Multithreading

Lecture 8

Department of Computer Science and Information Engineering

INHA University

Dr. Tamer ABUHMED

Outline

- Introduction
- Defining and Starting a Thread in Java
- Thread States
- Thread property access methods
- State transition methods for Thread
- Thread synchronization
- Thread Groups
- Multi-threading in Java Swing

What is Multithreading?

- Multithreading is similar to multi-processing.
- A multi-processing Operating System can run several processes at the same time
  - Each process has its own address/memory space
  - The OS's scheduler decides when each process is executed
    - Only one process is actually executing at any given time. However, the system appears to be running several programs simultaneously
- In a ***multithreaded*** application, there are several points of execution within the same memory space.
  - Each point of execution is called a thread
  - Threads share access to memory

What is Multithreading?

- In a single threaded application, one thread of execution must do everything
  - If an application has several tasks to perform, those tasks will be performed when the thread can get to them.
  - A single task which requires a lot of processing can make the entire application appear to be "sluggish" or unresponsive.
- In a multithreaded application, each task can be performed by a separate thread
  - If one thread is executing a long process, it does not make the entire application wait for it to finish.
- If a multithreaded application is being executed on a system that has multiple processors, the OS may execute separate threads simultaneously on separate processors.

Defining and Starting a Thread

- An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:
  1) Subclassing the Thread class and instantiating a new object of that class
  2) Implementing the Runnable interface
- In both cases the run() method should be implemented

Defining and Starting a Thread

give object to thread

Provide a Runnable object

```
public class HelloRunnable
implements Runnable {


    public void run() {

System.out.println("Hello from a
thread!");
    }


public static void main(String
args[]) {
Thread x = new Thread(new
HelloRunnable());
        x.start();
    }
}
```

invoke Thread.start in order to start the new thread

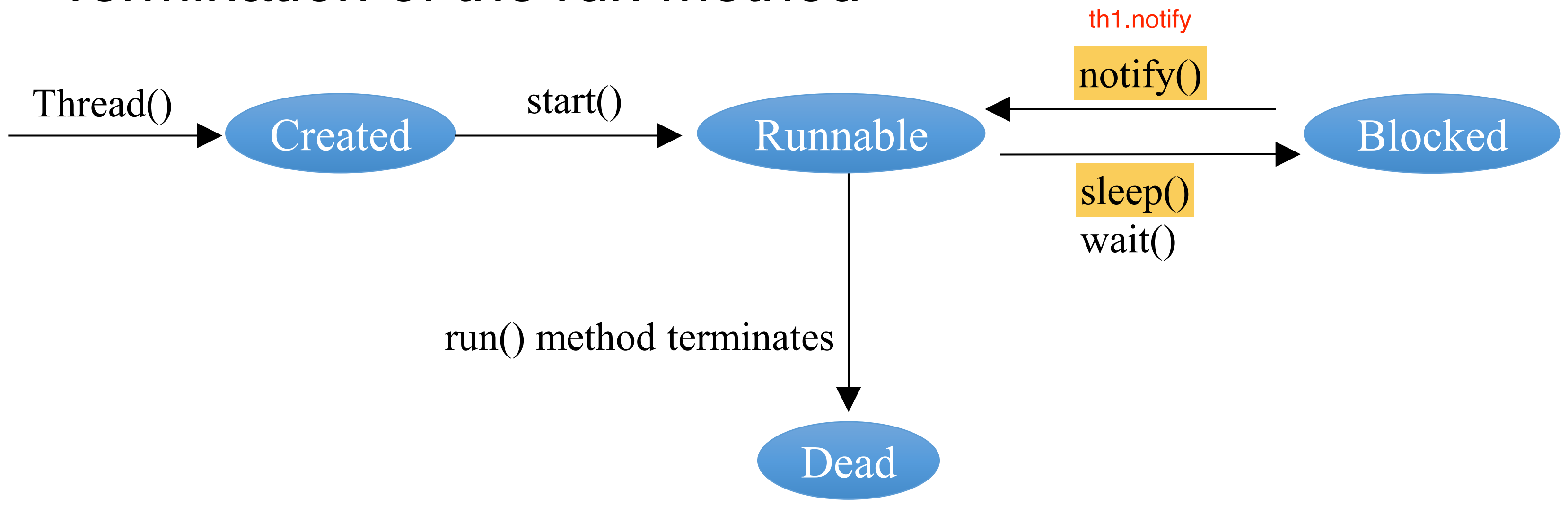
Subclass Thread

create thread object

```
public class threading
extends Thread {


    public void run() {

System.out.println("Hello
from a thread!");
    }


public static void
main(String args[]) {
threading x = new threading();
x.start();
    }
}
```

Thread States

- Threads can be in one of four states
  - Created, Running, Blocked, and Dead
- A thread's state changes based on:
  - Control methods such as **start**, **sleep**, **yield**, **wait**, notify
- Termination of the run method



Thread Example 1/2

OS system -> CPU 관리
thread using 관리 by CPU
same job but different finish

Java, No ==
.equal()

```
public class Racer implements Runnable
{

public static String winner;

public void race(){

for(int distance=1;distance<=100;distance++){

System.out.println("Distance Covered by
"+Thread.currentThread().getName()+
"is:"+distance +"meters");

//Check if race is complete if some one
//has already won

boolean isRaceWon =
this.isRaceWon(distance);

if(isRaceWon){

break;

}

  }

}
```

```
private boolean isRaceWon(int totalDistanceCovered){

boolean isRaceWon =  false;

if((Racer.winner==null )&&
                         (totalDistanceCovered==100)){

String winnerName =
Thread.currentThread().getName();

Racer.winner = winnerName; //setting the winner name

System.out.println("Winner is :"+Racer.winner);

isRaceWon = true;

}else if(Racer.winner==null){

isRaceWon = false;

}else if(Racer.winner!=null){

isRaceWon = true;

}

return isRaceWon;

}

@Override

public void run() {

this.race();

}

 }
```

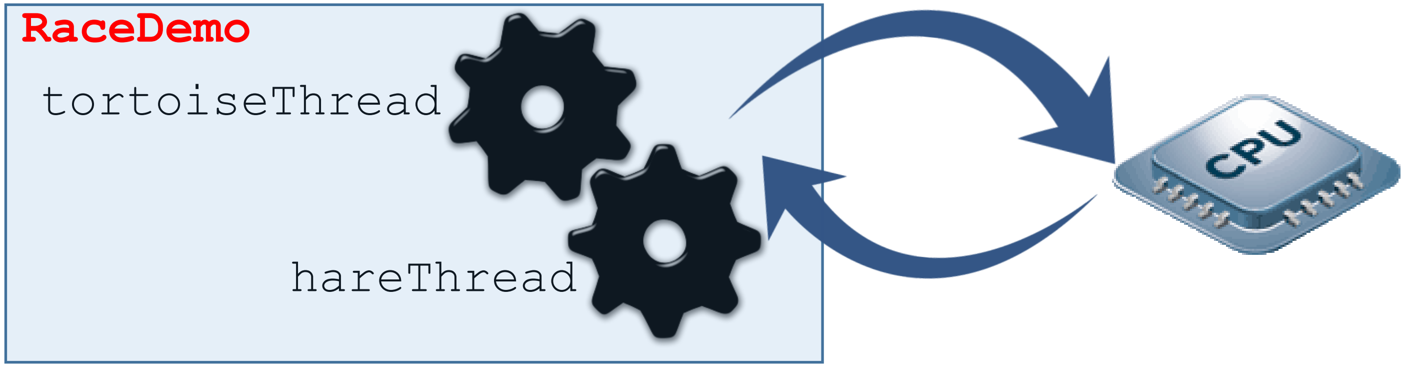
Thread Example 2/2

Test Thread Example

```
public class RaceDemo {

public static void main(String[] args) {

Racer racer = new Racer();

Thread tortoiseThread = new Thread(racer,
"Tortoise");

Thread hareThread = new Thread(racer, "Hare");

//Race to start. tell threads to start

tortoiseThread.start();

hareThread.start();

}

}
```

Output

```
Distance Covered by Tortoiseis:1meters

Distance Covered by Hareis:1meters

Distance Covered by Hareis:2meters

Distance Covered by Hareis:3meters

Distance Covered by Hareis:4meters

Distance Covered by Hareis:5meters

             .

             .

             .

Winner is :Hare

Distance Covered by Tortoiseis:2meters
```



Thread property access methods

- int getID() // every thread has a unique ID, since jdk1.5
- String getName(); setName(String)

// get/set the name of the thread

- ThreadGroup getThreadGroup();
- int getPriority() ; setPriority(int) // thread has priority in [0, 31]
- Thread.State getState() // return current state of this thread

- boolean isAlive()

Tests if this thread has been started and has not yet died. .

- boolean isDaemon()

Tests if this thread is a daemon thread.

- boolean isInterrupted()

Tests whether this thread has been interrupted.

State transition methods for Thread

- public synchronized native void start() {
  - start a thread by calling its run() method …
  - It is illegal to start a thread more than once }
- public final void join([long ms [, int ns]]);
  - Let current thread wait for receiver thread to die for at most ms+ns time
- static void yield() // callable by current thread only
  - Causes the currently executing thread object to temporarily pause and allow other threads to execute.
- void interrupt()
  - send an Interrupt request to a thread.
  - the "interrupted" status of the thread is set to true.
  - if the thread is blocked by sleep(), wait() or join(), the The *interrupted status* of the thread is cleared and an InterruptedException is thrown.
- public final void resume(); // deprecated
- public final void suspend();// deprecated→may lead to deadlock
- public final void stop(); // deprecated → lead to inconsistency

Thread synchronization

Memory Consistency Errors

- Problem with any multithreaded Java program :
  - **Two or more** Thread objects access the same pieces of data.
  - too little or no synchronization → there is inconsistency, loss or corruption of data.
  - too much synchronization → deadlock or system frozen.
- In between there is unfair processing where several threads can starve another one hogging all resources between themselves.

Memory Consistency Errors

1 thread can acces c
if th1 excute, lock c
than release c
if th2 excute lock c

Memory Consistency Errors

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;    }
}
```

Suppose Thread A invokes **increment** at about the same time Thread B invokes **decrement**. If the initial value of c is 0.

Memory Consistency Errors

occur when different threads have inconsistent views of what should be the same data.

Synchronized Solution

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

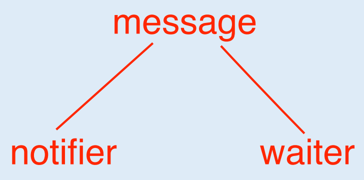
It is not possible for two invocations of synchronized methods on the same object to interleave

All threads invoke synchronized methods for the same object used by other thread are block (suspend execution) until the first thread is done

Wait() and notify() methods of Thread

2 different job access 1 data -> need synchronized

```
public class Message {

private String text;

public Message(String text) {

this.text = text;

}

public String getText() {

return text;

}

public void setText(String text)
{

this.text = text;

} }
```

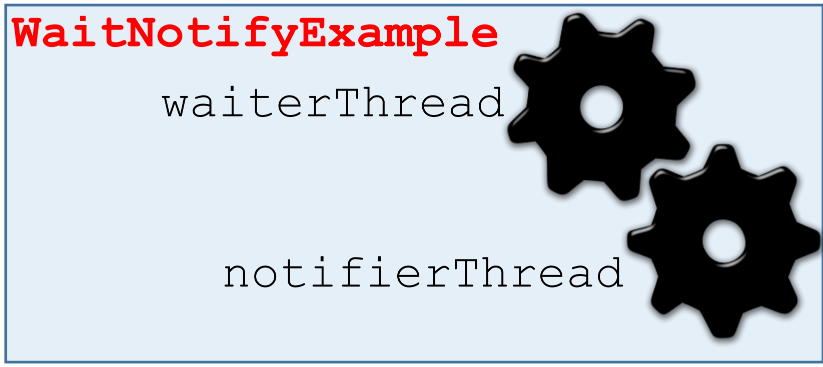


```
public class Waiter implements
Runnable {

Message message;

public Waiter(Message message) {

this.message = message;

}

@Override

public void run() {

synchronized (message) {

try {

System.out.println("Waiter is
waiting for the notifier at " +
new Date());

message.wait();

} catch (InterruptedException e)
{

e.printStackTrace();

}  }

System.out.println("Waiter is
done waiting at " + new Date());

System.out.println("Waiter got
the message: " +
message.getText());

} }
```

```
public class Notifier implements
Runnable {

Message message;

public Notifier(Message message) {

this.message = message;

}

@Override

public void run() {

System.out.println("Notifier is
sleeping for 3 seconds at " + new
Date());

try {

Thread.sleep(3000);

} catch (InterruptedException e1) {

e1.printStackTrace();

}

synchronized (message) {

message.setText("Notifier took a nap
for 3 seconds");

System.out.println("Notifier is
notifying waiting thread to wake up
at " + new Date());

message.notify();

} } }
```

Test Example

```
public class WaitNotifyExample {


public static void main(String[] args) {


Message message = new Message("Howdy");


Waiter waiter = new Waiter(message);

Thread waiterThread = new
Thread(waiter,"waiterThread");

waiterThread.start();


Notifier notifier = new Notifier(message);

Thread notifierThread = new
Thread(notifier, "notifierThread");

notifierThread.start();

} }
```



Output

```
Notifier is sleeping for 3 seconds at
Mon Nov 10 21:42:42 KST 2014

Waiter is waiting for the notifier at Mon
Nov 10 21:42:42 KST 2014

Notifier is notifying waiting thread to
wake up at Mon Nov 10 21:42:45 KST
2014

Waiter is done waiting at Mon Nov 10
21:42:45 KST 2014

Waiter got the message: Notifier took a
nap for 3 seconds
```

Thread Groups

- Every Java thread is a member of a thread group.
- Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.
- When creating a thread,
  - let the runtime system put the new thread in some reasonable default group (the current thread group) or
  - explicitly set the new thread's group.
- you cannot move a thread to a new group after the thread has been created.
  - when launched, main program thread belongs to main thread group.

Example

```
class ThreadGroupDemo
{
   public static void main (String [] args)
   {
  ThreadGroup tg = new ThreadGroup ("subgroup
1");
 Thread t1 = new Thread (tg, "thread 1");
 Thread t2 = new Thread (tg, "thread 2");
 Thread t3 = new Thread (tg, "thread 3");
 tg = new ThreadGroup ("subgroup 2");
 Thread t4 = new Thread (tg, "my thread");
 tg = Thread.currentThread().getThreadGroup
();
int agc = tg.activeGroupCount ();
System.out.println ("Active thread groups in
" + tg.getName () + " thread group: " + agc);
tg.list ();
   }
}
```

Output

```
Active thread groups in main thread group*2
java.lang.ThreadGroup[name=main,maxpri=10]
    Thread[main,5,main]
java.lang.ThreadGroup[name=subgroup 1,maxpri=10]
java.lang.ThreadGroup[name=subgroup 2,maxpri=10]
```

Multi-threading in Java Swing

Threading in Swing

- Threading matters a lot in Swing GUIs
  - You know: main's thread ends "early"
  - JFrame.setvisible(true) starts the "GUI thread"
- Swing methods run in a separate thread called the **Event-Dispatching Thread** (EDT)
  - Why? GUIs need to be responsive quickly
  - Important for good user interaction
- But: slow tasks can block the EDT
  - Makes GUI seem to hang
  - Doesn't allow parallel things to happen

Thread Rules in Swing

- All operations that update GUI components <u>must</u> happen in the EDT
  - These components are not thread-safe (later)
  - SwingUtilities.invokeLater(Runnable r) is a method that runs a task in the EDT when appropriate
- But execute slow tasks in separate *worker threads*
- To make common tasks easier, use a SwingWorker task

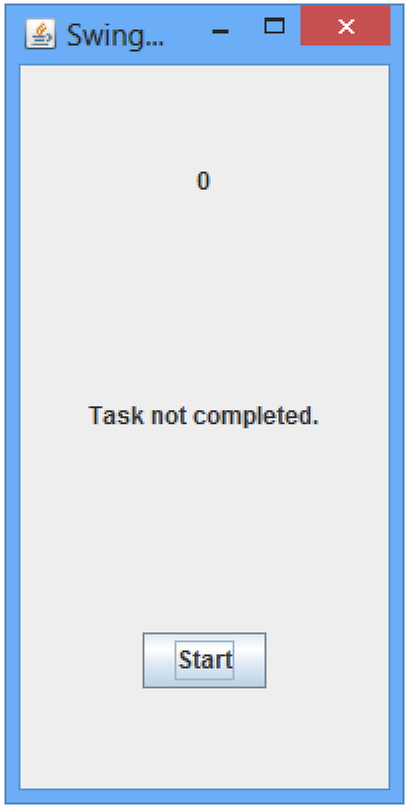
SwingWorker

- A class designed to be extended to define a task for a worker thread
  - Override method **doInBackground()**
    This is like run() – it's what you want to do
  - Override method **done()**
    This method is for updating the GUI afterwards
    - It will be run in the EDT
- For more info, see:
  http://download.oracle.com/javase/tutorial/uiswing/concurrency/
- Note you can get interim results too

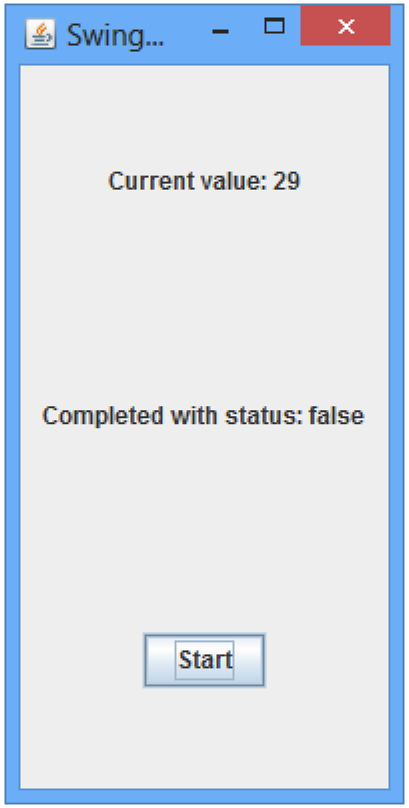
SwingWorker Example

```
public class MainFrame1 extends JFrame
{
private JLabel countLabel1 = new
JLabel("0");
private JLabel statusLabel = new
JLabel("Task not completed.");
private JButton startButton = new
JButton("Start");
public MainFrame1(String title) {
super(title);
setLayout(new GridBagLayout());
GridBagConstraints gc = new
GridBagConstraints();
gc.fill = GridBagConstraints.NONE;
gc.gridx = 0;
gc.gridy = 0;
gc.weightx = 1;
gc.weighty = 1;
add(countLabel1, gc);
gc.gridx = 0;
gc.gridy = 1;
gc.weightx = 1;
gc.weighty = 1;
add(statusLabel, gc);
gc.gridx = 0;
gc.gridy = 2;
gc.weightx = 1;
gc.weighty = 1;
add(startButton, gc);
startButton.addActionListener(new
ActionListener() {
public void actionPerformed(ActionEvent
arg0) {
start();
}
});
setSize(200, 400);
setDefaultCloseOperation(EXIT_ON_CLOSE)
;
setVisible(true);
}
private void start() {
SwingWorker<Boolean, Integer> worker =
new SwingWorker<Boolean, Integer>() {

@Override
//Note: do not update the GUI here
protected Boolean doInBackground()
throws Exception {
// Simulate useful work
for (int i = 0; i < 30; i++) {
Thread.sleep(100);
System.out.println("Hello: " + i);

// use publish to send values to
//process(), which used to update //the
GUI.
publish(i);
}
return false;
}
 @Override
// This will be called if you call
publish() from doInBackground()
// Can safely update the GUI here.
protected void process(List<Integer>
chunks) {
Integer value =
chunks.get(chunks.size() - 1);
countLabel1.setText("Current value:
" + value);
}
@Override
// This is called when the thread
finishes.
// Can safely update GUI here.
protected void done() {
try {
Boolean status = get();
statusLabel.setText("Completed with
status: " + status);
} catch (InterruptedException e) {
// TODO Auto-generated catch block
e.printStackTrace();
} catch (ExecutionException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}  }  };
worker.execute();
}
public static void main(String[]
args) {
SwingUtilities.invokeLater(new
Runnable() {
@Override
public void run() {
new MainFrame1("SwingWorker
Demo1");
}  });    }     }
```

Output



```
0

Task not completed.
```



```
Current value: 29

Completed with status: false
```

Summary

- Introduction
- Defining and Starting a Thread in Java
- Thread States
- Thread property access methods
- State transition methods for Thread
- Thread synchronization
- Thread Groups
- Multi-threading in Java Swing