Advanced Computer Programming

[Lecture 10]

Saeed Reza Kheradpisheh

kheradpisheh@ut.ac.ir

Department of Computer Science
Shahid Beheshti University
Spring 1397-98

MULTI-THREADING

- Itis often useful for a program to carry out two or more tasks at the same time. For example, a web browser can load multiple images on a web page at the same time.
- In this chapter, you will see how to implement this behavior by running tasks in multiple threads

Thread

Definition

A thread is a program unit that is executed independently of other parts of the program.

- Up to now, our programs had only a single thread.
- The Java virtual machine executes each thread for a short amount of time and then switches to another thread.
- If a computer has multiple CPUs, then some of the threads can run in parallel, one on each processor.

Running A Thread

Write a class that implements the `Runnable` interface.
That interface has a single method called `run`:

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

Place the code for your task into the `run` method of your class:

```
public class MyRunnable implements Runnable
{
   public void run()
   {
      Task statements
      . . .
   }
}
```

Running A Thread

Create an object of your subclass:

```
Runnable r = new MyRunnable();
```

Construct a `Thread` object from the runnable object:

```
Thread t = new Thread(r);
```

Call the `start` method to start the thread:

```
t.start();
```

Greeting Example

The greeting program should:

- Print a time stamp.
- Print the greeting.
- Wait a second.

Greeting Example

The greeting program should:

- Print a time stamp.
- Print the greeting.
- Wait a second.

Get the time stamp by constructing an object of the java.util.Date class.

```
Date now = new Date();
System.out.println(now + " " + greeting);
```

Greeting Example

The greeting program should:

- Print a time stamp.
- Print the greeting.
- Wait a second.

Get the time stamp by constructing an object of the java.util.Date class.

```
Date now = new Date();
System.out.println(now + " " + greeting);
```

To wait a second, we use the static sleep method of the Thread class (it is imported in java by defaults).

```
Thread.sleep(milliseconds)
```

Greeting Example

The greeting program should:

- Print a time stamp.
- Print the greeting.
- Wait a second.

Get the time stamp by constructing an object of the java.util.Date class.

```
Date now = new Date();
System.out.println(now + " " + greeting);
```

To wait a second, we use the static sleep method of the Thread class (it is imported in java by defaults).

```
Thread.sleep(milliseconds)
```

When a sleeping thread is interrupted, an InterruptedException is generated (it is imported in java by defaults).

Thread scheduling!

- The running time and running duration of threads is scheduled by the Operating System.
- The thread scheduler gives no guarantee about the order in which threads are executed.
- Each thread runs for a short amount of time, called a time slice. Then the scheduler activates another thread.
- Thus, you should expect that the order in which each thread gains control is somewhat random.

Terminating Threads

- When the run method of a thread has finished executing, the thread terminates.

Terminating Threads

- When the run method of a thread has finished executing, the thread terminates.
- However, sometimes you need to terminate a running thread. For example, several threads are looking for an item and a thread finds it, and other threads should be terminated.

Terminating Threads

- When the run method of a thread has finished executing, the thread terminates.
- However, sometimes you need to terminate a running thread. For example, several threads are looking for an item and a thread finds it, and other threads should be terminated.
- In the initial release of the Java library, the Thread class had a stop method to terminate a thread. It could lead to dangerous situations.

Terminating Threads

- When the run method of a thread has finished executing, the thread terminates.
- However, sometimes you need to terminate a running thread. For example, several threads are looking for an item and a thread finds it, and other threads should be terminated.
- In the initial release of the Java library, the Thread class had a stop method to terminate a thread. It could lead to dangerous situations.
- Instead of simply stopping a thread, you should notify the thread that it should be terminated using:

```
t.interrupt();
```

Terminating Threads

- The interrupt() method does not actually cause the thread to terminat. It merely sets True a boolean variable of the thread object.

Terminating Threads

- The interrupt() method does not actually cause the thread to terminat. It merely sets True a boolean variable of the thread object.
- The run method can check whether that flag has been set, by calling the static thread.interrupted() method.

Terminating Threads

- The interrupt() method does not actually cause the thread to terminat. It merely sets True a boolean variable of the thread object.
- The run method can check whether that flag has been set, by calling the static thread.interrupted() method.

For example in GreetingRunnable:

```
public void run()
{
   for (int i = 1; i <= REPETITIONS && !Thread.interrupted(); i++)
   {
      Do work.
   }
   Clean up.
}
```

Terminating Threads

- However, if a thread is sleeping, it can't execute code that checks for interruptions.

Terminating Threads

- However, if a thread is sleeping, it can't execute code that checks for interruptions.
- Therefore, the sleep method is terminated with an InterruptedException whenever a sleeping thread is interrupted.

Terminating Threads

- However, if a thread is sleeping, it can't execute code that checks for interruptions.
- Therefore, the sleep method is terminated with an InterruptedException whenever a sleeping thread is interrupted.

For example in GreetingRunnable:

```
public void run()
{
   for (int i = 1; i <= REPETITIONS && !Thread.interrupted(); i++)
   {
      Do work.
   }
   Clean up.
}
```

Terminating Threads

- The sleep method also throws an InterruptedException when it is called in a thread that is already interrupted.

We can use it to detect when a thread is intrupted

```
public void run()
{
   try
   {
      for (int i = 1; i <= REPETITIONS; i++)
      {
         Do work.
         Sleep.
      }
   }
   catch (InterruptedException exception)
   {
   }
   Clean up.
}
```

Race Conditions

When threads share access to a common object, they can conflict with each other.
Example:

- Each thread of the DepositRunnable class repeatedly deposits $100.
- Each thread of the WithdrawRunnable class repeatedly withdraws $100.

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
Withdrawing 100.0, new balance is 100.0
. . .
Withdrawing 100.0, new balance is 0.0
```

Race Conditions

When threads share access to a common object, they can conflict with each other.
Example:

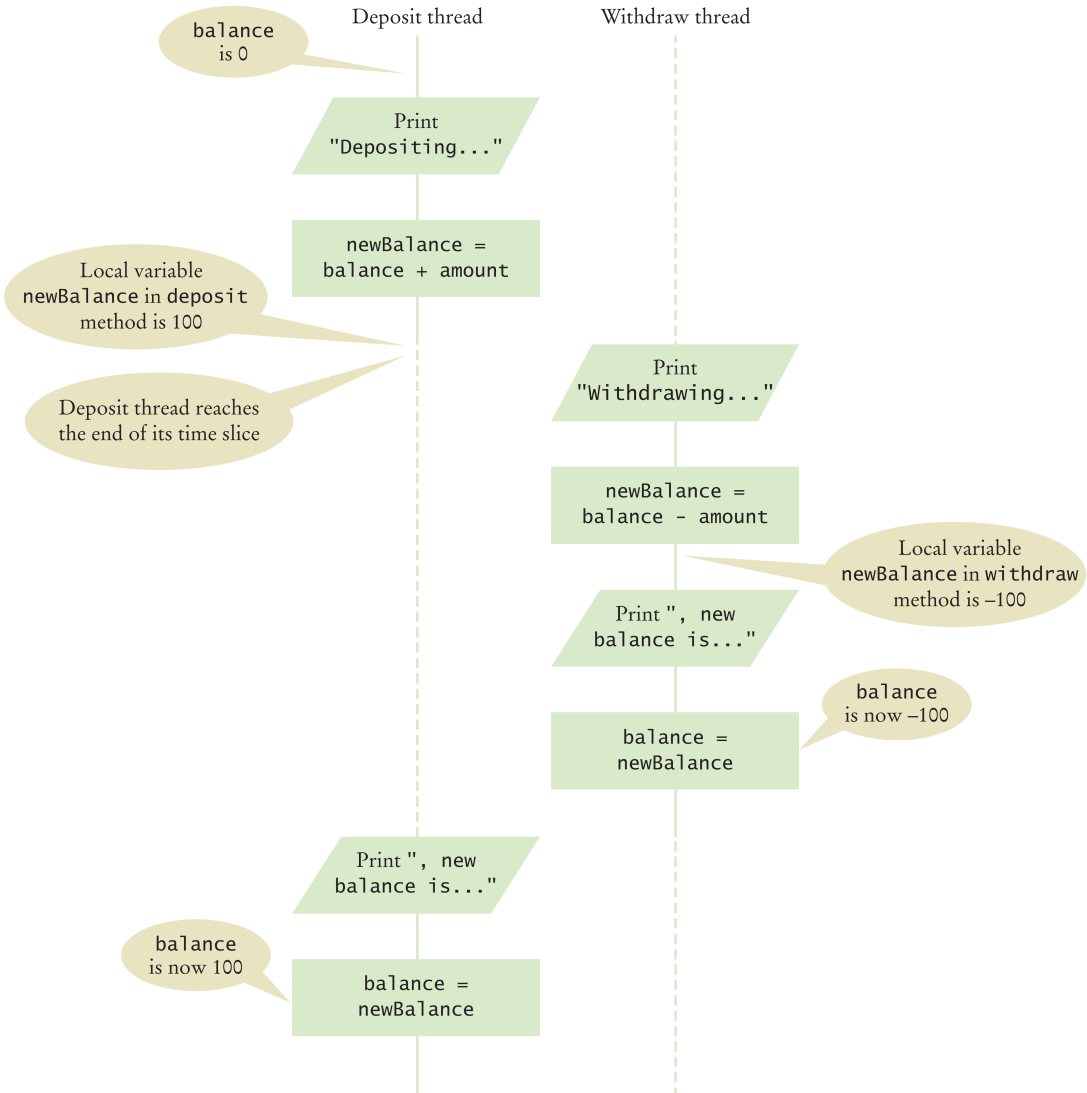
- Each thread of the DepositRunnable class repeatedly deposits $100.
- Each thread of the WithdrawRunnable class repeatedly withdraws $100.

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
Withdrawing 100.0, new balance is 100.0
. . .
Withdrawing 100.0, new balance is 0.0
```

- In the end, the balance should be zero.

Race Conditions

- However, you may sometimes notice messed-up output:



Synchronizing Object Access

```
public class BankAccount
{
   private Lock balanceChangeLock;
   . . .
   public BankAccount()
   {
      balanceChangeLock = new ReentrantLock();
      . . .
   }
}
```

Synchronizing Object Access: to avoid race condition

Declare a lock variable:

```
public class BankAccount
{
   private Lock balanceChangeLock;
   . . .
   public BankAccount()
   {
      balanceChangeLock = new ReentrantLock();
      . . .
   }
}
```

Lock it befor accessing the resource and ulock it afterwards:

```
balanceChangeLock.lock();
try
{
   Manipulate the shared resource.
}
finally
{
   balanceChangeLock.unlock();
}
```