ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν Πανεπιστήμιον Αθηνών

Διάλεξη
Εισαγωγή στη Java, Μέρος Γ

- Νήματα (Threads) στην Java
- Συγχρονισμός
- Producer-Consumer problem
- Singleton Pattern

- Νήματα (Threads) στην Java

Εισαγωγή στα νήματα (1)

- Σε υπολογιστικά περιβάλλοντα **εφαρμογές** «ανταγωνίζονται» να αποκτήσουν την δυνατότητα πρόσβασης και εκτέλεσης στην CPU (concurrent programming)
- Δυο ειδών οντότητες εκτέλεσης
  1. τις **διεργασίες** (processes)
  2. τα **νήματα** (threads)
- Η CPU επεξεργάζεται κάθε στιγμή μία οντότητα
  - Ψευδοπαραλληλία, όχι ταυτόχρονη εκτέλεση
- Ο χρόνος επεξεργασίας μοιράζεται ανάμεσα στα νήματα
  - Ο Scheduler αναθέτει χρονοθυρίδες, κύκλους επεξεργασίας της CPU

https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Εισαγωγή στα νήματα (2)

Διεργασίες

- Οι διεργασίες αποτελούν **αυτόνομες**, **ανεξάρτητες εφαρμογές** (προγράμματα) που διαθέτουν δικό τους memory space
  - **Δεν μοιράζονται κάτι μεταξύ τους**: πόρους π.χ. αρχεία ή μνήμη
- Το εκάστοτε λειτουργικό σύστημα υποστηρίζει την ύπαρξη πόρων για την επικοινωνία μεταξύ διεργασιών
  - Inter Process Communication-IPC resources στον ίδιο υπολογιστή
  - pipes, sockets…σε διαφορετικούς υπολογιστές

Εισαγωγή στα νήματα (3)

Νήματα

Τα νήματα γεννώνται και υπάρχουν μέσα σε μια διεργασία

- κάθε διεργασία στη JAVA έχει τουλάχιστον ένα νήμα, **το βασικό νήμα** (**main thread)**
- μια διεργασία μπορεί να δημιουργήσει ένα ή περισσότερα νήματα ώστε να εκτελέσει πολλά tasks «ταυτόχρονα»
- μία διεργασία τερματίζει μόνο εάν τερματίσουν όλα τα νήματά της
- εάν τερματίσουμε μία διεργασία, τερματίζουμε όλα τα νήματά της

Εισαγωγή στα νήματα (4)

Νήματα

- Τα νήματα **μοιράζονται πόρους της διεργασίας που** τα δημιουργεί όπως μνήμη, αρχεία κλπ αλλά διαθέτουν και το δικό τους address space.
- Μπορούν να χαρακτηριστούν και ως **lightweight processes.** Παρέχουν και αυτά ένα χώρο εκτέλεσης μόνο που η δημιουργία ενός thread **απαιτεί λιγότερους πόρους** να δεσμευτούν από το λειτουργικό σύστημα.
- Η αλλαγή στην προσπέλαση της CPU μεταξύ threads της ίδιας διεργασίας είναι λιγότερο χρονοβόρα (context switching) σε σχέση με τις διεργασίες.

Τα νήματα στην Java (1)

Η Java μας παρέχει δυο τρόπους για την δημιουργία νημάτων

1. extend την κλάση **Thread** (java.lang.Thread)

2. implement το **Runnable interface** (java.lang.Runnable)

Τα νήματα στην Java **Extends Thread**

1. Extends Thread

- Μια κλάση μπορεί να κάνει extend την **Thread** και να κάνει **override την μέθοδο run()** με σκοπό να ορίσει τα βήματα εκτέλεσης ενός thread. (Να σημειώσουμε πως η Thread κάνει implement το Runnable Interface)
- Ο constructor της κλάσης αυτής μπορεί να καλέσει τον constructor της Thread ρητά με χρήση της super().
- Η κλάση μας κληρονομεί την **μέθοδο start()** που ορίζεται στην κλάση Thread και με την κλήση της από ένα αντικείμενο μπορεί να ξεκινήσει την εκτέλεση του ένα νέο thread.
- Κάνουμε extend την Thread -> **Περιορισμός: δεν μπορούμε να κάνουμε extend καμία άλλη κλάση**

```
public class XThread extends Thread{
    private String threadName = null;

    public XThread(String threadName) {
        this.threadName = threadName;
    }

    @Override
    public void run() {
        System.out.println("Hello from " + threadName + " XThread");
    }
}
```

```
public class DemoThread {
    public static void main(String[] args) {
        XThread myXThread = new XThread("myXThread");
        myXThread.start();
    }
}
```

Τα νήματα στην Java Implements Runnable

2. Implements Runnable

- Μια κλάση μπορεί να κάνει **implement το interface Runnable** αρκεί να γραφτεί κώδικας για την **μέθοδο run()** που θα οριζει τον κύκλο ζωής ενός thread.
- Ακολούθως, ένα **αντικείμενο τύπου Thread** δημιουργείται και του περνάμε σαν όρισμα στον constructor του ένα αντικείμενο από την κλάση που δημιουργήσαμε και κάνει implement το Runnable.
- Η μέθοδος **start()** του αντικειμένου τύπου Thread καλείται και επιστρέφει μόλις γεννηθεί το thread

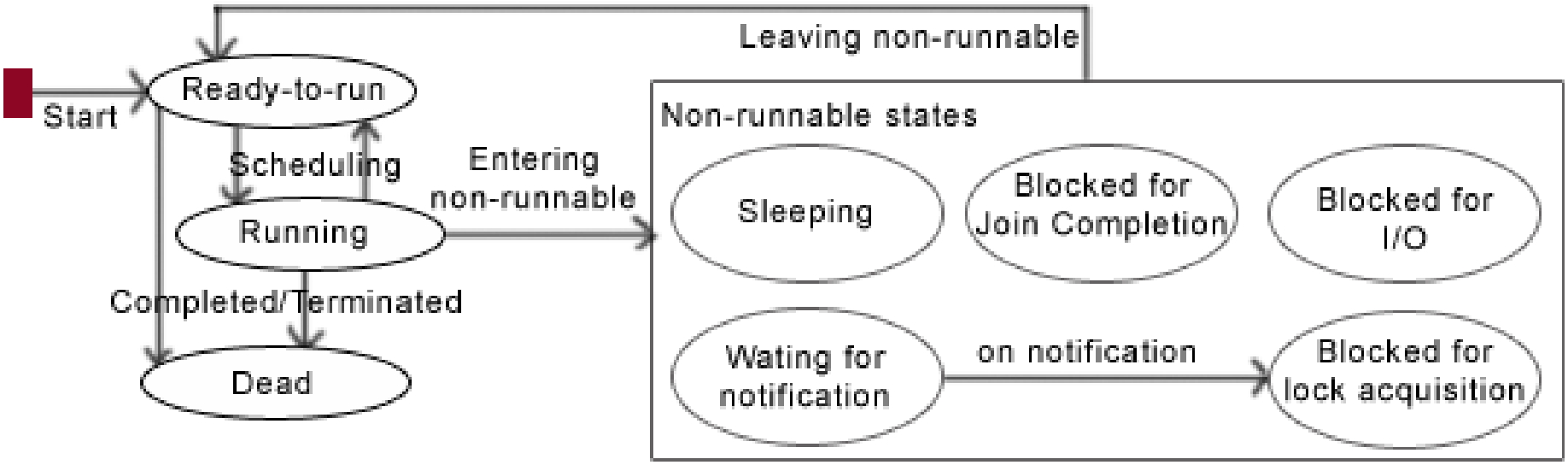
```
public class DemoThread {
    public static void main(String[] args) {
        RunnableThread myRunnable = new RunnableThread("myRunnable");
        Thread t1 = new Thread(myRunnable);
        t1.start();
    }
}
```

```
public class RunnableThread implements Runnable{

    private String threadName = null;

    public RunnableThread(String threadName) {
        this.threadName = threadName;
    }

    @Override
    public void run() {
        System.out.println("Hello from " + threadName + " RunnableThread");
    }
}
```

Συνίσταται η χρήση threads που κάνουν implement το Runnable interface αντί να κληρονομούν την Thread Class:

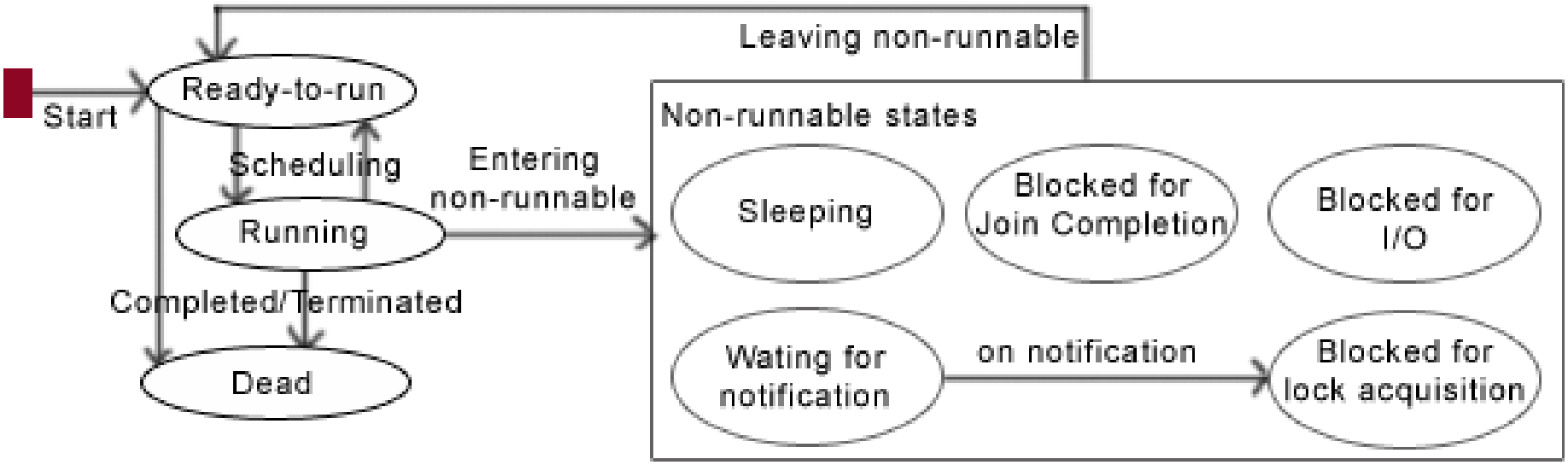
1. Αν κάνουμε extend την Thread, τότε **δεν μπορούμε να κάνουμε extend καμία άλλη κλάση**
2. Το Runnable interface μας υπαγορεύει να κάνουμε implement μόνο μια μέθοδο (την run) και **αποφεύγουμε να κληρονομήσουμε extra overhead της Thread.**

Κύκλος ζωής ενός thread (1)



- **New**: είναι η κατάσταση στην οποία δημιουργούμε ένα αντικείμενο τύπου Thread και **λίγο πριν καλέσουμε την start()**
- **Runnable (Ready-to-run):** Είναι η στιγμή που **καλείται η start()** και από εδώ ξεκινά η ζωή ενός thread. Σε αυτή την κατάσταση αναμένει πρόσβαση στην CPU. Ακόμη, σε αυτή την κατάσταση μπορούμε να μεταπέσουμε από τις **non-runnable** και **running**
- **Running:** Σε αυτή την κατάσταση το thread εκτελείται. Ο scheduler το επιλέγει μέσα από την runnable pool.

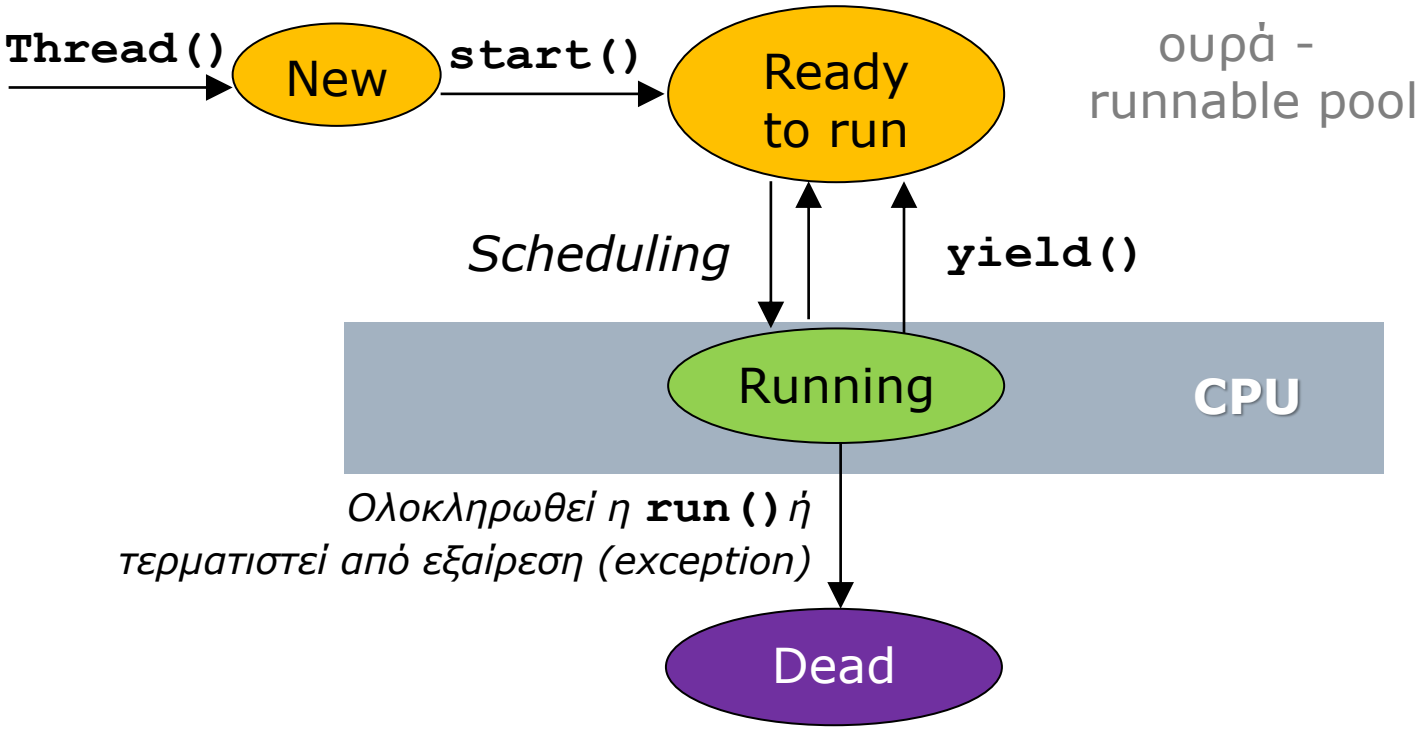
Κύκλος ζωής ενός thread (2)



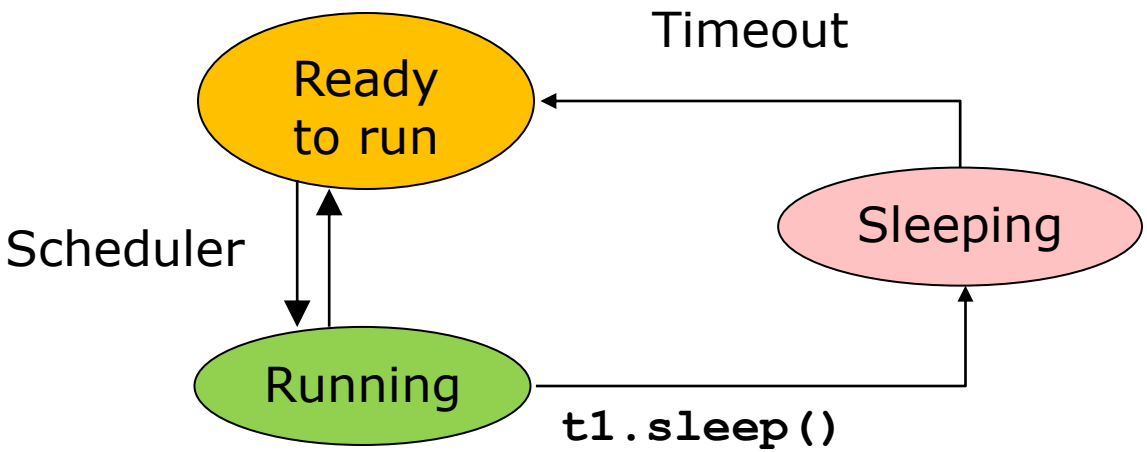
- **Dead:** Το thread σε αυτή την κατάσταση παύει να εκτελεί και γίνεται join από την διεργασία που το δημιούργησε.
- **Non-runnable state:**
  - Κλήση της static: void **sleep(long millisecond)** throws InterruptedException
  - Περιμένει notification από άλλο thread: final void **wait()** throws InterruptedException
  - Το thread περιμένει για **I/O πόρους**
  - Περιμένει τον τερματισμό ενός άλλου thread για να τερματίσει (**joint** completion)
  - Περιμένει μέχρι να κάνει **lock** κάποιο πόρο.

New – Read to run – Running - Dead

```
public class DemoThread {
    public static void main(String[] args) {
        RunnableThread myRunnable = new RunnableThread("myRunnable");
        Thread t1 = new Thread(myRunnable);
        t1.start();
    }
}
```



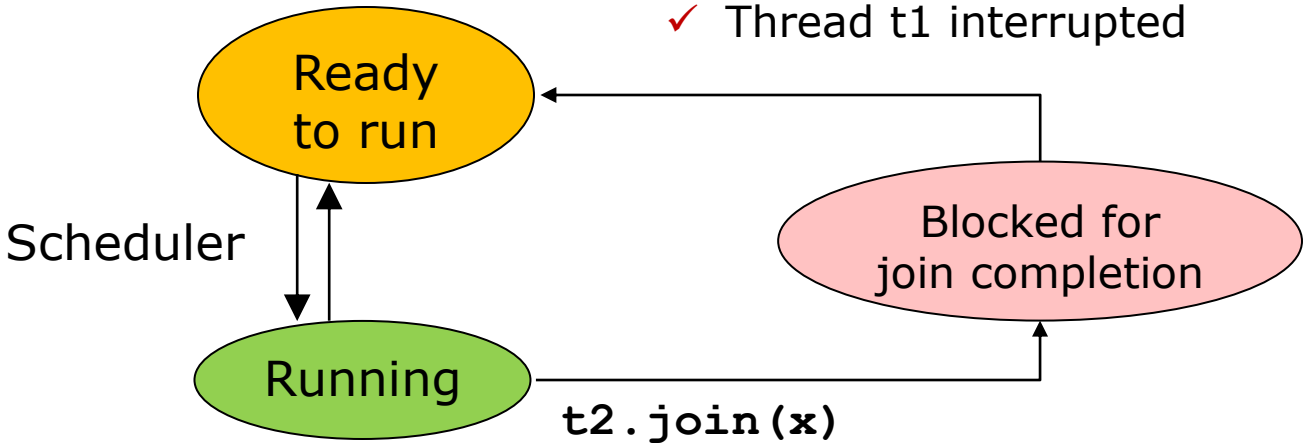
Non runnable state: Sleeping



1. Το thread (t_1) θα παραμείνει στην κατάσταση sleeping όσο χρόνο ορίζει η παράμετρος της sleep(
2. **Δεν απελευθερώνει τα locks** των αντικειμένων που μπορεί να κατέχει.

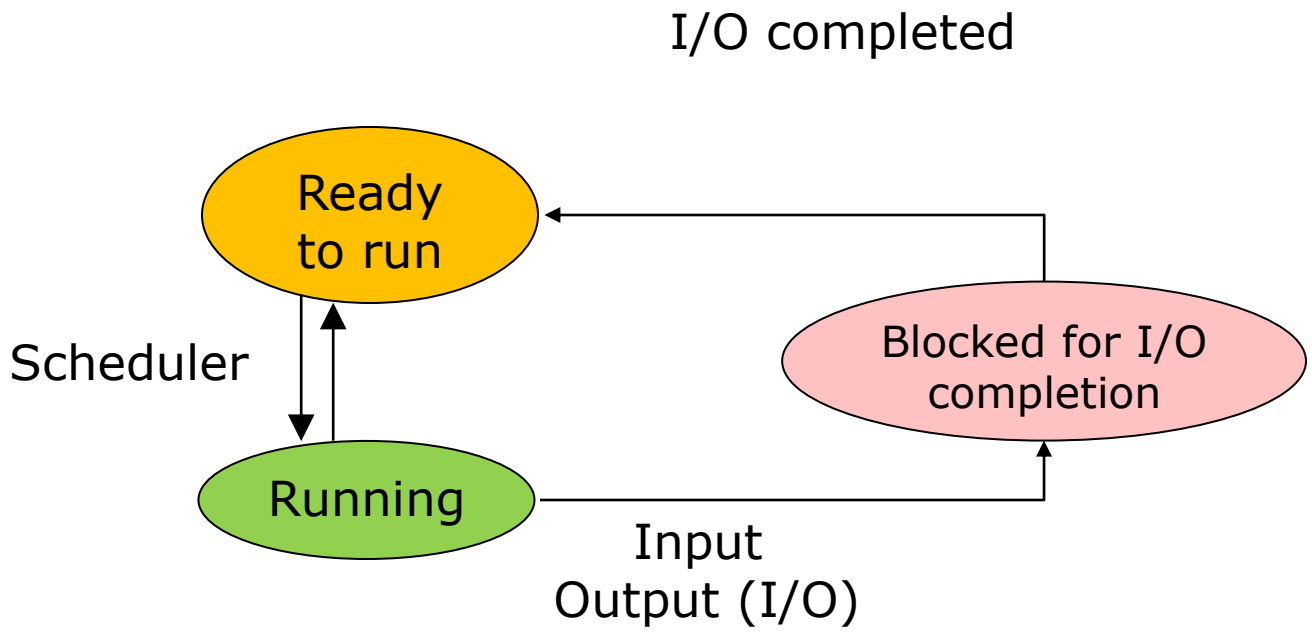
Non runnable state: Blocked for join

- ✓ Ολοκλήρωση thread t2
- ✓ Timeout ύστερα από **x** χρόνο – `t2.join(x)`
- ✓ Thread t1 interrupted

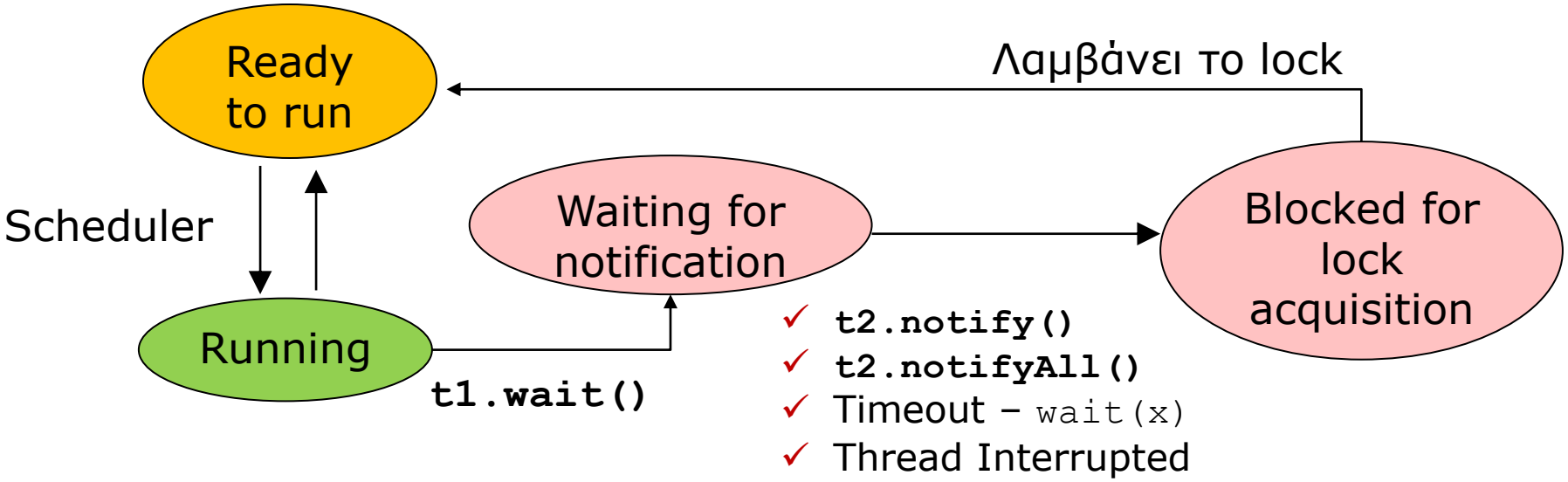


1. Το νήμα t1 καλεί t2.join(x).
2. Θα συνεχίσει η εκτέλεσή του t1 μόλις
   - ✓ ολοκληρωθεί το νήμα t2
   - ✓ περάσει χρόνος x

Non runnable state: Blocked for I/O



Non runnable states: Waiting for notification, Blocked for lock acquisition



1. Έστω το **αντικείμενο z** και ότι η t1 κατέχει τη κλειδαριά (lock) του z.
2. Μπαίνει σε *ουρά αναμονής (waiting)* και *απελευθερώνει* το lock του z (για άλλο thread).
3. Βγαίνει από την ουρά μόλις λάβει notification (notify() από άλλο thread π.χ. t2) και αναμένει το lock (blocked for lock acquisition).
4. Το νήμα t1 και t2 συντονίζονται για πρόσβαση σε κοινό πόρο.

Παράδειγμα

- Το κύριο (main) thread δημιουργεί ένα νέο thread MessageLoop και περιμένει να ολοκληρωθεί
- Εάν το παιδί διαρκεί αρκετά, το main thread το διακόπτει (interrupt)
- start(), isAlive(), join(1000), join(), interrupt()

https://docs.oracle.com/javase/tutorial/essential/concurrency/sleep.html

```
public class MessageLoop implements Runnable {
    public void run() {
        String importantInfo[] = {  "Mares eat oats",
                                    "Does eat oats",
                                    "Little lambs eat ivy",
                                    "A kid will eat ivy too"
                                    };

        try {
            for (int i = 0; i < importantInfo.length; i++) {
                // Pause for 4 seconds
                Thread.sleep(4000);
                // Print a message
                Message.threadMessage(importantInfo[i]);
            }
        } catch (InterruptedException e) {
            Message.threadMessage("I wasn't
        }
    }
}

public class ThreadDemo {
    public static void main(String args[])
      throws InterruptedExceptio

    long patience = 4000 * 2;

    if (args.length > 0) {
        try {
            patience = Long.parseLong(args[0]) * 1000;
        } catch (NumberFormatException e) {
            System.err.println("Argument must be an integer.");
            System.exit(1);
        }
    }

    Message.threadMessage("Starting MessageLoop thread");
    long startTime = System.currentTimeMillis();
    Thread t = new Thread(new MessageLoop());
    t.start();

    Message.threadMessage("Waiting for MessageLoop thread to finish");
    while (t.isAlive()) {
        Message.threadMessage("Still waiting...");

        t.join(1000);
        if (((System.currentTimeMillis() - startTime) > patience)
              && t.isAlive()) {
            Message.threadMessage("Tired of waiting!");
            t.interrupt();
            t.join();
        }
    }
    Message.threadMessage("Finally!");
}
```

```
public class Message {
    static void threadMessage(String message) {
        String threadName =
                Thread.currentThread().getName();
        System.out.println(threadName +
                " " + message);
    }
}
```

Static μέθοδος που τυπώνει το όνομα του thread που την κάλεσε και το μήνυμα που δέχεται ως παράμετρο

Η run ορίζει τον κύκλο ζωής του thread

Σε περίπτωση που το thread λάβει interrupt από κάποιο άλλο thread

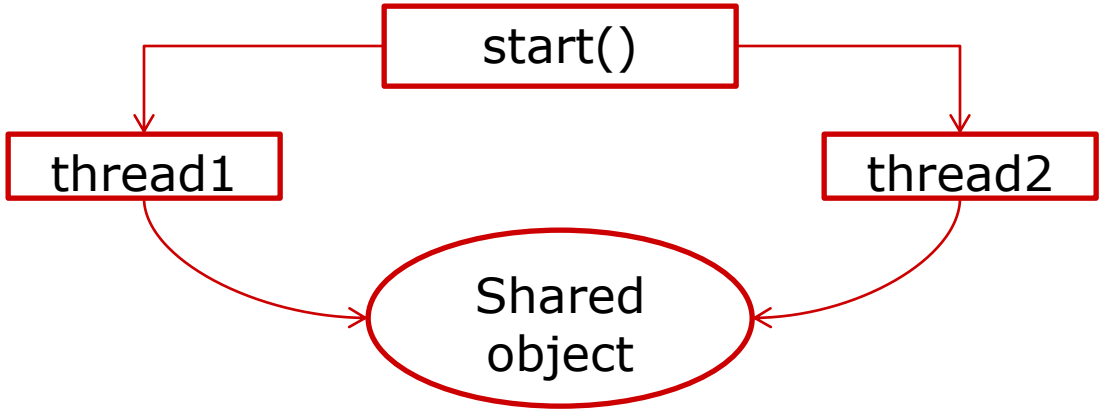
Το αντικείμενο t τύπου Thread μας παρέχει μεθόδους για να χειριστούμε το thread που γεννήσαμε

Η main εκτός από διεργασία αποτελεί και thread και μπορεί να καλέσει την threadMessage

- Συγχρονισμός

Συγχρονισμός threads

- Ζητήματα **ασυνέπειας δεδομένων** προκύπτουν όταν ένα **σύνολο από threads διαμοιράζονται κοινή μνήμη και η πρόσβαση** σε αυτή δεν γίνεται συγχρονισμένα.



- Η Java μας παρέχει την μεθοδολογία για να πετύχουμε **συγχρονισμό στην πρόσβαση ενός κοινού πόρου.**
- Μεγάλη προσοχή !
  - Να σχεδιάσουμε σωστά την ελεγχόμενη πρόσβαση
  - Να αποφύγουμε καταστάσεις τύπου
    - **deadlock:** μπλοκάρονται όλα τα threads
    - **starvation:** μπλοκάρεται συνεχώς ένα thread (ή περισσότερα) από άπληστα threads

Συγχρονισμός και μέθοδοι

```
public class SynchronizedDemo {

    public synchronized void m1() {
        try { Thread.sleep(2000); }
        catch (InterruptedException ie) {}
    }

    public synchronized void m2() {
        try { Thread.sleep(2000); }
        catch (InterruptedException ie) {}
    }

    public static void main(String[] args) throws InterruptedException {
        final SynchronizedDemo t = new SynchronizedDemo();
        Thread t1 = new Thread() { public void run() { t.m1(); } };
        Thread t2 = new Thread() { public void run() { t.m2(); } };

        t1.start();
        Thread.sleep(500);

        t2.start();
        Thread.sleep(500);

        System.out.println(t2.getState());
    }
}
```

Ορίζουμε ως synchronized τις μεθόδους της κλάσης. Κάθε φορά **μόνο ένα νήμα έχει πρόσβαση σε οποιαδήποτε synchronized μέθοδο ενός δεδομένου αντικειμένου**, μπλοκάροντας οποιαδήποτε άλλη κλήση τόσο στην ίδια όσο και στις υπόλοιπες synchronized μεθόδους. Η λειτουργία του synchronized βασίζεται στην ύπαρξη ενός intrinsic lock που διαθέτει το αντικείμενο.

Ορίζουμε σαν διαμοιραζόμενο πόρο το αντικείμενο t

Δημιουργούμε δυο νήματα που ζητούν πρόσβαση στο αντικείμενο μέσα από τις μεθόδους του.

Το πρόβλημα που μπορεί να προκύψει από αυτή τη σχεδίαση είναι να προκληθούν φαινόμενα starvation, μιας και απαγορεύουμε την πρόσβαση στον διαμοιραζόμενο πόρο για όλη τη διάρκεια της κλήσης μιας synchronized μεθοδου. Το νήμα 2 είναι σε κατάσταση **BLOCKED**

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Συγχρονισμός και block εντολών

```
public class SynchronizedDemo {

    private int counter;

    public SynchronizedDemo(int counter) {
        this.counter = counter;
    }

    public void increment() {
        synchronized(this) {
            this.counter++;
        }
        try { Thread.sleep(2000); }
        catch (InterruptedException ie) {}

    }

    public void decrement() {
        synchronized(this) {
            this.counter--;
        }
        try { Thread.sleep(2000); }
        catch (InterruptedException ie) {}
    }
```

```
    public int retrieve() {
        synchronized(this) {
            return this.counter;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        final SynchronizedDemo t = new SynchronizedDemo(0);
        Thread t1 = new Thread() { public void run() { t.increment(); } };
        Thread t2 = new Thread() { public void run() { t.decrement(); } };

        t1.start();
        Thread.sleep(500);

        t2.start();
        Thread.sleep(500);

        System.out.println(t.retrieve());
    }
```

Στην περίπτωση των synchronized statements πρέπει να δηλώσουμε ως προς ποιό αντικείμενο πάμε να κάνουμε lock .

Fine-grained συγχρονισμός. Επικεντρωνόμαστε στον διαμοιραζόμενο πόρο και δεν μπλοκάρουμε την λειτουργία για όλη τη διάρκεια εκτέλεσης της μεθοδου

https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Συγχρονισμός με χρήση αντικειμένων

```
public class SynchronizedDemo {

    private int resourceA;
    private int resourceB;

    private Object lockA = new Object();
    private Object lockB = new Object();

    public SynchronizedDemo(int resourceA, int resourceB) {
        this.resourceA = resourceA;
        this.resourceB = resourceB;
    }

    public int getResourceA() {
        synchronized(lockA) {
            return ++this.resourceA;
        }
    }
    public int getResourceB() {
        synchronized(lockB) {
            return ++this.resourceB;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        final SynchronizedDemo t = new SynchronizedDemo(10, 20);
        Thread t1 = new Thread() { public void run() { System.out.println(t.getResourceA());} };
        Thread t2 = new Thread() { public void run() { System.out.println(t.getResourceB());} };

        t1.start();
        Thread.sleep(500);

        t2.start();
        Thread.sleep(500);

    }
}
```

Ορισμός αντικειμένων με αποκλειστική χρήση για τον συγχρονισμό διαφορετικών πόρων

Η πρόσβαση στον ένα πόρο δεν αποκλείει την ταυτόχρονη προσπέλαση του άλλου

- Suppose resourceA and resourceB, that are never used together.
- All updates of these fields must be synchronized, but there's no reason to prevent an update of resourceA from being interleaved with an update of resourceB — and doing so reduces concurrency by creating unnecessary blocking.

Ατομική Εκτέλεση (Atomic execution)

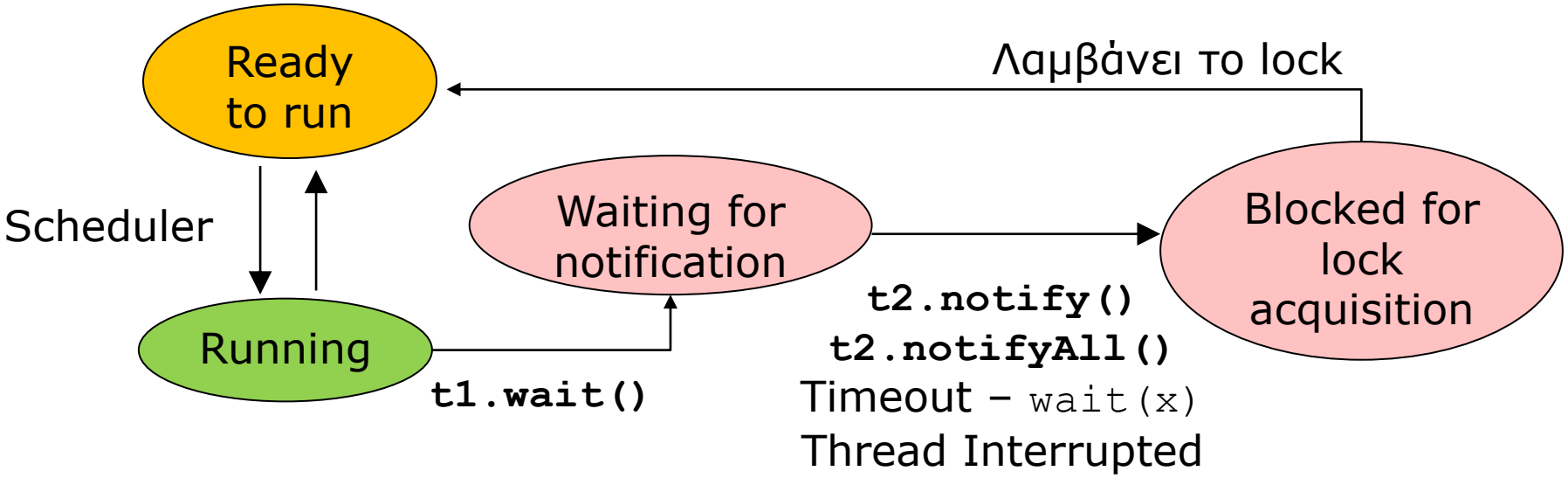
- Ατομική πράξη εκτέλεσης (Atomic action) ορίζουμε την πράξη που εκτελείται **επιτυχώς** και **με τη μια**.
  - Δεν μπορεί να σταματήσει στη μέση. Είτε εκτελείται **ολόκληρη** ή **καθόλου.**
- Ακόμα και απλές πράξεις μπορεί να μην είναι ατομικές λόγω του ότι αποσυντίθενται σε επιμέρους (χωρίς να καταλαβαίνει ο χρήστης).
  - Παράδειγμα: double d; d++;
    - Retrieve the current value of d.
    - Increment the retrieved value by 1. ← Χάσει τη CPU
    - Store the incremented value back in d.
- Παρόλα αυτά οι παρακάτω πράξεις είναι ατομικές:
  - Για primitive variables (εκτος long, double) και references, οι πράξεις read/write ειναι ατομικές.
  - Για όλες τις μεταβλητές που ορίζονται ως `volatile` (πτητικές) (εδώ συμπεριλαμβάνονται οι long, double).
- Οι αλλαγές που κάνουμε σε `volatile` variables είναι αμέσως οράτες σε άλλα νήματα, όλα τα reads/writes πηγαίνουν στην "main memory, δεν είναι cached;
- Ορίζοντας κάτι volatile δεν σημαίνει ότι εξασφαλίζουμε συγχρονισμό!!!

https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html https://dzone.com/articles/java-volatile-keyword-0

- Producer-Consumer problem
  - Παράδειγμα για τις wait(), notify(), notifyAll()

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

wait(), notify(), notifyAll()



Producer-Consumer problem

- Προβλήματα συγχρονισμού όπως το producer-consumer πρόβλημα, απαιτούν πέρα από συγχρονισμένη προσπέλαση σε έναν κοινό πόρο, πρόβλεψη και αποφυγή φαινομένων όπως το polling.
- Χαρακτηριστικό παράδειγμα είναι η περίπτωση όπου **δυο threads διαμοιράζονται έναν κοινό πόρο** (πχ ένα buffer). Ο ρόλος του ενός thread είναι να εισάγει (γράφει) δεδομένα (producer) και του άλλου να διαβάζει (consumer). Ο ***συντονισμός*** είναι απαραίτητος:
  - ο consumer δεν πρέπει να διαβάσει δεδομένα προτού ο producer γράψει κάτι νέο
  - ο producer δεν πρέπει να προσθέσει εάν ο consumer δεν έχει λάβει/διαβάσει τα προηγούμενα δεδομένα
- Σε αυτή την περίπτωση δεν αρκεί μόνο ο **συγχρονισμός αλλά απαιτείται και έλεγχος συνθηκών (συντονισμός)**.
- Η λύση που δίνει η Java στο πρόβλημα αυτό είναι μέσα από τις final μεθόδους της κλάσης Object wait(), notify(), notifyAll().

https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem

```
public class Producer implements Runnable {
    Q q;

    public Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}
```

```
public class Q {
    int n;
    synchronized int get() {
        System.out.println("Got: " + n);
        return n;
    }

    synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: " + n);
    }
```

Ο producer αυξάνει συνεχώς το πεδίο n του q

Κάθε φορά ένα thread θα καλεί μία από τις μεθόδους του q, μπλοκάρεται παράλληλα η κλήση της άλλης

Ο consumer θέλει να βλέπει μοναχά τα updates!!

Διαμοιραζόμενος πόρος

```
public class Consumer implements Runnable {
    Q q;

    public Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}
```

```
public class PC {
    public static void main(String args[]) {
        Q q = new Q();

        new Producer(q);
        new Consumer(q);

        System.out.println("Press Control-C to stop.");
    }
```

Μια ενδεικτική εκτέλεση

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

https://www.tutorialspoint.com/javaexamples/thread_procon.htm

```java
public class Producer implements Runnable {
    Q q;

    public Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}
public class Consumer implements Runnable {
    Q q;

    public Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}
public class PC {
    public static void main(String args[]) {
        Q q = new Q();

        new Producer(q);
        new Consumer(q);

        System.out.println("Press Control-C to stop.");
    }
}
```

```java
public class Q {
    int n;
    boolean valueSet = false;

    public synchronized int get() {
        if(!valueSet) {
            try {
            wait();
            } catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
            }

        System.out.println("Got: " + n);
        valueSet = false;

        notify();

        return n;
    }
    public synchronized void put(int n) {
        if(valueSet) {
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedEx
            }
        }
        this.n = n;
        valueSet = true;

        System.out.println("Put: " + n);

        notify();
    }
}
```

δεν έχει μπει νέα τιμή

έχει μπει νέα τιμή

έχει βάλει νέα τιμή

δεν έχει βάλει νέα τιμή

Η wait() κάνει το τρέχον thread να ελευθερώσει το lock του αντικειμένου q και να κάνει sleep μέχρι κάποιο άλλο thread της στείλει notify()

Η notify() ξυπνά το πρώτο thread που κάλεσε το wait() του αντικειμένου q. Εναλλακτική είναι η χρήση της notifyAll()

Μια ενδεικτική εκτέλεση

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

- Singleton Pattern

Singleton Pattern

- Η Java παρέχει ένα σύνολο από μεθόδους για την αποδοτική σχεδίαση κώδικα (Design Patters).
- Η πιο δημοφιλής από αυτές είναι και η Singleton Pattern και αφορά στην δημιουργία **μοναδικών** αντικειμένων μιας κλάσης.

```
class Singleton{

    private static Singleton instance = null;


    private Singleton(){
            //      Optional Code
    }

    public static synchronized Singleton getInstance()
    {
        if (instance == null){
            instance = new Singleton();
        }
        return instance;
    }

    public Object clone()throws CloneNotSupportedException
    {
        throw new CloneNotSupportedException();
    }

}
```

Το μοναδικό αντικείμενο της κλάσης

Private constructor. Κανείς δεν μπορεί να φτιάξει αντικείμενο για αυτήν την κλάση!

Μόνο στην πρώτη κλήση θα δημιουργηθεί αντικείμενο τύπου Singleton. Με το synchronized αποφεύγουμε να υπάρξουν 2 threads για την αρχικοποίηση

Κάνουμε override την clone() και πετάμε exception στην περίπτωση που πάμε να δημιουργήσουμε με κλώνο δεύτερο αντικείμενο

```
public class SingletonDemo{
        public static void main(String args[]){

            //Compilation error not allowed
            //Singleton obj = new Singleton();

            //create the Singleton Object..
            Singleton obj = Singleton.getInstance();


        }
}
```

Ερωτήσεις

- Μπορεί μια κλάση που υλοποιεί Singleton Design Pattern να κληρονομηθεί;

  Όχι καθώς ο constructor της ορίζεται ως private

- Πότε δημιουργείται το αντικείμενο της κλάσης;

  Την πρώτη φορά που θα κληθεί η getInstance()

- Μπορείτε να σκεφτείται περιπτώσεις χρήσης της Singleton Design Pattern;

  Όταν θέλω να αρχικοποιήσω παραμέτρους για την εφαρμογή μου μόνο μια φορά.
  Όταν θέλω να έχω έναν μοναδικό πόρο που να τον μοιράζονται όλα τα threads της εφαρμογής μου

- Έχετε συναντήσει παράδειγμα singleton κλάσης στο Java API;

  java.lang.Runtime

- Ποιό είναι το λάθος στον ακόλουθο κώδικα;

```
public static Singleton getInstance() {
        synchronized(this) { if (instance == null)  instance = new Singleton(); }
        return instance;
}
```