasMessage = true;
      this.message = message + " Put @ " + System.nanoTime();
      notify();
   }
```

39

Ví dụ

```
// consumer lấy thông báo và hiển thị
   public synchronized String getMessage() {
      while (!hasMessage) {
         // không có thông báo mới
         try {
            wait();  // nhả khóa
         } catch (InterruptedException e) { }
      }
      // yêu cầu khóa để thực hiện
      hasMessage = false;
      notify();
      return message + " Get @ " + System.nanoTime();
   }
}
```

40

Ví dụ

```
public class TestMessageBox {
   public static void main(String[] args) {
      final MessageBox box = new MessageBox();

      Thread producerThread = new Thread() {
         @Override
         public void run() {
            System.out.println("Producer thread started...");
            for (int i = 1; i <= 6; ++i) {
               box.putMessage("message " + i);
               System.out.println("Put message " + i);
            }
         }
      };
```

41

Ví dụ

```
      Thread consumerThread1 = new Thread() {
         @Override
         public void run() {
            System.out.println("Consumer thread 1
                                started...");
            for (int i = 1; i <= 3; ++i) {
               System.out.println("Consumer thread 1 Get " +
                                box.getMessage());
            }
         }
      };
```

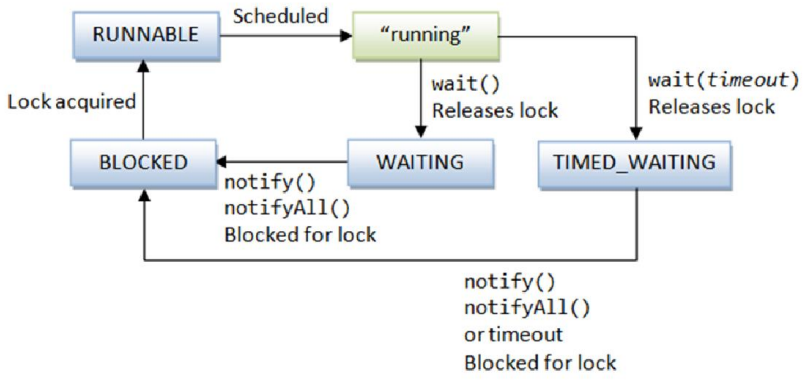
42

Ví dụ

```
    Thread consumerThread2 = new Thread() {
        @Override
        public void run() {
            System.out.println("Consumer thread 2
                    started...");
            for (int i = 1; i <= 3; ++i) {
                System.out.println("Consumer thread 2 Get " +
                    box.getMessage());
            }
        }
    };
    consumerThread1.start();
    consumerThread2.start();
    producerThread.start();
  }
}
```

43

wait() với timeout



44

3. ĐA LUỒNG TRÊN JAVA SWING

45

Cập nhật nội dung trên cửa sổ giao diện

```
public class MySwingApp extends JFrame{
   //Constructor
   pulic MySwingApp(){
        //Add Swing components...
        //Define an ActionListener to perform update at
        //regular interval
        ActionListener updateTask = new ActionListener() {
            @Override
          public void actionPerformed(ActionEvent evt) {
             update();   // updating method
             repaint();  // Refresh the JFrame
          }
       };
        new Timer(100, updateTask).start();
   }
   //Updating method
   public void update(){
        //do something to update content on window
   }
}
```

46

Cập nhật giao diện – Giải pháp khác

```
public class MySwingApp extends JFrame{
   //Constructor
   pulic MySwingApp(){
        //Add Swing components...
        //Create new thread to update
        Thread updateThread = new Thread() {
           @Override
           public void run(){
                while(true){
                   update();
                   repaint();
                   try{
                        Thread.sleep(100);
                   }catch(InterruptedException ignore) {}
                }
           }
   }
   //Updating method
   public void update(){
        //do something to update content on window
   }
}
```

47

javax.swing.SwingWorker<T,V>

- Chương trình với giao diện Java Swing hoạt động như thế nào?(Xem lại slide 17-20)
  1. Luồng main được tạo ra bởi phương thức `main()`
  2. Lời gọi `SwingUtilities.invokeLater()` tạo ra 3 luồng AWT-Windows, AWT-Shutdown, AWT-EventQueue-0
     - Luồng AWT-EventQueue-0 là luồng duy nhất xử lý các sự kiện trên cửa sổ đồ họa, còn gọi là luồng EDT
  3. Khi phương thức `main()` hoàn thành, luồng main đóng lại, luồng DestroyJavaVM được tạo ra
- Không nên thực hiện các thao tác "nặng" tính toán trên luồng EDT(Ví dụ: sử dụng vòng lặp)
- Làm cách nào để các luồng chạy ở phần sau giao diện (back-end)có thể giao tiếp với EDT → Swing Worker

48

javax.swing.SwingWorker<T,V>

```
// Thực hiện các thao tác tính toán ở luồng back-end
protected abstract T doInBackground() throws Exception

//Thực thi trên luồng EDT sau khi phương thức
//doInBackground() hoàn thành
protected void done()

//Đợi doInBackground() và lấy kết quả. Việc gọi phương
//thức get() trên luồng EDT sẽ khóa các sự kiện khác cho
//tới khi SwingWorker thực thi xong
public final T get() throws InterruptedException,
                                ExecutionException

//Thực thi StringWorker
public final void execute()
```

49

javax.swing.SwingWorker<T,V>

```
//Hủy thực thi SwingWorker
public final boolean cancel(boolean mayInterruptIfRunning)

//Trả về true nếu StringWorker đã thực thi xong
public final boolean isDone()

//Trả về true nếu StringWorker bị hủy trước khi thực thi
//xong
public final boolean isCancelled()
```

50

SwingWoker-Ví dụ

- Xem file SwingWorkerCounter.java

```
final SwingWorker<String, Void> worker = new
SwingWorker<String, Void>() {
        /** Schedule a compute-intensive task in a
         background thread */
        @Override
        protected String doInBackground() throws Exception {
           // Sum from 1 to a large n
           long sum = 0;
           for (int number = 1; number < 1000000000;
                                              ++number) {
              sum += number;
           }
           return sum + "";
        }
```

51

SwingWoker-Ví dụ(tiếp)

```
    /** Run in event-dispatching thread after
      doInBackground() completes */
        @Override
        protected void done() {
           try {
              // Use get() to get the result of
              //doInBackground()
              String result = get();
              // Display the result in the label
              lblWorker.setText("Result is " + result);
           } catch (InterruptedException e) {
              e.printStackTrace();
           } catch (ExecutionException e) {
              e.printStackTrace();
           }
        }
     };
```

52

SwingWoker-Ví dụ(tiếp)

```
btnStartWorker = new JButton("Start Worker");
add(btnStartWorker);
btnStartWorker.addActionListener(new
                              ActionListener() {
   @Override
   public void actionPerformed(ActionEvent e) {
      worker.execute();  // start the worker thread
      lblWorker.setText("  Running...");
      btnStartWorker.setEnabled(false);
   }
});
lblWorker = new JLabel("  Not started...");
add(lblWorker);
```

53

4. TIẾN TRÌNH TREO

54

Deadlock và Livelock

- Deadlock là tình trạng các luồng phải chờ nhau vô hạn
- Deadlock thường xảy ra khi các bên chờ nhau giải phóng tài nguyên. Ví dụ: Luồng 1 đang gọi phương thức synchronized của đối tượng X và chờ khóa của đối tượng Y. Luồng 2 đang gọi phương thức synchronized của đối tượng Y và chờ khóa của đối tượng X.
- Livelock xảy ra khi trong chuỗi các luồng có lời gọi tới nhau, một luồng nào đó rơi vào trạng thái bận, làm cho tất cả các luồng sinh ra nó bị khóa.
- Starvation xảy ra khi tài nguyên bị một luồng chiếm dụng trong thời gian quá dài

55

Deadlock – Ví dụ

```
public class TestThread {
   public static Object Lock1 = new Object();
   public static Object Lock2 = new Object();
   private static class ThreadDemo1 extends Thread {
      public void run() {
         synchronized (Lock1) {
            System.out.println("Thread 1: Holding lock
                                          1...");
            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 1: Waiting for lock
                                          2...");
            synchronized (Lock2) {
               System.out.println("Thread 1: Holding lock 1 &
                                          2...");
            }
         }
      }
   }
```

56

Deadlock – Ví dụ(tiếp)

```
private static class ThreadDemo2 extends Thread {
   public void run() {
      synchronized (Lock2) {
         System.out.println("Thread 2: Holding lock 2...");
         try { Thread.sleep(10); }
         catch (InterruptedException e) {}
         System.out.println("Thread 2: Waiting for lock 1...");
         synchronized (Lock1) {
            System.out.println("Thread 2: Holding lock
                                                  1 & 2...");
         }
      }
   }
}
 public static void main(String args[]) {

   ThreadDemo1 T1 = new ThreadDemo1();
   ThreadDemo2 T2 = new ThreadDemo2();
   T1.start();
   T2.start();
}
```

57

Deadlock – Ví dụ khác

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s"
                + "  has bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        }
```

58

Deadlock – Ví dụ khác(tiếp)

```
    public synchronized void bowBack(Friend bower) {
             System.out.format("%s: %s"
                 + " has bowed back to me!%n",
                 this.name, bower.getName());
         }
     }

    public static void main(String[] args) {
         final Friend alphonse =
             new Friend("Alphonse");
         final Friend gaston =
             new Friend("Gaston");
         new Thread(new Runnable() {
             public void run() { alphonse.bow(gaston); }
         }).start();
         new Thread(new Runnable() {
             public void run() { gaston.bow(alphonse); }
         }).start();
     }
}
```

59