IN5170 Models of Concurrency

Self-study material: Basics about Concurrency in Java

Andrea Pferscher

August 27, 2025

University of Oslo

Threads Basics

Threads in Java (I/IV)

- Threads are units of execution that allow programs to perform multiple tasks simultaneously
- An application begins with a single thread, called the main thread. It is possible to create threads from this main thread.

Processes vs. threads (in Java)

- A process is an independent instance running on its own memory space.
- A thread runs inside a process and shares its resources with other threads

- Here we focus on threads, multi-process applications in Java are possible but come with OS/JVM specific issues
- Both are handled by OS scheduler
- Both have costly context switches but threads are more light-weight
    - Only processes require full cache flushing as they change virtual memory
    - Thread-switch can retain caches, only changes processor state (registers etc.)

Threads in Java (II/IV)

- The `Thread` class encapsulates a system thread
- The `Runnable` interface is used to define thread behavior

Java

```java
class Printer implements Runnable {
  private String text;
  public Printer(String text) { this.text = text; }
  public void run() { System.out.println(text); }
  public static void main(String args[]) {
    Thread t1 = new Thread(new Printer("Hello"));
    Thread t2 = new Thread(new Printer("Concurrency"));
    t1.start(); t2.start();
  }}
```

Threads in Java (III/IV)

`Start` vs `run`

- `Thread.start()` starts a new *concurrent* thread
- `Runnable.run()` just executes the code *sequentially*
- `Thread.start()` calls `Runnable.run()` internally
- Calling `Runnable.run()` directly rarely makes sense

Threads in Java (IV/IV)

Common pattern for anonymous runnables to start a thread

Java

```java
// option 1 with Runnable interface
new Thread(new Runnable(){
    public void run() { /* do things */ }}).start();
// option 2 with lambdas
new Thread ( () -> { /* do things */ } ).start();
```

- Rather than declare a class that implements Runnable, it is possible to create an instance of that class, and pass that instance to the thread constructor.
- Lambdas offer a convenient short-cut to define an anonymous runnables more concisely and directly.

Shared State (I/III)

A shared state between threads is introduced through multiple means

- **Static state**
- Shared references to objects
- Resources, e.g., files

Java

```java
class C { public static int i = 0; } ...
new Thread ( () -> { C.i++; } ).start();
new Thread ( () -> { C.i++; } ).start();
```

A variable declared as static means there is only one copy of it for the entire class becoming a shared variable.

Shared State (II/III)

A shared state between threads is introduced through multiple means

- Static state
- **Shared references to objects**
- Resources, e.g., files

Java

```java
public class C { public int i = 0; } ...
final C c = new C();
new Thread ( () -> { c.i++ } ).start();
new Thread ( () -> { c.i++ } ).start();
```

What is final (immutable) is the reference to the object c, however its field i acts as a shared variable between the threads

Shared State (III/III)

Shared state between threads is introduced through multiple means

- Static state
- Shared references to objects
- **Resources, e.g., files**

Java

```java
new Thread ( () -> { new File("/path/").delete();} ).start();
new Thread ( () -> { new File("/path/").delete();} ).start();
```

In this case, one could have two different links pointing to the same file,
becoming a shared resource, which means that the program will try to delete the file twice.

Methods of Java Threads (I/III)

- Java offers some additional operations that are abstracted away in Await
- Methods of `Thread` object:
    - `join` **waits for the thread to finish**
    - `sleep` suspends the thread for at least *n* milliseconds
    - `yield` gives the scheduler the signal to schedule someone else first

Java

```
public static void main(String args[]) {
    Thread t1 = new Thread(new Printer("Hello"));
    Thread t2 = new Thread(new Printer("Concurrency"));
    t1.start(); t1.join(); // waits for t1 to finish
    t2.start();
  }
```

Methods of Java Threads (II/III)

- Java offers some additional operations that are abstracted away in Await
- Methods of `Thread` object:
  - `join` waits for the thread to finish
  - `sleep` **suspends the thread for at least *n* milliseconds**
  - `yield` gives the scheduler the signal to schedule someone else first

Java

```
int i = 0; //shared
  ...
  return m(){
    while(i == 0) Thread.sleep(10); //some constant chosen
    return 10/i;
  }
```

Methods of Java Threads (III/III)

- Java offers some additional operations that are abstracted away in Await
- Methods of `Thread` object:
  - `join` waits for the thread to finish
  - `sleep` suspends the thread for at least *n* milliseconds
  - `yield` **gives the scheduler the signal that it is possible to schedule someone else first**

Await

```
int i = 0; //shared
  ...
  return m(){
    while(i == 0) Thread.yield();
    return 10/i;
  }
```

Quiz: Java and Await

Await

```
co s1 || s2 oc; s3
```

Java

```
Thread t1 = new Thread(() -> {s1});
Thread t2 = new Thread(() -> {s2});
Thread t3 = new Thread(() -> {s3});
??
```

Quiz: Java and Await

Await

```
co s1 || s2 oc; s3
```

Java

```
Thread t1 = new Thread(() -> {s1});
Thread t2 = new Thread(() -> {s2});
Thread t3 = new Thread(() -> {s3});
t1.start(); t2.start();
t1.join(); t2.join();
t3.start();
```

In the remainder of this presentation, we mostly show the code inside the threads and omit the boilerplate code for the `Thread`/`Runnables`

Atomic Blocks and `synchronized`

Synchronization

- Java does not have atomic blocks in the same form as described in the Await language
- Instead it is possible to use synchronized methods and blocks

Java

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {c++;}
    public synchronized void decrement() {c--;}
    public synchronized int value() {return c;}
}
```

- Synchronized methods are atomic *per object*
- Only one thread can execute such a method at any time

Synchronization – Synchronized Methods

- A synchronized method locks the instance of the *object* that the method belongs to.
- All synchronized methods are synchronized with each other
- No two synchronized methods can be executed at the same time *in one instance*

Java

```java
SynchronizedCounter c1 = new SynchronizedCounter();
SynchronizedCounter c2 = new SynchronizedCounter();
Runnable r1 = new Runnable(){
    public void run() { c1.increment(); } }
Runnable r2 = new Runnable(){
    public void run() { c1.increment(); } }
Runnable r3 = new Runnable(){
    public void run() { c1.decrement(); } }
Runnable r4 = new Runnable(){
    public void run() { c2.increment(); } }
```

Synchronization – Synchronized Methods (continuation)

The following will not interleave on c1:

Java

```java
new Thread(r1).start();
new Thread(r2).start();
```

The following will also not interleave on c1:

Java

```java
new Thread(r1).start();
new Thread(r3).start();
```

Synchronization – Synchronized Methods (continuation)

The following will interleave – the call is to two *different objects*

Java

```
new Thread(r1).start();
new Thread(r4).start();
```

Synchronization – Synchronized Static Methods

- Synchronized static methods are *per class*
- A synchronized static methods locks the Class object, meaning it affects all instances of the class rather than only one particular instance.

Java

```
public class StaticSyncCounter {
    private static int c = 0;
    public synchronized static void increment() {c++;}
    public synchronized static void decrement() {c--;}
    public synchronized static int value() {return c;}
}
```

Synchronization – Synchronized Static Methods (continuation)

Java

```java
Runnable s1 = new Runnable(){
    public void run() { StaticSyncCounter.increment(); } }
Runnable s2 = new Runnable(){
    public void run() { StaticSyncCounter.decrement(); } }
```

The following code that starts two instances that will not interleave

Java

```java
new Thread(s1).start();
new Thread(s2).start();
```

Synchronization – Synchronized Blocks

- Synchronized blocks allow to synchronize only *a part* of a code, rather than an entire method.
- **One need to give the object from which the lock should be obtained. Any object can be a lock.**
- Synchronized methods can have **this** as the lock

Java

```
public class C(){
  int l = 0;
  int r = 0;
  void method(Object lock, boolean left){
    synchronized(lock){
      if(left) l++ else r++;
    }
  }
}
```

Synchronization – Synchronized Blocks (continuation)

- Synchronized blocks allows to synchronize only *a part* of a code, rather than an entire method.
- **One need to give the object from which the lock should be obtained. Any object can be a lock.**
- Synchronized methods can have **this** as the lock

The following will interleave

Java

```java
C c    = new C();
Object o1 = new Object();
Object o2 = new Object();
//thread 1 :
c.method(o1, true);
//thread 2 :
c.method(o2, false);
```

Synchronization – Synchronized Blocks (continuation)

- Synchronized blocks allows to synchronize only *a part* of a code, rather than an entire method.
- One need to give the object from which the lock should be obtained. Any object can be a lock.
- **Synchronized methods can have *this* as the lock**

Both code snippets are equivalent

Java

```java
public class C(){synchronized void method(){ ... } }
```

Java

```java
public class C(){ void method(){ synchronized(this) {...} }}
```

Atomic Expressions

JVM

Java is compiled down to JVM bytecode, which does not correspond 1-to-1 to machine instructions

Java

```
x++; // x is only local variable, declared as long
```

How many machine instructions will this be? **4-6**

JVM

The JVM has no registers, but loads from variables/heap onto a stack. All computations target the top values on the stack.

Atomic Expressions

JVM

Java is compiled down to JVM bytecode, which does not correspond 1-to-1 to machine instructions

Java

```
x++; // x is only local variable, declared as long
```

How many machine instructions will this be? **4-6**

```
LLOAD_1      // push value from local variable #1
LCONST_1     // push value 1
LADD         // add 2 top-most values
LSTORE_1     // store value into local variable #1
```

Atomic Expressions

JVM

Java is compiled down to JVM bytecode, which does not correspond 1-to-1 to machine instructions

Java

```
x++; // x is only local variable, declared as long
```

How many machine instructions will this be? **4-6**

- Reference reads and writes are atomic
- Basic type reads and writes except `long` and `double` are guaranteed to be atomic
- On 64bit machines, `long` and `double` reads and writes *might* be atomic, *might* be two instructions
- Accessing variables modified by `volatile` is always atomic

Quiz: Java and Await

Await

co $<$s1$>$ || $<$s2$>$ **oc**

Java

```
public class C() {
??
s1
??
s2
??
}
```

Quiz: Java and Await

Await

co <s1> || <s2> **oc**

Java

```
public class C() {
public static synchronized void m1(){s1}
public static synchronized void m2(){s2}
...

new Thread(() -> {C.m1();}).start();
new Thread(() -> {C.m2();}).start();
}
```

Weak Memory and `volatile`

Weak Memory

Java memory model

The JVM defines a *weak* memory model.

A weak memory model allows certain reordering in read and write operations.

- Mainly targeting performance, e.g., for cache optimization
- Can be done statically (by compiler) or dynamically (by processor)
- Which reorderings are allowed exactly, is defined by the memory model
- Strong memory model = no reorderings are allowed

Weak Memory

Independence

Reordering must take into account whether the operations are independent

- `x := 1; r := x;` cannot be reordered
- `r := x; x := 1;` cannot be reordered

because the result can change if the order of execution is changed

- Read-read reordering can reorder reads
- Write-read reordering can move a read before a write
- Read-write reorderings can move a write before a read
- Write-write reorderings change order of stores
- Some architectures also reorder other atomic operations

Reordering can happen if operations refer to different memory locations and therefore the order of execution does not affect the result of the execution.

Weak memory

Weak memory models can lead to very unintuitive results in concurrent settings

Assume `P1` and `P2` execute concurrently.

```
int x,y; //default 0

   P1:                                      P2:
    x  := 1;  //shared variable              y  := 1;  //shared variable
    r1 := y;  //register                     r2 := x;
    print r1;                                print r2;
```

What are possible outputs? Is `0,0` possible?

- If the read of `x` in `P2` and read of `y` in `P1` is reordered, then `0,0` is possible, e.g., the compiler may decide to do such reordering in `P1` and `P2` since locally the result will not change
- This output cannot be explained by reasoning about interleavings
- If the language does not require variables to be initialized, we get *out-of-thin-air* values.

Weak Memory

Sequential consistency

Most weak memory models guarantee *sequential consistency*: the result of execution is the same as if all the operations were executed in some sequential order. *If there is no data race, then the observable behavior of the program is as if under a strong memory model*.

`volatile` can help to mitigate some of the challenges posed by weak memory models by providing specific guarantees about the visibility and ordering of variables declared as `volatile`.

Weak Memory

Java

```
public class C {
  private volatile long l = 5;
  long incRet() { return l++; } //called from two threads
}
```

- All read and write accesses to `l` are atomic
- All write accesses to `l` are *immediately* visible to all threads
- In terms of memory model: no reads and writes to `l` are reordered before any write
- In terms of memory: `l` is read and written from global memory, not thread caches
- Does *not* introduce synchronization, but removes opportunities for optimization and makes access more expensive

Weak Memory

Weak memory is a complex topic, mostly relevant to low level architectures and compilers

- Further operations: read-own-write-early, read-others-write-early
- Leaks to programmer in concurrent settings
- Hard to debug, most languages have no clearly defined memory model
- Often hardware-dependent solutions

Rough guideline on when to use `volatile`

- If a field is not supposed to have data races, do not use volatile
- If a field will have data races, and you do not want to remove them, consider using volatile to avoid unintuitive behavior

Further Concepts

Java’s Standard Library

- Java’s standard library offers further data structures for common patterns or to encapsulate complex but efficient solutions
- Thread-safe collections (e.g., lists, maps, etc.) are less efficient, but internally race-free versions of collections
- Atomic classes encapsulate data with efficient, atomic access

Standard Library – Atomic Classes

- Atomic classes are available for data and references
- Operations that are atomic without `synchronized` blocks are more efficient (less blocking) but they can affect control flow and requires a deep understanding of how the operations work, and how they are used in the program e.g., the method `compareAndSet(a,b)` of `AtomicInteger` atomically updates the value to `b` if the current value matches `a`. The programmer will need to understand in the current control flow if the value will or will not be updated.
- `int` vs. `AtomicInteger`
  - `int` is a primitive data type. It is not thread-safe for operations that require atomicity. Multiple threads cannot safely perform operations that modify its value (e.g., incrementing the value) without additional synchronization.
  - `AtomicInteger` is thread-safe. Multiple threads can update a shared integer value without the risk of race conditions.

Standard Library – Atomic Classes (continuation)

Java

```java
public class SynchronizedCounter { private int c = 0;
    public synchronized void increment() {c++;}
    public synchronized void decrement() {c--;}
    public synchronized int value() {return c;} }
```

vs.

Java

```java
public class SynchronizedCounter {
    private AtomicInteger c = new AtomicInteger(0);
    public void increment() {c.incrementAndGet();}
    public void decrement() {c.decrementAndGet();}
    public int value() {return c.get();} }
```

Standard Library – Atomic Classes (continuation)

`AtomicReference` does *not* make atomic the content of the called methods.

Java

```java
AtomicReference<C> cache = new AtomicReference<C>();
cache.get();       // atomic
cache.get().m();  // the content of m is not executed atomically
```

Standard Library – Concurrent Collections

- Thread-safe collections provide atomic methods to access collections e.g., lists etc.

Examples of concurrent collections

Concurrent collections provide concurrent implementations that enable concurrent access

- `Map` vs. `ConcurrentMap`
    - Most implementations of `Map` are not thread-safe, e.g., `HashMap` is not thread-safe, meaning they can lead to race conditions and unpredictable behavior if accessed by multiple threads concurrently.
    - `ConcurrentMap` is thread-safe. All of its methods are designed to handle concurrent access from multiple threads without requiring explicit synchronization.
- `ConcurrentLinkedQueue` and variants for lists without random access
- `CopyOnWriteArrayList` makes an `ArrayList` concurrent by making a copy on every write, this is not efficient

Standard Library – Concurrent Collections (continuation)

Synchronized collections

Normal implementations, but add **synchronized** at the right places.

Java

```
<T> Collection<T> synchronizedCollection(Collection<T> c)
```

Java

```
ArrayList<Object> a = new ArrayList<>();
Collection<Object> b = Collections.synchronizedCollection(a);
// access through the object a is still unsafe
```

Non-thread-safe implementations of collections e.g., `Map`, tend to be faster in single-threaded environments or when explicit synchronization is not needed.

Thread Management

- In bigger applications, you may need to manage sets of threads
- We consider three concepts
    - Lifecycle of a thread object
    - Interrupts
    - Thread pools

Lifecycle

- A created thread object is **new**
- After calling `start` the thread is either
    - **Running**, i.e., executes right now
    - **Runnable**, i.e., waits to be scheduled (`yields` of other threads get you here)
    - **Waiting**/**Sleeping**/**Blocked**, i.e., waits for time to pass or some notification or lock
- Once the internal `run` method terminates, the object is **dead**
- Once a thread is dead it cannot be restarted

Interrupts

An interrupt is an *indication* to a thread that it should stop what it is doing and reconsider.

- Can be invoked using `t.interrupt();`
- This sets the `Thread.interrupted` flag, however the programmer must take care of reacting to this flag.
- `Thread.interrupted()` checks if the interrupt flag of the current thread has been set. This method also clears the interrupt flag when it is checked.
- Some methods will throw a `InterruptedException` if active (e.g., `Thread.sleep()` Throws `InterruptedException` if the thread is interrupted while it is sleeping)

Java

```java
//long computation 1
    if(Thread.interrupted){ /* handler */ }
    //long computation 2
```

Thread Pools

- Creating and starting threads is costly
- Dead threads cannot be reused
- Solution: create a set of threads that do not terminate, but wait for new runnables to execute
- Automatic scaling

Thread Pools (continuation)

An `ExecutorService` manages a set of threads, and accepts `Runnable` instance submissions

Java

```java
//has exactly 2 threads
ExecutorService service = Executors.newFixedThreadPool(2);
service.submit(() -> { /* do things */ });
service.submit(() -> { /* do things */ });
service.submit(() -> { /* do things */ });
//last runnable put in query, will be executed later
```

Thread Pools (continuation)

An ExecutorService manages a set of threads, and accepts Runnable instance submissions

Java

```java
ExecutorService service = Executors.newCachedThreadPool(0,3);
//starts with 0 threads
service.submit(() -> { /* do things */ });
service.submit(() -> { /* do things */ });
service.submit(() -> { /* do things */ });
//up to 3 threads running
```

Thread Pools (continuation)

- To join on such a task, we get a *Future*
- We will investigate futures in more detail in Part 2 of the course

Java

```
//has exactly 2 threads
ExecutorService service = Executors.newFixedThreadPool(2);
Future<Int> f = service.submit(() -> { /* do */ return 1;});
...
Int = f.get(); //essentially a join
```

- Thread pools have further capabilities (shutdown)
- Details very java-specific, omitted here

Outlook

- We use the material in this slides as the basis for obligs and exercises
- Next lectures connect concepts with corresponding Java concept