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
- Semaphore
- Synchronizers
- Concurrent Collection
- The Fork-Join Framework

Module contents

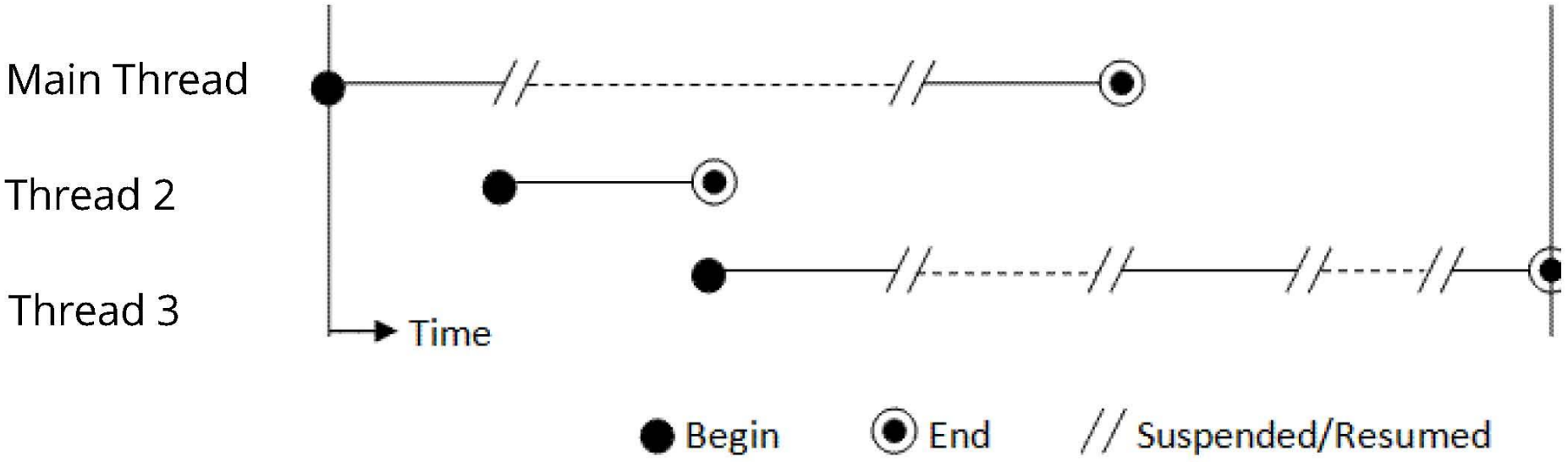
- **Introduction to Concurrent Programming**
- Creating Threads
- Important Methods in the Thread class
- Thread interruption. The interrupt() method
- The States of a Thread
- The Thread Scheduler. Thread Priority
- The Daemon Threads
- Thread Synchronization
- Synchronized Methods
- Synchronized Blocks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

Introduction to Concurrent Programming 1/3

- Java has been the first mainstream programming language to provide a first native support to concurrent programming
- – “conservative approach”: everything is still an object
- – + mechanisms for concurrency
- Extended with the java.util.concurrent library to provide a higher level support to concurrent programming Java 5 - 2004
- – semaphores, locks, synchronizers, etc
- – task frameworks

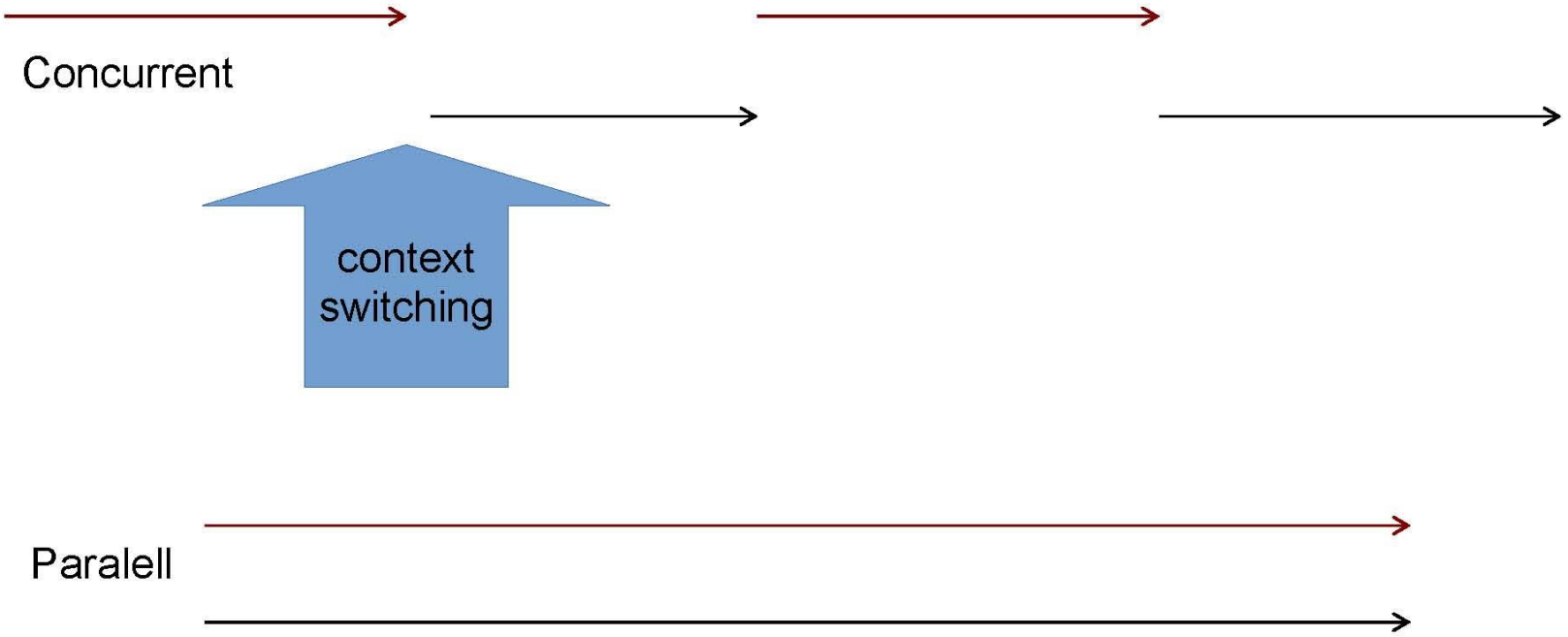
Introduction to Concurrent Programming 2/3

- Program with 3 threads running under a single CPU

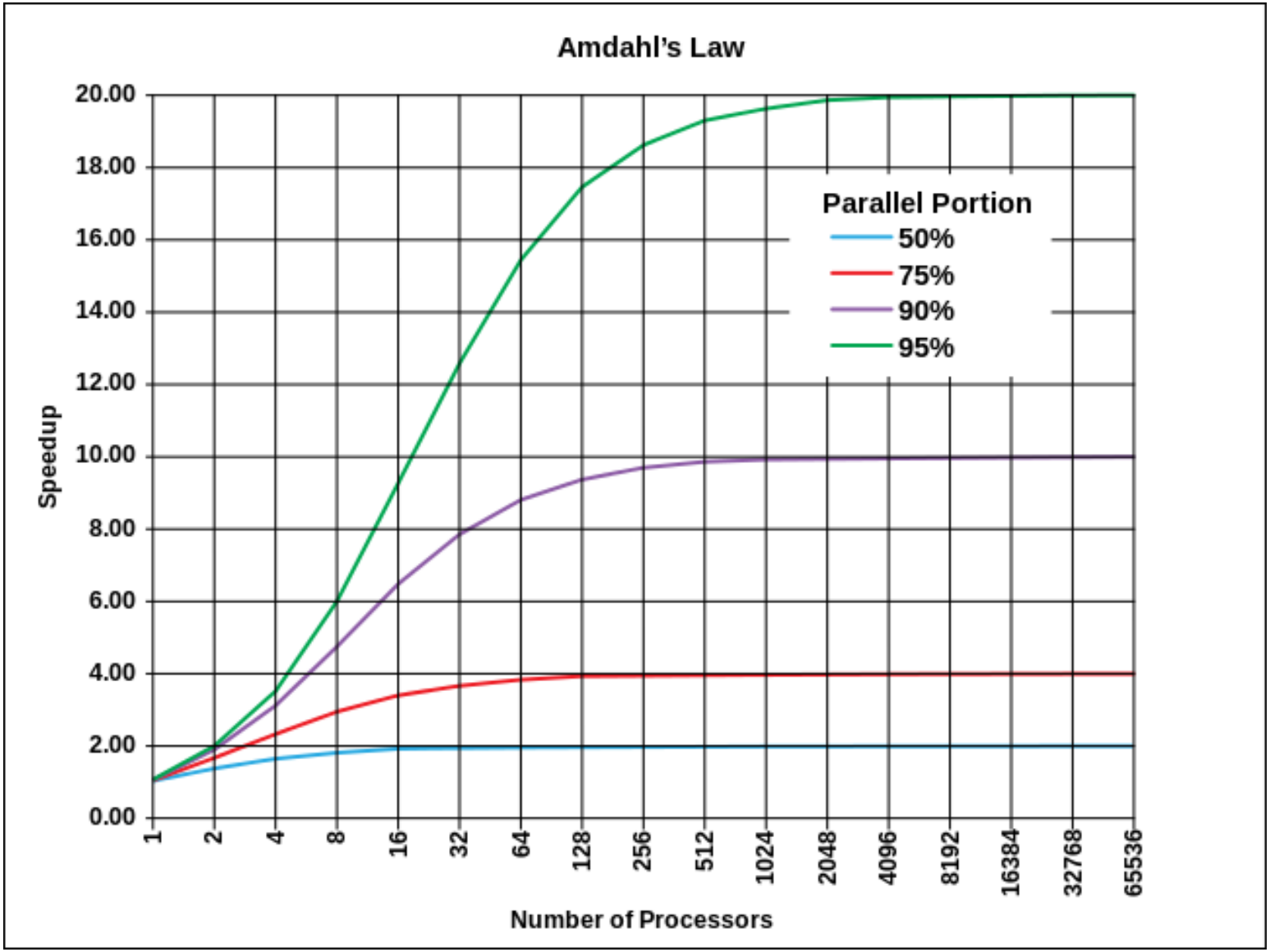


Introduction to Concurrent Programming 3/3

- Concurrency vs Parallelism



Amdahl's law


Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Parallel Portion
50%
75%
90%
95%
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

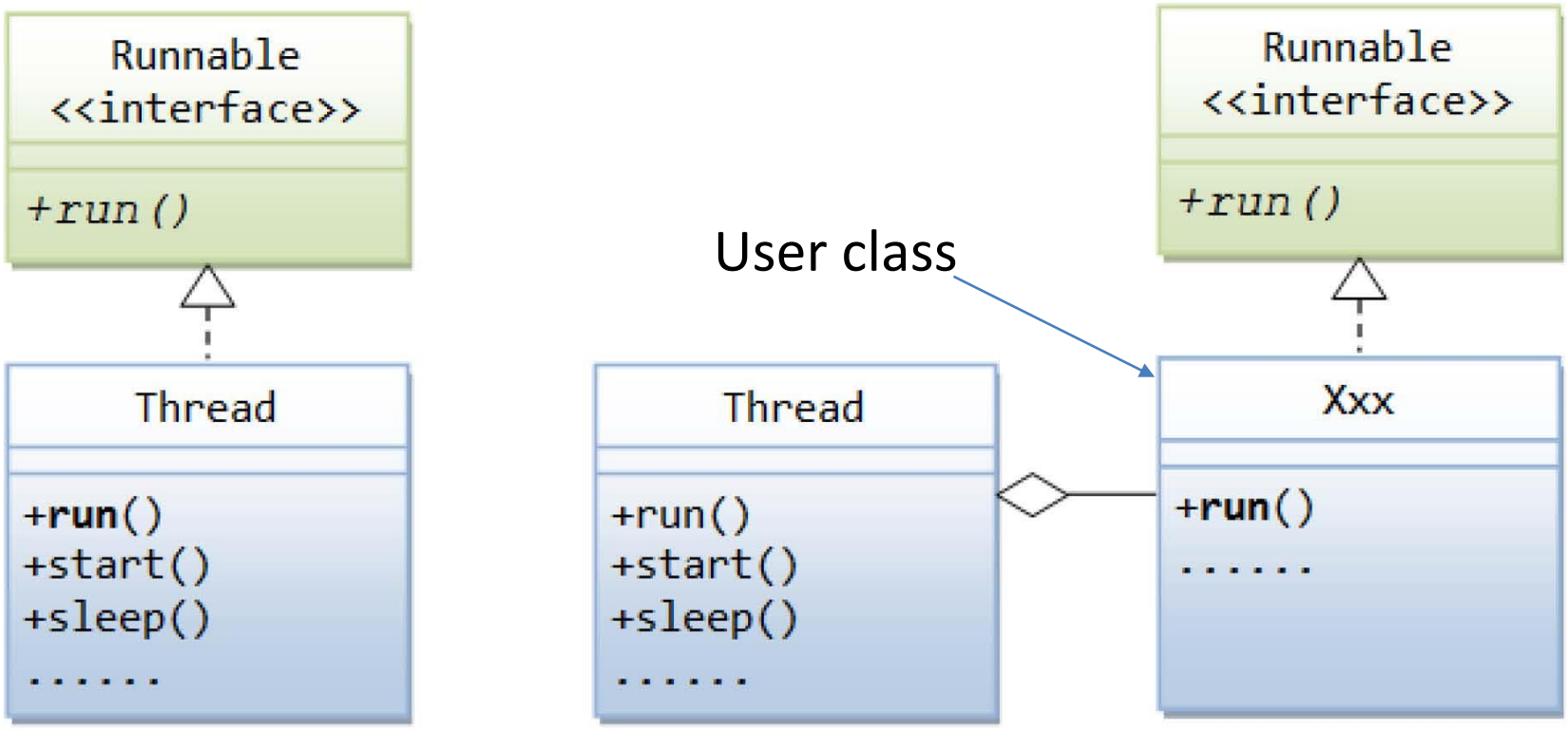
Module contents

- 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

Creating Threads 1/8

- You can define and instantiate a thread in one of two ways:
  - Implement the Runnable interface and pass it to Thread class constructor
  - Extend the java.lang.Thread class

Creating Threads 2/8



Creating Threads 5/8

- The Runnable interface declares a sole method, run()

```
1. public interface Runnable {
2.     public void run();
3. }
```

Runnable interface - is an abstraction of the task running in the thread and allows you to distinguish task execution from the logic of thread management

Creating Threads 6/8

```
1. public class MyTestRunnable implements Runnable {
2.     @Override
3.     public void run() {
4.         String name = Thread.currentThread().getName();
5.         for (int i = 0; i < 5; i++) {
6.             System.out.println("Thread:" + name + " i=" + i);
7.         }
8.     }
9. }
```

Thread.*currentThread*.getId()

Creating Threads 7/8

```
1.  public class Main {
2.      public static void main(String[] args) {
3.          System.out.println("main method start");
4.          MyTestRunnable run1 = new MyTestRunnable();
5.          Thread thr1 = new Thread(run1);
6.          System.out.println("thread created");
7.          thr1.start();
8.          System.out.println("thread started");
9.      }
10. }
```

run1.run(); - does not create new thread

Creating Threads 8/8

Console output

```
main method start
thread created
thread started
Thread:Thread-0 i=0
Thread:Thread-0 i=1
Thread:Thread-0 i=2
Thread:Thread-0 i=3
Thread:Thread-0 i=4
```

Creating Threads 3/8

```
1. public class MyTestThread extends Thread {
2.     @Override
3.     public void run() {
4.         for (int i = 0; i < 5; i++) {
5.             System.out.println("Thread:" +
6.             getName()+ " i="+i);
7.         }
8.     }
9. }
```

long getId();

Creating Threads 4/8

```
1. public class Main {
2.     public static void main(String[] args) {
3.         MyTestThread th1 = new MyTestThread();
4.         th1.start();
5.     }
6. }
```

th1.run(); - does not create new thread

Thread restart without creating new thread throws IllegalThreadStateException

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

Important methods in the Thread class 1/8

- **Pausing Execution with Sleep**
- Thread.sleep causes the current thread to suspend execution for a specified period
- Pause for 1 second: Thread.sleep(1000);
- Thread.sleep throws InterruptedException. This is an exception that sleep throws when another thread interrupts the current thread while sleep is active (calls the interrupt() method from the sleeping thread).

Important methods in the Thread class 2/8

```
1.  public class MyTestThread extends Thread {
2.      @Override
3.      public void run() {
4.          for (int i = 0; i < 5; i++) {
5.              System.out.println("Thread:" +
6.              getName()+ " i="+i);
7.              try {
8.                  sleep(1000);
9.              } catch (InterruptedException e) {
10.                 e.printStackTrace();
11.             }
12.         }
13.     }
14. }
```

Important methods in the Thread class 3/8

```
1.  public class MyTestThread extends Thread {
2.      @Override
3.      public void run() {
4.          for (int i = 0; i < 5; i++) {
5.              System.out.println("Thread:" +
6.              getName()+ " i="+i);
7.              try {
8.                  sleep(1000,100);
9.              } catch (InterruptedException e) {
10.                 e.printStackTrace();
11.             }
12.         }
13.     }
14. }
```

Important methods in the Thread class 4/8

- Using Thread's **Join()** Method

```
1.  System.out.println("main method start");
2.  MyTestRunnable run1 = new MyTestRunnable();
3.  Thread thr1 = new Thread(run1);
4.  thr1.start();
5.  System.out.println("thread started");
6.  try {
7.      thr1.join();
8.  } catch (InterruptedException e) {
9.      e.printStackTrace();
10. }
11. System.out.println("main method end");
```

Important methods in the Thread class 5/8

Console output

```
main method start
thread started
Thread:Thread-0 i=0
Thread:Thread-0 i=1
Thread:Thread-0 i=2
Thread:Thread-0 i=3
Thread:Thread-0 i=4
main method end
```

Important methods in the Thread class 6/8

- The **yield**( ) Method
- make the currently running thread head back to runnable to allow other threads of the same priority to get their turn

Important methods in the Thread class 7/8

```
public class ThreadYield {
    public static void main(String[] args) {
        Runnable r = () -> {
            int counter = 0;
            while(counter < 2){
                System.out.println(Thread.currentThread().getName());
                counter++;
                Thread.yield();
            }
        };
        new Thread(r).start();
        new Thread(r).start();
    }
}
```

Runnable is the Functional interface

Important methods in the Thread class 8/8

Output:

Thread-0
Thread-1
Thread-0
Thread-1

Thread-0
Thread-1
Thread-1
Thread-0

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

Thread work termination

```
public class MyThread implements Runnable {
    private boolean isActive;
    MyThread() {
        isActive = true;
    }
    void disable() {
        isActive = false;
    }
    public void run() {
        System.out.println(Thread.currentThread().getName() +" started");
        int counter = 1;
        while (isActive) {
            System.out.println("Loop " + counter++);
            try {Thread.sleep(400);} catch (InterruptedException e) {}
        }
System.out.println(Thread.currentThread().getName()  + " finished"); ï
```

Thread work termination

```
...
    public static void main(String[] args) {
        System.out.println("Main thread started");
        MyThread myThread = new MyThread();
        new Thread(myThread, "MyThread").start();
        try {
            Thread.sleep(1100);
            myThread.disable();
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted");
        }
        System.out.println("Main thread finished");
    }
}
```

Thread interaption. The interrupt() method 1/4

- **Interrupts**
- An *interrupt* is an indication to a thread that it should stop what it is doing and do something else.
- It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.
- For the interrupt mechanism to be used correctly, the thread to be interrupted must ensure that the Interrupt Status Flag value is checked in a loop by the **isInterrupted()** method or that its interrupt is processed, for example, by interrupting and handling an InterruptedException.

```
public class MyTestThread extends Thread{
    @Override
    public void run() {
        int i =0;
        while(!isInterrupted()){
            System.out.println("Thread " + getName()  + " i=" + i++);
        }       }
    public static void main(String[] args) {
        MyTestThread th1 = new MyTestThread();
        th1.start();
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        th1.interrupt();       }   }
```

Thread interaption. The interrupt() method 2/4

```
1.  public class MyTestThread extends Thread {
2.      @Override
3.      public void run() {
4.          int i = 0;
5.          while( true){
6.              System.out.println("Thread:" +
7.              getName()+ " i="+i++);
8.              try {
9.                  sleep(1000);
10.             } catch (InterruptedException e) {
11.                 return;
12.             }
13.         }
14.     }
15. }
```

Thread interaption. The interrupt() method 3/4

```
1.  public static void main(String[] args) {
2.      MyTestThread th1 = new MyTestThread();
3.      th1.start();
4.      try {
5.          Thread.sleep(5000);
6.      } catch (InterruptedException e) {
7.          e.printStackTrace();
8.      }
9.      th1.interrupt();
10. }
```

Thread interaption. The interrupt() method 4/4

Console output

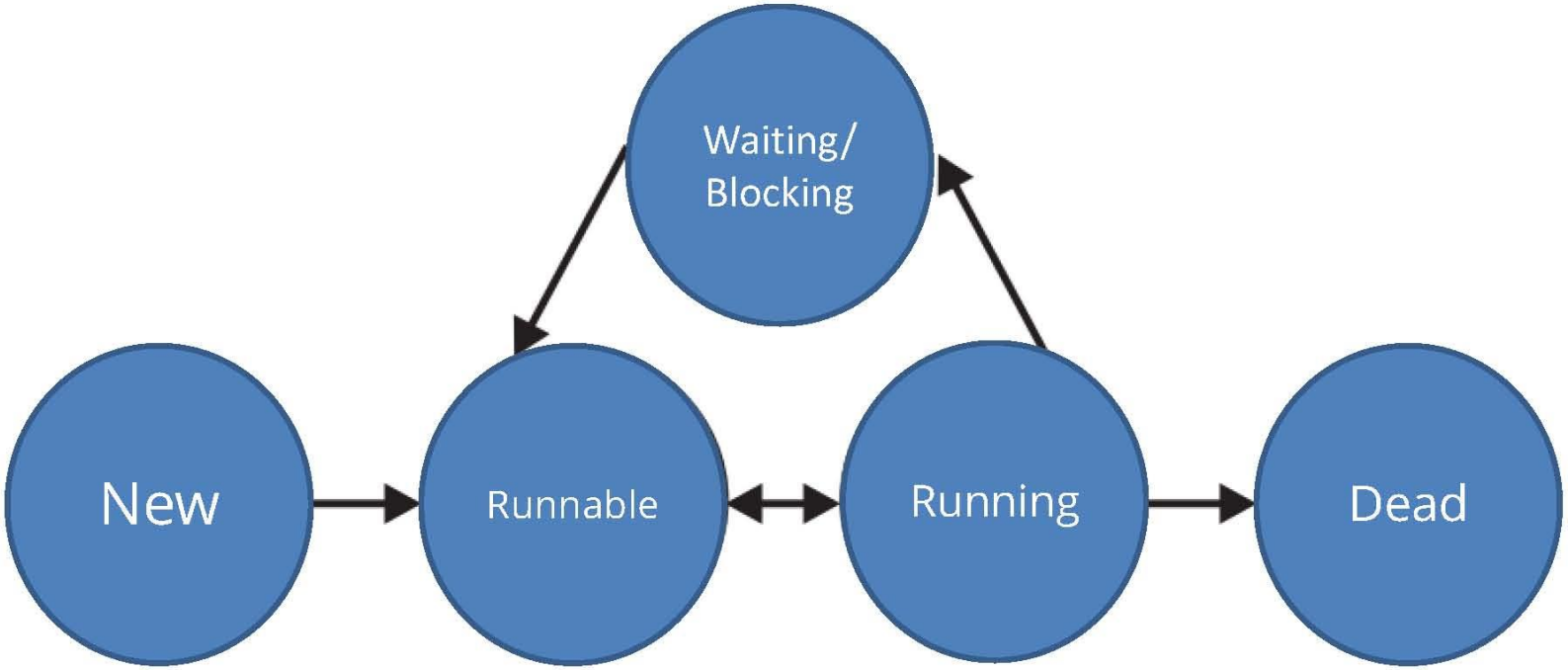
```
Thread:Thread-0 i=0
Thread:Thread-0 i=1
Thread:Thread-0 i=2
Thread:Thread-0 i=3
Thread:Thread-0 i=4
```

Module contents

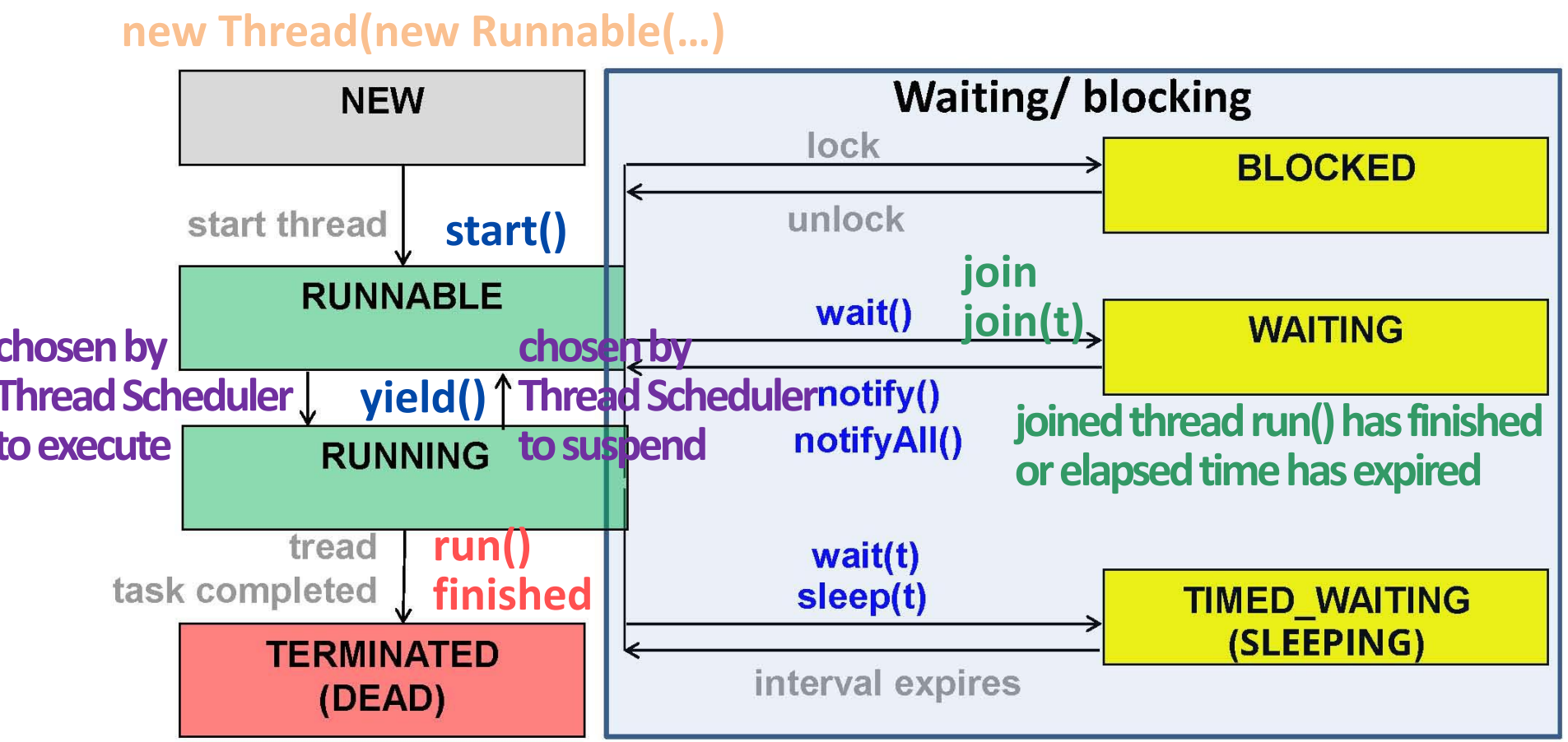
- Introduction to Concurrent Programming
- Creating Threads
- Important Methods of the Thread class
- Thread interruption. The interrupt() method
- **The States of a Thread**
- The Thread Scheduler. Thread Priority
- The Daemon Threads
- Thread Synchronization
- Synchronized Methods
- Synchronized Blocks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

The States of a Thread 1/2

- A thread can be only in one of five states



The States of a Thread 2/2



The States of a Thread

```
public class ThreadStatesTest extends Thread {
    @Override
    public void run() {
        try {
            System.out.println(getName() + " sleep(50)");
            Thread.sleep(50);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        System.out.println(getName() + " finished");
    }
    public static void main(String[] args) {
        try {
            Thread t = new ThreadStatesTest();
            System.out.println(t.getName() + " is created");
            …
```

The States of a Thread

```
...
        printInfo(1, t);
        System.out.println(t.getName() + " start()");
        t.start();
        printInfo(2, t);
        System.out.println(Thread.currentThread().
                           getName() + " sleep(10)");
        sleep(10);
        printInfo(3, t);
        /*joins main to t*/
        System.out.println(t.getName() + " t.join()");
        t.join();
        printInfo(4, t);
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
    System.out.println(Thread.currentThread().getName() + " finished");
} ...
```

The States of a Thread

```
...
    private static void printInfo(int count, Thread t) {
        System.out.println(String.valueOf(count) + ": "
                + t.getName() + ", State: " + t.getState()
                + ", isAlive=" + t.isAlive());
    }
}
```

Output:

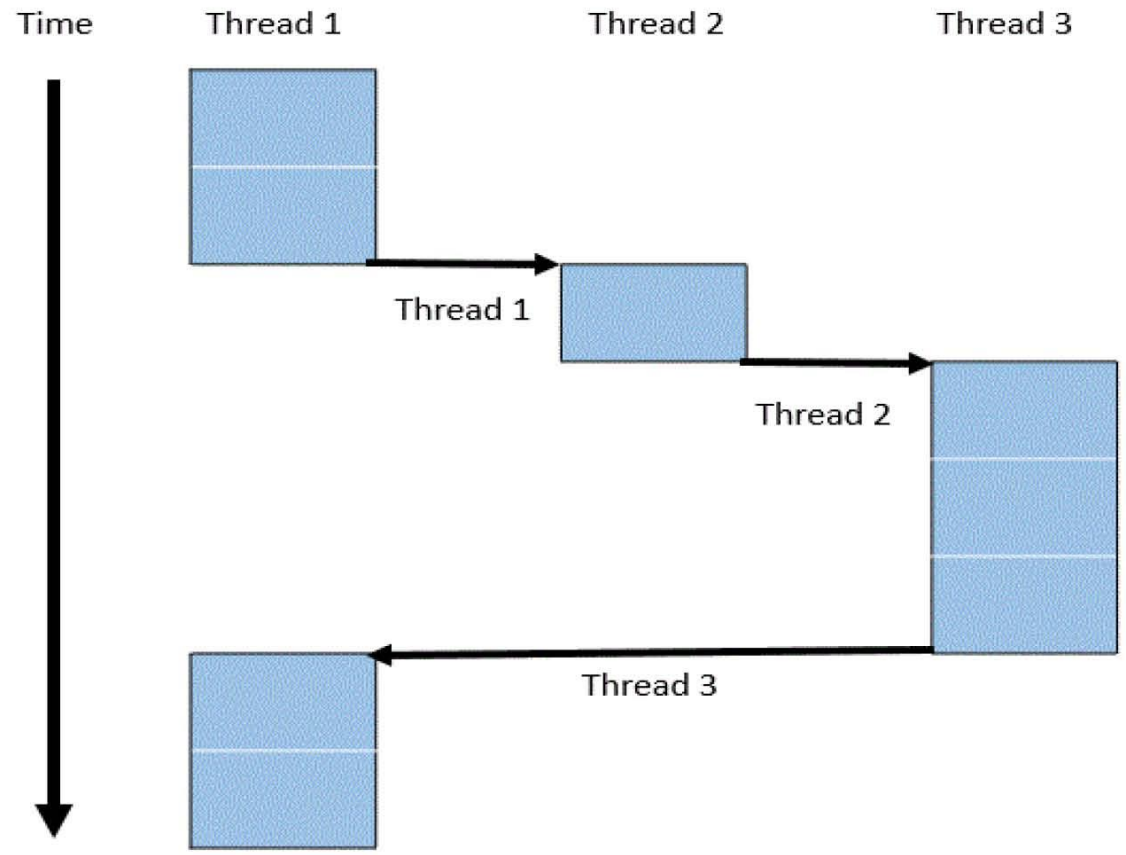
```
Thread-0 is created
1: Thread-0, State: NEW, isAlive=false
Thread-0 start()
2: Thread-0, State: RUNNABLE, isAlive=true
main sleep(10)
Thread-0 sleep(50)
3: Thread-0, State: TIMED_WAITING, isAlive=true
Thread-0 t.join()
Thread-0 finished
4: Thread-0, State: TERMINATED, isAlive=false
main finished
```

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 Thread Scheduler. Thread Priority 1/3

- The scheduler in most JVMs uses preemptive, priority-based scheduling



The Thread Scheduler. Thread Priority 2/3

- **Setting a Thread's Priority**

- The Thread class has the three following constants that define the range of thread priorities:
- Thread.MIN_PRIORITY (1)
- Thread.NORM_PRIORITY (5)
- Thread.MAX_PRIORITY (10)

The Thread Scheduler. Thread Priority 3/3

```
public class MyTestThread extends Thread {
   private double d;
   @Override
   public void run() {
      for (int i = 1; i < 10000000; i++) {        //heavy computational task
         d += (Math.PI + Math.E) / (double) i;
      }
      System.out.println("Thread :" + getName() +
                                                  ", Priority=" + getPriority());
   }
   public static void main(String[] args) {
      int numThreads = 8;  //must be even
      MyTestThread[] threads = new MyTestThread[numThreads];
      …
```

The Thread Scheduler. Thread Priority 3/3

```
…
    for (int i = 0; i < numThreads; i = i + 2) {
        threads[i] = createThread(Thread.MIN_PRIORITY);
        threads[i + 1] = createThread(Thread.MAX_PRIORITY);
    }
    for (MyTestThread thread : threads) {
        thread.start();
    }
  }
  private static MyTestThread createThread(int priority) {
    MyTestThread th = new MyTestThread();
    th.setPriority(priority);
    return th;
  }
}
```

The Thread Scheduler. Thread Priority 3/3

Output:

Thread :Thread-7, Priority=10
Thread :Thread-1, Priority=10
Thread :Thread-3, Priority=10
Thread :Thread-5, Priority=10
Thread :Thread-2, Priority=1
Thread :Thread-0, Priority=1
Thread :Thread-4, Priority=1
Thread :Thread-6, Priority=1

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 daemon threads 1/6

- There are two kinds of threads, *daemon threads* and *user threads*.

- The JVM exits when the only threads running are all daemon threads. In other words, the JVM considers its job done, when there is no more user threads and all the remaining threads are its infrastructure threads.

The daemon threads 2/6

```
1.  public class MyTestThread extends Thread {
2.      @Override
3.      public void run() {
4.          for (int i = 0; i < 5; i++) {
5.              System.out.println("Thread:" +
6.                      getName() + " i=" + i);
7.              try {
8.                  sleep(1000);
9.              } catch (InterruptedException e) {
10.             }
11.         }
12.     }
13. }
```

The daemon threads 3/6

```
1.  public static void main(String[] args) {
2.      MyTestThread myThread = new MyTestThread();
3.      myThread.start();
4.      try {
5.          Thread.sleep(2000);
6.      } catch (InterruptedException e) {
7.          e.printStackTrace();
8.      }
9.      System.out.println("method main() finished");
10. }
```

The daemon threads 4/6

Console output

```
Thread:Thread-0 i=0
Thread:Thread-0 i=1
method main() finished
Thread:Thread-0 i=2
Thread:Thread-0 i=3
Thread:Thread-0 i=4
```

The daemon threads 5/6

```
1.  public static void main(String[] args) {
2.      MyTestThread myThread = new MyTestThread();
3.      myThread.setDaemon(true);
4.      myThread.start();
5.      try {
6.          Thread.sleep(2000);
7.      } catch (InterruptedException e) {
8.          e.printStackTrace();
9.      }
10.     System.out.println("method main() finished");
11. }
```

The daemon threads 6/6

- After the thread is set to daemon

Console output

```
Thread:Thread-0 i=0
Thread:Thread-0 i=1
method main() finished
```

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

Thread Synchronization

- Thread Synchronization
- Threads communicate primarily by sharing access to fields and the objects reference fields refer to
- This form of communication is extremely efficient, but makes two kinds of errors possible: *thread interference* and *memory consistency errors*.
- The tool needed to prevent these errors is *synchronization*.

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 Methods 1/6

- To make a method **synchronized**, simply add the synchronized keyword to its declaration:

```
1. public synchronized void increment() {
2.     c++;
3. }
```

Synchronized Methods 2/6

- Not Synchronized

```
1. class MyCounter {
2.     private long cnt = 0;
3.     public void increment() {
4.         cnt++;
5.     }
6.     public long getValue() {
7.         return cnt;
8.     }
9. }
```

Synchronized Methods 3/6

```
1.  class MyCounterThread extends Thread{
2.      MyCounter m;
3.      int n;
4.      public MyCounterThread(MyCounter m,int n){
5.          this.m = m; this.n = n;
6.      }
7.      public void run(){
8.          for(int i=0;i<n;i++)
9.          {
10.             m.increment();
11.         }
12.     }
13. }
```

Synchronized Methods 4/6

```
1.  public static void main(String[] args) {
2.      MyCounter  m = new  MyCounter();
3.      MyCounterThread[] tg = new MyCounterThread[100];
4.      for(int i = 0 ; i < 100; i++){
5.          tg[i] = new MyCounterThread(m,1000000);
6.      }
7.      for(MyCounterThread t:tg){
8.          t.start();
9.      }
10.....
```

Synchronized Methods 5/6

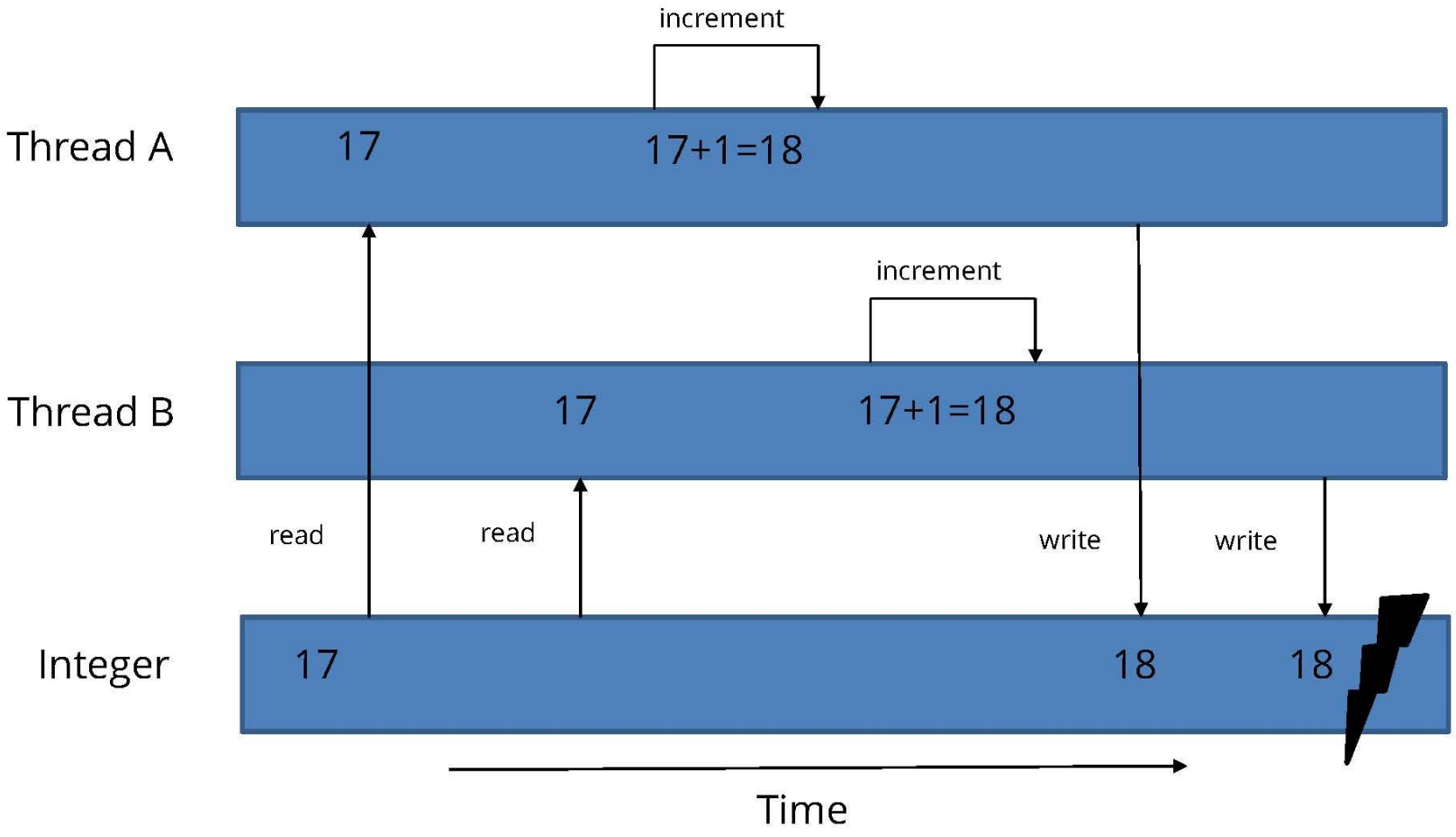
- ...

```
1.  try {
2.          for(MyCounterThread t:tg){
3.              t.join();
4.          }
5.      } catch (InterruptedException e) {
6.          e.printStackTrace();
7.      }
8.      System.out.println(m.getValue());
9.  }
10. }
```

Console output

85394622

Non-Synchronized increment



Synchronized Methods 6/6

- Synchronized

```
1. class MyCounter {
2.     private long cnt = 0;
3.     public synchronized void increment() {
4.         cnt++;
5.     }
6.     public synchronized long getValue() {
7.         return cnt;
8.     }
9. }
```

Console output

100000000

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 Blocks 1/7

- Synchronized blocks in Java are marked with the synchronized keyword.
- A synchronized block in Java is synchronized on some object.
- All synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time.
- All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Synchronized Blocks 2/7

- Synchronized Blocks (Statements)
- synchronized statements must specify the object that provides the intrinsic lock:

```
1. public void testSync() {
2.     synchronized(this) {
3.         //...
4.         myCount++;
5.     }
6. }
```

this is equivalent to

```
public synchronized void testSync() {
    //...
    myCount++;
}
```

Synchronized Blocks 3/7

```
1.  public class UserAccount {
2.      private int money;
3.      public UserAccount(int money) {
4.          this.money = money;
5.      }
6.      public int get() {
7.          return money;
8.      }
9.      public void set(int money) {
10.         this.money = money;
11.     }
12. }
```

Synchronized Blocks 4/7

```
1. class UserAction extends Thread {
2.     private UserAccount acc;
3.     private int withdraw;
4.     public UserAction(UserAccount acc, int withdraw) {
5.         this.acc = acc;
6.         this.withdraw = withdraw;
7.     }
•     ...
```

Synchronized Blocks 5/7

- ...

```
1.  public void run() {
2.      int has = acc.get();
3.      try {
4.          Thread.sleep(1);
5.      } catch (InterruptedException e) {
6.          e.printStackTrace();
7.      }
8.      if (has >= withdraw) {
9.          acc.set(acc.get() - withdraw);
10.     }
11. }
12. }
```

Synchronized Blocks 6/7

```
public static void main(String[] args) {
    /*Создаётся счёт с начальной суммой*/
    UserAccount acc = new UserAccount(500);
    for (int i = 0; i < 5; i++) {
        /*Создаются потоки, забирающие по 100 со счёта*/
        UserAction act = new UserAction(acc, 100);
        act.start();
    }
    try {
        Thread.sleep(1000);
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
    System.out.println("Баланс = " + acc.getMoney());
}
```

Output:

```
Get 100 from the account
Get 100 from the account
Get 100 from the account
Get 100 from the account
Get 100 from the account
Баланс = 200
```

Synchronized Blocks 7/7

- ...

```
1.  public void run() {
2.      synchronized (acc) {
3.          int has = acc.get();
4.          try {
5.              Thread.sleep(1);
6.          } catch (InterruptedException e) {
7.              e.printStackTrace();
8.          }
9.          if (has >= withdraw) {
10.             acc.set(acc.get() - withdraw);
11.         }
12.     }
```

Console output

0

Synchronized method vs Synchronized block

1. Synchronized block **reduce scope of lock**. As scope of lock is inversely proportional to performance, its always better to lock only critical section of code.
2. For synchronized block you can use **arbitrary any lock** to provide mutual exclusion to critical section code. On the other hand synchronized method always lock either on current object represented by this keyword or class level lock, if its static synchronized method.
3. Synchronized block can throw **NullPointerException** if expression provided to block as parameter evaluates to null, which is not the case with synchronized methods.

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 Wait/Notify Mechanism 1/10

- Threads often have to coordinate their actions. The most common coordination idiom is the *guarded block*. Such a block begins by polling a condition that must be true before the block can proceed. There are a number of steps to follow in order to do this correctly.

Guarded block example

```
public class GuardedLoop {
    private boolean joy;
    public void guardedJoy() {
        while (!joy) {              //guarded block
            System.out.println("Iterating...");
            try {  sleep(1000); } catch InterruptedException ex) {
            }
        }
        System.out.println("Joy has been achieved!");
    }
    public void setJoy(boolean joy) {
        this.joy = joy;
    }
}
```

Wastes processor time.
Don't do this!

Guarded block example

```
public class GuardedLoopThread extends Thread {
    GuardedLoop gl;
    public GuardedLoopThread(GuardedLoop gl) {
        this.gl = gl;
    }
    @Override
    public void run() {
        try {
            sleep(3000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        gl.setJoy(true);
    }
...
```

Guarded block example

```
...
   public static void main(String[] args) {
      GuardedLoop gl = new GuardedLoop();
      GuardedLoopThread loopThread =
                           new GuardedLoopThread(gl);
      loopThread.start();   //thread-deblocker start
      gl.guardedJoy();      //method with guard block start
   }
}
```

Output:

Iterating...
Iterating...
Iterating...
Joy has been achieved!

The Wait/Notify Mechanism 2/10

- Invoke Object.wait to suspend the current thread
- The invocation of wait does not return until another thread has issued a notification that some special event may have occurred — though not necessarily the event this thread is waiting for

The Wait/Notify Mechanism

```
public class GuardedLoop {
    private boolean joy;
    public synchronized void guardedJoy() {
        while (!joy) {
            System.out.println("Iterating...");
            try {
                wait();
            } catch (InterruptedException ex) {
            }
        }
        System.out.println("Joy and efficiency has been achieved!");
    }
    public synchronized void notifyJoy() {
        joy = true;
        notify();
    }   }
```

The Wait/Notify Mechanism

```
public class GuardedLoopThread extends Thread {
    GuardedLoop gl;
    public GuardedLoopThread(GuardedLoop gl) {
        this.gl = gl;
    }
    @Override
    public void run() {
        try {
            sleep(3000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        gl.notifyJoy();
    }
...
```

The Wait/Notify Mechanism

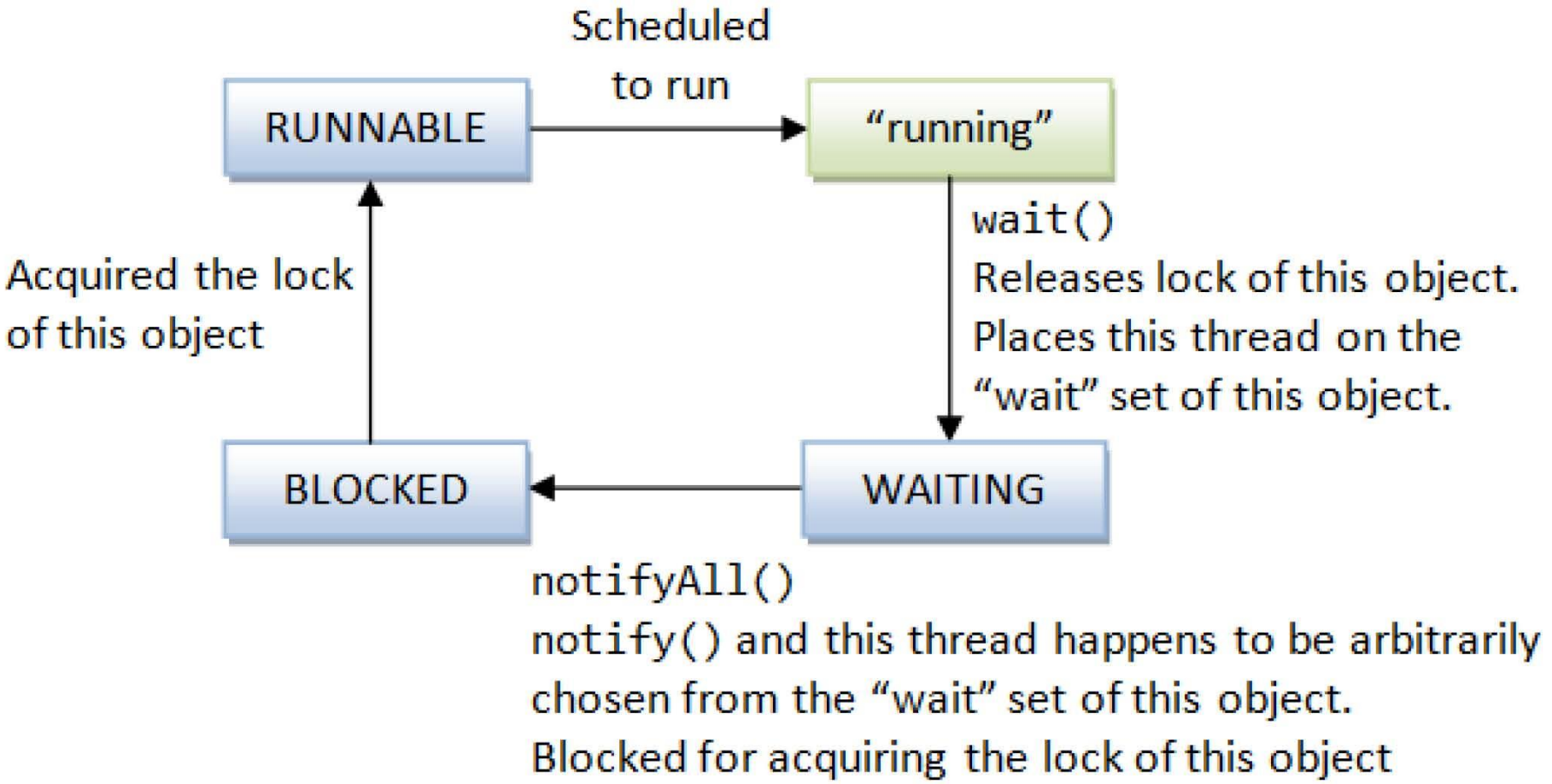
```
...
   public static void main(String[] args) {
      GuardedLoop gl = new GuardedLoop();
      GuardedLoopThread loopThread =
                        new GuardedLoopThread(gl);
      loopThread.start();    //запуск потока-разблокировщика
      gl.guardedJoy();       //запуск метода с защищенным блоком
   }
}
```

Output:

Iterating...

Joy and efficiency has been achieved!

The Wait/Notify Mechanism 3/10



The Wait/Notify Mechanism 4/10

```
1. class MyTestData {
2.     private byte[] data;
3.     public void setData(byte[] data) {
4.         this.data = data;
5.     }
6.     public boolean ready() {
7.         return data != null;
8.     }
9. }
```

The Wait/Notify Mechanism 5/10

```
1.  class DataGenerator extends Thread {
2.      MyTestData dat;
3.      public DataGenerator(MyTestData dat) {
4.          this.dat = dat;
5.      }
6.      public void run() {
7.          System.out.print("Generating Data....");
8.          try {
9.              Thread.sleep(1000);
10.         } catch (InterruptedException e) {
11.             e.printStackTrace();
12.         }
```

- ...

The Wait/Notify Mechanism 6/10

- ...

```
1.  byte[] data = new byte[1000];
2.          new Random().nextBytes(data);
3.          System.out.println("OK!!!");
4.          synchronized (dat) {
5.              dat.setData(data);
6.              dat.notifyAll();
7.          }
8.      }
9.  }
```

The Wait/Notify Mechanism 7/10

```
1. class DataSender extends Thread {
2.     MyTestData data;
3.     String user;
4.     public DataSender(MyTestData doc, String user){
5.         this.data = doc;
6.         this.user = user;
7.     }
```

- ...

The Wait/Notify Mechanism 8/10

```
1.  public void run() {
2.      System.out.println("Waiting for Data #" +
3.              + getId() + "...");
4.      synchronized (data) {
5.          try {
6.              while (!data.ready()){
7.                  data.wait();
8.              }
9.          } catch (InterruptedException e) {
10.             return;
11.         }
12.     }
13.     System.out.printf("Sending data to %s\r\n",user);
14. }
```

The Wait/Notify Mechanism 9/10

```
1.  public static void main(String[] args) {
2.      MyTestData data = new MyTestData();
3.      DataSender[] senders = {
4.              new DataSender(data, "user1"),
5.              new DataSender(data, "user2"),
6.              new DataSender(data, "user3"),
7.      };
8.      for (DataSender sender : senders)
9.          sender.start();
10.     DataGenerator pr = new DataGenerator(data);
11.     pr.start();
12. }
```

The Wait/Notify Mechanism 10/10

Console output

Waiting for Data #9...
Waiting for Data #11...
Waiting for Data #10...
Generating Data....
OK!!!
Sending data to user2
Sending data to user1
Sending data to user3

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 volatile keyword

Declaring a block of code (or method) synchronized has two important implications, commonly referred to as **atomicity** and **visibility**.

Atomicity means that only one thread can execute code protected by a given object-monitor (lock) at a time, preventing collisions of threads during the update of a state that is accessible from many threads.

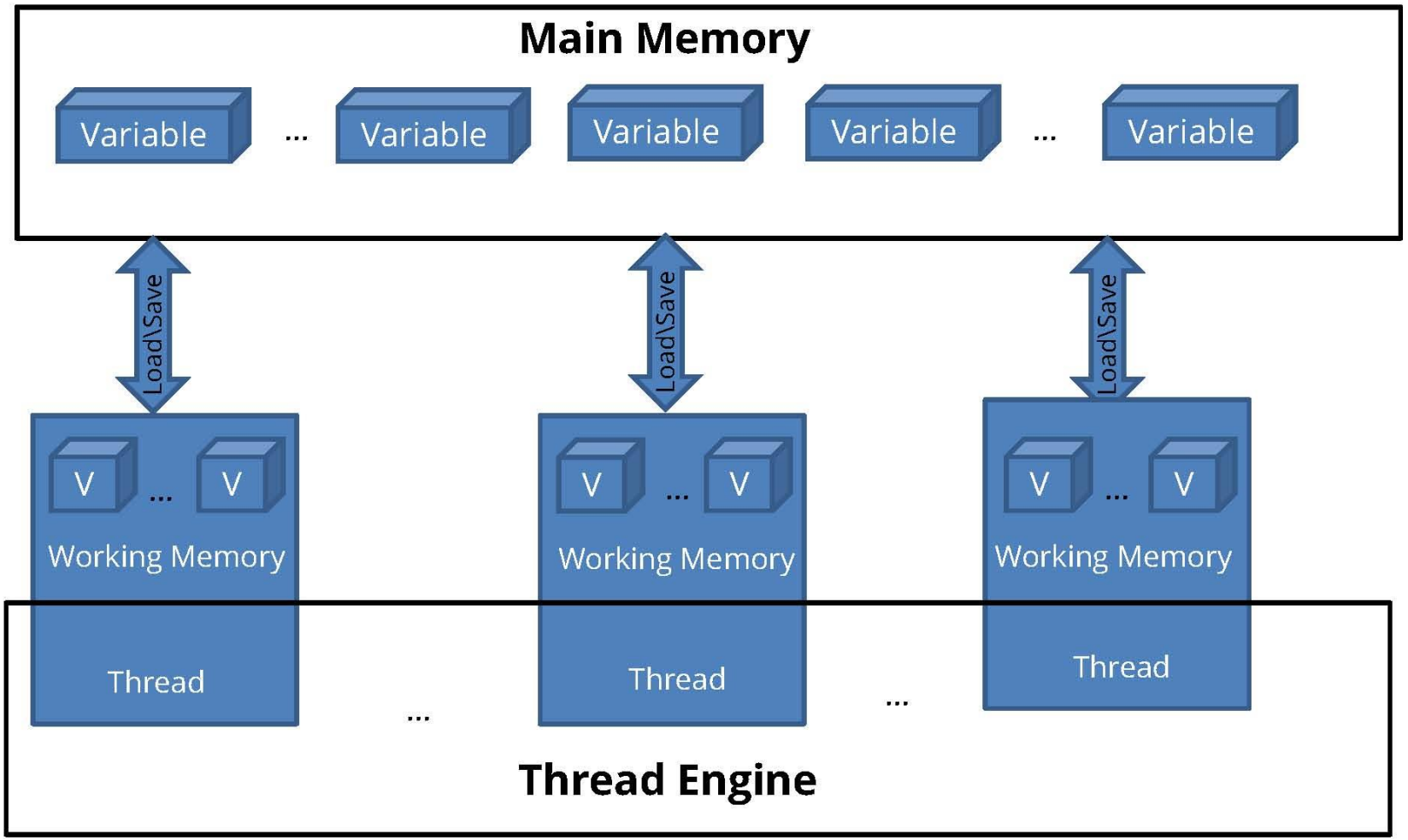
Visibility is related to the features of memory caching and program optimization during compilation. If the developer used synchronization, it will be checked at runtime that variable updates performed by one thread before exiting the synchronized block will be immediately visible to another thread when it enters the synchronized block protected by the same monitor (lock).

The volatile keyword

The **volatile** keyword only applies to variables and has the following effects in multithreaded programming:

1) the variable is always read from the main memory, and is never cached into the thread's memory, which means it is always available to any thread;
2) for read and write requests from multiple threads, the system guarantees that the write requests are first executed;
3) the atomicity of read/write operations is guaranteed, although this is relevant only for variables of type long and double, for other types these actions are already atomic. For all other operations like ++, synchronization is done externally, or atomic types are used like AtomicInteger from the java.util.concurrent.atomic package (will be considered later).

The volatile keyword 1/4



The volatile keyword 2/6

```
public class VolatileTest {

//    private static volatile int myInt = 0;
    private static int myInt = 0;

    public static void main(String[] args) {
        Thread listener = new ChangeListener();
        Thread changer = new ChangeMaker();
        listener.start();
        changer.start();
    }
```

...

The volatile keyword 3/6

```
…
static class ChangeMaker extends Thread {
    @Override
    public void run() {
        int localValue = myInt ;
        while (localValue < 5) {
            myInt = ++ localValue;
            System.out.printf("Incrementing myInt to
                                    %d%n", localValue);
            try {
                Thread.sleep(500);
             } catch (InterruptedException e) {
                e.printStackTrace();
             }
    }   } }     }
```

The volatile keyword 4/6

```
…
static class ChangeListener extends Thread {
    @Override
    public void run() {
        int localValue = myInt ;
        while (localValue < 5) {
            if (localValue != myInt) {
                System.out.printf("Got Change for myInt :
                                        %d%n", myInt );
                localValue= myInt ;
            }
        }
    }
}
```

The volatile keyword 5/6

for non-volatile myInt:

Thread ChangeListener started
Thread ChangeMaker started
Incrementing myInt to 1
Got Change for myInt : 1
Incrementing myInt to 2
Incrementing myInt to 3
Incrementing myInt to 4
Incrementing myInt to 5
BUILD STOPPED (total time: 5 seconds)

hangs in a loop

The volatile keyword 6/6

for volatile myInt:

```
Thread ChangeMaker started
Thread ChangeListener started
Incrementing myInt to 1
Incrementing myInt to 2
Got Change for myInt : 2
Incrementing myInt to 3
Got Change for myInt : 3
Got Change for myInt : 4
Incrementing myInt to 4
Got Change for myInt : 5
Incrementing myInt to 5
BUILD SUCCESSFUL
```

volatile vs synchronized

Synchronized can guarantee both *visibility* and *atomicity*, and **volatile** variables only guarantee *visibility*.

You can use volatile variables instead of synchronized only under limited circumstances. For volatile variables, both of the following criteria must be met to ensure the desired thread safety:

1) write in the variable do not depend on its current value;
2) the variable does not participate in invariants with other variables (does not depend on other variables).

volatile variable as Status Flag

```
public class StatusFlagTest extends Thread {
   boolean keepRunning = true;
//    volatile boolean keepRunning = true;
   @Override
   public void run() {
      while (keepRunning) {
      }
      System.out.println("Thread terminated.");
   }
   public static void main(String[] args) throws InterruptedException {
      StatusFlagTest t = new StatusFlagTest();
      t.start();
      Thread.sleep(1000);
      t.keepRunning = false;
      System.out.println("keepRunning set to false.");
   }    }
```

volatile variable as Status Flag

boolean keepRunning

Output:

keepRunning set to false.

hangs in a loop

BUILD STOPPED

volatile boolean keepRunning

Output:

keepRunning set to false.

Thread terminated.

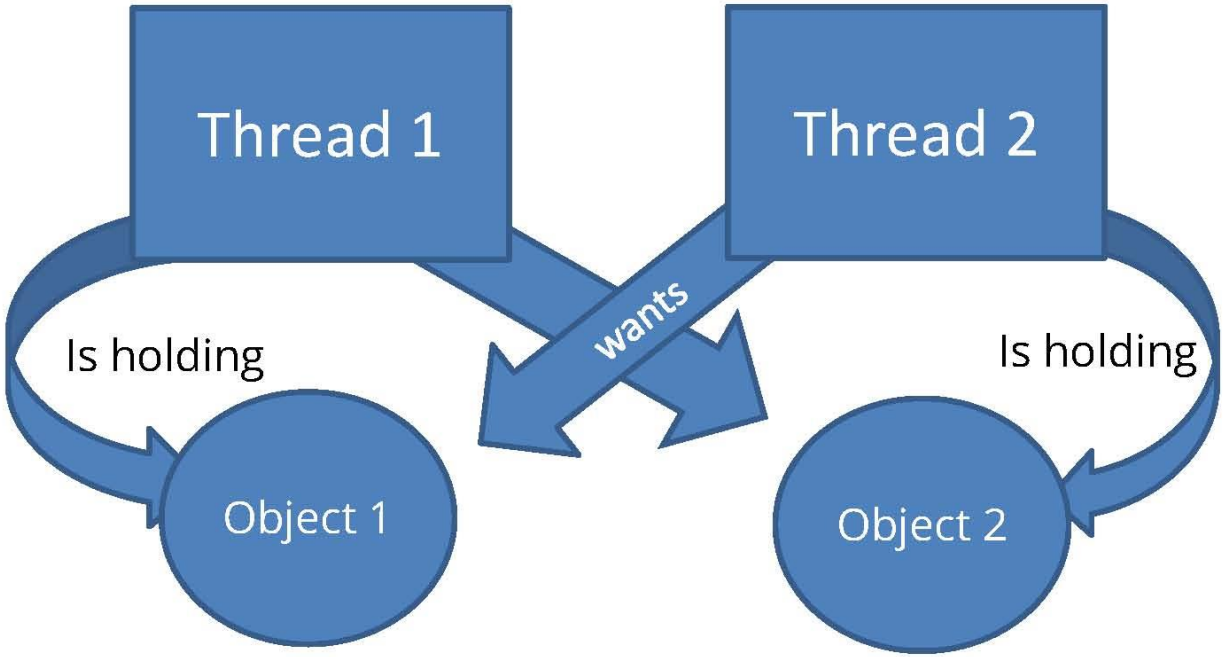
BUILD SUCCESSFUL

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

Deadlocks 1/5

- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other



DeadLocks 2/5

```
public class SimpleDeadLock extends Thread {
    public static final String obj0 = "obj0";
    public static final String obj1 = "obj1";
    public static void main(String[] args) {
        Thread t1 = new MyThread(obj0, obj1);
        Thread t2 = new MyThread(obj1, obj0);
        t1.start();
        t2.start();
    }
    /*Клас потоку*/
    private static class MyThread extends Thread {
        /*Об'єкти-монітори блокувань*/
        private String firstLock;
        private String secondLock;
       public MyThread(String firstLock, String secondLock) {...}
...
```

DeadLocks 3/5

```
...
    @Override
    public void run() {
       System.out.println(getName() + " is started");
       synchronized (firstLock) {
          System.out.println("Holding " + firstLock + " by "
                                          + Thread.currentThread().getName());
          try {sleep(10);} catch (InterruptedException ex) {
             ex.printStackTrace();
          }
          System.out.println(getName()
            + " is waiting for " + secondLock + "...");
          synchronized (secondLock) {
             System.out.println("Holding " + firstLock + " & " + secondLock
                                  + " by " + Thread.currentThread().getName());
          }
       }       }    }  }
```

DeadLocks 4/5

Output:

Thread-0 is started
Thread-1 is started
Holding obj0 by Thread-0
Holding obj1 by Thread-1
Thread-0 is waiting for obj1...
Thread-1 is waiting for obj0...
hangs while waiting

Deadlocks - jps, jstack 1/3

```
C:\>jps
7684 Jps
2920 SimpleDeadLock
1212

C:\>jstack 2920
2016-05-06 19:34:22
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.92-b14
mixed mode):
…
```

Deadlocks - jps, jstack 2/3

```
…
Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x0000000002cf98c8 (object
                        0x00000000e0f73450, a java.lang.Object),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x0000000002cf6e28 (object
                        0x00000000e0f73460, a java.lang.Object),
  which is held by "Thread-1":
…
```

Deadlocks - jps, jstack 3/3

...

Java stack information for the threads listed above:

==========

"Thread-1":

at thread.deadlocks.SimpleDeadLock$Thread2.run(SimpleDeadLock.java:57)

- waiting to lock <0x00000000e0f73450> (a java.lang.Object)

- locked <0x00000000e0f73460> (a java.lang.Object)

"Thread-0":

at thread.deadlocks.SimpleDeadLock$Thread1.run(SimpleDeadLock.java:36)

- waiting to lock <0x00000000e0f73460> (a java.lang.Object)

- locked <0x00000000e0f73450> (a java.lang.Object)

Found 1 deadlock.

Deadlocks elimination 1/5

```
public class SimpleDeadLockElimination extends Thread {
    public static final String obj0 = "obj0";
    public static final String obj1 = "obj1";
    /* Lock objects with a certain order of selection */
    public static String firstLock = null;
    public static String secondLock = null;
    public static void main(String[] args) {
        Thread t1 = new Thread1();
        Thread t2 = new Thread2();
        t1.start();
        t2.start();
    }
```

...

Deadlocks elimination 2/5

```
…
 /*Rule of selection of monitor objects: the object with the smaller
    hash code will be selected first*/
private static void selectLockRule() {
   if (obj0.hashCode() == obj1.hashCode()) {
      try {
         throw new Exception("Hashcode collision");
      } catch (Exception ex) {
         System.out.println(ex.getMessage());
      }
   } else if (obj0.hashCode() < obj1.hashCode()) {
      firstLock = obj0;
      secondLock = obj1;
   } else {
      firstLock = obj1;
      secondLock = obj0;
   }  …
```

Deadlocks elimination 3/5

```
…
   synchronized (firstLock) {
      System.out.println("Holding " + firstLock
         + " by " + Thread.currentThread().getName());
      try {
         sleep(10);
      } catch (InterruptedException ex) {
         ex.printStackTrace();
      }
     System.out.println(Thread.currentThread().getName()
           + " is waiting for " + secondLock + "...");
      synchronized (secondLock) {
         System.out.println("Holding " + firstLock
                     + " & " + secondLock + " by "
                + Thread.currentThread().getName());
      }
   }      }  …
```

Deadlocks elimination 4/5

```
    …
    /*Thread class*/
    private static class MyThread extends Thread {
        @Override
        public void run() {
            System.out.println(getName() + " is started");
            selectLockRule();
        }
    }
}
```

Output:

```
Thread-0 is started
Holding obj0 by Thread-0
Thread-1 is started
Thread-0 is waiting for obj1...
Holding obj0 & obj1 by Thread-0
Holding obj0 by Thread-1
Thread-1 is waiting for obj1...
Holding obj0 & obj1 by Thread-1
BUILD SUCCESSFUL
```

Dining philosophers problem