JAVA PROGRAMMING BASICS

Module 2: Java Object-oriented Programming

Training program

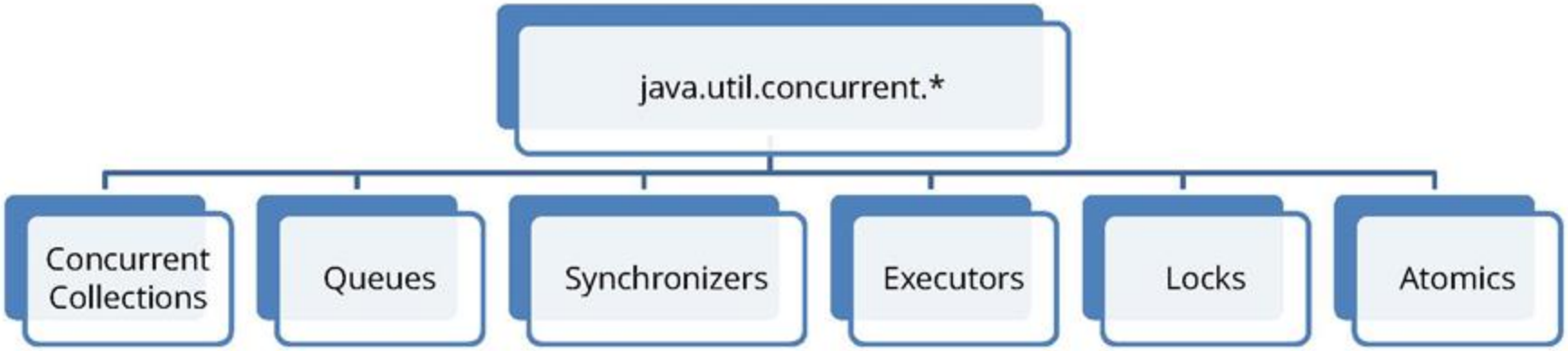
1. Classes and Instances
2. The Methods
3. The Constructors
4. Static Elements
5. Initialization sections
6. Package
7. Inheritance and Polymorphism
8. Abstract classes and Interfaces
9. String processing
10. Wrapper classes for primitive types
11. Exceptions and Assertions
12. Nested classes
13. Enums
14. Generics
15. Collections
16. Method overload resolution
17. **Multithreads**
18. Core Java classes
19. Object Oriented Design
20. Functional Programming

Module contents

- Introduction to Concurrent Programming
- Creating Threads
- Important Methods in the Thread class
- Thread interruption. The interrupt() method
- The States of a Thread
- The Thread Scheduler. Thread Priority
- The Daemon Threads
- Thread Synchronization
- Synchronized Methods
- Synchronized Blocks
- The Wait/Notify Mechanism
- The Volatile Keyword
- ThreadLocal class
- Deadlocks
- **Threads pool**
- **The ReentrantLock class**
- **Synchronizers**
- **Atomic Variables**
- **Concurrent Collection**
- **The Fork-Join Framework**

The java.util.concurrent packages

Since version 5.0, the Java platform has also included high-level concurrency APIs in the java.util.concurrent packages.



- **Concurrent Collections** work more efficiently in a multithreaded environment than the collections from the `java.util` package. Unlike the entire collection blocking while multithreaded access, locks on data segments are used and parallel reading of data optimized.

Fork/Join framework was added from Java 7

The java.util.concurrent packages

- **Queues** are non-blocking and blocking multithreading queues.
  Non-blocking queues ensure the speed of threads.
  Blocking queues are used when it is necessary to "slow down" the supplier or consumer thread, if some conditions are not met.
- **Synchronizers** are helper utilities for synchronizing threads, allowing the developer to control and / or restrict the work of multiple threads.
- **Executors** are tools for creating thread pools and scheduling asynchronous tasks to get results.
- **Locks** are alternative and more flexible thread synchronization mechanisms compared to the basic `synchronized, wait, notify, notifyAll`.
- **Atomics** - classes with support for atomic operations on primitives and references.
- When writing multithreaded programs in practice, it is often better to use the Concurrency API rather than work with Thread objects directly. The API are much more robust, and it is easier to handle complex interactions.

Java Concurrent API utils: TimeUnit enum

- java.util.concurrent.TimeUnit enum provides time representation at a given unit of granularity. It supports nanoseconds, microseconds, milliseconds, seconds, minutes, hours, and days units.
- It makes available methods to convert time across time units.
- It has methods **sleep**, **timedWait** and **timedJoin** for convenient use Thread's **sleep** and **join** and Object's **wait** methods with time units.
- It is used to perform timing and delay operations.

see timeunit\TimeUnitTask & Main

since Java 5

TimeUnit enum methods

Methods	Description
long **convert**(long sourceDuration, TimeUnit sourceUnit)	Converts the time duration inputted with its unit to the required unit.
long **toTIMEUNITNAMEs** (long duration)	Converts the time duration to TimeUnit **(toDays**, **toHours**, ...).
static TimeUnit **valueOf**(String name)	Returns the enum constant of the type with the specified name.
static TimeUnit [] **values**()	Returns an array containing the enum constants.
void **sleep**(long timeout)	Performs a Thread.sleep using this time unit. Pause for given TimeUnit.
void **timedWait**(Object obj, long timeout)	Performs a timed Object.wait using this time unit. Wait for the given time unit to execute.
void **timedJoin**(Thread thread, long timeout)	Performs a timed Thread.join using this time unit. Thread is provided to do work for a given time duration only.

Java Concurrent API utils: ThreadLocalRandom

- The java.util.Random class doesn't perform well in a multi-threaded environment due to contention – given that multiple threads share the same Random instance.
- Java introduced the java.util.concurrent. ThreadLocalRandom class for generating random numbers in a multi-threaded environment.
- ThreadLocalRandom is a combination of the ThreadLocal and Random classes and is isolated to the current thread.

```
int r1 = ThreadLocalRandom.current().nextInt(10);
int r2 = ThreadLocalRandom.current().nextInt(2, 6);
```

since Java 7

Java Concurrent API utils: ThreadLocalRandom

Modifier and Type	Method and Description
static ThreadLocalRandom **current**()	Returns the current thread's ThreadLocalRandom.
DoubleStream **doubles**() IntStream **ints**() LongStream **longs**() and for double/int/long: **doubles**(long streamSize) **doubles**(double randomNumberOrigin, double randomNumberBound) **doubles**(long streamSize, double randomNumberOrigin, double randomNumberBound)	Returns an effectively unlimited stream of pseudorandom double/int/long values, each between zero (inclusive) and one (exclusive) or between zero (inclusive) and one (exclusive) streamSize elements or between given origin (inclusive) and bound (exclusive) or between given origin (inclusive) and bound (exclusive) streamSize elements.
protected int **next**(int bits)	Generates the next pseudorandom number.
boolean **nextBoolean**() int **nextInt**() long **nextLong**() float **nextFloat**() double **nextDouble**()	Returns a pseudorandom boolean/int/long/float/double value between zero (inclusive) and one (exclusive).

Java Concurrent API utils: ThreadLocalRandom

Modifier and Type	Method and Description
double **nextDouble**(double bound) int **nextInt**(int bound) long **nextLong**(long bound) and for double/int/long: double **nextDouble**(double origin, double bound)	Returns a pseudorandom double/int/long value between zero (inclusive) and the specified bound (exclusive). or between given origin (inclusive) and bound (exclusive)
float **nextFloat**(float bound)	Returns a pseudorandom float value between zero (inclusive) and one (exclusive).
double **nextGaussian**()	Returns the next pseudorandom, Gaussian ("normally") distributed double value with mean 0.0 and standard deviation 1.0 from this random number generator's sequence.
boolean **nextBoolean**(boolean bound)	Returns a pseudorandom boolean value.
void **setSeed**(long seed)	Throws UnsupportedOperationException.

Java Concurrent API utils: ThreadLocalRandom

Random	ThreadLocalRandom
If different threads use the same instance of Random it results in contention and consequent performance degradation.	There is no contention because the random numbers generated are local to the current thread.
Uses Linear Congruential Formula to modify its seed value.	The Random generator is initialized using an internally generated seed.
Useful in applications where each thread has its own set of Random instances to use.	Useful in applications where multiple threads use random numbers in parallel in thread pools.
This is the Parent class.	This is the Child class.

see threadlocalrandom\RandomVsThreadLocalRandom

Module contents

- Introduction to Concurrent Programming
- Creating Threads
- Important Methods of the Thread class
- Thread interruption. The interrupt() method
- The States of a Thread
- The Thread Scheduler. Thread Priority
- The Daemon Threads
- Thread Synchronization
- Synchronized Methods
- Synchronized Blocks
- The Wait/Notify Mechanism
- The Volatile Keyword
- ThreadLocal class
- Deadlocks
- **Threads pool**
- The ReentrantLock class
- Synchronizers
- Atomic Variables
- Concurrent Collection
- The Fork-Join Framework

Threads pool 1/12

Thread per task is the bad way:

- creating a new thread for each request can onsume significant computing resources
- having many threads competing for the CPUs can impose other performance costs as well.

The Executor Framework provide a high-level approach to launching tasks and managing threads.

- internally, an executor maintains a thread pool that utilizes a number of reusable threads that are assigned tasks from a task queue, according to the execution policy of the executor.

Threads pool 2/12

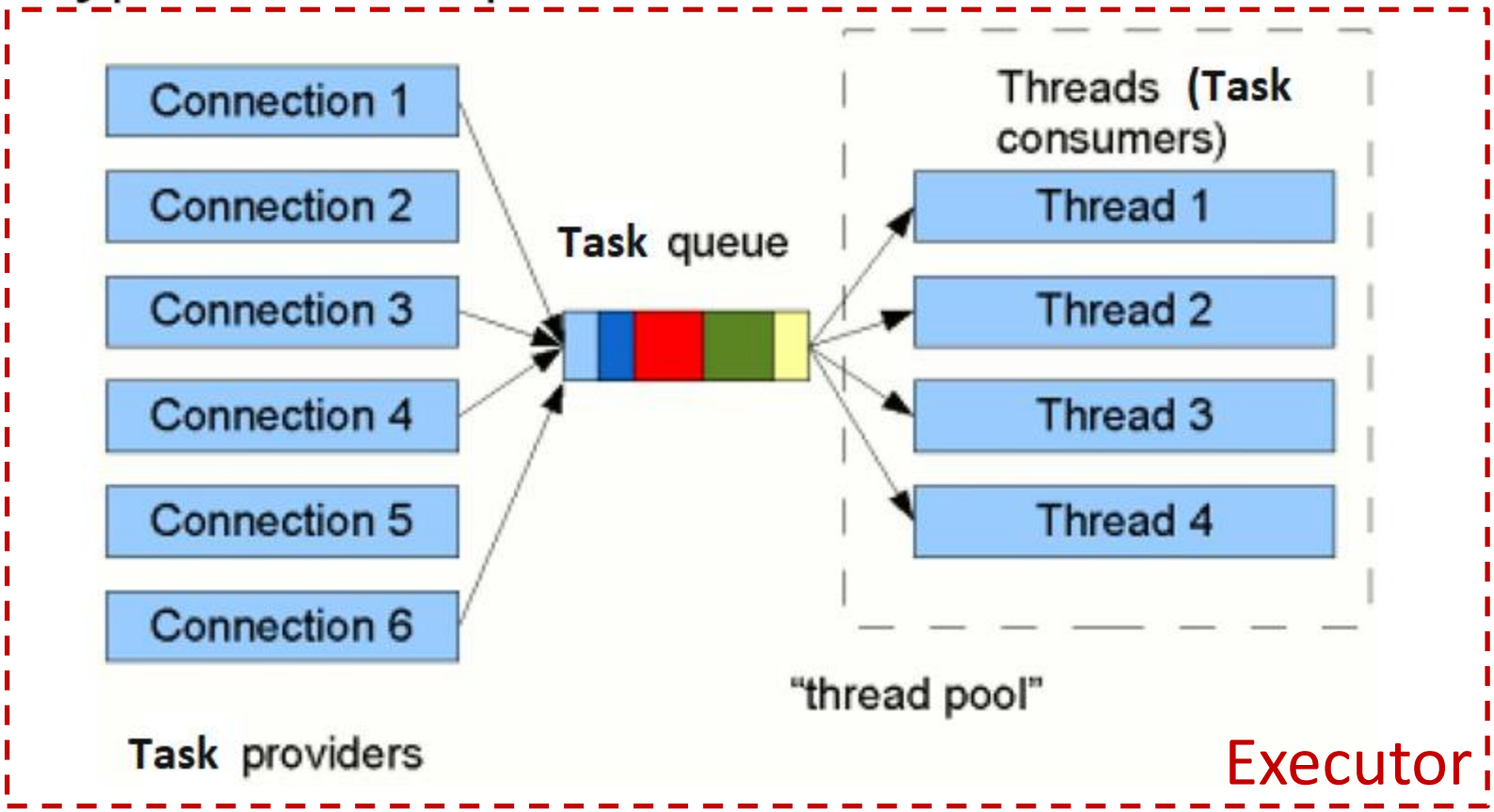
Reasons for using thread pools:

- gaining some performance when the threads are reused.
- better program design, letting you focus on the logic of your program.

- executors allow the submission of tasks for execution to be decoupled from the actual execution of tasks.
- executor provides the necessary support for managing its lifecycle: creating the executor, submitting tasks, managing the outcome of task execution, and shutting down the executor.

Threads pool 3/12

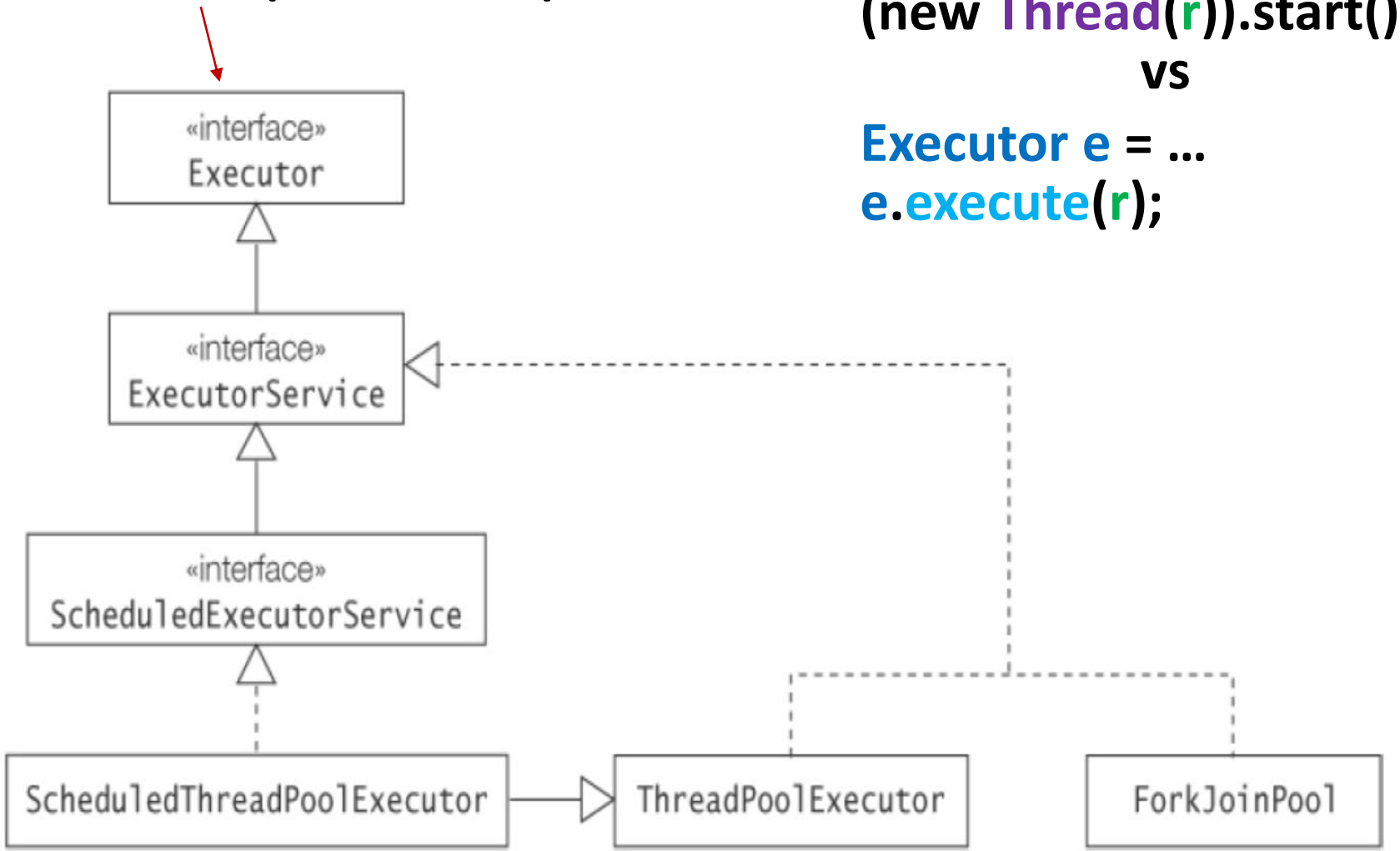
- Typical thread pool architecture



Thread pool 4/12

void execute (Runnable r)

```
Runnable r = …
(new Thread(r)).start();
        vs
Executor e = …
e.execute(r);
```



java.util.concurrent

Thread pool 6/12

```
Executor tpe = new ThreadPoolExecutor(int corePoolSize,
                                      int maximumPoolSize,
                                      long keepAliveTime,
                                      TimeUnit unit,
                                      BlockingQueue<Runnable> workQueue)
```

- When a new task is passed to the void ***execute(Runnable command)*** method, and at that time the number of threads in the pool is less than **corePoolSize**, a new thread is created for that task, even if other threads are idle.
- If the number of threads in the pool is greater than **corePoolSize** but less than **maximumPoolSize**, the task is put in the **workQueue** and given to the freed thread. A new thread will only be created if the **workQueue** is full.
- If there are more than **corePoolSize** threads in the pool, the excess threads will be killed if they are idle for more than **KeepAliveTime.** see threadpoolexecutor\MyTask & ThreadPoolMain

Threads pool 9/12

The **Executors** factory class is used to create instances of the ExecutorService object:

- **Executors.newSingleThreadExecutor()**: creates a single background thread **A**=1 **B**=1
- **Executors.newFixedThreadPool**(int numThreads): creates a fixed size thread pool **A**=numThreads **B**=numThreads
- **Executors.newCachedThreadPool()**: create a unbounded thread pool, with automatic thread reclamation **A**=0 **B**=Integer.MAX_VALUE time = 60 seconds

```
{  return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(A, B,
         0L, TimeUnit.MILLISECONDS,
         new LinkedBlockingQueue<Runnable>()));
```

Threads pool 10/12

public interface ExecutorService **extends** Executor, AutoCloseable

The basic ExecutorService methods:

Future<?> **submit**(Runnable task)

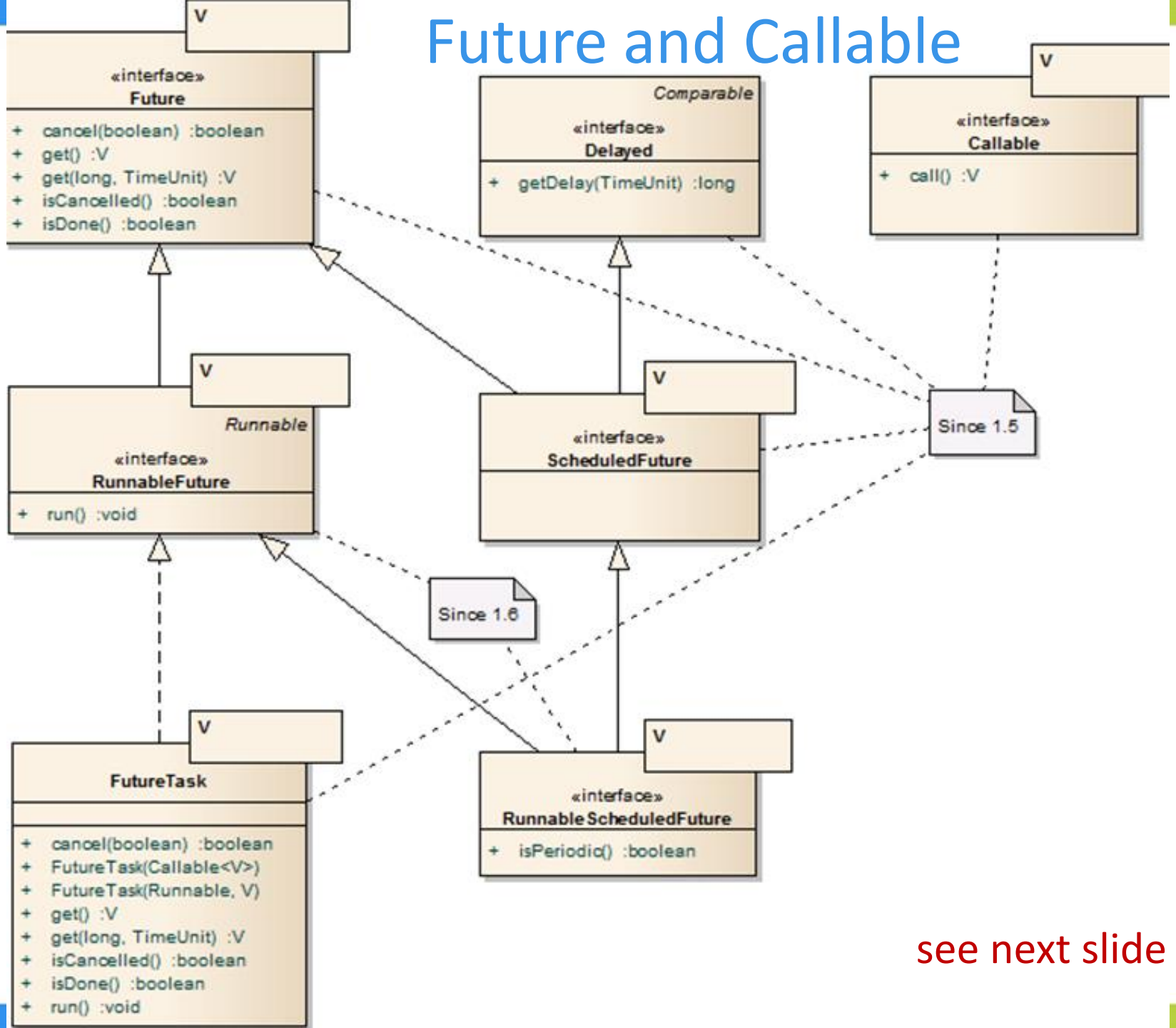
Future<T> **submit**(Callable<T> task)

Future<T> **submit**(Runnable task, T result)

Returns a Future object that can be used to track the progress of the task. public interface Future<V>

- The Executor's method void execute(Runnable command) is a “fire-and-forget” method
- So we have to prefer ExecutorService's Future<?> submit(Runnable task) method (Future<Void> - for Runnable task

Future and Callable



see next slide

Future and Callable

- `Callable<V>` is an analogue of the `Runnable` interface for tasks that return the value obtained in the thread. `Callable<V>` has single method `V call()`, which returns a `V` type value.
- Callable tasks, like `Runnable` tasks, can be passed to `submit` method of `ExecutorService`. Since the `submit` method does not wait for the task to complete, the `ExecutorService` object cannot return the task result directly. Instead, it returns a special `Future<V>` object with the result of the task execution (such execution is called **asynchronous**).
- The basic methods of `Future<V>` interface are `V get()` and `V get(long, TimeUnit)`, which wait for the task to complete and return the result value of the completed task.

```
package java.util.concurrent;

@FunctionalInterface
public interface Callable<V> {

   V call() throws Exception;
}
```

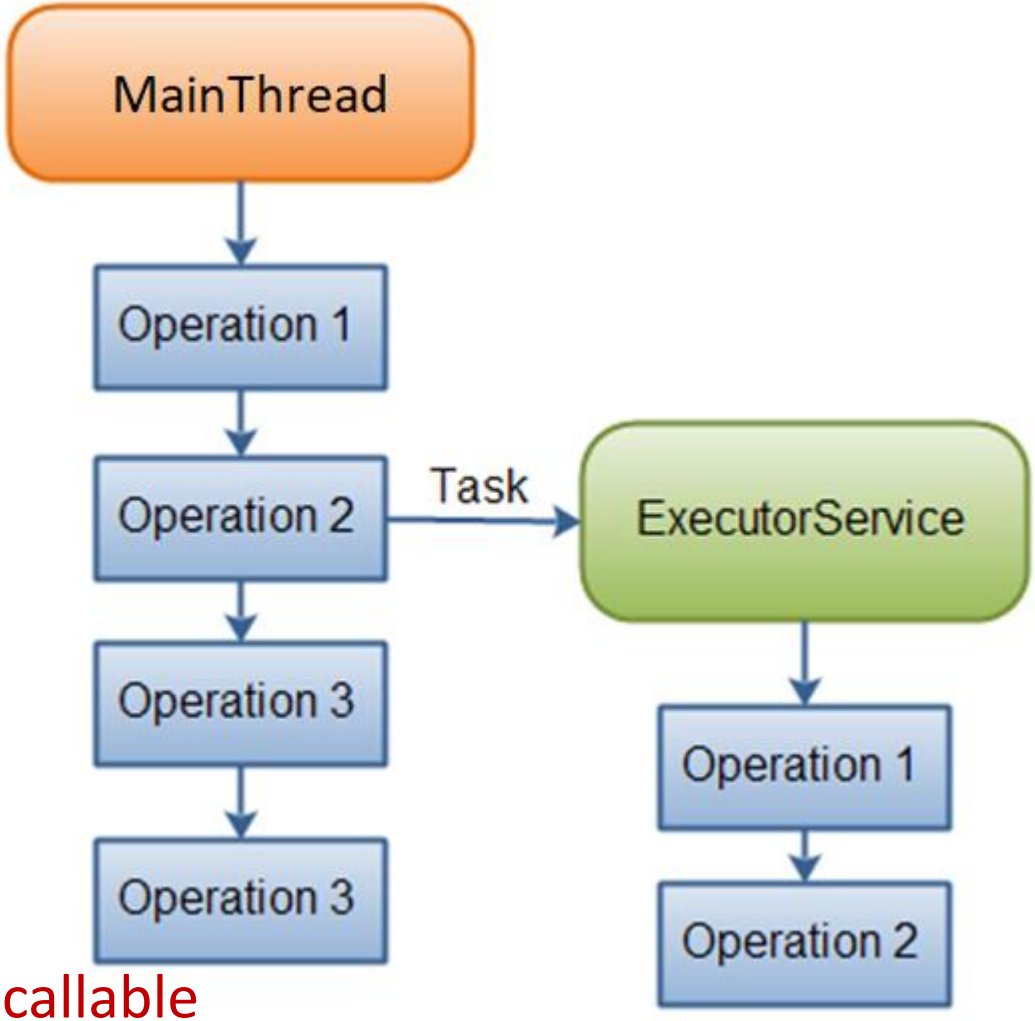
Future methods

Method name	Description
`boolean` **`isDone`**`()`	Returns `true` if task was completed, threw exception, or was cancelled.
`boolean` **`isCancelled`**`()`	Returns `true` if task was cancelled before it completed normally.
`boolean` **`cancel`**`(boolean mayInterruptIfRunning)`	Attempts to cancel execution of task and returns `true` if it was successfully cancelled or `false` if it could not be cancelled or is complete.
`V` **`get`**`()`	Retrieves result of task, waiting endlessly if it is not yet available.
`V` **`get`**`(long timeout, TimeUnit unit)`	Retrieves result of task, waiting specified amount of time. If result is not ready by time timeout is reached, checked `TimeoutException` will be thrown.

see callable\MyCallable & CallableMain

Task asynchronous execution

The ExecutorService then executes the task concurrently, independently of the thread that submitted the task.



see runnablevscallable

Task asynchronous execution

The ExecutorService has methods for Collection of task asynchronous execution:

```
<T> T invokeAny(Collection<? extends Callable<T>> tasks)
                    throws InterruptedException, ExecutionException;
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>>
                                    tasks) throws InterruptedException;
```

see executorservice\Result&CallableTask&TaskColectionMain

ExecutorService Interface

Returns	Method	Description
boolean	awaitTermination(long time out, TimeUnit unit)	Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
<T> List<Future <T>>	invokeAll(Collection<? extends Callable<T>> tasks)	Executes the given tasks, returning a list of Futures holding their status and results when all complete.
<T> List<Future <T>>	invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)	Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires, whichever happens first.
<T> T	invokeAny(Collection<? extends Callable<T>> tasks)	Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do.
<T> T	invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)	Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do before the given timeout elapses.

ExecutorService Interface

Returns	Method	Description
boolean	isShutdown()	Returns true if this executor has been shut down.
boolean	isTerminated()	Returns true if all tasks have completed following shut down.
void	shutdown()	Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
List<Runnable>	shutdownNow()	Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.
Future<?>	submit(Runnable task)	Submits a Runnable task for execution and returns a Future representing that task.
<T> Future<T>	submit(Runnable task, T result)	Submits a Runnable task for execution and returns a Future representing that task.
<T> Future<T>	submit(Callable<T> task)	Submits a value-returning task for execution and returns a Future representing the pending results of the task.

ScheduledExecutorService Interface

Modifier and Type	Method	Description
ScheduledFuture<?>	schedule(Runnable command, long delay, TimeUnit unit)	Submits a one-shot task that becomes enabled after the given delay.
<V> ScheduledFuture<V>	schedule(Callable<V> callable, long delay, TimeUnit unit)	Submits a value-returning one-shot task that becomes enabled after the given delay.
ScheduledFuture<?>	scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)	Submits a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period; that is, executions will commence after initialDelay, then initialDelay + period, then initialDelay + 2 * period, and so on.
ScheduledFuture<?>	scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)	Submits a periodic action that becomes enabled first after the given initial delay, and subsequently with the given delay between the termination of one execution and the commencement of the next.

see executorservice\Result&RunnableTask&ScheduledMain

Module contents

- Introduction to Concurrent Programming
- Creating Threads
- Important Methods of the Thread class
- Thread interruption. The interrupt() method
- The States of a Thread
- The Thread Scheduler. Thread Priority
- The Daemon Threads
- Thread Synchronization
- Synchronized Methods
- Synchronized Blocks
- The Wait/Notify Mechanism
- The Volatile Keyword
- ThreadLocal class
- Deadlocks
- Threads pool
- **The ReentrantLock class**
- Synchronizers
- Atomic Variables
- Concurrent Collection
- The Fork-Join Framework

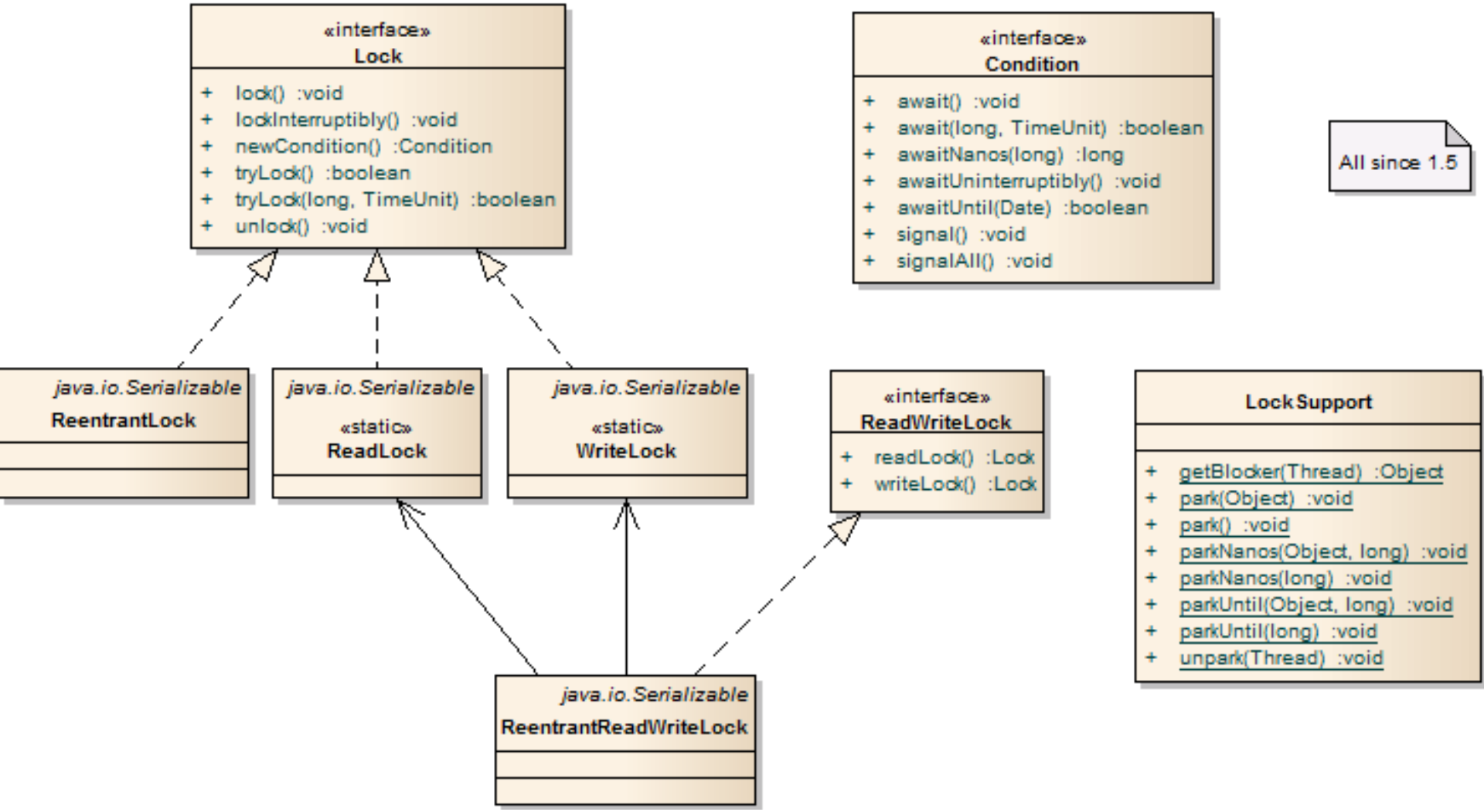
Locks

Synchronization with `synchronized` has some limitations:

1) You have to acquire and release the monitor in the same block of code (method or code block);
2) a thread can wait for access to a critical section *indefinitely* and we can not influence this;
3) it is impossible to interrupt a thread waiting for a lock;
4) it is impossible to poll or try to get a lock without being prepared for a long wait;
5) the *fairness* of lock (granting it to threads in the order they asked for it) is not supported;
6) we can not separate the read and write operations of critical section.

`The java.util.concurrent.locks` package has interfaces and classes that provide much more flexible locking and waiting conditions other than the built-in `wait/notify/notifyAll` and monitor synchronization mechanism and can prevent threads from potential resource starvation.

Locks



A thread needs to acquire a *Lock implementation instance* in order to gain exclusive access to a critical section guarded by the instance. Afterward, the instance must be *unlocked*, so that other threads can try to gain access to the critical section.

Locks

- **Lock** is a basic interface that provides a more flexible locking than `synchronized`. While using multiple Lock objects, the order in which they are released can be arbitrary. It is also possible to use an alternative scenario if the Lock object has already been captured (tryLock()).
- **Condition** is an interface that describes condition variables that can be associated with Lock objects. They are similar in use to the `Object.wait`, but offer enhanced functionality.
  In particular, multiple Condition objects can be associated with a single Lock object. To avoid compatibility issues, Condition method names differ from their respective versions of Object: it contains `await / signal / signalAll` methods alternative to the standard `wait / notify / notifyAll`.
- **ReentrantLock** Only one thread can enter a protected block. Such a thread can invoke, directly or indirectly, other synchronized methods of the object without being blocked—that is it can acquire the lock on the same object several times (reentrant synchronization). The class supports fair and non-fair blocking of threads.

Locks

- **ReadWriteLock** - Additional interface for creating read/write locks. Such locks are extremely useful when the system has a lot of reads and few writes.
- **ReentrantReadWriteLock** - Very often used in multithreaded services and caches, showing very good performance gains over synchronized blocks. In fact, the class operates in two mutually exclusive modes: many readers read data in parallel and when only one writer writes data.
- **ReentrantReadWriteLock.ReadLock** - Read lock for readers, obtained via readWriteLock.readLock().
- **ReentrantReadWriteLock.WriteLock** - Write lock for writers, obtained via readWriteLock.writeLock ().
- **LockSupport** - Designed for building classes with locks. Contains methods for parking threads instead of the deprecated Thread.suspend() and Thread.resume() methods.

The ReentrantLock class 2/5

- Synchronized keyword doesn't support fairness
- A thread can be blocked waiting for lock, for an indefinite period of time and there was no way to control that.

- A reentrant lock supports fair thread blocking.
- A reentrant lock supports polled locking when the thread does not acquired lock can continue another work.

see synchronizedmeth vs reentrantlock

The ReentrantLock class 3/5

Lock interface provide opportunity to acquire a lock by different ways:

- **void lock()** - acquire the lock if it's available. If the lock isn't available, a thread gets blocked until the lock is released by **void unlock()** method.
- **boolean tryLock()** - attempts to acquire the lock immediately (if done, returns true), if the lock is not available returns false and the thread continues to run.
- **boolean tryLock(long timeout, TimeUnit unit)** - this is similar to tryLock(), but if the lock is not available, the thread waits for a time and try to acquire the lock before exiting (returns false after time elapsed).
- **void lockInterruptibly()** - acquire the lock if it's available until this lock is released or the thread is interrupted by other thread.

see trylock & lockinterruptibly

Thread coordination with Condition

- **Condition** is an interface that describes condition variables that can be associated with Lock objects. They are similar in use to the `Object.wait`, but offer enhanced functionality. In particular, multiple Condition objects can be associated with a single Lock object. To avoid compatibility issues, Condition method names differ from their respective versions of Object: it contains `await / signal / signalAll` methods alternative to the standard `wait / notify / notifyAll`.

```
public interface Condition {
    void await() throws InterruptedException;
    void awaitUninterruptibly();
    long awaitNanos(long nanosTimeout) throws InterruptedException;
    boolean await(long time, TimeUnit unit)
                                        throws InterruptedException;
    boolean awaitUntil(Date deadline) throws InterruptedException;
    void signal();
    void signalAll();                          see condition
}
```

Reentrance - nested methods

- **ReentrantLock** is a concrete implementation of Lock interface provided in Java concurrency package from Java 5 onwards
- Thread can acquire the same lock multiple times without any issue.
- Reentrant locking increments special thread-personal counter (unlocking - decrements) and the lock will be released only when counter reaches zero.

This is similar to the semantics of nested synchronized.

see reentrancy

Locks

- **ReadWriteLock** - Additional interface for creating read/write locks. Such locks are extremely useful when the system has a lot of reads and few writes.

```
public interface ReadWriteLock {
    /**
     * Returns the lock used for reading.
     */
    Lock readLock();

    /**
     * Returns the lock used for writing.
     */
    Lock writeLock();
}
```

see readwritelock

Module contents

- Introduction to Concurrent Programming
- Creating Threads
- Important Methods of the Thread class
- Thread interruption. The interrupt() method
- The States of a Thread
- The Thread Scheduler. Thread Priority
- The Daemon Threads
- Thread Synchronization
- Synchronized Methods
- Synchronized Blocks
- The Wait/Notify Mechanism
- The Volatile Keyword
- ThreadLocal class
- Deadlocks
- Threads pool
- The ReentrantLock class
- **Synchronizers**
- Atomic Variables
- Concurrent Collection
- The Fork-Join Framework

Synchronizers

In addition to the objects-implementations of the Lock interface, the Java Concurrent API offers high-level synchronizers:

- **Semaphore** - an object that controls access to one or more shared resources by threads using an access permissions counter;
- **CountDownLatch** - an object that blocks all threads until a certain condition is met, when the condition is met, it releases all threads at the same time (waiting multiple concurrent events);
- **FutureTask** - an object that describes an abstract computation, that produces an asynchronously retrieved result.
- **Cyclic Barrier** - an object that allows threads to be synchronized at a common point at which a specified number of threads meet and block. Once all threads have arrived at the point, the optional operation is performed and the barrier is broken and the waiting threads are released (synchronizing tasks in a common point);
- **Exchanger** - an object designed to exchange data between threads at a certain point of both threads operations;
- **Phaser** - an object that allows to control execution of concurrent tasks divided in phases. All the threads must finish one phase before they can continue with the next one (since Java 7).

Semaphore 1/10

- Conceptually, a semaphore maintains a set of permits.
- Each acquire() blocks if necessary until a permit is available, and then takes it.
- Each release() adds a permit, potentially releasing a blocking acquirer.

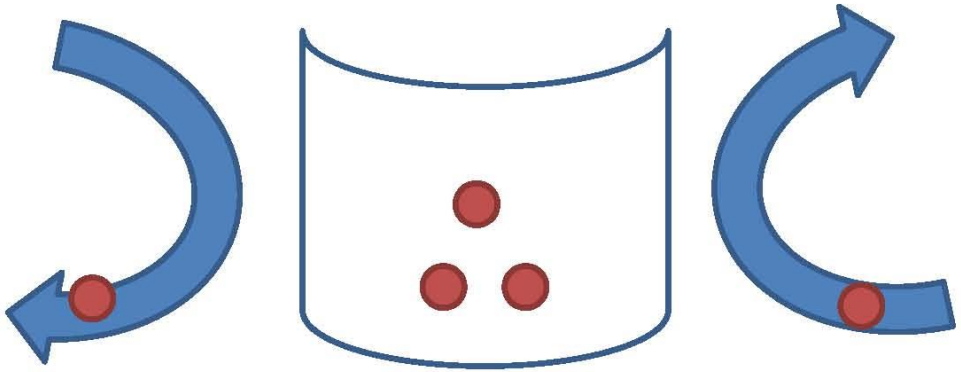
```
Semaphore semaphore = new Semaphore(3);
```

permits

Semaphore 2/10

- A semaphore is an object with a counter that counts the amount of free resources.

Semaphore s = new Semaphore (3)



s.acquire()

s.release()

fair

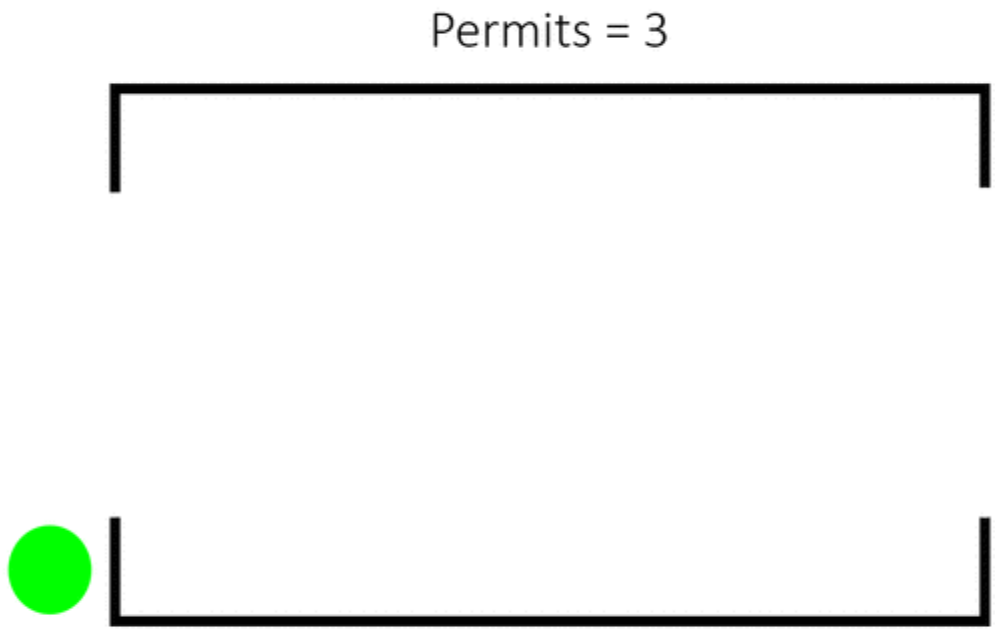
Semaphore sem = new Semaphore(3, true);

Semaphore

Method	Action performed
void **acquire**()	Acquires a permit, if one is available and returns immediately, reducing the number of available permits by one .If the current thread is interrupted while waiting for a permit then InterruptedException is thrown.
void **acquireUninterruptibly()**	Acquires a permit if one is available and returns immediately, reducing the number of available permits by one. If the current thread is interrupted while waiting for a permit then it will continue to wait.
boolean **tryAcquire()**	Acquires a permit, if one is available and returns immediately, with the value true, reducing the number of available permits by one. If no permit is available then this method will return immediately with the value false.
boolean **tryAcquire(long timeout, TimeUnit unit)**	Acquires a permit if one is available and returns immediately, with the value true, reducing the number of available permits by one. If the specified waiting time elapses then the value false is returned. If the time is less than or equal to zero, the method will not wait at all.
void **release()**	Releases a permit, returning it to the semaphore.

Semaphore

Semaphore



Semaphore s = new Semaphore(3, true);

see semaphore\acquire & tryacquire

CountDownLatch

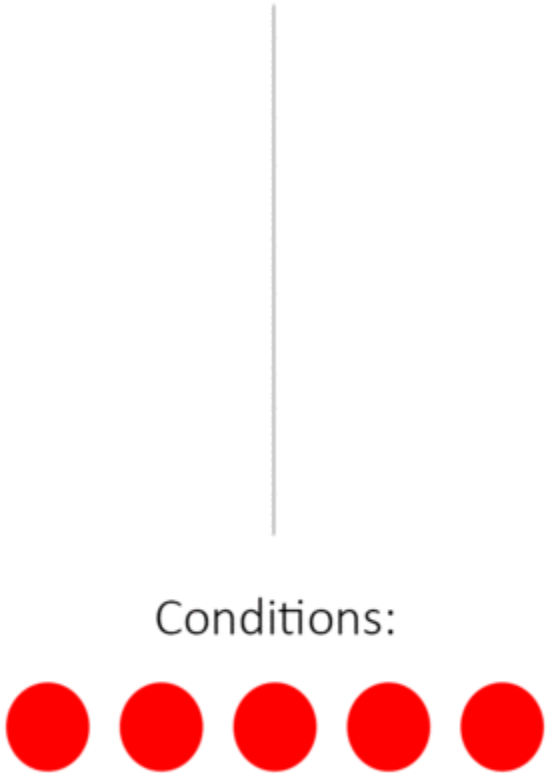
- **CountDownLatch** - an object that delays all threads until a certain condition is met, when the condition is met, it releases all threads at the same time
- The CountDownLatch object is created by the constructor with the parameter initializes the latch counter.
- The **void await()** method of the CountDownLatch class causes the thread on which it is invoked to wait until the latch count is reduced to zero or the thread is interrupted by the interrupt() method invoke from it.
- The **void countDown()** method of the CountDownLatch class decrements the CountDownLatch counter by 1, releasing together all threads waiting for the counter to reach 0. The number of **countDown()** calls that raises the latch, independent of the number of threads involved.

CountDownLatch 1/5

```
CountDownLatch counter = new CountDownLatch(5);
counter.await();
```

count = 5

```
counter.countDown();
```



CountDownLatch

- **CountDownLatch** class has
  public boolean await(**long** timeout, TimeUnit unit)
  throws InterruptedException method that cause the current thread to wait until one of the following events occurs:
  - the latch count reaches 0,
  - the thread is interrupted,
  - any specified waiting time has elapsed.
- The **CountDownLatch** instance can only go up once, and therefore cannot be reused.
- The **CountDownLatch** mechanism is not used to protect a shared resource or a critical section. It is used to synchronize one or more threads with the execution of various tasks.

see countdownlatch

FutureTask

- **FutureTask** acts like a latch: it implements a **RunnableFuture** (descendant of **Future** interface), which describes an abstract computation that produces a result, which is implemented using the **Callable** interface.
- This computation can be in a state of execution awaiting, execution itself, or completion (normal completion, cancellation, or interruption). **FutureTask**, having accepted the completed state, remains in it forever.
- The behavior of the **Future.get** method depends on the state of the task. If it is completed, the method will return the result immediately. Otherwise, it will block progress until the task reaches a completed state,
- and then return a result or throw an exception.

FutureTask

- **FutureTask** will transfer the result from the thread performing the calculation to the thread retrieving the result. The **FutureTask** specification guarantees the secure publication of the result through such a transfer.
- **FutureTask** is used by the **Executor** task framework to represent asynchronous tasks and potentially time-consuming computations that run before the results are needed.

see futuretask

CyclicBarrier

- **CyclicBarrier** is a synchronization point where a specified in constructor number of parallel threads meet and block.
- Once all threads have arrived, all threads unblocked and continue and a Runnable **barrierAction** is started (if the barrier was initialized with it).
- After all threads release **CyclicBarrier** is auto-reset so it can be used again.
- Barriers are similar to latches in that they block a group of threads until some event occurs. But unlike a latch, a barrier forces threads to pass the barrier point together at the same time to continue working. Latches are for waiting for events, and barriers are for waiting for other threads. Also latch is not autoreset like barrier.

CyclicBarrier

CyclicBarrier barrier = new CyclicBarrier(3, new BarrierAction());

parties = 3

CyclicBarrier

- The **int await()** method of the CyclicBarrier object indicates to the thread from which the method was called that it came to the barrier (it returns the thread arrival index N-1 for the first thread to arrive and 0 for the last thread to arrive). This thread is put on WAITING state until all other threads reach the barrier.
- The **int await (long timeout, TimeUnit unit)** method of the CyclicBarrier object puts the current thread on WAITING state until all other threads reach the barrier or parameter timeout elapsed.

see cyclicbarrier

CyclicBarrier

- The CyclicBarrier object can be reset to its initial state bythe **reset()** method of the CyclicBarrier class. When this occurs, all the threads that were waiting in the **await()** method receive a **BrokenBarrierException** exception.
- A CyclicBarrier object can be in a special state denoted by broken. When there are various threads waiting in the **await()** method and one of them is interrupted, this thread receives an **InterruptedException** exception, but the other threads that were waiting receive a **BrokenBarrierException** exception and CyclicBarrier
- is placed in the broken state.
- The CyclicBarrier class provides the **isBroken()** method, then returns *true* if the object is in the broken state; otherwise it returns *false*.

see cyclicbarrier with thread interrupt

Exchanger

- **Exchanger<V>** is a two-party barrier in which the parties exchange data at the barrier point.
- The exchanger is the point of synchronization of a pair of threads: the thread that calls the exchanger method **V exchange(V x)** is blocked and waiting for another thread. When another thread calls the same method (say come to exchange point), objects will be exchanged: each will receive an argument of the **V exchange(V x)** method called in another thread.
- The Exchanger class has another version of the exchange method: **exchange(V x, long time, TimeUnit unit)**. The thread will be sleeping until it's interrupted, the other thread arrives, or the specified timeout elapsed.

Exchanger

```
Exchanger<String> exchanger = new Exchanger<>();
```

T

exchanger.exchange(“Some string”);

see exchanger

Phaser

- **Phaser** is a reusable synchronization barrier, similar in functionality to CyclicBarrier and CountDownLatch but supporting more flexible usage.
- **Phaser** allows to synchronize threads that represent a single phase or stage of a common action.
- **Phaser** determines the synchronization object that waits until a certain phase is completed. Phaser then moves on to the next phase and waits for it to complete again.
- **Phaser** provides us with the mechanism to synchronize the threads at the end of each phase, so no thread starts its second phase until all the threads have finished the first one.
- Each phase can have a different number of threads waiting for advancing to another phase.

Since JDK 7

Phaser

- To create a Phaser object, you use one of the constructors:

```
Phaser()
Phaser(int parties)
Phaser(Phaser parent)
Phaser(Phaser parent, int parties)
```

`parties` - the number of parties (threads) each of it performs some phase of common action, that registered by the constructor (at phase 0).

`parent` - the parent Phaser object.

- To participate in the coordination, the thread needs to ***register*** itself with the *Phaser* instance.

Phaser

Basic methods of the Phaser class:

- **int register()** - registers a new party that performs phases, and returns the number of the current phase (usually phase 0);
- **int arrive()** - reports that the party has completed the phase and returns the number of the current phase, when calling this method, the thread does not stop, but continues to run;
- **int arriveAndAwaitAdvance()** is similar to the arrive() method, except that it causes the Phaser object to wait for all other parties to complete the phase;

Phaser

- **int awaitAdvance(int phase)** - if parameter phase is equal to the current phase number, suspends the thread in which this method is called until the end of the current phase. Returns the number of the next phase, or an argument if it is negative, or a (negative) current phase if it is complete;
- **int arriveAndDeregister()** - notifies the completion of all phases by the party and removes it from registration. Returns the current phase number or a negative number if the Phaser synchronizer has shut down;
- **int getPhase()** - returns the current phase number.

Phaser

- When working with the Phaser class, its object is usually created first. Next, you need to register all the parties (threads) involved in the implementation of the phases by **register()** method (or by constructor with parameters).
- Then each party (thread) performs actions that make up the phase. And the Phaser synchronizer waits until all parties (threads) complete the phase execution.
- To notify the synchronizer that a phase is complete, the party (thread) must call the **arrive()** or **arriveAndAwait Advance()** method. After that, the synchronizer proceeds to the next phase.

Phaser

```
arriveAndAwaitAdvance();
arrive();
awaitAdvance(i);
arriveAndDeregister();
register();
```

phase = i

parties = 5

arrived = 0

see phaser

Phaser

Synchronizers

In addition to the objects-implementations of the Lock interface, the Java Concurrent API offers high-level synchronizers:

- **Semaphore** - an object that controls access to one or more shared resources by threads using an access permissions counter;
- **CountDownLatch** - an object that blocks all threads until a certain condition is met, when the condition is met, it releases all threads at the same time (waiting multiple concurrent events);
- **FutureTask** - an object that describes an abstract computation, that produces an asynchronously retrieved result.
- **Cyclic Barrier** - an object that allows threads to be synchronized at a common point at which a specified number of threads meet and block. Once all threads have arrived at the point, the optional operation is performed and the barrier is broken and the waiting threads are released (synchronizing tasks in a common point);
- **Exchanger** - an object designed to exchange data between threads at a certain point of both threads operations;
- **Phaser** - an object that allows to control execution of concurrent tasks divided in phases. All the threads must finish one phase before they can continue with the next one (since Java 7).