Java Programming 2
Threads

Mary Ellen Foster

MaryEllen.Foster@glasgow.ac.uk

"juice" by Lindsey Turner
https://flic.kr/p/4npCkA

Concurrent programming: basics

- Process
  - Self-contained execution environment – own memory space
  - Often synonymous with programs or applications – however, an application may consist of several processes (e.g., Google Chrome)
- Thread
  - Lightweight processes: shared memory space
  - Every process has at least one thread

https://www.howtogeek.com/124218/why-does-chrome-have-so-many-open-processes/

Why use concurrent techniques?

Tasks can be divided into subtasks; subtasks can be executed in parallel

Theoretical possible performance gain (Amdahl's Law):

If F is the percentage of the program which cannot run in parallel and n is the number of processes, then the maximum performance gain is $\frac{1}{F+\left(\frac{1-F}{n}\right)}$

Why NOT use concurrent techniques?

Threads can access shared data – two main potential problems

Visibility

Thread A reads shared data which is later changed by thread B without thread A being aware of the change

Access

Several threads access and change shared data at the same time

Can lead to

Liveness failure – program does not react any more due to problems in shared access

Safety failure – program creates incorrect data

Concurrent programming in Java

Java uses **multithreaded** programming within a single process

Basic building block: **Thread** class

Useful helper package: **java.util.concurrent** (since Java 1.5)

Note: all work we have done so far is taking place in the context of a single main **Thread** object – can be accessed and manipulated just like other Threads

Exception: Swing also has its own thread

Creating a Thread

Preferred technique: implement the **Runnable** interface and define **run()** method

(Can also subclass Thread but not as flexible/general)

Create thread based on **Runnable** class and use **start()** to start it

```
public class HelloRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("Hello from a thread");
    }

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

Based on https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html

Interrupting a Thread

From outside: call **interrupt()** on the **Thread** object (non-static)

Inside the **Thread**

If you call a method that could throw **InterruptedException** (e.g, **sleep**(), **join**()), just return after it is caught

Otherwise, periodically check **Thread.interrupted()** (static) and return if it is true

```
while(true) {
    doSomethingTimeConsuming();
    if (Thread.interrupted()) {
        break;
    }
}
```

Pausing a Thread

Use **Thread.sleep()** (static) to pause execution for a specified period

- Lets other threads use system resources
- Wait for something time-dependent to finish

Duration can be specified in milliseconds or nanoseconds – not guaranteed to be precise, depends on underlying OS and might be interrupted

Thread.sleep() throws **InterruptedException** (checked) – indicates that another thread interrupted the sleep and the thread should terminate

- *If you do not have another thread that can interrupt, you can usually just ignore this exception*

Thread.sleep() example (main thread)

```
String[] importantInfo = {
    "Mares eat oats",
    "Does eat oats",
    "Little lambs eat ivy",
    "A kid will eat ivy too" };

for (String info : importantInfo) {
    // Pause for (approximately) 4 seconds
    Thread.sleep(4000);
    // Print a message
    System.out.println(info);
}
```

Waiting for a thread to terminate

Use **join()** method (not static)

Can also throw **InterruptedException** – must be caught

You should do this at the end of any multithreaded program to be sure it terminates

```
Thread t = new Thread(new HelloRunnable());
t.start();
try {
    t.join();
} catch (InterruptedException ex) {
    // handle or ignore it
}
```