MJC LT School
Multithreading

Lectures & demos prepared and conducted by

Danila Varatyntsev

Ivan Mazaliuk

Kiryl Klachkou

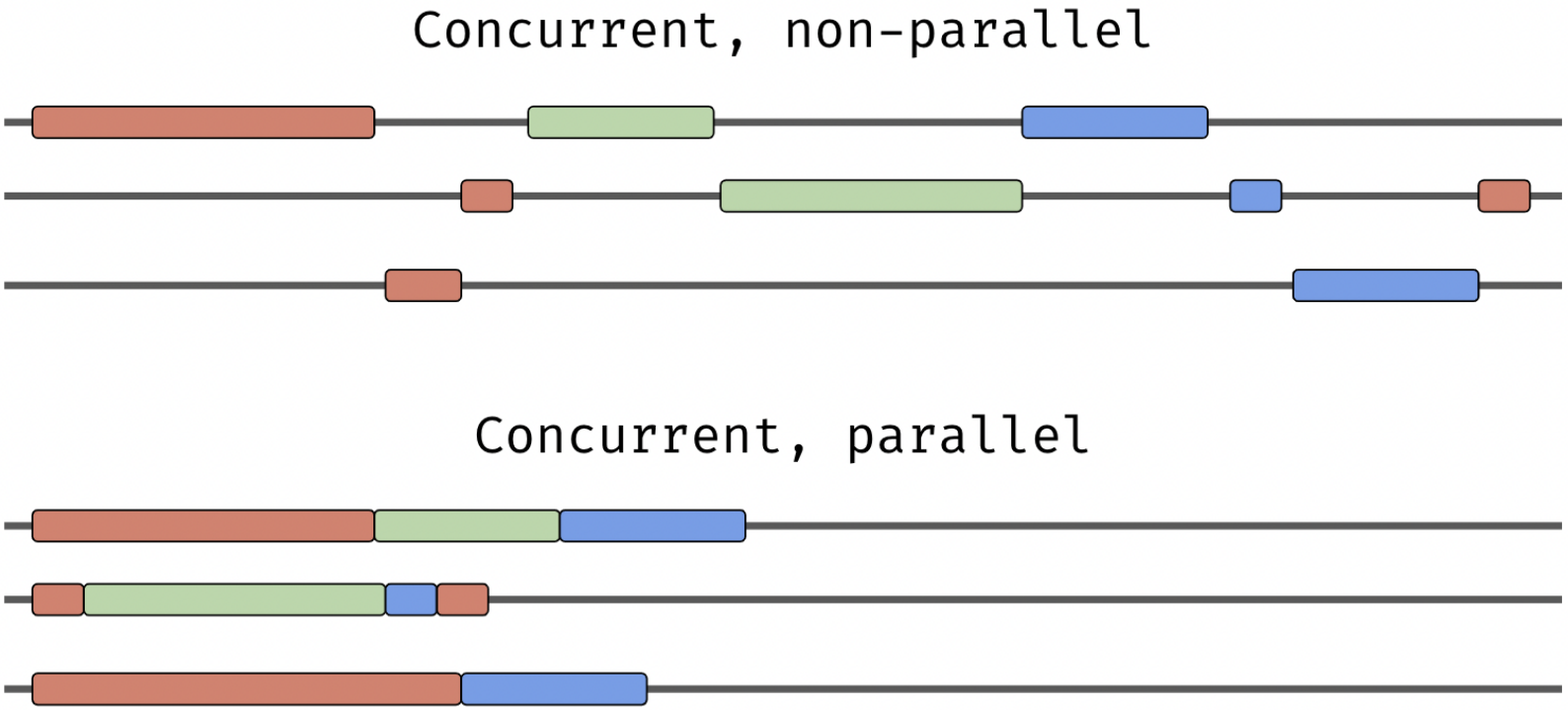
Multithreading key definitions

Multithreading – in computer architecture, is the ability of a central processing unit to provide multiple threads of execution concurrently.

Thread – sequence of code that can be executed separately. May be suspended and resumed.

Scheduler – special system software that defines which threads are executed on CPU cores at each time. They are required since usually there are more threads than cores.

Concurrent vs Parallel



Multithreading in Java

- Java supported multithreading since the very beginning. All standard libraries were developed with multithreading in mind.
- Threads in Java are controlled with **Thread** class.
- Ways of creating thread in Java:
  - Extend **Thread** class
  - Create a new instance of **Thread** with **Runnable** as a constructor argument
- Runnable is a standard Java interface that incapsulates any executable job. Runnable definition:

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

Demo 1

Thread states

NEW
A thread that has not yet started is in this state.

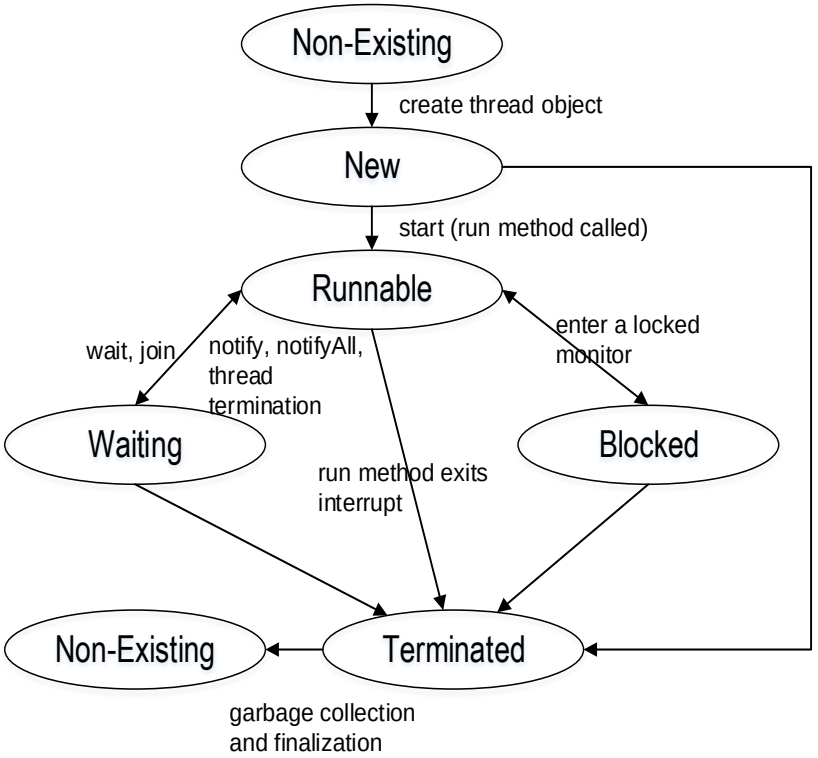
RUNNABLE
A thread executing in the Java virtual machine is in this state.

BLOCKED
A thread that is blocked waiting for a monitor lock is in this state.

WAITING
A thread that is waiting indefinitely for another thread to perform a particular action is in this state.

TIMED_WAITING
A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.

TERMINATED
A thread that has exited is in this state.



Thread methods

Method	Value
getName() / setName()	Returns or sets thread name
join()	Current thread wait for another thread to finish
run()	Thread entry point
start()	**run()** in new thread
getState()	Returns current thread's state as **Thread.State** constant
interrupt()	Interrupt a thread
currentThread()	Returns current thread object
sleep(long millis)	Temporarily ceases execution

isAlive

- **isAlive** – another way to check thread's state.
- **isAlive** and **getState** relation:

getState()	isAlive()
NEW	
RUNNABLE	+
BLOCKED	+
WAITING	+
TIMED_WAITING	+
TERMINATED	

Main thread

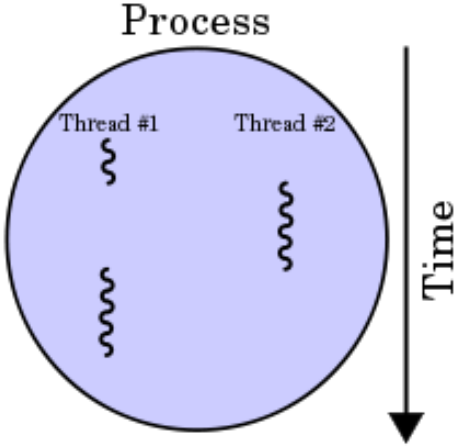
- When a Java program starts up, one thread begins running immediately. This is usually called the *main* thread of our program, because it is the one that is executed when our program begins. It is the thread from which other “child” threads will be spawned.

Demo 2, 3

Thread priorities

- OS Scheduler uses threads' priorities to determine which threads should get more CPU time. In general, threads with higher priority get more CPU time.

- Scheduling example on a single core:



Java thread priorities

Java threads may have priorities from **MIN_PRIORITY** to **MAX_PRIORITY**. Currently these constants are set to 1 and 10, respectively.

The default thread priority is equal **NORM_PRIORITY**, which is currently 5.

You can set a priority with **setPriority(int level)** method and get it with **getPriority()**

Priority of **Main** thread is set to **NORM_PRIORITY.**

Demo 4

Daemon threads

Daemon threads work in background and are not an integral part of Java application.

Java application is finished when all non-daemon threads are finished.

Garbage collection is done by daemons.

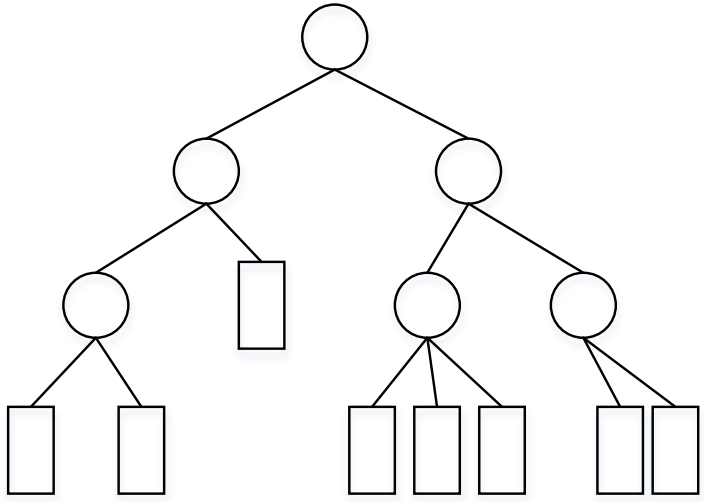
You can call setDaemon(boolean value) before thread starts

You can use getDaemon() if you want to know if thread is a daemon

Demo 5

Thread groups

- Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually. For example, you can start or suspend all the threads within a group with a single method call.
- **ThreadGroup**s can be nested. They form a tree where each **ThreadGroup** except a root one, has a parent .



Thread groups

Method	Action
activeCount()	Number of active threads in group and it's subgroups
interrupt()	Interrupts all threads in group
getParent()	Returns a "parent" thread group or null
isDaemon()	Returns whether group is daemon
setDaemon()	Sets the daemon flag
list()	Prints the info about all threads in group
destroy()	Destroys a group and its subgroups. They must have no threads, IllegalThreadStateException is thrown
setMaxPriority(int) / getMaxPriority()	Sets or returns maximum priority for thread group's threads

Thread groups

- Thread group may be a daemon group. Such thread group is automatically destroyed when becomes empty. This flag doesn’t have any relation to **Thread.isDaemon** flag.

- Thread group max priority:
  - If a thread with a high priority is added to group with lower priority, its priority will become equal thread group’s
  - If a thread with a low priority is added to group with higher priority, its priority will not change

Demo 6

Exception handling

- Parent threads don’t handle uncaught exceptions from child threads. Unhandled exceptions get lost.

- To handle exceptions, you can call:
  - Thread.setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh) – to set global default exception handler
  - thread.setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh) – to set an exception handler for that thread

- Or you can extend ThreadGroup and override uncaughtException(Thread t, Throwable e), that is called when a child thread does not handle exception.

Thread interruption

interrupt(): To interrupt the thread.

interrupted(): To check if the thread is already interrupted but it clears the interrupt status.

isInterrupted(): To check if the thread is already interrupted and does not clear its status.

Demo 7

Synchronization

Synchronization

Synchronization – is required when several threads access the same area in memory. It guarantees that only one thread may access it at a time.

Lack of synchronization may lead to unexpected program results.

A key concept for synchronization is **monitor.** Each Java object (not a primitive) has an associated monitor, that may be used to guarantee mutually exclusive access to a piece of code. Monitors are also called **mutexes**.

Synchronized keyword

- **synchronized** is a Java keyword for synchronizing access to code. It can be used it two ways:
  - Synchronized method – uses monitor of an object that is referred to as **this** inside the method. If the method is **static,** uses monitor of a corresponding **Class** object.

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

  - Synchronized block – uses monitor of **lockObject.**

```
synchronized (lockObject) { ... }
```

- If two threads try to enter a monitor at the same time, only one will succeed. Another sleeps until the first exits synchronized code and releases the monitor.

- When a thread waits for a busy monitor, it's state changes to **BLOCKED**.

Demo 8-10

Wait & notify

- Fine-grained control over the program threads can be achieved through communication between threads.

Method	Effect
wait()	Causes the current thread to wait until it is awakened, typically by being notified or interrupted
notify()	Wakes up a single thread that is waiting on this object's monitor
notifyAll()	Wakes up all threads that are waiting on this object's monitor

Demo 11

Deadlock

- Deadlock is a special type of errors in code when two threads have cycle dependency in pair of synchronized objects. In other words:
  - Thread-1 owns monitor-1 and waits for monitor-2
  - Thread-2 owns monitor-2 and waits for monitor-1
- Deadlock may include more than 2 threads, but such cases are uncommon.
- One way of avoiding deadlocks is to enter monitors of objects in the same order.
- Detect deadlock using tools like JStack available in DJK package.

Demo 12

Suspending and resuming threads

- You can pause an execution of a thread with **suspend()** method. Later it can be continued with **resume()** call.

- Methods are **deprecated** because they can easily lead to deadlocks in application. These methods should be used rarely with a great caution.

Volatile

- An undesired effect is allowed by Java Memory Model. If a usual variable is shared between multiple threads and one of them makes changes to it, others are not guaranteed to see that change on subsequent reads.
- The following code may run forever even if **run** flag is updated to **false from another thread.**

```
boolean isRun = true;
....
while (isRun) {do smth}
```

- To avoid such problems shared variables should be made **volatile.** Volatile variables are not cached in processors, reads are directed to main memory, writes are immediately flushed there, too.
- Only variables that are shared between threads may be made **volatile,** thus you can't make a local variable **volatile**.

Demo 13

Parallel Streams

Java Parallel Streams is a feature of Java 8 and higher, meant for utilizing multiple cores of the processor. ... Whereas by using parallel streams, we can divide the code into multiple streams that are executed in parallel on separate cores and the final result is the combination of the individual outcomes.

java.util.concurrent

Concurrent

Java 5 introduced a new **java.util.concurrent** package that contains many classes useful for multithreaded programs:

- java.util.concurrent – thread safe collections and executor services
- java.util.concurrent.locks – locking facilities
- java.util.concurrent.atomic – atomic wrappers for primitives and references

Locks

Locks are used instead of synchronized methods and code blocks to provide mutual exclusive access.

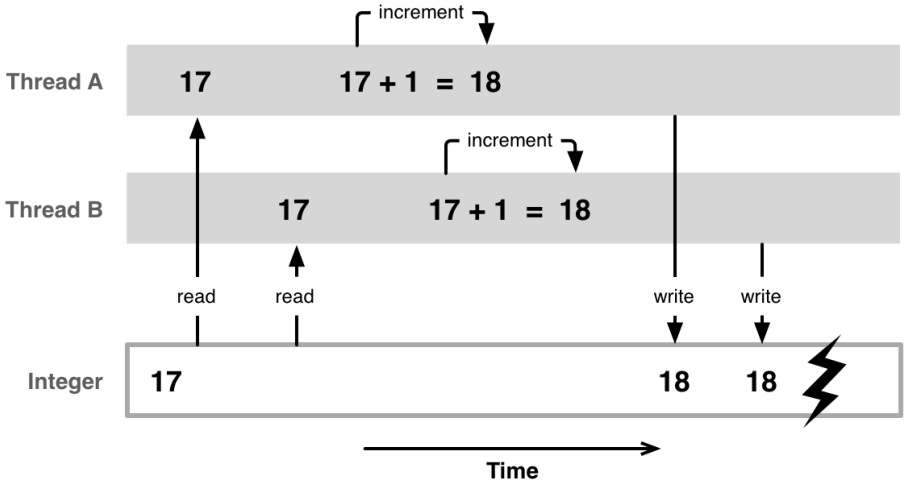
Lock may be acquired with **lock()** and later released with **unlock()** methods

Comparing to synchronized keyword, locks allow to specify wait timeout to write more error-resilient programs.

Demo 14

Atomic

- Atomic operations are operations that can't be divided by OS scheduler. Also, it's impossible to observe intermediate state during it.
- Usual java increment for both volatile and non-volatile variables takes 3 actions:
    1. reading variable,
    2. adding 1 to it,
    3. writing it back to memory.
- Between steps 1 and 2, for example, another thread can increment the same variable, which will lead to a lost update.
- In Java some operations may be made atomic using **AtomicInteger**, **AtomicBoolean**, **AtomicLong**, **AtomicReference** and some more.



Atomic integer internals

- **AtomicInteger** holds a volatile int.
- For many operations it uses CAS (Compare-and-set) native instructions. CAS is provided by many processor architectures. It allows to atomically set a value of a field only if it equals some specified value.
- Main methods of **AtomicInteger:**

Method	Effect
get()	Returns current value
set(value)	Sets the value without any conditions
compareAndSet(expected, newValue)	Atomically sets the value only if current equals **expected**
addAndGet(delta)	Adds **delta** to current value and returns result
incrementAndGet()	Increments current value and returns result

Demo 15

Collections

- java.util.concurrent provides new collections, that may be safely used in multithreaded environment. Unsynchronized access to standard Java collections may result in exceptions, lost updates and other types of undesired behavior.
- Concurrent collections include:
    1. Lists
    2. Maps
    3. Sets
    4. Queues
    5. Deques

Demo 16

Executors

- **java.util.concurrent** provides few classes that allow to execute tasks without knowing about underlying threads. All executors have a thread pool, that is used to run submitted tasks:
  - **Executor** – can **execute(Runnable)**, doesn't return any value
  - **ExecutorService** – introduces **Callable** interface, that incapsulates runnable code and returns some value.
  - **ScheduledExecutorService** – allows to schedule execution of **Callable** and **Runnable** tasks
  - Executors – utility class for creating Executors
- Types of executors' thread pools:
  - Single-threaded pool
  - Fixed thread pool
  - Cached thread pool

Demo 17-18

Callable and Future

- Callable interface the following definition:

```
public interface Callable<V> {
    V call() throws Exception;
}
```

- Future<V> – Java interface that wraps some value that will be available in future.

Future<V> methods	Effect
V get()	Blocks until result is available
V get(long timeout, TimeUnit unit)	Blocks until result is available, but no longer than timeout
cancel(boolean mayInterrupt)	Cancels **Future** execution, interrupting thread, if argument is true and future started executing
isCancelled() / isDone()	Returns boolean that indicates whether future is cancelled or done

Demo 19

Thanks for your attention, the main part of this presentation is over.

Fork-Join pool

- Fork-Join pool – framework for executing divide-and-conquer tasks. This is a class of tasks that are executed in the following manner:
  - If task is small enough, thread calculates its results (array to sort has just few elements)
  - Otherwise, task is split into several subtasks, that are executed in arbitrary threads (**fork** stage)
  - Subtasks may also be split, and so on
  - Task waits for subtasks to complete (**join** stage)
  - When subtasks are finished, task aggregates their results and returns to the caller
- Fork-Join pool can execute two different types of tasks:
  - **RecursiveAction –** doesn't return any result, acts like **Runnable**
  - **RecursiveTask –** returns some value, acts like **Callable**

Demo 20

Some more classes worth to review

Class	Short description
AtomicIntegerArray	Atomic version of **int[]**
LongAdder	Collection of **AtomicLong** for better “add” operation performance under the heavy contention
ReentrantReadWriteLock	Lock that provides separation of reader and writer threads
Semaphore	Synchronization class that may allow multiple threads pass it simultaneously
CountDownLatch	A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
CompletableFuture	A Future that supports dependent functions and actions that trigger upon its completion.
Phaser	A reusable synchronization barrier, provides more flexible usage than **CountDownLatch**