ENTERPRISE JAVA PROGRAMMING / PROGRAMMING II

MODULES CT545 / CT875, SEMESTER 2, ACADEMIC YEAR 2020-2021
NATIONAL UNIVERSITY OF IRELAND, GALWAY

Lecture 3a

Lecturer: Dr Patrick Mannion

TOPICS

- The basics of *multithreading* in Java (part 1)

MULTITHREADING

What are threads and multithreading?

- Firstly, a few preliminaries...
- A (computer) program represents a set of instructions.
- A *process* is the sequential sequence of instructions actually executed by a *Central Processing Unit* (CPU) when a program runs.
- A process is an *instance* of the program, in the same way as an object is an instance of a class
- Each process consists of a number of *threads* (sub-processes "inside" a process)

MULTITHREADING

What are threads and multithreading?

- If in a single process two or more threads exist, we speak of *multithreading*
- Threads are able to execute independently from each other, even at the same time -> *parallel computing*
- All threads within the same parent process typically work with resources assigned to the parent process, such as the memory of the process
- Threads depend on their parent process. If the parent process ends, so do all its threads.

  That's why threads are sometimes called *lightweight processes*.

MULTITHREADING

What are threads and multithreading?

- *Concurrency* means that two or more computing processes or threads can run fully or partially independent from each other, in particular, they can run simultaneously
- These processes or threads are either executed in parallel on different CPUs, or they run seemingly at the same time by letting a *scheduler* frequently switch them back and forth between these threads/processes (*time slicing*)

MULTITHREADING

What are threads and multithreading?

- Concurrency can happen on the system level (*multitasking; concurrency of processes*), or within a single process (using multithreading)

- Strictly speaking, concurrency is not the same as multithreading, but in the context of Java, these two words are normally exchangable
- Also, usually, "concurrency" is typically used in the same sense as *parallel computing*, however with a focus on patterns of interaction between the processes/threads.
- So, although strictly speaking the following words do not have the same meaning, in practice they are often used interchangeably: multithreading ≈ concurrency ≈ parallelism

MULTITHREADING

What are threads and multithreading?

- Multithreading, and concurrency in particular, is useful in order to, for example:
  - Cooperate with other programs (e.g., a server with multiple clients over a network)
  - Avoid that users have to wait for the program's response
  - **Make programs faster** (if there are multiple cores)
- Most modern programming languages (like Java) support multithreading
- Java allows that a program creates multiple threads which either execute concurrently (*concurrent threads*) or sequentially (one thread starts after another thread has ended)
- Whether concurrent Java threads run literally or just seemingly in parallel depends on the system architecture...

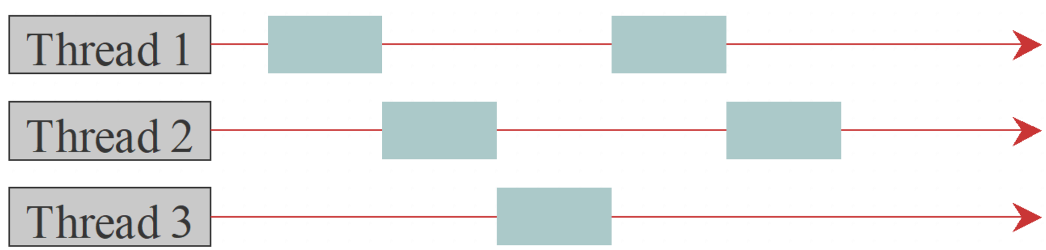
MULTITHREADING

What are threads and multithreading?

Java executes multiple concurrent threads as follows:

Truly parallel execution of multiple threads on multiple CPUs (or multiple cores of the same CPU).

or:



Multiple threads sharing a single CPU by means of *time slicing* using a scheduler. The scheduler assigns the threads their running times (normally equitably).

MULTITHREADING

What are threads and multithreading?

- From a programmer's point of view, a thread is a certain path of execution within the program
- The part of the program which a thread executes is called a *task* (of the thread)
- In other words, a task is a description of what a thread should do

- Thread and process are runtime concepts, whereas task/program are pieces of code
- The threads of a certain program can run concurrently, but also sequentially (a thread starts after another thread has ended). In principle, a program can create a new thread at any time.

MULTITHREADING

What are threads and multithreading?

- If two (or more) threads run concurrently, it is possible that they the computation of the first thread *depends* on results computed by the other thread (*thread interaction*), or that other kinds of thread interference occurs.
- The interaction and interference of threads is what makes multithreading challenging from a software developer's point of view.
- Wherever possible, we shall avoid dependencies between threads. Ideally, each thread runs independently from other parallel running threads

MULTITHREADING

How to create and run threads?

- The Java class `Thread` allows to create threads. More precisely, you create a thread *object* (object of class Thread) in memory first (sometimes also called "thread") and afterward you start it (creates the actual thread)
- Each thread performs a certain task, which also needs to be specified

MULTITHREADING

How to create and run threads?

- The task of the thread is defined using a different class (the *task class* or *runnable class*) which implements the interface `Runnable` (Java API):
- This class needs to implement the `run()` method in interface `Runnable`.
- `run()` is the task the thread should execute, i.e., what the thread should do.
- Don't call `run()` directly (*) to launch the thread! It is automatically called by Java when you start the thread as explained on the next slide.

(*) You can call `run()` directly, but then it will be called just like any ordinary method (i.e., it would be executed on the current thread - no new thread would be launched).

MULTITHREADING

How to create and run threads?

- To create a thread, first, create an object of the task class
- Then, we need to create an object of class `Thread`
- At this, we pass to the constructor of `Thread` the object of the task class
- E.g., `Thread threadObject = new Thread(objectOfTaskClass)`
  Note that this does <u>not</u> yet create (start) the thread itself…!

MULTITHREADING

How to create and run threads?

- We start the thread by invoking method `start()` of the thread object (<u>not</u> `run()`!). E.g., `threadObject.start();`
- Internally, this automatically also calls `run()` (as defined in the task class). The thread ends regularly when `run()` returns. You don't call `run()` manually.
- The thread is typically not finished directly after it has been started: the code below `start()`, the started thread, and possibly previously started threads run simultaneously.

MULTITHREADING

How to create and run threads?

- You can let multiple threads run concurrently by invoking `start()` multiple times shortly after each other
- More generally: if a thread is started before another thread has ended, these two threads run (partially) simultaneously. This is the classic way to do parallel/concurrent computing in Java.
- You can run 100s of (simple) threads simultaneously on a normal PC or laptop

MULTITHREADING

How to create and run threads?

- The same task (as specified in a certain task class by implementing method `run()`) can be used by multiple threads (i.e., you can pass the same instance of the task class multiple times to the constructor of class `Thread`)
- You can of course also create multiple concurrently running threads which perform *different* tasks (defined using different task classes)
- Remark: each Java program has at least one thread, because the `main`-method also runs on a (hidden) thread

MULTITHREADING

How to create and run threads?

java.lang.Runnable ← TaskClass

```java
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ... //initialization of the task class object (optional)
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

```java
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

MULTITHREADING

How to create and run threads?

- There is an alternative way of creating threads, by extending the class `Thread` (which also implements `Runnable`)

```
class MyThread extends Thread {
        …
        public void run() {
                // the task. What the thread shall do
                // ...
        }
}

MyThread t = new MyThread();
t.start();
```

- However, although this seems somewhat simpler (you need only one class instead of two), it is less flexible, because it is less "pure" (a thread and the task it runs are two different things)

MULTITHREADING

How to create and run threads?

- An example for a multithreaded (i.e. concurrent) program...

MULTITHREADING

Example: Running tasks as threads

```
...
public static void main(String[] args) {
     Counter c = new Counter();
     for(int i=0; i<3; i++) { // create three simultaneous threads
                Thread t = new Thread(c);
                t.start(); // starts the thread
      }
}
...
```

See files TestLecture3aCounter.java and Counter.java on Blackboard

MULTITHREADING

Example: Running tasks as threads (cont'd)

```
public class Counter implements Runnable { // the task class
   private int totalCounterNum = 0, limit = 4;

  private void pause(double seconds) {
     try { Thread.sleep(Math.round(1000*seconds)); }
      catch(InterruptedException e) { … }
  }

  public void run() { // the task (the code the respective thread should execute)
    int counterNum = totalCounterNum++;
    for(int i=0; i<=limit; i++) {
        System.out.printf("Counter %s: %s%n",counterNum, i);
        pause(Math.random());
    }
  }
}
```

MULTITHREADING

How to create and run threads?

- By the way, the previous example also shows how to pause the current thread for a number of milliseconds, namely using static(!) method `Thread.sleep`:

```
try {
   Thread.sleep(Math.round(1000*seconds));
} catch(InterruptedException e) {
    System.out.println(e);
}
```

MULTITHREADING

Example: Running tasks as threads (cont'd)

```
Counter 0: 0
Counter 1: 0
Counter 2: 0
Counter 1: 1
Counter 1: 2
Counter 0: 1
Counter 1: 3
Counter 2: 1
Counter 0: 2
Counter 0: 3
Counter 1: 4
Counter 2: 2
Counter 2: 3
Counter 0: 4
Counter 2: 4
...
```

$\Rightarrow$ The precise processing times of a thread which runs simultaneous with other threads is normally <u>not foreseeable</u>!

$\Rightarrow$ Only as long as these threads don't share anything (e.g., variables, files, ...), this is not a problem

MULTITHREADING

The class Thread

«interface»
java.lang.Runnable

java.lang.Thread

+Thread()	Creates a default thread.
+Thread(task: Runnable)	Creates a thread for a specified task.
+start(): void	Starts the thread that causes the run() method to be invoked by the JVM.
+isAlive(): boolean	Tests whether the thread is not dead.
+setPriority(p: int): void	Sets priority p (ranging from 1 to 10) for this thread.
+join(): void	Waits for this thread to finish.
+<u>sleep(millis: long): void</u>	Puts the runnable object to sleep for a specified time in milliseconds.
+<u>yield(): void</u>	Causes a thread to temporarily pause and allow other threads to execute.
+interrupt(): void	Interrupts this thread.

MULTITHREADING

Race conditions and monitors (file BuggyCode.java)

- Multithreading is very powerful but can be problematic if two or more threads can access the same resource (e.g., a shared variable or file). E.g., the following code is erroneous:

```
public class BuggyCode implements Runnable { // task class
  private int gc = 0; // we need this to generate "unique" numbers

  public void init() { // we create the threads from within a
                       // method in the task class (that's fine)
   Thread t;
   for(int i=1; i<=4; i++) { // we create four threads
          t = new Thread(this);
          t.start(); // launches the new thread
   }
  }
```

MULTITHREADING

Race conditions and monitors

(buggy example continued)

```
public void run() {  // the task simultaneously performed by each
   // of the threads

     // for some reason, we now want to compute a number which is unique for this thread:

     int uniqueThreadNumber = gc;

     System.out.println("Set uniqueThreadNumber to "+
         uniqueThreadNumber);

     gc = gc + 1; // we increase gc, so that the next thread which
       // executes run() uses a new "unique" number

     ...

   }

}
```

MULTITHREADING

Race conditions and monitors

- The example creates four threads which run concurrently. Each thread should use a unique number, to be put into local variable `uniqueThreadNumber` and generated with the help of a global counter (field `gc`) which is increased afterwards.
- However, what might happen instead is something unexpected like this:

```
Set uniqueThreadNumber to 0
Set uniqueThreadNumber to 1
Set uniqueThreadNumber to 1
...
```

(not unique, i.e., an error!)

MULTITHREADING

Race conditions

- What went wrong here…? Multiple threads were accessing and modifying a *shared resource* (in this case variable `gc`) concurrently (i.e., in parallel with each other) in an unprotected way. Concretely:

  At the same time while thread X is executing lines 1 and 2 (but not yet 3!) in

```
line1:  int uniqueThreadNumber = gc;
line2:  System.out.println("Set uniqueThreadNumber to "
         + uniqueThreadNumber);
line3:  gc = gc + 1;
```

  …some other thread Y might execute *the same lines* before thread X has a chance to increase `gc` in line 3…!
  Therefore, each of these threads assigns local variable `uniqueThreadNumber` the *same* number! => error

MULTITHREADING

Race conditions

- Not only a variable can be a shared resource, but <u>any</u> shared information piece or device (e.g., a data structure stored in an object in memory, a field of an object or class, a hardware device, a file, the state of a program or component or device accessible via a network...)

  => Anything which can be accessed by more than one more thread concurrently

- However, if none of the threads is able to modify the shared resource, there is no problem. Only if one or more of the threads can modify (write) their shared resource, the aforementioned type of error can occur!
- For such an error to occur, it is <u>not</u> necessary that the threads execute the same code, just that they access the same resource.

MULTITHREADING

Race conditions

- The previously described problem is a so-called *race condition*: multiple threads access the same resource (e.g., some shared variable like `gc`) simultaneously and the result depends on the timing of this access.
- "Race", because the two threads "compete" for the access of the shared resource
- The problem is the unforeseeable timing of the access (i.e., which thread accesses the resource first). Because the timing is unforeseeable, the result is also unforeseeable, which is unwanted.
- Race conditions are the most common problem with multithreaded programs (but there are also other kinds of concurrency-related problems which we don't cover in this lecture, such as deadlocks!).

MULTITHREADING

Race conditions

- Code is called *thread safe* if it accesses shared resources in a way which cannot cause any race conditions
- There are several ways to make code thread safe. One is to make the shared resource *immutable* ("read-only").
- Making the shared resource immutable is the preferred approach.
- A great way to ensure immutability is to use *Java 8 Streams* (`java.util.stream`) (not to be confused with *I/O streams*). You will learn more about Java 8 Streams later in this semester.
- Java 8 Stream transformation operations don't change data in-place but create modified copies if needed…

MULTITHREADING

Race conditions

- Actually, we don't even need to create threads manually at all with Java 8 Streams, we can simply make a stream parallel and let Java do the multithreading automatically for us (behind the scenes, it still creates threads, of course):

```
a = someList.parallelStream()
            .filter(…)  // operations run in parallel now
            .map(…)
            .average();
```

- Java splits the original stream into different chunks (sub-streams) and lets stream operations (like `map`) operate in parallel on these chunks, independently from each other using different threads.
  => Java 8 streams are a great way to do parallel computing in Java
- Requirement: the actual operations (e.g., what you want `map`, filter, etc to do) must not perform any mutation of shared resources!

MULTITHREADING

Race conditions

- But sometimes, immutability is impossible or difficult to achieve, so we need to look at other approaches too...
- Also, parallel computing with Java 8 Streams is still less common in some industry settings than "traditional" multithreading

MULTITHREADING

Race conditions

- Another simple solution attempt (not a real solution!) looks like this:

```
public void run() {
  int uniqueThreadNumber = gc++;  // seemingly an atomic
    // operation (is it?), seemingly cannot be interrupted by
    // other threads (can't it?)
  System.out.println("Setting uniqueThreadNumber to "
      + uniqueThreadNumber);
  ...
}
```

  - However, there are two problems: firstly, `gc++` is actually not atomic in Java.
    Secondly, not all operations could be made atomic even if we had a way to perform simple operations like incrementing variables in an atomic way.

MULTITHREADING

Race conditions

- A few actually atomic operations are provided in Java >=7 with

  https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html
- They allow, e.g., for atomic increments. Could be used to correct our buggy example from a few slides ago!
- But not available for more complex tasks

MULTITHREADING

Race conditions (file FixedCodeSynchronized.java)

- Another solution is to use keyword `synchronized:` very general and very common, but less efficient than immutability or atomicity:

```
public void run() {
  ...
  synchronized(this) {  // forbids concurrent access of
     // the following block (the "critical section"):
     int uniqueThreadNumber = gc;
     System.out.println("Setting uniqueThreadNumber to "
          +  uniqueThreadNumber );
     gc = gc + 1;
      ...
  }
}
```

MULTITHREADING

Race conditions

- Threads must be prevented from simultaneously entering certain parts of the program, known as *critical sections.*
- A critical section (the green block of code in the example) is where the concurrent access of a shared resource takes place.
- If a block of code is "guarded" using `synchronized(`*objectx*`) { … }` no thread can enter this block if some other thread is currently executing this block for the same *object* or any other block of code which is also guarded by `synchronized(`*objectx*`){...}`
- In other words, `synchronized` prevents two or more threads from simultaneously accessing some resource which is shared among them.
- *objectx* can be any Java object, but typically it is the shared resource itself (provided the shared resource is representable as a Java object)

MULTITHREADING

Race conditions

- Often, `synchronized` is used to synchronize *entire methods*:

```
public synchronized void someMethod() {
      ...   // critical section
}
```

- Assume that some thread has invoked this method on object `someObject` (i.e., using `someObject.someMethod()`).

  Then no other thread can enter this method if that other thread called the method also on object `someObject` - until the first thread has finished executing `someMethod`.

- Threads which are blocked off from entering a synchronized method wait until the first-comer thread has left the method. Then the next thread enters the method and blocks it for any further threads, etc.

MULTITHREADING

Race conditions

- However, it would be possible for other threads to enter method `someMethod` concurrently if those other threads had invoked this method on a *different* object (e.g., using `anotherObject.someMethod()`).
- In other words, `synchronized` protects a method only for the particular object on which the method is called.
- The next slide explains how Java achieves this…

MULTITHREADING

Race conditions

- So, what happens here technically...?

```
public synchronized void someMethod() {
    ...
}
```

- Every Java object has internally a so-called *lock*
- When a thread invokes `someObject.someMethod()`, it tries to *acquire the lock* of `someObject`

  (Analogously when a thread enters a `synchronized(`*object*`) { ... }` –block of code)

- After a thread has successfully acquired the lock of the object, any other thread which tries to acquire the *same object's* lock *blocks* (waits) until the first thread *releases* that lock (i.e., until it leaves the synchronized method or block of code).

MULTITHREADING

Race conditions

- Remark:

```
synchronized method() {
    ...
}
```

is just "syntactic sugar" for

```
method() {
    synchronized(this) {
        ...
    }
}
```

MULTITHREADING

Race conditions

- <u>Question</u>: Why does the following work as expected, even though we synchronize on "`this`" instead of object `gc` directly?

```
public void run() {
   ...
   synchronized(this) {  // forbids concurrent access of
      // the following block (the "critical section"):
      int uniqueThreadNumber = gc;
      System.out.println("Setting uniqueThreadNumber to "
            +  uniqueThreadNumber );
      gc = gc + 1;
       ...
   }
}
```

MULTITHREADING

Race conditions

- Answer: All involved threads had been launched using the same task class object, that is, "`this`" always represents the same object here(*) for all four threads, since for each thread Java called `run()` on the very same task class object (represented by keyword "`this`").
- Therefore, the first thread who enters the synchronized code block blocks off all subsequent threads trying to enter this block of code (as the other threads also had been called on the same object represented by "`this`"), until it has finished executing the code block.

(*) This is not necessarily so, but in this particular example

MULTITHREADING

Race conditions

- Could we have achieved the same effect by using `synchronized(gc)`?
- **Answer: no. Java does not allow synchronization on primitive variables such as int (and you will get a compiler error)**

```
public void run() {
   ...
   synchronized(gc) {  // compiler error
       int uniqueThreadNumber = gc;
       System.out.println("Setting uniqueThreadNumber to "
            +  uniqueThreadNumber );
       gc = gc + 1;
        ...
   }
}
```

MULTITHREADING

Race conditions (file BuggyCodeSynchronizedIntObject.java)

- Could we have achieved the same effect by using `synchronized(gc)` if gc is a boxed int (i.e. instance of the Integer class)?
- **Answer: no. Java does allow you to synchronize on boxed primitive variables such as Integer, but you shouldn't do this -> race conditions!**

```
// using a boxed int instead. This code compiles.
private Integer gc = new Integer(0); public void run() {
  ...
  synchronized(gc) {  // synchronizing on a boxed int.
                      // Don't do this!
      int uniqueThreadNumber = gc;
      System.out.println("Setting uniqueThreadNumber to "
          + uniqueThreadNumber );
      gc = gc + 1;
       ...
  }
}
```

MULTITHREADING

Race conditions (file FixedCodeAtomicInt.java)

- The AtomicInteger class provides a thread safe integer that can be used for e.g. counters as per our example
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html

```
// AtomicInteger can be used for atomically incremented
   (i.e. thread safe) counters
private AtomicInteger gc = new AtomicInteger(0);


public void run() {
  /* getAndIncrement() is an atomic operation that
  increments gc and returns the previous (not incremented)
  value. Note how this allows us to avoid wrapping the
  code in run() in a synchronized block */
   int uniqueThreadNumber = gc.getAndIncrement();
   System.out.println("Setting uniqueThreadNumber to " +
  uniqueThreadNumber );
  }
```

MULTITHREADING

Race conditions and monitors

- The Java approach to multithread-synchronization (attempting to prevent race conditions) is called the *monitor approach*.
- It provides for
  - *Mutual exclusion* (using the `synchronized` keyword)
  - *Thread cooperation* using so-called *condition variables*
- Topic to be continued in the next lecture…