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. Exceptions and Assertions
11. Nested classes
12. Enums
13. Wrapper classes for primitive types
14. Generics
15. Collections
16. Method overload resolution
17. Multithreads
18. Core Java Classes
19. Object Oriented Design

Module contents

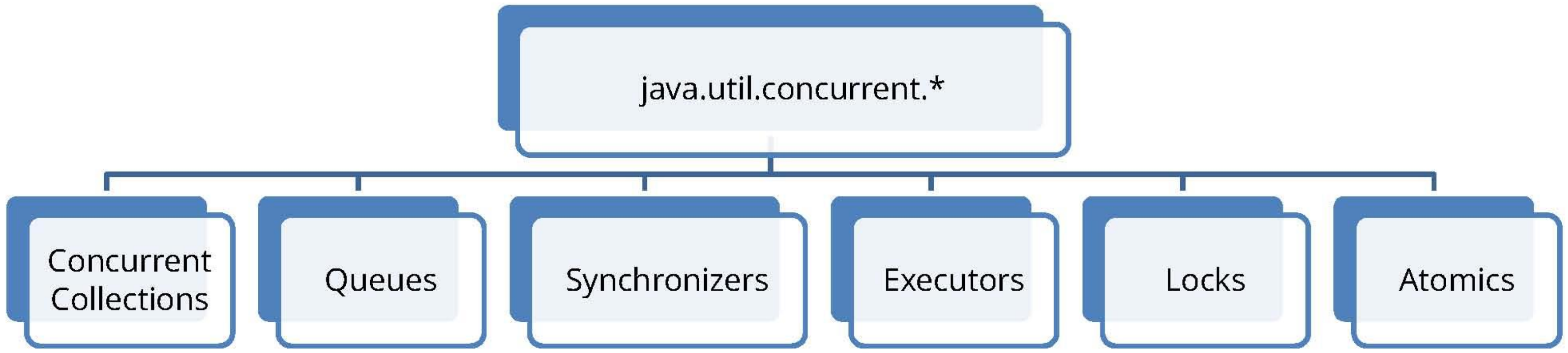
- Introduction to Concurrent Programming
- Creating Threads
- Important Methods in the Thread class
- Thread interruption. The interrupt() method
- The States of a Thread
- The Thread Scheduler. Thread Priority
- The Daemon Threads
- Thread Synchronization
- Synchronized Methods
- Synchronized Blocks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

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
- 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.



Threads pool 1/12

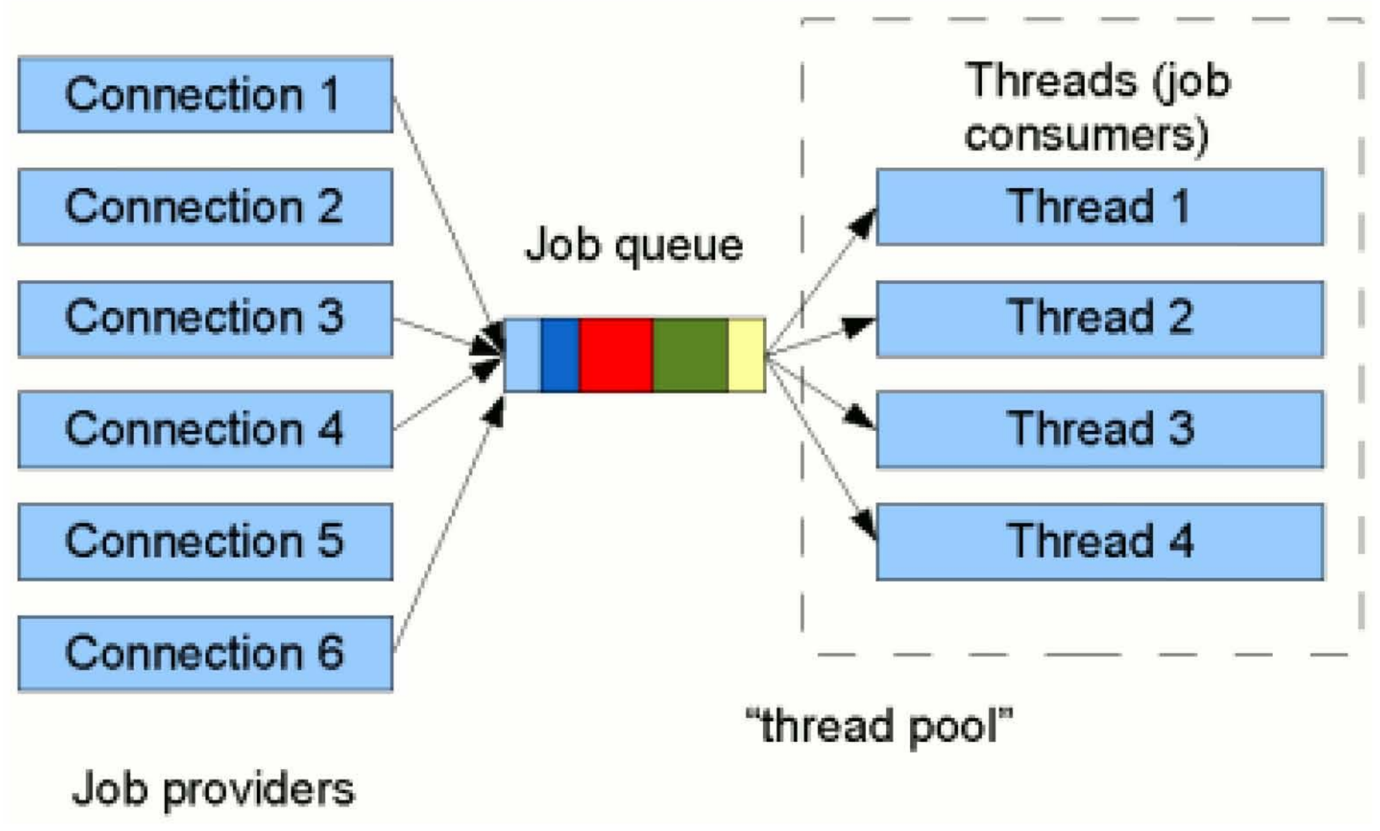
- Thread per task is the bad way:
- - creating a new thread for each request can consume significant computing resources
- - having many threads competing for the CPUs can impose other performance costs as well.

Threads pool 3/12

- 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.

Threads pool 2/12

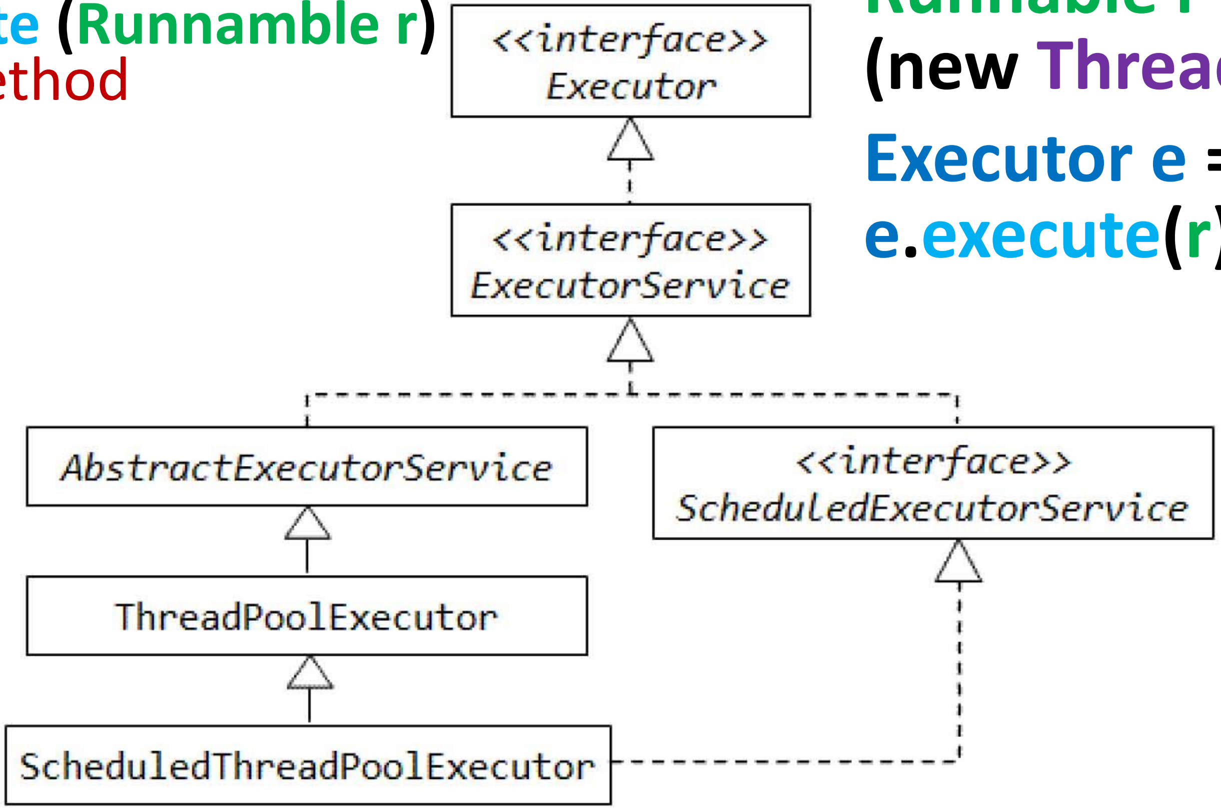
- Typical thread pool architecture



Threads pool 4/12

- Executor - interfases for thread pool implementations

void execute (Runnamble r)
the only method



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

Threads pool 5/12

Tasks execution:

Returns a Future object that can be used to track the progress of the task.

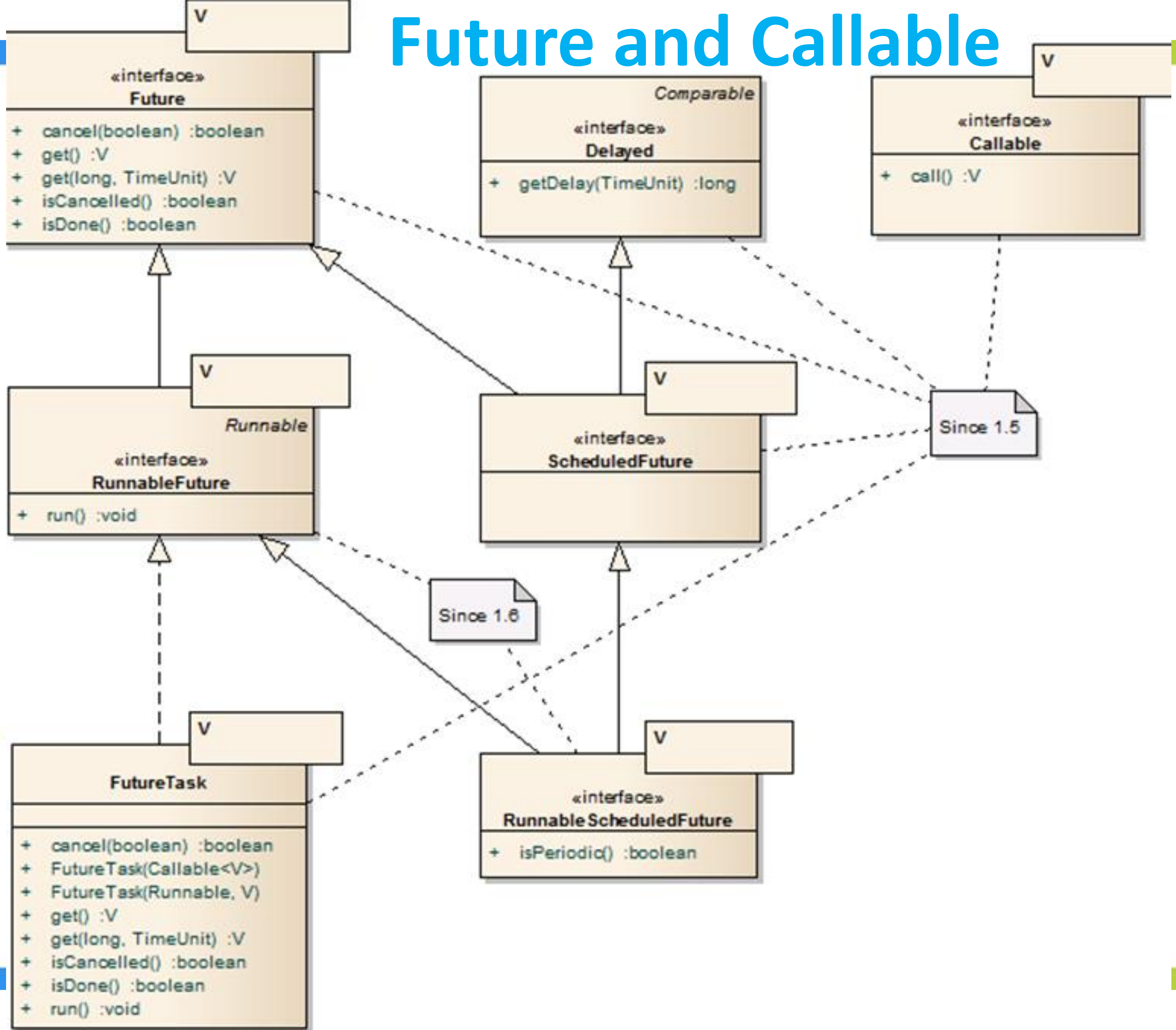
The basic ExecutorService methods:

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

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

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

Future and Callable



Threads pool 6/12

List<Future<T>> **invokeAll(**Collection<? extends Callable<T>> tasks**)** - returning a list of Futures holding their status and results when all complete.

T **invokeAny(**Collection<? extends Callable<T>> tasks**)** - methods execute the tasks in the given collection.

ExecutorService Interface

Returns	Method	Description
boolean	awaitTermination(long timeout, 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.

Threads pool 7/12

```java
class MyTask implements Runnable {
    String taskInfo;
    public MyTask(String taskInfo){
        this.taskInfo =  taskInfo;
    }
    @Override
    public void run() {
        System.out.println(taskInfo);
        //...
    }
}
```

Threads pool 8/12

```
public static void main(String[] args) {
   ThreadPoolExecutor tpe =
         new ThreadPoolExecutor(
               5, 10, 30L, TimeUnit.SECONDS,
               new LinkedBlockingQueue<Runnable>());
   MyTask[] tasks = new MyTask[25];
   for (int i = 0; i < tasks.length; i++) {
      tasks[i] = new MyTask("Task " + i);
      tpe.execute(tasks[i]);
   }
   tpe.shutdown();
}
```

ThreadPoolExecutor

Output:

```
Task-0
Task-2
Task-1
Task-7
Task-8
…
```

```
ThreadPoolExecutor tpe = new ThreadPoolExecutor(5, 10, 30L,
TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>(5));
```

```
java.util.concurrent.RejectedExecutionException: Task
executors.threadpoolexecutor.MyTask@67f89fa3 rejected from
java.util.concurrent.ThreadPoolExecutor@4ac68d3e
[Running, pool size = 10, active threads = 9, queued tasks = 5,
completed tasks = 6]
```

Threads pool 9/12

- **Executors.newSingleThreadExecutor()**: creates a single background thread
- **Executors.newFixedThreadPool**(int numThreads): creates a fixed size thread pool
- **Executors.newCachedThreadPool()**: create a unbounded thread pool, with automatic thread reclamation

Threads pool 10/12

```java
public class MyTestCallable  implements Callable<String> {
   private int workNumber;
   MyTestCallable(int workNumber) {
      this.workNumber = workNumber;
   }
   public String call() {
      for (int i = 1; i <= 5; i++) {
         System.out.printf("Work %d: %d\n", workNumber, i);
         try {
            Thread.sleep((int) (Math.random() * 1000));
         } catch (InterruptedException e) {
         }
      }
      return "work " + workNumber;
   }
}
```

Threads pool 11/12

```java
int numOfWorks = 20;
ExecutorService pool = Executors.newFixedThreadPool(4);
MyTestCallable works[] = new MyTestCallable[numOfWorks];
Future[] futures = new Future[numOfWorks];
for (int i = 0; i < numOfWorks; ++i) {
   works[i] = new MyTestCallable(i + 1);
   futures[i] = pool.submit(works[i]);
}
for (int i = 0; i < numOfWorks; ++i) {
   try {
      System.out.println(futures[i].get() + " ended");
   } catch (Exception ex) {
      ex.printStackTrace();
   }
}
```

Threads pool 12/12

Console output

```
Work 1: 1
Work 4: 1
Work 2: 1
Work 3: 1
Work 2: 2
Work 3: 2
Work 2: 3
Work 1: 2
Work 3: 3
…
```

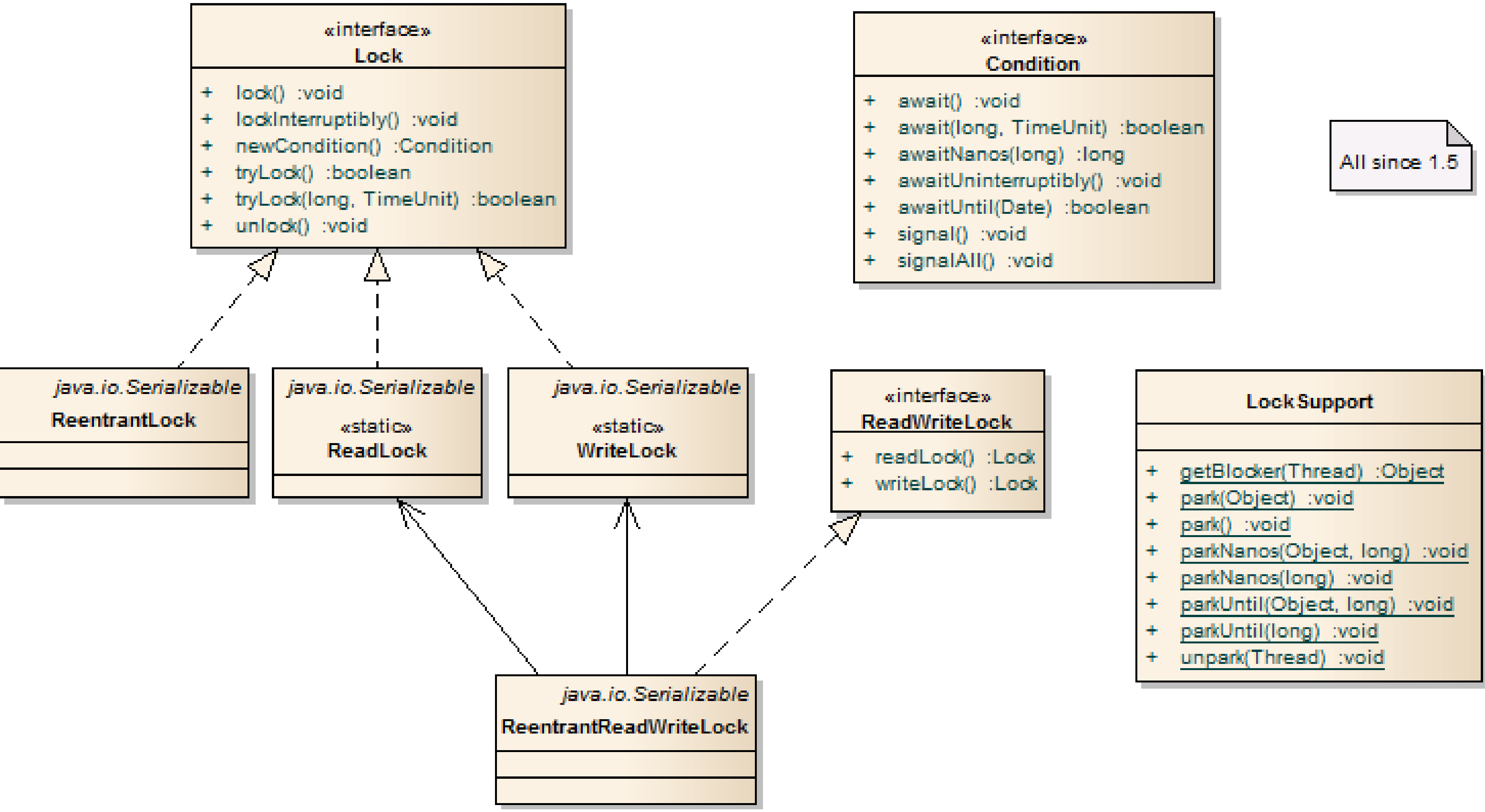
```
…
Work 7: 1
work 1 ended
work 2 ended
work 3 ended
Work 5: 4
Work 4: 5
Work 6: 2
work 4 ended
Work 8: 1
…
```

```
…
Work 19: 3
Work 19: 4
work 17 ended
Work 20: 4
Work 18: 5
work 18 ended
Work 19: 5
Work 20: 5
work 19 ended
work 20 ended
```

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
- Deadlocks
- Threads pool
- **The ReentrantLock class**
- Synchronizers
- Atomic Variables
- Concurrent Collection
- The Fork-Join Framework

Locks



«interface»
Lock
+ lock() :void
+ lockInterruptibly() :void
+ newCondition() :Condition
+ tryLock() :boolean
+ tryLock(long, TimeUnit) :boolean
+ unlock() :void
«interface»
Condition
+ await() :void
+ await(long, TimeUnit) :boolean
+ awaitNanos(long) :long
+ awaitUninterruptibly() :void
+ awaitUntil(Date) :boolean
+ signal() :void
+ signalAll() :void
All since 1.5
java.io.Serializable
ReentrantLock
java.io.Serializable
«static»
ReadLock
java.io.Serializable
«static»
WriteLock
«interface»
ReadWriteLock
+ readLock() :Lock
+ writeLock() :Lock
LockSupport
+ getBlocker(Thread) :Object
+ park(Object) :void
+ park() :void
+ parkNanos(Object, long) :void
+ parkNanos(long) :void
+ parkUntil(Object, long) :void
+ parkUntil(long) :void
+ unpark(Thread) :void
java.io.Serializable
ReentrantReadWriteLock

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.

The ReentrantLock class 3/5

- Lock interface provide opportunity of acquiring lock by different ways:
- lock()
- lockInterruptible()
- tryLock()
- tryLock(long timeout, TimeUnit timeUnit)
- There is only one method for unlocking the lock:
- unlock()

The ReentrantLock class 4/5

- **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.

The ReentrantLock class 5/5

```
public class MyCounter {
    private int x = 1;
    public synchronized void increment() {
        try {
            for (int i = 0; i < 4; i++) {
            System.out.printf("%s %d \n", Thread
                                .currentThread().getName(), x);
                x++;
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
        }
    }
}
```

The ReentrantLock class 6/5

```
public class MyCounterThread extends Thread {

    MyCounter res;

    MyCounterThread(MyCounter res) {
        this.res = res;
    }

    @Override
    public void run() {
        res.increment();
    }
}
```

The ReentrantLock class 7/5

```
public class Main {
    public static void main(String[] args) {
        MyCounter commonResource = new MyCounter();
        for (int i = 0; i < 5; i++) {
            Thread t = new
                            MyCounterThread(commonResource);
            t.setName("Поток " + i);
            t.start();
        }
    }
}
```

The ReentrantLock class 8/5

```
public class MyCounter {
    private int x = 1;
    ReentrantLock locker;
    public MyCounter(ReentrantLock locker) {
        this.locker = locker;
    }
    public void increment() {
        try {
            locker.lock();   //Получение блокировки
            System.out.println(Thread.currentThread()
                                        .getName() +": Lock acquired.");
...
```

The ReentrantLock class 9/5

```
…
        for (int i = 0; i < 4; i++) {
          System.out.printf("%s %d \n", Thread
                                   .currentThread().getName(), x);
          x++;

          Thread.sleep(500);
        }
      } catch (InterruptedException e) {
      } finally {
        locker.unlock();  //освобождение блокировки
        System.out.println(Thread.currentThread()
                                  .getName() +": Lock released.");
      }   }   }
```

The ReentrantLock class 10/5

```
Поток 0: Lock acquired.
Поток 0 1
Поток 0 2
Поток 0 3
Поток 0 4
Поток 0: Lock released.
Поток 3: Lock acquired.
Поток 3 5
Поток 3 6
Поток 3 7
Поток 3 8
Поток 3: Lock released.
Поток 4: Lock acquired.
Поток 4 9
Поток 4 10
Поток 4 11
Поток 4 12
Поток 4: Lock released.
Поток 1: Lock acquired.
Поток 1 13
Поток 1 14
Поток 1 15
Поток 1 16
Поток 1: Lock released.
Поток 2: Lock acquired.
Поток 2 17
Поток 2 18
Поток 2 19
Поток 2 20
Поток 2: Lock released.
```

The ReentrantLock class 11/5

```
public class MyCounter {
    private int x = 1;
    ReentrantLock locker;
    public MyCounter(ReentrantLock locker) {
        this.locker = locker;
    }
    public void increment() {
        try {
            boolean flag = locker.tryLock(3000,
                                    TimeUnit.MILLISECONDS);
            if (flag) {
                try {
                    System.out.println(Thread.currentThread()
...                           .getName() + ": Lock acquired.");
```

The ReentrantLock class 12/5

```
…
                for (int i = 0; i < 4; i++) {
                    System.out.printf("%s %d \n",
                            Thread.currentThread().getName(), x);
                    x++;
                    Thread.sleep(500);
                }
            }finally {
                locker.unlock();
                System.out.println(Thread.currentThread()
                                        .getName() +": Lock released.");
            }
        }
…
```

The ReentrantLock class 13/5

```
else{
        System.out.println(Thread.currentThread()
         .getName() +": Can't get lock. Contimue to work.");
         }
        } catch (InterruptedException e) {
        }
    } }
```

```
Поток 1: Lock acquired.
Поток 1 1
Поток 1 2
Поток 1 3
Поток 1 4
Поток 1: Lock released.
Поток 0: Lock acquired.
Поток 0 5
Поток 0 6
Поток 3: Can't get lock. Contimue to work.
Поток 2: Can't get lock. Contimue to work.
Поток 4: Can't get lock. Contimue to work.
Поток 0 7
Поток 0 8
Поток 0: Lock released.
СБОРКА УСПЕШНО ЗАВЕРШЕНА
```

The ReentrantLock class 14/5
Condition using – thread coordination with wait/notify/notifyAll

```
public class Producer implements Runnable {
    Store store;
    public Producer(Store store) {
        this.store = store;
    }
    @Override
    public void run() {
        for (int i = 1; i < 6; i++) {
            store.put();
        }
    }
}
```

Supplier of goods to the store

The ReentrantLock class 14/5
Condition using – thread coordination with wait/notify/notifyAll

```
public class Consumer implements Runnable {
    Store store;
    public Consumer(Store store) {
        this.store = store;
    }
    @Override
    public void run() {
        for (int i = 1; i < 6; i++) {
            store.get();
        }
    }
}
```

Store buyer

The ReentrantLock class 14/5
Condition using – thread coordination with wait/notify/notifyAll

```
public class Store {
    private int product = 0;
    public synchronized void get() {
        try {
            //поки немає доступних товарів на складі, очікуємо
            while (product < 1) {
                this.wait();
            }
            product--;
            System.out.println("Consumer bought 1 product");
            System.out.println("Goods in stock: " + product);
...
```

The ReentrantLock class 14/5

Condition using – thread coordination with wait/notify/notifyAll

```
..
        //сигналізуємо про можливість блокування
        this.notifyAll();
    } catch (InterruptedException ex) {
        System.out.println(ex.getMessage());
    }
  }
   public synchronized void put() {
     try {
       //поки на складі 3 товари, чекаємо звільнення місця
        while (product >= 3) {
          this.wait();
        }   …
```

The ReentrantLock class 14/5
Condition using – thread coordination with wait/notify/notifyAll

```
..
          product++;
          System.out.println("Producer added 1 product");
          System.out.println("Goods in stock: " + product);
          //сигналізуємо про можливість блокування
          this.notifyAll();
      } catch (InterruptedException e) {
          System.out.println(e.getMessage());
      }
  }
}
```

The ReentrantLock class 14/5
Condition using – thread coordination with wait/notify/notifyAll

```
public static void main(String[] args) {
    Store store = new Store();
    Producer producer = new Producer(store);
    Consumer consumer = new Consumer(store);
    new Thread(producer).start();
    new Thread(consumer).start();
}
```

The ReentrantLock class 14/5
Condition using – thread coordination with wait/notify/notifyAll

```
Producer added 1 product
Goods in stock: 1
Producer added 1 product
Goods in stock: 2
Producer added 1 product
Goods in stock: 3
Consumer bought 1 product
Goods in stock: 2
Consumer bought 1 product
Goods in stock: 1
Consumer bought 1 product
Goods in stock: 0
…
```

```
…
Producer added 1 product
Goods in stock: 1
Producer added 1 product
Goods in stock: 2
Consumer bought 1 product
Goods in stock: 1
Consumer bought 1 product
Goods in stock: 0
```

The ReentrantLock class 14/5
Condition using – thread coordination with Condition await/signal/signalAll

```
public class Store {
    private int product = 0;
    ReentrantLock locker;
    Condition condition;
    public Store() {
        locker = new ReentrantLock();
        condition = locker.newCondition();
    }
    public void get() {
        try {
            locker.lock();
...
```

The ReentrantLock class 14/5

Condition using – thread coordination with Condition await/signal/signalAll

```
…
        //поки немає доступних товарів на складі, очікуємо
          while (product < 1) {condition.await();  }
          product--;
          System.out.println("Consumer bought 1 product ");
          System.out.println("Goods in stock: " + product);
           //сигналізуємо про можливість блокування
          condition.signalAll();
      } catch (InterruptedException ex) {
          System.out.println(ex.getMessage());
      } finally {
          locker.unlock();
      }      }  …
```

The ReentrantLock class 14/5

Condition using – thread coordination with Condition await/signal/signalAll

```
…
    public void put() {
        try {
            locker.lock();
          //поки на складі 3 товари, чекаємо звільнення місця
            while (product >= 3) {
                condition.await();
            }
            product++;
            System.out.println("Producer added 1 product ");
            System.out.println("Goods in stock: " + product);
…
```

The ReentrantLock class 14/5

Condition using – thread coordination with Condition await/signal/signalAll

```
…
            //сигналізуємо про можливість блокування
            condition.signalAll();
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        } finally {
            locker.unlock();
        }
    }
}
```

Running the program will give the result similar to the previous program.

The ReentrantLock class 14/5

Reentrance demonstration – nested methods

```
public class NestedMethodsTask {
    int x = 1;
    ReentrantLock locker;
    public NestedMethodsTask(ReentrantLock locker) {
        this.locker = locker;
    }
    public void outerMethod() {
        try {
          locker.lock();
          System.out.println(Thread.currentThread().getName()
            + ": Lock acquired and " + locker.getHoldCount()
            + " lock hold in outerMethod().");
...
```

The ReentrantLock class 14/5

Reentrance demonstration

```
…
        for (int i = 0; i < 2; i++) {
            System.out.printf("%s %d \n",
                    Thread.currentThread().getName(), x);
            x++;
            Thread.sleep(500);
        }
        innerMethod();
    } catch (InterruptedException e) {
    } finally {
      locker.unlock();
      System.out.println(Thread.currentThread().getName()
        + ": Lock released and " + locker.getHoldCount()
        + " lock hold in outerMethod().");        }    }   …
```

The ReentrantLock class 14/5
Reentrance demonstration

```
…
   public void innerMethod() {
      try {
        locker.lock();
        System.out.println(Thread.currentThread().getName()
           + ": Lock acquired and " + locker.getHoldCount()
           + " lock hold in innerMethod().");
          System.out.printf("%s %d \n",
                     Thread.currentThread().getName(), x);
          x++;
          Thread.sleep(500);
      } catch (InterruptedException ex) {
      }
…
```

The ReentrantLock class 14/5

Reentrance demonstration

```
…
        finally {
           locker.unlock();
          System.out.println(Thread.currentThread().getName()
            + ": Lock released and " + locker.getHoldCount()
            + " lock hold in innerMethod().");
        }
    }
}
```

The ReentrantLock class 14/5

Reentrance demonstration

```
public class MyThread extends Thread {
    NestedMethodsTask res;
    MyThread(NestedMethodsTask res) {
        this.res = res;
    }
    @Override
    public void run() {
        try {
            sleep((long) (Math.random() * 1000));
        } catch (InterruptedException ex) {
        }
        res.outerMethod();
} }
```

The ReentrantLock class 14/5
Reentrance demonstration

```
public static void main(String[] args) {
    ReentrantLock locker = new ReentrantLock();
    NestedMethodsTask commonResource =
                                    new NestedMethodsTask(locker);
    for (int i = 0; i < 2; i++) {
        Thread t = new MyThread(commonResource);
        t.setName("Поток " + i);
        t.start();
    }
}
```

The ReentrantLock class 14/5
Reentrance demonstration

```
Поток 0: Lock acquired and 1 lock hold in outerMethod().
Поток 0 1
Поток 0 2
Поток 0: Lock acquired and 2 lock hold in innerMethod().
Поток 0 3
Поток 0: Lock released and 1 lock hold in innerMethod().
Поток 0: Lock released and 0 lock hold in outerMethod().
Поток 1: Lock acquired and 1 lock hold in outerMethod().
Поток 1 4
Поток 1 5
Поток 1: Lock acquired and 2 lock hold in innerMethod().
Поток 1 6
Поток 1: Lock released and 1 lock hold in innerMethod().
Поток 1: Lock released and 0 lock hold in outerMethod().
```

The ReentrantLock class 14/5

When to use Reentrant lock

Use **ReentrantLock** objects when we need something that is not supported by **synchronized**, such as:

- releasing threads for another job that do not receive a lock;
- waiting for a lock for a certain amount of time;
- arranging a lock that can be interrupted by another thread; the use of several variable blocking conditions or the organization of a blocking poll.

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
- Deadlocks
- Threads pool
- The ReentrantLock class
- **Synchronizers**
- Atomic Variables
- Concurrent Collection
- The Fork-Join Framework

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.

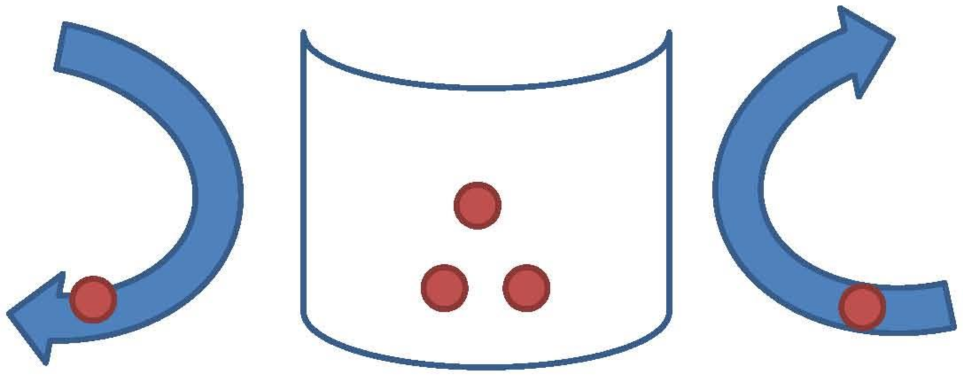
```
Semaphore semaphore = new Semaphore(5);
```

permits

Semaphore 2/10

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

Semaphore s = new Semaphore (3)



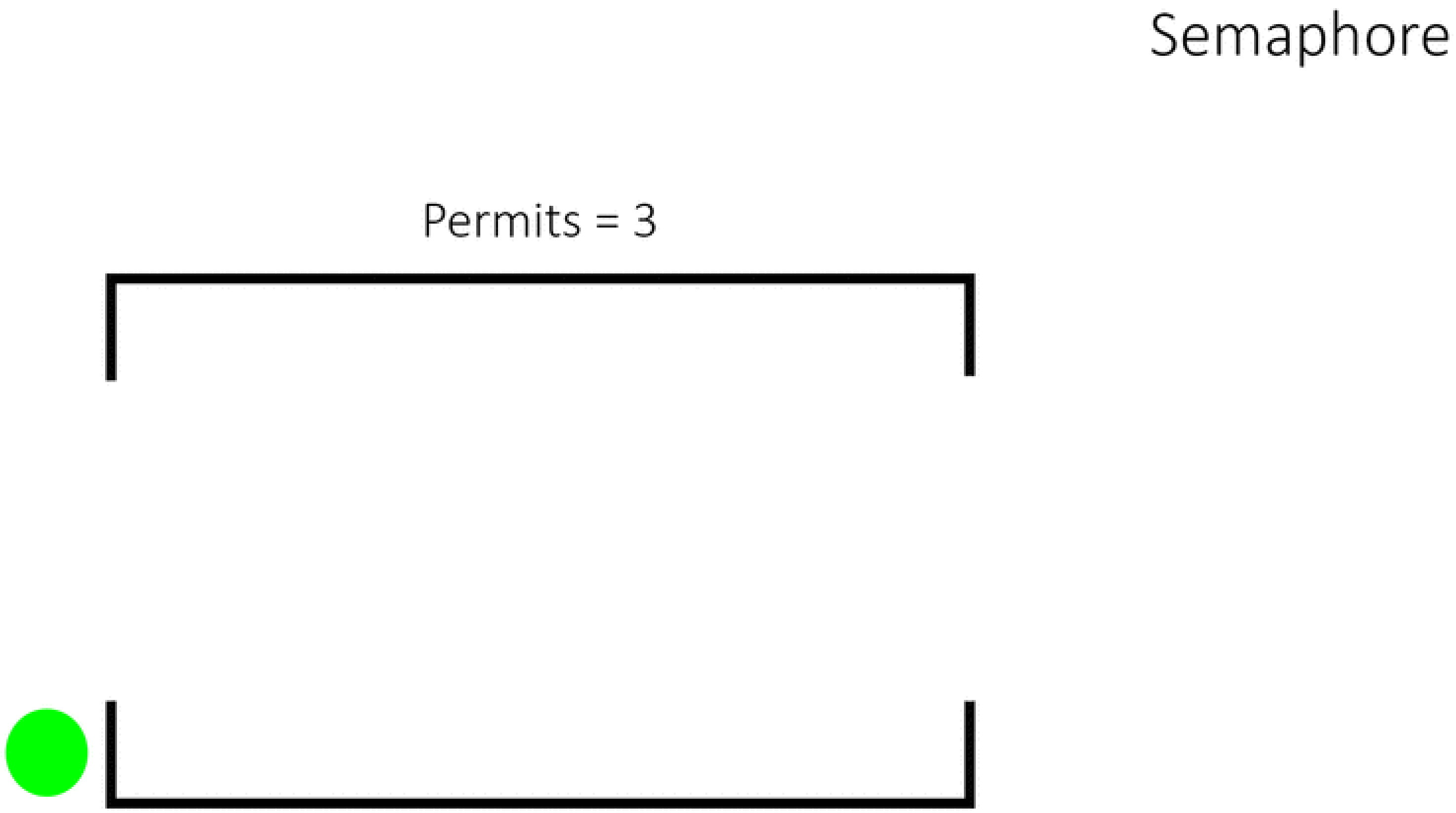
s.acquire()

s.release()

fair

Semaphore sem = new Semaphore(5, true);

Semaphore



```
Semaphore s = new Semaphore(3, true);
```

Semaphore 3/10

```
public class Task implements Runnable {
    Semaphore semaphore;
    public Task(Semaphore semaphore) {
        this.semaphore = semaphore;
    }
     public void run() {
        boolean permit = false;
        try { permit = semaphore.tryAcquire(3000,
                                              TimeUnit.MILLISECONDS);
         if (permit) {
               System.out.println(Thread.currentThread().getName()
                                                   + ": Permit acquired");
              sleep(5000);
...
```

Semaphore 4/10

```
        } else {
                System.out.println(Thread.currentThread().getName()
                                        + ": Could not acquire permit");
          }
        } catch (InterruptedException ex) {
        } finally {
            if (permit) {
                semaphore.release();
                System.out.println(Thread.currentThread(). getName()
                                                        + ": Permit released");
             }
         }
    }
}
```

Semaphore 5/10

```
public class SemaphoreDemo {
   public static void main(String[] args) throws InterruptedException {
      ExecutorService executor = Executors.newFixedThreadPool(10);
      Semaphore semaphore = new Semaphore(3);
      Task task = new Task(semaphore);
      for (int i = 0; i < 10; i++) {
         executor.submit(task);
         Thread.sleep(500);    //потрібно підбирати час, щоб побачити
      }                         //роботу семафору
      executor.shutdown();
   }
}
```

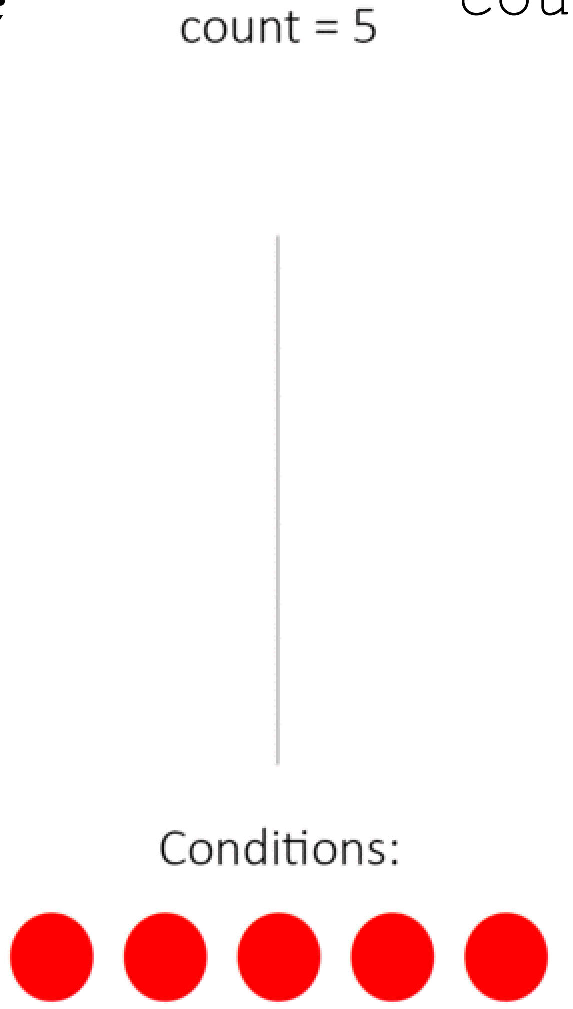
Semaphore 10/10

Output:

```
pool-1-thread-1: Permit acquired
pool-1-thread-2: Permit acquired
pool-1-thread-3: Permit acquired
pool-1-thread-4: Could not acquire permit
pool-1-thread-1: Permit released
pool-1-thread-5: Permit acquired
pool-1-thread-2: Permit released
pool-1-thread-6: Permit acquired
pool-1-thread-3: Permit released
pool-1-thread-7: Permit acquired
pool-1-thread-8: Could not acquire permit
pool-1-thread-9: Could not acquire permit
pool-1-thread-10: Could not acquire permit
pool-1-thread-5: Permit released
pool-1-thread-6: Permit released
pool-1-thread-7: Permit released
```

CountDownLatch 1/5

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



CountDownLatch 3/5

```
class Runner extends Thread {
    private CountDownLatch timer;
    public Runner(CountDownLatch cdl, String name) {
        timer = cdl;
        this.setName(name);
        System.out.println(this.getName() +
                " ready and waiting to start");
        start();
    }
...
```

CountDownLatch 4/5

```
 ...
public void run() {
        try {
            timer.await();
        } catch (InterruptedException ie) {
            System.err.println("interrupted -"+
                    "can't start running the race");
        }
        System.out.println(this.getName() +
                " started running");
    }
}
```

CountDownLatch 5/5

```java
CountDownLatch counter = new CountDownLatch(5);
new Runner(counter, "Carl");
new Runner(counter, "Joe");
new Runner(counter, "Jack");
System.out.println("Starting the countdown ");
long countVal = counter.getCount();
while (countVal > 0) {
   Thread.sleep(1000);
   System.out.println(countVal);
   if (countVal == 1) {
      System.out.println("Start");
   }
   counter.countDown();
   countVal = counter.getCount();
}
```

CountDownLatch

```
run:
Carl ready and waiting to start
Joe ready and waiting to start
Jack ready and waiting to start
Starting the countdown
5
4
3
2
1
Start
Carl started running
Jack started running
Joe started running
```

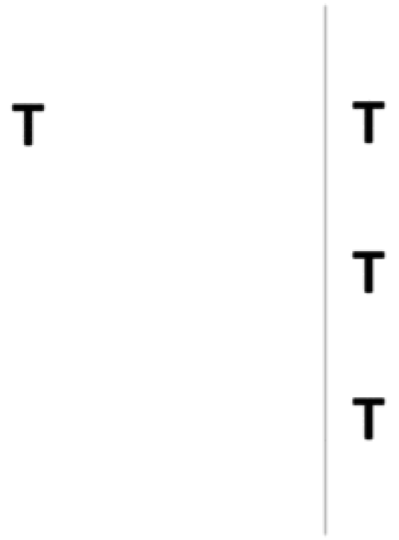
CyclicBarrier

- **CyclicBarrier** is a synchronization point where a specified in constructor number of parallel threads meet and block. Once all threads have arrived, an option barrierAction is performed (or not performed if the barrier was initialized without it), and, after the barrierAction, the barrier breaks and the waiting threads are "released".
- The number of parties to be "met" and, optionally, the action to be taken when the parties have met, but before when they are "released“ is passed to the barrier constructor (`CyclicBarrier (int parties)` and `CyclicBarrier (int parties, Runnable barrierAction)`).

CyclicBarrier

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

parties = 3



barrierAction

CyclicBarrier

- The **int await()** method of a CyclicBarrier object indicates to the thread from which the method was called that it came to the barrier. This thread is put on WAITING state until all other threads, the number of which is specified in the constructor as a party parameter, reach the barrier (they will call the int await() method of the CyclicBarrier object).
- There is an **int await (long timeout, TimeUnit unit)** method of the CyclicBarrier object, which puts the current thread on WAITING state until all other threads reach the barrier or the time interval specified as a parameter expires.

CyclicBarrier

```
public class Car implements Runnable {
    private int carNumber;
    public Car(int carNumber) {
        this.carNumber = carNumber;
    }
    @Override
    public void run() {
        try {
            System.out.printf("Car №%d drove up to the ferry.\n",
                                                                   carNumber);
            Ferry.BARRIER.await();
            System.out.printf("Car №%d continued to move.\n",
                                                                   carNumber);
        } catch (Exception e) {         }
    } }
```

CyclicBarrier

```
public class FerryBoat implements Runnable {
    @Override
    public void run() {
        try {
            Thread.sleep(500);
            System.out.println("FerryBoat ferrying cars!");
        } catch (InterruptedException e) {
        }
    }
}
```

CyclicBarrier

```
public class Ferry {
    static final CyclicBarrier BARRIER =
                                    new CyclicBarrier(3, new FerryBoat());
    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 9; i++) {
            new Thread(new Car(i)).start();
            Thread.sleep(400);
        }
    }
}
```

CyclicBarrier

Output:

```
Car №0 drove up to the ferry.
Car №1 drove up to the ferry.
Car №2 drove up to the ferry.
Car №3 drove up to the ferry.
FerryBoat ferrying cars!
Car №2 continued to move.
Car №1 continued to move.
Car №0 continued to move.
Car №4 drove up to the ferry.
Car №5 drove up to the ferry.
Car №6 drove up to the ferry.
FerryBoat ferrying cars!
…
```

```
…
Car №3 continued to move.
Car №5 continued to move.
Car №4 continued to move.
Car №7 drove up to the ferry.
Car №8 drove up to the ferry.
FerryBoat ferrying cars!
Car №8 continued to move.
Car №7 continued to move.
Car №6 continued to move.
```

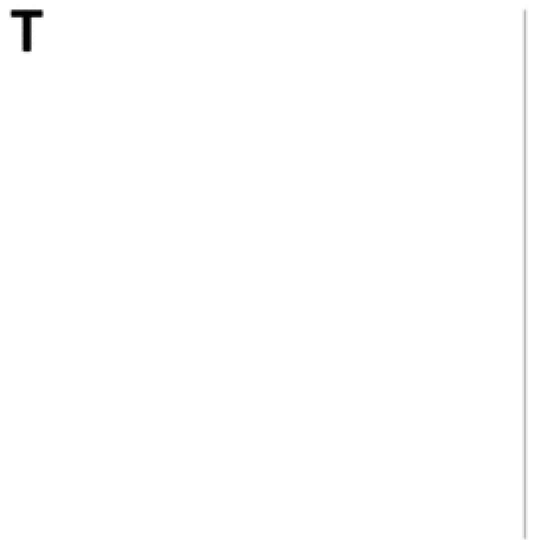
The barrier is called Cyclic because it can be reused after the expected streams are released.

Exchanger

- **Exchanger<V>** objects can exchange data between two threads at a certain point in both threads. 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, objects will be exchanged: each will receive an argument of the **V exchange(V x)** method called in another thread.

Exchanger

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



```
exchanger.exchange("Some string");
```

Exchanger

```
public class Truck implements Runnable {
   private int number;
   private String dep;
   private String dest;
   private String[] parcels;
   public Truck(int number, String departure,
                    String destination, String[] parcels) {
      this.number = number;
      this.dep = departure;
      this.dest = destination;
      this.parcels = parcels;
   }
…
```

Exchanger

```
…
   @Override
   public void run() {
     try {
       System.out.printf("The truck №%d was loaded with:
                                    %s и %s.\n", number, parcels[0], parcels[1]);
       System.out.printf("The truck №%d went from point
                                              %s to point %s.\n", number, dep, dest);
       Thread.sleep(1000 + (long) Math.random() * 5000);
       System.out.printf("The truck №%d arrived at point E.\n", number);
       parcels[1] = Delivery.EXCHANGER.exchange(parcels[1]);
       System.out.printf("The parcel for point %s was
                                         moved to truck №%d.\n", dest, number);
       Thread.sleep(1000 + (long) Math.random() * 5000);
       System.out.printf("Truck №%d arrived at point %s and delivered
         the parcels: %s and %s.\n", number,  dest, parcels[0], parcels[1]);
     } catch (InterruptedException e) {         }          } }
```

Exchanger

```
public class Delivery {
    static final Exchanger<String> EXCHANGER = new Exchanger<>();
    public static void main(String[] args) throws InterruptedException {
        String[] p1 = new String[]{"{parcel A->D}",
                                    "{parcel A->C}"};      //for 1-st truck
        String[] p2 = new String[]{"{parcel B->C}",
                                    "{parcel B->D}"};      //for 2-nd truck
        new Thread(new Truck(1, "A", "D", p1)).start();
        Thread.sleep(100);
        new Thread(new Truck(2, "B", "C", p2)).start();
    }
}
```

Exchanger

Output:

```
The truck №1 was loaded with: {parcel A->D} и {parcel A->C}.
The truck №1 went from point A to point D.
The truck №2 was loaded with: {parcel B->C} и {parcel B->D}.
The truck №2 went from point B to point C.
The truck №1 arrived at point E.
The truck №2 arrived at point E.
The parcel for point D was moved to truck №1.
The parcel for point C was moved to truck №2.
Truck №1 arrived at point D and delivered the parcels: {parcel A->D}
and {parcel B->D}.
Truck №2 arrived at point C and delivered the parcels: {parcel B->C}
and {parcel A->C}.
```

Phaser

- **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

- 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) that must perform all phases of the action.

`parent` - the parent Phaser object.

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 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 (flows) involved in the implementation of the phases by register() method (or by constructor with parameters).
- Then each side (thread) performs a certain set of actions that make up the phase. And the Phaser synchronizer waits until all parties (threads) finish completion of execution of a phase. 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

Phaser

Phaser

```
public class Passenger extends Thread {
    int departure;
    int destination;
    public Passenger(int departure, int destination) {
        this.departure = departure;
        this.destination = destination;
        System.out.println(this + " waiting at the bus stop №"
                                        + this.departure);
    }
    @Override
    public void run() {
        try {
            System.out.println(this + " got on the bus.");
...
```

Phaser

```
…
        while (Bus.PHASER.getPhase() < destination) {
          /*Поточний потік завершив фазу, очікуємо
           завершення фази іншими потоками*/
          Bus.PHASER.arriveAndAwaitAdvance();
        }
        Thread.sleep(1);
        System.out.println(this + " left the bus.");
        /*Всі фази поточний потік завершив*/
        Bus.PHASER.arriveAndDeregister();
      } catch (InterruptedException e) {
      }
    }
    @Override
    public String toString() {
      return "Passenger{" + departure + " -> "  + destination + '}';
    } }
```

Phaser

```
public class Bus {
    static final Phaser PHASER = new Phaser(1); //Реєстрація
              //потоку Main. Фази 0 та 6 - автобусний парк, 1 - 5 зупинки
    public static void main(String[] args) throws InterruptedException {
        ArrayList<Passenger> passengers = new ArrayList<>();
        for (int i = 1; i < 5; i++) {           //Генерація пасажирів
            if ((int) (Math.random() * 2) > 0) {
                passengers.add(new Passenger(i, i + 1));    //виходить на
            }                                               //наступній зупинці
            if ((int) (Math.random() * 2) > 0) {
                passengers.add(new Passenger(i, 5));   //виходить на кінцевій
            }                                          //зупинці
        }
...
```

Phaser

```
…
      for (int i = 0; i < 7; i++) {
        switch (i) {
          case 0:
            System.out.println("The bus left the park.");
            /*Потік Main (автобус) завершив фазу 0*/
            PHASER.arrive();
            break;
          case 6:
            System.out.println("The bus went to park.");
            /*Потік Main (автобус) завершив всі фази*/
            PHASER.arriveAndDeregister();
            break;
…
```

Phaser

```
…
        default:
         int currentBusStop = PHASER.getPhase();
         System.out.println("Bus stop № " + currentBusStop);
         /*Перевіряємо, чи є пасажири на зупинці*/
         for (Passenger p : passengers) {
           if (p.departure == currentBusStop) {
             /*Якщо на зупинці є пасажири,
             реєструємо новий потік*/
             PHASER.register();
             p.start();
           }           }
         /*Поточний потік завершив фазу, очікуємо
          її завершення іншими потоками*/
         PHASER.arriveAndAwaitAdvance();
       }  //кінець switch
     }  //кінець for       }    }
```

Phaser

Output:

```
Passenger{2 -> 3} waiting at the bus stop № 2
Passenger{4 -> 5} waiting at the bus stop № 4
Passenger{4 -> 5} waiting at the bus stop № 4
The bus left the park.
Bus stop № 1
Bus stop № 2
Passenger{2 -> 3} got on the bus.
Bus stop № 3
Passenger{2 -> 3} left the bus.
Bus stop № 4
Passenger{4 -> 5} got on the bus.
Passenger{4 -> 5} got on the bus.
Bus stop № 5
Passenger{4 -> 5} left the bus.
Passenger{4 -> 5} left the bus.
The bus went to the park.
```

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
- Deadlocks
- Threads pool
- The ReentrantLock class
- Synchronizers
- **Atomic Variables**
- Concurrent Collection
- The Fork-Join Framework

Compare And Swap (CAS) algorithm

There are 3 parameters for a CAS operation:

- a memory location V where value has to be replaced
- old value A which was read by thread last time
- new value B which should be written over V

1) Suppose first V = 10 and there are threads 1 and 2 that want to read and increment the values in the memory cell V:
   $V = 10, A_1 = 0, B_1 = 0, A_2 = 0, B_2 = 0$
2) Threads 1 and 2 want to increase the value of V, they both read the value:
   $V = 10, A_1 = 10, B_1 = 0, A_2 = 10, B_2 = 0$
3) Threads 1 and 2 increase the read value by 1 in their local variables (also remembering the previous values):
   $V = 10, A_1 = 10, B_1 = 11, A_2 = 10, B_2 = 11$

…

… Compare And Swap (CAS) algorithm

4) Let thread 1 access the memory cell first and compare the value of V with the last read value:
   $V = 10, A_1 = 10, B_1 = 11, A_2 = 10, B_2 = 11$
   V will be swapped as 11:
   $V = 11, A_1 = 11, B_1 = 11, A_2 = 10, B_2 = 11$
5) When thread 2 accesses a memory cell, it performs a similar operation:
   In this case, $V = 11$ is not equal to $A_2 = 10$, so the value is not replaced and returns the current value of $V = 11$. Thread 2 updates the last read value in A_2
   $V = 11, A_1 = 11, B_1 = 11, A_2 = 11, B_2 = 11$

```
if  (A == V ) {
    V = B
} else  {
    operation failed
}
   return V
```

Compare And Swap (CAS) algorithm

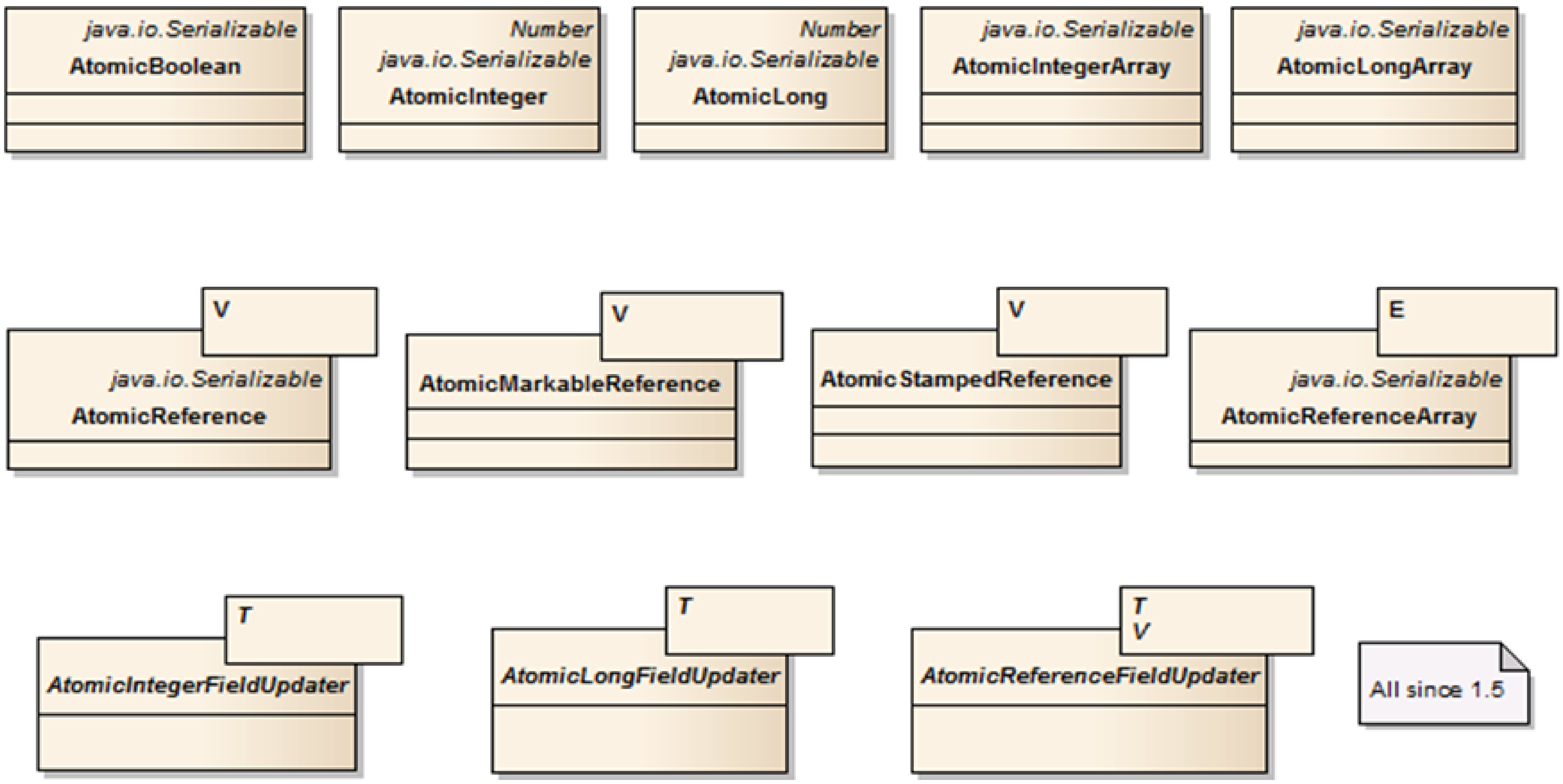
…

6) Now thread 2 will repeat the increment operation again with the values:
   $V = 11, A_1 = 11, B_1 = 11, A_2 = 11, B_2 = 12$
7) When thread 2 now has access to the cell and no other thread has changed its value during this time, it executes the CAS-algorithm, replaces the value of V with its incremental (because $A_2 = 11$ was equal to $V = 11$).
   New values will be:
   $V = 12, A_1 = 11, B_1 = 11, A_2 = 11, B_2 = 12$

```
if  (A == V ) {
    V = B
} else  {
    operation failed
}
   return V
```

Atomics

java.util.concurrent.atomic



Added in Java 5

Atomics

AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference:

`boolean compareAndSet(type expect, type update)` - takes two arguments of the corresponding types: the estimated current and new value. The method atomically sets the object to a new value if the current value is equal to the expected one, and returns true. If the current value changes, the method will return false and the new value will not be set.

`type getAndSet(type newValue)` - atomically unconditionally sets a new value and returns the old one.

In addition **AtomicInteger** and **AtomicLong** has:

`type getAndIncrement()` - atomic increment of the current value and return of the old value (equivalent to the operation `i++`);

`type incrementAndGet()` - atomic increment of the current value and return of the old value after increase (equivalent to operation `++i`); …

type is boolean or int or long or V

Atomics

...

`type getAndDecrement()` - atomic decrement of current value and return of old value (equivalent to operation `i--`);

`type decrementAndGet()` - atomic decrement of the current value and return of the old value after reduction (equivalent to the `--i` operation);

`type addAndGet(type delta)` - atomic addition of value-argument to the current one, returns a new value after addition;

`type getAndAdd(type delta)` - atomic addition of an argument value to the current one, returns the old value.

Also, all these classes have methods for obtaining the current value of `type get()` and unconditionally setting the specified value of `void set(type newValue)`.

...

type is boolean or int or long or V

Atomics

AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray contain methods for working with array elements, similar to the methods of the classes AtomicInteger, AtomicLong, AtomicReference. The difference between these methods is in adding an additional argument that points to the index of the element in the array `i`, for example, `boolean compareAndSet(int i, type expect, type update)`, `type getAndIncrement(int i)`, etc.

...

type is boolean or int or long or V

Atomics

...
AtomicIntegerFieldUpdater, AtomicLongFieldUpdater, AtomicReferenceFieldUpdater contain methods for updating the values of object fields by their names using reflection, similar to the methods of the AtomicInteger, AtomicLong, AtomicReference.. The difference between these methods is adding an additional argument indicating the `obj` object whose field is being updated, for example, `boolean compareAndSet(T obj, type expect, type update)`, `type getAndIncrement method (T obj)`, etc.

The **AtomicMarkableReference** class supports object references along with a tag bit that can be updated atomically.

The **AtomicStampedReference** class supports a reference to an object along with an integer "stamp" that can be updated atomically.

Atomic Variables 1/5

- Composite read/write operations such as the increment/decrement operation on volatile variables are not performed atomically

```
class MyCounter {
public int cnt1;
public volatile int cnt2;
public AtomicInteger cnt3 = new AtomicInteger(0);
}
```

Atomic Variables 2/5

```java
class MyCountThread extends Thread{
    MyCounter m;
    int n;
    public MyCountThread(MyCounter m,int n){
        this.m = m; this.n = n;
    }
    public void run(){
        for(int i=0;i<n;i++)
        {
            this.m.cnt1++; this.m.cnt2++;
            this.m.cnt3.getAndIncrement();
        }
    }
}
```

Atomic Variables 3/5

```
public static void main(String[] args) {
    MyCounter  m = new  MyCounter();\
    MyCountThread[] tg = new MyCountThread[100];
    for(int i = 0 ; i < 100; i++){
        tg[i] = new MyCountThread(m,1000000);
    }
    for(MyCountThread t:tg){
        t.start();
    }
...
```

Atomic Variables 4/5

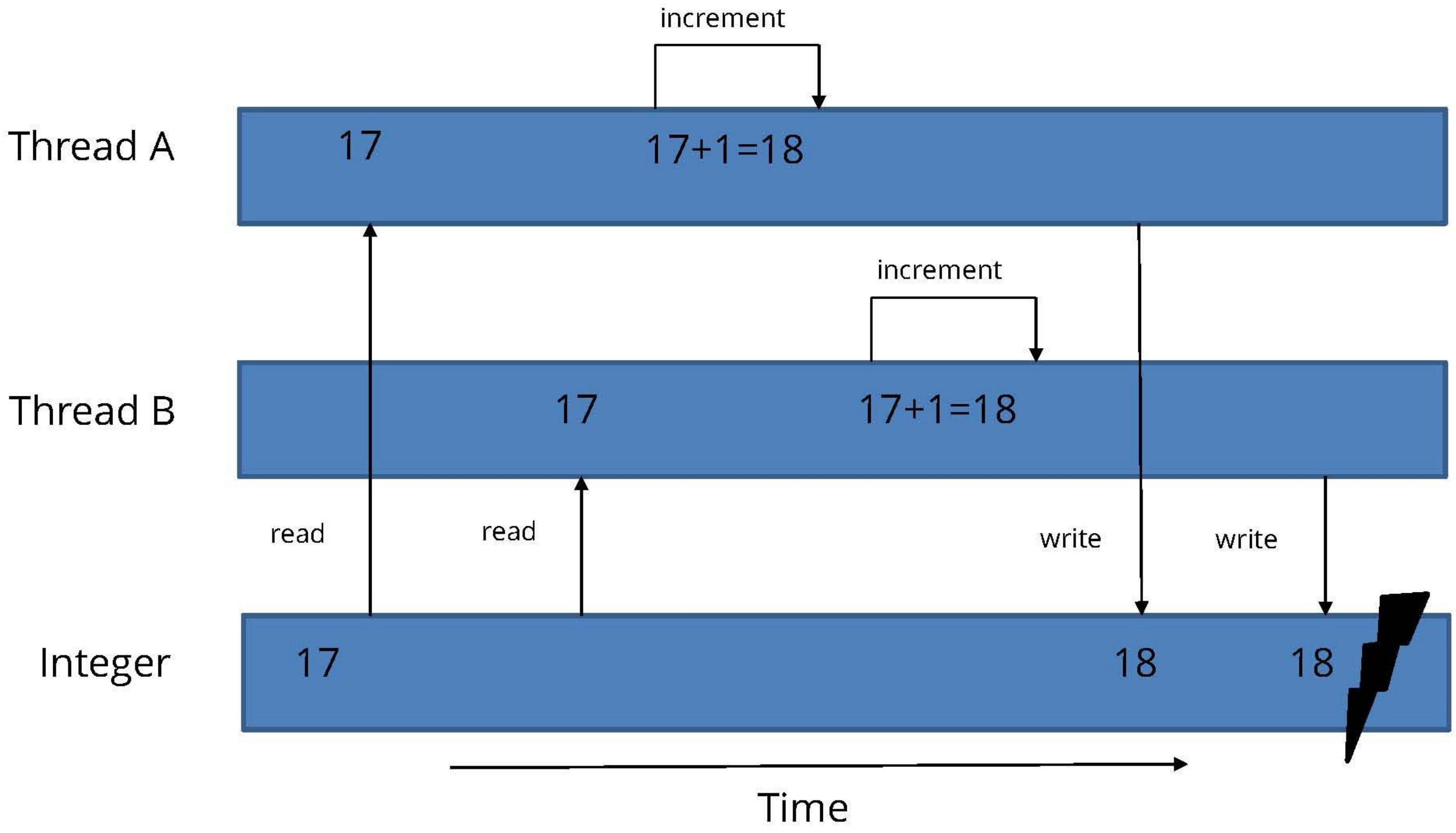
- **…**

```
    try {
        for(MyCountThread t:tg){
            t.join();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.printf("int: %s ,volatile: %s,
                Atomic: %s", m.cnt1,m.cnt2,m.cnt3);
}
```

Console output

```
int: 49793826 ,volatile: 99998132,Atomic: 100000000
```

Atomic Variables 5/5



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
- Deadlocks
- Threads pool
- The ReentrantLock class
- Synchronizers
- Atomic Variables
- **Concurrent Collection**
- The Fork-Join Framework

Synchronized Collections

java.util.Collections has methods that return synchronized collections:

- public static <T> Collection<T> synchronizedCollection(Collection<T> c),
- public static <T> List<T> synchronizedList(List<T> list),
- public static <T> Set<T> synchronizedSet(Set<T> s),
- public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m),
- public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s),
- public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)

Synchronized collections are static nested classes of the **Collections** class and have fields - a reference to the backed collection (parameter of the corresponding method from the above) and the object whose monitor is used for synchronization

Synchronized Collections

In java.util.Collections:

```
static class SynchronizedCollection<E>
                                    implements Collection<E>, Serializable {
…
    final Collection<E> c;      // Backing Collection
    final Object mutex;         // Object on which to synchronize
    SynchronizedCollection(Collection<E> c) {
        this.c = Objects.requireNonNull(c);
        mutex = this;
    }
…
    public boolean add(E e) {
        synchronized (mutex) {return c.add(e);}
    }
…
```

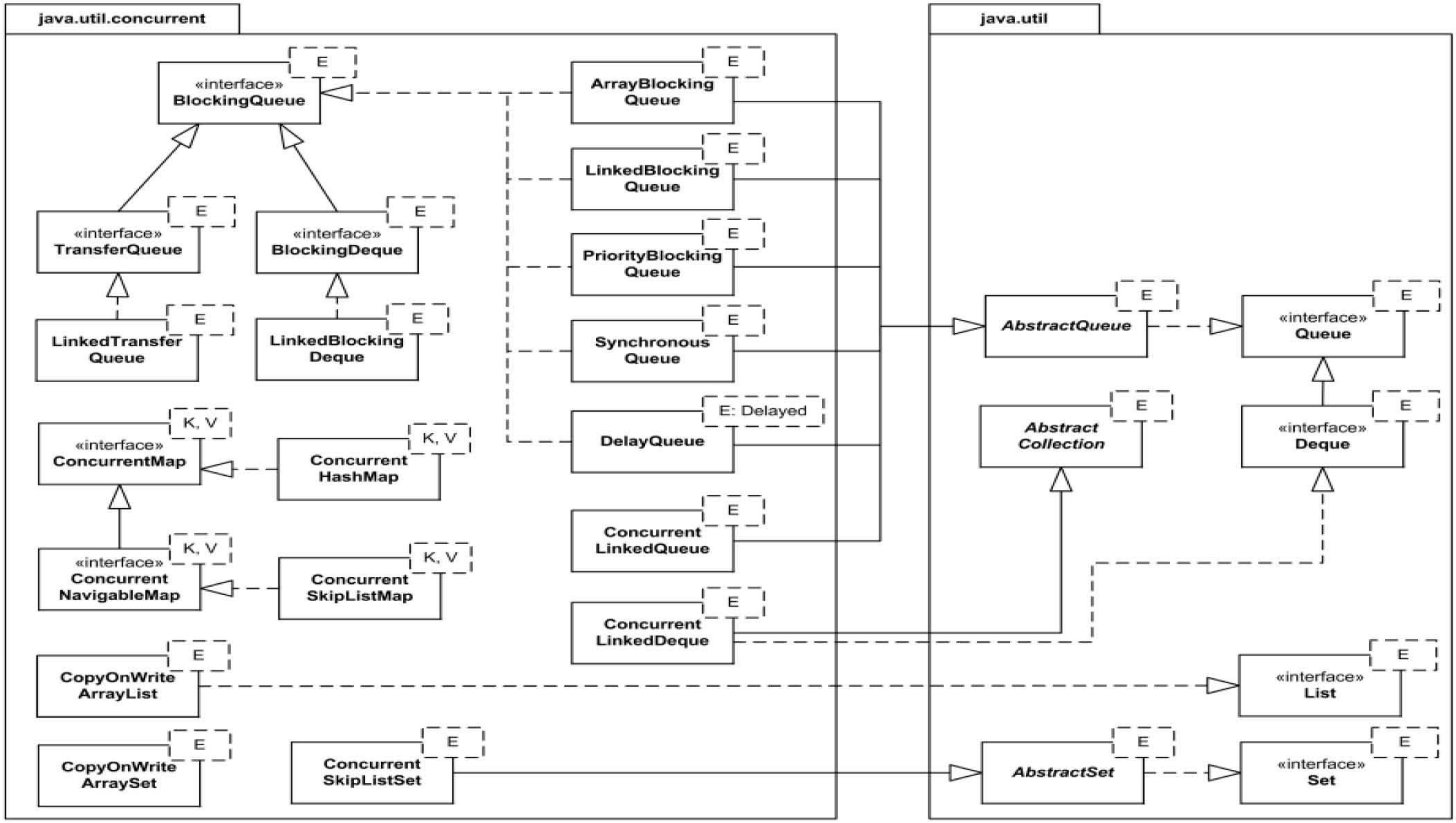
Synchronized collections are low efficiency

Concurrent Collections 1/16

- Concurrent, thread safe implementations of several collections
- Queues → ConcurrentLinkedQueue or one of the blocking queues
- HashMap → ConcurrentHashMap
- TreeMap → ConcurrentSkipListMap
- ArrayList → CopyOnWriteArrayList
- ArraySet → CopyOnWriteArraySet

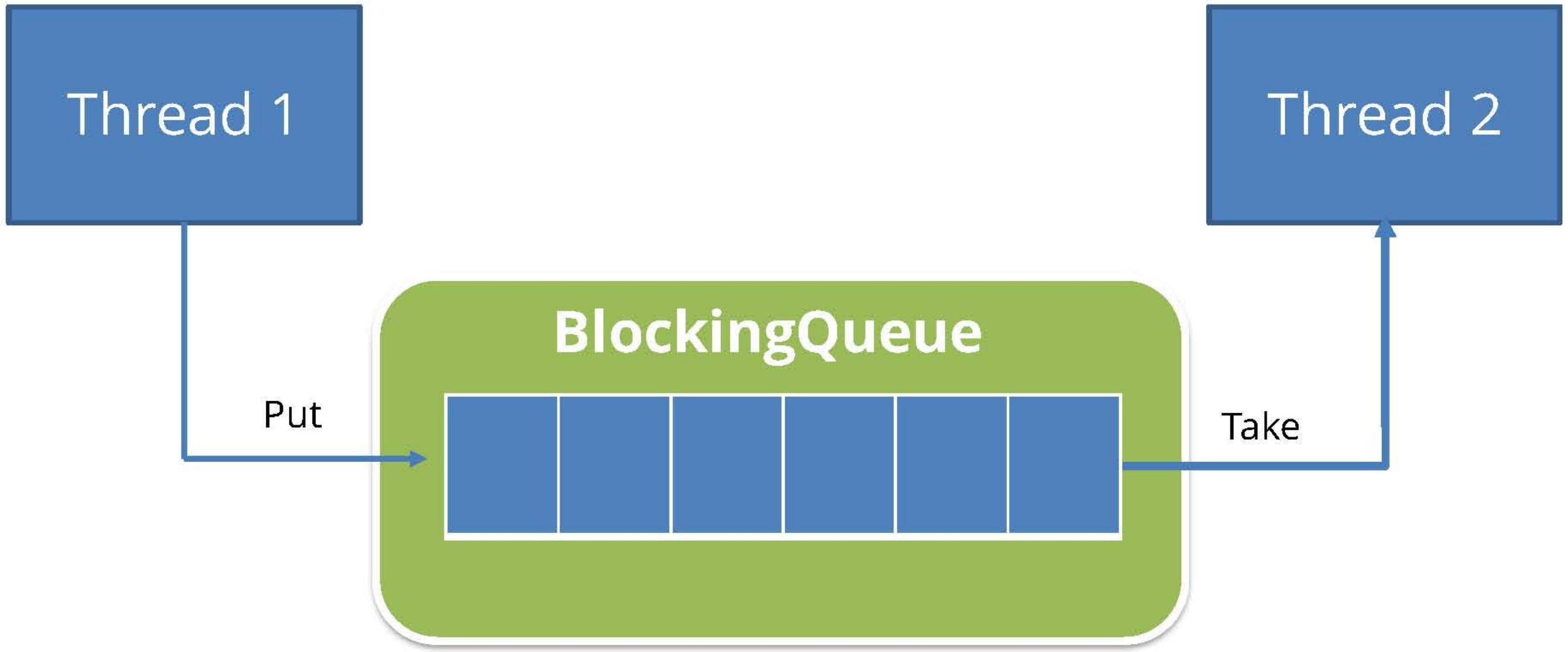
since Java 5

Concurrent Collections 2/16



Concurrent Collections 3/16

- The BlockingQueue



Concurrent Collections 4/16

```java
class Producer implements Runnable {
    private final BlockingQueue<Long> queue;
    private long i;
    Producer(BlockingQueue<Long> q) { queue = q; }
    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                queue.put(produce());
            Thread.sleep(1000);
            }
        } catch (InterruptedException ex) { }
    }
    Long produce() {
        return i++;
    }
}
```

Concurrent Collections 5/16

```java
class Consumer implements Runnable {
   private final BlockingQueue<Long> queue;
   Consumer(BlockingQueue<Long> q) {
      queue = q; }
   public void run() {
      try {
         while (!Thread.currentThread().isInterrupted()) {
         consume(queue.take()); }
      } catch (InterruptedException ex) { }
   }
   void consume(Long dt) {
      System.out.println(dt);
   }
}
```

Concurrent Collections 6/16

```
public static void main(String[] args){
BlockingQueue<Long> q = new ArrayBlockingQueue<>(10);
Producer p = new Producer(q);
Consumer c1 = new Consumer(q);
new Thread(p).start();
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
new Thread(c1).start();
```

Concurrent Collections 7/16

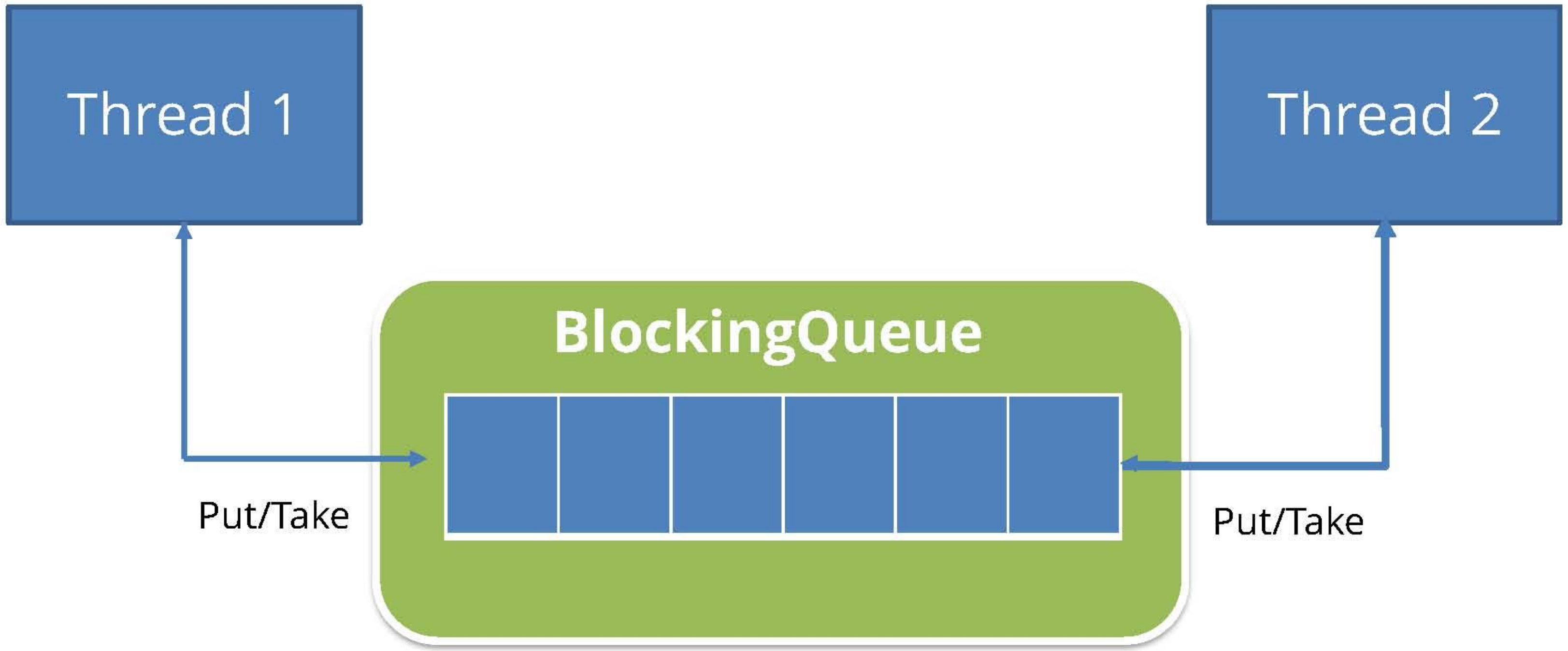
Console output

```
0
1
2
3
4
5
6
7
8
9
10
```

На самом деле вначале поставщиком в очередь поставляются элементы в течение 5 сек с 0 по 4, а затем запускается потребитель, потребляющие каждую секунду элементы с 0 по 10 и пока он потребляет с 0 по 4 элемент, поставщик успевает добавить в очередь оставшиеся с 5 по 10.

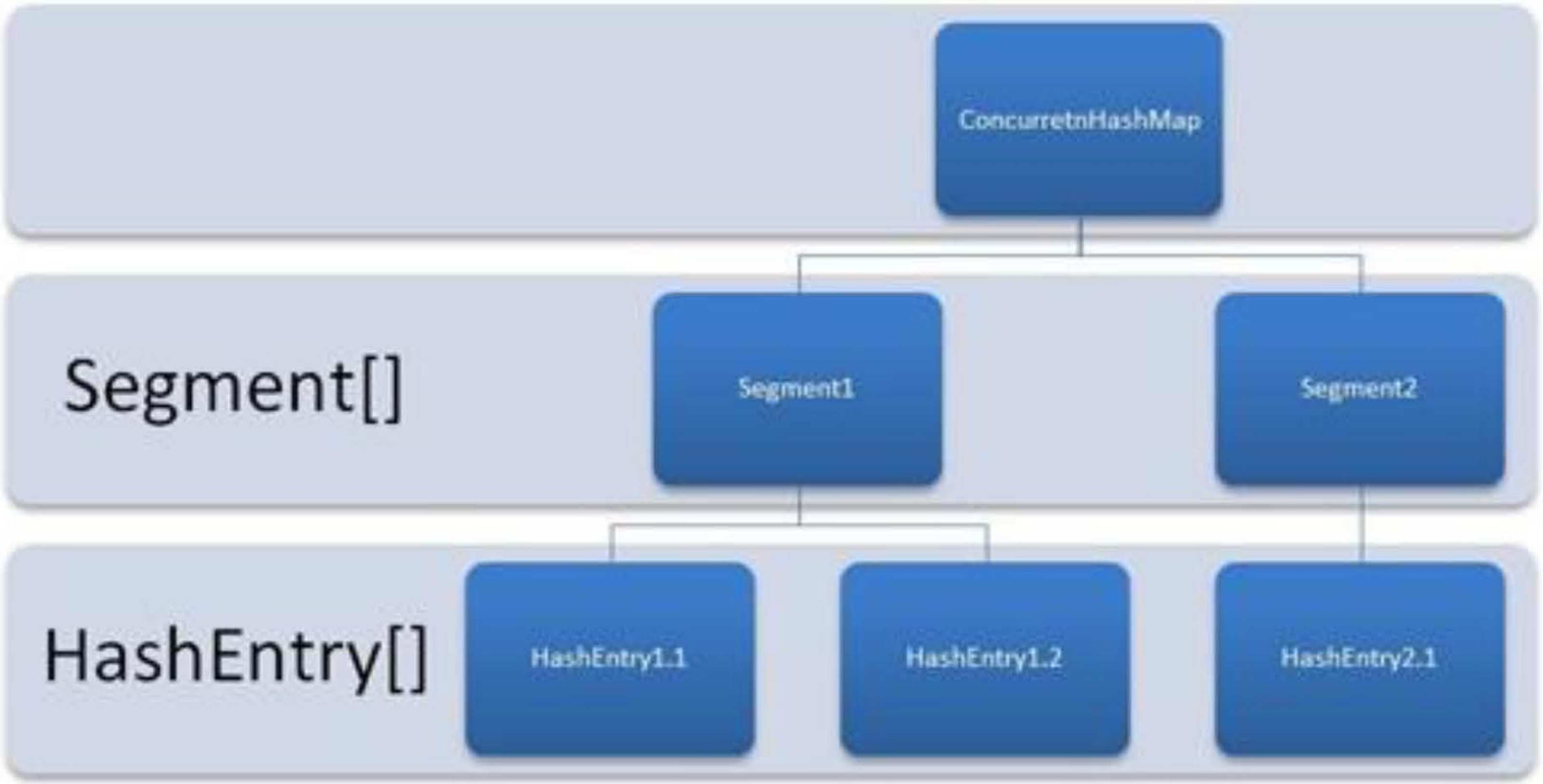
Concurrent Collections 8/16

- The BlockingDeque



Concurrent Collections 9/16

- The ConcurrentHashMap



Concurrent Collections 10/16

```
public class TestConcHashMap extends Thread{
   private String name;
   private static Map<String,String> cmap=new
ConcurrentHashMap<String,String>();
   TestConcHashMap(String name){
      this.name=name;
   }
   public void run() {
      cmap.put(name+"1","A");
      cmap.put(name+"2","B");
      cmap.put(name+"3","C");
      cmap.put(name+"4","D");
      cmap.put(name+"5","E");
      System.out.println(name +" completed.");
   }...
```

Concurrent Collections 11/16

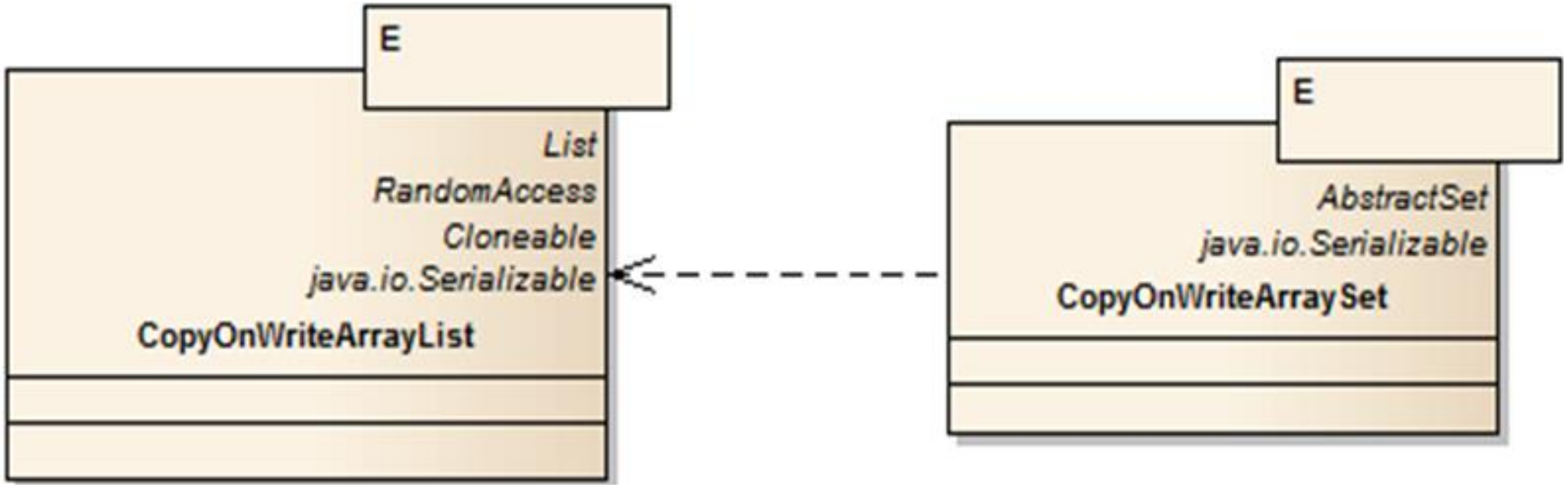
```
...
 public static void main(String[] args) {
      TestConcHashMap th1= new TestConcHashMap("One");
      TestConcHashMap th2= new TestConcHashMap("Two");
      th1.start(); th2.start();
      try {
         th1.join();  th2.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
      System.out.println(cmap);
   }
}
```

Concurrent Collections 12/16

<u>Console output</u>

```
One completed.
Two completed.
{Two1=A, Two3=C, One1=A, Two2=B, Two5=E, Two4=D, One3=C,
One2=B, One4=D, One5=E}
```

CopyOnWrite collections



CopyOnWriteArrayList<E> - additional methods:

int indexOf(E e, int index)
int lastIndexOf(E e, int index)
boolean addIfAbsent(E e)
int addAllAbsent(Collection<? extends E> c)

CopyOnWriteArraySet<E> - no additioanl methods

Concurrent Collections 13/16

```
public class TestCopOnWrArrLst {
        public static void main(String[] args){
            final CopyOnWriteArrayList<Integer> numbers =
new CopyOnWriteArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
            // new thread to concurrently modify the list
            new Thread(new Runnable() {
```

- ...

Concurrent Collections 14/16

- ...

```java
@Override
    public void run() {
        try {
            Thread.sleep(250);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        numbers.add(10);
        System.out.println("numbers:" + numbers);
    }
}).start();
```

here array is copied - see next slide

- ...

Concurrent Collections 15/16

- **...**

```
for (int i : numbers) {
            System.out.println(i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Concurrent Collections 16/16

Console output

```
1
2
3
numbers:[1, 2, 3, 4, 5, 10]
4
5
```

add(E e) method of CopyOnWriteArrayList

```
private transient volatile Object[] array;

public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        newElements[len] = e;
        setArray(newElements);
        return true;
    } finally {
        lock.unlock();
    }
}
```

Module contents

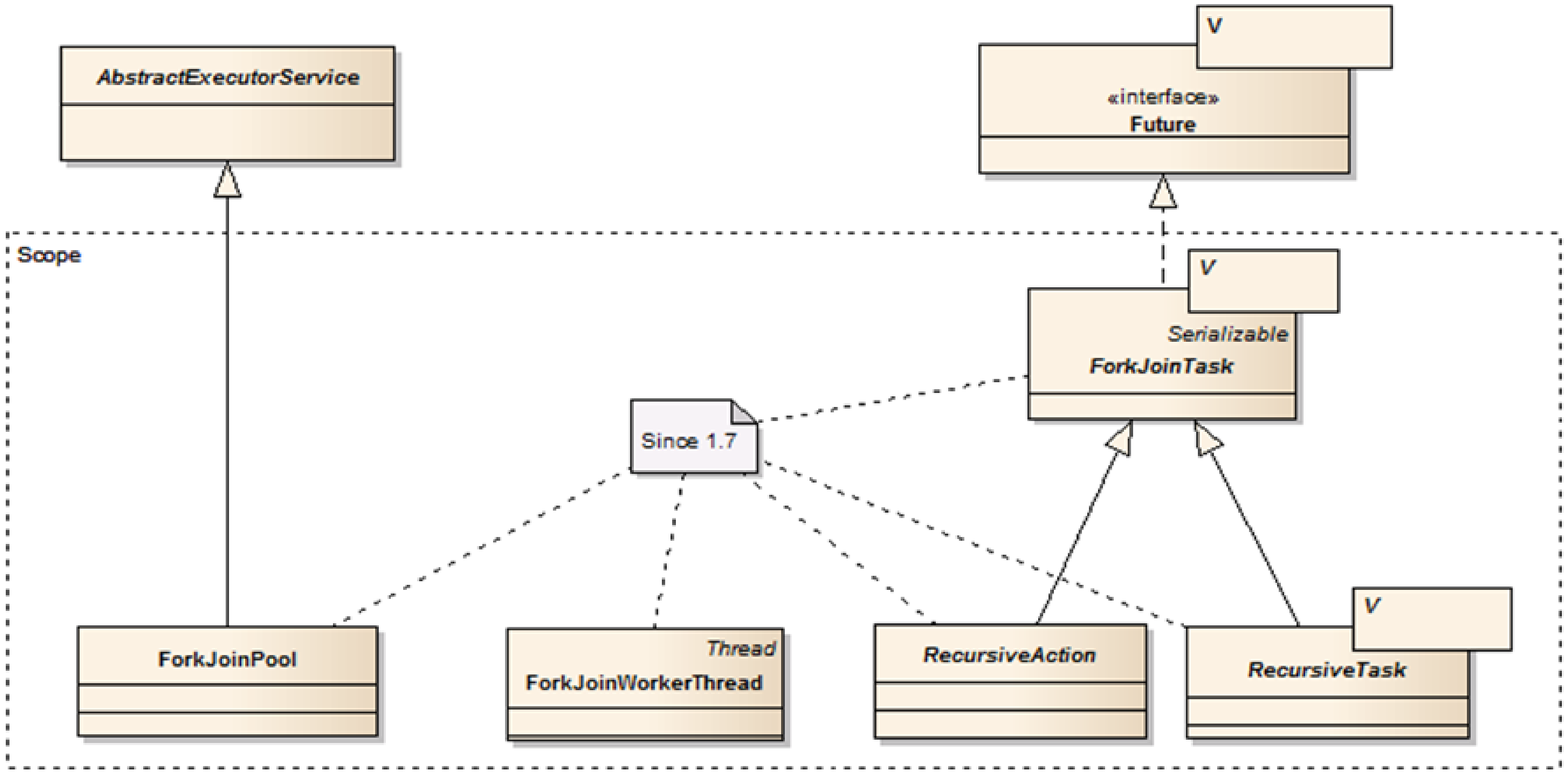
- Introduction to Concurrent Programming
- Creating Threads
- Important Methods of the Thread class
- Thread interruption. The interrupt() method
- The States of a Thread
- The Thread Scheduler. Thread Priority
- The Daemon Threads
- Thread Synchronization
- Synchronized Methods
- Synchronized Blocks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

The Fork-Join Framework 1/12

- The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors
- It is designed for work that can be broken into smaller pieces recursively.
- The goal is to use all the available processing power to enhance the performance of your application.
- The fork / join framework distributes tasks for worker threads in a thread pool. Worker threads that have completed their task can intercept (steal) tasks from other threads that are still busy.

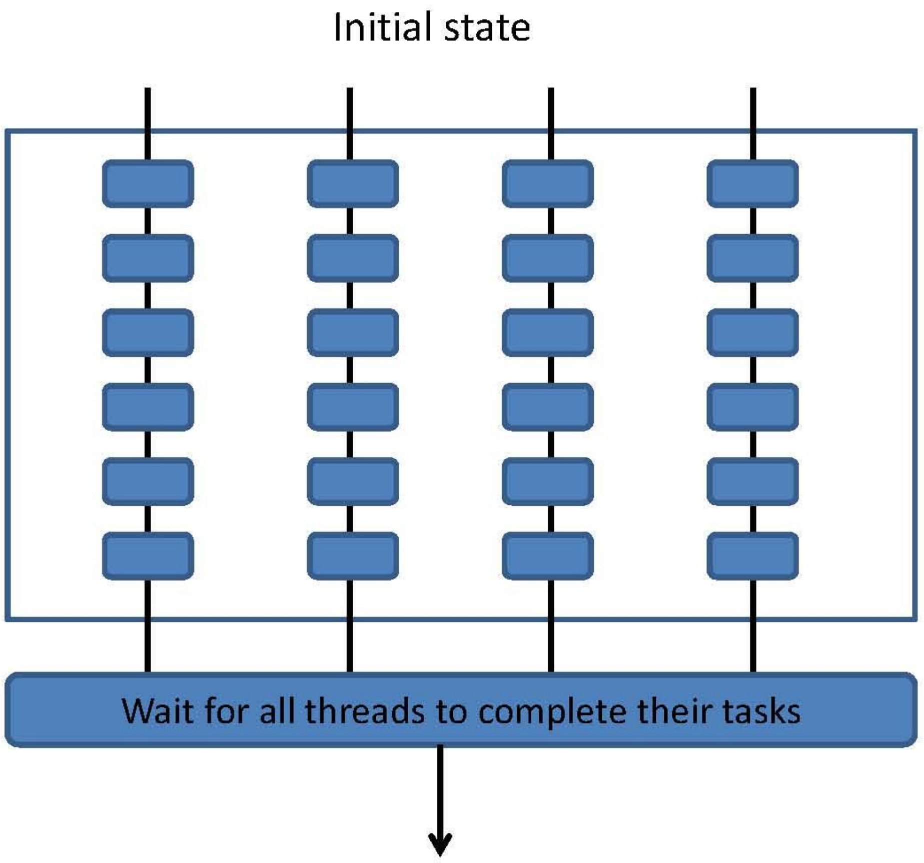
since Java 7

Fork-Join Framework



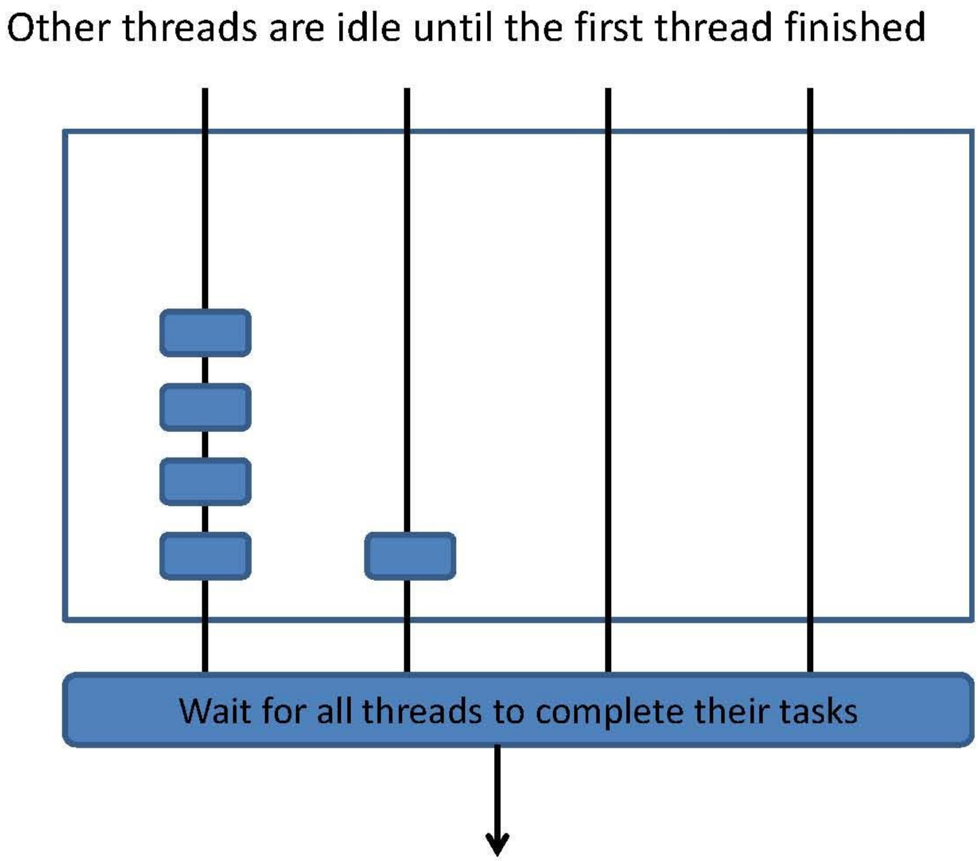
The Fork-Join Framework 2/12

- Work Stealing



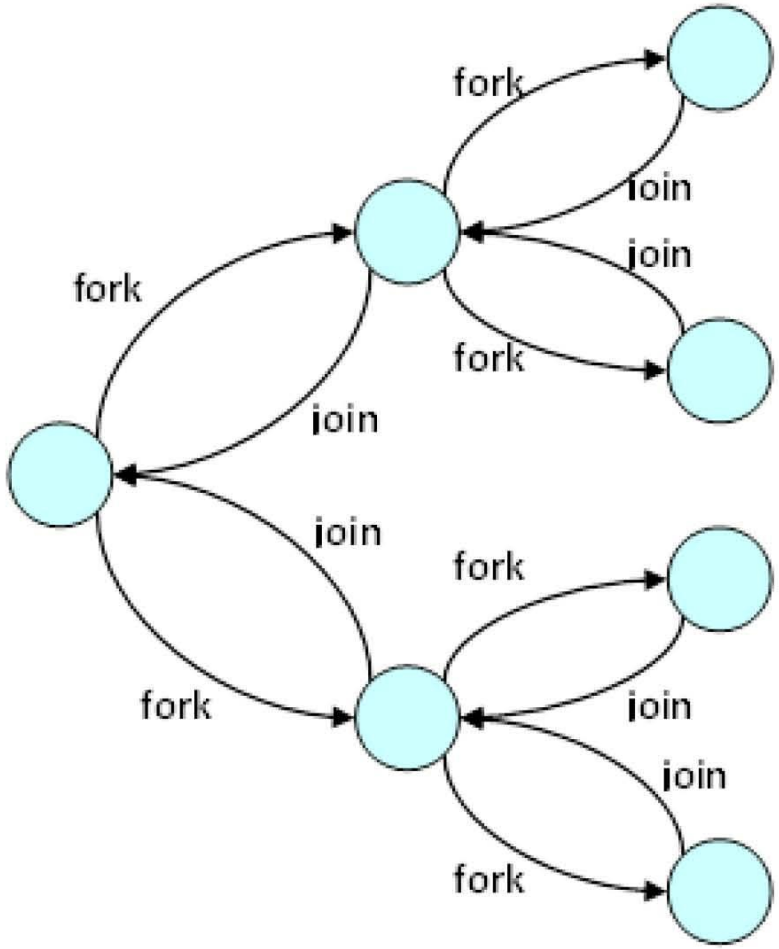
The Fork-Join Framework 3/12

- Work Stealing

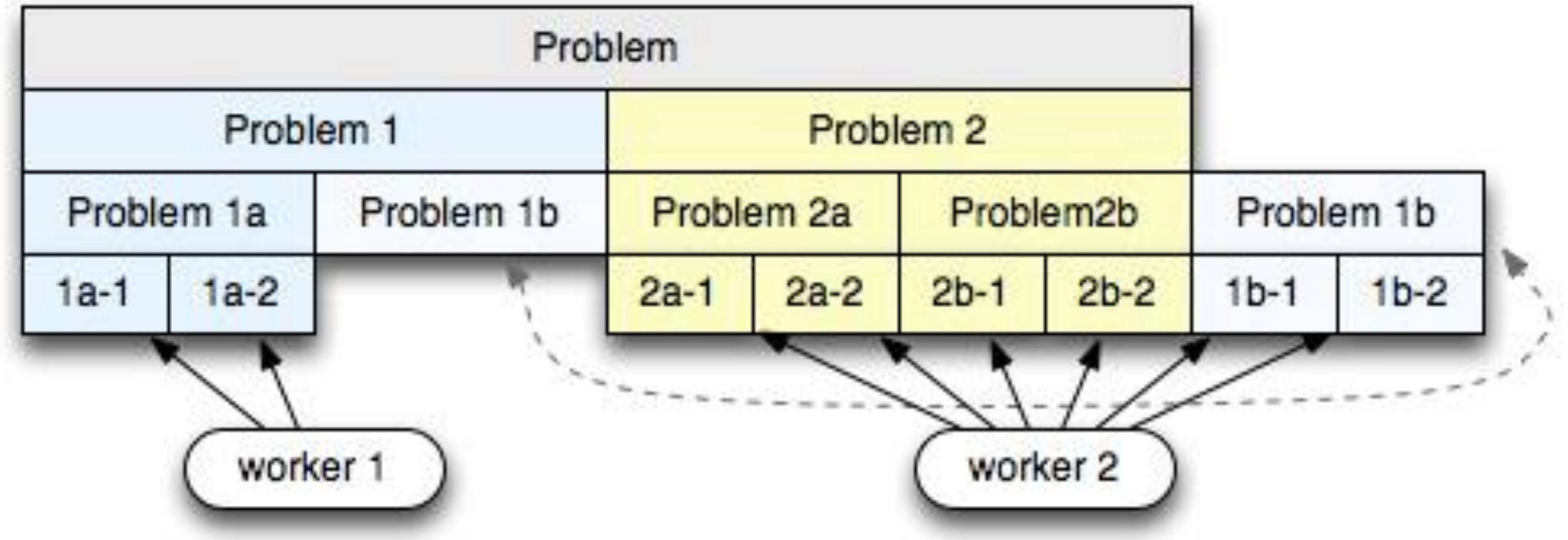

Other threads are idle until the first thread finished
Wait for all threads to complete their tasks

The Fork-Join Framework 4/12

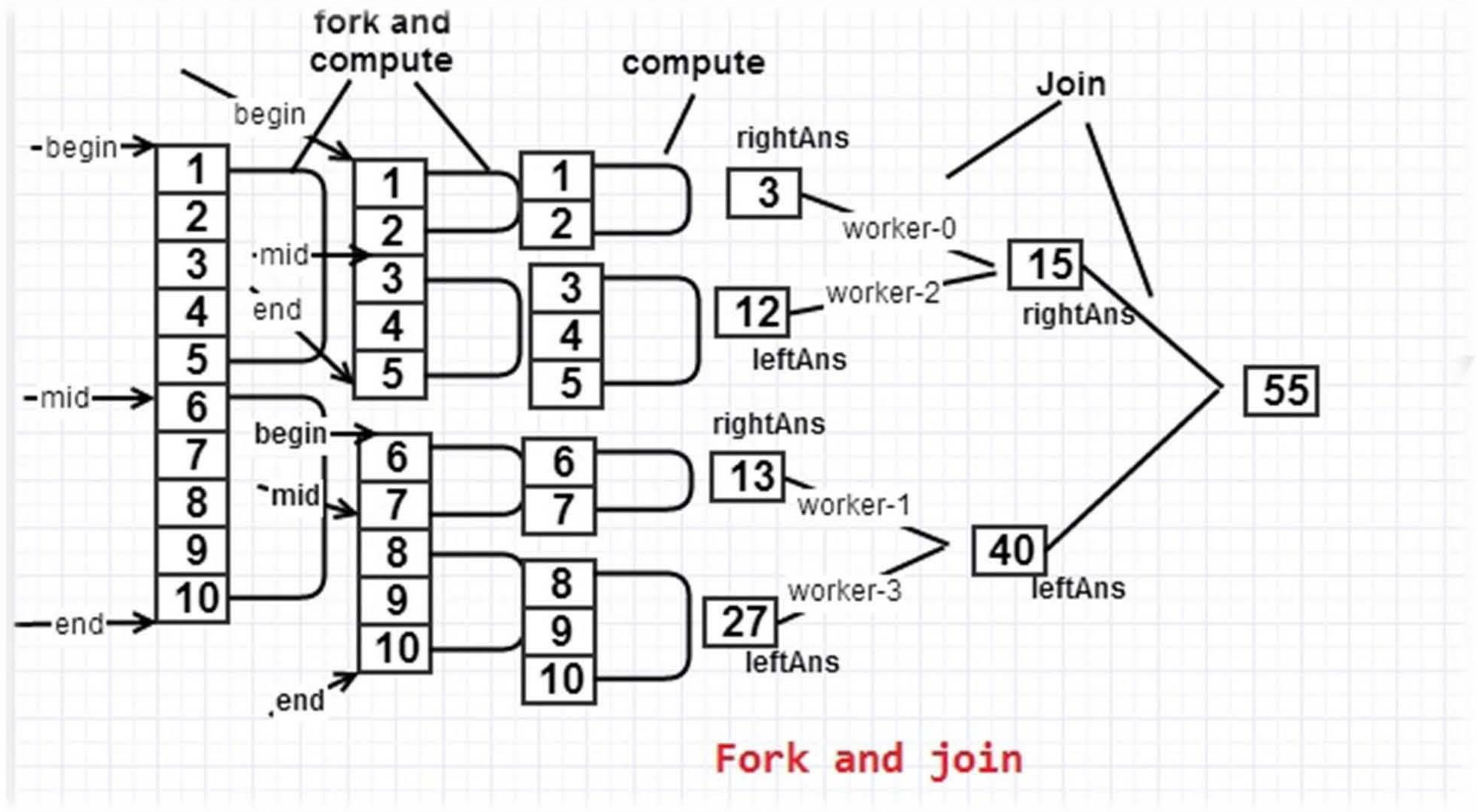
- The fork/join framework is distinct because it uses a *work-stealing* algorithm
- Worker threads that run out of things to do can steal tasks from other threads that are still busy
- The fork and join principle consists of two steps which are performed recursively. These two steps are the fork step and the join step.



The Fork-Join Framework 6/12



The Fork-Join Framework 7/12



The Fork-Join Framework 8/12

```java
public class SumOfNUsingForkJoin {
    private static long N = 1000_000L;
    private static final int NUM_THREADS = 10;
    static class RecSumOfN extends RecursiveTask<Long> {
        long from, to;
        public RecSumOfN(long from, long to) {
            this.from = from;
            this.to = to;
        }
```

The Fork-Join Framework 9/12

```
public Long compute() {
   if ((to - from) <= N / NUM_THREADS) {
      long localSum = 0;
      for (long i = from; i <= to; i++) {
         localSum += i;
      }
      System.out.printf("\t Summing of range %d to %d is %d %n",
            from, to, localSum);
      return localSum;
   } else {
      long mid = (from + to) / 2;
      System.out.printf("Forking into two ranges: " +
            "%d to %d and %d to %d %n", from, mid, mid, to);
      RecSumOfN firstHalf = new RecSumOfN(from, mid);
      firstHalf.fork();
      RecSumOfN secondHalf = new RecSumOfN(mid + 1, to);
      long resultSecond = secondHalf.compute();
      return firstHalf.join() + resultSecond;
   }
}
```

The Fork-Join Framework 10/12

```
 public static void main(String[] args) {
       ForkJoinPool pool = new ForkJoinPool(NUM_THREADS);
       long computedSum = pool.invoke(new RecSumOfN(0, N));
       long formulaSum = (N * (N + 1)) / 2;
       System.out.printf("Sum for range 1..%d; computed sum =
%d, formula sum = %d %n", N,computedSum, formulaSum);
   }
}
```

The Fork-Join Framework 11/12

- Console output

```
Forking computation into two ranges: 0 to 500000 and 500000 to 1000000
Forking computation into two ranges: 500001 to 750000 and 750000 to 1000000
Forking computation into two ranges: 750001 to 875000 and 875000 to 1000000
Forking computation into two ranges: 875001 to 937500 and 937500 to 1000000
Forking computation into two ranges: 0 to 250000 and 250000 to 500000
Forking computation into two ranges: 250001 to 375000 and 375000 to 500000
Forking computation into two ranges: 375001 to 437500 and 437500 to 500000
        Summing of range 937501 to 1000000 is 60546906250
        Summing of range 437501 to 500000 is 29296906250
        Summing of range 375001 to 437500 is 25390656250
Forking computation into two ranges: 250001 to 312500 and 312500 to 375000
Forking computation into two ranges: 0 to 125000 and 125000 to 250000
        Summing of range 875001 to 937500 is 56640656250
Forking computation into two ranges: 750001 to 812500 and 812500 to 875000
Forking computation into two ranges: 125001 to 187500 and 187500 to 250000
```

The Fork-Join Framework 12/12

```
 Summing of range 187501 to 250000 is 13671906250
          Summing of range 125001 to 187500 is 9765656250
          Summing of range 312501 to 375000 is 21484406250
          Summing of range 812501 to 875000 is 52734406250
Forking computation into two ranges: 0 to 62500 and 62500 to 125000
          Summing of range 62501 to 125000 is 5859406250
          Summing of range 0 to 62500 is 1953156250
Forking computation into two ranges: 500001 to 625000 and 625000 to 750000
Forking computation into two ranges: 625001 to 687500 and 687500 to 750000
          Summing of range 687501 to 750000 is 44921906250
          Summing of range 625001 to 687500 is 41015656250
Forking computation into two ranges: 500001 to 562500 and 562500 to 625000
          Summing of range 562501 to 625000 is 37109406250
          Summing of range 500001 to 562500 is 33203156250
          Summing of range 250001 to 312500 is 17578156250
          Summing of range 750001 to 812500 is 48828156250
Sum for range 1..1000000; computed sum = 500000500000, formula sum = 500000500000
```