

# G51PRG: Introduction to Programming Second semester Lecture 11

Natasha Alechina  
School of Computer Science & IT  
[nza@cs.nott.ac.uk](mailto:nza@cs.nott.ac.uk)

## This lecture: multiple concurrent processes (threads)

Advanced topic, but a very good feature of Java!

- What is a thread
- Why use multiple threads
- Issues and problems involved
- Java threads

Lecture 11: threads

2

## Example applications

- Many applications have to perform several tasks simultaneously.
- For example, on-line banking allows many people to connect to the bank database simultaneously, so the server program is doing many different things concurrently.
- Web browser allows you to scroll up and down the page while it is downloading images etc.: this is two things at once.
- Note that if the machine these programs are running on has only one processor, they don't really happen in parallel, but are interleaved. If there are several processors then they can be really executed in parallel.

Lecture 11: threads

3

## What is a thread

Choose whatever definition you prefer:

- A thread is a sequence of steps executed one at a time.
- A thread is a single sequential flow of control within a program.
- A thread is a separately runnable subprocess.
- Other names: lightweight process, execution context.

Lecture 11: threads

4

## Example

A single cash dispenser thread could do something like

- Read the bank card and pin code; if OK,
- Prompt the user with options
- If the option is to withdraw money, prompt for amount
- Dispense the required amount
- When the money is taken, update the user's balance:  
`current amount = current amount - amount withdrawn`

Lecture 11: threads

5

## Why use multiple threads

- In the example above - obvious: cash dispensers should be able to operate in parallel
- In some cases (e.g. user interfaces) multiple threads are not strictly necessary, but make the program much more transparent and the resulting application much more usable. Each independent task (such as scrolling or downloading) can be programmed separately and can be executed concurrently with other tasks.

Lecture 11: threads

6

## Issues and problems involved

- *race hazard*
- *starvation*
- *deadlock*

Lecture 11: threads

7

## Race hazard

- A *race hazard* occurs when several threads are racing each other trying to access the same shared resource, and may modify it in an interleaved way.
- For example, two people using two different cash machines withdrawing money from the same account.

Lecture 11: threads

8

## Race hazard example

Thread 1	Thread 2
withdraw 20 pounds	
	withdraw 30 pounds
balance = 300 pounds	balance = 300 pounds
new balance = 280 pounds	new balance = 270 pounds
Result: 50 pounds withdrawn, only 30 subtracted.	

Lecture 11: threads

9

## Cure: synchronisation

- In the example, threads 1 and 2 should be *synchronised* to avoid interleaving.
- Thread 1 could have locked the account when it got the request to withdraw money.
- All other threads which need to access the same account would have to wait until Thread 1 is finished and the lock is released.

Lecture 11: threads

10

## Synchronisation example

Thread 1	Thread 2
withdraw 20 pounds	
LOCK THE ACCOUNT	
	withdraw 30 pounds
	ACCOUNT LOCKED
balance = 300 pounds	
new balance = 280 pounds	
RELEASE LOCK	
	LOCK THE ACCOUNT
	balance = 280 pounds
	new balance = 250 pounds

Lecture 11: threads

11

## Other problems

- *Starvation* occurs when one or more threads are blocked from gaining access to a resource and thus cannot make progress.
- *Deadlock* occurs when two or more threads are waiting on a condition that cannot be satisfied. For example, Thread 1 is waiting for Thread 2 to give it resource A and Thread 2 is waiting for Thread 1 to release resource B.

A system is *fair* when each thread gets enough access to limited resource to make reasonable progress.

Lecture 11: threads

12

## Lessons from race hazard

The threads should be able to

- lock an object (e.g. bank account)
- wait for a resource
- notify other threads when the resource is available

Lecture 11: threads

13

## Thread class in Java

Constructors:

- `Thread()`
- `Thread(Runnable r)` (takes anything with a `run()` method and turns it into a thread)

Lecture 11: threads

14

## Thread class in Java continued

Methods:

- `start()` start the thread
- `run()` execute the sequence of steps the thread performs
- `sleep(long time)` throws `InterruptedException` pauses for `time` milliseconds
- `void setPriority(int newPriority)`: threads may have different priority and scheduled accordingly.
- `wait()`, `wait(long timeout)`, `notify()`, `notifyAll()` inherited from `Object`
- `yield()`: give another thread a chance to run

Lecture 11: threads

15

## Creating a Thread in Java

Two ways to create a thread in Java:

- subclass the **Thread** class and override its `run()` method (the default implementation does nothing).
- implement **Runnable** interface (has the `run()` method).

Lecture 11: threads

16

## How to stop a thread

- `stop()` method is deprecated (because unsafe), do not use it!
- Best of all is to arrange for the thread to return when it is no longer needed.

Lecture 11: threads

17

## Examples (from Arnold and Gosling)

- One thread prints "ping" every 33 milliseconds, another prints "PONG" every 100 milliseconds.
- The first class, `PingPongA`, extends `Thread`.
- The second class, `PingPongB`, implements `Runnable`.

Lecture 11: threads

18

## PingPongA

```
public class PingPongA extends Thread {
    private String word; // what to print
    private int delay; // how long to pause

    public PingPongA(String whatToSay, int
        delayTime) {
        word = whatToSay;
        delay = delayTime;
    }
}
```

Lecture 11: threads

19

## PingPongA continued

```
public void run() {
    try {
        for(;;) {
            System.out.print(word + " ");
            sleep(delay); // wait until next time
        }
    } catch (InterruptedException e) {
        return; // end this thread
    }
}
```

Lecture 11: threads

20

## PingPongA use

```
public static void main(String[] args) {
    new PingPongA("ping", 33).start();
    new PingPongA("PONG", 100).start();
}
```

Lecture 11: threads

21

## Note: start() rather than run()!

- If you want to create several threads in your program which execute concurrently, do not call their run() method: then all threads will run in order (first thread will run till it returns, then the second thread, etc....).
- Call start() method: then all threads will start and their run() methods will get a chance to run concurrently.

Lecture 11: threads

22

## PingPongA trace

```
ping PONG ping ping PONG ping ping ping
PONG
```

Lecture 11: threads

23

## PingPongB

```
public class PingPongB implements
    Runnable {
    private String word; // what to print
    private int delay; // how long to pause

    public PingPongB(String whatToSay, int
        delayTime) {
        word = whatToSay;
        delay = delayTime;
    }
}
```

Lecture 11: threads

24

## PingPongB continued

```
public void run() {
    try {
        for(;;) {
            System.out.print(word + " ");
            Thread.sleep(delay);
        }
    } catch (InterruptedException e) {
        return;        // end this thread
    }
}
```

Lecture 11: threads

25

## PingPongB use

```
public static void main(String[] args) {
    Runnable ping = new PingPongB("ping",
        33);
    Runnable pong = new PingPongB("PONG",
        100);
    new Thread(ping).start();
    new Thread(pong).start();
}
```

Lecture 11: threads

26

## Synchronized methods

- Blocks of code and methods which access the same object from separate threads are called critical sections. They are identified with *synchronized* keyword.

- For example,

```
public synchronized double getBalance(){
    return balance
}
```

- The thread which called a synchronized method gets a lock on the object whose method was called. Other threads cannot call a synchronized method on the same object until the object is unlocked.

Lecture 11: threads

27

## Synchronized statements

- To lock an object without invoking a synchronized method, synchronized statement can be used. It consists of two parts: an object to be locked and a statement to execute when the lock is obtained.

```
synchronized(balance){
    ...
}
```

Lecture 11: threads

28

## wait()

- Wait for some condition to become true:

```
synchronized void doWhenCondition(){
    while (!condition) {
        wait();
    }
    ...
}
```

Lecture 11: threads

29

## notify()

- Tell other threads if you changed something they may be interested in:

```
synchronized void changeCondition() {
    ... change some value used in a
    condition test
    notifyAll();
}
```

Lecture 11: threads

30

## Example: counter

- A counter class which has an integer variable (counter) and two synchronised methods, increment() and decrement().
- A counter can be incremented up to 10 and decremented down to 0.
- Only one thread at a time has access to the counter, so there is no race hazard.
- Two kinds of threads, Incrementers and Decrementers.
- Incrementers call increment(), Decrementers call decrement().

Lecture 11: threads

31

## Example: Counter class

```
class Counter {  
    int counter;  
  
    public Counter(int i) {  
        counter = i;  
    }  
}
```

Lecture 11: threads

32

## Example: Counter class

```
public synchronized void increment() {  
    try {  
        while (counter >= 10) wait();  
    } catch (InterruptedException e) {  
        return;  
    }  
    counter++;  
    notifyAll();  
}
```

Lecture 11: threads

33

## Example: Counter class

```
public synchronized void decrement() {  
    try {  
        while (counter <= 0) wait();  
    } catch (InterruptedException e) {  
        return;  
    }  
    counter--;  
    notifyAll();  
}
```

Lecture 11: threads

34

## Example: Incrementer class

```
class Incrementer extends Thread {  
    String name;  
    Counter c;  
    Incrementer(String name, Counter q){  
        this.c = q;  
        this.name = name;  
        System.out.println("Incrementer" +  
            name + " created");  
    }  
}
```

Lecture 11: threads

35

## Example: Incrementer class

```
public void run() {  
    while(true){  
        c.increment();  
        System.out.println(name + "  
incremented the counter to "+c.counter);  
        try {  
            Thread.sleep(10);  
        } catch (InterruptedException e){  
            return;  
        }  
    }  
}
```

Lecture 11: threads

36

## Decrementer is similar

```
public static void main(String[] args) {  
    Counter c = new Counter(0);  
    new Incrementer("x", c).start();  
    new Incrementer("y", c).start();  
    new Decrementer("z", c).start();  
    new Decrementer("w", c).start();  
}
```

Lecture 11: threads

37

## Summary

- Threads are subprocesses running within the same program.
- Threads can compete for resources; the programmer should ensure fairness
- Threads can be synchronised, so that one thread gets access to an object only when another is finished
- Threads can wait for one another and notify each other of changes.
- New keyword: synchronized
- More details: Java Gently, Arnold and Gosling , Sun Java Tutorial: <http://java.sun.com/docs/books/tutorial/essential/threads>

Lecture 11: threads

38