MJC LT School
Multithreading in Java

Light talk authors

Danila Varatyntsev,
Senior Software Engineer

Inomjon Kadirov,
Software Engineer

Key definitions

Multitasking - multiple tasks/processes running concurrently on a CPU. The operating system executes these tasks by switching between them very frequently. The unit of concurrency, in this case, is a Process

Multithreading - multiple parts of the same program running concurrently. In this case, we go a step further and divide the same program into multiple parts/threads and run those threads concurrently

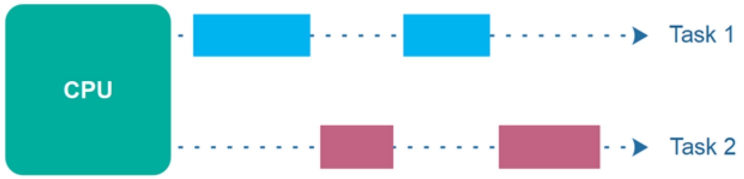
Process - a program in execution. It has its own address space, a call stack, and link to any resources such as open files

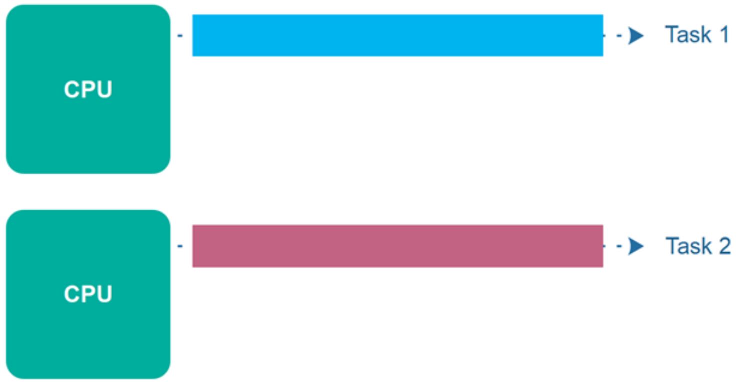
Thread – part of running process. A thread refers to a lightweight process. A thread will use the process's memory area or execution environment. Context switch time between threads is less because switch is done within the same process's execution environment or memory area.

Concurrent VS Parallel execution

Concurrency - A condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism

Parallelism - A condition that arises when at least two threads are executing simultaneously





Multithreading in Java

Java supported multithreading since the very beginning. All standard libraries were developed with multithreading in mind

Threads in Java are controlled through Thread class

A Multithreaded program contains two or more parts that can run concurrently

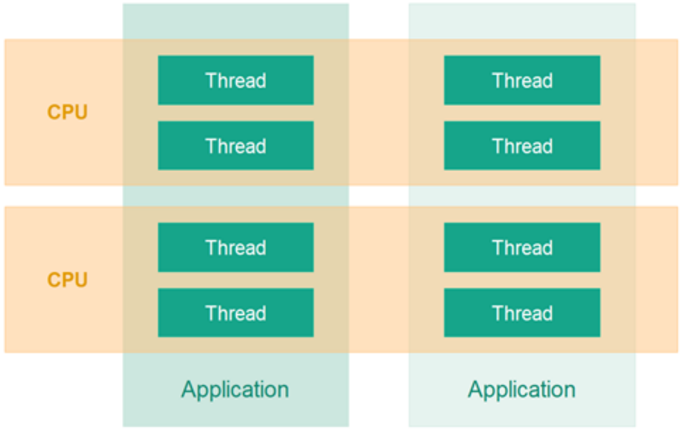
Each part of multithreaded program is called Thread

Each thread defines a separate path of execution

Threads can be created in the following ways:

- Extending Thread class;
- Implementing Runnable interface.

Either of these approaches may be used. Since multiple inheritance doesn't allow us to extend more than one class at a time, implementing the Runnable interface may help us in this situation.



Demo time – Thread creation

Demo - Thread Creation

Thread life cycle and states

NEW
A newly created thread that has not started the execution.

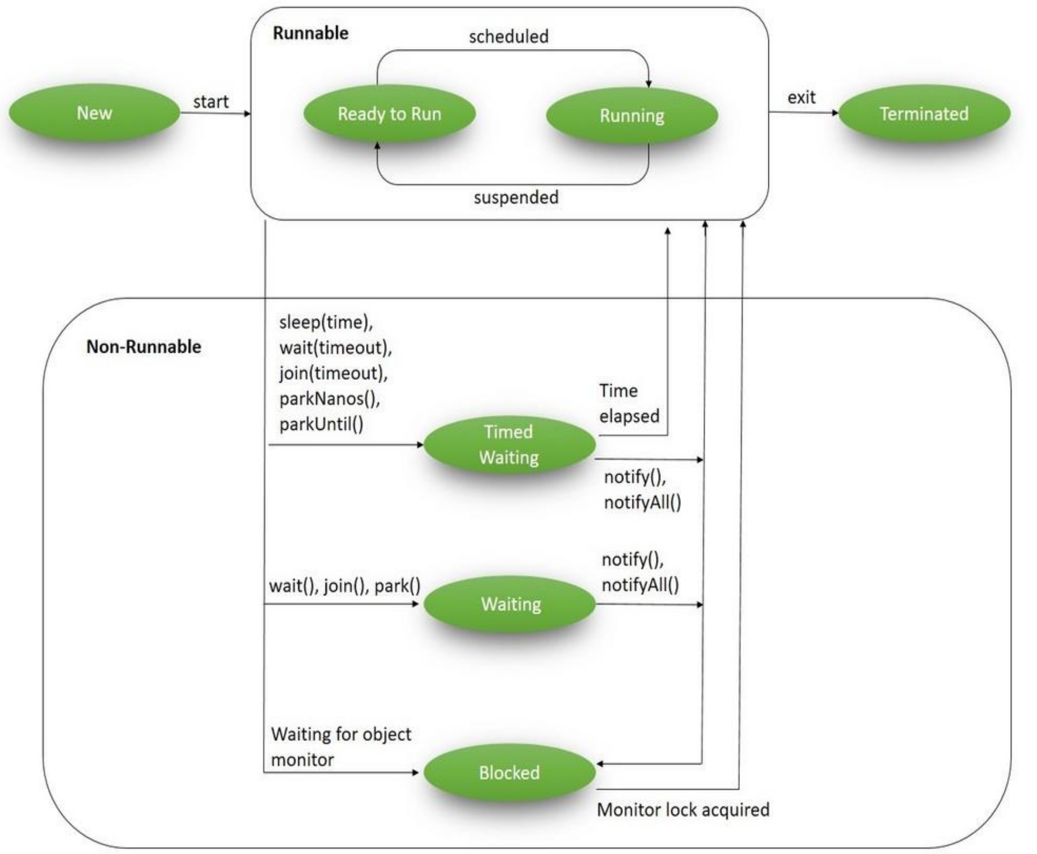
RUNNABLE
Either running or ready for execution but it's waiting for resource allocation

BLOCKED
Waiting to acquire a monitor lock to enter or re-enter a synchronized block/method

WAITING
Waiting for some other thread to perform a particular action without any time limit

TIMED_WAITING
Waiting for some other thread to perform a specific action for a specified period

TERMINATED
Has completed its execution

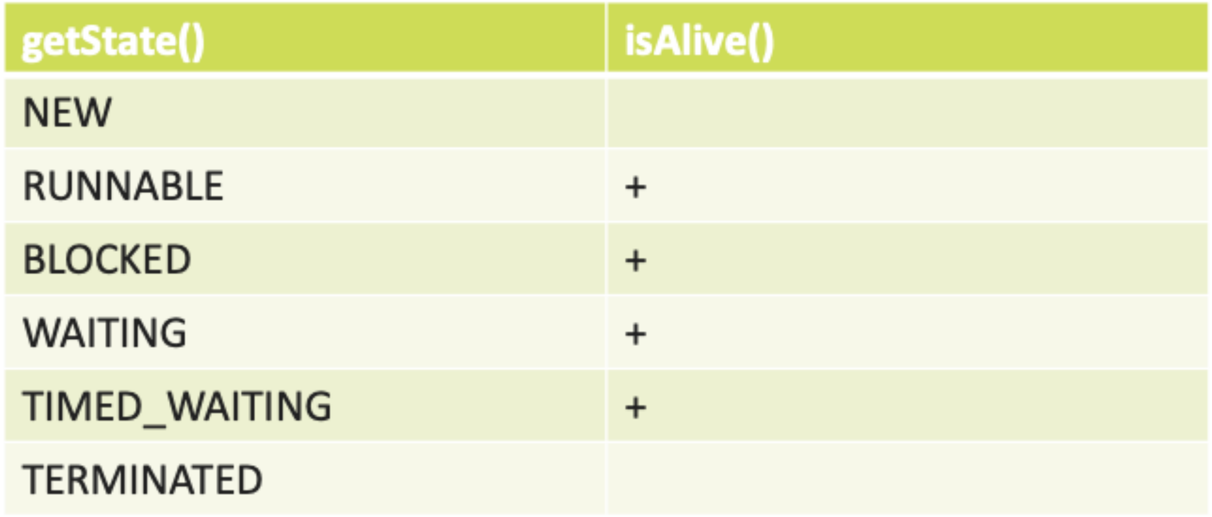


Threads methods

Method	Description
getName()/setName(String name)	Returns or sets thread name
getPriority()/setPriority(int priority)	Returns or sets thread priority
getState()	Returns the state of this thread
start()	Start a thread by calling run() method
run()	Entry point for a thread
join()	Current thread waits for another thread to finish
sleep(long millis)	Temporarily suspend thread for a specified time
interrupt()	Interrupts this thread

isAlive() and getState() relation

- isAlive() – returns true if the thread has been started (may not yet be running) but has not yet completed its run method
- getState() – returns the exact state of the thread.



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 your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

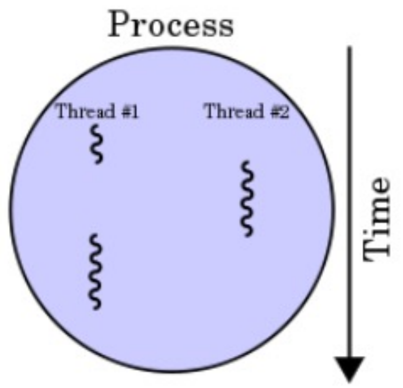
Demo time – thread states, main thread

Demo:

- Thread states
- Main thread

Thread priority

- OS Thread Scheduler uses thread’s priorities to determine which threads should get more CPU time. In general, threads with higher priority get more CPU time.
- Scheduling exampling on a single core system:



Java Thread Priority

- Java threads may have priorities from MIN_PRIORITY to MAX_PRIORITY. Currently these constants are set to 1 and 1-, respectively.
- The default thread priority is equal to NORMAL_PRIORITY, which is equal to 5;
- You can set a priority with setPriority(int level) or get it with getPriority()
- Priority of Main thread is set to NORM_PRIORITY
- The JVM supports a scheduling algorithm called **fixed-priority pre-emptive scheduling**. All Java threads have a priority, and the JVM serves the one with the highest priority first. The thread scheduler might choose low-priority threads for execution to avoid starvation.
- Context switching - a thread’s priority is used to decide when to switch from one running thread to the next.
    - A thread can voluntarily relinquish control(yielding, sleeping, blocked)
    - A thread can be preempted by a higher-priority thread.

Demo time – Thread priority

Thread Priority

Daemon threads

A low-priority thread that runs in the background to perform tasks such as garbage collection;

A service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads die, JVM terminates daemon thread automatically;

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

You can use setDaemon(boolean value) before thread starts;

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

Demo – Daemon threads

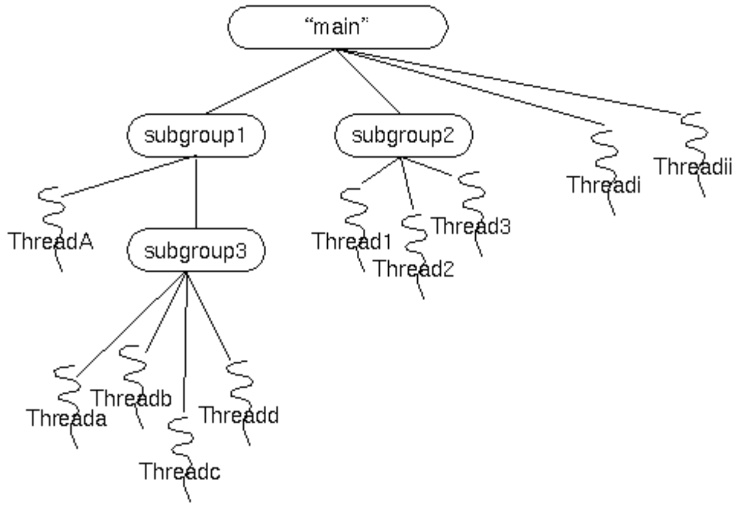
Daemon threads

Thread Groups

Thread groups provide 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 can be nested. They form a tree where each ThreadGroup except a root one, has a parent.

A thread is allowed to access information about its own thread group but not to access information about its thread group's parent thread group or any other thread group.



ThreadGroup methods

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() / deprecated in JDK 17	Returns whether group is daemon
setDaemon(boolean v) / deprecated in JDK 17	Sets the daemon flag
list()	Prints the info about all threads in group
destroy() / deprecated in JDK 17	Destroys a group and its subgroups. They must have no threads, illegalThreadStateException is thrown
setMaxPriority(int v)/ 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 a group with lower priority, its priority will become equal to thread group’s;
    - If a thread with a low priority is added to a group with higher priority, its priority will not change.

Demo – Thread Group

Demo time – Thread Group

Exception handling

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

- To handle exceptions, you can use:
    - Thread.setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler) – to set global default exception handler;
    - thread.setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler) – to set an exception handler for this thread.

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

Thread interruption

interrupt() – to interrrupt 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 – Exception handling

Demo time – Exception handling

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. Monitors are also called mutexes. In overall a monitor is a synchronization mechanism that allows threads to have:

- Mutual exclusion – only one thread can execute the method at a certain point of time, using locks;

- Cooperation – the ability to make threads wait for certain conditons to met, using wait-set

Synchronized keyword

- synchronized keyword is used 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 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 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 - Synchronization

Demo time - Synchronization

wait() and notify()

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

Methods	Effect
wait()	Causes the current thread to wait until is awakened, typically by being notified or interrrupted
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’ monitor

Demo time – Interthread communication

Demo - Interthread communication

Deadlock

- Deadlock is a special type errors, it occurs 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 than2 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.

Demo -Deadlock

Demo - Deadlock

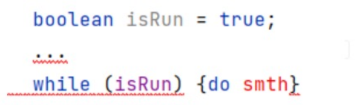
Suspending and resuming threads

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 - Volatile

Demo - Volatile

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.

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 time - Locks

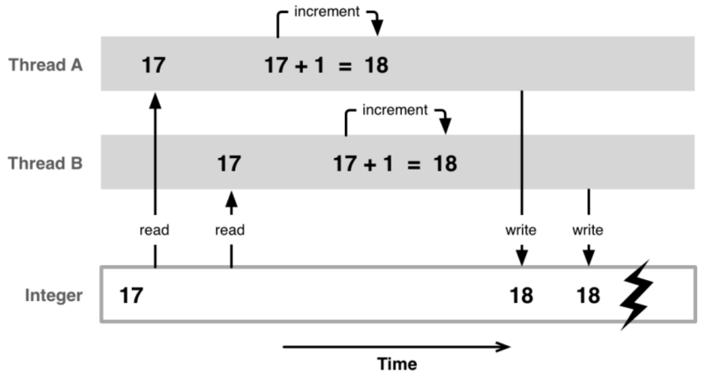
Demo - Locks

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 nonvolatile 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 condition
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 time - Atomic

Demo - Atomic

Thread safe collections

- Thread safe collections include:

    1. Lists
    2. Maps
    3. Sets
    4. Queues
    5. Deques

```
Collections.synchronizedXXX(collection)
```

```
List<String> safeList = Collections.synchronizedList(new ArrayList<>());
```

```
Map<Integer, String> unsafeMap = new HashMap<>();
Map<Integer, String> safeMap = Collections.synchronizedMap(unsafeMap);
```

- 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 are categorized into 3 groups based on their thread safety mechanisms:
    - Copy-on-write collections – CopyOnWriteArrayList, CopyOnWriteArraySet
    - CAS collection – ConcurrentLinkedQueue, ConcurrentSkipListMap
    - concurrent collections using a special lock object (java.util.concurrent.lock.Lock) – ConcurrentHashMap, LinkedBlockingQueue

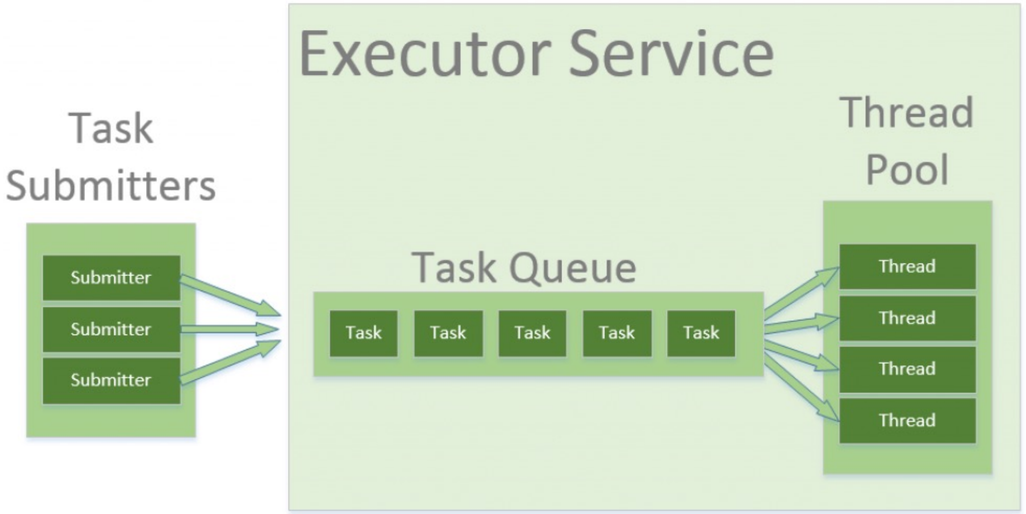
Demo time – thread safe collections

Demo – Thread Safe Collections

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 time - Executors

Demo – Executors, ExecutorService

Callable and Future

- Callable interface the following definition:

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

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

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 time – Callable, Future

Demo- Callable, Future

Thanks for your attention,

the main part of this presentation is over.

Fork-Join framework

- Fork-join framework – 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 resturns 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 time – Fork-Join pool

Demo – Fork-Join pool

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 seperation 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 dependant functions and actions that trigger upon its completion.
Phaser	A reusable synchronization barrier, provides more flexible usage than CountDownLatch