<epam> Excellence in Software Engineering

BDO

JAVA ACADEMY

9. Writing Multithreaded Application

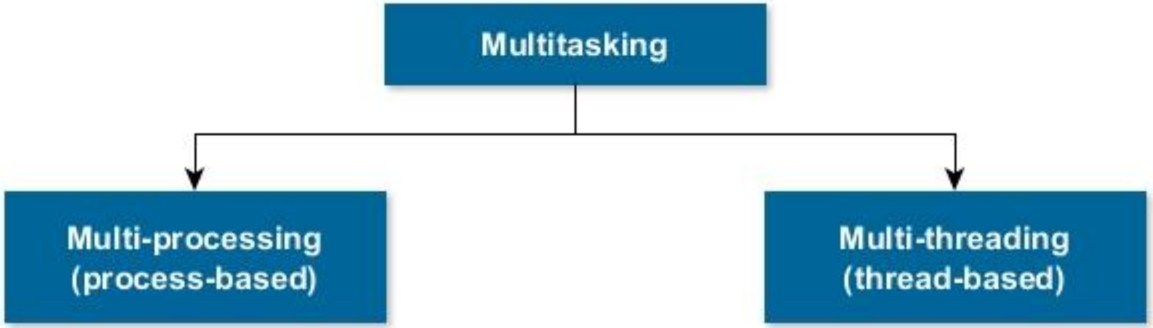
Table of Contents

1. Introduction

Overview
Threads and Processes
What is a Process?
What is a Thread?
Types of Threads
Daemon Threads
Normal vs. Daemon Threads
Benefits of Threads
Risk of Threads

Overview

- concurrent execution of multiple tasks is called multitasking



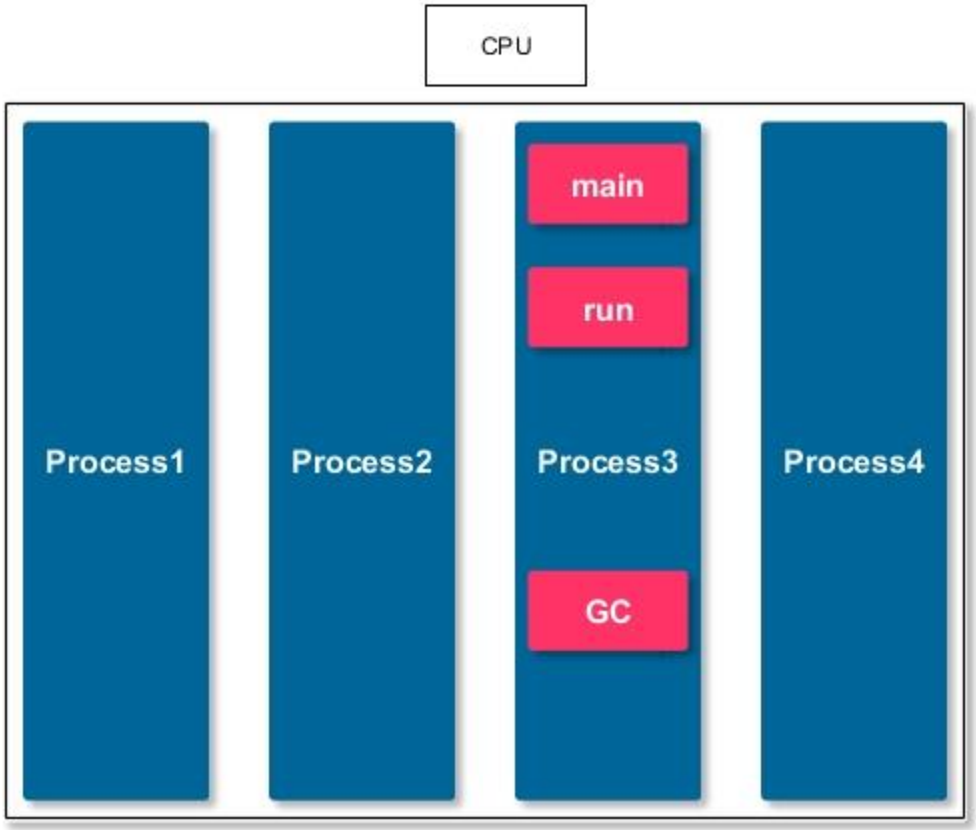
Overview cont'd

- multithreading is a specialized form of multitasking
- a multithreaded program contains two or more parts (threads) that can run concurrently
- each thread defines a separate path of execution
- Java provides built-in support for multithreaded programming

Threads and Processes

- in concurrent programming, there are two basic units of execution
  - processes
  - threads

Threads and Processes cont'd



What is a Process?

- isolated, independently executing programs
- OS allocates resources
    - memory
    - file handlers
    - security credentials
    - etc.

- processes communicate with one another through
    - sockets
    - signal handlers
    - shared memory
    - semaphores
    - files

What is a Thread?

- shorthand for thread of control
- threads are called lightweight processes
- a section of code executed independently of other threads of control within a single program
- smallest sequence of programmed instructions that can be managed independently by
    - an operating system scheduler
    - JVM thread scheduler
- threads share process-wide resources
    - memory address space
- memory that can be shared between threads is called shared memory or heap memory
- all instance fields, static fields, and array elements are stored in heap memory

What is a Thread? cont’d

- each thread has its own
  - program counter
  - stack and local variables

- a thread is a path of execution through a program

- single threaded programs -> one path of execution

- multiple threaded programs -> two or more

Types of Threads

- threads are divided into two types
  - normal threads
  - daemon threads

- when JVM starts up
  - main thread is the only non-daemon thread
  - other threads created by the JVM are daemon threads (e.g. garbage collection)

- when a new thread is created, it inherits the daemon status of the thread that created it
  - by default any threads created by the main thread are also normal threads

Daemon Threads

- sometimes you want to create a thread that performs some helper function
    - you do not want the existence of this thread to prevent the JVM from shutting down

Daemon Threads cont'd

- daemon threads should be used sparingly, few processing activities can be safely abandoned at any time with no cleanup
    - it is dangerous to use daemon threads for tasks that might perform any sort of I/O
        - when the JVM shuts down, all threads stop
        - note that a daemon thread in the middle of I/O may be stopped without being given a chance to clean up properly

- daemon threads are best saved for "housekeeping" tasks
    - a background thread that periodically removes expired entries from an in-memory cache

Normal vs. Daemon Threads

- only difference is what happens when they exit
  - when a thread exits, the JVM performs an inventory of running threads
- if only daemon threads are left
  - the JVM automatically exits
    - initiated when the last non-deamon thread terminates
- when the JVM halts
  - remaining daemon threads are abandoned
  - finally blocks are not executed
  - stacks are not unwound

Benefits of Threads

- to maintain responsiveness of user interfaces (e.g., Swing)
- to monitor status of some resource (e.g., DB)
- exploiting multiple processor cores
- handling asynchronous events
- some problems are intrinsically parallel
  - dining philosophers problem
  - producer-consumer problem

Risk of Threads

- safety hazards
- liveness hazards
- performance hazards

Risk of Threads: Safety Hazards

- unsafe code may result in race condition

```
public class SequenceGenerator {
    private int currentSequence = 0;

    public int getNextSequence() {
        return currentSequence++;
    }
}
```

- proper synchronization should be done
- ordering of operations in multiple threads is unpredictable
- happens-before relationship
    - a guarantee that memory writes by one specific statement are visible to another specific statement

Risk of Threads: Liveness Hazards

- a liveness failure occurs when an activity gets into a state such that it permanently unable to make forward progress

- deadlock - a liveness hazard scenario
  - Thread A waits for a lock to be released by Thread B and vice versa
  - these programs will wait for ever

- livelock
  - a thread often acts in response to the action of another thread and vica versa
    - then livelock may result
  - as with deadlock, livelocked threads are unable to make further progress
  - the threads are not blocked
    - they are simply too busy responding to each other to resume work

Risk of Threads: Performance Hazards

- context switches
  - scheduler suspends the active thread temporarily so another thread can run
  - saving and restoring execution context
  - CPU time spent scheduling threads

- when threads share data
  - they must use synchronization which prevents compiler optimizations
  - flush or invalidate memory caches
  - create synchronization traffic on shared memory bus
    - this bus has limited bandwidth
    - is shared across all processors

Table of Contents

2. Java Threading Fundamentals

- Threads in Java
- Defining a Thread
- Instantiating a Thread
- Starting a Thread
- Thread Termination
- Thread Cancellation
- Thread Scheduler
- Thread Priorities
- Thread Groups
- Threads and Exceptions
- Handling Uncaught Exceptions
- Thread-Safety in Java

Threads in Java

- context switches
  - scheduler suspends the active thread temporarily so another thread can run
  - saving and restoring execution context
  - CPU time spent scheduling threads

- when threads share data
  - they must use synchronization which prevents compiler optimizations
  - flush or invalidate memory caches
  - create synchronization traffic on shared memory bus
    - this bus has limited bandwidth
    - is shared across all processors

Defining a Thread

- extending java.lang.Thread class

```
public class SampleThread extends Thread {
    public void run() {
        system.out.println("Running....");
    }
}
```

- implementing java.lang.Runnable interface

```
public class SampleRunnable implements Runnable {
    public void run() {
        system.out.println("Running....");
    }
}
```

Instantiating a Thread

- we can use any of the above mentioned ways to define a thread
- but can instantiate a thread only by using java.lang.Thread class

- if we create a thread by extending Thread class

```
SampleThread thread1 = new SampleThread();
```

- if we create a thread by implementing Runnable interface;

```
SampleRunnable runnable1 = new SampleRunnable();
Thread thread2 = new Thread(runnable1);
```

Starting a Thread

- use the java.lang.Thread class's start() method to start a thread

```
public class TestThread {
    public static void main(String... args) {
        SampleThread t1 = new SampleThread();
        t1.start();
    }
}
```

- not recyclable
  - once if we start a thread, it **can never be started again**
  - **java.lang.IllegalThreadStateException** will be thrown otherwise

Thread Termination

- thread terminates when
  - Runnable object's *run()* method returns
  - *System.exit()* called
  - in case of *InterruptedException*

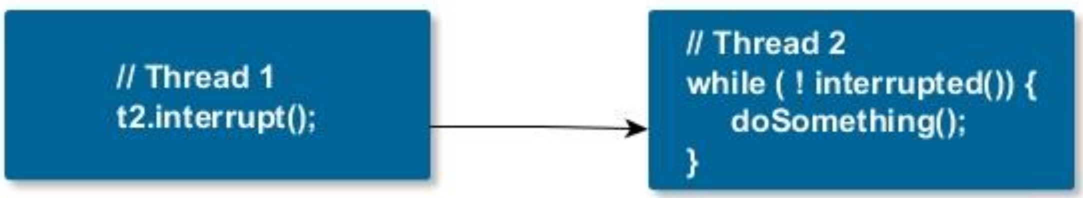
- can wait for thread to terminate
  - use *join()* method on the thread

- application terminates when last non-daemon thread terminates
  - not necessarily the main thread

- daemon status of a thread determined by flag
  - *setDaemon()*

Thread Cancellation

- stopping a thread prematurely
  - use *interrupt()*
  - do not use deprecated *stop()* method



- *interrupt()*
  - send signal to thread

- *isInterrupted()*
  - return true if thread has been interrupted

- *interrupted()* - static
  - check if current thread has been interrupted
  - clear interrupted status flag

Thread Scheduler

- part of the JVM that decides which thread should run at any given moment
  - takes threads out of the run state
  - most JVMs map Java threads directly to native threads on the underlying OS

- any thread in the runnable state can be chosen by the scheduler to be the one and only running thread.

- if a thread is not in a runnable state, then it cannot be chosen to be the currently running thread.

- we do not control the thread scheduler, but we can sometimes influence it

Thread Scheduler cont'd

- thread will execute until
  - blocking call executed e.g. *sleep()*
  - higher priority thread becomes runnable
  - thread scheduler decides to stop it
- **Thread.yield()** hints that another thread can run
  - equal priority
  - if no runnable threads available then no-op
  - scheduler may ignore the call

Thread Priorities

- threads have priorities
  - inherited from creating thread
  - range is **Thread.MIN_PRIORITY** to **Thread.MAX_PRIORITY**
  - standard default is **Thread.NORM_PRIORITY**
- use the following syntax

```
SampleThread thread1 = new SampleThread();
thread1.setPriority(8);
thread1.start();
```

Thread Groups

- threads belong to a thread group
  - helps administration, scheduling and security
- a thread group can be passed to thread constructor
  - default to same group as creator

```
ThreadGroup tg = new ThreadGroup();
Runnable rObj = new RunnableImpl();
Thread t = new Thread(tg, rObj);
```

Thread Groups cont'd

- a thread group can have properties set
    - apply to all threads in group
    - *maxPriority*, *daemon*

- thread can only interact with threads in same group
    - *interrupt()*, *destroy()*, etc.

Threads and Exceptions

- most exceptions are synchronous
  - related to a specific thread
- handled within causing thread
  - stack unwinds to appropriate handler
- Java Runtime provides special handler for uncaught exceptions
  - default behaviour to print stack trace and exit
  - can be overridden

Handling Uncaught Exceptions

- implement Thread.UncaughtExceptionHandler

```
public interface UncaughtExceptionHandler {
    void uncaughtException ( Thread t, Throwable e );
}
```

- uncaught exception handler can be installed per thread
  - or as default

```
...
Thread.setDefaultUncaughtExceptionHandler ( ... );
...
Thread.currentThread().setUncaughtExceptionHandler( ... );
...
```

Uncaught Exceptions Example

```
public class ExceptionProcessor implements
                        Thread.UncaughtExceptionHandler {
    public void uncaughtException( Thread t, Throwable e ) {
        System.err.println( "Unhandled exception (" + e.toString() +
                ") in Thread " + t.getName() + " Check logs for details." );
        logExceptionDetails(t,e);
    }
}
```

```
public static void main( String[] args ) {
    Thread.currentThread().setUncaughtExceptionHandler(
        new ExceptionProcessor() );
    doStuff( … );
}
```

Thread-Safety in Java

- immutable objects are by default thread-safe because there state can not be modified once created
  - since String is immutable in Java, its inherently thread-safe.
- read only or **final variables** in Java are also thread-safe in Java
- locking is one way of achieving thread-safety in Java
- static variables if not synchronized properly becomes major cause of thread-safety issues

Thread-Safety in Java cont’d

- example of thread-safe class in Java: Vector, Hashtable, ConcurrentHashMap, String etc.

- atomic operations in Java are thread-safe
    - e.g. reading a 32 bit int from memory because its an atomic operation it can't interleave with other thread

- local variables are also thread-safe because each thread has there own copy and using local variables is good way to writing thread-safe code in Java

- **volatile keyword** in Java can also be used to instruct thread
    - not to cache variables
    - read from main memory
    - can instruct JVM not to reorder or optimize code from threading perspective

Volatile Variables

- alternative, weaker form of synchronization

- ensures that updates to a variable are propagated predictably to other threads

- when a field is declared volatile
    - the compiler and runtime are put on a notice
    - this variable is shared
    - operations on it should not be reordered with other memory operations

- volatile variables are not cached in registers or in caches
    - read of a volatile variable always returns the most recent write by any thread

- accessing a volatile variable performs no locking
    - lighter-weight synchronization than **synchronized**

Lab01

1. Thread Fundamentals - Instantiation

- create a Thread instance using subclassing class Thread and also one by implementing the Runnable interface.
- give a name attribute to both, which you're setting using their constructor
- both Thread instances should print out their name upon execution
- start both
- try to create and start more instances and note that the actual execution might not be in that order

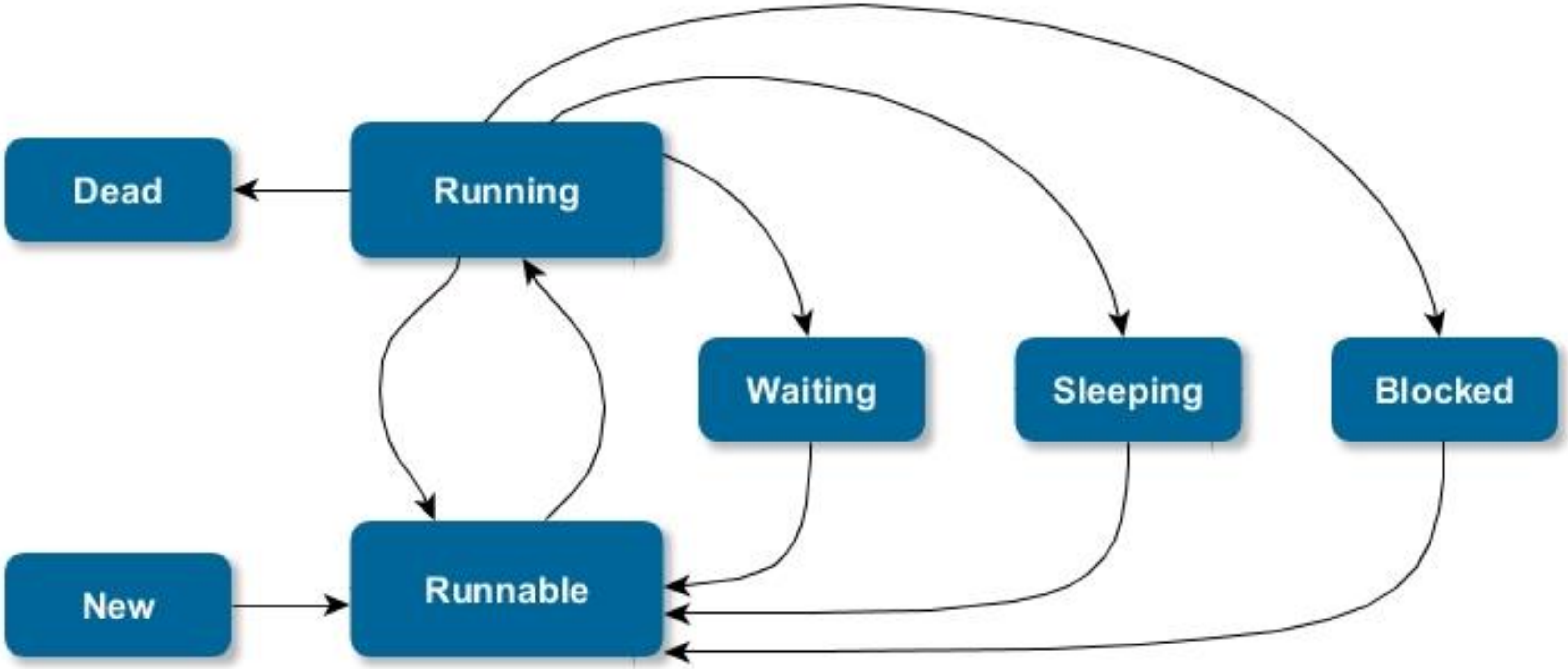
- solution: ThreadExample.java

Table of Contents

3. Manage and Control Thread Lifecycle

Thread Lifecycle
Thread States
Controlling Threads
Yielding
Sleeping
Monitors, Waiting, and Notifying
Object Lock and Synchronization
wait() and notify()
The join() Method
Interruption in Blocking Calls
What is InterruptedException?

Thread Lifecycle



Thread States

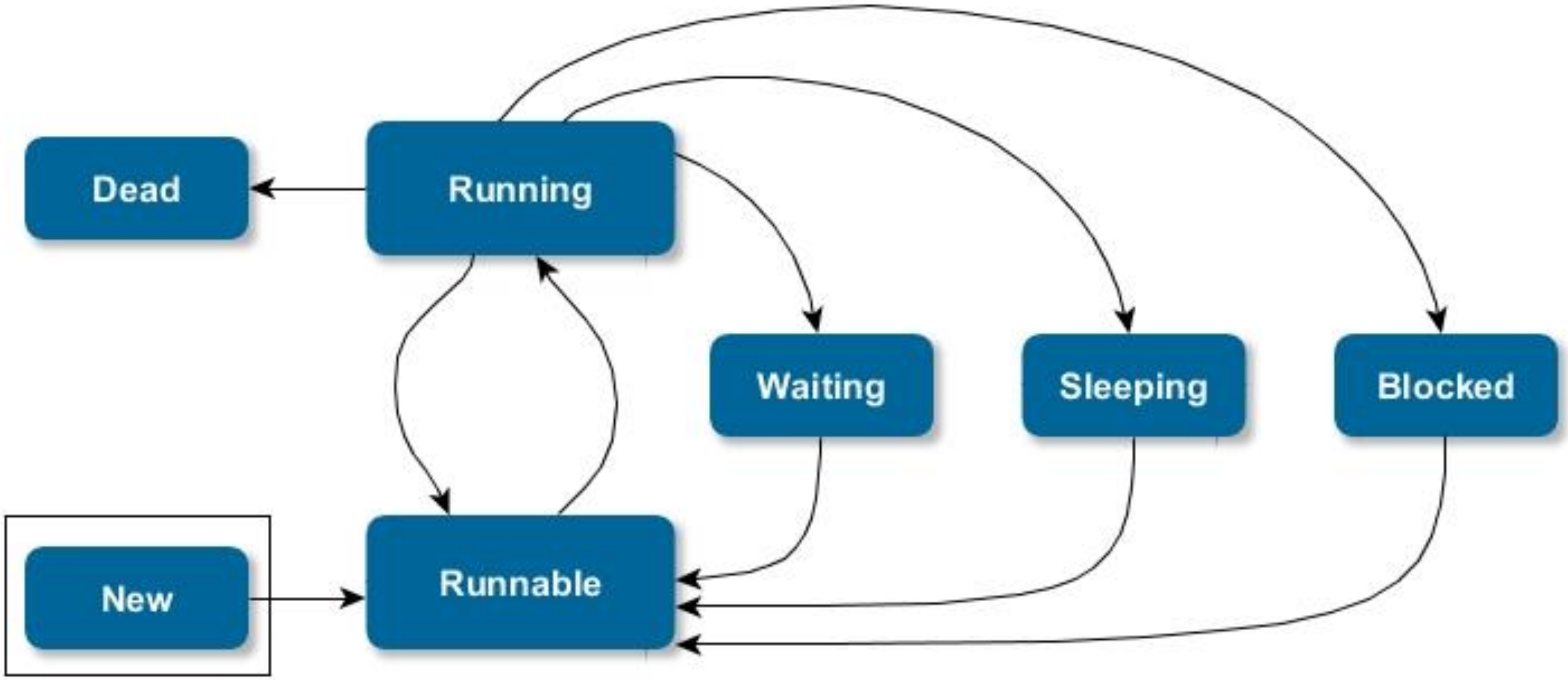
- the thread scheduler's job is to move threads in and out of the **running state**
- while the thread scheduler can move a thread from the running state back to runnable
    - other factors can cause a thread to move out of running, but not back to runnable
- when the thread's *run()* method completes, in which case the thread moves from the running state directly to the **dead state**

Thread States: New

- **new**
  - the state the thread is in after the Thread instance has been created
  - *start()* method has not been invoked on the thread

- at this point, the **thread** is considered **not alive**

Thread States: New cont'd



Thread States: Runnable

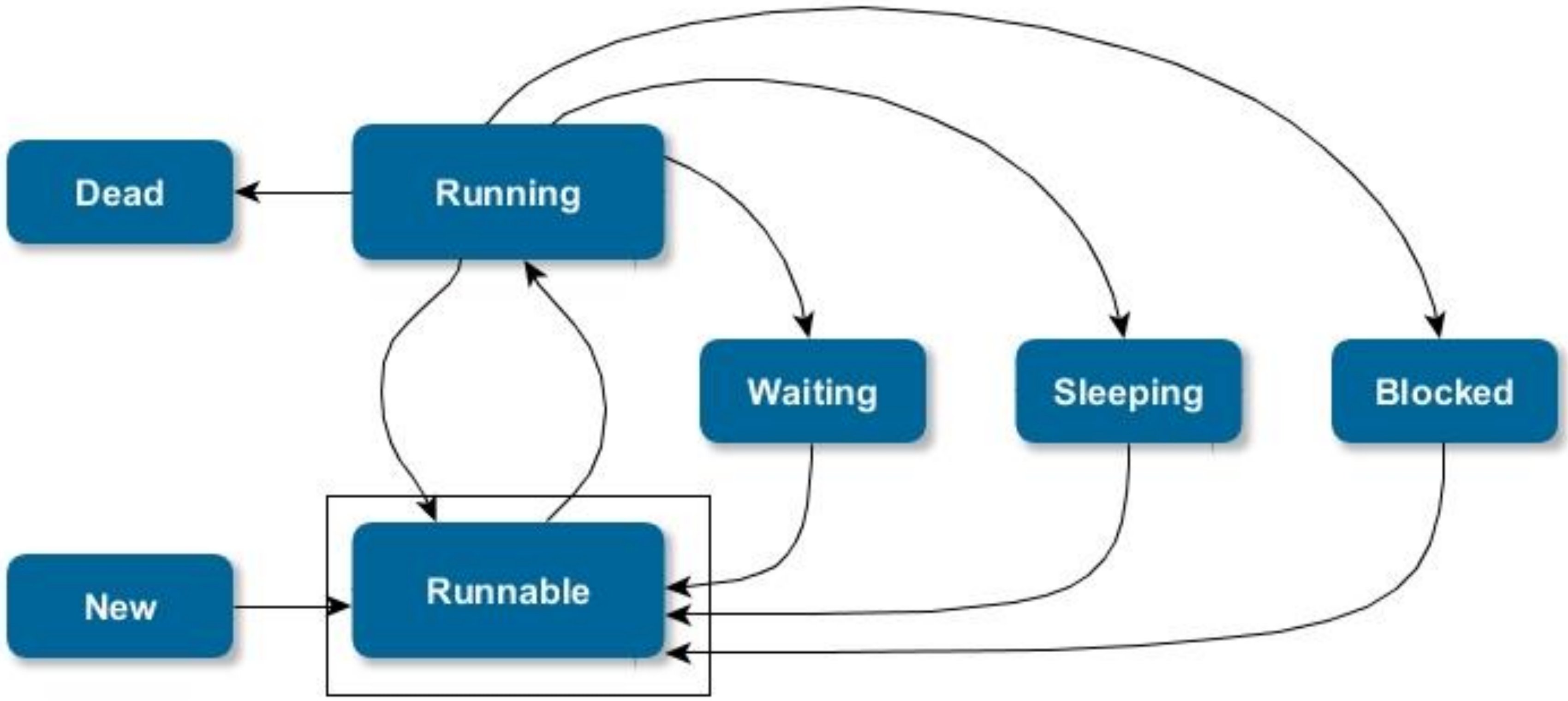
- **runnable**
  - the state a thread is in when it is eligible to run
  - the scheduler has not selected it to be the running thread

- a thread first enters the ready state when the *start()* method is invoked

- a thread can also return to the runnable state after either running or coming back from a blocked, waiting, or sleeping state

- when the thread is in the runnable state, it is considered **alive**

Thread States: Runnable cont'd



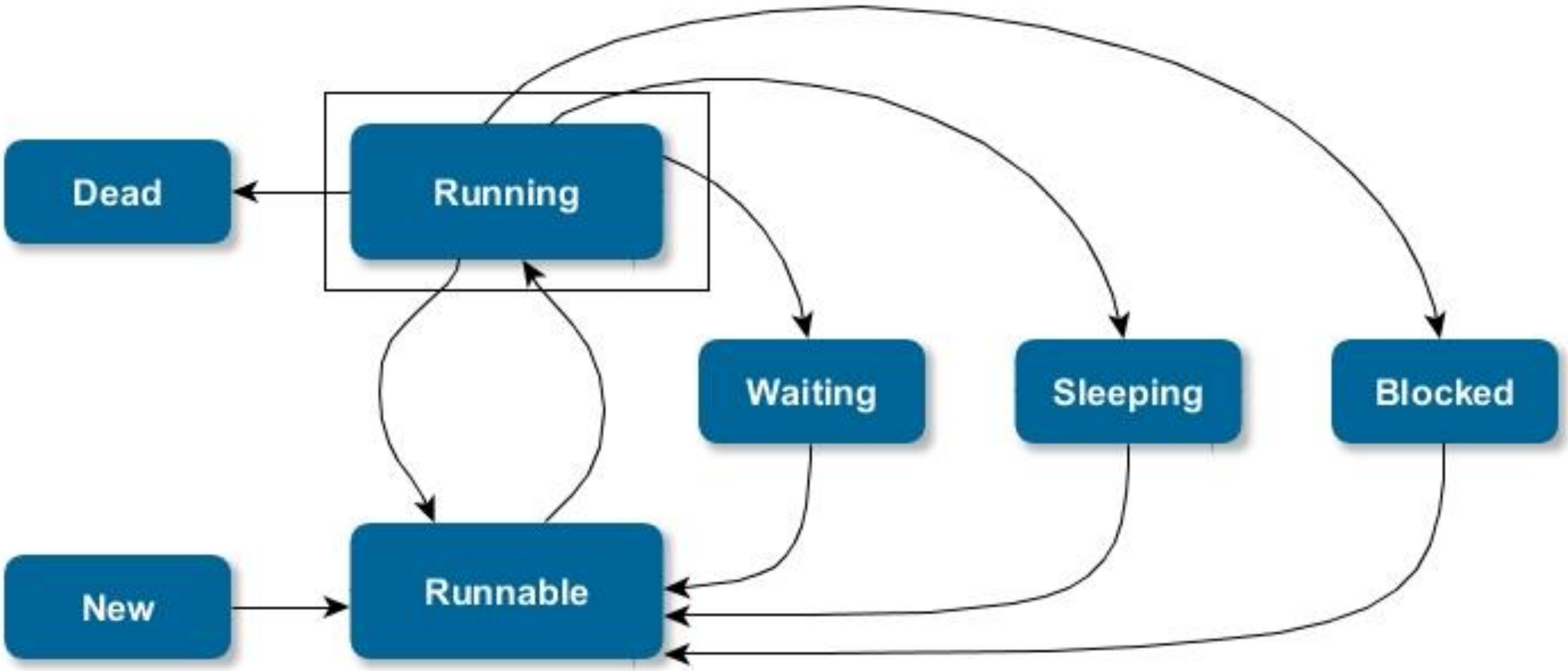
Thread States: Running

- **running**
  - the state a thread is in when the thread scheduler selects it (from the runnable pool) to
  - be the currently executing process

- a thread can transition out of a running state for several reasons

- only way to get to the running state: **the scheduler chooses a thread from the runnable pool**

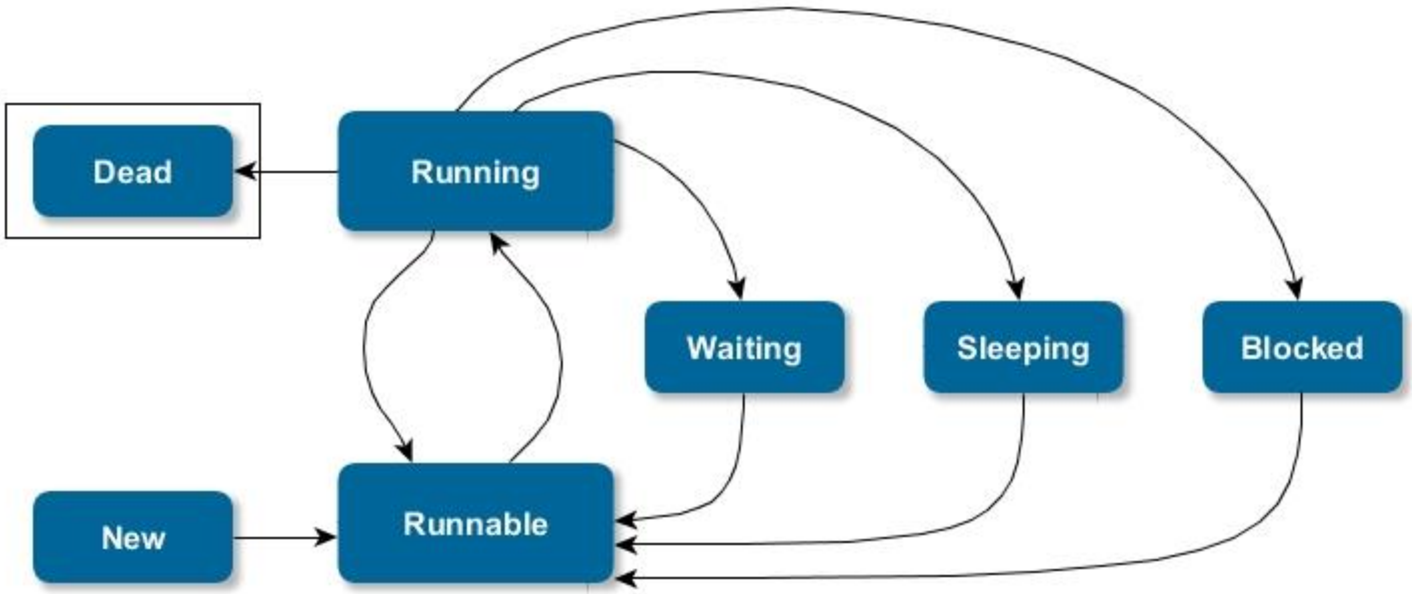
Thread States: Running cont’d



Thread States: Dead

- **dead**
  - a thread is considered dead when its *run()* method completes
  - it may still be a viable Thread object, but it is no longer a separate thread of execution
- once a thread is dead, it can never be brought back to life!
- if you invoke *start()* on a dead Thread instance, you'll get a runtime exception
- if a thread is dead, it is no longer considered to be alive

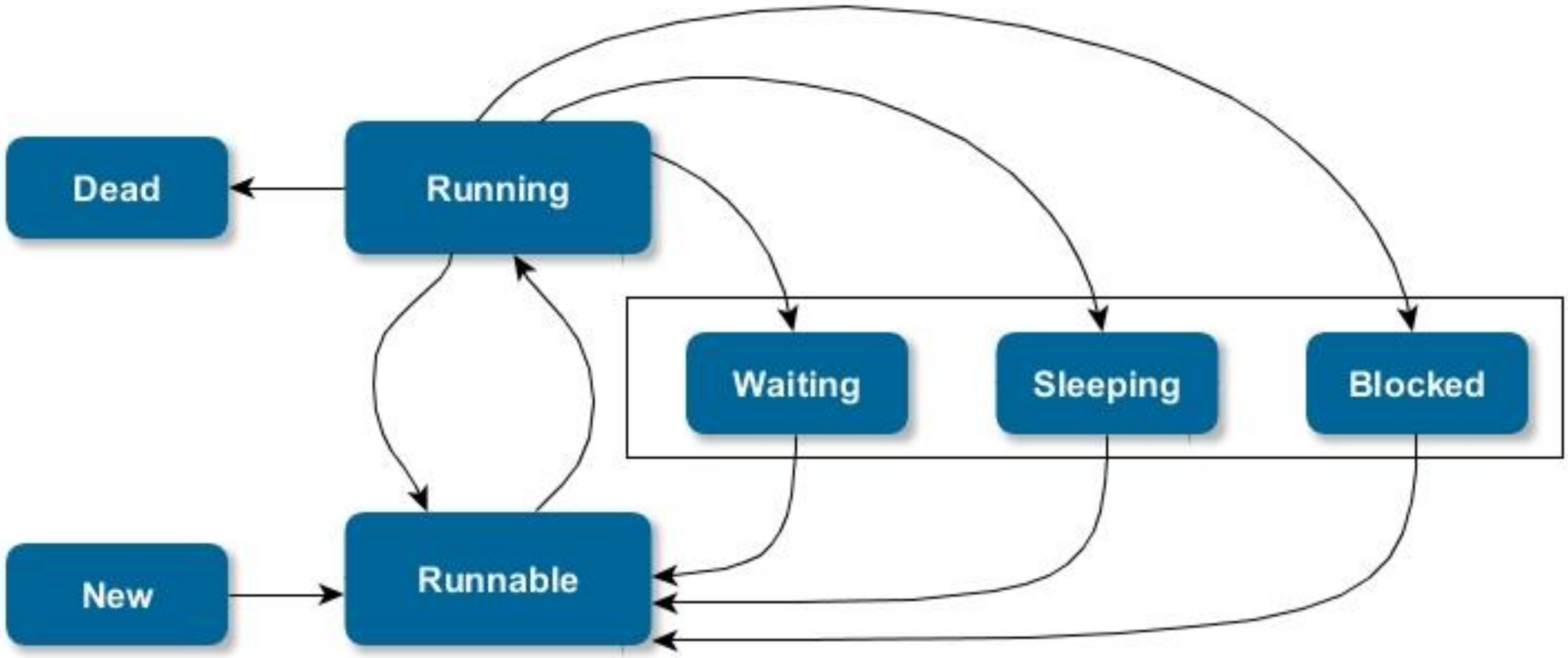
Thread States: Dead cont'd



Thread States: Non-Running

- **various non-running states**
  - **waiting**
  - **sleeping**
  - **blocked** (there are non-running states related to **monitors)**

Thread States: Non-Running cont'd



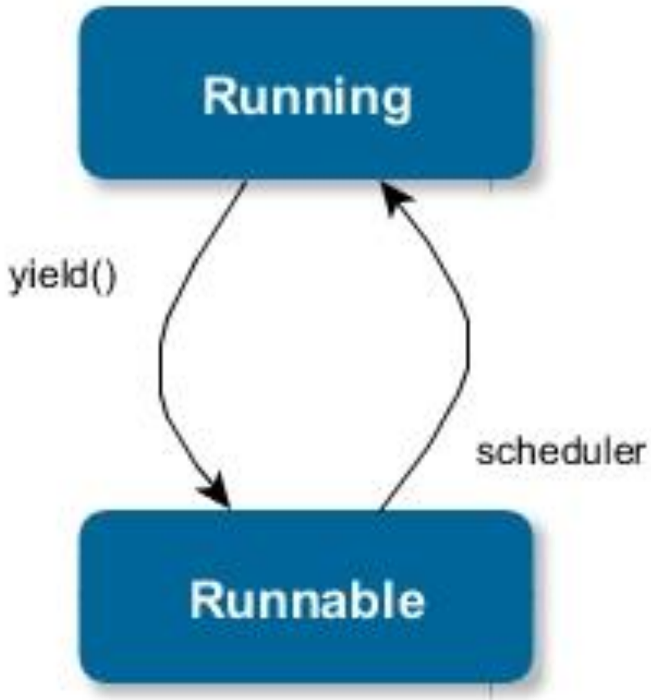
Controlling Threads

- moving threads from state to state
- pathways are
  - yielding
  - sleeping and then waking up
  - blocking and then continuing
  - waiting and then being notified

Yielding

- static method: **Thread.yield()**
- suppose to make the currently running thread head back to **runnable state**
  - allow other threads of the same priority to get their turn
- in reality the *yield()* method is not guaranteed to do what it claims
- even if *yield()* does cause a thread to step out of running and back to runnable state
  - there is no guarantee the yielding thread will not just be chosen again over all the others
- a *yield()* will not ever cause a thread to go to the **waiting**/**sleeping**/**blocked** state

Yielding cont’d



Sleeping

- static method: **Thread.sleep()**
- forcing a thread to go into a sleeping state before coming back to runnable
  - where it still has to be the currently running thread

```
try {
    Thread.sleep(5*60*1000); // Sleep for 5 minutes
} catch (InterruptedException ex) { }
```

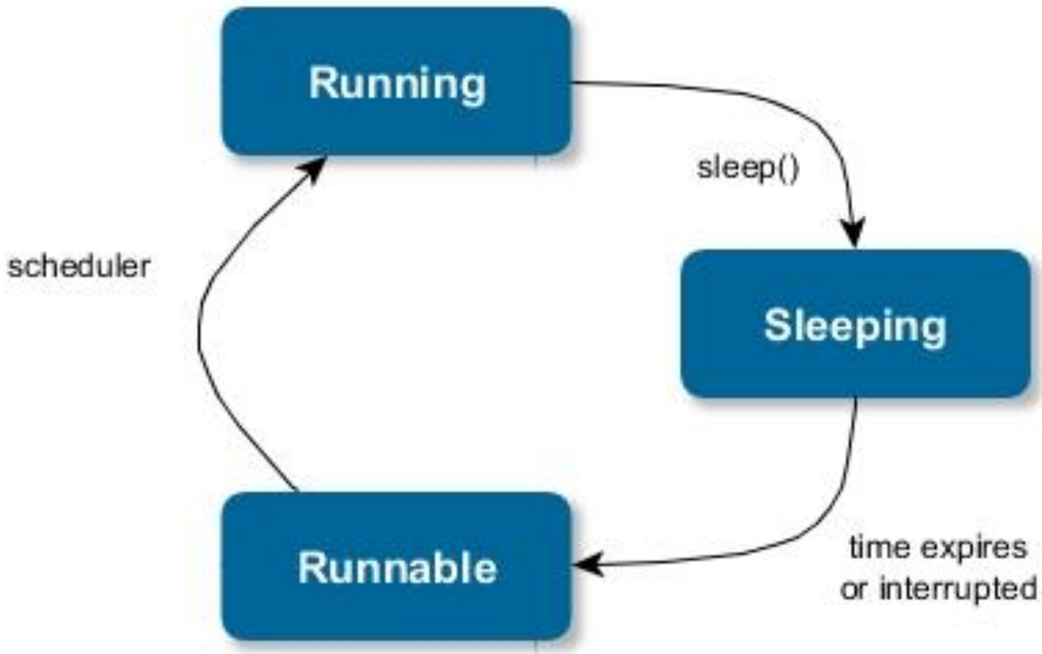
- notice that the *sleep()* method can throw a checked **InterruptedException**
  - you must acknowledge the exception with a handle or declare

Sleeping cont’d

- when the executing code hits a *sleep()* call, it puts the **currently running thread** to sleep

- when a thread encounters a sleep call
  - it must go to sleep for at least the specified number of milliseconds
  - unless it is interrupted before its wake-up time (InterruptedException is thrown)

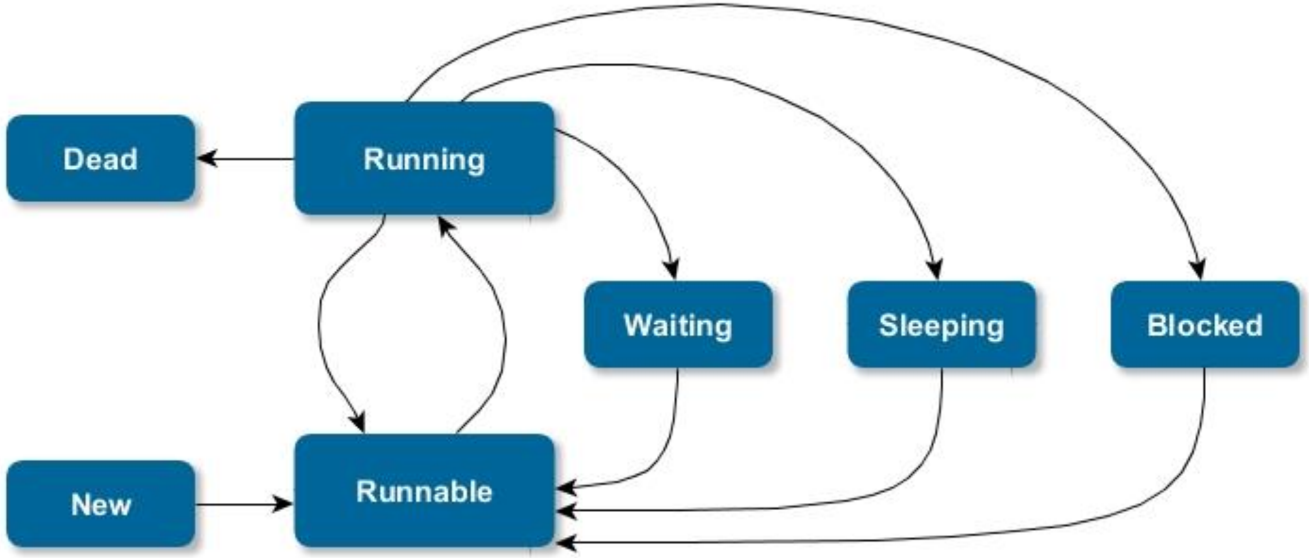
Sleeping cont'd



Monitors, Waiting, and Notifying

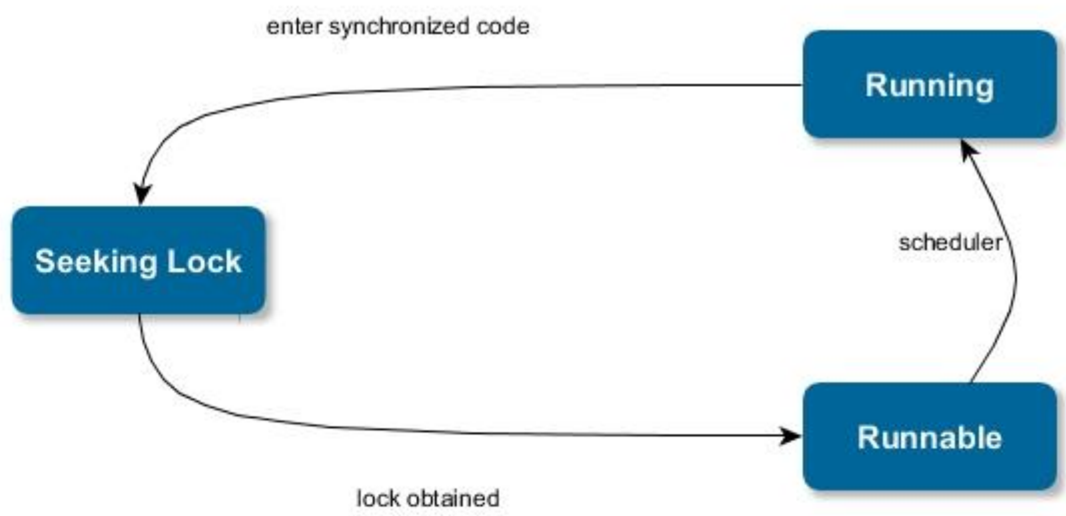
Java's monitor support provides the following resources:

- a lock for each object.
- the *synchronized* keyword for accessing an object's lock.
- the *wait()*, *notify()*, and *notifyAll()* methods, which allow the object to control client threads



Object Lock and Synchronization

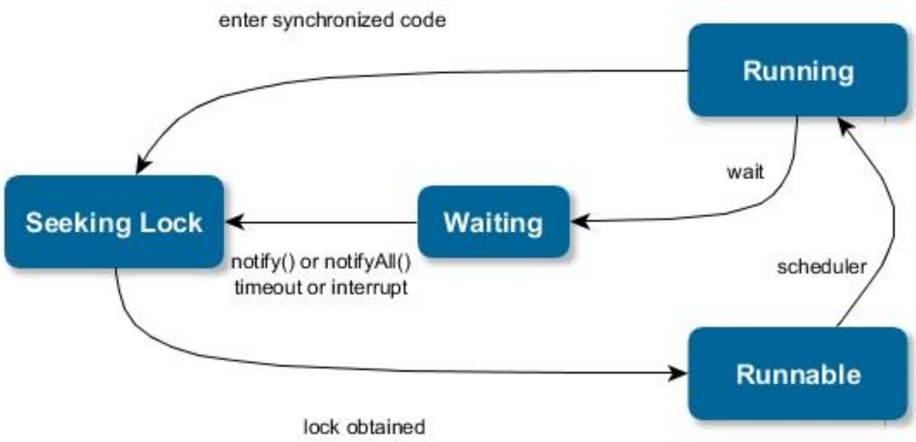
- there are two ways to mark code as synchronized
  - synchronize an entire method
  - synchronize a code block



wait() and notify()

- *wait()*
  - the calling thread gives up the CPU
  - the calling thread gives up the lock
  - the calling thread goes into the monitor's waiting pool

- *notify()*
  - one arbitrarily chosen thread
  - moved out of the monitor's waiting pool
  - goes the **seeking lock** state



- thread that was notified
  - must re-acquire the monitor's lock before it can proceed

wait() and notify() cont'd

- wait(), notify(), and notifyAll() must be called from within a synchronized context!
- a thread cannot invoke a wait or notify method on an object unless it owns that object's lock.

The join() Method

- non-static method of class Thread: *join()*
- lets one thread "join onto the end" of another thread.
- a call to *join()* is guaranteed to cause the current thread to stop executing
    - until the thread it joins with completes
    - or if the thread it is trying to join with is not alive

```
public class Test {
    public static void main(String[] args) {
        SampleThread thread1 = new SampleThread();
        thread1.start();
        // Some more code here
        thread1.join();
    }
}
```

Interruption in Blocking Calls

- *sleep()*, *wait()* and *join()* can be interrupted
  - *InterruptedException* thrown

- exception clears interrupted flag
  - may need to interrupt again to pass on behaviour

```
public void doSomethingAndSleep() {
    try {
        doSomething();
        Thread.sleep(some_time);
    } catch ( InterruptedException ie ) {
        Thread.currentThread.interrupt();
    }
}
```

What is InterruptedException?

- there is no way to simply stop a running thread in java
- Thread.interrupt() is a way to tell the thread to stop what it is doing
- if the thread is in a blocking call, the blocking call will throw an InterruptedException otherwise the interrupted flag of the tread will be set
- a Thread or a Runnable that is interruptable should check from time to time
    - Thread.currentThread().isInterrupted()
    - if it returns true, cleanup and return

Lab02

2. Thread Lifecycle – Sleeping a thread (State: Sleeping)

- extend Thread created in Lab01 with that each should count from 1 to 10, and sleep in each iteration for a given time (e.g.: 2 seconds).
- tart both
- note that the execution order may be changing
- re-iterate the need for catching InterruptedException, what it means and why it is needed

- solution: ThreadExampleSleep.java

Lab03

3. Thread Lifecycle – Monitoring lifecycle state: Dead

- extend the previously created main thread to wait until both threads finished and print something out
- first, try to solve it with sleeping the main thread
- second, try using the join() method, thus simplifying your code.
- re-iterate the difference between sleep and join
  - also mark it down that even though join() can be called without parameters, it is advised to use a maximum waiting time
- solution: ThreadExampleSleep.java

Lab04

4. Controlling threads – Thread Priorities

- modify the previously created counter thread to print its priority upon start
- in the main method, create a large number (use a constant) of threads without starting them (store the references) and assign a priority to them based on a logic you prefer
- upon execution notice, that threads with higher priority tend to start earlier, even if you called the start() method later

- solution: ThreadExamplePrio.java

Lab05

5. Controlling threads – Thread priorities & yielding

- create running statistics of the threads, log the starting time of each thread and the thread should save the total time it used for execution into an object, which is reachable by the main thread.
- use yielding on each iteration (you can play around switching it on/off)
- after all threads finished execution (use join()), print the average running time of the threads per priority.
- notice, that threads with higher priority executed significantly faster, than threads with lower priority.
- play around with the parameters! (iteration count, thread count, priority assignment logic)

- solution: ThreadExamplePrio.java

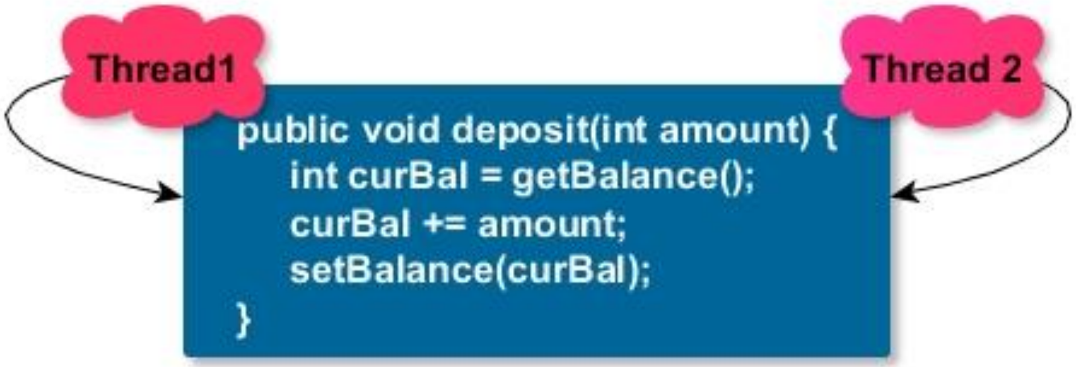
Table of Contents

4. Synchronization

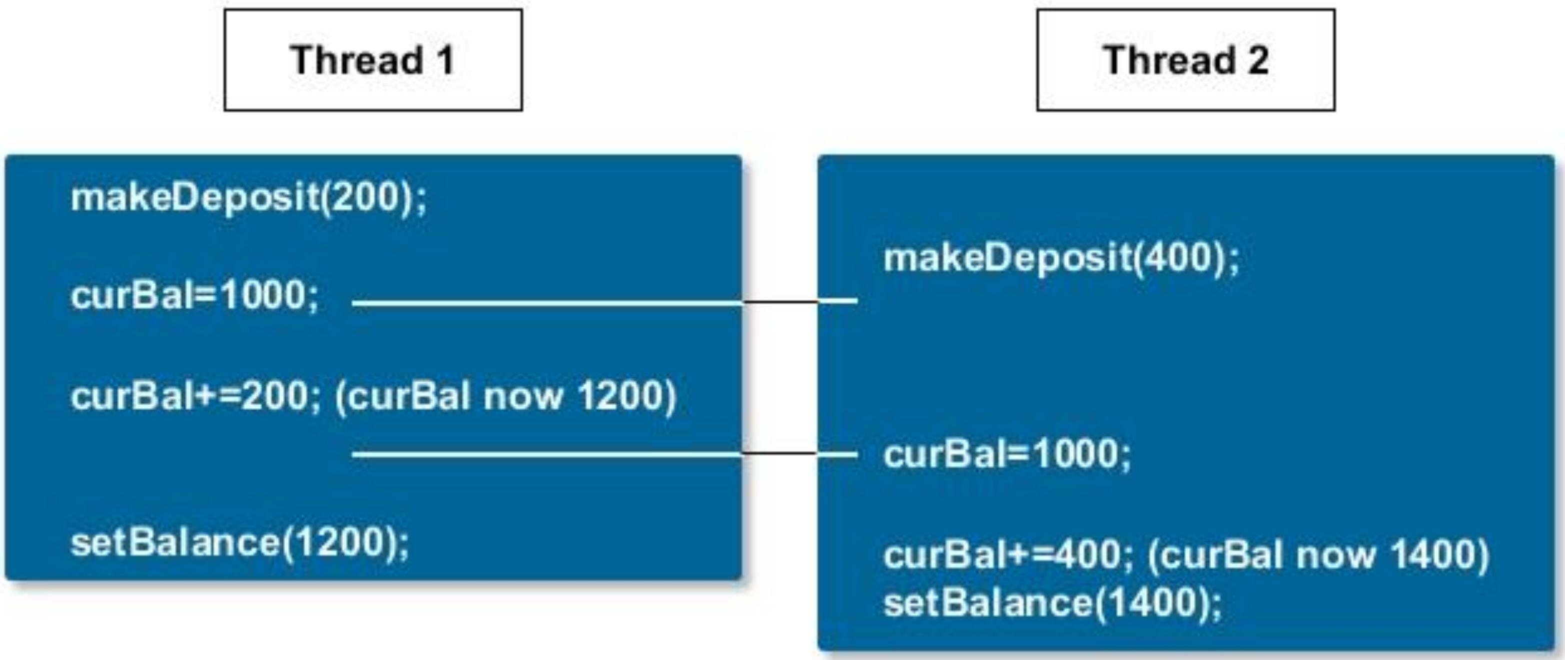
- Why Should We Worry?
- Synchronization
- Monitor
- Basic Data Access Synchronization
- Synchronized Blocks
- Synchronizing Execution
- Producer-Consumer Example
- Basic Synchronization
- Solution Using Basic Synchronization

Why Should We Worry?

- once started, threads work independently of each other
- care is needed when multiple threads access the same data
  - race condition
- may also be necessary to synchronize execution
  - prevent one or more threads from proceeding until some condition is met



Why Should We Worry? cont'd



Synchronization

- race condition
  - race condition occurs when two threads attempt to use the same resource at the same time
- places where we refer/use the mutable variables
  - these mutable variables can be accessed by multiple threads simultaneously
- another use of synchronization is memory visibility
- synchronization ensures atomicity and visibility

Synchronization cont’d

- a monitor is an object that is used for mutual exclusion, or mutex
- only one thread can own a monitor at a given time
- when a thread acquires a lock, it is said to have entered the monitor
- all other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor
- these other threads are said to be waiting for the monitor

Basic Data Access Synchronization

- every object in java has exactly one built-in lock by default
  - called intrinsic lock
  - guarantees mutual exclusion
- basis of protecting shared data
  - synchronized method acquires lock before proceeding
- synchronized keyword
- only one thread in any synchronized method of object at a time

```
public synchronized void deposit ( int amount ) {
    int curBal = getBalance();
    curBal += amount;
    setBalance(curBal);
}
```

Synchronized Blocks

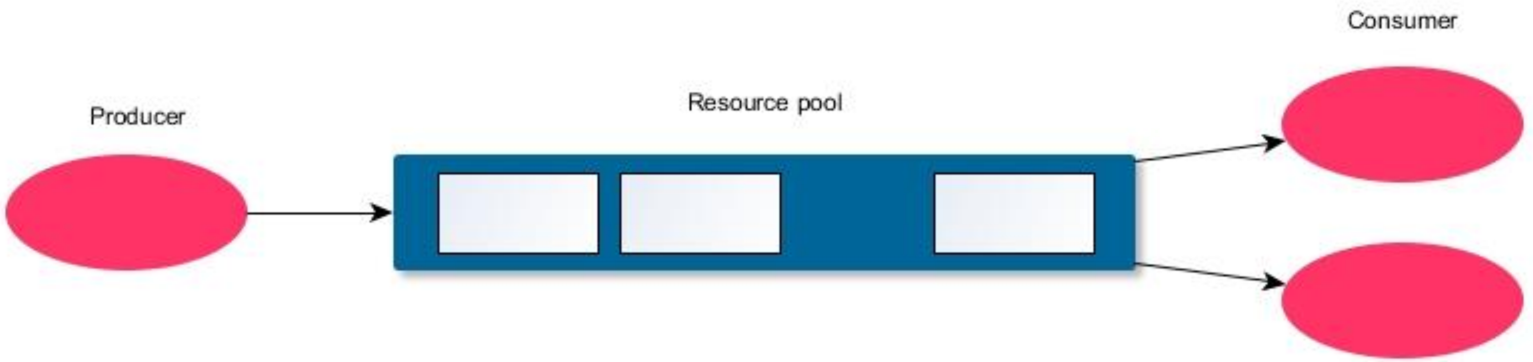
- more flexible than synchronizing entire methods
  - leads to finer grained locking
  - offers potentially higher throughput
- can synchronize on any object

```
synchronized ( myAccount ) {
    int curBal = getBalance();
    curBal += amount;
    setBalance(curBal);
}
```

```
synchronized ( yourAccount ) {
    int curBal = getBalance();
    curBal += amount;
    setBalance(curBal);
}
```

Synchronizing Execution

- Producer-Consumer problem



- cannot consume if resource pool is empty
    - cannot produce if poll is full
- synchronization between producers and consumers is necessary

Example Producer

```
public class Producer extends Thread {
    private Letters pool;
    private final static String ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    public Producer(Letters thePool, String threadName) {
        super(threadName); // Sets the thread's name
        this.pool = thePool; // Reference to the shared resource
    }

    public void run(){
        char ch;
        // Add 10 letters to the pool
        for (int i = 0; i < 10; i++) {
            ch = ALPHABET.charAt((int)(Math.random() * 26));
            pool.addLetter(ch);
            // Diagnostic print
            System.out.println("Thread: " + getName() + " added " + ch );
            // Random wait before we add the next letter
            try{
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e ){}
        }
    }
}
```

Example Consumer

```
public class Consumer extends Thread {
    private Letters pool;
    public Consumer( Letters thePool, String threadName ) {
        super(threadName); // Sets the thread's name
        this.pool = thePool; // Reference to the shared resource
    }

    public void run () {
        char ch;
        // Take 10 letters from the pool
        for(int i = 0; i < 10 ; i++) {
            ch = pool.takeLetter();
            // Diagnostic print
            System.out.println("Thread: " + getName() + " took letter : " + ch);
            // Random wait before we grab the next letter
            try {
                sleep((int)(Math.random() * 2000));
            } catch ( InterruptedException e ){}
        }
    }
}
```

Driver Program and Shared Resource

- shared resource represented as an interface
  - allows different concrete implementations
  - built using different synchronization primitives

```
public class ProducerConsumerDriver {
    private static Letters letterPool = new LettersImplClass();
    public static void main(String[] args) {
        // 2 producers and 2 consumers for now...
        new Producer(letterPool, "Producer1" ).start();
        new Producer(letterPool, "Producer2" ).start();
        new Consumer(letterPool, "Consumer1" ).start();
        new Consumer(letterPool, "Consumer2" ).start();
    }
}
```

```
public interface Letters {
  void addLetter(char c);
  char takeLetter();
}
```

Basic Synchronization

- *wait()* blocks calling thread until some condition is satisfied
  - use object to represent notification point
- *notifyAll()* wakes up all threads blocked in *wait()*
- must hold lock on object
  - *wait()* atomically releases lock before blocking thread
- threads should check condition before proceeding
  - maybe only one may proceed

```
synchronized ( obj ) {
    while ( ! condition ) {
        try {
            obj.wait();
        } catch ( InterruptedException ie ) {}
    }
}
```

```
synchronized ( obj ) {
    condition = true;
    obj.notifyAll();
}
```

Solution Using Basic Synchronization

```
public class LettersImplBasic implements Letters {
    private final static int BUFFER_CAPACITY = 6;
    private char[] buffer = new char[BUFFER_CAPACITY];
    private int next = 0;
    private boolean isFull = false;
    private boolean isEmpty = true;
    public synchronized void addLetter(char ch) {
        // wait until pool has room for new letter
        while ( isFull ) {
            try {
                wait();
            } catch ( InterruptedException e ) {}
        }
        // add the letter to the next available spot
        buffer[next++] = ch;
        // are we full?
        if (next == BUFFER_CAPACITY) {
            isFull = true;
        }
        isEmpty = false;
        notifyAll();
    }
// continued over...
```

Solution Using Basic Synchronization cont’d

```
    // …continued from above…
    public synchronized char takeLetter(){
    // wait until the pool becomes non-empty
        while (isEmpty == true) {
            try {
                wait(); // we'll exit this when isEmpty turns false
            } catch (InterruptedException e) {}
        }
        // decrement the count, since we're going to remove one letter
        next--;
        // Was this the last letter?
        if (next == 0){
            isEmpty = true;
        }
        // we know the pool can't be full, because we took a letter
        isFull = false;
        notifyAll();
        return(buffer[next]); // return char to Consumer thread
    }
}
```

Lab06

6. Lock and Synchronization

- create an example in which you can demonstrate memory inconsistency
  - you are free to create any example
  - if you’re having troubles finding out a context, imagine a class which wraps an attribute for which a method exists for modifying this attribute
- run a number of threads, try to find the occurrence of inconsistency
- solve the inconsistent state above with method synchronization
- can it be solved with synchronizing only a subset of the method?

- solution: ThreadExampleWaitNotify.java

Lab07

7. Monitors, Waiting and Notifying

- create 3 Threads: MessageBroker, MessageConsumer and MessageProducer
  - the producer should send a flow of messages one-by-one to the MessageBroker if its buffer is empty, while the MessageConsumer should take messages from the MessageBroker, if its buffer if not empty
- first, solve it with sleeping the threads while checking the MessageBroker’s empty state
- second, upgrade the solution with the usage of wait() and notify on the MessageBroker, which should notify the waiting threads if the empty state changes

- solution: ThreadExampleWaitNotify.java

Lab08

8. Deadlock (optional)

- create a code, that can possibly end up in a deadlock situation
- to make it more visible, place prints, so that it demonstrates which object locked which resource

Table of Contents

5. High Level Concurrency Support

- Concurrency Classes
- Concurrent Collections
- Blocking Queues
- Blocking Queue Producer-Consumer Example
- Task Management - The Old Way
- The Executor Framework
- Implementing the Executor Interface
- Using the Executors
- Supplied Executor Implementations
- Customising a ThreadPoolExecutor
- Executor Lifecycle
- The Future Interface
- Working with Callables and Futures
- Callable/Future Example
- The CompletionService
- CompletionService Example

Concurrency Classes

- new set of classes to promote concurrency
  - based on work by Doug Lea
  - java.util.concurrent package
  - high level utilities for managing concurrency
  - concurrent collections classes

- original Java threading primitives too low level
  - inflexible models for locking
  - leads to repetition of code
  - prone to errors
  - thread safe rather than thread-hot

- thread-safe library: a library that is safe to call from a multithreaded application
- thread-hot library: library that is multithreaded internally
  - library that does processor-intensive calculations that can be sped up through parallel programming

Concurrent Collections

- previous collection classes not thread safe
  - apart from very old Vector/Hashtable
- thread safety provided by decorating classes
  - Collections.synchronizedSet (existing Set)
  - not efficient
- several new classes provided in Java 5 to offer more scalability and better performance
  - java.util.concurrent.ConcurrentHashMap
  - java.util.concurrent.CopyOnWriteArrayList
  - java.util.concurrent.CopyOnWriteArraySet

Blocking Queues

- extends Queue interface with blocking operations
  - take() - remove element, wait until this is possible
  - put() - add element, wait until this is possible
- ArrayBlockingQueue
  - ordered FIFO backed by array, bounded
- LinkedBlockingQueue
  - ordered FIFO, may be bounded
- PriorityBlockingQueue
  - blocking version of PriorityQueue
- SynchronousQueue
  - rendezvous channel, each put() must be matched by a take() and vice versa

Blocking Queue Producer-Consumer Example

- recall example of letters pool from earlier
  - implementation using blocking queue

```
public class LettersImplQueue implements Letters {
    private static final int BUFFER_CAPACITY = 6;
    private BlockingQueue<Character> pool =
    new ArrayBlockingQueue<Character>(BUFFER_CAPACITY);
    public void addLetter(char c) {
        try {
            pool.put(c); // Blocks if Queue is full, ie. no space
        } catch ( InterruptedException e ) {
            Thread.currentThread().interrupt();
        }
    }

    public char takeLetter() {
        Character c = null;
        try {
            c = pool.take(); // Blocks if queue is empty
        } catch ( InterruptedException e ) {
            Thread.currentThread().interrupt();
        }
        return c;
    }
}
```

Task Management - The Old Way

- conventional model for server implementation
  - thread per request

```
public class MyServer {
    public static void main(String[] args) {
        ServerSocket socket = new ServerSocket( portnum );
        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            new Thread(r).start();
        }
    }
}
```

- thread creation relatively resource intensive
  - does not scale

The Executor Framework

- separates task submission from task execution

- based on Executor interface

Implementing the Executor Interface

- to support thread-per-task
  - "conventional" model shown before

```
public class ThreadPerTask implements Executor {
    public void execute( Runnable task ) {
        new Thread(task).start();
    }
}
```

- to support in-thread task execution
  - single threaded server

```
public class SingleThreadExecutor implements Executor {
    public void execute( Runnable task ) {
        task.run();
    }
}
```

Using the Executors

```
public class ExecutorBasedService {
    static Executor engine = new ThreadPerTask();
    // static Executor engine = new SingleThreadExecutor();
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(4000);
            while ( true ) {
                final Socket conn = sock.accept();
                Runnable r = new Runnable() {
                    public void run() {
                        handleRequest(conn);
                    }
                };
                engine.execute(r);
            }
        } catch ( … ) { … }
    }

    static void handleRequest ( Socket c ) {
        …
        System.out.println("Thread ID: " + Thread.currentThread().getId() );
    }
}
```

```
Thread ID: 7
Thread ID: 8
...
```

```
Thread ID: 1
Thread ID: 1
...
```

Supplied Executor Implementations

- constructed through static class
- Executors.newFixedThreadPool(int n)
  - fixed set of n threads operating off unbounded queue
  - new threads created to replace threads that crash
- Executors.newCachedThreadPool()
  - unlimited size thread pool
  - new threads created on demand
  - threads unused for 60 seconds terminated and removed
- Executors.newSingleThreadExecutor()
  - like example shown earlier

Customising a ThreadPoolExecutor

- as produced by *newFixedThreadPool()* and *newCachedThreadPool()*
  - *getPoolSize()* to discover size of the pool
  - *beforeExecute()* and *afterExecute()* for logging, instrumentation…

- can also provide a Thread Factory

```
// Ensure all newly created threads are daemon threads
public class DaemonThreadFactory implements ThreadFactory {
    public Thread newThread ( Runnable r ) {
        Thread thr = new Thread ( r );
        thr.setDaemon(true);
        return thr;
    }
}
```

```
public class ExecutorBasedService {
    static Executor engine = Executors.newFixedThreadPool(10);
    public static void main(String[] args) {
        ( (ThreadPoolExecutor)engine ).setThreadFactory(
        new DaemonThreadFactory() );
    ...
```

Executor Lifecycle

- managed through ExecutorService interface
  - extends Executor interface
  - factory built Executor classes implement interface

- no new tasks accepted in "shutting down" state
  - shutdown() method waits for existing tasks to complete
  - shutdownNow() attempts to cancel existing tasks

The Future Interface

- represents a task
  - may be in any stage of execution (not started, running, finished)

- supports querying of task status
  - and cancellation of task

- implemented by FutureTask class
  - wrapper for Runnable object
  - or Callable object

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

Working with Callables and Futures

- create unit of work as object implementing Callable interface
  - implement *call()* method rather than *run()* method

- calling thread passes Callable object to Executor
  - using *submit()* rather than *execute()*
  - *submit()* returns Future object

- calling thread fetches result of work unit through Future object's get() method
  - if task complete, return result immediately
  - if task not complete, block until complete then return result
  - *ExecutionException* if task throws an exception
  - *CancellationException* if task was cancelled
  - *InterruptedException* if waiting thread was interrupted
  - *TimeoutException* if optional timeout expires

Callable/Future Example

```
public class CallableExample implements Callable<String> {
    public String call() {
        String result = "My Result";
        Thread.currentThread().sleep(3000); // wait 3 seconds
        return result;
    }
}
```

```
...
ExecutorService es = Executors.newSingleThreadExecutor();
Future<String> f = es.submit( new CallableExample() );
try {
    while ( ! f.isDone() ) {
        System.out.print(".");
        Thread.currentThread().sleep(500);
    }
    String callableResult = f.get();
} catch (InterruptedException ie) {
    // we were interrupted while waiting
} catch (ExecutionException ee) {
    // the task threw an exception
}
...
```

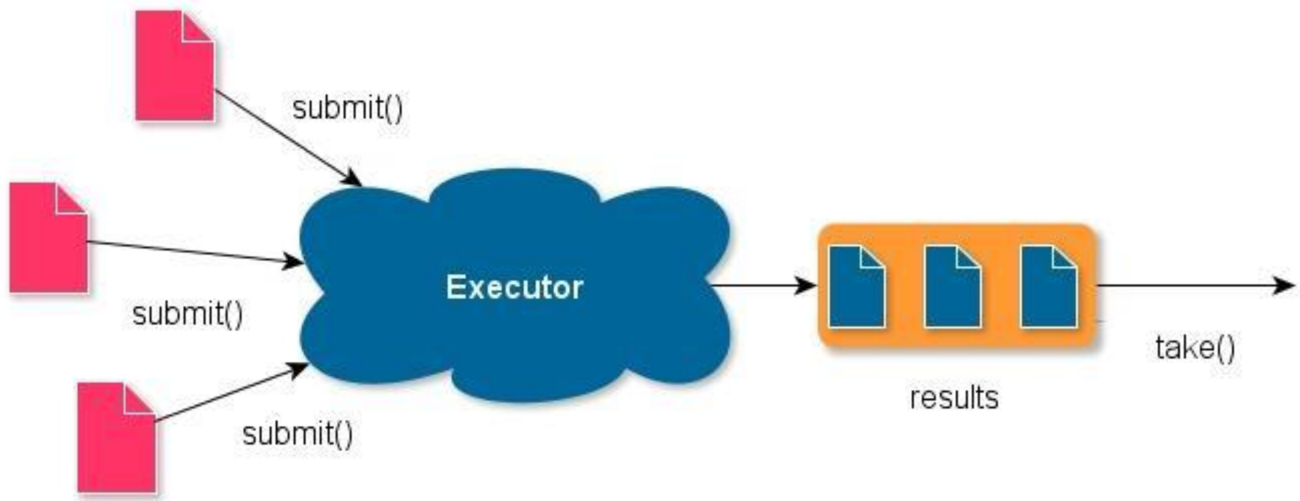
request to execute the task

The CompletionService

- combination of execution service and Queue-like interface for results
  - decouples task execution from result collection

- submit tasks using *submit()* method

- fetch results using *take()* or *poll()*



CompletionService Example

```
void solve(Executor e, Collection<Callable<Result>> solvers)
                                              throws InterruptedException {
    CompletionService<Result> ecs = new ExecutorCompletionService<Result>(e);
    int n = solvers.size();
    List<Future<Result>> futures = new ArrayList<Future<Result>>(n);
    Result result = null;
    try {
        for (Callable<Result> s : solvers) {
            futures.add(ecs.submit(s));
        }

        for (int i = 0; i < n; ++i) {
            try {
                Result r = ecs.take().get(); // Get first Future task to complete
                if (r != null) { // We want a non null result
                    result = r;
                    break;
                }
            } catch(ExecutionException ignore) {}
        }
    } finally {
        for (Future<Result> f : futures) {
            f.cancel(true);
        }
    }
    if (result != null)
    use(result);
}
```

Lab09

9. Executors – ExecutorService

- take one of your previous exercises, in which you needed to start multiple instances of the same thread and refactor it to use an ExecutorService
- after you have passed over all the needed Runnables to the Executor, mark it so, that it can not accept further tasks
- if the Executor does not terminate after 60 seconds, then mark it so that it cancels all currently executing tasks

- solution: ThreadPoolExample.java

Lab10

10. Executors – Callable, Future

- refactor the previously created code, so that the executed tasks are returning value, after completion, for instance a sum of the counted values
- in your main thread, aggregate the resulted values (for instance add them) and print it.
- do not forget to shut down the ExecutorService appropriately!

- solution: ThreadPoolExampleFuture.java

Excellence in
Software Engineering

THANK YOU FOR YOUR KIND ATTENTION!