Lecture 02: Concurrency (Part 2)

(20 slides)

Review

In the previous session:

- Definitions: Program, Process, Thread
- Multi-processing system
- Multi-threading programming in Java
- Thread Fundamentals in Java
- Thread states

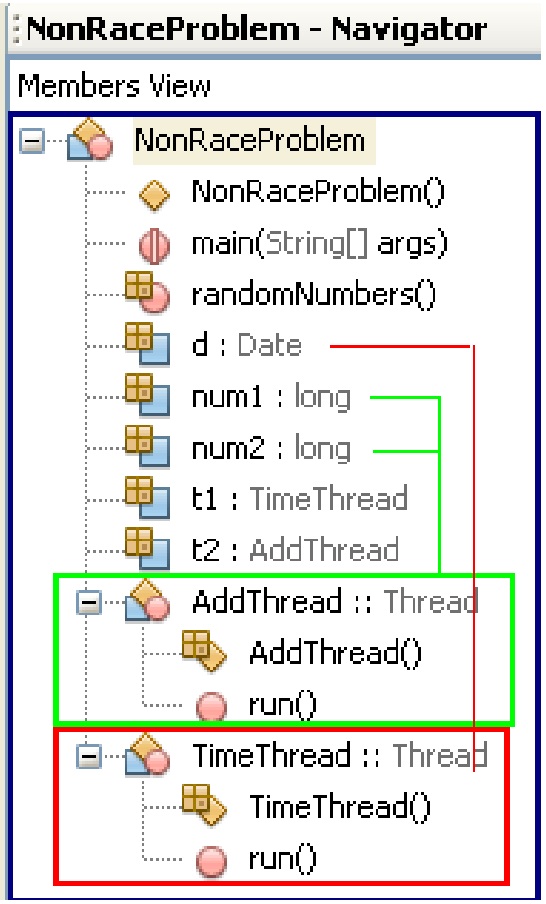
Objectives

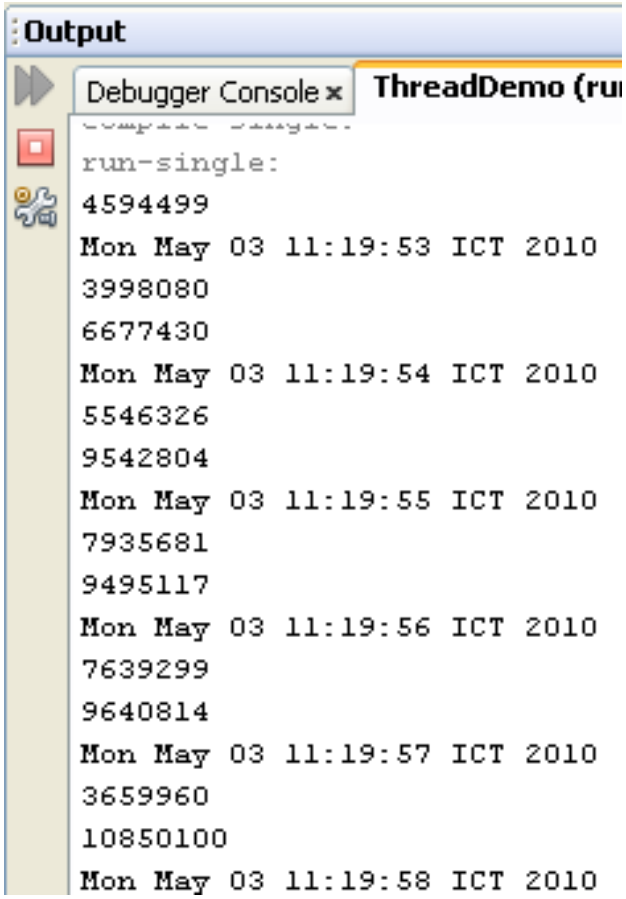
- How to develop a non-race multi-thread applications
- How to develop a multi-thread applications in which some threads accessing common resources?
- **Contents:**
  - Demonstrations
  - Monitors, Waiting and Notifying
  - Deadlocks

Non-race Demonstration

This program contains 2 threads:

- The first thread that will print out the system time for every second.
- The second will print out the sum of two random integers for every half of a second.





Non-race Demonstration…

```java
import java.util.Date;
public class NonRaceProblem {
    Date d=null;
    long num1=0, num2=0;
    // 2 threads of inner Threads, declared belo
    TimeThread t1 = new TimeThread ();
    AddThread t2 = new AddThread ();

    public NonRaceProblem() {
        d=new Date(System.currentTimeMillis());
        randomNumbers(); t1.start(); t2.start();
    }
    void randomNumbers(){
        num1= Math.round(Math.random()*10000000);
        num2= Math.round(Math.random()*100000
    }

    public static void main(String args[]){
        NonRaceProblem obj= new NonRaceProblem();
    }
}
```

```java
    // Inner class 1: Thread for printing out the time
    class TimeThread extends Thread{
        TimeThread() { super(); }
        public void run() {
            while (true){
                try {
                    System.out.println(d);
                    this.sleep(1000);
                    d=new Date(System.currentTimeMillis());
                }
                catch(java.lang.InterruptedException e){
                    e.printStackTrace();
                }
            }
        }
```

```java
// Inner class 2: Thread for printing out sum of 2 numbers
class AddThread extends Thread{
    AddThread() { super(); }
    public void run() {
        while (true){
            try { System.out.println(num1+ num2);
                randomNumbers();
                this.sleep(500);
            }
            catch(java.lang.InterruptedException e){
                e.printStackTrace();
            }
        }
    }
}
```

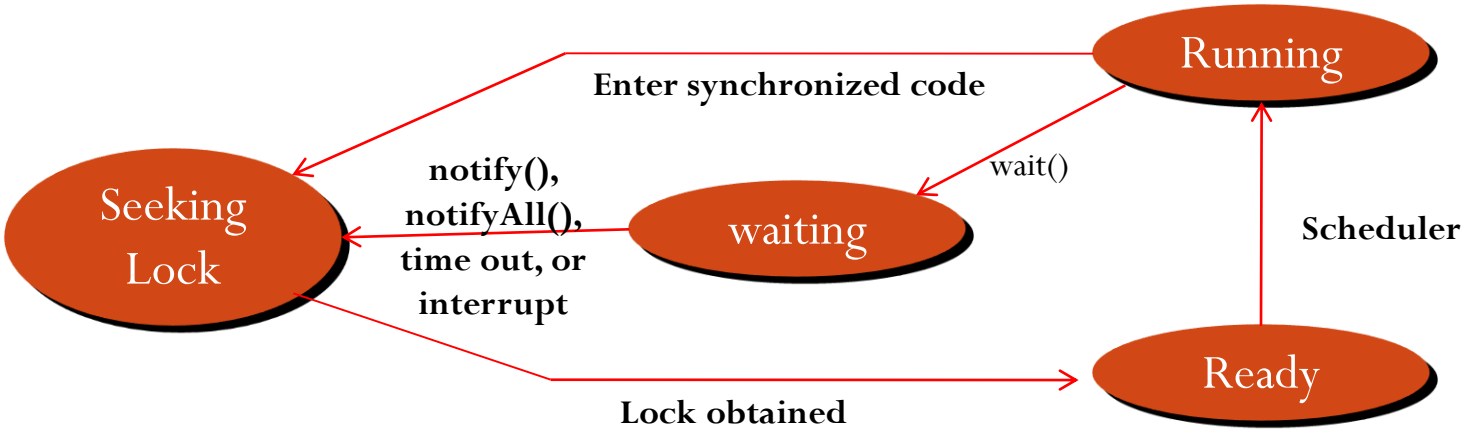
Output

Debugger Console | ThreadDemo (r

```
run-single:
4594499
Mon May 03 11:19:53 ICT 2010
3998080
6677430
Mon May 03 11:19:54 ICT 2010
5546326
9542804
Mon May 03 11:19:55 ICT 2010
```

4- Monitors, Waiting and Notifying

- Some threads can access common resources concurrently. We must synchronize accessing common resources ➔ Every object has a *lock*
- *Lock*: an extra variable is added by the compiler for monitoring the state of common resource. Before a thread accesses the common resource, the lock is tested.
- After a thread has the lock (it is permitted to access the common resource), it can access common resource. When it did, it needs notifying to others thread (wait-notify mechanism).



Monitors, Waiting and Notifying…

- *Two ways to mark code as synchronize*
  - Synchronize an entire method: *Let the* ***synchronized*** modifier in the method's declaration.

```
synchronized Type Method(args){
      <code>
}
```

A class contains synchronized code is called monitor.

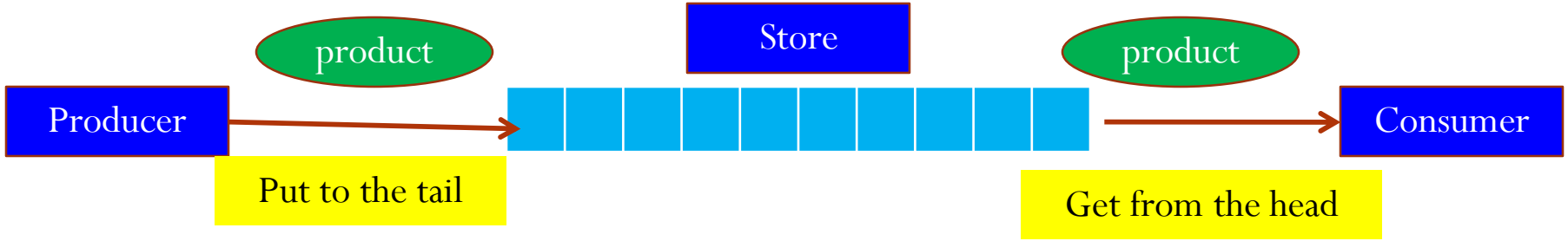
  - Synchronize some method of an object

```
Type Method ( args){
    ......
    synchronized ( object_var){
      object_var.method1(args);
      object_var.method2(args);
    }
}
```

Monitor: A technique to encapsulate common resources to a class. When a thread wants to access common resources, it must call a public synchronized method. As a result, common resources are accessed in successive manner.

Demo: The Producer-Consumer Problem

- Producer makes a product then puts it to a store.
- Consumer buys a product from a store.
- Selling Procedure: First In First Out



Attention!:
Store is common resource of 2 threads: producer and consumer.
➔ Store is a monitor and it's activities needs synchronization

Synchronizing:
* After a thread accessed common resource, it should sleep a moment or it must notify to the next thread (or all thread) in the thread-pool to awake and execute.
* Use the **synchronized** keyword to declare a method that will access common resource.

The Producer-Consumer Problem

```java
public class Store {
    int maxN=50;  // maximum number of products can be contains int the store
    long [] a;    // product list
    int n;        // current number of products
    public Store() {  n=0; a=new long[maxN];  }
    private boolean empty() { return n==0; }
    private boolean full() { return n==maxN; }
    public /* synchronized */ boolean put(long x){
       if (full()) return false;
       a[n++]=x;
       try  { Thread.sleep(500); }
       catch (Exception e){  }
       return true;
    }
    public /* synchronized */ long get(){
       long result=0;
       if (!empty()) {
           result=a[0]; // get the product at the front of line
           for (int i=0;i<n-1;i++) a[i]=a[i+1]; // shift products up.
           n--;
       }
       try  {  Thread.sleep(500);  }
       catch (Exception e){       }
       return result;
    }
}
```

A product is simulated as a number.

/* synchronized */
No synchronization

The Producer-Consumer Problem

```java
public class Producer extends Thread{
    Store store=null;
    long index=1;     // index of product that will be made
    public Producer(Store s) {
        store=s;
    }
    public void run(){
        while (true){
            try {
                boolean result= store.put(index);
                if (result==true) System.out.println("** Product " + (index++)+ " is made.");
                else System.out.println("Store is full!");
             }
            catch (Exception e) {
            }
        }
    }
}
```

The Producer-Consumer Problem

```java
public class Consumer extends Thread {
    Store store=null;
    public Consumer(Store s) {
        store=s;
    }
    public void run(){
        while (true){
            try {
                long x= store.get();
                if (x>0) System.out.println("-- Product " + x + " is bought.");
                else System.out.println("Consumer is waiting for new product.");
            }
            catch (Exception e) {
            }
        }
    }
}
```

```java
public class ProducerConsumerProblem {
    Store store;
    Producer pro;
    Consumer con;
    public ProducerConsumerProblem() {
        store= new Store();  pro= new Producer(store); con= new Consumer(store);
        pro.start(); con.start();
    }
    public static void main (String args[]){
        ProducerConsumerProblem obj=new ProducerConsumerProblem();
    }
}
```

The Producer-Consumer Problem

Synchronization is not used

Output - ThreadDemo (run-single) #2

```
compile:
run-single:
** Product 1 is made.
-- Product 1 is bought.
-- Product 2 is bought.
** Product 2 is made.
Consumer is waiting for new product.
** Product 3 is made.
** Product 4 is made.
-- Product 3 is bought.
** Product 5 is made.
-- Product 4 is bought.
```

Synchronization is used:

Output - ThreadDemo (run-single)

```
run-single:
** Product 1 is made.
** Product 2 is made.
-- Product 1 is bought.
-- Product 2 is bought.
Consumer is waiting for new product.
Consumer is waiting for new product.
Consumer is waiting for new product.
Consumer is waiting for new product.
** Product 3 is made.
** Product 4 is made.
-- Product 3 is bought.
-- Product 4 is bought.
Consumer is waiting for new product.
Consumer is waiting for new product.
```

Program of synchronized block

```
class Table{

 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}
```

Program of synchronized block

```
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
```

Program of synchronized block

```
public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
Output:5
       10
       15
       20
       25
       100
       200
       300
       400
       500
```

The account withdraw in the bank

ThreadSynchronizedDemo.java

Source History

```java
package Concurrency;

public class ThreadSynchronizedDemo implements Runnable{

   private int money=5000;

    public void run(){
       withdraw();
    }

    public void withdraw(){

            try {
                if(money>0){
                money=money-5000;
                System.out.println("The remaining Your money is :" + money);
                Thread.sleep(1000);
                }
                else
                    System.out.println("You have run out of money");
            } catch (Exception e) {
               System.out.println(e);
            }

    }
}
```

The account withdraw in the bank

ThreadSynchronizedDemo.java | ThreadSynchronized_use.java

Source | History

```java
package Concurrency;

public class ThreadSynchronized_use {
    public static void main(String[] args) {
        ThreadSynchronizedDemo t=new ThreadSynchronizedDemo();
        Thread t1=new Thread(t);
        Thread t2=new Thread(t);
         t1.start();
         t2.start();

    }
}
```

Notifications | Output - Java_Desktop (run)

```
run:
You have run out of money
The remaining Your money is :0
BUILD SUCCESSFUL (total time: 1 second)
```

The account withdraw in the bank

ThreadSynchronizedDemo.java | ThreadSynchronized_use.java

Source | History

```java
package Concurrency;

public class ThreadSynchronizedDemo implements Runnable{

    private int money=5000;

    public  void run(){
        withdraw();
    }

    public synchronized void withdraw(){

            try {
                if(money>0){
                money=money-5000;
                System.out.println("The remaining Your money is :" + money);
                Thread.sleep(1000);
                }
                else
                    System.out.println("You have run out of money");
            } catch (Exception e) {
               System.out.println(e);
            }

    }
}
```

The account withdraw in the bank

ThreadSynchronizedDemo.java | ThreadSynchronized_use.java

Source | History

```java
package Concurrency;

public class ThreadSynchronized_use {
    public static void main(String[] args) {
        ThreadSynchronizedDemo t=new ThreadSynchronizedDemo();
        Thread t1=new Thread(t);
        Thread t2=new Thread(t);
         t1.start();
         t2.start();

    }
}
```

Notifications | Output - Java_Desktop (run)

```
run:
The remaining Your money is :0
You have run out of money
BUILD SUCCESSFUL (total time: 1 second)
```

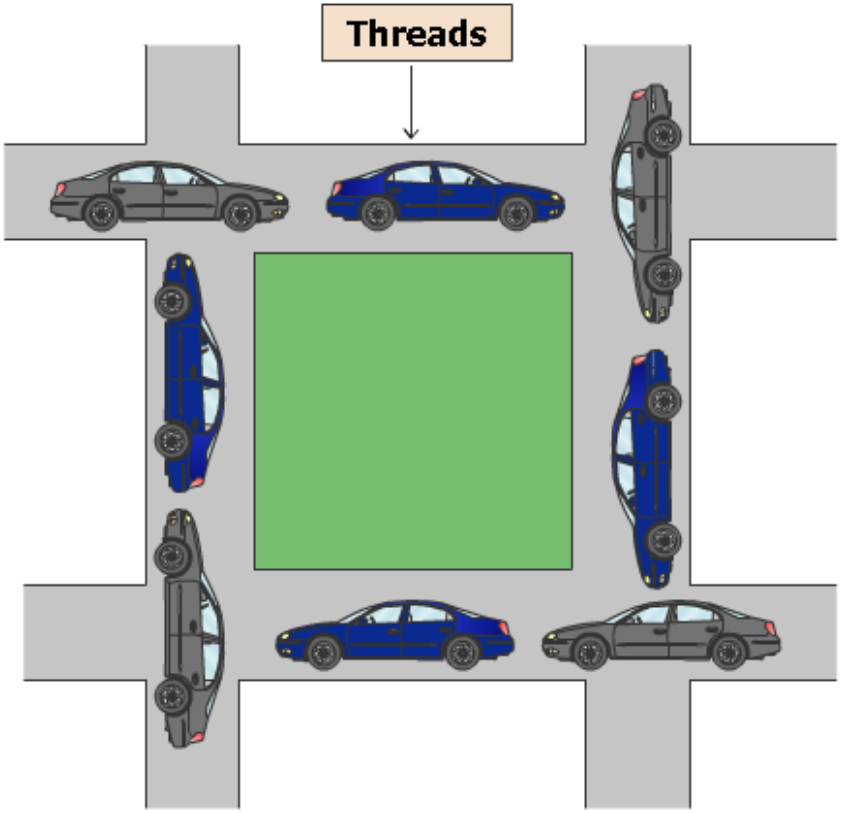
5 - Deadlock

<u>What is deadlock?</u>

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other → All threads in a group halt.

<u>When does deadlock occur?</u>

There exists a circular wait the lock that is held by other thread.



Nothing can ensure that DEADLOCK do not occur.

Deadlock Demo.

DeadlockDemo.java

Source History

```java
package Concurrency;

public class DeadlockDemo  implements Runnable{
    private int money=10000;

    public void run(){
        withdraw();
    }
    public synchronized void withdraw(){
            money=money-5000;
            showballance();

    }

    public  synchronized void showballance(){
        System.out.println("Your ballance is :" + money);
        withdraw();
    }

}
```

Deadlock Demo.

DeadlockDemo.java | Deadlock_use.java

Source | History

```java
package Concurrency;

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

        DeadlockDemo t=new DeadlockDemo();
        Thread t1=new Thread(t);
        Thread t2=new Thread(t);
        t1.start();
        t2.start();
    }
}
```

Notifications | Output - Java_Desktop (run)

```
        at Concurrency.DeadlockDemo.withdraw(DeadlockDemo.java:12)
Your ballance is :-60835000
Your ballance is :-60840000
        at Concurrency.DeadlockDemo.showballance(DeadlockDemo.java:18)
Your ballance is :-60845000
        at Concurrency.DeadlockDemo.withdraw(DeadlockDemo.java:12)
Your ballance is :-60850000
        at Concurrency.DeadlockDemo.showballance(DeadlockDemo.java:18)
        at Concurrency.DeadlockDemo.withdraw(DeadlockDemo.java:12)
        at Concurrency.DeadlockDemo.showballance(DeadlockDemo.java:18)
        at Concurrency.DeadlockDemo.withdraw(DeadlockDemo.java:12)
```

Send and recive Mail

Wait-Notify Mechanism, a way helps preventing deadlocks

Send and recive Mail

Message.java

Source History

```java
package Concurrency;

public class Message {

    private String content;

    public Message() {
        this.content = null;
    }

    public  String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }


}
```

Send and recive Mail

Message.java | Senduser.java

Source | History

```
1
2   package Concurrency;
3
4   public class Senduser  extends Thread{
5
6       Message a;
7
8       public Senduser(Message a) {
9           this.a = a;
10      }
        public  void run(){
            synchronized(a){
13              a.setContent("Tomorrow afternoon we met in fpt University HCMC");
14              a.notifyAll();
15          }
16      }
17  }
```

Thread table

Thread	Code Addr	Duration (mili sec)	CPU	State
Thread 1	10320	15	1	Suspended → Ready
Thread 2	40154	17	2	Suspended
Thread 3	80166	22	1	Suspended
…	…	…	…	….

Send and recive Mail

Message.java | Senduser.java | reciveuser.java

Source | History

```
1
2   package Concurrency;
3
4   public class reciveuser extends Thread{
5       Message b;
6
7       public reciveuser(Message b) {
8           this.b = b;
9       }
10
        public void run(){
            synchronized(b){
13              try {
14                  System.out.println("Waiting for your messages....");
15                  b.wait();
16                  System.out.println("I Have received your message");
17                  System.out.println(b.getContent());
                } catch (Exception ex) {
19                  System.out.println("An error recived message");
20              }
21
22          }
23      }
24
25  }
```

Thread table

Thread	Code Addr	Duration (mili sec)	CPU	State
Thread 1	10320	15	1	Suspended → Ready
Thread 2	40154	17	2	Suspended
Thread 3	80166	22	1	Suspended
…	…	…	…	….

Send and recive Mail

Message.java | Senduser.java | reciveuser.java | sendrecive_message.java

Source | History

```java
package Concurrency;

public class sendrecive_message {
    Message ms;
    Senduser su;
    reciveuser ru;

    public sendrecive_message() {
        this.ms = new Message();
        this.su = new Senduser(ms);
        this.ru = new reciveuser(ms);
        su.start();ru.start();
    }

    public static void main(String[] args) {
        sendrecive_message obj=new sendrecive_message();
        System.out.println("Good luck to you");
    }
```

Output - Java_Desktop (run)

```
run:
Good luck to you
Waiting for your messages....
I Have received your message
Tomorrow afternoon we met in fpt University HCMC
BUILD SUCCESSFUL (total time: 0 seconds)
```

Summary

Concepts were introduced:

- Definitions: Program, Process, Thread
- Multi-processing system
- Multi-threading programming in Java
- Thread Fundamentals in Java
- Synchronizing access to common resource.
- Monitoring thread states: Wait-notify mechanism.
- If you want some tasks executing concurrently, multi-threading is a solution.

Exercises

Workshop 1 (with report): The producer-consumer problem and

Workshop 2 (with report): Send and recive Mail

Thank You