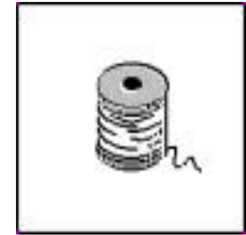
Thread Programming

Agenda

- The thread concept
- Thread synchronization
- Cooperation among threads
- The Executor framework

The thread concept



A **thread** is a single sequential flow of control in a program.

Java allows multiple threads to exist simultaneously.

Threads may be executed either on a multi-processor machine, or in *simulated parallel* on a single-processor machine on a time-sharing basis.

Threads

Advantages:

- Makes applications more responsible to input.
- Allows a server to handle multiple clients simultaneously.
- May take advantage of multiple processors.

Complications:

- Interruption of a thread may leave an object in an inconsistent state (*safety problem*)
- A thread may block the execution of other threads (*liveness problem*)

Creation of threads

Threads can be declared in one of two ways:

(1) by extending the `Thread` class

(2) by implementing the `Runnable` interface

Extending the `Thread` class

```
public class MyThread extends Thread {
    public void run() {
        // the thread body
    }

    // other methods and fields
}
```

Creating and starting a thread:

```
new MyThread().start();
```

Example

```
public class Counter1 extends Thread {
    protected int count, inc, delay;

    public Counter1(int init, int inc, int delay) {
        this.count = init; this.inc = inc; this.delay = delay;
    }

    public void run() {
        try {
            for (;;) {
                System.out.print(count + " ");
                count += inc;
                sleep(delay);
            }
        } catch (InterruptedException e) {}
    }

    public static void main(String[] args) {
        new Counter1(0, +1, 33).start();
        new Counter1(0, -1, 100).start();
    }
}
```

Output

```
0 0 1 -1 2 -2 3 4 5 -3 6 -4 7 -5 8 9 -6 10 11 -7 12 13 -8
14 15 -9 16 17 -10 18 19 -11 20 21 -12 22 23 -13 24 25 -14
26 27 28 -15 29 30 -16 31 32 -17 33 34 35 -18 36 37 -19 38
39 40 -20 41 42 -21 43 44 -22 45 46 47 -23 48 49 50 -24 51
52 53 -25 54 55 56 -26 57 58 59 -27 60 61 62 -28 63 64 65
-29 66 67 68 -30 69 70 71 -31 72 73 74 -32 75 76 77 -33 78
79 80 -34 81 82 -35 83 84 -36 85 86 87 -37 88 89 90 -38 91
92 93 -39 94 95 96 -40 97 98 99 -41 100 101 102 -42 103 104
...
```

Implementing the **Runnable** interface

```
public class MyRunnable extends AnotherClass
                        implements Runnable
{
    public void run() {
        // the thread body
    }

    // other methods and fields
}
```

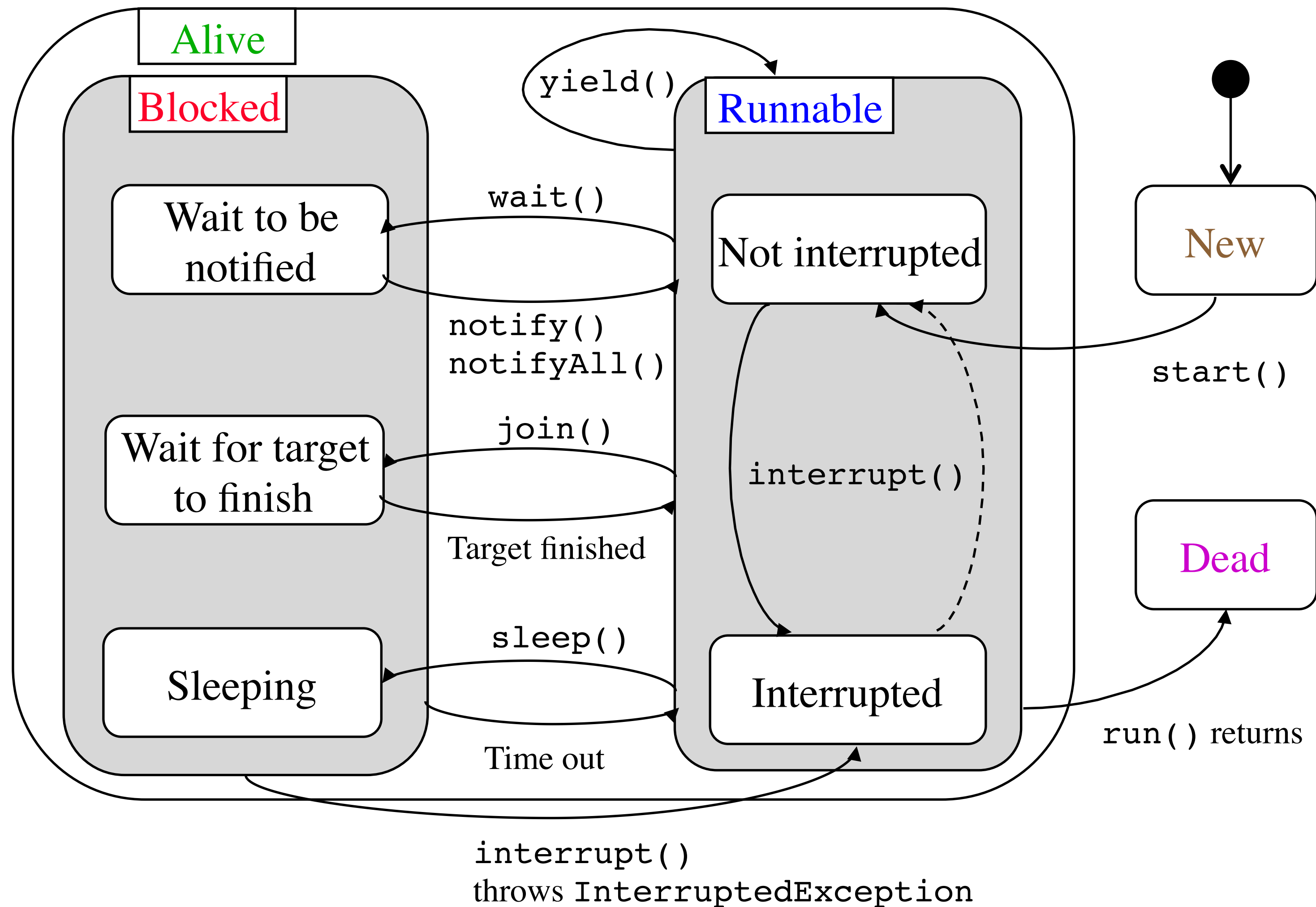
Creating and starting a thread:

```
new Thread(new MyRunnable()).start();
```

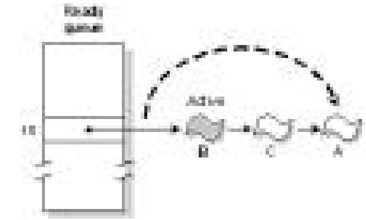
Example

```
public class Counter2 implements Runnable {
    protected int count, inc, delay;

    public Counter2(int init, int inc, int delay) {
        this.count = init; this.inc = inc; this.delay = delay;
    }

    public void run() {
        try {
            for (;;) {
                System.out.print(count + " ");
                count += inc;
                Thread.sleep(delay);
            }
        } catch (InterruptedException e) {}
    }

    public static void main(String[] args) {
        new Thread(new Counter2(0, +1, 33)).start();
        new Thread(new Counter2(0, -1, 100)).start();
    }
}
```

The life cycle of a thread



Thread priorities

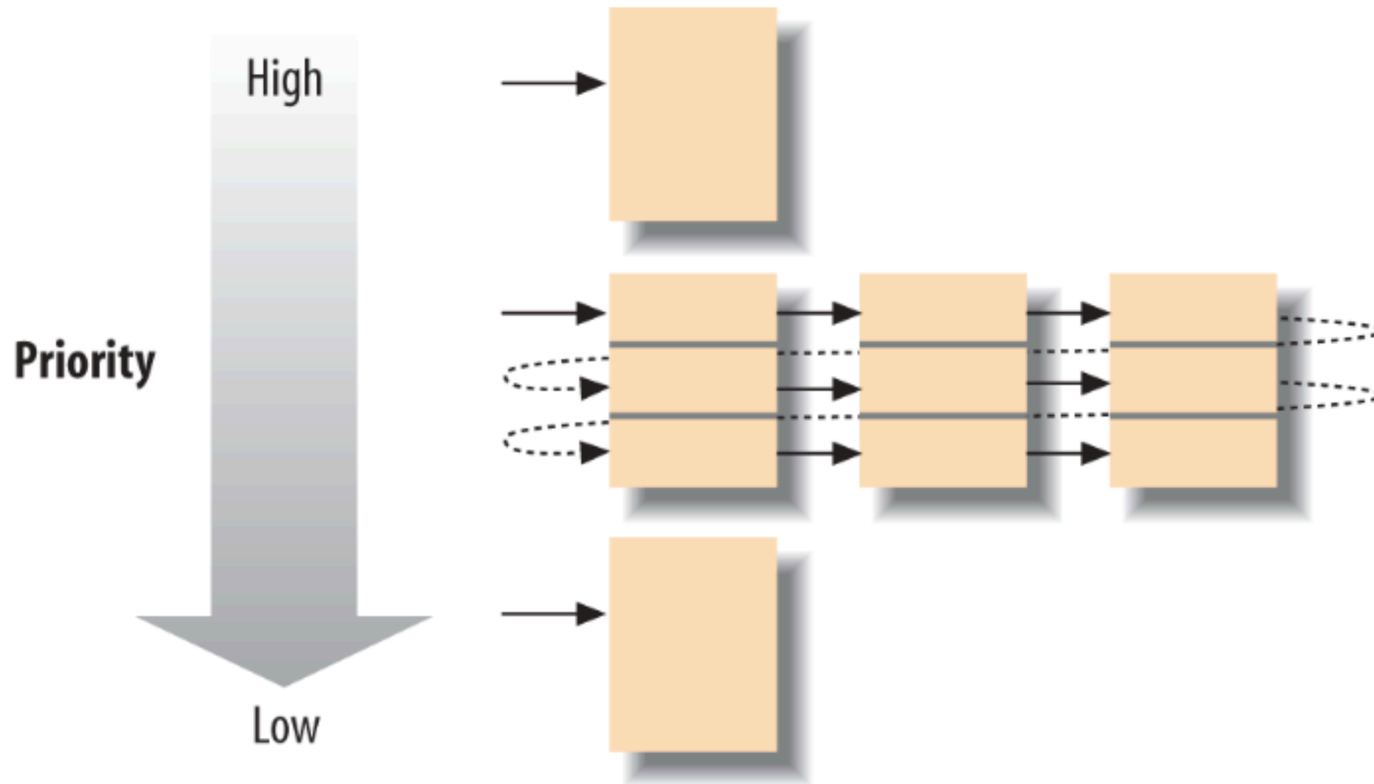


Each thread contains a **priority** attribute (an integer between 1 and 10).

The priority value of a thread is assigned at its creation. By default, a new thread has the same priority as the one that creates it. The priority of a thread may be changed during its lifetime.

The JVM will select the runnable threads with the highest priority for execution. Thus, a thread of higher priority will preempt a thread of lower priority. If more than one runnable thread has the same highest priority, one of them will be selected *arbitrarily*.

Priority preemptive, time-sliced scheduling



Synchronization

While an object is being modified by a thread, it can be in an inconsistent state.

It is important to ensure that no other tread accesses the object in this situation.

Example

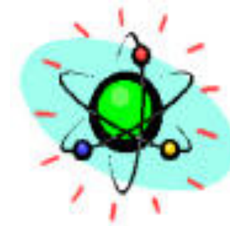
```
public class Account {
    // ...
    public boolean withdraw(long amount) {
        if (amount <= balance) {
            long newBalance = balance - amount;
            balance = newBalance;
            return true;
        }
        return false;
    }

    private long balance;
}
```

This implementation is valid when used in single-thread programs. However, it cannot safely be used in multithreaded programs!

Race hazard

Balance	Withdrawal1	Widthdrawal2
1000000	`widthdraw(1000000)`	`widthdraw(1000000)`
1000000	`amount <= balance`	
1000000		`amount <= balance`
1000000	`newBalance = ...;`	
1000000		`newBalance = ...;`
0	`balance = ...;`	
0		`balance = ...;`
0	`return true;`	
0		`return true;`

Class `Account` is not **thread-safe**



Atomic operations

Java guarantees that reading and assignment of variables of primitive types, except `long` and `double`, are atomic (cannot be interrupted).

All other operations should be explicitly *synchronized* to ensure atomicity.

Critical regions

A **critical region** is a section of program code that should be executed by only one thread at a time.

Java provides a *synchronization* mechanism to ensure that, while a thread is executing statements in a critical region, no other thread can execute statements in the same critical region at the same time.

Synchronization may be applied to methods or a block of statements.

Synchronized instance method

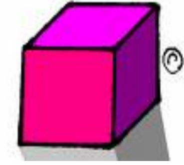
```
class MyClass {
    synchronized void aMethod() {
        «do something»
    }
}
```

The entire method body is the critical region.

Synchronization for the bank account example

```
public class Account {
    // ...
    public synchronized boolean withdraw(long amount) {
        if (amount <= balance) {
            long newBalance = balance - amount;
            balance = newBalance;
            return true;
        }
        return false;
    }

    private long balance;
}
```

This implementation is thread-safe.



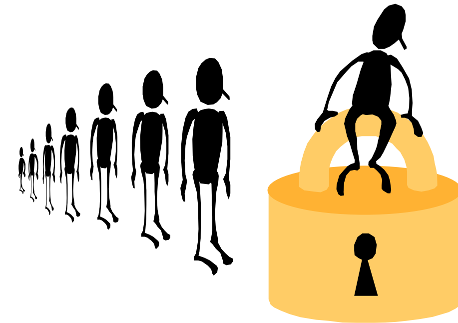
Synchronized block

```
synchronized(exp) {
    «do something»
}
```

where `exp` must be of reference type.

The statements enclosed in the synchronized block comprise the critical region.

Locks



The synchronization mechanism is implemented by associating each object with a **lock**.

A thread must obtain *exclusive possession* of a lock before entering a critical region.

- For a synchronized instance method, the lock associated with the receiving object `this` is used.
- For a synchronized block, the lock associated with the result of the expression `exp` is used.

Synchronized instance methods

```
class MyClass {
    synchronized void aMethod() {
        «do something»
    }

    synchronized void anotherMethod() {
        «do something»
    }
}
```

The union of the bodies of the synchronized methods is one critical region.

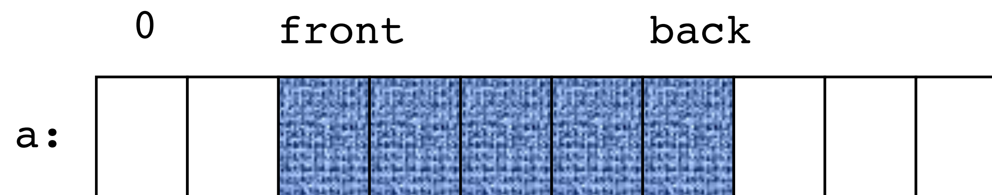
Release of locks

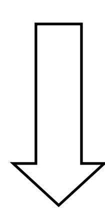
A lock is released when the thread leaves the critical region.

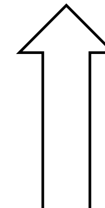
The lock may also be released temporarily before leaving the critical region, when the `wait` method is invoked.

A bounded queue

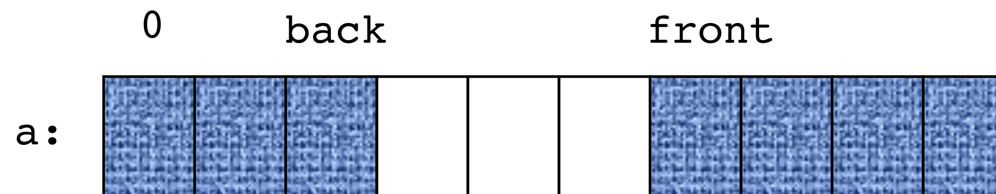
(implemented using a circular array)



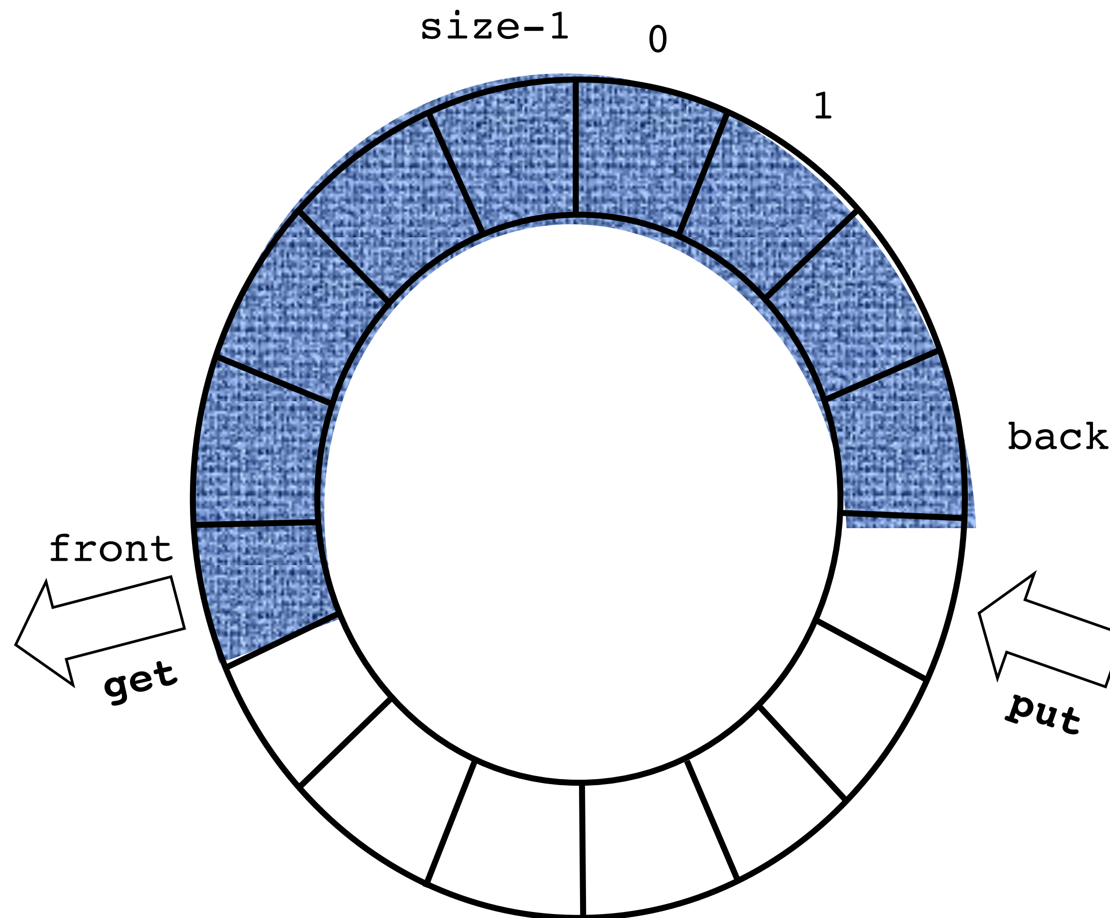




get **put**



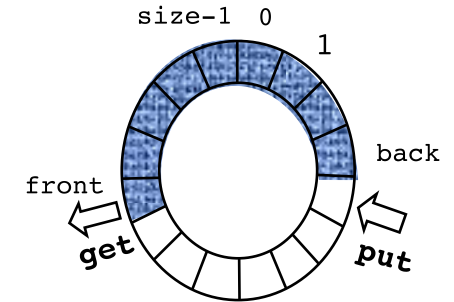
Circular array



A bounded queue
(sequential version)

```
public class BoundedQueue {
    protected Object[] a;
    protected int front, back, size, count;

    public BoundedQueue(int size) {
        if (size > 0) {
            this.size = size;
            a = new Object[size];
            back = size - 1;
        }
    }

    public boolean isEmpty() { return count == 0; }
    public boolean isFull() { return count == size; }
    public int getCount() { return count; }

    // put, get
}
```

cont'd on next page



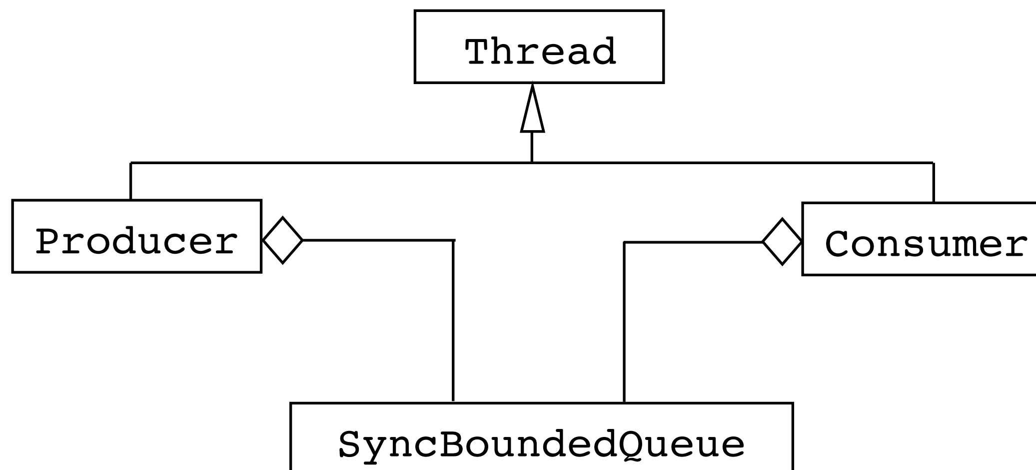

```
public void put(Object obj) {
    if (obj != null && !isFull()) {
        back = (back + 1) % size;
        a[back] = obj;
        count++;
    }
}


public Object get() {
    if (!isEmpty()) {
        Object result = a[front];
        a[front] = null;
        front = (front + 1) % size;
        count--;
        return result;
    }
    return null;
}
```

Bounded queue
(fully synchronized version)

```
public class SyncBoundedQueue extends BoundedQueue {
    public SyncBoundedQueue(int size) { super(size); }

    public synchronized boolean isEmpty() { return super.isEmpty(); }
    public synchronized boolean isFull() { return super.isFull(); }
    public synchronized int getCount() { return super.getCount(); }
    public synchronized void put(Object obj) { super.put(obj); }
    public synchronized Object get() { return super.get(); }
}
```

Application example

A typical use of the `SyncBoundedQueue` class is to serve as a buffer between a producer and a consumer, both of which are threads.



The Producer class

```
public class Producer extends Thread {
    protected BoundedQueue queue;
    protected int n;

    public Producer(BoundedQueue queue, int n) {
        this.queue = queue; this.n = n;
    }

    public void run() {
        for (int i = 0; i < n; i++) {
            queue.put(new Integer(i));
            System.out.println("produce: " + i);
            try {
               sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) {}
        }
    }
}
```

The Consumer class

```
public class Consumer extends Thread {
    protected BoundedQueue queue;
    protected int n;

    public Consumer(BoundedQueue queue, int n) {
        this.queue = queue; this.n = n;
    }

    public void run() {
        for (int i = 0; i < n; i++) {
            Object obj = queue.get();
            if (obj != null)
                System.out.println("\tconsume: " + obj);
            try {
                sleep((int)(Math.random() * 400));
            } catch (InterruptedException e) {}
        }
    }
}
```

Test program

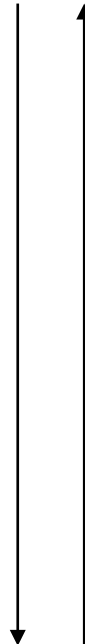
```
public static void main(String args[]) {
    BoundedQueue queue = new SyncBoundedQueue(5);
    new Producer(queue, 15).start();
    new Consumer(queue, 10).start();
}
```

Java applications always begin with the `main()` method, called in a **user thread**.

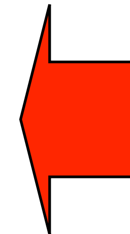
An application terminates when all user threads have terminated, or when the `exit` method from `System` or `Runtime` is called.

Output

```
produce: 0
    consume: 0
produce: 1
produce: 2
produce: 3
produce: 4
    consume: 1
produce: 5
produce: 6
produce: 7
    consume: 2
produce: 8
produce: 9
    consume: 3
```



```
produce: 10
produce: 11
produce: 12
    consume: 4
produce: 13
produce: 14
    consume: 5
    consume: 6
    consume: 8
    consume: 10
    consume: 13
```



The producer produces items faster than the consumer consumes the items.
This implementation causes items to be lost!

Cooperation among threads

Synchronization ensures mutual exclusion of two or more threads in the critical regions.

However, there is also a need for threads to **cooperate**.

For this purpose, **guarded suspension** is used. A thread may be suspended until a certain condition (*guard*) becomes true, at which time its execution may be continued.

`wait`, `notify` and `notifyAll`

Guarded suspension can be implemented using the the `wait()`, `notify()` and `notifyAll()` methods of the `Object` class.

The `wait` method should be invoked when a thread is temporarily unable to continue and we want other threads to proceed.

The `notify()` and `notifyAll()` methods should be invoked when we want a thread to notify other threads that they may proceed.

`wait`

- Must only be invoked in a synchronized method or a synchronized block
- The thread is suspended and waits for a `notify`-signal
- Releases the lock associated with the receiving object
- The awakened thread must reobtain the lock before it can resume at the point immediately following the invocation of the `wait()` method.
- The variants `wait(long millis)` and `wait(long millis, int nanos)` make it possible to specify a maximum wait time.

`notify`

- Must only be invoked in a synchronized method or a synchronized block.
- Wakes up one of the suspended threads waiting on the given object.
- `notifyAll()` wakes up all threads waiting on a given object. At most one of them proceeds.

Cooperation among producer and consumer

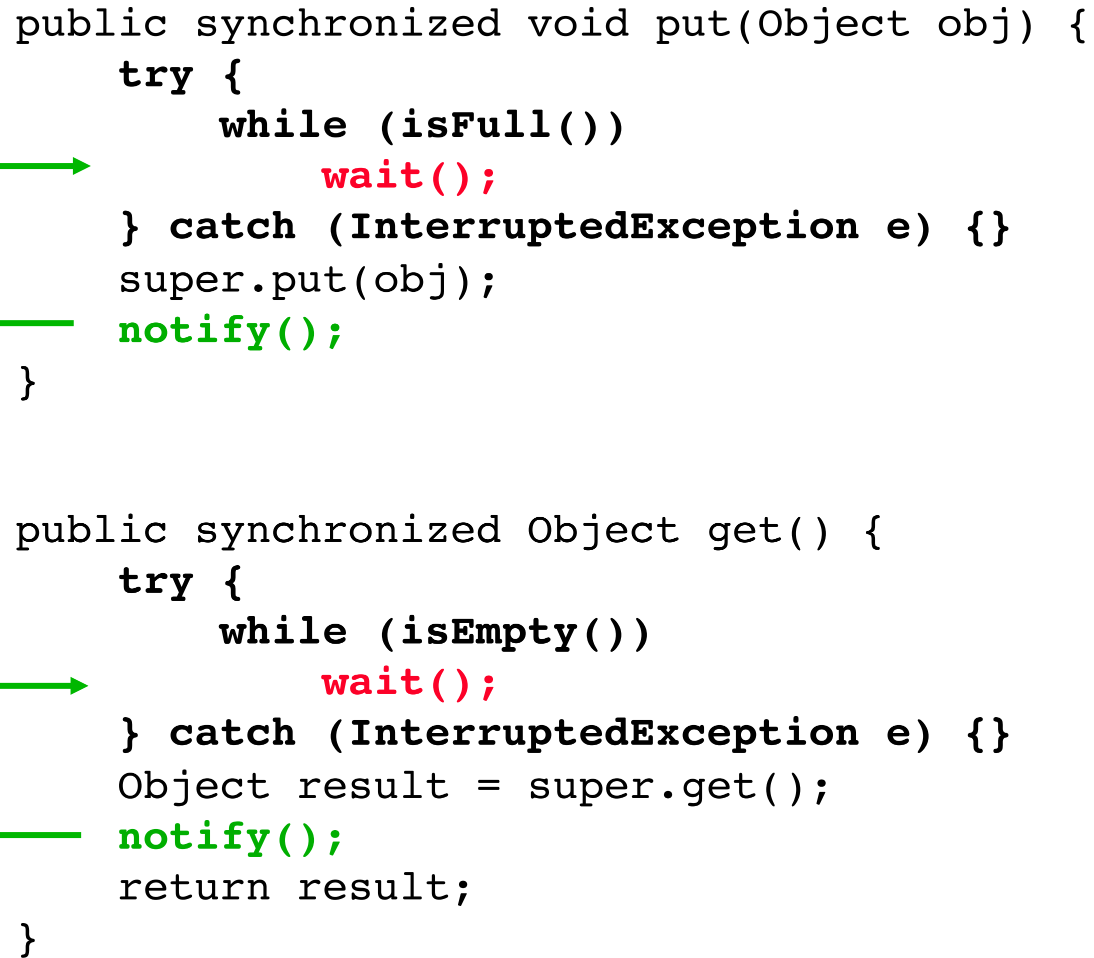
When the producer attempts to put a new item into the queue while the queue is full, is should wait for the consumer to consume some of the items in the queue, making room for the new item.

When the consumer attempts to retrieve an item from the queue while the queue is empty, it should wait for the producer to produce items and put them into the queue.

Bounded queue with guarded suspension

```
public class BoundedQueueWithGuard extends BoundedQueue {
    public BoundedQueueWithGuard(int size) { super(size); }

    public synchronized boolean isEmpty() { return super.isEmpty(); }
    public synchronized boolean isFull() { return super.isFull(); }
    public synchronized int getCount() { return super.getCount(); }

    public synchronized void put(Object obj) { ... }
    public synchronized Object get() { ... }

    public static void main(String args[]) {
        BoundedQueueWithGuard queue = new BoundedQueueWithGuard(5);
        new Producer(queue, 15).start();
        new Consumer(queue, 10).start();
    }
}
```

cont’d on next page


public synchronized void put(Object obj) {
try {
while (isFull())
wait();
} catch (InterruptedException e) {}
super.put(obj);
notify();
}
public synchronized Object get() {
try {
while (isEmpty())
wait();
} catch (InterruptedException e) {}
Object result = super.get();
notify();
return result;
}

Output

```
produce: 0
    consume: 0
produce: 1
produce: 2
produce: 3
produce: 4
    consume: 1
produce: 5
produce: 6
produce: 7
    consume: 2
    consume: 3
produce: 8
produce: 9
    consume: 4
```

```
    consume: 5
produce: 10
    consume: 6
produce: 11
produce: 12
    consume: 7
    consume: 8
produce: 13
produce: 14
    consume: 9
    consume: 10
    consume: 11
    consume: 12
    consume: 13
    consume: 14
```

No items are lost

Design guidelines

The `wait()` method should always be called in a loop testing the condition being waited upon.

When a synchronized object changes its state it should usually call the `notifyAll()` method. In this way all waiting threads get a chance to check if they are able to resume their execution.

Problems with concurrency

- Starvation
- Dormancy
- Deadlock
- Premature termination

Starvation

Some threads don’t make progress.

Happens if there always exist some threads with a higher priority, or if a thread with the same priority never releases the processor.

Avoid “busy waiting”, such as

```
while (x < 2)
    ;  // nothing
```

Dormancy

A waiting thread is never awakened.

Happens if a waiting thread is not notified.

When in doubt, use the `notifyAll()` method.

Deadlock

Two or more threads are blocked, each waiting for resources held by another thread.

It is usually caused by two or more threads competing for resources and each thread requiring exclusive possession of those resources simultaneously.

```
synchronized(a) {
    synchronized(b) {
        ...
    }
}
```

thread1

```
synchronized(b) {
    synchronized(a) {
        ...
    }
}
```

thread2

An example of deadlock

```
public class DiskDrive {
    public synchronized InputStream openFile(String fileName) {
        ...
    }

    public synchronized void writeFile(String fileName,
                                       InputStream in) {
        ...
    }

    public synchronized void copy(DiskDrive destination,
                                  String fileName) {
        InputStream in = openFile(fileName);
        destination.writeFile(fileName, in);
    }
}
```

Scenario

thread1: `c.copy(d, file1)`	**thread2:** `d.copy(c, file2)`
Invoke `c.copy(...)`	
Obtains lock of `c`	
	Invoke `d.copy(...)`
	Obtains lock of `d`
Invoke `c.openFile(...)`	
	Invoke `d.openFile(...)`
Invoke `d.writeFile(...)`	
Unable to obtain lock of `d`	
	Invoke `c.writeFile(...)`
	Unable to obtain lock of `c`

Deadlock!

Prevention of deadlock

The runtime system is neither able to detect nor prevent deadlocks. It is the responsibility of the programmer to ensure that deadlock cannot occur.

An often applied technique is **resource ordering**:

```
synchronized(a) {
    synchronized(b) {
        ...
    }
}
```

thread1

```
synchronized(a) {
    synchronized(b) {
        ...
    }
}
```

thread2

Deadlock cannot occur

Premature termination

When a thread is terminated before it should be, impeding the progress of other threads.

The `Thread` class

```
public class Thread implements Runnable {
    public Thread();
    public Thread(Runnable target);
    public Thread(String name);
    public Thread(Runnable target, String name);
    public Thread(ThreadGroup group, String name);
    public Thread(ThreadGroup group, Runnable target);
    public Thread(ThreadGroup group, Runnable target, String name);

    public static final int MIN_PRIORITY = 1;
    public static final int NORM_PRIORITY = 5;
    public static final int MAX_PRIORITY = 10;
```

cont'd on next page

```java
public static int activeCount();
public static void dumpStack();
public static boolean interrupted();
public static native void sleep(long millis)
                          throws InterruptedException;
public static native void sleep(long millis, int nanos)
                          throws InterruptedException;
public static native void yield();

public final String getName();
public final int getPriority();
public final ThreadGroup getThreadGroup();
public void interrupt();
public final native boolean isAlive();
public final native boolean isDaemon();
public final native boolean isInterrupted();
```

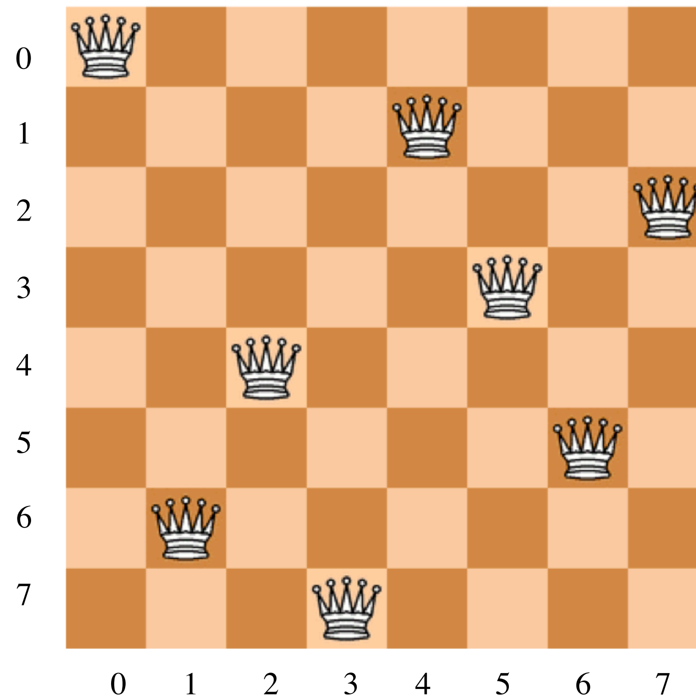
A daemon thread is a (background) thread that does not prevent the JVM from exiting when the program finishes but the thread is still running.

cont'd on next page

```
    public final synchronized void join()
                                   throws InterruptedException;
    public final synchronized void join(long millis)
                                   throws InterruptedException;
    public final synchronized void join(long millis, int nanos)
                                   throws InterruptedException;
    public void run();
    public final void setDaemon(boolean on);
    public final void setName(String name);
    public final void setPriority(int newPriority);
    public String toString();
}
```

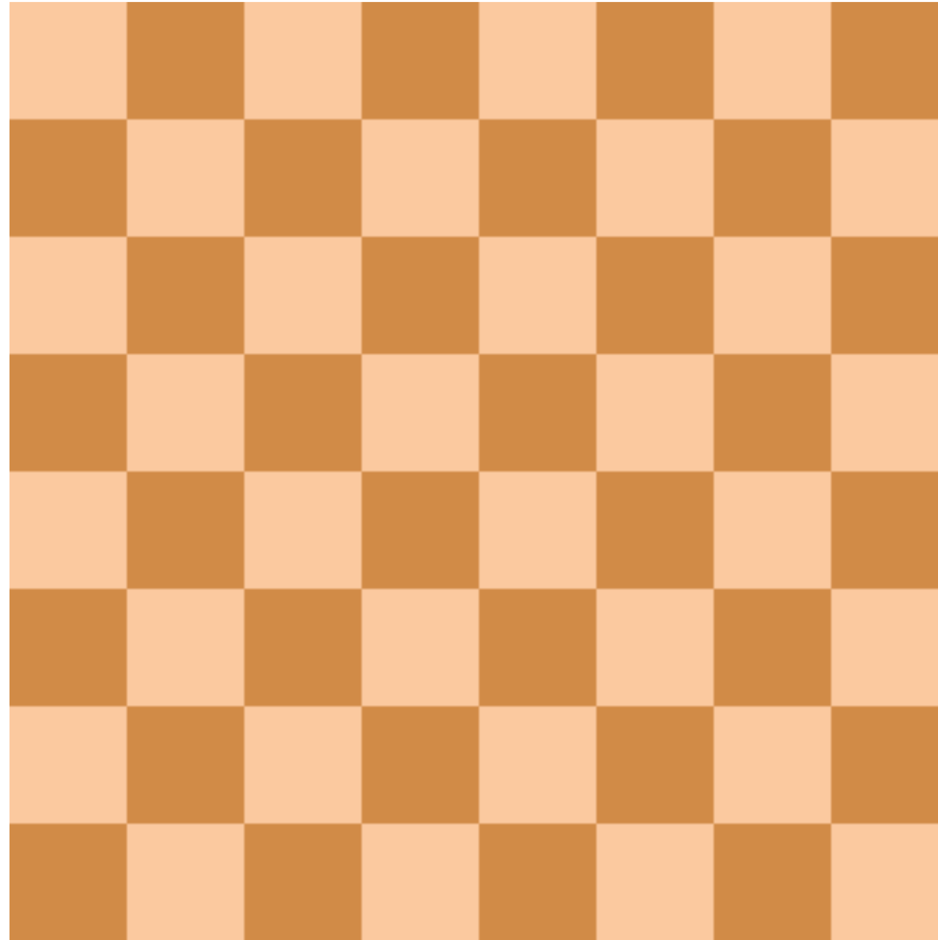
Solution of the 8-queen problem

Problem: Place 8 queens on a chessboard so that no queen is under attack by another; that is, there is at most one queen at each row, column and diagonal.



One out of the 92 solutions

Animation of backtracking



Java solution of the *N*-Queens problem

Given that *r* non-attacking queens already have been placed on the rows 0, ..., *r*-1, try to find all solutions in which a queen is placed in row *r* in all possible ways.

`q[c]`: Column c is occupied.
`up[r + c]`: Up-diagonal is occupied.
`down[c – r + n – 1]`: Down-diagonal is occupied.

```
static void tryRow(int r) {
    for (int c = 0; c < n; c++) {
         if (!q[col] && !up[r + c] && !down[c - r + n - 1]) {
             if (r == n - 1)
                 solutions++;
             else {
                 q[c] = up[r + c] = down[c - r + n - 1] = true;
                 tryRow(r + 1);
                 q[c] = up[r + c] = down[c - r + n - 1] = false;
             }
         }
    }
}
```

Class NQueens

```
public class NQueens {
    static int n;
    static boolean[] q, up, down;
    static int solutions;

    public static void main(String[] args) {
        n = Integer.parseInt(args[0]);
        q = new boolean[n];
        up = new boolean[2 * n - 1];
        down = new boolean[2 * n -1];
        tryRow(0);
        System.out.println("n = " + n + ", solutions = " + solutions);
    }

    static void tryRow(int row) {...}
}
```

Test runs

on a 2.3GHz i7 quad core MacBook Pro

Output	CPU time
n = 8, solutions = 92	0m0.132s
n = 9, solutions = 352	0m0.138s
n = 10, solutions = 724	0m0.150s
n = 11, solutions = 2680	0m0.152s
n = 12, solutions = 14200	0m0.211s
n = 13, solutions = 73712	0m0.578s
n = 14, solutions = 365596	0m2.891s
n = 15, solutions = 2279184	0m17.795s
n = 16, solutions = 14772512	2m0.479s
n = 17, solutions = 95815104	14m42.369s
n = 18, solutions = 666090624	108m27.237s

The current world record is
n = 26, solutions = 22,317,699,616,364,044

Parallelizing the program

Use n threads, one for each possible placement of a queen on row 0.

```
public class NQueens {
    static int n;

    public static void main(String[] args) {
        n = Integer.parseInt(args[0]);
        Task[] task = new Task[n];
        for (int col = 0; col < n; col++)
            (task[col] = new Task(col)).start();
        int solutions = 0;
        for (int col = 0; col < n; col++)
            solutions += task[col].getSolutions();
        System.out.println("n = " + n + ", solutions = " + solutions);
    }

    static class Task extends Thread { ... }
}
```

Class Task

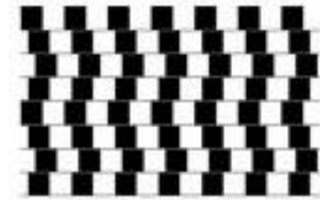
```
static class Task extends Thread {
    private int col;
    private boolean[] q, up, down;
    private int solutions;

    Task(int col) {
        this.col = col;
        q = new boolean[n];
        up = new boolean[2 * n - 1];
        down = new boolean[2 * n -1];
    }
```

```
    public void run() {
        q[col] = up[col] = down[col + n - 1] = true;
        tryRow(1);
    }

    void tryRow(int r) { ... }

    int getSolutions() {
        try {
            join();
            return solutions;
        } catch (InterruptedException e) {
            throw new RuntimeException("Internal task error\n");
        }
    }

}
```

Comparison of runnning times

n	Without threads	With threads
8	0m0.132s	0m0.131s
9	0m0.138s	0m0.139s
10	0m0.150s	0m0.149s
11	0m0.152s	0m0.148s
12	0m0.211s	0m0.168s
13	0m0.578s	0m0.234s
14	0m2.891s	0m0.650s
15	0m17.795s	0m3.419s
16	2m0.479s	0m22.436s
17	14m42.369s	2m41.301s
18	108m27.237s	20m27.125s

`java.util.concurrent`



Thread pools with the Executor framework.

An **Executor** can be described as a collection of threads and a work queue of tasks waiting to get executed. The treads are constantly running and checking the work queue for new work.

Executors

An executor is an object that executes `Runnable` tasks.

Separates task submission from execution policy:

- Use `anExecutor.execute(aRunnable)`
- Instead of `new Thread(aRunnable).start()`

Creation of Executors

Sample `ExecutorService` implementations of `Executors`

- **`newSingleThreadExecutor`**
  A pool of one thread, working from an unbounded queue
- **`newFixedThreadPool(int N)`**
  A fixed pool of N threads, working from an unbounded queue
- **`newCachedThreadPool`**
  A variable size pool that grows as needed and shrinks when idle
- **`newScheduledThreadPool(int N)`**
  Pool for executing tasks after a given delay, or periodically

A simple example

```
class Doubler implements Runnable {
    Doubler(int[] a, int from, int to) {
        this.a = a;
        this.from = from;
        this.to = to;
    }

    private int[] a;
    private int from, to;

    public void run() {
        for (int i = from; i <= to; i++)
            a[i] *= 2;
    }
}
```

cont'd on next page

A simple example (Cont’d)

```
public class ExecutorExample {
    public static void main(String[] args) {
        int[] a = { 1, 4, 5, 6, 8, 7, 4, 3, 2 };
        ExecutorService executor =
            Executors.newFixedThreadPool(3);
        executor.execute(new Doubler(a, 0, 2));
        executor.execute(new Doubler(a, 3, 5));
        executor.execute(new Doubler(a, 6, 8));
        executor.shutdown();
        while (!executor.isTerminated())
            ;
        for (int i = 0; i < a.length; i++)
            System.out.print(a[i] + " ");
        System.out.println();
     }
}
```

`Callable` and `Future`

In case you expect your threads to return a result you can use the `Callable` interface.

`Callable` uses generics to define the type of object which is returned.

If you submit a `Callable` object to an executor, a `Future` object is returned. To retrieve the result use the `get()` method (blocks until ready).

A simple example

```
class Adder implements Callable<Integer> {
    Adder(int[] a, int from, int to) {
        this.a = a;
        this.from = from;
        this.to = to;
    }

    int[] a;
    int from, to;

    public Integer call() {
        int sum = 0;
        for (int i = from; i <= to; i++)
            sum += a[i];
        return sum;
    }
}
```

cont'd on next page

A simple example (Cont'd)

```
public class CallableFutureExample {
    public static void main(String[] args) {
        int[] a = { 1, 4, 5, 6, 8, 7, 4, 3, 2 };
        ExecutorService executor =
            Executors.newFixedThreadPool(3);
        List<Future<Integer>> futures =
            new ArrayList<Future<Integer>>();
        futures.add(executor.submit(new Adder(a, 0, 2)));
        futures.add(executor.submit(new Adder(a, 3, 5)));
        futures.add(executor.submit(new Adder(a, 6, 8)));
        executor.shutdown();
        int sum = 0;
        for (Future<Integer> f : futures) {
            try {
                sum += f.get();
            } catch(Exception e) { e.printStackTrace(); }
        }
        System.out.println(sum);
    }
}
```

An N-Queens program that uses `Callable` and `Future`

```
static class Task implements Callable<Integer> {
    private int col;
    private boolean[] q, up, down;
    private int solutions;

    Task(int col) { ... }

    public Integer call() {
        q[col] = up[col] = down[col + n - 1] = true;
        tryRow(1);
        return solutions;
    }

    void tryRow(int row) { .. }
}
```

The main method

```
public static void main(String[] args) {
    n = Integer.parseInt(args[0]);
    ExecutorService executor =
        Executors.newFixedThreadPool(n);
    List<Future<Integer>> futures = new ArrayList<>();
    for (int col = 0; col < n; col++)
        futures.add(executor.submit(new Task(col)));
    executor.shutdown();
    int solutions = 0;
    for (Future<Integer> f : futures) {
        try {
            solutions += f.get();
        } catch(Exception e) { e.printStackTrace(); }
    }
    System.out.println("n = " + n + ", solutions = " + solutions);
}
```