Πτυχιακή Εργασία με θέμα:

Συντρέχων Προγραμματισμός με Java

Πραξιτέλης Νικόλαος Κουρουπέτρογλου

Τμήμα Εφαρμοσμένης Πληροφορικής

Πανεπιστήμιο Μακεδονίας

Περιεχόμενα

Εισαγωγή

Η παρούσα εργασία παρουσιάζει τις βασικές έννοιες του συντρέχοντος προγραμματισμού και τους μηχανισμούς υλοποίησής τους από τη γλώσσα προγραμματισμού Java. Δίνεται ιδιαίτερη έμφαση στην παρουσίαση όλων των εννοιών με βάση συγκεκριμένα παραδείγματα κώδικα, έτοιμα προς εκτέλεση, έτσι ώστε ο αναγνώστης να είναι σε θέση να ελέγξει άμεσα τη κατανόηση των μηχανισμών που παρουσιάζονται.

Εισαγωγή στον συντρέχοντα προγραμματισμό

Στης μέρες μας κατά τη διάρκεια της ενασχόλησής μας με τον υπολογιστή μας, μπορούμε να εκτελέσουμε πολλές λειτουργίες ταυτόχρονα. Για παράδειγμα εφαρμογές όπως από την αναπαραγωγή μουσικής τη μορφοποίηση ενός αρχείου έως και την ανάγνωση/αποστολή ηλεκτρονικών μηνυμάτων. Αυτό οφείλεται λόγο ότι ένα λειτουργικό σύστημα έχει την ικανότητα να επιτρέπει την ταυτόχρονη εκτέλεση των εργασιών. Ακόμη και μια μόνο εφαρμογή συχνά πρέπει να εκτελεί πολλές λειτουργίες ταυτόχρονα. Για παράδειγμα η εφαρμογή αναπαραγωγής μουσικής και βίντεο πρέπει να φορτώνει διαφορετικά αρχεία ήχου και βίντεο, να τα αποκωδικοποιεί ανάλογα με την κωδικοποίηση του κάθε αρχείου, να τα διοχετεύει στη συσκευή ήχου και να τα αναπαράγει. Η ιδιότητα αυτή ονομάζεται *συντρέχουσα εκτέλεση* (*concurrency*) και το αντίστοιχο λογισμικό *συντρέχων* (*concurrent*).

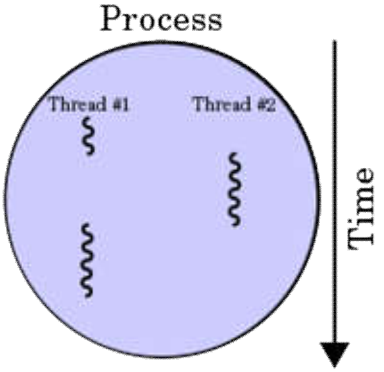
Ο περιλαμβάνει όλες τις λειτουργίες και τους μηχανισμούς που παρέχονται από μια γλώσσα προγραμματισμού έτσι ώστε να εκτελούνται πολλαπλές εργασίες ταυτόχρονα και να επικοινωνούν, να ανταλλάσσουν δεδομένα και να συγχρονίζονται μεταξύ τους. Αυτή η ικανότητα γίνεται πρακτικά ορατή είτε ως παράλληλη εκτέλεση προγραμμάτων σε συστήματα υπολογιστών με πολλούς επεξεργαστές είτε ως χρονομερισμένη εκτέλεση προγραμμάτων σε συστήματα υπολογιστών με ένα επεξεργαστή.

Η Java, όπως και οι περισσότερες γλώσσες προγραμματισμού, προσφέρει συγκεκριμένες κλάσεις για συντρέχοντα προγραμματισμό στα πακέτα java.util.concurrent.

Διεργασίες και νήματα

Για να κατανοήσουμε τον όρο και πως εφαρμόζεται ο συγκεκριμένος τύπος προγραμματισμού, πρέπει να ορίσουμε δύο βασικές έννοιες δομών που απαρτίζουν μια εφαρμογή υπολογιστή.

- Διεργασία (*process*): ορίζεται ως το στιγμιότυπο ενός προγράμματος που εκτελείται. Ένα πρόγραμμα απαρτίζεται από ένα σύνολο υποπρογραμμάτων, δηλαδή ακολουθιών εντολών. Μια διεργασία απαρτίζεται από την εκτέλεση αυτών των εντολών ή μέρος αυτών. Μια διεργασία έχει στη διάθεσή της ένα πλήρες ιδιωτικό σύνολο πόρων, που τις έχουν εκχωρηθεί από το λειτουργικό σύστημα ώστε να μπορέσει να εκτελεστεί.
- Νήμα (*thread*): ορίζεται ως η μικρότερη δυνατή μονάδα (ακολουθία) εκτέλεσης εντολών. Ένα νήμα υπάρχει μόνο μέσα σε μια διεργασία -ή αλλιώς κάθε διεργασία έχει τουλάχιστο ένα νήμα ή και περισσότερα ανεξάρτητες μονάδες εντολών προς εκτέλεση. Πολλαπλά νήματα που βρίσκονται μέσα σε μια διεργασία μοιράζονται τους πόρους της διεργασίας.. Παρακάτω παρουσιάζεται μια διεργασία με δύο νήματα που εκτελούνται ανεξάρτητα το ένα από το άλλο:



Τα νήματα διαφέρουν από τις διεργασίες ως προς τα εξής:

1. Οι διεργασίες είναι ανεξάρτητες οντότητες ενώ τα νήματα είναι ένα “υποσύνολο” μιας διεργασίας
2. Οι διεργασίες περιλαμβάνουν περισσότερες πληροφορίες και δεδομένα (πόρους) από ότι τα νήματα, πολλαπλά νήματα μιας διεργασίας μοιράζονται μεταξύ τους από κοινού αυτές τις πληροφορίες (πόρους).
3. Η συνεργασία διεργασιών σε επίπεδο λειτουργικού συστήματος επιτυγχάνεται με τη Inter Process Communication- IPC Η συνεργασία – επικοινωνία μεταξύ των νημάτων εντός μιας διεργασίας επιτυγχάνετοιχες μεθόδους και συνήθως είναι σημαντικά ταχύτερη από ότι η επικοινωνία διεργασιών.

Τα νήματα μπορούν να θεωρηθούν “ ” διεργασίες με βάση την λειτουργικότητά τους, το μέγεθός τους και το φόρτο εργασίας τους. Αυτό τα καθιστά ευκολότερα διαχειρίσιμα από το λειτουργικό σύστημα. Η βέλτιστη διαχείριση τους από το λειτουργικό σύστημα με βάση το υπολογιστικό σύστημα που είναι εγκατεστημένο επιφέρει τον δίνει την αίσθηση του των εν λόγω νημάτων και την βελτίωση της απόδοσης των εφαρμογών μας.

Στη Java κάθε εφαρμογή μπορεί να ξεκινήσει με ένα κύριο νήμα το οποίο μπορεί να δημιουργήσει άλλα νήματα. Τα νήματα στην Java είναι αντικείμενα τύπου Thread και υπάρχουν δύο βασικές μεθόδους δημιουργίας νημάτων

1. Άμεσα, μέσω της δημιουργία και διαχείριση αντικειμένων Thread από το κύριο νήμα της Java εφαρμογής.
2. Έμμεσα, μέσω της δημιουργία και διαχείριση νημάτων μέσω ενός εκτελεστή (executor).

Βασική διαχείριση νημάτων

Δημιουργία νημάτων στην Java

Στην Java τα νήματα είναι όπως και τα υπόλοιπα στοιχεία της Java ένα object). Η δημιουργία ενός στιγμιότυπου της κλάσης `Thread` γίνεται με δύο τρόπους:

1. Επεκτείνοντας την κλάση `Thread` επικαλύπτοντας την μέθοδο `run()`.
2. Κατασκευάζοντας μια κλάση που υλοποιεί την διεπαφή `Runnable` και έπειτα δημιουργώντας ένα αντικείμενο τύπου `Thread` εισάγοντας το `Runnable` αντικείμενο ως παράμετρο.

Παρακάτω παρουσιάζονται οι αντίστοιχες υλοποιήσεις

-

```
public class HelloThread extends Thread {

        public void run() {
                while (true) {
                        System.out.println("Hello world from thread " +
                                        Thread.currentThread().getName() +
                                        " with Id " + Thread.currentThread().getId());
                }
        }
```

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

                System.out.println("Hello from Thread " +
                                        Thread.currentThread().getName() +
                                        " with Id " + Thread.currentThread().getId());

                Thread thread1 = new HelloThread();
                thread1.start();
        }
}
```

-

```
public class HelloThread implements Runnable {

        public void run() {
                while (true) {
                        System.out.println("Hello world from thread " +
                                        Thread.currentThread().getName() +
                                        " with Id " + Thread.currentThread().getId());
                }
        }
```

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

                System.out.println("Hello from Thread " +
                                        Thread.currentThread().getName() +
                                        " with Id " + Thread.currentThread().getId());

                HelloThread hello1 = new HelloThread();
                Thread thread1 = new Thread(hello1);
                thread1.start();
        }
}
```

-

```
Hello from Thread main with Id 1
Hello world from thread Thread-0 with Id 8
```

```
Hello world from thread Thread-0 with Id 8
Hello world from thread Thread-0 with Id 8
Hello world from thread Thread-0 with Id 8
```

-

Κάθε εφαρμογή Java κατά την εκτέλεσή της από τη Java Virtual Machine (JVM) έχει τουλάχιστον ένα βασικό νήμα το οποίο εκτελεί τη μέθοδο main(). Αυτό είναι το κύριο νήμα εκτέλεσης της εφαρμογής μας. Όταν καλούμε την μέθοδο start() του Thread αντικειμένου, δημιουργούμε ένα νέο νήμα εκτέλεσης εντολών. Έτσι λοιπόν η εφαρμογή μας μπορεί να έχει πολλά νήματα εκτέλεσης εντολών όσο καλούμε την μέθοδο start().

Επίσης πρέπει να σημειωθεί ότι με την δημιουργία ενός αντικειμένου κλάσης Thread, δεν δημιουργείται αυτόματα ένα νέο νήμα. Επίσης, ούτε η κλήση της μεθόδου run(), δημιουργεί ένα νήμα. Μόνο όταν καλούμε την μέθοδο start() δημιουργούμε ένα νήμα εκτέλεσης εντολών.

Συγκρίνοντας τις δύο υλοποιήσεις φαίνεται ότι πιο πρακτική είναι η δεύτερη υλοποίηση με τη διασύνδεση Runnable. Αυτό συμβαίνει διότι ένα αντικείμενο τύπου Runnable που έχει τις ιδιότητες ενός αντικειμένου κλάσης Thread έχει την ικανότητα να επεκταθεί και σε άλλες κλάσεις πέρα από αυτήν της Thread. Αντίθετα, η πρώτη υλοποίηση με επέκταση της κλάσης Thread είναι πιο περιοριστική έναντι της δεύτερης. Συνεπώς για το υπόλοιπο της συγκεκριμένης μελέτης – εργασίας θα χρησιμοποιούμε την δεύτερη υλοποίηση με αντικείμενα τύπου Runnable, τα οποία μας δίνουν την δυνατότητα εφαρμογή APIs διαχείρισης νημάτων υψηλοτέρου επιπέδου. Παρακάτω παρουσιάζονται διαφορές μεθόδους βασικής διαχείρισης νημάτων.

Ανάκληση και επεξεργασία ιδιοτήτων νημάτων

Η κλάση Thread διατηρεί πληροφορίες για τις ιδιότητες των νημάτων που μας βοηθούν να αναγνωρίσουμε το ένα νήμα από το άλλο, αλλά και μπορούμε να αλλάξουμε τις τιμές των συγκεκριμένων χαρακτηριστικών όπως πχ την προτεραιότητά του νήματος. Τα χαρακτηριστικά - ιδιότητες είναι τα εξής:

- Id: Αυτή η ιδιότητα αποθηκεύει ένα μοναδικό αναγνωριστικό για κάθε νήμα. Η μέθοδος ανάκλησης αυτής της ιδιότητας είναι η getId().
- Name: Αυτή η ιδιότητα αποθηκεύει το όνομα του Νήματος. Η μέθοδος ανάκλησης αυτής της ιδιότητας είναι η getName() και η αλλαγή του ονόματος ενός νήματος γίνεται με την μέθοδο setName() της κλάσης Thread.
- Priority: Αυτή η ιδιότητα αποθηκεύει την προτεραιότητα του κάθε αντικειμένου Thread. Η τιμές που δέχεται ως όρισμα είναι από το 1..10, όπου 1 είναι η χαμηλή προτεραιότητά και 10 η υψηλότερη. Η μέθοδος ανάκλησης αυτής της ιδιότητας είναι η getPriority() και η αλλαγή του ονόματος ενός νήματος γίνεται με την μέθοδο setPriority() της κλάσης Thread.
- *State*: Αυτή η ιδιότητα αποθηκεύει την τρέχουσα κατάσταση ενός νήματος. Η μέθοδος ανάκλησης αυτής της ιδιότητας είναι η getState(). Οι τιμές που μπορεί να πάρει είναι οι εξής:
  - NEW: Η Κατάσταση για τα νήματα που μόλις δημιουργήθηκαν
  - RUNNABLE: Ένα νήμα είναι υποψήφιο για εκτέλεση. Στην ουσία το νήμα εκτελείται μέσω της JVM και του υποκείμενου λειτουργικού συστήματος
  - WAITING: Είναι η κατάσταση όπου ένα νήμα αναστέλλει την εκτέλεσή του και περιμένει ένα σήμα από άλλο νήμα. Η Java παρέχει τον μηχανισμό wait-notify. Έτσι ένα νήμα μεταβαίνει σε κατάσταση Waiting όταν καλεί την μέθοδο wait() και ανακαλείται από αυτήν την

κατάσταση όταν καλείται η μέθοδος `notify()`.

- `TIMED_WAITING`: Είναι η κατάσταση όπου ένα νήμα έχει ανασταλεί για ένα ορισμένο χρονικό διάστημα. Αυτό επιτυγχάνεται με την χρήση της μεθόδου `sleep()`. Μετά το πέρας αυτό του χρονικού διαστήματος η κατάσταση του νήματος μετατρέπεται σε `RUNNABLE`.
- `BLOCKED`: Είναι η κατάσταση όπου δύο ή περισσότερα νήματα που συντρέχουν και διαμοιράζονται κοινούς πόρους και είναι αναγκασμένα να βρίσκονται στην κατάσταση αναστολής αν κάποιο από αυτά τα νήματα έχει καταλάβει τους κοινούς πόρους.
- `TERMINATED`: Ένα νήμα εισέρχεται σε αυτήν την κατάσταση όταν έχει ολοκληρωθεί η εκτέλεση της μεθόδου `run()`. Το νήμα τότε τερματίζει την εκτέλεσή του.

Διακοπή εκτέλεσης νημάτων

Υπάρχουν περιπτώσεις που θέλουμε να διακόψουμε την εκτέλεση μιας εφαρμογής ή ενός νήματος μιας εφαρμογής. Η Java παρέχει τον μηχανισμό της (`interruption`) που επιφέρει την διακοπή της εκτέλεσης ενός νήματος. Στο παρακάτω παράδειγμα παρουσιάζεται η χρήση της διακοπής ενός νήματος που υπολογίζει το παραγοντικό των αριθμών και το αρχικό μητρικό της εφαρμογής νήμα μετά το πέρας 250 msec:

-

```
public class Factorial implements Runnable {

        public void run() {
                long startTime = System.currentTimeMillis();

                double i = 1;
                while (true) {
                        double temp = 1;

                        for (int j = 1; j <= i; j++) {
                                temp *= j;
                        }

                        System.out.println("the factorial of " + i + " is " + temp);
                        i++;

                        if (Thread.currentThread().isInterrupted()) {
                                System.out.println("The Factorial Procedure is" +
                                                " Interrupted");

                                long estimatedTime = System.currentTimeMillis()
                                                                        - startTime;
                                System.out.println("time elapsed " + estimatedTime
                                                                + "milliseconds");

                                return;
                        }
                }
        }

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

```
                Thread fact1 = new Thread(new Factorial());
                fact1.start();

                try {
                        Thread.sleep(250);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }

                fact1.interrupt();

        }
}
```

-

```
the factorial of 1.0 is 1.0
the factorial of 2.0 is 2.0
the factorial of 3.0 is 6.0
the factorial of 4.0 is 24.0
the factorial of 5.0 is 120.0
the factorial of 6.0 is 720.0
the factorial of 7.0 is 5040.0
the factorial of 8.0 is 40320.0
```

...

...

```
the factorial of 120.0 is 6.689502913449124E198
the factorial of 121.0 is 8.09429852527344E200
the factorial of 122.0 is 9.875044200833598E202
the factorial of 123.0 is 1.2146304367025325E205
the factorial of 124.0 is 1.5061417415111404E207
the factorial of 125.0 is 1.8826771768889254E209
The Factorial Procedure is Interrupted
time elapsed 252 milliseconds
```

-

Κάθε αντικείμενο τύπου `Thread` έχει μία ιδιότητα που δέχεται δύο λογικές τιμές `True/False` και δείχνει εάν στο συγκεκριμένο νήμα έχει διακοπεί η λειτουργία του. Όταν καλούμε την μέθοδο `interrupt()` ενός νήματος θέτει αυτήν την ιδιότητα του αντικειμένου ως `True`. Ενώ η μέθοδος `isInterrupted()` μας επιστρέφει την τιμή αυτής της ιδιότητας ελέγχου διακοπής ενός αντικειμένου νήματος.

Αναστολή και συνέχεια εκτέλεσης νημάτων

Υπάρχουν περιπτώσεις που σε κάποιο νήμα πρέπει να διακοπεί η εκτέλεσή του για ορισμένο χρονικό διάστημα και μετά το πέρας αυτού του χρονικού διαστήματος να συνεχιστεί η εκτέλεσή του, για παράδειγμα το νήμα που είναι υπεύθυνο για την αύξηση της τιμής κατά ένα του δείκτη των δευτερολέπτων της εφαρμογής ρολογιού του λειτουργικού συστήματος και για το υπόλοιπο διάστημα του ενός δευτερολέπτου να βρίσκεται σε κατάσταση αναστολής η συγκεκριμένη διεργασία έως ότου της ζητηθεί από την μητρική της διεργασία να συνεχίσει την εκτέλεσή της. Κατά την διάρκεια αναστολής του νήματος, το νήμα αυτό δεν δεσμεύει κάποιους πόρους του συστήματος.

Η μέθοδος `sleep()` της κλάσης `Thread` δέχεται ως όρισμα ένα ακέραιο αριθμό ο οποίος υποδηλώνει τον χρόνο σε msec που θα παραμείνει το νήμα σε κατάσταση αναστολής. Μετά το πέρας αυτού του χρονικού

διαστήματος, δεσμεύουν το νήμα πόρους για να συνεχίσει την εκτέλεσή του. Παρακάτω παρουσιάζεται ένα παράδειγμα χρήσης της μεθόδου `sleep()`. Στο παρακάτω παράδειγμα ένα νήμα προσομοιώνει την εκκίνηση λειτουργίας ενός Server. Επειδή ο Server δεν εκκινείται αμέσως, και υπάρχει συνεπώς κάποια χρονική καθυστέρηση, προσομοιώνουμε την αναμονή με την μέθοδο `sleep()`.

-

```
import java.util.Random;

public class Server implements Runnable {

        public void run() {

                long startTime = System.currentTimeMillis();

                System.out.println("Server is booting");

                try {   //wait for booting
                        System.out.println("...waiting for Server to boot...");
                        Thread.currentThread().sleep(new Random().nextInt(10)*1000);
                }
                catch (InterruptedException e) {
                        e.printStackTrace();
                }

                System.out.println("Server is initializing");

                try {   //wait for initialization
                        System.out.println("...waiting for Server to initialize " +
                                        "its parameters...");
                        Thread.currentThread().sleep(new Random().nextInt(10)*1000);
                }
                catch (InterruptedException e) {
                        e.printStackTrace();
                }

                System.out.println("Server is ready");

                long estimatedTime = System.currentTimeMillis() - startTime;
                System.out.println("time elapsed " + estimatedTime/1000.0
                                + " seconds");
        }

        public static void main(String[] args) {

                Thread ServerThread = new Thread(new Server());
                ServerThread.start();
        }
}
```

-

Όταν καλούμε τη μέθοδο `sleep()`, το νήμα αποδεσμεύει την κεντρική μονάδα επεξεργασίας και αναστέλλει την εκτέλεσή του για ορισμένο χρονικό διάστημα. Κατά τη διάρκεια αυτής της χρονικής περιόδου δεν απασχολεί καθόλου την επεξεργαστή. Όταν ένα νήμα που βρίσκεται σε κατάσταση TIMED WAITING, μπορεί να δεχτεί διακοπή και να εμφανίσει μια τύπου `InterruptedException`, δηλαδή να διακοπεί οριστικά η εκτέλεσή του, χωρίς να περιμένει σε αναστολή έως ότου λήξει το χρονικό διάστημα

που αντιστοιχεί στη κλήση `sleep()`.

Αναμονή τερματισμού νημάτων

Υπάρχουν περιπτώσεις, όπου πρέπει ένα νήμα να αναμένει τον τερματισμό ενός άλλου νήματος. Για παράδειγμα, ένα νήμα επεξεργάζεται δεδομένα που στη συνεχεία θα χρησιμοποιηθούν από κάποιο άλλο νήμα. Το τελευταίο νήμα αναγκαστικά θα περιμένει την ολοκλήρωση της εκτέλεσης του πρώτου νήματος. Για αυτό το λόγο υπάρχει στην κλάση `Thread` η μέθοδος `join()`. Όταν χρησιμοποιούμε αυτήν την μέθοδο καλώντας την μέσω ενός αντικειμένου `Thread`, τότε η μέθοδος αυτή αναστέλλει την εκτέλεσή του νήματος που κάλεσε αυτήν την μέθοδο έως ότου το αντικείμενο τύπου `Thread` από όπου προέρχεται αυτή η μέθοδος διεκπεραιώσει την εκτέλεσή του. Στο παρακάτω παράδειγμα υπολογίζουμε την εφαπτομένη τριγωνομετρικών αριθμών χρησιμοποιώντας την ακολουθία Taylor για τον υπολογισμό του ημιτόνου και του συνημιτόνου. Η μητρική διεργασία της εφαρμογής κάνοντας χρήση της μεθόδου `join()` αναμένει τον τερματισμό της εκτέλεσης του νήματος που είναι υπεύθυνο για τον υπολογισμό της εφαπτομένης 45 μοιρών.

-

```
public class TaylorTrigSeries implements Runnable {

        private double degrees;
        private double TanResult;

        public TaylorTrigSeries(double degrees) {
                this.degrees = degrees;
        }

        public void run() {

                double TanResult = TaylorSine(this.degrees) /
                TaylorCosine(this.degrees);

                System.out.println("The tangent of " + degrees + " degrees is "
                                + TanResult);

        }

        public double TaylorSine(double degrees) {
                double sum = 0;
                for (int counter = 0; counter < 100; counter++) {
                        sum += ((Math.pow(-1.0, counter) *
                                        Math.pow(degrees2rad(degrees),
                                        (2 * counter + 1))) /
                                        (factorial(2 * counter + 1)));
                }
                return sum;
        }

        public double TaylorCosine(double degrees) {
                double sum = 0;
                for (int counter = 0; counter < 100; counter++) {
                        sum += (Math.pow(-1.0, counter) *
                                        Math.pow(degrees2rad(degrees),
                                        2.0 * counter) /
                                        (factorial(2.0 * counter)));
```

```
                }
                return sum;
        }

        public double factorial(double n) {
                if (n <= 1)
                        return 1;
                else
                        return n * factorial(n - 1);
        }

        public double degrees2rad(double degrees) {
                return (degrees * Math.PI / 180.0);
        }

        public static void main(String[] args) {

                long startTime = System.currentTimeMillis();

                Thread TrigThread = new Thread(new TaylorTrigSeries(45));
                TrigThread.start();

                // wait for the finalization of this heavy operations
                try {
                        TrigThread.join();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }

                long estimatedTime = System.currentTimeMillis() - startTime;
                System.out.println("¥nheavy mathematical" +
                                " operations are over¥n"
                                + "time elapsed " + estimatedTime / 1000.0
                                + " seconds");
        }
}
```

-

```
The tangent of 45.0 degrees is 1.0

heavy mathematical operations are over
time elapsed 0.017 seconds
```

-

Κατά την εκτέλεση αυτής της εφαρμογής, παρατηρούμε ότι το μητρικό νήμα της μεθόδου main αφού εκκινήσει το νήμα που υπολογίζει τους τριγωνομετρικούς αριθμούς, αναμένει στη μέθοδο `join()` το συγκεκριμένο νήμα να ολοκληρώσει την εκτέλεσή του. Μόλις ολοκληρωθεί η εκτέλεσή του τότε το μητρικό νήμα εκτυπώνει ένα μήνυμα ενημέρωσης στο χρήστη ότι οι υπολογισμοί έχουν ολοκληρωθεί. Επίσης πρέπει να σημειωθεί ότι πρέπει να περικλείουμε την μέθοδο `join()` σε τμήμα κώδικα `try-catch` διότι αυτή η μέθοδος μπορεί να παρουσιάσει μία εξαίρεση τύπου διακοπής (`InterruptedException`). Ο λόγος είναι ότι όσο αναμένει το πρώτο νήμα το δεύτερο να ολοκληρώσει την εκτέλεσή του υπάρχει περίπτωση το δεύτερο νήμα να ανασταλεί..

Υπηρεσίες (ή δαίμονες)

Στην Java υπάρχει μια ειδική κατηγορία νημάτων που ονομάζονται υπηρεσίες ή δαίμονες (daemon threads). Γενικά μια υπηρεσία είναι μια εφαρμογή που εκτελείται στο παρασκήνιο χωρίς να έχουμε άμεσο έλεγχο και να υπάρχει άμεση αλληλεπίδραση της συγκεκριμένης εργασίας της υπηρεσίας με τον χρήστη . Οι συγκεκριμένες υπηρεσίες έχουν χαμηλή προτεραιότητα και εκτελούνται όταν δεν υπάρχουν άλλα νήματα προς εκτέλεση. Ο τρόπος λειτουργίας τους είναι κυρίως ένας ατέρμων βρόχος ο οποίος σε κάθε επανάληψή του εκτελεί τις διάφορες εργασίες που είναι προγραμματισμένος διεκπεραιώσει, όπως για παράδειγμα υπηρεσίες όπως FTP, SSH, HTTP, έλεγχος ουράς προτεραιότητας εκτύπωσης ,Java Garbage Collector κτλ και έπειτα μεταβαίνει σε κατάσταση αναμονής έως ότου του δοθεί ξανά άδεια χρήσης του επεξεργαστή για να εκτελέσει η υπηρεσία τις εργασίες της.

Στο παρακάτω παράδειγμα παράγουμε την ακολουθία ακέραιων αριθμών Fibonacci, η ακολουθία παράγεται από ένα νήμα – υπηρεσία. Σε κάθε νέο αριθμό που υπολογίζει μας τον εκτυπώνει στην οθόνη τον νέο αριθμό.

-

```
public class DaemonFibonacci extends Thread {

        public DaemonFibonacci() {
                setDaemon(true);
        }

        public void run() {
                int prev = 0;
                int curr = 1;
                int temp;
                int counter = 0;

                System.out.println("Fibonacci Sequence Service has started");

                while (true) {
                        //print the 1st Fibonacci number
                        if (counter == 0) {
                                System.out.println("The Daemon Service "
                                                + "got you a new fibonacci number " + prev);
                                counter++;

                                //print the 2nd Fibonacci number
                        } else if (counter == 1) {
                                System.out.println("The Daemon Service "
                                                + "got you a new fibonacci number " + curr);
                                counter++;

                                //print the rest Fibonacci numbers
                        } else {

                                System.out.println("The Daemon Service "
                                                + "got you a new fibonacci number "
                                                + (prev + curr));

                                temp = curr;
                                curr = prev + curr;
                                prev = temp;
                        }
```

```
                        try {
                                Thread.sleep(3000);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }

        public static void main(String[] args) {

                Thread FibonService = new Thread(new DaemonFibonacci());
                FibonService.start();

                //sleep for some period of time
                try {
                        Thread.sleep(30000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }

                //interrupt the main thread
                System.out.println("Main Thread ending");
                Thread.currentThread().interrupt();

        }
}
```

-

```
Fibonacci Sequence Service has started
The Daemon Service got you a new fibonacci number 0
The Daemon Service got you a new fibonacci number 1
The Daemon Service got you a new fibonacci number 1
The Daemon Service got you a new fibonacci number 2
The Daemon Service got you a new fibonacci number 3
The Daemon Service got you a new fibonacci number 5
The Daemon Service got you a new fibonacci number 8
The Daemon Service got you a new fibonacci number 13
The Daemon Service got you a new fibonacci number 21
The Daemon Service got you a new fibonacci number 34
The Daemon Service got you a new fibonacci number 55
```

-

Παρατηρούμε ότι το παραπάνω νήμα διαφέρει από τα άλλα νήματα λόγο ότι είναι ένα νήμα – υπηρεσία. Η JVM μπορεί να τερματιστεί ενώ ακόμη η υπηρεσία εκτελείται, επειδή η υπηρεσία εκτελείται στο παρασκήνιο. Για να ορίσουμε ένα νήμα ως υπηρεσία πρέπει πριν αρχίσει η εκτέλεσή του με την μέθοδο `start()` πρέπει στον κατασκευαστή της κλάσης του νήματος να καλέσουμε την μέθοδο `setDaemon()`. Όταν όμως ξεκινήσει την εκτέλεσή του το νήμα τότε δεν μπορούμε να αλλάξουμε το συγκεκριμένο νήμα από υπηρεσία σε μη-υπηρεσία, δηλαδή κανονικό νήμα.

Βασικοί μηχανισμοί συγχρονισμού νημάτων

Εισαγωγή: το πρόβλημα της κοινής χρήσης δεδομένων

Στο συντρέχοντα προγραμματισμό ένα θέμα που προκύπτει συχνά είναι ο διαμοιρασμός κοινών δεδομένων και πόρων κατά την εκτέλεση πολλαπλών νημάτων, δηλαδή πολλαπλά νήματα έχουν πρόσβαση στα συγκεκριμένα δεδομένα και είτε απλώς διαβάζουν την τιμή τους είτε τα επεξεργάζονται και αποθηκεύουν νέες τιμές στα κοινά δεδομένα. Το πρόβλημα έγκειται στο γεγονός ότι οι κοινή χρήση δεδομένων μπορεί να δημιουργήσει συνθήκες δεδομένων. Τα σφάλματα που δημιουργούνται από τη κοινή χρήση δεδομένων από πολλαπλά νήματα ονομάζονται (race conditions).

Ένα κοινό παράδειγμα συνθήκης ανταγωνισμού μεταξύ νημάτων εμφανίζεται κατά την ανάγνωση / εγγραφή μεταβλητών. Για παράδειγμα στο παρακάτω πρόγραμμα δύο νήματα έχουν πρόσβαση σε μία κοινή κλάση που έχει μία μόνο μια ιδιότητα, μια μεταβλητή με αρχική τιμή 10 και μία μέθοδο που αυξάνει την τιμή της κοινής μεταβλητής κατά μία μονάδα. Αναμένουμε μετά την εκτέλεση των δύο νημάτων η κοινή μεταβλητή να έχει τιμή 12:

-

```
public class ReadWriteRace implements Runnable {

	private int aValue;

	public ReadWriteRace(int n) {
		this.aValue = n;
	}

	public void incrementValue() {
		this.aValue++;
	}

	public int getValue() {
		return this.aValue;
	}

	public void run() {

		System.out.println("I've got this value "
				+ getValue() + " and I am thread "
				+ Thread.currentThread().getName());

		incrementValue();

		System.out.println("I've incremented, new value "
				+ getValue() + " and I am thread "
				+ Thread.currentThread().getName());

	}

	public static void main(String[] args) {

		ReadWriteRace aReadWriteRace = new ReadWriteRace(10);

		Thread t1 = new Thread(aReadWriteRace);
```

```
                Thread t2 = new Thread(aReadWriteRace);

                t1.setName("Thread1");
                t2.setName("Thread2");

                t1.start();
                t2.start();
        }
}
```

-

```
I've got this value 10 and I am thread Thread1
I've got this value 10 and I am thread Thread2
I've incremented, new value 11 and I am thread Thread1
I've incremented, new value 12 and I am thread Thread2
```

- Σχόλια:

Στο συγκεκριμένο παράδειγμα εμφανίζεται εντόνως το πρόβλημα των συνθηκών ανταγωνισμού μεταξύ των δύο νημάτων που επιφέρει σφάλματα ασυνέπειας. Ενώ αναμέναμε λόγω της εκτέλεσης των δύο νημάτων να αυξηθεί η τιμή της μοιραζόμενης μεταβλητής κατά δύο μονάδες, παρατηρούμε ότι τελικά τα νήματα έχουν μία ασυνεπή σχέση μεταξύ τιμής της μεταβλητής και το αναμενόμενο αποτέλεσμα. Εδώ παρατηρούμε ότι αν και τα δύο νήματα το t1 & t2 στον κώδικα ξεκινούν με την αντίστοιχη σειρά εκτέλεσης, στα αποτελέσματα βλέπουμε ότι το πρώτο νήμα αποκτά πρόσβαση ανάγνωσης της κοινής μεταβλητής. Με βάση τα παραπάνω αποτελέσματα συμπεραίνουμε ότι η σειρά εκτέλεσης των εντολών της εφαρμογής από τα συντρέχοντα νήματα είναι ο εξής:

1. Το 1ο νήμα διαβάζει την τιμή 10 από την μεταβλητή
2. Το 2ο νήμα διαβάζει την τιμή 10 από την μεταβλητή
3. Το 2ο νήμα αυξάνει κατά ένα την κοινή μεταβλητή και εκτυπώνει την τιμή του 11
4. Το 1ο νήμα αυξάνει κατά ένα την κοινή μεταβλητή και εκτυπώνει την τιμή του 12

Πρέπει να σημειωθεί ότι η σειρά εκτέλεσης του συγκεκριμένου τμήματος κώδικα να είχε διαφορετική σειρά εκτέλεσης, επειδή η JVM διαχειρίζεται τα νήματα για το πιο θα ξεκινήσει να δεσμεύει πόρους και πιο θα τους αποδεσμεύσει για να δώσει την θέση του σε άλλο νήμα προς εκτέλεση.

Ο σωστός τρόπος θα έπρεπε να ήταν σε αυτό το ( ) προσπέλασης και επεξεργασίας της κοινής μεταβλητής ένα από τα δύο νήματα, να αναμένει πριν προσπελάσει και επεξεργαστεί την μοιραζόμενη μεταβλητή ενώ αντιθέτως το δεύτερο νήμα αφού το πρώτο θα αναμένει να αποκτήσει κάθε δικαίωμα προσπέλασης και επεξεργασίας της κοινής μοιραζόμενης μεταβλητής. Δηλαδή με την εξής σειρά:

1. Το πρώτο νήμα διαβάζει την τιμή 10 από την μεταβλητή
2. Το δεύτερο νήμα αναμένει για να διαβάσει την κοινή μεταβλητή
3. Το πρώτο νήμα αυξάνει κατά ένα την κοινή μεταβλητή και γράφει στην μνήμη την τιμή 11
4. Το δεύτερο νήμα αποκτά δικαίωμα πρόσβασης και διαβάζει την τιμή 11 από την μεταβλητή
5. Το δεύτερο νήμα αυξάνει κατά ένα την κοινή μεταβλητή και γράφει στην μνήμη την τιμή 12

Το πρόβλημα που εμφανίζεται στην προκειμένη περίπτωση ξεκινά από την πρόσβαση και προσπέλαση της κοινής μεταβλητής από πολλά νήματα. Το τμήμα του κώδικα της εφαρμογής που δημιουργεί αυτήν την ασυνέπεια ονομάζεται ( ). Για το συγκεκριμένο πρόβλημα έχουν αναπτυχθεί πολλοί μηχανισμοί επίλυσης. Οι μηχανισμοί αυτοί ονομάζονται γενικά μηχανισμοί .

Στην Java, όταν ένα νήμα προσπαθεί να αποκτήσει πρόσβαση σε ένα κρίσιμο τμήμα, πρέπει να χρησιμοποιήσει συγκεκριμένους μηχανισμούς της γλώσσας προγραμματισμού, ώστε πρώτον να ελέγξει εάν κάποιο άλλο νήμα έχει ήδη πρόσβαση στην συγκεκριμένη κρίσιμη περιοχή, εάν όχι τότε εισέρχεται τότε το νήμα στη κρίσιμη περιοχή και αποκτά πρόσβαση στους εν λόγω πόρους και δεδομένα της. Εάν όμως το συγκεκριμένο νήμα δεν μπορέσει να αποκτήσει πρόσβαση στο κρίσιμο τμήμα της εφαρμογής λόγο δέσμευσής του από κάποιο άλλο νήμα, τότε είναι αναγκασμένο να αναμένει με βάση τους μηχανισμούς συγχρονισμού των νημάτων, έως ότου το νήμα που έχει δεσμευμένο το κρίσιμο τμήμα το αποδεσμεύσει.

Οι βασικοί μηχανισμοί που παρέχονται για τον συγχρονισμό των νημάτων είναι δύο και είναι οι εξής:

- Η λέξη κλειδί – μηχανισμός `synchronized`
- Η διασύνδεση `Lock`

Συγχρονισμένες μέθοδοι

Χρησιμοποιώντας τη λέξη κλειδί `synchronized` της Java δημιουργούμε μία ελεγχόμενη συντρέχουσα πρόσβαση πολλών νημάτων σε μία μέθοδο η οποία αναφέρεται σε ένα αντικείμενο. Η λογική είναι η εξής: όταν ένα νήμα προσπαθεί να αποκτήσει πρόσβαση σε μία μέθοδο ενός αντικειμένου η οποία είναι ορισμένη με την λέξη κλειδί `synchronized` ενώ ένα άλλο νήμα έχει ήδη αποκτήσει πρόσβαση σε συγκεκριμένο κρίσιμο τμήμα του κώδικα για το ίδιο αντικείμενο, τότε θα ανασταλεί η εκτέλεση του πρώτου νήματος έως ότου το δεύτερο νήμα αποδεσμεύσει τη συγκεκριμένη κρίσιμη περιοχή. Ας δούμε πως θα αλλάξει το προηγούμενο παράδειγμα με τον πειραματισμό μίας μοναδικής μοιραζόμενης μεταβλητής από δύο νήματα:

-

```
public class ReadWriteSyncMethodRace
implements Runnable {

	private int aValue;

	public ReadWriteSyncMethodRace(int n) {
		this.aValue = n;
	}

	public synchronized void GetAndIncrementValue() {

		System.out.println("I've got this value "
				+ this.aValue
				+ " and I am thread "
				+ Thread.currentThread().getName());

		this.aValue++;

		System.out.println("I've incremented, new value "
				+ this.aValue
				+ " and I am thread "
				+ Thread.currentThread().getName()+"¥n");
	}

	public void run() {

		GetAndIncrementValue();
```

```
        }

        public static void main(String[] args) {

                ReadWriteSyncMethodRace aReadWriteRace =
                        new ReadWriteSyncMethodRace(10);

                Thread t1 = new Thread(aReadWriteRace);
                Thread t2 = new Thread(aReadWriteRace);

                t1.setName("Thread1");
                t2.setName("Thread2");

                t1.start();
                t2.start();
        }
}
```

-

```
I've got this value 10 and I am thread Thread1
I've incremented, new value 11 and I am thread Thread1

I've got this value 11 and I am thread Thread2
I've incremented, new value 12 and I am thread Thread2
```

- Σχόλια:

Τώρα παρατηρούμε ότι υπάρχει πλήρη συνέπεια δεδομένων κατά την διάρκεια εκτέλεσης των δύο νημάτων. Και έχουμε το επιθυμητό αποτέλεσμα όπως περιγράφτηκε στο προηγούμενο παράδειγμα.

Όταν υπάρχουν παραπάνω από μια συγχρονισμένες μέθοδοι σε ένα αντικείμενο τότε μόνο μια τέτοια μέθοδος μπορεί να εκτελείται κάθε φορά, ενώ οι υπόλοιπες συγχρονισμένες μέθοδοι αναστέλλονται,

Οι κλήσεις συγχρονισμένων μεθόδων ενός αντικειμένου υπακούν στη λογική της ακολουθιακής ακεραιότητας (happens before), δηλαδή οι αλλαγές στη κατάσταση του αντικειμένου που επιβάλλει η εκτέλεση ενός συγχρονισμένου νήματος είναι άμεσα ορατές στα επόμενα νήματα.

Προβλήματα συγχρονισμού

Ο συγχρονισμός των νημάτων στο κρίσιμο τμήμα όπως φαίνεται επιλύει τα σφάλματα ασυνέπειας δεδομένων, μπορεί ωστόσο να δημιουργήσει ανταγωνισμό μεταξύ των νημάτων, ο οποίος εμφανίζεται όταν δύο ή περισσότερα νήματα προσπαθούν ταυτόχρονα να αποκτήσουν πρόσβαση στις κοινές περιοχές μνήμης. Αυτό αναγκάζει τη Java να αναστέλλει την εκτέλεσή τους. Το αποτέλεσμα αυτού του ανταγωνισμού για διαμοιραζόμενες περιοχές μνήμης παίζει σημαντικό ρόλο στην ζωτικότητα (liveness) των νημάτων που οδηγεί μερικές φορές στην λιμοκτονία (starvartion) τους.

Ο μη προσεκτικός συγχρονισμός νημάτων μπορεί επίσης να οδηγήσει σε κύκλους αναμονής (wait for cycles) όπου μια μέθοδος κατέχει ένα πόρο Α και ζητά άλλο πόρο Β, ενώ μια άλλη μέθοδος κατέχει το πόρο Β και ζητά το πόρο Α. Το αποτέλεσμα αυτού του κυκλικού ανταγωνισμού για διαμοιραζόμενες περιοχές μνήμης παίζει σημαντικό ρόλο στην ασφάλεια (security) των νημάτων που οδηγεί μερικές φορές σε αδιέξοδα (deadlocks).

Συγχρονισμός τμήματος μεθόδου

Πέραν της συγχρονισμένης προστασίας μιας μεθόδου με την λέξη – κλειδί `synchronized`, μπορούμε να προστατέψουμε συγκεκριμένα διακριτά τμήματα κώδικα μέσα σε μία μέθοδο όπως για παράδειγμα

συγκεκριμένες προσπελάσεις μεταβλητών, είτε έναν βρόχο επαναλήψεων κτλ. Ο εν λόγω λοιπόν μηχανισμός είναι η λέξη – κλειδί `synchronized(parameter)`. Το όρισμα (`parameter`) στον συγκεκριμένο μηχανισμό διαχείρισης συγχρονισμού είναι συνήθως είτε το ίδιο το αντικείμενο της κλάσης που εμπεριέχει μέσα της το κρίσιμο τμήμα της μεθόδου της ιδίας κλάσης, είτε μια ανεξάρτητη μεταβλητή τύπου `Object` που είναι μεν ιδιότητα της κλάσης αλλά δεν χρησιμοποιείτε πουθενά στον κώδικα της εφαρμογής παρά μόνο ως παράμετρος στον παραπάνω μηχανισμό. Ο ρόλος αυτής της παραμέτρου είναι ότι όταν ένα νήμα είναι μέσα στο κρίσιμο τμήμα κάνοντας χρήση αυτής της παραμέτρου, τότε τα υπόλοιπα νήματα πρέπει να παραμένουν σε κατάσταση αναστολής έως ότου αποδεσμεύσει το κρίσιμο τμήμα και την παράμετρο το πρώτο νήμα. Στο παρακάτω παράδειγμα διαπραγματευόμαστε πάλι το αρχικό παράδειγμα της ενότητας, δηλαδή τα δύο νήματα που διαμοιράζονται μία κοινή μεταβλητή και αυξάνουν τον δείκτη κατά μία μονάδα. Το αναμενόμενο αποτέλεσμα μετά από την ολοκλήρωση της εκτέλεσης των νημάτων είναι από την αρχική τιμή που είχε η μεταβλητή που είναι 10 να γίνει 12.

-

```
public class ReadWriteSyncMethodBlockRace implements Runnable {

	private int aValue;
	private Object SyncControlObj = new Object();

	public ReadWriteSyncMethodBlockRace(int n) {
		this.aValue = n;
	}

	public void GetAndIncrementValue() {

		synchronized (SyncControlObj) {

			System.out.println("I've got this value "
					+ this.aValue
					+ " and I am thread "
					+ Thread.currentThread().getName());

			this.aValue++;

			System.out.println("I've incremented, new value "
					+ this.aValue
					+ " and I am thread "
					+ Thread.currentThread().getName()
					+ "¥n");
		}
	}

	public void run() {

		GetAndIncrementValue();
	}

	public static void main(String[] args) {

		ReadWriteSyncMethodBlockRace aReadWriteRace =
			new ReadWriteSyncMethodBlockRace(10);

		Thread t1 = new Thread(aReadWriteRace);
		Thread t2 = new Thread(aReadWriteRace);
```

```
                t1.setName("Thread1");
                t2.setName("Thread2");

                t1.start();
                t2.start();
        }
}
```

-

```
I've got this value 10 and I am thread Thread1
I've incremented, new value 11 and I am thread Thread1

I've got this value 11 and I am thread Thread2
I've incremented, new value 12 and I am thread Thread2
```

-

Όταν χρησιμοποιούμε τον μηχανισμό – λέξη-κλειδί `synchronized` για να προστατέψουμε ένα τμήμα του κώδικα της εφαρμογής μας, πρέπει να χρησιμοποιήσουμε ένα αντικείμενο ως παράμετρο, ακόμη και το αντικείμενο της ίδιας της κλάσης του κώδικα της εφαρμογής μας δίνοντας ως παράμετρο το λέξη-κλειδί `this`. Η JVM μας εγγυάται ότι μόνο ένα νήμα θα αποκτήσει πρόσβαση σε όλα τα τμήματα του κώδικα που είναι προστατευμένα με αυτό το αντικείμενο ως παράμετρο.

Υπό συνθήκη συγχρονισμός

Υπάρχουν περιπτώσεις, όπου η αναστολή ή εκτέλεση ενός νήματος εξαρτάται από την κατάσταση ενός άλλου νήματος, δηλαδή ένα νήμα πρέπει να περιμένει ένα σήμα από κάποιο άλλο για να συνεχίσει. Σε αυτή τη περίπτωση ο έλεγχος πρόσβασης ενός νήματος σε μια κρίσιμη περιοχή εξαρτάται από ένα άλλο νήμα. Ένα τέτοιο παράδειγμα είναι το κλασσικό πρόβλημα Παραγωγού – Καταναλωτή. Στο συγκεκριμένο παράδειγμα έχουμε δύο κλάσεις, τον Παραγωγό και τον Καταναλωτή. Ο Παραγωγός εισάγει δεδομένα σε ένα μοιραζόμενο ενταμιευτή συγκεκριμένου μεγέθους, ενώ ο Καταναλωτής εξάγει δεδομένα από τον ενταμιευτή. Αντικείμενα της κλάσης Παραγωγός αυξάνουν κατά μία μονάδα την τιμή ενός μοιραζόμενου μετρητή ενώ αντικείμενα της κλάσης του Καταναλωτή μειώνουν κατά μία μονάδα την τιμή του μετρητή.

Λόγω του ότι έχουμε μία μοιραζόμενη μεταβλητή, πρέπει να ελέγξουμε την πρόσβαση της χρησιμοποιώντας τους προηγούμενους μηχανισμούς συγχρονισμού. Ωστόσο δημιουργούνται νέες απαιτήσεις ελέγχου πρόσβασης. Αυτές είναι οι εξής: ο Παραγωγός δεν μπορεί να αυξήσει άλλο την κοινή μεταβλητή όταν έχει φτάσει σε μία συγκεκριμένη μέγιστη τιμή, ενώ αντιθέτως ο Καταναλωτής δεν μπορεί να μειώσει την τιμή της κοινής μεταβλητής όταν αυτή γίνει μηδέν.

Σε τέτοιες περιπτώσεις που μας περιορίζουν ακόμη περισσότερο τον έλεγχο πρόσβασης σε κρίσιμα τμήματα των νημάτων η Java παρέχει τις μεθόδους `wait()`, `notify()` και `notifyAll()`. Ένα νήμα μπορεί να καλέσει την μέθοδο `wait()`, μέσα σε ένα ελεγχόμενα συγχρονισμένο τμήμα κώδικα. Όταν την καλεί, η JVM μεταβάλει το νήμα σε κατάσταση αναμονής (`WAITING`), για να το επαναφέρει πίσω σε κατάσταση εκτέλεσης πρέπει να καλέσουμε την μέθοδο `notify()` είτε την μέθοδο `notifyAll()`, μέσα σε ένα ελεγχόμενα συγχρονισμένο τμήμα κώδικα.

Στο παρακάτω παράδειγμα υλοποιούμε τον πρόγραμμα Παραγωγός – Καταναλωτής. Χρησιμοποιούμε τέσσερις κλάσεις. Μια κλάση – νήμα για τον παραγωγό και μια κλάση νήμα για τον καταναλωτή. Ο παραγωγός "τοποθετεί" στοιχεία στην κοινή μοιραζόμενη κλάση που ονομάζεται ShelfStorage που προσομοιώνει το ράφι των προϊόντων ενώ ο καταναλωτής "καταναλώνει" προϊόντα από τη μοιραζόμενη κλάση ράφι. Η κλάση ShelfStorage που είναι κοινή – μοιραζόμενη για τα δύο νήματα υλοποιεί τις μεθόδους "τοποθέτηση – RefreshShelf" που της χρησιμοποιεί αντικείμενα κλάσης Παραγωγός και την

μέθοδο “αγορά - buy” στοιχείων που την χρησιμοποιεί αντικείμενα της κλάσης Καταναλωτής.

-

```
import java.util.LinkedList;
import java.util.Random;

public class Shelf {

	private String productName;
	private int maxSize;
	private LinkedList<Integer> storage;

	public Shelf(int maxSize, String name) {
		this.productName = name;
		this.maxSize = maxSize;
		storage = new LinkedList<Integer>();
	}

	public synchronized void ShelfRefresh() {

		while (storage.size() == maxSize) {
			try {
				wait();
			} catch (InterruptedException e) {
				System.out.println("The Producer closes his store");
				System.exit(0);
			}
		}

		int amount = new Random().nextInt(5)+5;
		int counter=0;
		while(storage.size()<maxSize && counter<amount) {
			int item = storage.size() + 1;
			storage.offer(item);
			counter++;
		}

		System.out.println("The Producer updates the "
				+ productName + "'s "
				+ "shelf with new items"
				+ " #of items: " + storage.size());

		notifyAll();
	}

	public synchronized void buy() {

		int amount = new Random().nextInt(5);

		if (amount == 0)
			System.out.println("A Customer just browse and buys nothing");
		else {
			System.out.println("A Customer wants to buy "
					+ amount + " items");

			while (storage.size() < amount) {
```

```
                                try {
                                        wait();
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                        }
                        for (int i = 0; i < amount; i++) {
                                storage.poll();

                                System.out.println("A Consumer buys a "
                                                + productName
                                                + ", items left "
                                                + storage.size());

                                notifyAll();
                        }
                }
        }
}
```

```
public class Producer implements Runnable {

        private Shelf shelf;

        public Producer(Shelf shelf) {
                this.shelf = shelf;
        }

        public void run() {
                for(int i=0; i<200; i++) {
                        shelf.ShelfRefresh();
                }
        }
}
```

```
public class Consumer implements Runnable {

        private Shelf shelf;

        public Consumer(Shelf shelf) {
                this.shelf = shelf;
        }

        public void run() {
                for (int i = 0; i < 200; i++) {
                        shelf.buy();
                }
                System.out.println("It seems no more Consumers will " +
                                "pay the store a visit for today");
        }
}
```

```
public class MainProducerConsumer {
```

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

                Shelf shelf = new Shelf(30, "mango");

                Producer producer = new Producer(shelf);
                Thread thread1 = new Thread(producer);

                Consumer consumer = new Consumer(shelf);
                Thread thread2 = new Thread(consumer);

                thread1.start();
                thread2.start();

                try {
                        Thread.sleep(4000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }

                System.out.println("Store is closing");
                thread1.interrupt();
                thread2.interrupt();
        }
}
```

-

```
The Producer updates the mango's shelf with new items #of items: 9
The Producer updates the mango's shelf with new items #of items: 15
The Producer updates the mango's shelf with new items #of items: 23
The Producer updates the mango's shelf with new items #of items: 30
A Customer wants to buy 2 items
A Consumer buys a mango, items left 29
A Consumer buys a mango, items left 28
A Customer wants to buy 3 items
A Consumer buys a mango, items left 27
A Consumer buys a mango, items left 26
A Consumer buys a mango, items left 25
A Customer wants to buy 1 items
A Consumer buys a mango, items left 24
A Customer wants to buy 3 items
A Consumer buys a mango, items left 23
A Consumer buys a mango, items left 22
A Consumer buys a mango, items left 21
A Customer wants to buy 2 items
A Consumer buys a mango, items left 20
A Consumer buys a mango, items left 19
A Customer just browse and buys nothing
A Customer just browse and buys nothing
A Customer wants to buy 3 items
A Consumer buys a mango, items left 18
A Consumer buys a mango, items left 17
A Consumer buys a mango, items left 16
A Customer just browse and buys nothing
A Customer wants to buy 2 items
A Consumer buys a mango, items left 15
A Consumer buys a mango, items left 14
It seems no more Consumers will pay the store a visit for today
```

```
The Producer updates the mango's shelf with new items #of items: 23
The Producer updates the mango's shelf with new items #of items: 30
Store is closing
The Producer closes his store
```

- :

Οι μέθοδοι buy() & RefreshShelf(), της κλάσης ShelfStorage, υλοποιούν το έργο του υπό συνθήκη συγχρονισμού. Η μέθοδος RefreshShelf() ελέγχει εάν το ράφι έχει διαθέσιμο χώρο, ελέγχοντας το μέγεθος της λίστας που προσομοιώνει τον αριθμό των αντικειμένων στην λίστα και αν υπάρχει διαθέσιμος χώρος, τοποθετεί ένα νέο στοιχείο. Εάν είναι όμως γεμάτο τότε καλεί την μέθοδο wait() και αναμένει σε κατάσταση αναμονής έως ότου βρεθεί ξανά διαθέσιμος χώρος στο ράφι των προϊόντων (στοιχείων)

Όταν το νήμα του Καταναλωτή καλεί την notifyAll(), το νήμα του Παραγωγού ενεργοποιείται σε κατάσταση εκτέλεσης και ξανά-ελέγχει την συνθήκη συγχρονισμού εάν έχει δημιουργηθεί νέος διαθέσιμος χώρος στο ράφι. Παρόμοια συμπεριφορά εμφανίζει η μέθοδος buy() που την καλεί το νήμα του Καταναλωτή. Αρχικά ελέγχει εάν υπάρχουν διαθέσιμα προϊόντα (στοιχεία) προς πώληση. Εάν το ράφι είναι κενό τότε καλεί την μέθοδο wait() έως ότου το νήμα του Παραγωγού καλέσει την μέθοδο notifyAll(), τότε το νήμα από κατάσταση αναμονής μετατρέπεται σε κατάσταση εκτέλεσης και ξανά-ελέγχει την συνθήκη συγχρονισμού.

Συγχρονισμός με κλειδώματα (κλειδαριές)

Ο δεύτερος μηχανισμός για τον συγχρονισμό τμημάτων κώδικα που αποτελούν κρίσιμα τμήματα βασίζεται στην διασύνδεση Lock και στις κλάσεις που την (όπως η κλάση ReentrantLock). Ο συγκεκριμένος μηχανισμός έχει περισσότερες δυνατότητες από τον μηχανισμό – λέξη-κλειδί synchronized. Τα πλεονεκτήματα του συγκεκριμένου μηχανισμού είναι τα εξής:

1. Ενώ η λέξη-κλειδί synchronized ελέγχει τη προσπέλαση ενός κρίσιμου τμήματος συνολικά, η διασύνδεση Lock παρέχει πιο σύνθετες και ευέλικτες δομές για τον συγχρονισμό επί του κρίσιμου τμήματος.
2. Μία από αυτές τις δομές της αναφερθείσας διασύνδεσης είναι η μέθοδος tryLock(). Η μέθοδος προσπαθεί να αποκτήσει πρόσβαση σε μία κλειδαριά και αν δεν μπορεί επειδή η κλειδαριά έχει καταληφθεί από άλλο νήμα, επιστρέφει πίσω την κλειδαριά. Ενώ με τη λέξη-κλειδί synchronized, όταν ένα νήμα προσπαθήσει να αποκτήσει πρόσβαση σε ένα κρίσιμο τμήμα που κατέχει ελεγχόμενη συγχρονισμένη πρόσβαση, ενώ ένα άλλο νήμα ήδη βρίσκεται και έχει πρόσβαση στο κρίσιμο τμήμα, τότε το αρχικό τμήμα αναστέλλεται η εκτέλεσή του, έως ότου το δεύτερο νήμα ολοκληρώσει την εκτέλεσή του. Με τις κλειδαριές, εκτελούμε την μέθοδο tryLock(). Μας επιστρέφει μία λογική τιμή που μας υποδεικνύει εάν το κρίσιμο τμήμα που προστατεύεται με την κλειδαριά εκτελείται από το ίδιο το νήμα που κάλεσε την μέθοδο.
3. Επίσης η διασύνδεση Lock μας επιτρέπει πολλαπλές προσπελάσεις των μεταβλητών της εφαρμογής από τα νήματα αλλά προστατεύει την επεξεργασία και αποθήκευση νέων τιμών επί των μεταβλητών. Δηλαδή μόνο ένα νήμα έχει δικαίωμα επεξεργασίας μιας μεταβλητής ενώ τα υπόλοιπα νήματα πρέπει να αναμένουν.
4. Τέλος ο συγκεκριμένος μηχανισμός παρέχει καλύτερη απόδοση σε σχέση με την λέξη-κλειδί synchronized, ώστε να μειώνονται να φαινόμενα λιμοκτονίας των νημάτων.

Στο παρακάτω παράδειγμα θα προσπαθήσουμε να συγκρίνουμε τους δύο μηχανισμούς συγχρονισμού. Ορισμένες εταιρίες, εκτελούν τραπεζικές συναλλαγές μέσω κοινών τραπεζικών λογαριασμών. Η πρόσβαση στα κρίσιμα τμήματα των συναλλαγών (δηλαδή η πρόσβαση και τροποποίηση των κοινών τραπεζικών λογαριασμών) πρέπει να ελεγχθεί με μηχανισμούς συγχρονισμού:

-

```
public class Account {

        private double balance;

        public double getBalance() {
                return balance;
        }

        public void setBalance(double balance) {
                this.balance = balance;
        }

        public synchronized void addAmount(double amount) {
                // acquired lock on from Account Object
                double tmp = balance;
                tmp += amount;
                balance = tmp;
        }

        public synchronized void subtractAmount(double amount) {
                // acquired lock on from Account Object
                double tmp = balance;
                tmp -= amount;
                balance = tmp;
        }
}
```

```
public class Company implements Runnable {

        //company's account attribute
        private Account account;

        public Company(Account account) {
                this.account = account;
        }

        //add company's profits
        public void run() {
                for (int i = 0; i < 100; i++) {
                        account.addAmount(1000);
                }
        }
}
```

```
public class Bank implements Runnable {

        //the bank registers two company's accounts
        private Account account1;
        private Account account2;
```

```
        public Bank(Account account1, Account account2) {
                this.account1 = account1;
                this.account2 = account2;
        }

        //bank transactions like wages, security etc
        public void run() {
                for (int i = 0; i < 100; i++) {
                        account1.subtractAmount(1000);
                        account2.subtractAmount(1000);
                }
        }
}
```

```
public class CompanyTransaction implements Runnable {

        private Account Account1, Account2;

        public CompanyTransaction(Account acc1, Account acc2) {
                this.Account1 = acc1;
                this.Account2 = acc2;
        }

        //start a transaction at a time between the two companies
        int turn = 0;
        public void run() {
                for (int i = 0; i < 20; i++) {
                        if (turn % 2 == 0) {
                                this.Account1.subtractAmount(100 + i * i);
                                this.Account2.addAmount(100 + i * i);
                        } else {
                                this.Account2.subtractAmount(100 + i * i);
                                this.Account1.addAmount(100 + i * i);
                        }
                        turn++;
                }
        }
}
```

```
public class MainAccountTransactions {

        public static void main(String[] args) {

                //the company1's account object
                Account account1 = new Account();
                account1.setBalance(10000);

                Company company1 = new Company(account1);
                Thread companyThread1 = new Thread(company1);


                //the company2's account object
                Account account2 = new Account();
                account2.setBalance(10000);
```

```
                Company company2 = new Company(account2);
                Thread companyThread2 = new Thread(company2);

                //the thread of the inter-transactions between
                //the two companies
                CompanyTransaction transactions =
                        new CompanyTransaction(account1,account2);
                Thread transactionsThread = new Thread(transactions);

                //the bank object thread
                Bank bank = new Bank(account1, account2);
                Thread bankThread = new Thread(bank);


                System.out.printf("Account1 : Initial Balance: %f¥n", account1
                                .getBalance());

                System.out.printf("Account2 : Initial Balance: %f¥n", account2
                                .getBalance());

                //Start the threads.
                companyThread1.start();
                companyThread2.start();
                transactionsThread.start();
                bankThread.start();

                try {
                        //waiting for all the threads to finish
                        companyThread1.join();
                        companyThread2.join();
                        bankThread.join();
                        transactionsThread.join();

                        Thread.sleep(2000);

                        System.out.printf("¥n -end of fiscal year- ¥n");
                        System.out.printf("Account1 : Final Balance: %f¥n", account1
                                        .getBalance());

                        System.out.printf("Account2 : Final Balance: %f¥n", account2
                                        .getBalance());

                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

- Αποτελέσματα:

```
Account1 : Initial Balance: 10000,000000
Account2 : Initial Balance: 10000,000000

 -end of fiscal year-
Account1 : Final Balance: 10190,000000
Account2 : Final Balance: 9810,000000
```

-

Στην Java ο μηχανισμός – λέξη-κλειδί synchronized υλοποιείται με κλειδαριές. Κάθε αντικείμενο στην Java συνδέεται με μία (intrisic lock) που συσχετίζεται με το αντικείμενο αυτό. Όταν ένα νήμα προσπαθεί να αποκτήσει πρόσβαση σε ένα τμήμα του κώδικα ή σε μία μέθοδο που ελέγχεται από τον μηχανισμό – λέξη-κλειδί synchronized, τότε αποκτά αρχικά την εγγενή κλειδαριά της του αντικειμένου. Σε περίπτωση στατικών μεθόδων ή στατικών μεταβλητών το νήμα θα έχει πρόσβαση στην κλειδαριά της κλάσης του αντικειμένου.

Ωστόσο στο προηγούμενο παράδειγμα αναδύεται ένα κόμη πρόβλημα εάν στην κλάση CompanyTransaction δύο διαφορετικά νήματα A και B προσπαθούν σχεδόν ταυτόχρονα να μεταφέρουν χρήματα. Δηλαδή έστω το νήμα A προσπαθεί να αποκτήσει πρόσβαση στον λογαριασμό account1 και αναμένει να αποκτήσει πρόσβαση στον λογαριασμό account2 ενώ το νήμα B έχει αποκτήσει πρόσβαση στον λογαριασμό account2 και αναμένει να αποκτήσει πρόσβαση στον λογαριασμό account1, τότε αυτού του τύπου αναμονής των δύο νημάτων μας οδηγεί σε (*deadlock*) και η εφαρμογή μας πρέπει να επανεκκινήσει.

Μια πιο “καθαρή” προσέγγιση υλοποιείται με την κλάση ReentrantLock και την μέθοδό της tryLock(). Η συγκεκριμένη κλάση έχει την ευελιξία ότι εάν ένα νήμα δεν μπορεί να αποκτήσει όλες τις απαραίτητες κλειδαριές για να έχει δικαίωμα εκτέλεσης του κρίσιμου τμήματος. Τότε τις αποδεσμεύει και προσπαθεί ξανά. Ας δούμε πως θα μετατραπεί το προηγούμενο παράδειγμα και συγκεκριμένα το κομμάτι του κώδικα που αλλάζει είναι η κλάση Account που περιλαμβάνει όλα τα κρίσιμα τμήματα με την χρήση των κλειδαριών. Οι υπόλοιπες κλάσεις παραμένουν όπως είναι χωρίς αλλαγές:

-

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Account {

	private double balance;
	private final Lock addAmountLock = new ReentrantLock();
	private final Lock subtractAmountLock = new ReentrantLock();

	public double getBalance() {
		return balance;
	}

	public void setBalance(double balance) {
		this.balance = balance;
	}

	public void addAmount(double amount) {
		// acquired lock from Account Object
		addAmountLock.lock();
		try {
			double tmp = balance;
			tmp += amount;
			balance = tmp;
		}
		finally {
			addAmountLock.unlock();
```

```
                }
        }

        public void subtractAmount(double amount) {
                // acquired lock from Account Object
                subtractAmountLock.lock();
                try {
                        double tmp = balance;
                        tmp -= amount;
                        balance = tmp;
                }
                finally {
                        subtractAmountLock.unlock();
                }
        }
}
```

-

```
Account1 : Initial Balance: 10000,000000
Account2 : Initial Balance: 10000,000000

 -end of fiscal year-
Account1 : Final Balance: 10190,000000
Account2 : Final Balance: 9810,000000
```

-

Όταν ένα νήμα Α καλεί μία από τις μεθόδους – κρίσιμα τμήματα της κλάσης `Account`, εάν κανένα άλλο νήμα από πριν δεν έχει αποκτήσει πρόσβαση στην κλειδαριά, τότε η μέθοδος δίνει στο νήμα Α τον έλεγχο της κλειδαριάς και το δικαίωμα της εκτέλεσης του κρίσιμου τμήματος. Διαφορετικά εάν το δεύτερο νήμα Β εκτελεί ήδη το κρίσιμο τμήμα τότε η μέθοδος `lock()`, αναστέλλει το νήμα Α έως ότου το νήμα Β αφήσει το κρίσιμο τμήμα. Στο τέλος του κρίσιμου τμήματος πρέπει να χρησιμοποιούμε την μέθοδο `unlock()` για να ελευθερώσουμε την κλειδαριά. Εάν δεν καλέσουμε αυτή την μέθοδο, τότε όλα τα υπόλοιπα νήματα θα περιμένουν για πάντα, δημιουργώντας έτσι καταστάσεις αδιεξόδου διότι όλα τα υπόλοιπα νήματα προσπαθούν να δεσμεύσουν την κλειδαριά του κρίσιμου τμήματος. Με το τμήμα κώδικα `try - catch` στο κρίσιμο τμήμα, στο κομμάτι `finally` συμπεριλαμβάνουμε την μέθοδο `unlock()` ώστε να αποδεσμεύσουμε την κλειδαριά.

Σύνθετοι μηχανισμοί συγχρονισμού νημάτων

Εισαγωγή

Στην προηγούμενη ενότητα εξετάσαμε τον βασικό συγχρονισμό νημάτων για την διαχείριση του κρίσιμου τμήματος των μοιραζόμενων κοινών μεταβλητών μεταξύ των προς εκτέλεση νημάτων για την αποφυγή σφαλμάτων ασυνέπειας των κοινών δεδομένων. Οι μηχανισμοί που εξετάστηκαν είναι ο μηχανισμός λέξη-κλειδί `Synchronized` και ο μηχανισμός κλειδωμάτων της διασύνδεσης `Lock`.

Σε αυτήν την ενότητα θα εξετάσουμε σύνθετες δομές για τον συγχρονισμό πολλαπλών νημάτων.. Οι συγκεκριμένοι μηχανισμοί παρέχουν νέες δυνατότητες και έχουν περισσότερα χαρακτηριστικά από εκείνους της προηγούμενης ενότητας. Οι σύνθετες δομές συγχρονισμού που θα παρουσιαστούν είναι οι εξής:

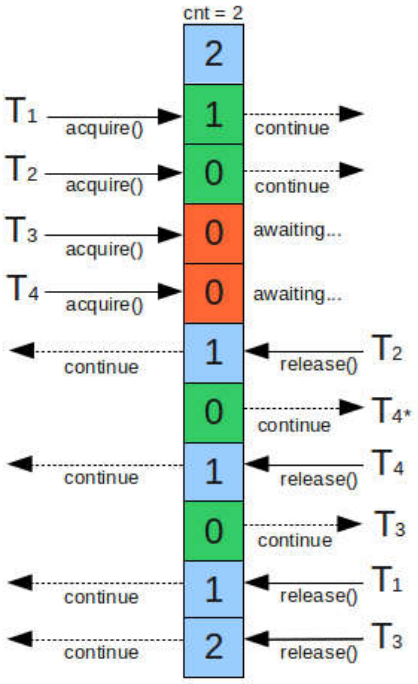
1. `Semaphores`: είναι ένας μετρητής ακεραίων που ελέγχει την πρόσβαση σε ένα ή και περισσότερα μοιραζόμενα τμήματα στην μνήμη.

2. CountDownLatch: Η κλάση CoutnDownLatch είναι ένας μηχανισμός που επιτρέπει σε νήματα να αναμένουν για την ολοκλήρωση διαφόρων εργασιών. Μόλις αυτές οι εργασίες ολοκληρωθούν τότε τα συγκεκριμένα εν αναμονή νήματα μπορούν να συνεχίσουν το υπόλοιπο της εκτέλεσής τους.
3. CyclicBarrier: Η κλάση CyclicBarrier είναι ένας μηχανισμός συγχρονισμού, που έχει την ικανότητα να συγχρονίζει όλα τα νήματα κατά την διάρκεια της εκτέλεσής τους σε ένα συγκεκριμένο κοινό τμήμα του κώδικά τους.
4. Phaser: Η κλάση Phaser είναι ένας νέος σχετικά μηχανισμός της Java, που ελέγχει και διαχωρίζει την εκτέλεση των συντρεχουσών εργασιών των νημάτων σε στάδια - φάσεις. Δηλαδή κατά την διάρκεια εκτέλεσης των νημάτων της εφαρμογής, πρέπει πρώτα όλα τα νήματα να εκτελούν πλήρως μία φάση της εφαρμογής και να περιμένουν τα υπόλοιπα για τον τερματισμό τους, και στην συνέχεια να συνεχίζουν στην επόμενη φάση του κώδικα της εφαρμογής.
5. Exchanger: Η κλάση Exchanger είναι ένας μηχανισμός της Java που παρέχει την δυνατότητα ανταλλαγής δεδομένων μεταξύ δύο νημάτων.

Στις επόμενες ενότητες θα εξετάσουμε κάθε παραπάνω μηχανισμό ξεχωριστά μαζί με ανάλογο παράδειγμα κώδικα.

Έλεγχος συγχρονισμένης πρόσβασης σε πόρους με semaphores

Γενικά στην επιστήμη των υπολογιστών Σηματοφόρος ή σηματοφορέας ή σημαφόρος (Semaphore) είναι μια δομή δεδομένων που έχει ως λειτουργικό σκοπό τον συγχρονισμό των συντεχόντων διεργασιών εντός του λειτουργικού συστήματος. Στην Java όταν ένα νήμα προσπαθεί να αποκτήσει πρόσβαση σε κοινές μοιραζόμενες περιοχές μνήμης, αρχικώς πρέπει να αποκτήσει ένα σημαφόρο. Ένας σημαφόρος στην ουσία είναι ένας μετρητής. Εάν ο μετρητής είναι μεγαλύτερος από το μηδέν και ένα νήμα ζητά πρόσβαση στο κρίσιμο τμήμα ή σε μοιραζόμενες μεταβλητές από τον σημαφόρο, τότε μειώνεται κατά μία μονάδα η τιμή του μετρητή του σημαφόρου και επιτρέπει πρόσβαση στις μοιραζόμενες μεταβλητές. Αντιθέτως ένα ο μετρητής του σημαφόρου είναι μηδέν, τότε ο σημαφόρος θέτει το νήμα προς εκτέλεση σε κατάσταση αναστολής έως ότου ο μετρητής γίνει ξανά μεγαλύτερος από το μηδέν. Όταν ένα νήμα αποδεσμεύει το κρίσιμο τμήμα και τις μοιραζόμενες μεταβλητές, πρέπει να αυξήσει τον μετρητή του σημαφόρου κατά μία μονάδα και να τον αποδεσμεύσει. Η έννοια του μετρητή στο σημαφόρο συμβολίζει τον αριθμό των μοιραζόμενων διαθέσιμων πόρων που μπορούν να δοθούν για επεξεργασία στα προς εκτέλεση νήματα. Παραδείγματος χάριν στην παρακάτω απεικόνιση προσομοιώνεται η πρόσβαση σε δύο μοιραζόμενους πόρους μιας εφαρμογής με χρήση των σημαφόρων που έχουν αρχικοποιηθεί με την τιμή δύο:



Στο παρακάτω παράδειγμα υλοποιούμε την προαναφερθείσα εφαρμογή Παραγωγού – Καταναλωτή με χρήση των σημαφόρων. Η κλάση που επιδέχεται τις περισσότερες αλλαγές είναι η κλάση `Shelf`, και η Main, ενώ οι υπόλοιπες δύο `Producer` & `Consumer` μένουν αναλλοίωτες.

-

```
import java.util.LinkedList;
import java.util.Random;
import java.util.concurrent.Semaphore;

public class Shelf {

	private Semaphore ShelfSemaphore;
	private String productName;
	private int maxSize;
	private LinkedList<Integer> storage;

	public Shelf(int maxSize, String name) {
		this.productName = name;
		this.maxSize = maxSize;
		storage = new LinkedList<Integer>();
		ShelfSemaphore = new Semaphore(1,true);
	}

	public void ShelfRefresh() {

		try {
			ShelfSemaphore.acquire();
```

```
                    if (storage.size() == this.maxSize)
                            System.out.println("Shelf is full of "
                                            + productName + "s");
                    else {
                            int amount = new Random().nextInt(5)+5;
                            int counter=0;
                            while(storage.size()<maxSize && counter<amount) {
                                    int item = storage.size() + 1;
                                    storage.offer(item);
                                    counter++;
                            }
                            System.out.println("The Producer updates the "
                                            + productName
                                            + "'s " + "shelf with new items"
                                            + " #of items: "
                                            + storage.size());
                    }
            } catch (InterruptedException e) {
                    e.printStackTrace();
            } finally {
                    ShelfSemaphore.release();
            }
    }

    public void buy() {

            try {
                    ShelfSemaphore.acquire();
                    if (storage.size() == 0)
                            System.out.println(productName
                                            + "'s are out of stock");

                    else if (storage.size() > 0) {
                            int amount = new Random().nextInt(5);

                            if (amount == 0)
                                    System.out.println("A Customer just " +
                                                    "browse and buys nothing");

                            else if (amount > storage.size())
                                    System.out.println("less than " + amount
                                                    + " products in shelf");
                            else {
                                    System.out.println("A Customer wants to buy "
                                                    + amount
                                                    + " items");

                                    for (int i = 0; i < amount; i++) {
                                            storage.poll();

                                            System.out.println("A Consumer buys a "
                                                            + productName
                                                            + ", items left "
                                                            + storage.size());
                                    }
                            }
                    }
            } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
                } finally {
                        ShelfSemaphore.release();
                }
        }
}
```

```
public class Producer implements Runnable {

        private Shelf shelf;

        public Producer(Shelf shelf) {
                this.shelf = shelf;
        }

        public void run() {
                for (int i = 0; i < 10; i++) {
                        shelf.ShelfRefresh();
                }
        }

}
```

```
public class Consumer implements Runnable {

        private Shelf shelf;

        public Consumer(Shelf shelf) {
                this.shelf = shelf;
        }

        public void run() {
                for (int i = 0; i < 10; i++) {
                        shelf.buy();
                }
                System.out.println("It seems no more Consumers will " +
                                "pay the store a visit for today");
        }
}
```

```
public class MainProducerConsumer {

        public static void main(String[] args) {

                Shelf shelf = new Shelf(30, "mango");

                Producer producer = new Producer(shelf);
                Thread thread1 = new Thread(producer);

                Consumer consumer = new Consumer(shelf);
                Thread thread2 = new Thread(consumer);

                thread1.start();
```

```
                thread2.start();

                try {
                        thread1.join();
                        thread2.join();
                        Thread.sleep(4000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }

                System.out.println("Store is closing");
        }
}
```

-

```
The Producer updates the mango's shelf with new items #of items: 5
A Customer wants to buy 2 items
A Consumer buys a mango, items left 4
A Consumer buys a mango, items left 3
A Customer wants to buy 2 items
A Consumer buys a mango, items left 2
A Consumer buys a mango, items left 1
A Customer just browse and buys nothing
less than 3 products in shelf
less than 2 products in shelf
less than 4 products in shelf
The Producer updates the mango's shelf with new items #of items: 8
A Customer wants to buy 1 items
A Consumer buys a mango, items left 7
The Producer updates the mango's shelf with new items #of items: 14
A Customer wants to buy 3 items
A Consumer buys a mango, items left 13
A Consumer buys a mango, items left 12
A Consumer buys a mango, items left 11
The Producer updates the mango's shelf with new items #of items: 17
A Customer wants to buy 4 items
A Consumer buys a mango, items left 16
A Consumer buys a mango, items left 15
A Consumer buys a mango, items left 14
A Consumer buys a mango, items left 13
The Producer updates the mango's shelf with new items #of items: 20
The Producer updates the mango's shelf with new items #of items: 27
The Producer updates the mango's shelf with new items #of items: 30
Shelf is full of mangos
Shelf is full of mangos
Shelf is full of mangos
A Customer just browse and buys nothing
It seems no more Consumers will pay the store a visit for today
Store is closing
```

- :

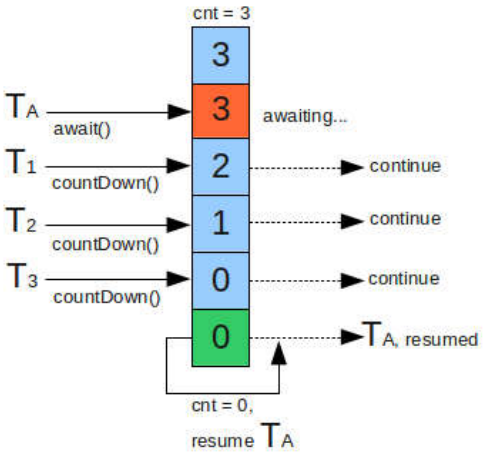
Το βασικό στοιχείο του παραπάνω παραδείγματος είναι ότι υπάρχει συνέπεια στο κρίσιμο τμήμα του κώδικα και προστατεύονται πλήρως οι μοιραζόμενες περιοχές των νημάτων. Επίσης ένα ακόμη βασικό στοιχείο είναι η αρχικοποίηση του αντικειμένου τύπου Semaphore. Εξετάζοντας τις παραμέτρους αρχικοποίησής του βλέπουμε ότι δέχεται ως παράμετρο την τιμή 1, με σκοπό να δημιουργήσουμε έ ν α

δυαδικό σημαφόρο (binary Semaphore). Ο δυαδικός σημαφόρος προστατεύει την πρόσβαση στη μοιραζόμενη μεταβλητή Shelf δίνοντας έτσι το δικαίωμα προσπέλασής και επεξεργασίας της στο κρίσιμο τμήμα είτε των αντικειμένων της κλάσης Παραγωγού είτε τα αντικείμενα της κλάσης Καταναλωτή. Δηλαδή όταν για παράδειγμα ένα αντικείμενο της κλάσης Παραγωγού αποκτά την μεταβλητή 1 του σημαφόρου και ως επακόλουθο την πρόσβαση στο κρίσιμο τμήμα τότε όταν ένα αντικείμενο της κλάσης Καταναλωτή προσπαθεί να αποκτήσει πρόσβαση στο κρίσιμο τμήμα και στις μοιραζόμενες μεταβλητές πρέπει να περιμένει λόγο ότι ο σημαφόρος που προσπαθεί να αποκτήσει είναι μηδέν, δηλαδή το κρίσιμο τμήμα και η μοιραζόμενη μεταβλητή έχει καταληφθεί από το αντικείμενο κλάσης Παραγωγού. Έτσι λοιπόν μέχρι να να ολοκληρωθεί η εργασία του του Παραγωγού επί το κρίσιμο τμήμα ο Καταναλωτής πρέπει να περιμένει και αντιστρόφως.

Τέλος η δεύτερη παράμετρος με την οποία αρχικοποιήθηκε ο σημαφόρος είναι μια λογική τιμή true. Με αυτήν την αρχικοποίηση θέτουμε τον σημαφόρο να είναι πιο “δίκαιος” μεταξύ των εν αναμονή νημάτων. Δηλαδή όταν επιλέγεται πιο νήμα θα αποκτήσει πρόσβαση στον σημαφόρο, θα είναι εκείνο που βρισκόταν περισσότερο σε κατάσταση αναμονής. Εάν δεν τοποθετούσαμε τη δεύτερη παράμετρο είτε αντί για true είχαμε false, τότε η επιλογή των νημάτων για να αποκτήσουν πρόσβαση στον σημαφόρο θα γινόταν χωρίς κριτήριο.

Διαχείριση συγχρονισμού με την ολοκλήρωση συνόλου εργασιών

Υπάρχουν περιπτώσεις σε εφαρμογές όπου τα συντρέχοντα νήματα πρέπει να περιμένουν έως ότου ένα σύνολο λειτουργιών έχουν πλήρως εκτελεστεί, για παράδειγμα η αναμονή της εφαρμογής βίντεο-συνδιαλλαγής που αναμένει τον επιτρεπόμενο ελάχιστο αριθμό χρηστών προς συνομιλία. Η κλάση που υλοποιεί την συγκεκριμένη περίπτωση είναι η κλάση της βιβλιοθήκης Java Concurrency API, που ονομάζεται CountDownLatch. Αντικείμενα της κλάσης αρχικοποιούνται με έναν ακέραιο αριθμό που είναι ένας εσωτερικός μετρητής, ο οποίος συμβολίζει τον αριθμό των νημάτων που θα πρέπει να ολοκληρώσουν την εκτέλεσή τους ώστε ένα νήμα που βρίσκεται σε αναμονή να ξεκινήσει την δική του εκτέλεση αφού τα υπόλοιπα έχουν ολοκληρώσει τις δικές τους εργασίες. Όταν ένα νήμα θέλει να αναμένει κάποια εργασία να ολοκληρωθεί τότε καλεί την μέθοδο await(). Η μέθοδος αυτή θέτει το καλούμενο νήμα σε κατάσταση αναμονής έως ότου άλλες εργασίες ολοκληρωθούν. Όταν μία από αυτές τις λειτουργίες ολοκληρωθούν καλούν την μέθοδο countDown(), που μειώνει κατά μία μονάδα τον εσωτερικό μετρητή του αντικειμένου της κλάσης CountDownLatch. Μόλις γίνει μηδέν τότε ξεκινά την δική του εκτέλεση το εν αναμονή των υπολοίπων νημάτων νήμα. Η παρακάτω απεικόνιση μας παρουσιάζει την βασική ιδέα λειτουργίας της κλάσης CountDownLatch, όπου ένα νήμα περιμένει τρία άλλα νήματα αφού ολοκληρώσουν και τα τρία την εκτέλεσή τους, τότε ξεκινά αυτό την δική του εκτέλεση:



Στο παρακάτω παράδειγμα προσομοιώνουμε πάλι την εφαρμογή του εξυπηρετητή, κλάση Server που είχαμε αρχικώς εξετάσει στην αρχική ενότητα. Σε αυτό το παράδειγμα προσομοιώνουμε πάλι τον εξυπηρετητή αλλά τώρα με την κλάση `CountDownLatch`. Το νήμα του εξυπηρετητή αναμένει κάνοντας χρήση της μεθόδου `await()`, τις υπηρεσίες που είναι αντικείμενα τύπου Service όπως φαίνεται στην παρακάτω υλοποίηση, του να αρχικοποιηθούν και να ολοκληρώσουν την εκτέλεσή τους. Μόλις την ολοκληρώσουν ο εξυπηρετητής είναι έτοιμος να δεχθεί τα διαδικτυακά αιτήματα.

-

```
import java.util.concurrent.TimeUnit;

public class Service implements Runnable {

	private String serviceName;
	private Server server;

	public Service(String serviceName, Server server) {
		this.serviceName = serviceName;
		this.server = server;
	}

	public void run() {

		//do your computations - calculations
		long duration = (long) (Math.random() * 10);
		try {
			TimeUnit.SECONDS.sleep(duration);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		//when you are ready call this method
		//to decrement the latch
		server.ReadyService(serviceName);
	}
}
```

```
import java.util.concurrent.CountDownLatch;

public class Server implements Runnable {

	private final CountDownLatch latch;

	public Server() {
		//wait for 3 services to finish
		this.latch = new CountDownLatch(3);
	}

	public void ReadyService(String name) {
		System.out.println("Service "+ name +" is ready");
		//Service's operation is finished
		latch.countDown();
	}

	public void run() {
		long startTime = System.currentTimeMillis();

		System.out.println("Server initialization, awaiting for #"
				+latch.getCount() +" services to finish");
		try {
			//waiting for 3 services to finish
			latch.await();

			//when all the other threads have finished their exexutions
			//start this thread execution
			System.out.println("Server: all services are ready");

			System.out.println("Server's setting some last parameters");
			Thread.currentThread().sleep(3000);

			System.out.printf("Server is UP! ");

			long estimatedTime = System.currentTimeMillis() - startTime;
			System.out.println("time elapsed " + estimatedTime / 1000.0
					+ " seconds");

		} catch(InterruptedException e) {
			e.printStackTrace();
		}
	}

	public static void main(String[] args) {

		//server
		Server server = new Server();
		Thread serverThread = new Thread(server);

		//services
		Service webService = new Service("Web Service",server);
		Service messageService = new Service("MessageService Service",server);
		Service dataBaseService = new Service("Data Base Service",server);

		Thread webServiceThread = new Thread(webService);
		Thread messageServiceThread = new Thread(messageService);
```

```
                Thread dataBaseServiceThread = new Thread(dataBaseService);

                serverThread.start();
                webServiceThread.start();
                messageServiceThread.start();
                dataBaseServiceThread.start();
        }
}
```

-

```
Server initialization, awaiting for #3 services to finish
Service MessageService Service is ready
Service Web Service is ready
Service Data Base Service is ready
Server: all services are ready
Server's setting some last parameters
Server is UP! time elapsed 8.08 seconds
```

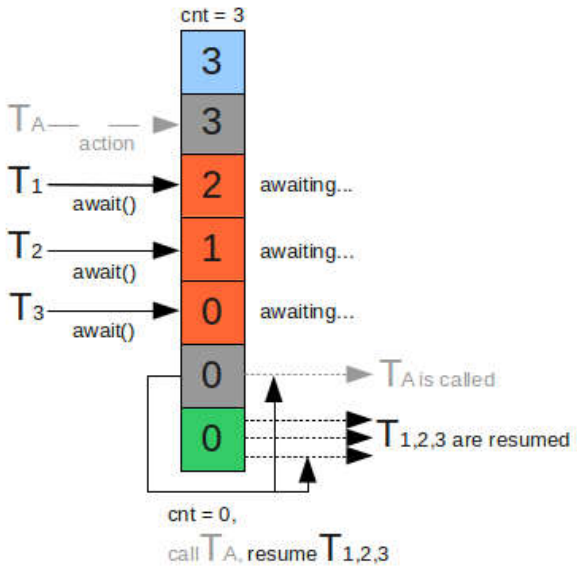
- Σχόλια:

Οι μέθοδοι αρχικοποίησης, `await()` και `countDown()` της κλάσης `CountDownLatch` που χρησιμοποιήσαμε είναι τα βασικά χαρακτηριστικά για να δημιουργήσουμε εφαρμογές που ένα νήμα αναμένει να ξεκινήσει την εκτέλεσή του όταν τα υπόλοιπα νήματα τερματίσουμε την δική τους εκτέλεση. Με την αρχικοποίηση της παραμέτρου της θέτουμε πόσες εργασίες πρέπει να ολοκληρωθούν για να ξεκινήσει το εν αναμονή νήμα την δική του εκτέλεση. Η μέθοδος `await()`, καλείται από το εν αναμονή νήμα για να αναμένει τα υπόλοιπα τον τερματισμό των εργασιών τους. Τέλος η μέθοδος `countDown()`, καλείτε από τα νήματα που έχουν ολοκληρώσει την εκτέλεση των εργασιών τους.

Αν δεν υπήρχε η συγκεκριμένη κλάση θα έπρεπε να χρησιμοποιήσουμε τις μεθόδους `wait()` και `notifyAll()` κάτι που απαιτεί μεγαλύτερη προγραμματιστική εμπειρία και περισσότερες γραμμές κώδικα, με μεγαλύτερες πιθανότητες σφαλμάτων.

Συγχρονισμός εργασιών σε καθορισμένο κοινό σημείο

Η κλάση `CyclicBarrier` έχει την δυνατότητα να συγχρονίζει δύο ή και περισσότερα νήματα σε ένα καθορισμένο κοινό σημείο. Τα χαρακτηριστικά λειτουργίας της συγκεκριμένης κλάσης είναι παρόμοια με της `CountDownLatch` αφού και εδώ τα νήματα πρέπει να αναμένουν τα υπόλοιπα νήματα να ολοκληρώσουν όλα τις εργασίες τους ωστόσο υπάρχουν μερικές διαφορές με το τρόπο λειτουργίας σε αυτήν την νέα κλάση. Η κλάση `CyclicBarrier` επιτρέπει πολλαπλά νήματα να αναμένουν το ένα με το άλλο σε ένα καθορισμένο κοινό σημείο έως ότου όλα αυτά τα νήματα ολοκληρώσουν τις εργασίες τους. Έπειτα μπορούν να συνεχίσουν στο επόμενο τμήμα των εργασιών τους ή το τελικό τμήμα εργασιών της εφαρμογής μας όταν οι εργασίες του κάθε νήματος έχει ολοκληρωθεί ξεχωριστά. Παρακάτω απεικονίζεται ο τρόπος λειτουργίας του `CyclicBarrier`:



Η κλάση `CyclicBarrier` αρχικοποιείται με παράμετρο ένα ακέραιο αριθμό, που υποδεικνύει τον αριθμό των νημάτων που θα συγχρονιστούν σε ένα κοινό καθορισμένο σημείο. Όταν ένα από τα παραπάνω νήματα καταφθάσει στο συγκεκριμένο σημείο του κώδικα της εφαρμογής, καλεί την μέθοδο `await()` για να περιμένει τα υπόλοιπα νήματα στο κοινό σημείο. Όταν και το τελευταίο νήμα καλέσει την μέθοδο `await()`, παύει η αναμονή όλων των εν αναμονή νημάτων και συνεχίζουν τις προς εκτέλεση εργασίες τους.

Ένα ενδιαφέρον χαρακτηριστικό της κλάσης `CyclicBarrier` είναι ότι ως δεύτερο παράμετρο κατά την αρχικοποίηση των αντικειμένων της κλάσης της μπορούμε να εισάγουμε ένα `Runnable` αντικείμενο, το οποίο η κλάση θα το εκτελέσει όταν όλα τα νήματα θα έχουν φτάσει το κοινό καθορισμένο σημείο. Αυτό το χαρακτηριστικό καθιστά την συγκεκριμένη κλάση κατάλληλη για τον παραλληλισμό των εργασιών χρησιμοποιώντας προγραμματιστικές τεχνικές “διαίρει και βασίλευε”.

Στο παρακάτω παράδειγμα υλοποιούμε όλα τα προαναφερθέντα σε ένα αφηρημένο πρόβλημα “διαίρει και βασίλευε”. Για ένα γενικό και αφηρημένο πρόβλημα μοιράζουμε το συνολικό χρόνο εκτέλεσής του σε 4 νήματα – εργάτες. Μόλις ολοκληρώσουν τις εργασίες τους, ενημερώνουν την main μέθοδο και με την σειρά της έπειτα αρχίζει να εκτελεί ένα νέο `Runnable` αντικείμενο.

-

```
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.BrokenBarrierException;
import java.util.Date;

public class ParallelDistribution {
        private static final int MAX_THREADS = 4;

        private static class WorkerThread implements Runnable {

                private CyclicBarrier cyclicBarrier;
                private String name;
                private double workingTime;
```

```
        //initializing a worker thread to work in parallel with other workers
        public WorkerThread(CyclicBarrier cyclicBarrier, String name,
                        double workingTime) {
                this.name = name;
                this.cyclicBarrier = cyclicBarrier;
                this.workingTime = workingTime;
        }

        public void run() {
                try {
                        Date startTime = new Date();

                        // start executing on your part of a heavy-weight operation
                        System.out.println(startTime + " : Thread#" + name
                                        + " Executing "
                                        + (Integer.parseInt(name) + 1)
                                        + "/" + MAX_THREADS
                                        + " Part of Work");

                        // processing of Thread's work
                        Thread.sleep((long) workingTime * 1000);

                        // Thread's work is finished
                        System.out.println(new Date() + " : Thread#" + name
                                        + " Finished Part of "
                                        + (Integer.parseInt(name) + 1)
                                        + "/" + MAX_THREADS
                                        + " Part of Work");

                        //wait for each other thread to finish its work
                        cyclicBarrier.await();

                } catch (BrokenBarrierException e) {
                        e.printStackTrace();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}

public static void main(String[] args) {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(MAX_THREADS,
                        new Runnable() {

                                //this thread executes when all the workers-threads
                                //have finished their work
                                public void run() {
                                        System.out.printf("MainThread Report: ");
                                        System.out.printf("Cyclic Barrier Finished " +
                                                        "processing a heavy-weight " +
                                                        "operation");
                                }
                        });

        System.out.println("Distributing a Heavy-Weight Job in "
                        + MAX_THREADS+ " Threads");

        // total process time, if only 1 thread was going to
```

```
                // execute a heavy-weight operation
                double totalTime = (double) (Math.random() + 1) * 25;

                System.out.println("Spawning Threads");
                for (int i = 0; i < MAX_THREADS; i++) {
                        Thread t = new Thread(new WorkerThread(cyclicBarrier, ("" + i),
                                        totalTime / MAX_THREADS));

                        try {
                                Thread.sleep(100);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        } //end of try-catch

                        t.start();
                }
        } //end of main
}
```

-

```
Distributing a Heavy-Weight Job in 4 Threads
Spawning Threads
Sun Jan 26 00:41:25 EET 2014 : Thread#2 Executing 3/4 Part of Work
Sun Jan 26 00:41:25 EET 2014 : Thread#1 Executing 2/4 Part of Work
Sun Jan 26 00:41:25 EET 2014 : Thread#0 Executing 1/4 Part of Work
Sun Jan 26 00:41:25 EET 2014 : Thread#3 Executing 4/4 Part of Work
Sun Jan 26 00:41:34 EET 2014 : Thread#2 Finished Part of 3/4 Part of Work
Sun Jan 26 00:41:34 EET 2014 : Thread#1 Finished Part of 2/4 Part of Work
Sun Jan 26 00:41:34 EET 2014 : Thread#0 Finished Part of 1/4 Part of Work
Sun Jan 26 00:41:34 EET 2014 : Thread#3 Finished Part of 4/4 Part of Work
MainThread Report: Cyclic Barrier Finished processing a heavy-weight operation
```

- Σχόλια:

Όπως αναφέρθηκε προηγουμένως, η κλάση `CyclicBarrier` έχει ένα εσωτερικό μετρητή, για τον έλεγχο της άφιξης των νημάτων στο κοινό σημείο συγχρονισμού. Κάθε νήμα που φτάνει στο σημείο καλεί την μέθοδο `await()` για να ενημερώσει το αντικείμενο `CyclicBarrier` ότι έχει καταφτάσει ένα νήμα στο καθορισμένο σημείο συγχρονισμού. Τότε η κλάση `CyclicBarrier` θέτει το νήμα σε κατάσταση αναμονής έως ότου όλα τα υπόλοιπα υπό εκτέλεση νήματα καταφθάσουν στο σημείο συγχρονισμού. Τότε (προαιρετικώς) μπορεί το αντικείμενο της κλάσης `CyclicBarrier` να εκτελέσει ένα `Runnable` αντικείμενο, αφού έχει οριστεί στην αρχικοποίηση του συγκεκριμένο αντικειμένου τύπου `CyclicBarrier`.

Εκτέλεση συγχρονισμένων τμηματικών εργασιών

Στην Java, έχει δημιουργηθεί ένα καινούργιος μηχανισμός συγχρονισμού των εργασιών ενός προγράμματος που μας παρέχει την δυνατότητα για να εκτελούμε τμηματικά και σε φάσεις συγχρονισμένες εργασίες. Ο μηχανισμός αυτός είναι η κλάση `Phaser`. Όταν έχουμε ένα πρόγραμμα που το κύριο φόρτο εργασίας του είναι χωρισμένο σε βήματα (φάσεις), τότε η παραπάνω κλάση έχει κατάλληλους μηχανισμούς ώστε να συγχρονίζει τα συντρέχοντα νήματα στο καθορισμένο σημείο που υποδεικνύει το τέλος του κάθε βήματος. Επίσης οι μηχανισμοί που κατέχει η συγκεκριμένη κλάση, έχουν την δυνατότητα ώστε μόνο όταν φτάσουν όλα τα νήματα σε τέλος ενός βήματος, τότε να ξεκινήσει το επόμενο βήμα εκτέλεσης της εφαρμογής.

Κατά την αρχικοποίηση των αντικειμένων της κλάσης, ως παράμετρο δέχονται έναν ακέραιο αριθμό

που υποδεικνύει τον αριθμό των εργασιών που θα λάβουν μέρος στην διαδικασία του συγχρονισμού των βημάτων εκτέλεσης της εφαρμογής. Δηλαδή έστω ότι μια εργασία απαρτίζεται από τρία βήματα, στο κάθε βήμα στο καθορισμένο σημείο που ολοκληρώνεται τα νήματα που θα απασχοληθούν πρέπει το ένα να αναμένει το άλλο.

Αρχικά λοιπόν πρέπει να αρχικοποιήσουμε τον `Phaser` μας με την παράμετρο τρία διότι τρία είναι τα νήματα που θα απασχοληθούν. Έπειτα σε κάθε καθορισμένο σημείο που ολοκληρώνεται ένα βήμα, το κάθε νήμα πρέπει να χρησιμοποιεί την μέθοδο `arriveAndAwaitAdvance` της κλάσης `Phaser`. Η συγκεκριμένη μέθοδος αναγκάζει όλα τα νήματα να αναμένουν το ένα το άλλο έως ότου όλα έχουν καλέσει αυτήν την μέθοδο και μπορούν να συνεχίσουν να εργάζονται στο επόμενο βήμα. Σε κάθε βήμα καλώντας την μέθοδο `getPhase()`, της κλάσης `Phaser`, μπορούμε να γνωρίζουμε σε πιο βήμα-φάση βρίσκεται το πρόγραμμά μας. Όταν όλα τα νήματα προχωρούν στην εκτέλεσή τους μετά το τέλος της μεθόδου `arriveAndAwaitAdvance` τότε η το βήμα-φάση της μεθόδου `getPhase()` αυξάνεται κατά μία μονάδα.

Στο παρακάτω παράδειγμα υλοποιούμε τον υπολογισμό όλων των τριγωνομετρικών αριθμών των 30, 45, 60 μοιρών χρησιμοποιώντας τις μεθόδους υπολογισμού τριγωνομετρικών σειρών Taylor όπως υλοποιήσαμε σε προηγούμενη ενότητα.. Κάθε μοίρα υπολογίζεται σε τρεις φάσεις. Και χρησιμοποιούμε τρία νήματα για τον υπολογισμό όλων των τριγωνομετρικών αριθμών.

-

```
import java.util.concurrent.Phaser;

public class PhaserTrigSeries {

        public static double TaylorSine(double degrees) {
                double sum = 0;
                for (int counter = 0; counter < 100; counter++) {
                        sum += ((Math.pow(-1.0, counter)
                                        * Math.pow(degrees2rad(degrees),
                                                        (2 * counter + 1)))
                                                        / (factorial(2 * counter + 1)));
                }
                return sum;
        }

        public static double TaylorCosine(double degrees) {
                double sum = 0;
                for (int counter = 0; counter < 100; counter++) {
                        sum += (Math.pow(-1.0, counter)
                                        * Math.pow(degrees2rad(degrees),
                                                        2.0 * counter)
                                                        / (factorial(2.0 * counter)));
                }
                return sum;
        }

        public static double factorial(double n) {
                if (n <= 1)
                        return 1;
                else
                        return n * factorial(n - 1);
        }

        public static double degrees2rad(double degrees) {
```

```
            return (degrees * Math.PI / 180.0);
    }

    public static void main(String[] args) {

            final int[] degreesArray = { 30, 45, 60 };
            final double[] sinArray = new double[3];
            final double[] cscArray = new double[3];
            final double[] cosArray = new double[3];
            final double[] secArray = new double[3];
            final double[] tanArray = new double[3];
            final double[] cotArray = new double[3];

            final int workStep = degreesArray.length;
            int workers = 3;

            //create a phaser with 3 workers-threads
            final Phaser phaser = new Phaser(workers);

            new Thread("Worker1") {
                    //Thread - Worker to calculate
                    //sine(sin) & cosine(cos)

                    public void run() {
                            for (int i = 0; i < workStep; i++) {

                                    try {
                                            Thread.sleep(20);
                                    } catch (InterruptedException e) {
                                            e.printStackTrace();
                                    }

                                    sinArray[i] = TaylorSine(degreesArray[i]);
                                    cosArray[i] = TaylorCosine(degreesArray[i]);
                                    System.out.println("Worker1, Phase: "
                                                    + (phaser.getPhase()+1)
                                                    + " degrees:" + degreesArray[i] + " sin:"
                                                    + sinArray[i] + " cos:" + cosArray[i]);

                                    //wait for the all the other threads to finish
                                    //their computations, in order to proceed to the
                                    //next phase - iteration
                                    phaser.arriveAndAwaitAdvance();
                            }
                    }
            }.start();

            new Thread("Worker2") {
                    //Thread - Worker to calculate
                    //cosecant(csc) & secant(sec)

                    public void run() {
                            for (int i = 0; i < workStep; i++) {

                                    try {
                                            Thread.sleep(40);
                                    } catch (InterruptedException e) {
                                            e.printStackTrace();
                                    }
```

```
                                cscArray[i] = 1 / sinArray[i];
                                secArray[i] = 1 / cosArray[i];

                                System.out.println("Worker2, Phase: "
                                                + (phaser.getPhase()+1)
                                                + " degrees:" + degreesArray[i] + " csc:"
                                                + cscArray[i] + " sec:" + secArray[i]);

                                //wait for the all the other threads to finish
                                //their computations, in order to proceed to the
                                //next phase - iteration
                                phaser.arriveAndAwaitAdvance();

                        }
                }
        }.start();

        new Thread("Worker3") {
                //Thread - Worker to calculate
                //tangent(tan) & cotangent(cot)

                public void run() {
                        for (int i = 0; i < workStep; i++) {

                                try {
                                        Thread.sleep(60);
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }

                                tanArray[i] = sinArray[i]
                                                / cosArray[i];
                                cotArray[i] = cosArray[i]
                                                / sinArray[i];

                                System.out.println("Worker3, Phase: "
                                                + (phaser.getPhase()+1)
                                                + " degrees:" + degreesArray[i] + " tan:"
                                                + tanArray[i] + " cot:" + cotArray[i]);

                                //wait for the all the other threads to finish
                                //their computations, in order to proceed to the
                                //next phase - iteration
                                phaser.arriveAndAwaitAdvance();
                        }
                }
        }.start();
    }// end of main
}// end of class
```

-

```
Worker1, Phase: 1 degrees:30 sin:0.49999999999999994 cos:0.8660254037844386
Worker2, Phase: 1 degrees:30 csc:2.0000000000000004 sec:1.1547005383792517
Worker3, Phase: 1 degrees:30 tan:0.5773502691896257 cot:1.7320508075688774
Worker1, Phase: 2 degrees:45 sin:0.7071067811865475 cos:0.7071067811865475
```

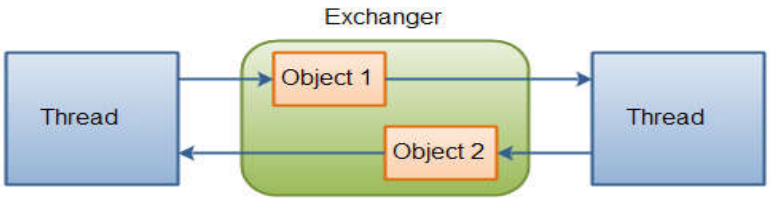
```
Worker2, Phase: 2 degrees:45 csc:1.4142135623730951 sec:1.4142135623730951
Worker3, Phase: 2 degrees:45 tan:1.0 cot:1.0
Worker1, Phase: 3 degrees:60 sin:0.8660254037844385 cos:0.5000000000000001
Worker2, Phase: 3 degrees:60 csc:1.1547005383792517 sec:1.9999999999999996
Worker3, Phase: 3 degrees:60 tan:1.7320508075688765 cot:0.577350269189626
```

- Σχόλια:

Όπως αναφέρθηκε προηγουμένως, το αντικείμενο τύπου Phaser, γνωρίζει τον αριθμό των νημάτων που θα εργαστούν και στο τέλος του κάθε βήματος (φάσης) θα συγχρονιστούν. Κάθε φορά που αναμένει ένα νήμα το άλλο να φτάσει στο καθορισμένο σημείο που σηματοδοτεί το τέλος μιας λειτουργίας, η Phaser το θέτει σε κατάσταση αναμονής έως ότου όλα τα υπόλοιπα νήματα φτάσουν στο συγκεκριμένο σημείο και καλέσουν την μέθοδο arriveAndAwaitAdvance.

Ανταλλαγή δεδομένων μεταξύ συγχρονισμένων εργασιών

Ένας επιπλέον σύνθετος μηχανισμός συγχρονισμού της Java, η κλάση Exchanger, μας επιτρέπει ανταλλαγή δεδομένων μεταξύ δύο συγχρονισμένων εργασιών. η κλάση Exchanger αφού αρχικώς συγχρονίσει δύο νήματα σε ένα καθορισμένο σημείο, στη συνέχεια ανταλλάσσουν μία δομή δεδομένων όπως απεικονίζεται παρακάτω:



Η κλάση Exchanger μπορεί να μας φανεί χρήσιμη, σε καταστάσεις όπου ένα νήμα χρειάζεται από ένα άλλο νήμα τα δεδομένα του για να συνεχίσει τους υπολογισμούς του. Τα δύο αυτά νήματα έχουν μία κοινή δομή δεδομένων για να ανταλλάζουν δεδομένα μεταξύ τους.

Στο Παρακάτω παράδειγμα υλοποιούμε ξανά τον υπολογισμό των τριγωνομετρικών αριθμών των 30, 45 και 60 μοιρών με τις συναρτήσεις σειρών Taylor. Ένα νήμα υπολογίζει το ημίτονο ενώ ένα άλλο ταυτοχρόνως το συνημίτονό τους. Όταν τελειώσουν τους υπολογισμούς, ανταλλάζουν τα αποτελέσματα μεταξύ τους για να υπολογίσουν την εφαπτομένη αντιστοίχως.

-

```
import java.util.ArrayList;
import java.util.concurrent.Exchanger;

public class SinTaylorExchanger implements Runnable {

        private int[] degrees = { 30, 45, 60 };
        private double[] sinArray = new double[degrees.length];

        private ArrayList<Double> buffer;
        private final Exchanger<ArrayList<Double>> exchanger;

        //constructor
```

```
	public SinTaylorExchanger(ArrayList<Double> buffer,
			Exchanger<ArrayList<Double>> exchanger) {
		this.buffer = buffer;
		this.exchanger = exchanger;
	}

	public void run() {
		//compute your sine part,
		for (int i = 0; i < degrees.length; i++) {
			sinArray[i] = TaylorSine(degrees[i]);

			//add your sine computations to your buffer
			buffer.add(sinArray[i]);

			System.out.println(Thread.currentThread().getName()
					+" degrees " + degrees[i] +
					" sin:" + sinArray[i]);
		}

		//exchange your sine buffer to the cosine Thread
		//and get the cosine buffer results
		try {

			Thread.sleep(20);
			System.out.println("-Threads Data Exchange Proceed-");
			buffer = exchanger.exchange(buffer);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}

		//compute the rest of the Trigonometric identities
		for (int i = 0; i < degrees.length; i++) {
			double cos = buffer.get(i);

			System.out.println(Thread.currentThread().getName()
					+" degrees: "+degrees[i]+" cos:"+cos
					+" tan is:" + (sinArray[i]/cos));

		}

	}

	public double TaylorSine(int degrees) {
		double sum = 0;
		for (int counter = 0; counter < 100; counter++) {
			sum += ((Math.pow(-1.0, counter) * Math.pow(degrees2rad(degrees),
					(2 * counter + 1))) / (factorial(2 * counter + 1)));
		}
		return sum;
	}

	public double factorial(int n) {
		if (n <= 1)
			return 1;
		else
			return n * factorial(n - 1);
	}

	public double degrees2rad(double degrees) {
```

```
                return (degrees * Math.PI / 180.0);
        }
}
```

```
import java.util.ArrayList;
import java.util.concurrent.Exchanger;

public class CosTaylorExchanger implements Runnable {

        private int[] degrees = { 30, 45, 60 };
        private double[] cosArray = new double[degrees.length];

        private ArrayList<Double> buffer;
        private final Exchanger<ArrayList<Double>> exchanger;

        //constructor
        public CosTaylorExchanger(ArrayList<Double> buffer,
                        Exchanger<ArrayList<Double>> exchanger) {
                this.buffer = buffer;
                this.exchanger = exchanger;
        }

        public void run() {
                //compute your cosine part,
                for (int i = 0; i < degrees.length; i++) {
                        cosArray[i] = TaylorCosine(degrees[i]);

                        //add your cosine computations to your buffer
                        buffer.add(cosArray[i]);

                        System.out.println(Thread.currentThread().getName()+
                                        " degrees " + degrees[i] +
                                        " cos:" + cosArray[i]);
                }

                //exchange your cosine buffer to the sine Thread
                //and get the sine buffer results
                try {
                        Thread.sleep(40);
                        System.out.println("-Threads Data Exchange Proceed-");
                        buffer = exchanger.exchange(buffer);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }

                //compute the rest of the Trigonometric identities
                for (int i = 0; i < degrees.length; i++) {
                        double sin = buffer.get(i);

                        System.out.println(Thread.currentThread().getName()
                                        +" degrees: " + degrees[i] + " sin:" + sin
                                        +" tan is:" + (sin/cosArray[i]));
                }
        }
```

```
        public double TaylorCosine(int degrees) {
                double sum = 0;
                for (int counter = 0; counter < 100; counter++) {
                        sum += (Math.pow(-1.0, counter)
                                        * Math.pow(degrees2rad(degrees), 2.0 * counter)
                        / (factorial(2 * counter)));
                }
                return sum;
        }

        public double factorial(int n) {
                if (n <= 1)
                        return 1;
                else
                        return n * factorial(n - 1);
        }

        public double degrees2rad(double degrees) {
                return (degrees * Math.PI / 180.0);
        }
}
```

```
public class MainExchanger {

        public static void main(String[] args) {

                ArrayList<Double> buffer1=new ArrayList<Double>();
                ArrayList<Double> buffer2=new ArrayList<Double>();

                Exchanger<ArrayList<Double>> exchanger =
                                                                new Exchanger<ArrayList<Double>>();

                SinTaylorExchanger sinCalc = new SinTaylorExchanger(buffer1, exchanger);
                CosTaylorExchanger cosCalc = new CosTaylorExchanger(buffer2, exchanger);

                Thread sinThread = new Thread(sinCalc,"Thread1");
                Thread cosThread = new Thread(cosCalc,"Thread2");

                sinThread.start();
                cosThread.start();
        }
}
```

-

```
-Thread1 starts Sine Taylor Series Computations-
-Thread2 starts Cosine Taylor Series Computations-
Thread1 degrees:30 computes sin:0.49999999999999994
Thread1 degrees:45 computes sin:0.7071067811865475
Thread2 degrees:30 computes cos:0.8660254037844386
Thread1 degrees:60 computes sin:0.8660254037844385
Thread2 degrees:45 computes cos:0.7071067811865475
Thread2 degrees:60 computes cos:0.5000000000000001
```

```
-Thread1 starts to exchange his data-
-Thread2 starts to exchange his data-
Thread1 degrees: 30 got cos:0.8660254037844386, so tan is:0.5773502691896257
Thread2 degrees: 30 got sin:0.49999999999999994, so tan is:0.5773502691896257
Thread1 degrees: 45 got cos:0.7071067811865475, so tan is:1.0
Thread2 degrees: 45 got sin:0.7071067811865475, so tan is:1.0
Thread1 degrees: 60 got cos:0.5000000000000001, so tan is:1.7320508075688765
Thread2 degrees: 60 got sin:0.8660254037844385, so tan is:1.7320508075688765
```

-

Μέχρι το καθορισμένο σημείο που θα γίνει η ανταλλαγή των δεδομένων, το κάθε νήμα υπολογίζει τους δικούς του τριγωνομετρικούς αριθμούς, το ένα το ημίτονο το άλλο νήμα το συνημίτονο. Όταν και τα δύο νήματα καλούν την μέθοδο `exchange()`, το νήμα που υπολόγισε τα ημίτονα και τα αποθήκευσε στην δική της δομή δεδομένων προς ανταλλαγή (`buffer`) και αντιστοίχως το δεύτερο νήμα για τα συνημίτονα, τότε σε εκείνο το σημείο η κλάση `Exchanger`, συγχρονίζει τα δύο αυτά νήματα και ο `buffer` του ενός νήματος αντιγράφει τα δεδομένα του δεύτερου νήματος και αντιστρόφως. Στη συνέχεια τα δύο νήματα αφού έχουν τα απαραίτητα δεδομένα, μπορούν να υπολογίσουν τις εφαπτομένες των παραπάνω μοιρών.

Εκτελεστές νημάτων

Εισαγωγή στην βιβλιοθήκη εκτελεστών (Executors API)

Μέχρι στιγμής έχουμε εξετάσει τις βασικές δομές συντρέχοντος προγραμματισμού στην Java στις οποίες δημιουργούμε διάφορα `Runnable` αντικείμενα, στη συνέχεια τα εκτελούμε μέσω μία κλάσης `main`. Ο συγκεκριμένος τύπος προγραμματισμού είναι επαρκής για βασική χρήση νημάτων αλλά για πιο σύνθετο προγραμματισμό και για μεγάλες εφαρμογές συνήθως είναι χρήσιμοι μηχανισμοί υψηλότερου επιπέδου. Τέτοιες εφαρμογές είναι εκείνες που χρησιμοποιούν μεγάλο ή και μεταβλητό πλήθος εργασών που πρέπει να εκτελεστούν από νήματα. Εάν κρατήσουμε μόνο τους μηχανισμούς συγχρονισμού που έχουν αναφερθεί μέχρι τώρα, τότε οι μεγάλες εφαρμογές με πολλές συγχρονισμένες εργασίες αντιμετωπίζουν αρκετά προβλήματα διαχείρισης. Η εκτέλεση μιας εργασίας από ένα μόνο συγκεκριμένο νήμα σε μια εφαρμογή πολλών εργασιών μπορεί να επηρεάσει σημαντικά την απόδοση της εφαρμογής. Επίσης όσο μεγαλώνουν οι απαιτήσεις και το μέγεθος της εφαρμογής τόσο αυξάνεται ο αριθμός των νημάτων, με αποτέλεσμα να εξαντλούνται σταδιακά οι πόροι του υπολογιστή.

Οι μηχανισμοί που έχουν αναπτυχθεί από την Java για την επίλυση των παραπάνω προβλημάτων βρίσκονται στη διασύνδεση Εκτελεστές (Executor Interface) της `java.util.concurrent`, μαζί με τις υπό-υλοποιήσεις της.

Ο παραπάνω μηχανισμός έχει την ικανότητα διαχείρισης και εκτέλεσης Runnable αντικειμένων. Αρκεί μόνο να δημιουργήσουμε τα Runnable αντικείμενα, και να δοθούν έπειτα ως παράμετρος σε ένα Executor. Ο μηχανισμός αυτός διαχωρίζει τα στάδια της δημιουργίας μιας εργασίας και την εκτέλεσή της. Επίσης έχει μηχανισμούς ορθής χρήσης των νημάτων για να αυξάνεται η απόδοση της εφαρμογής, δηλαδή σε κάθε νέα ή υπάρχουσα εργασία που πρέπει να εκτελεστεί σε μία εφαρμογή, η διασύνδεση executor θα χρησιμοποιήσει τα ήδη υπάρχοντα νήματα χωρίς να δημιουργεί νέα.

Ένα ακόμη χαρακτηριστικό της διασύνδεσης Executor είναι η διασύνδεση Callable. Έχει ομοιότητες με την διασύνδεση Runnable, ωστόσο έχει συγκεκριμένες μεθόδους όπως την μέθοδο call() που επιστρέφει δεδομένα – αποτελέσματα πίσω για παράδειγμα στην main κλάση της εφαρμογής μας. Τα δεδομένα – αποτελέσματα που επιστρέφονται πίσω από την διασύνδεση Callable είναι ότι είναι αντικείμενα τύπου Future που υλοποιούν την διασύνδεση Future.

Δημιουργία Executor βασικής διαχείρισης εργασιών

Η πρώτη επαφή με την διασύνδεση Executor, είναι το να δημιουργήσουμε ένα αντικείμενο της κλάσης ThreadPoolExecutor. Μόλις έχουμε δημιουργήσει το συγκεκριμένο αντικείμενο μπορούμε να καταχωρίσουμε για εκτέλεση Runable ή Callable αντικείμενα.

Στο παρακάτω παράδειγμα εκτελούμε 4 νήματα (κλάση `TaskFactorial`) που το καθένα ως εργασία υπολογίζει το παραγοντικού ενός τυχαίου αριθμού. Ωστόσο τα νήματα τα διαχειρίζεται για την εκτέλεσή τους ένας executor (κλάση `FactorialManager`) και όχι το κύριο γονικό νήμα της main.

-

```
import java.util.Date;
import java.util.concurrent.TimeUnit;

public class TaskFactorial implements Runnable {

	private Date initDate;
	private String name;

	//thread constructor
	public TaskFactorial(String name) {
		initDate = new Date();
		this.name = name;
	}

	public void run() {
		System.out.printf("%s: Task %s: Created on: %s¥n", Thread
				.currentThread().getName(), name, initDate);
		System.out.printf("%s: Task %s: Started on: %s¥n", Thread
				.currentThread().getName(), name, new Date());
		try {
			Long number = Math.round(Math.random()*10);

			Long duration = (long) (Math.random() * 10);
			TimeUnit.SECONDS.sleep(duration);

			System.out.printf("%s: Task %s: has calculated " +
					"the factorial of the number %d which is %d¥n",
					Thread.currentThread().getName(), name, number,
				factorial(number));

		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.printf("%s: Task %s: Finished on: %s¥n", Thread
				.currentThread().getName(), name, new Date());

	}

	public long factorial(long n) {
		if (n <= 1)
			return 1;
		else
			return n * factorial(n - 1);
	}
```

```
}
```

```
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

public class FactorialManagerCashedPool {

	private ThreadPoolExecutor executor;
	private long startTime;

	//executor constructor
	public FactorialManagerCashedPool() {
		executor = (ThreadPoolExecutor) Executors.newCachedThreadPool();
		startTime = System.currentTimeMillis();
	}

	public void executeTasks(TaskFactorial taskFactorial) {
		System.out.printf("Manager: A new task has arrived¥n");

		// start executing its threads that is just registered
		executor.execute(taskFactorial);

		System.out.printf("Manager: Pool Size: %d¥n",
				executor.getPoolSize());

		System.out.printf("Manager: Active Count: %d¥n",
				executor.getActiveCount());

		System.out.printf("Manager: Completed Tasks: %d¥n",
				executor.getCompletedTaskCount());
	}

	// finalize the executor thread
	public void endServer() {
		executor.shutdown();

		while (executor.isTerminating()) {/* wait until it shutdowns */}

		System.out.println("¥nExecutor shuts down, Completed Tasks:"
				+ executor.getCompletedTaskCount()+" time elapsed in seconds "
				+(System.currentTimeMillis()-startTime)/1000.0);
	}
}
```

```
public class MainFactorialManager {

	public static void main(String[] args) {
		FactorialManager factorialManager = new FactorialManager();

		//register to the executor 4 threads to run
		for (int i = 0; i < 4; i++) {
			TaskFactorial taskFactorial = new TaskFactorial("Task " + i);
```

```
                        factorialManager.executeTasks(taskFactorial);
                }

                //stop executor
                factorialManager.endServer();
        }
}
```

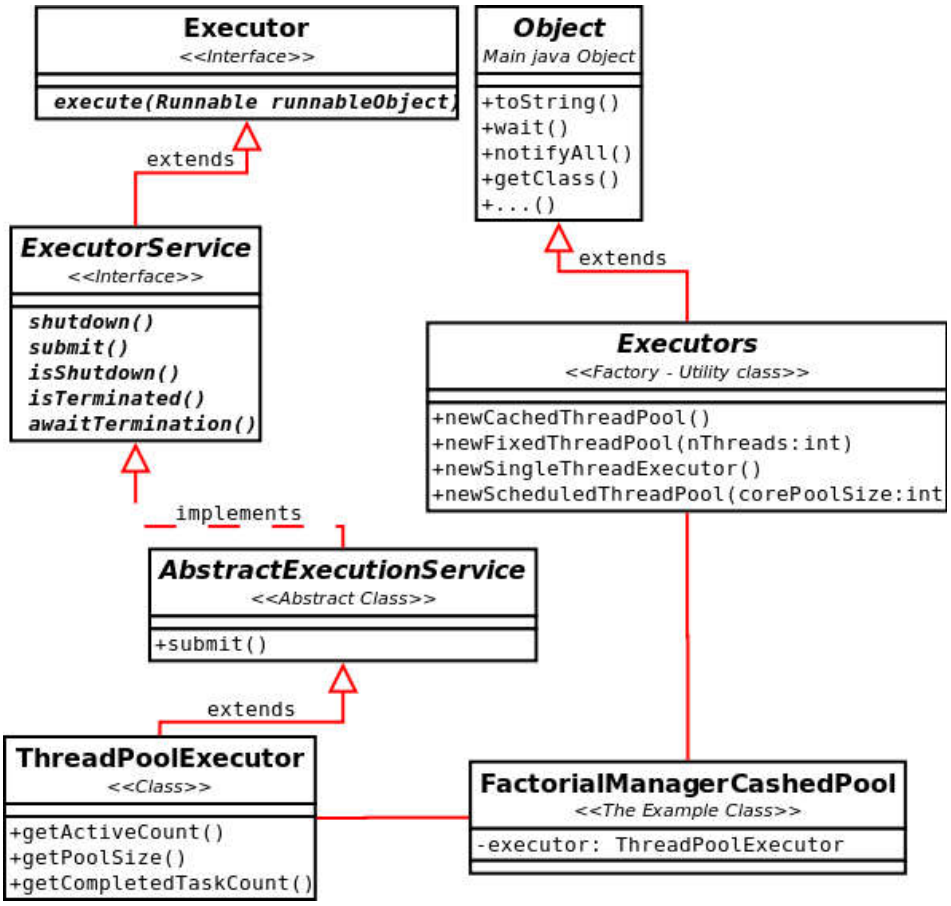
-

```
Manager: A new task has arrived
pool-1-thread-1: Task Task 0: Created on: Sat Feb 01 03:42:45 EET 2014
Manager: Pool Size: 1
Manager: Active Count: 1
pool-1-thread-1: Task Task 0: Started on: Sat Feb 01 03:42:45 EET 2014
Manager: Completed Tasks: 0
Manager: A new task has arrived
Manager: Pool Size: 2
Manager: Active Count: 2
Manager: Completed Tasks: 0
Manager: A new task has arrived
pool-1-thread-2: Task Task 1: Created on: Sat Feb 01 03:42:45 EET 2014
pool-1-thread-2: Task Task 1: Started on: Sat Feb 01 03:42:45 EET 2014
Manager: Pool Size: 3
Manager: Active Count: 3
Manager: Completed Tasks: 0
pool-1-thread-3: Task Task 2: Created on: Sat Feb 01 03:42:45 EET 2014
pool-1-thread-3: Task Task 2: Started on: Sat Feb 01 03:42:46 EET 2014
Manager: A new task has arrived
Manager: Pool Size: 4
Manager: Active Count: 4
Manager: Completed Tasks: 0
Manager: A new task has arrived
pool-1-thread-4: Task Task 3: Created on: Sat Feb 01 03:42:46 EET 2014
pool-1-thread-4: Task Task 3: Started on: Sat Feb 01 03:42:46 EET 2014
pool-1-thread-5: Task Task 4: Created on: Sat Feb 01 03:42:46 EET 2014
pool-1-thread-5: Task Task 4: Started on: Sat Feb 01 03:42:46 EET 2014
Manager: Pool Size: 5
Manager: Active Count: 5
Manager: Completed Tasks: 0
pool-1-thread-1: Task Task 0: has calculated the factorial of the number 4 which is 24
pool-1-thread-1: Task Task 0: Finished on: Sat Feb 01 03:42:49 EET 2014
pool-1-thread-3: Task Task 2: has calculated the factorial of the number 2 which is 2
pool-1-thread-3: Task Task 2: Finished on: Sat Feb 01 03:42:50 EET 2014
pool-1-thread-5: Task Task 4: has calculated the factorial of the number 2 which is 2
pool-1-thread-5: Task Task 4: Finished on: Sat Feb 01 03:42:50 EET 2014
pool-1-thread-4: Task Task 3: has calculated the factorial of the number 7 which is 5040
pool-1-thread-4: Task Task 3: Finished on: Sat Feb 01 03:42:51 EET 2014
pool-1-thread-2: Task Task 1: has calculated the factorial of the number 7 which is 5040
pool-1-thread-2: Task Task 1: Finished on: Sat Feb 01 03:42:53 EET 2014

Executor shuts down, Completed Tasks:5 time elapsed in seconds 7.162
```

-

Το κλειδί αυτού του παραπάνω παραδείγματος είναι η κλάση `ThreadPoolExecutor` που εκτελεί τα νήματα. Για να καταλάβουμε το πως λειτουργεί το παραπάνω παράδειγμα ώστε να καταλάβουμε και τα

παρακάτω παραδείγματα θα παρουσιάσουμε σε UML Class διάγραμμα τις συσχετίσεις που ισχύουν στο JDK της Java για την διασύνδεση της Executor. Ένα τμήμα από το συνολικό UML Class διάγραμμα της διασύνδεσης Executor και τις νέες κλάσεις που χρησιμοποιήσαμε για το συγκεκριμένο παράδειγμα είναι το εξής:



1. Διασύνδεση Executor: Αντικείμενα της διασύνδεσης Executor, εκτελούν Runnable αντικείμενα. Αυτή η διασύνδεση παρέχει τους μηχανισμούς που χρειάζονται για την εκτέλεση ενός νήματος χωρίς άμεσα να καλούμε την μέθοδο start() του νήματος, αλλά χρησιμοποιώντας την μέθοδο execute(). Αυτό μας επιτρέπει την διαχείριση των διαφόρων νημάτων από μία συγκεκριμένη διασύνδεση. Παράδειγμα κώδικα:

```
Executor executor = anExecutor;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
```

2. Διασύνδεση ExecutorService: η διασύνδεση ExecutorService είναι υποδιασύνδεση της Executor και επεκτείνει (extends) την διασύνδεση Executor. Το αντικείμενο ExecutorService παρέχει μεθόδους για τον τερματισμό της διαχείρισης των νημάτων από το αντικείμενο ExecutorService (shutdown, terminate, isShutdown κτλ) και παρέχει την μέθοδο submit() που είναι επέκταση της μεθόδου Executor.execute(Runnable r) που μας επιστρέφει ένα αντικείμενο τύπου Future το οποίο παρέχει τον έλεγχο της πορείας των εργασιών που θα εξετάσουμε στα επόμενα παραδείγματα.
3. Αφηρημένη κλάση AbstractExecutionService: Αποτελεί την υλοποίηση της ExecutorService για

την εκτέλεση των μεθόδων της ExecutorService.

4. Η κλάση ThreadPoolExecutor: Αντικείμενα της ThreadPoolExecutor έχουν κληρονομήσει τις κατάλληλες μεθόδους για την εκτέλεση των εργασιών με την χρήση δεξαμενής νημάτων. Στο τέλος αλλά και κατά την διάρκεια εκτέλεσης των εργασιών, τα συγκεκριμένα αντικείμενα διατηρούν μερικά βασικά στατιστικά δεδομένα για την όλη πορεία εκτέλεσης των εργασιών, όπως για παράδειγμα ο αριθμός των ολοκληρωμένων εργασιών. Η αρχικοποίηση των συγκεκριμένων αντικειμένων γίνεται με την χρήση της εργοστασιακής κλάσης Executors, είτε χειροκίνητα κάτι που απαιτεί εμπειρία και ορθή χρήση.
5. Η κλάση Executors: Η κλάση Executors είναι εργοστασιακή και βοηθητική κλάση και παρέχει τις κατάλληλες παραμέτρους για την αρχικοποίηση της ThreadPoolExecutor. Επίσης παρέχει παραμέτρους που είναι χρήσιμοι για παρόμοιες αρχικοποιήσεις και η χρήση της θα παρουσιαστεί σε επόμενα παραδείγματα.

Στο παράδειγμα δημιουργήσαμε μία δεξαμενή δεδομένων ThreadPoolExecutor, που ο τύπος δεξαμενής νημάτων είναι με την βοήθεια των παραμέτρων της κλάσης Executors τύπου CashedThreadPool. Αυτό γίνεται με την χρήση της μεθόδου newCachedThreadPool() της κλάσης Executors. Η μέθοδος επιστρέφει ένα αντικείμενο τύπου ExecutorService ώστε να συμβαδίζει με την κλάση ThreadPoolExecutor. Αλλά και για να έχει πρόσβαση σε όλες τις μεθόδους της εν λόγω κλάσης. Η CashedThreadPool έχει την ιδιότητα να χρησιμοποιεί τα υπάρχοντα νήματα στην δεξαμενή νημάτων για την ολοκλήρωση των εργασιών. Αυτό έχει ως πλεονέκτημα ότι μειώνεται ο χρόνος δημιουργίας νέων νημάτων για μία νέα εργασία προς εκτέλεση. Το μειονέκτημα της συγκεκριμένης δεξαμενής νημάτων είναι ότι έχει συγκεκριμένο αριθμό νημάτων για την εκτέλεση εργασιών και όταν έχουμε πολλές εργασίες σε μία τέτοια δεξαμενή τότε υπάρχει περίπτωση υπερφόρτωσής της. Για παράδειγμα στην αρχή της εκτέλεσης του παραπάνω παραδείγματος, όταν δημιουργείται για πρώτη φορά το αντικείμενο ThreadPoolExecutor, δεν διαθέτει κάποιο νήμα στην δεξαμενή νημάτων για εκτέλεση μιας εργασίας. Όταν έρχεται μία νέα εργασία τύπου TaskFactorial για πρώτη φορά, τότε δημιουργεί ένα νέο νήμα Runnable για την εκτέλεση της νεοεισερχόμενης εργασίας. Στο τέλος ολοκλήρωσης της εργασίας αυτής, διατηρεί το αντικείμενο Runnable για επόμενες εργασίες. Όσο όμως δεν διαθέτει τα απαραίτητα νήματα στην δεξαμενή νημάτων για την εκτέλεση νέων εργασιών, τότε είναι αναγκασμένο να δημιουργεί νέα. Για αυτό το λόγο καλό είναι να χρησιμοποιούμε την δεξαμενή CashedThreadPool μόνο όταν έχουμε ένα λογικό αριθμό εργασιών προς εκτέλεση για σύντομο χρονικό διάστημα εκτέλεσης.

Στη συγκεκριμένη δεξαμενή νημάτων, αφού δημιουργηθεί, για να εκτελέσει οι εργασίες τύπου Runnable χρησιμοποιούμε την μέθοδο execute(). Επίσης χρησιμοποιήσαμε μερικές μεθόδους εκτύπωσης στατιστικών αποτελεσμάτων κατά την διάρκεια εκτέλεσης:

- getPoolSize(): Η μέθοδος που επιστρέφει το αριθμό των νημάτων στην δεξαμενή νημάτων του Executor.
- getActiveCount(): Η μέθοδος που επιστρέφει τον αριθμό των νημάτων που εκτελούν εργασίες στην δεξαμενή νημάτων Executor.
- getCompletedTaskCount(): Η μέθοδος που επιστρέφει τον αριθμό των ολοκληρωμένων εργασιών από την δεξαμενή νημάτων Executor.
- isTerminating(): Η μέθοδος που επιστρέφει που επιστρέφει την τιμή true εάν ο Executor βρίσκεται στην διαδικασία τερματισμού. Δηλαδή μετά τη χρήση της μεθόδου shutdown() αλλά δεν έχει ολοκληρωθεί ακόμη η διαδικασία τερματισμού.

Ένα βασικό τμήμα γενικά όλων των Executor είναι ότι ρητώς πρέπει να τερματιστεί η λειτουργία τους, διαφορετικά ο Executor θα συνεχίζει να λειτουργεί και θα αναμένει για νέες προς εκτέλεση εργασίες και το κυρίως πρόγραμμα δεν θα τερματίζει την εκτέλεσή του. Για να ορίσουμε ότι πρέπει να τερματιστεί η

εκτέλεση του Executor χρησιμοποιούμε την μέθοδο shutdown(). Μετά το πέρας του τερματισμού λειτουργίας του Executor, εάν προσπαθήσουμε να στείλουμε ως όρισμα μία νέα εργασία τότε θα απορριφθεί και θα κληθεί μία εξαίρεση τύπου RejectedExecutionException.

Δημιουργία Executor με δεξαμενή νημάτων σταθερού μεγέθους

Η βασική και απλή μορφή χρήσης μίας δεξαμενής νημάτων για την εκτέλεση εργασιών γίνεται μέσω της κλάσης ThreadPoolExecutor όπου τα αντικείμενά της παραμετροποιούνται με την μέθοδο newCachedThreadPool() της βοηθητικής κλάσης Executors. Ωστόσο όπως αντιληφθήκαμε από την προηγούμενο παράδειγμα, όταν έχουμε ένα μεγάλο αριθμό από εργασίες προς εκτέλεση και η δεξαμενή νημάτων ThreadPoolExecutor μορφής newCachedThreadPool(), και η δεξαμενή δεν διαθέτει αρκετά νήματα στο εσωτερικό της, τότε είναι αναγκασμένη να δημιουργήσει νέα Runnable αντικείμενα στα οποία θα αναθέσει τις προς εκτέλεση εργασίες. Το αποτέλεσμα είναι υπερφόρτωση του υπολογιστή και χαμηλή απόδοση στην εφαρμογή.

Για να αποφύγουμε αυτό το πρόβλημα η βοηθητική κλάση Executors παρέχει μία μέθοδο για την κατασκευή σταθερού μεγέθους δεξαμενής νημάτων Executor. Ο συγκεκριμένος Executor, έχει στην δεξαμενή του ένα συγκεκριμένο αριθμό νημάτων για την εκτέλεση εργασιών και δεν δημιουργεί καινούργια νήματα όταν οι εργασίες που πρέπει να εκτελεστούν είναι περισσότερες από από τον αριθμό τον διαθέσιμων νημάτων. Το αποτέλεσμα είναι είναι εναπομείνασες εργασίες να ανασταλούν και να αναμένουν έως ότου αποδεσμευτεί ένα νήμα από την δεξαμενή νημάτων του Executor. Με αυτό το τρόπο μπορούμε να ελέγξουμε τον αριθμό νημάτων της εφαρμογής μας. Στο παρακάτω παράδειγμα πειραματιζόμαστε με το προηγούμενο παράδειγμα της παραγωγής του παραγοντικού από 5 διαφορετικές εργασίες – νήματα που τις διαχειρίζεται ένας ThreadPoolExecutor αλλά με σταθερό μέγεθος δεξαμενής ίσο με 2 εργασίες. Με βάση το προηγούμενο παράδειγμα αλλαγές επιδέχεται μόνο η TaskManager.

-

```
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

public class FactorialManagerFixedPool {

        private ThreadPoolExecutor executor;
        private long startTime;

        //executor constructor
        public FactorialManagerFixedPool() {
                executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(3);
                startTime = System.currentTimeMillis();
        }

        public void executeTasks(TaskFactorial taskFactorial) {
                System.out.printf("Manager: A new task has arrived¥n");

                // start executing its threads that is just registered
                executor.execute(taskFactorial);

                System.out.printf("Manager: Pool Size: %d¥n",
                                executor.getPoolSize());

                System.out.printf("Manager: Get Task Count: %d¥n",
```

```
                                executor.getTaskCount());

                System.out.printf("Manager: Completed Tasks: %d¥n",
                                executor.getCompletedTaskCount());
        }

        // finalize the executor thread
        public void endServer() {
                executor.shutdown();

                while (executor.isTerminating()) {/* wait until it shutdowns */
                }

                System.out.println("¥nExecutor shuts down, Completed Tasks:"
                                + executor.getCompletedTaskCount()+" time elapsed in seconds "
                                +(System.currentTimeMillis()-startTime)/1000.0);
        }
}
```

-

```
Manager: A new task has arrived
pool-1-thread-1: Task Task 0: Created on: Sat Feb 01 03:50:27 EET 2014
Manager: Pool Size: 1
Manager: Get Task Count: 1
Manager: Completed Tasks: 0
pool-1-thread-1: Task Task 0: Started on: Sat Feb 01 03:50:27 EET 2014
Manager: A new task has arrived
Manager: Pool Size: 2
Manager: Get Task Count: 2
Manager: Completed Tasks: 0
pool-1-thread-2: Task Task 1: Created on: Sat Feb 01 03:50:27 EET 2014
pool-1-thread-2: Task Task 1: Started on: Sat Feb 01 03:50:27 EET 2014
Manager: A new task has arrived
Manager: Pool Size: 3
Manager: Get Task Count: 3
Manager: Completed Tasks: 0
Manager: A new task has arrived
Manager: Pool Size: 3
Manager: Get Task Count: 4
Manager: Completed Tasks: 0
Manager: A new task has arrived
Manager: Pool Size: 3
Manager: Get Task Count: 5
Manager: Completed Tasks: 0
pool-1-thread-3: Task Task 2: Created on: Sat Feb 01 03:50:27 EET 2014
pool-1-thread-3: Task Task 2: Started on: Sat Feb 01 03:50:27 EET 2014
pool-1-thread-1: Task Task 0: has calculated the factorial of the number 8 which is 40320
pool-1-thread-1: Task Task 0: Finished on: Sat Feb 01 03:50:34 EET 2014
pool-1-thread-1: Task Task 3: Created on: Sat Feb 01 03:50:27 EET 2014
pool-1-thread-2: Task Task 1: has calculated the factorial of the number 4 which is 24
pool-1-thread-2: Task Task 1: Finished on: Sat Feb 01 03:50:34 EET 2014
pool-1-thread-2: Task Task 4: Created on: Sat Feb 01 03:50:27 EET 2014
pool-1-thread-2: Task Task 4: Started on: Sat Feb 01 03:50:34 EET 2014
pool-1-thread-1: Task Task 3: Started on: Sat Feb 01 03:50:34 EET 2014
pool-1-thread-3: Task Task 2: has calculated the factorial of the number 4 which is 24
pool-1-thread-3: Task Task 2: Finished on: Sat Feb 01 03:50:35 EET 2014
```

```
pool-1-thread-2: Task Task 4: has calculated the factorial of the number 8 which is 40320
pool-1-thread-2: Task Task 4: Finished on: Sat Feb 01 03:50:38 EET 2014
pool-1-thread-1: Task Task 3: has calculated the factorial of the number 4 which is 24
pool-1-thread-1: Task Task 3: Finished on: Sat Feb 01 03:50:40 EET 2014

Executor shuts down, Completed Tasks:5 time elapsed in seconds 13.185
```

-

Σε αυτή τη περίπτωση, έχουμε κάνει χρήση της μεθόδου `newFixedThreadPool()` της βοηθητικής – εργοστασιακής κλάσης `Executors`. Αυτή η μέθοδος δημιουργεί ένα αντικείμενο Executor με συγκεκριμένο αριθμό νημάτων στην δεξαμενή νημάτων που χρησιμοποιούνται προς εκτέλεση εργασιών. Επίτηδες παραμετροποιήσαμε την δεξαμενή νημάτων ώστε να είναι μικρότερη από τον αριθμό των εργασιών προς εκτέλεση και το αποτέλεσμα ήταν μερικές εργασίες να αναμένουν έως ότου κάποιο νήμα της δεξαμενής αποδεσμευτεί για να εκτελέσει τα εν αναμονή εργασίες. Αυτό παρουσιάζεται αν συγκρίνουμε τα αποτελέσματα αυτού και του προηγούμενου παραδείγματος. Δηλαδή και στα δύο παραδείγματα η μέθοδος `getActiveCount()` / `getTaskCount()` μας επιστρέφει τον αριθμό των εργασιών προς εκτέλεση ενώ η διαφορά φαίνεται με την μέθοδο `getPoolSize()` που ενώ στο προηγούμενο παράδειγμα συνεχώς η δεξαμενή αυξάνεται για να ικανοποιήσει την συνεχώς αυξανόμενη ζήτηση `Runnable` νημάτων για εκτέλεση εργασιών, στο παρόν παράδειγμα η δεξαμενή παραμένει σταθερή ασχέτως τον αριθμό των προς εκτέλεση εργασιών. Τέλος, χρησιμοποιήσαμε στο συγκεκριμένο παράδειγμα την μέθοδο `getTaskCount()` που μας επιστρέφει τον αριθμό των εργασιών που κατέχει ο `Executor` ασχέτως της κατάστασης που βρίσκονται.

Εκτέλεση εργασιών και διαχείριση των αποτελεσμάτων τους

Μέχρι τώρα έχουμε εξετάσει την εκτέλεση των νημάτων για την διεκπεραίωση μίας ή πολλών εργασιών. Ωστόσο τα αποτελέσματα που παράγονται μετά την ολοκλήρωση των εργασιών μένουν μόνο στα νήματα χωρίς να μπορούν να προωθηθούν για περαιτέρω επεξεργασία σε άλλα νήματα ή εργασίες. Ωστόσο ένα από τα πλεονεκτήματα του πακέτου βιβλιοθηκών `Executor` είναι η επιστροφή αποτελεσμάτων κατά την διάρκεια εκτέλεσης συντρεχουσών εργασιών. Οι υλοποιήσεις που παρέχονται είναι οι εξής:

1. `Callable`: είναι μία διασύνδεση με βασικό χαρακτηριστικό την μέθοδο `call()`. Με αυτή τη μέθοδο υλοποιούμε την λογική εκτέλεσης μιας εργασίας. Έτσι η διασύνδεση `Callable` επιδέχεται παραμετροποίηση με σκοπό να ορίζει ο χρήστης και με βάση το σκοπό της κάθε εργασίας το είδος των δεδομένων που θα επιστρέφονται.
2. `Future`: Είναι μία διασύνδεση που κατέχει κατάλληλες μεθόδους για την ανάκληση των αποτελεσμάτων που παράγονται από ένα `Callable` αντικείμενο. Επίσης έχει μεθόδους για την διαχείριση της κατάστασης και πορείας εκτέλεσης εργασιών ενός `Callable` αντικειμένου.

Στο παρακάτω παράδειγμα χρησιμοποιούμε τις παρακάτω διασυνδέσεις για να δημιουργήσουμε μία τυχαία εικόνα. Συγκεκριμένα το παρακάτω παράδειγμα δέχεται 2 βασικές παραμέτρους για την παραγόμενη εικόνα. Ο πρώτος είναι το ύψος που για χάρη ευκολίας θα είναι ίσο με το πλάτος της και η δεύτερη παράμετρός της είναι ότι μπορούμε να χωρίσουμε την εικόνα σε μικρότερα πλακίδια (`tiles`). Το αποτέλεσμα είναι ότι τα νήματα – εργασίες μπορούν να παράγουν μία λωρίδα πλακιδίων που είναι ίσο με το ακέραιο πηλίκο του ύψους της εικόνας δια των αριθμό των πλακιδίων. Επίσης το θετικό της ανάθεσης της διαδικασίας για την παραγωγή τμημάτων της εικόνας από εργασίες – νήματα σε μία δεξαμενή νημάτων `Executor` είναι ο καταμερισμός της όλης διαδικασίας. Η κλάσεις που θα παρουσιαστούν παρακάτω είναι μία `Callable` κλάση που αντιπροσωπεύει την δημιουργία των χρωμάτων των πλακιδίων της τυχαίας εικόνας που τα έχουν ανατεθεί σε κάθε εργασία, την main και την κλάση που δημιουργεί ένα πλαίσιο Frame που παρουσιάζει την τελική παραγόμενη εικόνα.

•

```
import java.util.ArrayList;
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.TimeUnit;

public class TaskRandomImage implements Callable<ArrayList<Integer>> {

        private int MaxX;
        private int MaxY;
        private int tiles;
        private Integer sleep_time;

        public RandomImageGenerator(int MaxX, int seqments, int sleep_time) {
                this.MaxX = MaxX;
                this.MaxY = this.MaxX;
                this.tiles = seqments;
                this.sleep_time = sleep_time;
        }

        @Override
        public ArrayList<Integer> call() throws Exception {

                Random random = new Random();

                ArrayList<Integer> result = new ArrayList<Integer>();

                //produce RGB random values for each tile
                for (int j = 0; j < MaxY; j += (MaxY / tiles)) {
                        int red, green, blue;
                        red = random.nextInt(255);
                        green = random.nextInt(255);
                        blue = random.nextInt(255);

                        result.add(red);
                        result.add(green);
                        result.add(blue);

                }
                //so-called sleep after hard work
                TimeUnit.MILLISECONDS.sleep(sleep_time);
                return result;
        }
}
```

```
import java.util.ArrayList;
import java.util.Random;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class MainRandomImageCreator{
```

```
//Max height and width of the random image
private static final int MaxX = 500;

//create a tiles x tiles random tiled image
private static final int tiles = 4;

public static void main(String[] args) {

        ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors
                        .newFixedThreadPool(5);

        ArrayList<Future<ArrayList<Integer>>> resultList =
                new ArrayList<Future<ArrayList<Integer>>>();

        Random random = new Random();

        for (int i = 0; i < MaxX; i+=MaxX/tiles) {
                int sleep_time = random.nextInt(10);
                //create a callable thread task that will
                //compute MaxX/tiles part of image
                RandomImageCallableTask task = new RandomImageCallableTask(MaxX,
                                tiles ,sleep_time);

                //submit it to the thread pool executor
                Future<ArrayList<Integer>> result = executor.submit(task);
                resultList.add(result);
        }

        //wait until all the tasks are finished
        do {
                for (int i = 0; i < resultList.size(); i++) {
                        Future<ArrayList<Integer>> result = resultList.get(i);
                        System.out.printf("is Task %d ready?: %s¥n"
                                        , i, result.isDone());
                }
                try {
                        TimeUnit.MILLISECONDS.sleep(1);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        } while (executor.getCompletedTaskCount() < resultList.size());


        //get all the results from the Callable objects
        System.out.printf("¥n--All Tasks are done! Main process" +
                        " the image results--¥n¥n");
        for (int i = 0; i < resultList.size(); i++) {
                try {
                        Future<ArrayList<Integer>> result = resultList.get(i);
                        ArrayList<Integer> aux_table = new ArrayList<Integer>();

                        aux_table = result.get();

                        System.out.println("RGB values created by thread "+ (i+1));
                        //get the 3 RGB colors per tile per callable thread
                        for(int j=0; j<aux_table.size(); j++) {
                                if (j%3==0)
                                        System.out.println("red:¥tgreen:¥tblue: values " +
```

```
                                                        "for tile "+i+","+ (j/3));
                                int aux = j%3;
                                switch(aux) {
                                case 0:
                                        System.out.print(""+ aux_table.get(j));
                                        break;
                                case 1:
                                        System.out.print("¥t"+ aux_table.get(j));
                                        break;
                                case 2:
                                        System.out.println("¥t"+ aux_table.get(j));
                                        break;
                                }
                        }
                        System.out.println();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                } catch (ExecutionException e) {
                        e.printStackTrace();
                }
        }

        //produce the random image in a Java Frame
        new RandomImageExecutorCreatorFrame(resultList, MaxX, tiles);

        //shut down the executor
        executor.shutdown();
    } // end of main
} // end of class
```

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.util.ArrayList;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class RandomImageExecutorCreatorFrame extends JFrame {
    private static final long serialVersionUID = 1L;

    public RandomImageExecutorCreatorFrame(
                ArrayList<Future<ArrayList<Integer>>> resultList, int MaxX,
                int tiles) {

        super("Random Image Executor Creator");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(MaxX, MaxX);
        setLocationRelativeTo(null);
        setVisible(true);
        setResizable(false);
```

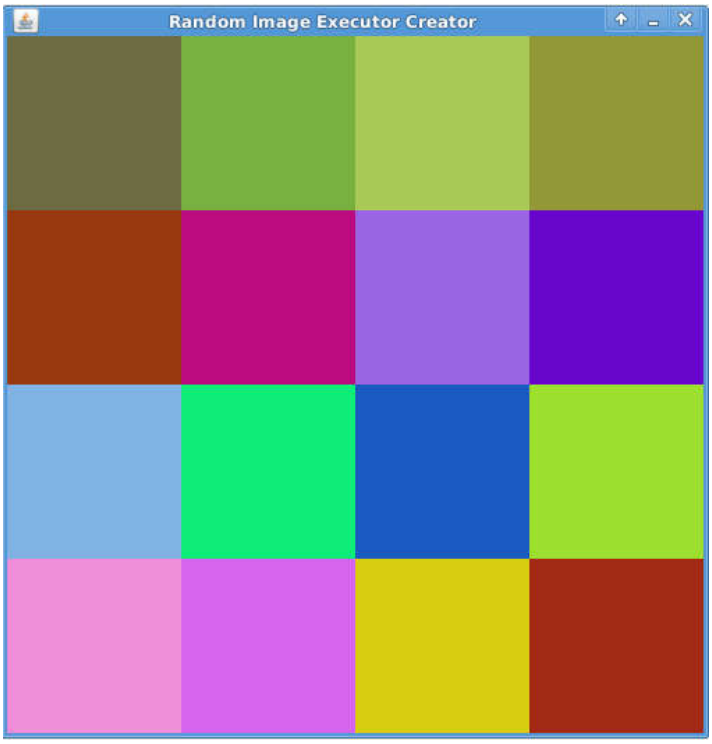
```
                //Square class the prints the tiles
                ExecutorSquares squares = new ExecutorSquares(resultList, MaxX, tiles);
                getContentPane().add(squares);

                pack();
        }
}

class ExecutorSquares extends JPanel {


        private ArrayList<Future<ArrayList<Integer>>> resultList;
        private int tiles;
        private int MaxX;
        private int MaxY;

        public ExecutorSquares(ArrayList<Future<ArrayList<Integer>>> resultList, int MaxX,
                        int segments) {

                this.resultList = resultList;
                this.MaxX = MaxX;
                this.MaxY = this.MaxX;
                this.tiles = segments;
        }

        @Override
        public Dimension getPreferredSize() {
                return new Dimension(MaxX, MaxY);
        }

        @Override
        protected void paintComponent(Graphics g) {
                super.paintComponent(g);
                Graphics2D g2 = (Graphics2D) g;

                //traverse the all the results from all the Callable Tasks
                for (int i = 0; i < resultList.size(); i++) {
                        try {
                                Future<ArrayList<Integer>> result = resultList.get(i);

                                ArrayList<Integer> aux_table = new ArrayList<Integer>();
                                aux_table = result.get();

                                int[] color_values = new int[aux_table.size()];
                                int counter = 0;
                                for (int j = 0; j < aux_table.size(); j++) {
                                        color_values[counter] =(int) aux_table.get(j);
                                        counter++;
                                }
                                int step = 0;
                                for (int j = 0; j < MaxY; j += (MaxY / tiles)) {

                                        //then start printing the RGB color per tile
                                        Color color = new Color(
                                                        color_values[step], color_values[(step+1)],
                                                        color_values[(step+2)]);
                                        g2.setColor(color);
                                        g2.fillRect(
                                                        i*(MaxX/tiles), j, MaxY / tiles,
```

```
                                                MaxY / tiles);
                                        step +=3;
                                }
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        } catch (ExecutionException e) {
                                e.printStackTrace();
                        }
                }
        }
}
```

-

```
is Task 0 ready?: true
is Task 1 ready?: true
is Task 2 ready?: true
is Task 3 ready?: true

--All Tasks are done! Main process the image results--

RGB values created by thread 1
red:    green:  blue: values for tile 0,0
109     107     65
red:    green:  blue: values for tile 0,1
153     57      18
red:    green:  blue: values for tile 0,2
129     180     228
red:    green:  blue: values for tile 0,3
239     142     217

RGB values created by thread 2
red:    green:  blue: values for tile 1,0
120     177     64
red:    green:  blue: values for tile 1,1
187     11      127
red:    green:  blue: values for tile 1,2
13      237     119
red:    green:  blue: values for tile 1,3
214     101     238

RGB values created by thread 3
red:    green:  blue: values for tile 2,0
170     202     86
red:    green:  blue: values for tile 2,1
154     101     228
red:    green:  blue: values for tile 2,2
26      90      195
red:    green:  blue: values for tile 2,3
217     205     17

RGB values created by thread 4
red:    green:  blue: values for tile 3,0
148     153     56
red:    green:  blue: values for tile 3,1
105     6       205
red:    green:  blue: values for tile 3,2
158     224     47
```

```
red:    green:  blue: values for tile 3,3
163     42      21
```



-

Σε αυτό το παράδειγμα μαθαίνουμε πως να χρησιμοποιούμε την `Callable` διασύνδεση για την εκτέλεση συντρεχουσών εργασιών τα οποία επιστρέφουν και ένα αποτέλεσμα. Αρχικά η κλάση `RandomImageCallableTask` παράγει τα RGB χρώματα των πλακιδίων των τμημάτων της εικόνας που της ανατίθενται (τα τμήματα της εικόνας είναι ίσα με `MaxX/tiles`). Στη συνέχεια η κλάση επιστρέφει ένα πίνακα με αυτές τις τιμές με την χρήση της μεθόδου `call()`. Από τη πλευρά της κλάσης Main οι εργασίες ανατίθενται προς εκτέλεση μέσω μίας δεξαμενής νημάτων `Executor` με την μέθοδο `submit()` του `Executor`. Η μέθοδος αυτή δέχεται ένα `Callable` αντικείμενο ως παράμετρο και επιστρέφει ένα `Future` αντικείμενο το οποίο το χρησιμοποιούμε ως εξής:

1. Μπορούμε να ελέγχουμε τη κατάσταση κάθε εργασίας, δηλαδή ελέγχουμε την κατάσταση πορείας εκτέλεσης της κάθε εργασίας με τα αντικείμενα τύπου `Future`, ώστε η Main μέθοδος να αναμένει όλα τα αποτελέσματα των εργασιών τους. Αυτό διεκπεραιώνεται με την χρήση της μεθόδου `isDone`.

2. Μπορούμε να προσπελάσουμε τα δεδομένα που επιστρέφονται από κάθε εργασία από την μέθοδο `call()` με την χρήση της μεθόδου `get()` των `Future` αντικειμένων. Η μέθοδος αυτή αναμένει έως ότου ένα `Callable` αντικείμενο έχει ολοκληρώσει την εκτέλεση της μεθόδου `call()` και έχει επιστρέψει τα αποτελέσματά του. Εάν το νήμα – εργασία έχει διακοπεί ενώ η μέθοδος `get()` αναμένει για τα αποτελέσματα, τότε δημιουργείται μία εξαίρεση. τύπου `InterruptedException`. Εάν η μέθοδος `call()` διακοπεί τότε η μέθοδος αυτή δημιουργεί μία εξαίρεση τύπου `ExecutionException`

Μπορούμε να καλέσουμε την μέθοδο `get()` ενός `Future` αντικειμένου ενώ η εργασία που ελέγχει αυτό το αντικείμενο δεν έχει ακόμη ολοκληρωθεί. Τότε αναγκαστικά αυτή η μέθοδος `get()` αναστέλλει τη λειτουργία της έως ότου ολοκληρωθεί η λειτουργία της εργασίας. Η διασύνδεση `Future` παρέχει επίσης μία ακόμη έκδοση της μεθόδου `get()`.

- Η μέθοδος `get(long timeout, TimeUnit unit)`: Αυτή η έκδοση της μεθόδου `get` αναμένει τα αποτελέσματά του αντικειμένου `Future` (αν δεν είναι έτοιμα) στον καθορισμένο χρόνο αναμονής.

Αν στη συγκεκριμένη χρονική περίοδο ακόμη τα αποτελέσματα δεν είναι έτοιμα τότε η μέθοδος μας επιστρέφει την τιμή `null`.

Εκτέλεση εργασιών μετά από χρονική καθυστέρηση

Η δομή `Executor` παρέχει την κλάση `ThreadPoolExecutor` για την εκτέλεση `Callable` και `Runnable` εργασιών μέσα σε μία δεξαμενή νημάτων με την ευκολία της αποφυγής δημιουργίας νημάτων χειρωνακτικά. Ωστόσο υπάρχουν περιπτώσεις που δεν χρειάζεται (ή δεν πρέπει) να εκτελέσουμε μία εργασία αμέσως. Δηλαδή μπορούμε να εκτελέσουμε μία εργασία έπειτα από μία χρονική περίοδο είτε να εκτελέσουμε μία εργασία περιοδικά. Για αυτό το λόγο το πακέτο δομών `Executor` παρέχει την κλάση `ScheduledThreadPoolExecutor`.

Στο παρακάτω παράδειγμα παρουσιάζεται το πως να δημιουργούμε ένα `ScheduledThreadPoolExecutor` και πως να το χρησιμοποιούμε για την χρονικά προγραμματισμένη εκτέλεση μιας εργασίας. Συγκεκριμένα προσομοιώνουμε την λειτουργία του ρολογιού όπου κάθε εργασία εκτελείται η μία μετά την άλλη με καθυστέρηση από την προηγούμενη κατά ένα δευτερόλεπτο.

-

```
import java.util.Date;
import java.util.concurrent.Callable;

public class TaskClockDelay implements Callable<String> {

	@Override
	public String call() throws Exception {
		System.out.printf("%s¥n", new Date());
		return "";
	}
}
```

```
import java.util.Date;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class MainClockDelay {

	public static void main(String[] args) {

		int hours = 24;
		int minutes = 60;
		int seconds = 60;

		/*
```

```
                * Create an executor of the ScheduledThreadPoolExecutor class using the
                * newScheduledThreadPool() method of the Executors utility factory
                * class passing 1 as a parameter.
                */
               ScheduledThreadPoolExecutor executor = (ScheduledThreadPoolExecutor) Executors
                               .newScheduledThreadPool(1);

               int duration = hours * minutes * seconds;

               /*
                * Initialize and start a few tasks with the schedule() method of the
                * ScheduledThreadPoolExecutor instance.
                */
               for (int i = 0; i < duration; i++) {
                       TaskClockDelay taskClockDelay = new TaskClockDelay();
                       executor.schedule(taskClockDelay, i + 1, TimeUnit.SECONDS);
               }

               // executor shutdown
               executor.shutdown();

               /*
                * Wait for the finalization of all the tasks using the
                * awaitTermination() method of the executor.
                */
               try {
                       executor.awaitTermination(1, TimeUnit.DAYS);
               } catch (InterruptedException e) {
                       e.printStackTrace();
               }

               // Write a message to indicate the time when the program finishes.
               System.out.printf("Main: Clock Ends at: %s¥n", new Date());

       } //end of main
}
```

-

```
Sun Apr 27 22:24:44 EEST 2014
Sun Apr 27 22:24:45 EEST 2014
Sun Apr 27 22:24:46 EEST 2014
Sun Apr 27 22:24:47 EEST 2014
Sun Apr 27 22:24:48 EEST 2014
Sun Apr 27 22:24:49 EEST 2014
Sun Apr 27 22:24:50 EEST 2014
```

-

Το βασικό χαρακτηριστικό του παραπάνω παραδείγματος είναι η κλάση `Main` και η διαχείριση του `ScheduledThreadPoolExecutor`. Όπως και στα προηγούμενα παραδείγματα χρησιμοποιούμε την πακέτο βοηθητικών κλάσεων `Executors`. Ως όρισμα δώσαμε το 1 που ορίζει ότι ο αριθμός των νημάτων στην δεξαμενή θα είναι 1. Για να εκτελέσουμε μια εργασία σε ένα `scheduled executor` μετά από κάποια ορισμένη χρονική στιγμή, χρησιμοποιούμε την μέθοδο `schedule()`. Αυτή η μέθοδος χρησιμοποιεί τα παρακάτω ορίσματα:

1. Την εργασία που θα εκτελέσουμε

2. Την χρονική περίοδο που θα αναμένει η εργασία έως ότου αρχίσει να εκτελεστεί.
3. Τον τύπου της χρονικής περιόδου (λεπτά δευτερόλεπτα κτλ) με την βοήθεια της κλάσης `TimeUnit`.

Επίσης μπορούμε να χρησιμοποιήσουμε Runnable αντικείμενα για να δημιουργήσουμε τις εργασίες που θα εκτελεστούν με μία χρονική καθυστέρηση.

Εκτέλεση εργασιών περιοδικά (σε τακτά χρονικά διαστήματα)

Το πακέτο δομών `Executor` που παρέχει την κλάση `ThreadPoolExecutor` για την εκτέλεση συγχρονισμένων εργασιών χρησιμοποιώντας μία δεξαμενή νημάτων ώστε να αποφεύγομαι τη χειροκίνητη έναρξη και τερματισμό του κάθε νήματος ξεχωριστά. Όταν μια εργασία τερματιστεί τότε η εργασία αυτή διαγράφεται από τη δεξαμενή εργασιών. Ωστόσο εάν θέλουμε να εκτελέσουμε την ίδια ακριβώς εργασία ξανά σε τακτά χρονικά διαστήματα, πρέπει να την στείλουμε ξανά στον `Executor`. Για αυτό το λόγο το πακέτο δομών `Executor` παρέχει την δυνατότητα να εκτελούμε περιοδικά εργασίες διαμέσου της κλάσης `ScheduledThreadPoolExecutor`.

Στο παρακάτω παράδειγμα παρουσιάζεται ο τρόπος χρήσης περιοδικότητας των εργασιών. Συγκεκριμένα ανά 4 δευτερόλεπτα υπολογίζουμε το pi του κύκλου με την μέθοδο Monte Carlo και εκτυπώνουμε στην οθόνη το ανάλογο αποτέλεσμα, το πρότυπο pi, και την διαφορά τους.

-

```
import java.util.Date;
import java.util.Random;

/*
 * Computing the pi with monte carlo simulation
 * http://www.eveandersson.com/pi/monte-carlo-circle
 */

public class TaskMonteCarloPi implements Runnable {

	private String name;

	public TaskMonteCarloPi(String name) {
		this.name = name;
	}

	@Override
	public void run() {
		double x, y;
		int range = 2;
		int lowerBound = -1;
		Random random = new Random(System.currentTimeMillis());
		int hits = 0, limit = 100000;

		for (int i = 0; i < limit; i++) {

			// get a random point and its x,y coordinates
			x = random.nextDouble() * range + lowerBound;
			y = random.nextDouble() * range + lowerBound;

			// calculate the distance between this point and the circle's center
```

```
                        // if its in the circle increment hits
                        if (Math.pow(x, 2) + Math.pow(y, 2) < 1)
                                hits++;
                }

                System.out.printf("%s: Starting at : %s¥n", name, new Date());
                System.out
                                .printf("Monte Carlo computed Pi: %f¥n", (4.0 * hits / limit));
                System.out.printf("Monte Carlo original Pi: %f¥n", Math.PI);
                System.out.printf("Monte Carlo Pi diffehttp://www.it.uom.gr/rence: %f¥n¥n",
                                (4.0 * hits / limit) - Math.PI);
        }
}
```

```
import java.util.Date;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;

public class MainPeriodicalMonteCarloPi {

        public static void main(String[] args) {

                /*
                 * Create ScheduledThreadPoolExecutor using the newScheduledThreadPool()
                 * method of the Executors class. Pass the number 1 as the parameter to
                 * that method.
                 */
                ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

                System.out.printf("Main: Starting at: %s¥n", new Date());

                // Create a new Task object.
                TaskMonteCarloPi taskMonteCarloPi = new TaskMonteCarloPi("Task");

                /*
                 * Send it to the executor using the scheduledAtFixRate() method. Use as
                 * parameters the task created earlier, the number 1 is the initial
                 * waiting period, the number 4 is the extra waiting period for the next
                 * task to start, and the constant TimeUnit.SECONDS. This method returns
                 * a ScheduledFuture object that you can use to control the status of
                 * the task.
                 */
                ScheduledFuture<?> result = executor.scheduleAtFixedRate(
                                taskMonteCarloPi, 1, 4, TimeUnit.SECONDS);

                /*
                 * Create a loop to write the time remaining for the next execution of
                 * the task. In the loop, use the getDelay() method of the
                 * ScheduledFuture object to get the number of milliseconds until the
                 * next execution of the task.
                 */
                for (int i = 0; i < 100; i++) {
                        System.out.printf("Main: Delay for Task: %d milliseconds left¥n", result
                                        .getDelay(TimeUnit.MILLISECONDS));
                        // Sleep the Main thread during 500 milliseconds.
```

```
                        try {
                                TimeUnit.MILLISECONDS.sleep(500);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }

                // Finish the executor using the shutdown() method.
                executor.shutdown();

                try {
                        TimeUnit.SECONDS.sleep(5);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }

                System.out.printf("Main: Finished at: %s¥n", new Date());
        }
}
```

-

```
Main: Starting at: Sun Apr 27 23:05:15 EEST 2014
Main: Delay for Task: 997 milliseconds left
Main: Delay for Task: 495 milliseconds left
Main: Delay for Task: -5 milliseconds left
Task: Starting at : Sun Apr 27 23:05:16 EEST 2014
Monte Carlo computed Pi: 3,144440
Monte Carlo original Pi: 3,141593
Monte Carlo Pi difference: 0,002847

Main: Delay for Task: 3476 milliseconds left
Main: Delay for Task: 2975 milliseconds left
Main: Delay for Task: 2473 milliseconds left
Main: Delay for Task: 1973 milliseconds left
Main: Delay for Task: 1471 milliseconds left
Main: Delay for Task: 970 milliseconds left
Main: Delay for Task: 468 milliseconds left
Main: Delay for Task: -32 milliseconds left
Task: Starting at : Sun Apr 27 23:05:20 EEST 2014
Monte Carlo computed Pi: 3,134600
Monte Carlo original Pi: 3,141593
Monte Carlo Pi difference: -0,006993
```

-

Όταν θέλουμε να εκτελέσουμε μία περιοδική εργασία, χρειαζόμαστε ένα αντικείμενο `ScheduledExecutorService`. Επίσης για την αρχικοποίησης του συγκεκριμένου `Executor` χρησιμοποιούμε την βοηθητική λειτουργική κλάση `Executors`. Στη συγκεκριμένη περίπτωση χρησιμοποιούμε την μέθοδο `newScheduledThreadPool()` για να δημιουργήσουμε ένα `ScheduledExecutorService` αντικείμενο. Η μέθοδος δέχεται ως παράμετρο τον αριθμό των νημάτων για την δεξαμενή νημάτων. Επειδή έχουμε μόνο μία εργασία για αυτό το λόγο ως παράμετρο δίνουμε το 1.

Όταν έχουμε επιλέξει τον κατάλληλο `executor` για την περιοδική εργασία, έπειτα δίνουμε ως παράμετρο την εργασία στον `executor` στην μέθοδο `scheduledAtFixedRate()`. Η μέθοδος αυτή δέχεται 4 παραμέτρους: Την προς περιοδική εκτέλεση εργασία, τον χρόνο καθυστέρησης για την έναρξη εκτέλεσης για πρώτη φορά της εργασίας, τον χρονική διάρκεια μεταξύ των εκτελέσεων της εργασίας και την χρονική

μονάδα μέτρησης χρησιμοποιώντας την κλάση TimeUnit.

Η μέθοδος scheduledAtFixedRate() μας επιστρέφει ένα αντικείμενο ScheduledFuture, το οποίο επεκτείνει την διασύνδεση Future, με μεθόδους για τις scheduled εργασίες. Η διασύνδεση ScheduledFuture είναι μία παραμετροποιημένη διασύνδεση. Δηλαδή στο συγκεκριμένο παράδειγμα, ενώ η εργασία μας είναι ένα Runnable αντικείμενο, για να γίνει δεκτό από τΗ ScheduledFuture διασύνδεση ώστε να γνωρίζει τι είδους αντικείμενα θα επιστρέφει, την παραμετροποιούμε με το σύμβολο "?".

Κάναμε χρήση της μεθόδου getDelay() της διασύνδεσης ScheduledFuture. Η μέθοδος αυτή επιστρέφει το χρόνο που απομένει μέχρι την επόμενη εκτέλεσης μιας εργασίας. Η μέθοδος αυτή επιστρέφει μία σταθερά τύπου TimeUnit με βάση το τύπου του TimeUnit που θέλουμε να τροποποιηθεί το αποτέλεσμα της μεθόδου.

Μπορούμε επίσης να μεταβάλλουμε τον τρόπο λειτουργίας της μεθόδου shutdown() του στιγμιότυπου του ScheduledThreadPoolExecutor. Η προεπιλεγμένη συμπεριφορά είναι ότι οι scheduled εργασίες τερματίζονται αμέσως όταν καλούμε την μέθοδο shutdown(). Ωστόσο μπορούμε να το αλλάξουμε αυτό κάνοντας χρήση της μεθόδου setContinueExistingPeriodicTasksAfterShutdownPolicy() της κλάσης ScheduledThreadPoolExecutor με τιμή true. Έτσι οι περιοδικές εργασίες δεν θα τερματίζονται μετά από το κάλεσμα της μεθόδου shutdown().

Ακύρωση της εκτέλεσης μιας εργασίας σε έναν executor

Όσο εργαζόμαστε με ένα ένα executor, δεν χρειάζεται να διαχειριζόμαστε χειροκίνητα τα νήματα της δεξαμενής νημάτων. Ο executor είναι υπεύθυνος για την δημιουργία, διαχείριση και τερματισμό τους. Μερικές φορές, ίσως θέλουμε να ακυρώσουμε την εκτέλεση μιας εργασίας και να την τερματίσουμε πρόωρα. Για αυτή τη περίπτωση χρησιμοποιούμε την μέθοδο cancel() της διασύνδεσης Future που μας επιτρέπει να ακυρώνουμε μια εργασία.

Στο παρακάτω παράδειγμα παρουσιάζεται ο τρόπος που μπορούμε να ακυρώνουμε εργασίες μέσα σε ένα executor. Η προς εκτέλεση εργασία υπολογίζει το pi του κύκλου με τη μέθοδο Markov Chain Monte Carlo.

-

```
import java.util.Date;
import java.util.Random;
import java.util.concurrent.Callable;

/*
 * A class that simulates the Markov Chain Monte Carlo
 * pi calculation
 * for more information
 * http://math.stackexchange.com/questions/210653/confusion-related-to-the-calculation-of-value-of-pi
 */
public class TaskMarkovPi implements Callable<String> {

        private String name;

        public TaskMarkovPi(String name) {
                this.name = name;
        }
```

```
        @Override
        public String call() throws Exception {

                while (true) {
                        // get a random point's coordinates
                        double x = 1.0;
                        double y = 1.0;
                        double delta_x, delta_y;

                        // the step to take its time to a new point
                        double delta = 0.3;
                        double range = delta * 2;

                        Random random = new Random(System.currentTimeMillis());
                        double hits = 0;
                        int limit = 100000;

                        for (int i = 0; i < limit; i++) {
                                // get a new displacement for the x,y coordinates of the initial
                                // point
                                delta_x = random.nextDouble() * range - delta;
                                delta_y = random.nextDouble() * range - delta;

                                // if the new point is in the boundaries then its acceptable
                                if ((Math.abs(x + delta_x) < 1) & (Math.abs(y + delta_y) < 1)) {
                                        x = x + delta_x;
                                        y = y + delta_y;
                                }

                                // if the distance between this point and the center of the
                                // circle 0(0,0)
                                // is smaller than the radius then this point is in the circle,
                                // then
                                // increment the hits variable
                                if (Math.pow(x, 2) + Math.pow(y, 2) < 1)
                                        hits++;
                        }
                        System.out.printf("%s: Starting at : %s¥n", name, new Date());
                        System.out.printf("Markov computed Pi: %f¥n", (4.0 * hits / limit));
                        System.out.printf("Markov original Pi: %f¥n", Math.PI);
                        System.out.printf("Markov Pi difference: %f¥n",
                                        (4.0 * hits / limit) - Math.PI);
                        System.out.println("_____");
                        System.out.println("Task " + this.getClass().getSimpleName()
                                        + " is waiting for 1.5 seconds¥n");
                        Thread.sleep(1500);

                }
        }

}
```

```
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
```

```
public class MainCancelMarkovPi {

	public static void main(String[] args) {

		/*
		 * Create a ThreadPoolExecutor object using the newCachedThreadPool()
		 * method of the Executors class.
		 */
		ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors
				.newCachedThreadPool();

		// Create a new Task object.
		TaskMarkovPi taskMarkovPi = new TaskMarkovPi("task");

		int executionTime = 4;
		// Send the task to the executor using the submit() method.
		System.out.printf("Main: Executing the Task for %d seconds¥n", executionTime);
		Future<String> result = executor.submit(taskMarkovPi);

		// Put the main task to sleep for 20 seconds.
		try {
			TimeUnit.SECONDS.sleep(executionTime);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}

		/*
		 * Cancel the execution of the task using the cancel() method of the
		 * Future object named result returned by the submit() method. Pass the
		 * true value as a parameter of the cancel() method.
		 */
		System.out.printf("Main: Canceling the Task¥n");
		result.cancel(true);

		System.out.printf("Main: Canceled: %s¥n", result.isCancelled());
		System.out.printf("Main: Done: %s¥n", result.isDone());

		//Finish the executor with the shutdown() method
		executor.shutdown();
		System.out.printf("Main: The executor has finished¥n");

	}
}
```

-

```
Main: Executing the Task for 4 seconds
task: Starting at : Mon Apr 28 05:48:48 EEST 2014
Markov computed Pi: 3,130600
Markov original Pi: 3,141593
Markov Pi difference: -0,010993
-----------------------------------
Task TaskMarkovPi is waiting for 1.5 seconds

task: Starting at : Mon Apr 28 05:48:49 EEST 2014
Markov computed Pi: 3,151440
```

```
Markov original Pi: 3,141593
Markov Pi difference: 0,009847

------------------------------------
Task TaskMarkovPi is waiting for 1.5 seconds

task: Starting at : Mon Apr 28 05:48:51 EEST 2014
Markov computed Pi: 3,116800
Markov original Pi: 3,141593
Markov Pi difference: -0,024793

------------------------------------
Task TaskMarkovPi is waiting for 1.5 seconds

Main: Canceling the Task
Main: Canceled: true
Main: Done: true
Main: The executor has finished
```

-

Χρησιμοποιούμε την μέθοδο `cancel()` της διασύνδεσης `Future` όταν θέλουμε να ακυρώσουμε μία εργασία που βρίσκεται μέσα σε ένα `executor`. Ανάλογα με την παράμετρο της της μεθόδους `cancel()` και την κατάσταση της εργασίας, η συμπεριφορά της μεθόδου τότε αναλόγως μεταβάλλεται:

- Εάν μια εργασία που έχει ολοκληρωθεί είτε δεν μπορεί να ακυρωθεί τότε η μέθοδος μας επιστρέφει την τιμή `false`.
- Εάν η εργασία είναι σε κατάσταση αναμονής με σκοπό να λάβει ένα αντικείμενο τύπου `Thread` για να αρχίσει την εκτέλεσή της, τότε η εργασία ακυρώνεται από την μέθοδο `cancel()`. Εάν η εργασία είναι ήδη σε εκτέλεση, τότε εάν έχει οριστεί η μέθοδος `cancel()` με την τιμή `true`, τότε η εργασία θα ακυρωθεί. Εάν η τιμή της παραπάνω παραμέτρου είναι `false` και η εργασία είναι σε εκτέλεση τότε δεν θα ακυρωθεί.

Διαχωρισμός των εργασιών και επεξεργασία των αποτελεσμάτων τους στον Executor

Συνήθως, όταν εκτελούμε συγχρονισμένες εργασίες με χρήση ενός `Executor`, στέλνουμε `Runnable` ή `Callable` εργασίες σε ένα `Executor` και λαμβάνουμε `Future` αντικείμενα. Ωστόσο μπορούμε να βρεθούμε σε καταστάσεις όπου χρειαζόμαστε να στείλουμε εργασίες σε ένα executor σε ένα αντικείμενο - κλάση και θέλουμε να επεξεργαστούμε έπειτα τα αποτελέσματα της εργασίας σε ένα διαφορετικό προς εκτέλεση αντικείμενο - κλάση. Για αυτές τις περιπτώσεις το πακέτο `Java Concurrency API` παρέχει την κλάση `CompletionService` class.

Η κλάση `CompletionService` παρέχει μία μέθοδο που στέλνει τις εργασίες σε ένα εκτελεστή και μία ακόμη διαφορετική μέθοδο ώστε να λαμβάνει τα `Future` αντικείμενα για την κάθε επόμενη εργασία που έχει ολοκληρώσει την εκτέλεσή της.

Η παραπάνω κλάση χρησιμοποιεί ένα `Executor` αντικείμενο για την εκτέλεση των εργασιών. Όμως το πλεονέκτημα αυτής της κλάσης είναι ότι η κλάση `CompletionService` είναι μία μοιραζόμενη κλάση όπως και τα αντικείμενά της, έτσι λοιπόν ενώ στέλνει εργασίες σε ένα `Executor` για την εκτέλεσή τους, οι υπόλοιπες εργασίες που έχουν ήδη ολοκληρώσει την εκτέλεσή τους μπορεί να προχωρούν στην επεξεργασία των αποτελεσμάτων τους. Ο περιορισμός όμως που ισχύει για την `CompletionService` είναι ότι μπορεί να δεχθεί μόνο `Future` αντικείμενα για εργασίες που έχουν ολοκληρώσει την εκτέλεσή τους. Έτσι τα `Future` αντικείμενα μπορούν να χρησιμοποιηθούν για την ανάκτηση και περαιτέρω επεξεργασία των αποτελεσμάτων τους.

Στο παρακάτω παράδειγμα θα μάθουμε για την χρήση της κλάσης CompletionService για να διαχωρίζουμε τις εργασίες του Executor που έχουν ήδη ολοκληρώσει την εκτέλεσή τους από εκείνες που βρίσκεται ήδη σε εκτέλεση. Συγκεκριμένα υπολογίζουμε το pi του κύκλου με μια μέθοδο τύπου Monte Carlo (Buffon Needle Problem). Πιο συγκεκριμένα η κλάση main ζητάει από μία Request κλάση να δημιουργήσει ένα Task (εργασία). Η συγκεκριμένη εργασία εκτελείται και ταυτόχρονα εκτελείται μία κλάση PostProcessor η οποία αναμένει τις ολοκληρωμένες εργασίες και επεξεργάζεται τα αποτελέσματά τους.

-

```
import java.util.concurrent.CompletionService;

public class NeedlePiTaskRequest implements Runnable {

	private String name;
	private CompletionService<Double> service;

	// Implement the constructor of the class that initializes the two
	// attributes.
	public NeedlePiTaskRequest(String name, CompletionService<Double> service) {
		this.name = name;
		this.service = service;
	}

	@Override
	public void run() {
		/*
		 * Implement the run() method. Create a needlePiGenerator object and
		 * send them to the CompletionService object using the submit() method.
		 */
		TaskNeedlePiGenerator taskNeedlePiGenerator = new TaskNeedlePiGenerator(name,
				"Report");
		service.submit(taskNeedlePiGenerator);
	}

}
```

```
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.TimeUnit;

public class TaskNeedlePiGenerator implements Callable<Double> {

	private String sender;
	private String title;

	// the distance between 2 woods in a parquette floor
	private double parquetteDistance = 1;

	// the niddle length
	private double needleLength = 1;

	private Random myGenerator;
	private int myHits = 0;
```

```
	private int myTries = 0;
	private int tries = 10000;
	private double threshold = 1e-2;
	private double best_approx_pi = 0;

	public TaskNeedlePiGenerator(String sender, String title) {
		this.myGenerator = new Random();
		this.sender = sender;
		this.title = title;
	}

	/*
	 * Implement the call() method. First, put the thread to sleep for a random
	 * period of time.
	 */
	@Override
	public Double call() throws Exception {
		Random random = new Random();
		try {
			Long duration = (long) (1 + random.nextInt(8));
			System.out
					.printf(
							"%s_%s: NeedlePiGenerator: Generating" +
							" a report during %d seconds¥n",
							this.sender, this.title, duration);
			TimeUnit.SECONDS.sleep(duration);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		// then generate a needle pi calculation
		return drop();
	}

	/**
	 * Drops a needle between 2 lines distance "parquettedDistance" measures the
	 * distance between the 2 needle's tips measures the distance between the
	 * needle's tip and a parquett line counts whether the needle hits the
	 * parquett line in order to compute pi
	 */
	public double drop() {

		// check the new pi approximation if is better than the previous one
		for (int i = 0; i < tries; i++) {

			// compute a random start point for a needdle
			double x_start_tip = parquetteDistance * myGenerator.nextDouble();

			// compute a random angle between the needle's tips
			double angle = 90 * myGenerator.nextDouble();

			// compute the needle's end tip vertical distance from the other tip
			double x_end_tip_distance = needleLength
					* Math.sin(Math.toRadians(angle));

			// compute the distance of a needle to the parquette's edge
			double distance = parquetteDistance - x_start_tip;

			// if there is a hit then
			if (distance <= x_end_tip_distance)
```

```
                        myHits++;
                    myTries++;

                    double new_temp_pi = (2.0 * myTries * needleLength)
                                    / (myHits * parquetteDistance);

                    if (Math.abs((Math.PI - new_temp_pi)) <= threshold)
                        best_approx_pi = new_temp_pi;

                }
                return best_approx_pi;
        }

        /**
         * Gets the number of times the needle hit a line.
         */
        public int getHits() {
                return myHits;
        }

        /**
         * Gets the total number of times the needle was dropped.
         */
        public int getTries() {
                return myTries;
        }
}
```

```
import java.util.Date;
import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class NeedlePiPostProcessor implements Runnable {
        // This class will get the results of the NeedlePiGenerator tasks.

        private CompletionService<Double> service;
        private boolean end;

        private double threshold = 1e-2;
        private double best_approx_pi = 0;

        public NeedlePiPostProcessor(CompletionService<Double> service) {
                this.service = service;
                end = false;
        }

        @Override
        /*
         * While the attribute end is false, call the poll() method of the
         * CompletionService interface to get the Future object of the next task
         * executed by the completion service that has finished.
         */
        public void run() {
```

```
                int awaitTime = 20;
                System.out
                                .println("The post processor task will wait " +
                                                "for tasks to finish for "
                                                + awaitTime + " seconds");
                while (!end) {
                        try {
                                Future<Double> result = service.poll(awaitTime,
                                                TimeUnit.SECONDS);

                                if (result != null) {
                                        /*
                                         * get the results of the task using
                                         * the get() method of the
                                         * Future object and write those results to the console.
                                         */
                                        double report = result.get();

                                        System.out.printf(
                                                        "PostProcessor: " +
                                                        "%s A new Needle Pi Received: %s¥n",
                                                        new Date(), report);

                                        if (Math.abs((Math.PI - report)) <= threshold)
                                                best_approx_pi = report;

                                }
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        } catch (ExecutionException e) {
                                e.printStackTrace();
                        }
                }
                System.out.printf("Best Pi approximation: %.10f¥n", best_approx_pi);
                System.out.printf("PostProcessor: End¥n");
        }

        public void setEnd(boolean end) {
                this.end = end;
        }
}
```

```
import java.util.Date;
import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class MainNeedlePi {

        public static void main(String[] args) {

                /*
                 * Create ThreadPoolExecutor using the newCachedThreadPool() method of
                 * the Executors class.
                 */
                ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors
```

```
                .newCachedThreadPool();

        /*
         * Create CompletionService using the executor created earlier as a
         * parameter of the constructor.
         */
        CompletionService<Double> service = new ExecutorCompletionService<Double>(
                executor);

        /* create needle Pi simulation tasks */

        //the number of tasks
        int length = 6;

        //create the needle pi task requests in order to compute the needle pi
        NeedlePiTaskRequest[] NeedleTester = new NeedlePiTaskRequest[length];
        for (int i = 0; i < length; i++) {
                NeedleTester[i] = new NeedlePiTaskRequest("NeedleTester" + i,
                                service);
        }

        //create the thread tasks to perform the execution
        Thread[] NeedleTesterThread = new Thread[length];
        for (int i = 0; i < length; i++) {
                NeedleTesterThread[i] = new Thread(NeedleTester[i]);
        }

        // create a task for post process
        NeedlePiPostProcessor PostProcessor = new NeedlePiPostProcessor(service);
        Thread PostProcessorThread = new Thread(PostProcessor);

        // start the task threads
        System.out
                        .printf(
                                        "Main: At %s, Starting the Needle Pi " +
                                        "simulation tasks Threads¥n",
                                        new Date());

        for (int i = 0; i < length; i++) {
                NeedleTesterThread[i].start();
        }

        PostProcessorThread.start();

        // waiting for the tasks to finish by joining them
        try {
                System.out
                                .printf("Main: Waiting for the Needle Pi" +
                                                " task generators to finish.¥n");
                for (int i = 0; i < length; i++) {
                        NeedleTesterThread[i].join();
                }
        } catch (InterruptedException e) {
                e.printStackTrace();
        }

        // executor shutdown
        int executionTime = 60;
        System.out.printf("Main: Shutting down the " +
```

```
                                "executor in %d seconds.¥n",
                                executionTime);

                executor.shutdown();
                try {
                        executor.awaitTermination(executionTime, TimeUnit.SECONDS);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }

                // send a massage to the post process task to halt its execution
                PostProcessor.setEnd(true);
                System.out.println("Main: Ends");

        }

}
```

-

```
Main: At Tue Apr 29 11:40:27 EEST 2014, Starting the Needle Pi simulation tasks Threads
Main: Waiting for the Needle Pi task generators to finish.
The post processor task will wait for tasks to finish for 20 seconds
NeedleTester3_Report: NeedlePiGenerator: Generating a report during 1 seconds
NeedleTester0_Report: NeedlePiGenerator: Generating a report during 1 seconds
NeedleTester1_Report: NeedlePiGenerator: Generating a report during 8 seconds
NeedleTester5_Report: NeedlePiGenerator: Generating a report during 2 seconds
NeedleTester2_Report: NeedlePiGenerator: Generating a report during 8 seconds
NeedleTester4_Report: NeedlePiGenerator: Generating a report during 4 seconds
Main: Shutting down the executor in 60 seconds.
PostProcessor: Tue Apr 29 11:40:28 EEST 2014 A new Needle Pi Received: 3.1333228889237037
PostProcessor: Tue Apr 29 11:40:28 EEST 2014 A new Needle Pi Received: 3.151276102088167
PostProcessor: Tue Apr 29 11:40:29 EEST 2014 A new Needle Pi Received: 3.1317160826594788
PostProcessor: Tue Apr 29 11:40:31 EEST 2014 A new Needle Pi Received: 3.1515053415344774
PostProcessor: Tue Apr 29 11:40:35 EEST 2014 A new Needle Pi Received: 3.1494661921708187
PostProcessor: Tue Apr 29 11:40:35 EEST 2014 A new Needle Pi Received: 3.1352876626430475
Best Pi approximation: 3,1352876626
PostProcessor: End
Main: Ends
```

-

Στην κλάση `main` του παραπάνω παραδείγματος, δημιουργήσαμε έναν `ThreadPoolExecutor` με την μέθοδο `newCachedThreadPool()` της βοηθητικής λειτουργικής κλάσης `Executors`. Έπειτα χρησιμοποιήσαμε αυτό το `executor` αντικείμενο για να αρχικοποιήσουμε το αντικείμενο `CompletionService`, επειδή χρειάζεται έναν `executor` για να εκτελεί τις εργασίες. Για να εκτελέσουμε μία εργασία χρησιμοποιώντας την `CompletionService` χρησιμοποιούμε την μέθοδο `submit()` στην `NeedlePiTaskRequest` κλάση.

Όταν μία από τις εργασίες που εκτελούνται έχει ολοκληρώσει την εκτέλεσή της, τότε η κλάση `completion service` αποθηκεύει το `Future` αντικείμενο που διατηρεί τα αποτελέσματα της εκτέλεσης της εργασίας. Η μέθοδος `poll()` στην κλάση `NeedlePiPostProcessor` αποκτά πρόσβαση στο χώρο αποθήκευσης αυτών των `Future` αντικειμένων για να ανακτήσει ένα από αυτά, και αν βρει κάποιο επιστρέφει το πρώτο νεοεισαχθέν `Future` αντικείμενο. Αφού έχει ανακτηθεί αυτό το `Future` αντικείμενο, έπειτα διαγράφεται από το προσωρινό χώρο αποθήκευσης που βρισκόταν από την δομή της κλάσης

completion service. Επίσης η μέθοδος poll() δέχεται ως παράμετρο την χρονική διάρκεια που χρειάζεται για να αναμένει την ολοκλήρωση της εκτέλεσης μιας εργασίας.

Αφού λοιπόν δημιουργούμε το αντικείμενο CompletionService, στη συνέχεια δημιουργούμε NeedlePiTaskRequests αντικείμενα τα οποία τα δημιουργήσουν με την σειρά τους TaskNeedlePiGenerator αντικείμενα που θα υπολογίζουν το pi του κύκλου με την γνωστή μέθοδο Buffon Needle Pi Problem. Η εργασία NeedlePiPostProcessor επεξεργάζεται τα αποτελέσματα που παράγονται από τις εργασίες υπολογισμού του pi του κύκλου και υπολογίζει την καλύτερη προσέγγιση του pi.

Η κλάση CompletionService μπορεί να εκτελέσει Callable και Runnable εργασίες. Βέβαια και τα Runnable αντικείμενα δεν παράγουν αποτελέσματα και η όλη φιλοσοφία της CompletionService για μετέπειτα επεξεργασία των αποτελεσμάτων δεν εφαρμόζεται για τα Runnable αντικείμενα.

Η κλάση CompletionService παρέχει 2 μεθόδους για να ανακαλούμε τα Future αντικείμενα τον ολοκληρωμένων εργασιών. Οι μέθοδοι είναι οι εξής:

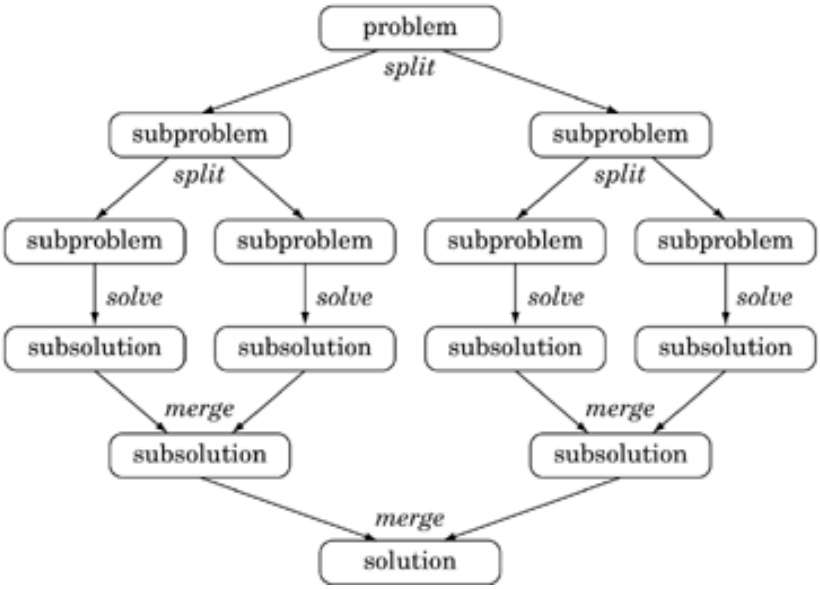
- poll(): που ελέγχει εάν υπάρχει κάποιο Future αντικείμενο από μία ολοκληρωμένη εργασία, εάν δεν υπάρχει μας επιστρέφει την τιμή null. Διαφορετικά μας επιστρέφει το πρώτο Future αντικείμενο από την εργασία που πρώτη είχε ολοκληρωθεί.
- take(): Η μέθοδος αυτή, ελέγχει εάν υπάρχουν Future αντικείμενα στον προσωρινό χώρο αποθήκευσης της κλάσης CompletionService. Εάν είναι άδειος τότε αναστέλλει την λειτουργία του νήματος που κάλεσε τη συγκεκριμένη μέθοδο έως ότου κάποιο Future αντικείμενο εμφανιστεί. Όταν όμως ο συγκεκριμένος χώρος αποθήκευσης έχει Future αντικείμενα, τότε επιστρέφει το πρώτο Future αντικείμενο και το διαγράφει από τον προσωρινό χώρο αποθήκευσης που υπήρχε.

Η δομή Fork / Join

Εισαγωγή

Η Java περιλαμβάνει μία επιπρόσθετη εφαρμογή της διασύνδεσης ExecutorService που είναι εξειδικευμένη για προβλήματα συγκεκριμένου τύπου. Η δομή αυτή ονομάζεται Fork/Join. Η συγκεκριμένη δομή είναι σχεδιασμένη ώστε να επιλύει προβλήματα τα οποία μπορούν να διασπαστούν σε μικρότερες εργασίες χρησιμοποιώντας την τεχνική "*διαίρει και βασίλευε - divide and conquer*". Η συγκεκριμένη τεχνική λειτουργεί ως εξής: Κάθε εργασία ελέγχει το μέγεθος του προβλήματος που πρέπει να επιλυθεί. Εάν είναι μεγαλύτερο από το μέγεθος που προκαθορίζεται από τον προγραμματιστή τότε η εργασία αυτή διαιρεί το πρόβλημα σε μικρότερες εργασίες που θα εκτελεστούν από το ίδιο πακέτο δομών Fork/Join. Εάν το μέγεθος του προβλήματος είναι μικρότερο από προκαθορισμένο μέγεθος, τότε η εργασία αυτή επιλύει – εκτελεί το πρόβλημα της εφαρμογής αμέσως και έχει την δυνατότητα να επιστρέψει τα αποτελέσματα της εκτέλεσής της.

Το παρακάτω διάγραμμα παρουσιάζει τη γενική ιδέα του "διαίρει και βασίλευε":



Δεν υπάρχει κάποιος κανόνας για να καθορίσουμε εκείνο το κρίσιμο μέγεθος του προβλήματος που κρίνει εάν μια εργασία θα διαιρεθεί για παράδειγμα σε δύο νέες εργασίες. Αυτό εξαρτάται από τα χαρακτηριστικά του προβλήματος που πρέπει να επιλυθεί (πολυπλοκότητα, απαιτούμενη μνήμη) αλλά και την ισχύ του διαθέσιμου υπολογιστικού συστήματος. Συνηθίζεται να πειραματιζόμαστε με διαφορετικά κρίσιμα μεγέθη ώστε να επιλέξουμε το καλύτερο για κάθε πρόβλημα που θα εκτελεστεί από την εφαρμογή. Η κλάση που χρησιμοποιούμε για την δημιουργία νέων μικρότερων εργασιών από τις ήδη υπάρχουσες είναι η `ForkJoinPool`.

Η κλάση `ForkJoinPool` είναι μια ειδική περίπτωση `Executor`. Η δομή αυτή βασίζεται σε 2 συγκεκριμένες λειτουργίες:

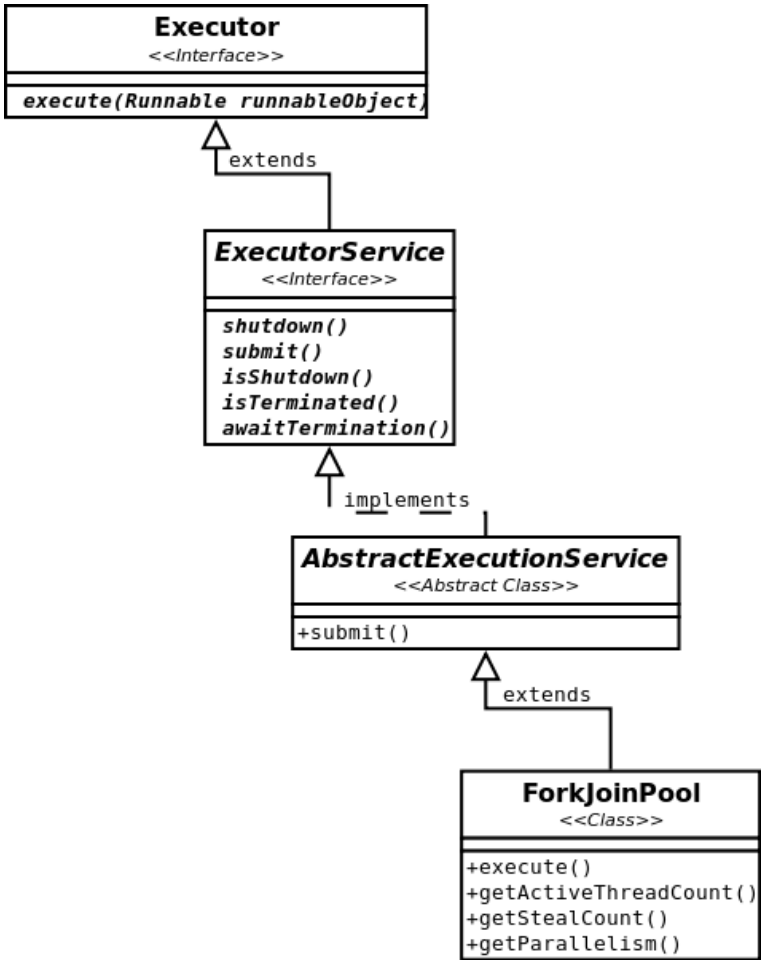
- Τη λειτουργία `fork`: Όπου με την συγκεκριμένη λειτουργία μπορούμε να διαιρέσουμε μια εργασία σε μικρότερες εργασίες και επίσης να τις εκτελέσουμε χρησιμοποιώντας την ίδια πάλι κλάση `ForkJoinPool`.
- Τη λειτουργία `join`: Όπου μια εργασία αναμένει το τερματισμό των εργασιών που έχουν δημιουργηθεί από την διαίρεση της αρχικής εργασίας.

Η δομή `Fork / Join`, όπως και κάθε `executor`, περιέχει μία δεξαμενή νημάτων, τα οποία εκμεταλλεύονται πλήρως τον χρόνο εκτέλεσής τους, για αυτό γενικά έχουμε γενικά καλή απόδοση των εφαρμογών τύπου “διαίρει και βασίλευε”. Ωστόσο για να επιτευχθεί καλή απόδοση, οι εργασίες που εκτελούνται από τη δομή `Fork / Join` πρέπει να υπακούν στους εξής περιορισμούς:

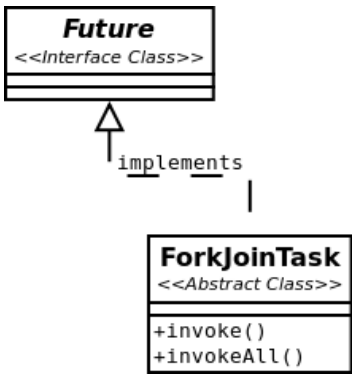
1. Οι εργασίες μπορούν να χρησιμοποιήσουν τις λειτουργίες `fork()` και `join()` ως μηχανισμούς συγχρονισμού. Εάν χρησιμοποιήσουν άλλους μηχανισμούς συγχρονισμού τότε τα νήματα – εργασίες δεν μπορούν να εκτελέσουν τη συνέχεια της. Για παράδειγμα εάν χρησιμοποιήσουμε μέσα σε μια εργασία τη μέθοδο `sleep()` μέσα στη δομή `Fork / Join`, τότε το νήμα που εκτελεί μία εργασία θα ανασταλεί .
2. Εργασίες που χρησιμοποιούν λειτουργίες I/O (εισόδου/εξόδου), όπως για παράδειγμα ανάγνωση και εγγραφή δεδομένων από/σε ένα αρχείο, δυσχεραίνουν τη λειτουργία της δομής `Fork / Join`.
3. Οι εργασίες δεν μπορούν να παρουσιάζουν εξαιρέσεις (`exceptions`). Για αυτό το λόγο πρέπει να εισαχθούν μέσα στο κώδικα της εφαρμογής ειδικού τύπου εξαιρέσεις.

Ο πυρήνας της δομής `Fork / Join` απαρτίζεται από τις ακόλουθες κλάσεις:

1. `ForkJoinPool`: Εφαρμόζει την διασύνδεση `ExecutorService`, και διαχειρίζεται τα νήματα που εκτελούν τις διάφορες εργασίες – υποεργασίες μέσα σε μια δεξαμενή νημάτων. Τέλος παρέχει πληροφορίες σχετικά με την κατάσταση των εργασιών και την συνολική πορεία της εκτέλεσής τους. Το UML διάγραμμα που μας δείχνει τα στοιχεία που κληρονομεί αλλά και μερικές από τις μεθόδους της συγκεκριμένης κλάσης είναι το εξής:

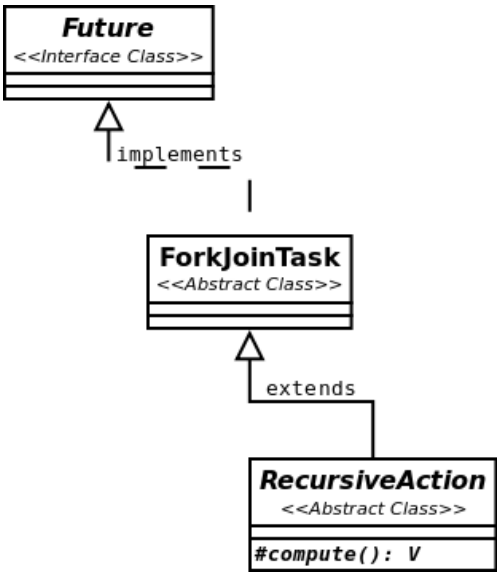


2. `ForkJoinTask`: Είναι μια βασική κλάση για τις εργασίες που εκτελούνται σε ένα `ForkJoinPool`. Παρέχει τους μηχανισμούς εκείνους για την εκτέλεση των λειτουργιών `fork() & join()` μέσα σε μία εργασία και τις μεθόδους ελέγχου της συνολικής κατάστασης των εργασιών. Το UML διάγραμμα της συγκεκριμένης κλάσης είναι το εξής:

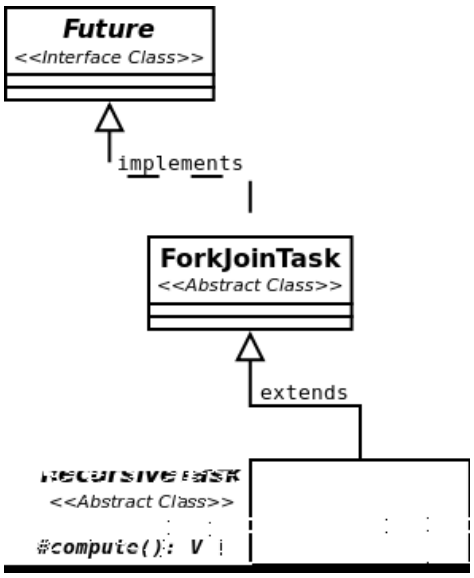


Για να εφαρμόσουμε την δομή `Fork / Join`, η κλάση της εφαρμογής που επιλύει ένα πρόβλημα τύπου “διαίρει και βασίλευε” εφαρμόζει (`implement`) μέσα στον κώδικά της μία από τις παρακάτω κλάσεις:

1. Την κλάση `RecursiveAction`: Η οποία είναι μία αφηρημένη κλάση και χρησιμοποιείται για εργασίες, υποεργασίες που δεν επιστρέφουν κάποιο αποτέλεσμα. Το UML διάγραμμα της είναι το εξής:



2. Την κλάση `RecursiveTask`: Η οποία είναι μία αφηρημένη κλάση και χρησιμοποιείται για τις εργασίες, υποεργασίες που επιστρέφουν κάποιο αποτέλεσμα Το UML διάγραμμα της είναι το εξής:



Δημιουργία μιας δεξαμενής νημάτων Fork / Join χωρίς επιστροφή αποτελεσμάτων από τις εργασίες .

Σε αυτήν την ενότητα θα μάθουμε τα βασικούς τρόπους χειρισμού της δομής Fork/Join. Αυτά περιλαμβάνουν:

1. Τη δημιουργία των αντικειμένων `ForkJoinPool` καθώς και η εκτέλεση των εργασιών
2. Τη δημιουργία των αντικειμένων της κλάσης `ForkJoinTask` που περιλαμβάνει τον κώδικα για την υλοποίηση των εργασιών.

Τα κύρια χαρακτηριστικά της δομής `Fork/Join` που θα χρησιμοποιήσουμε στην συγκεκριμένη ενότητα είναι τα παρακάτω:

1. Θα δημιουργήσουμε ένα αντικείμενο `ForkJoinPool`
2. Θα υλοποιήσουμε μέσα στον κορμό της εργασίας που πρέπει να επιλύσει ένα πρόβλημα με τη τεχνική “διαίρει και βασίλευε” ως εξής:

```
Function DivideNConquerExecute(task) {

      if (problem size > default size) {

            //divide task into smaller subtasks
             tasks=divide(task);

            //then execute them
             DivideNConquerExecute(tasks);
      }
      else {
            resolve problem using another algorithm;
      }
}
```

Στο συγκεκριμένο παράδειγμα μία κλάση παράγει μία λίστα από ακέραιους αριθμούς. Μία ακόμη

κλάση υλοποιεί τον παραπάνω αλγόριθμο και στο τέλος της αναδρομής κάθε υποεργασία που δημιουργείται ελέγχει εάν ο ακέραιος αριθμός που κατέχει είναι πρώτος (`prime`) και εκτυπώνει το ανάλογο αποτέλεσμα στην οθόνη. Οι εργασίες δεν επιστρέφουν κάποιο αποτέλεσμα για αυτό το λόγο χρησιμοποιούμε την κλάση `RecursiveAction`.

-

```
import java.util.Random;

/*
 * This class defines and prints a list of integers
 */

public class ListCreator {
        private final int[] list;

        public ListCreator(int n) {
                list = new int[n];
                Random generator = new Random(System.currentTimeMillis());
                for (int i = 0; i < list.length; i++) {             i
                        list[i] = generator.nextInt(20);
                }
        }

        public int
```

```
// constructor
public TaskListSeparator(int[] array) {
        this.list = array;
}

@Override
protected void compute() {
        // if the list is fully separated into abstract integer elements
        // then each thread in the pool print the given thread
        if (list.length == 1) {
                result = list[0];
                System.out.println("I am thread " + Thread.currentThread()
                                + " and I have got the number "
                                + result + " is this a prime number? "
                                + isPrime(result));

                // sleep for 2 seconds
                try {
                        Thread.currentThread().sleep(2000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
        /*
         * create two new TaskListSeparator objects, one to process the first
         * half of the list and the other to process the second half and execute
         * them in ForkJoinPool using the invokeAll() method.
         */
        else {
                int midpoint = list.length / 2;
                int[] l1 = Arrays.copyOfRange(list, 0, midpoint);
                int[] l2 = Arrays.copyOfRange(list, midpoint, list.length);
                TaskListSeparator s1 = new TaskListSeparator(l1);
                TaskListSeparator s2 = new TaskListSeparator(l2);
                invokeAll(s1, s2);
        }
}

public boolean isPrime(int input) {

        // Check multiples of two
        if (input % 2 == 0) {
                if (input != 2)
                        return false;
                else
                        return true;
        }

        // otherwise
        for (int i = 3; i <= input - 1; i++) {
                if (input % i == 0) {
                        return false;
                }
        }

        return true;
```

```
        }
}
```

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.TimeUnit;

public class MainListForkJoin {

        public static void main(String[] args) {

                // create a list
                ListCreator test = new ListCreator(23);

                // create a task that will divide and separate
                TaskListSeparator task = new TaskListSeparator(test.getList());

                // Create a ForkJoinPool object using the constructor without
                // parameters. Thus the number of threads area equal to the PC's processors
                ForkJoinPool pool = new ForkJoinPool();

                // Execute the task in the pool using the execute() method.
                pool.execute(task);

                /*
                 * a block of code that shows information about the evolution of the
                 * pool every 3 seconds writing to the console the value of some
                 * parameters of the pool until the task finishes its execution.
                 */
                do {
                        System.out.printf("Main: Thread Count: %d¥n", pool
                                        .getActiveThreadCount());
                        System.out.printf("Main: Thread Steal: %d¥n", pool.getStealCount());
                        System.out.printf("Main: Parallelism: %d¥n", pool.getParallelism());
                        try {
                                //sleep for 6 seconds
                                TimeUnit.MILLISECONDS.sleep(6000);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                } while (!task.isDone());

                //Shut down the pool using the shutdown() method.
                pool.shutdown();
                System.out.println("----Main: The ForkJoinPool" +
                                " has shutted down----");

        }

}
```

-

```
Printing a random list of numbers

19      2       18      14      17
```

```
8       0       12      18      17
8       3       6       14      15
4       1       19      4       11
11      9       4

Main: Thread Count: 1
I am thread Thread[ForkJoinPool-1-worker-1,5,main] and I have got the number 19 is this a prime number? true
I am thread Thread[ForkJoinPool-1-worker-2,5,main] and I have got the number 3 is this a prime number? true
Main: Thread Steal: 0
Main: Parallelism: 2
I am thread Thread[ForkJoinPool-1-worker-2,5,main] and I have got the number 6 is this a prime number? false
I am thread Thread[ForkJoinPool-1-worker-1,5,main] and I have got the number 2 is this a prime number? true
I am thread Thread[ForkJoinPool-1-worker-2,5,main] and I have got the number 14 is this a prime number? false
I am thread Thread[ForkJoinPool-1-worker-1,5,main] and I have got the number 18 is this a prime number? false
I am thread Thread[ForkJoinPool-1-worker-2,5,main] and I have got the number 15 is this a prime number? false
I am thread Thread[ForkJoinPool-1-worker-1,5,main] and I have got the number 14 is this a prime number? false
Main: Thread Count: 2
Main: Thread Steal: 0
Main: Parallelism: 2
I am thread Thread[ForkJoinPool-1-worker-2,5,main] and I have got the number 4 is this a prime number? false
I am thread Thread[ForkJoinPool-1-worker-1,5,main] and I have got the number 17 is this a prime number? true
I am thread Thread[ForkJoinPool-1-worker-2,5,main] and I have got the number 1 is this a prime number? true
I am thread Thread[ForkJoinPool-1-worker-1,5,main] and I have got the number 8 is this a prime number? false
I am thread Thread[ForkJoinPool-1-worker-2,5,main] and I have got the number 19 is this a prime number? true
I am thread Thread[ForkJoinPool-1-worker-1,5,main] and I have got the number 0 is this a prime number? false
Main: Thread Count: 2
Main: Thread Steal: 0
Main: Parallelism: 2
I am thread Thread[ForkJoinPool-1-worker-2,5,main] and I have got the number 4 is this a prime number? false
I am thread Thread[ForkJoinPool-1-worker-1,5,main] and I have got the number 12 is this a prime number? false
I am thread Thread[ForkJoinPool-1-worker-2,5,main] and I have got the number 11 is this a prime number? true
I am thread Thread[ForkJoinPool-1-worker-1,5,main] and I have got the number 18 is this a prime number? false
I am thread Thread[ForkJoinPool-1-worker-2,5,main] and I have got the number 11 is this a prime number? true
I am thread Thread[ForkJoinPool-1-worker-1,5,main] and I have got the number 17 is this a prime number? true
Main: Thread Count: 2
Main: Thread Steal: 0
Main: Parallelism: 2
I am thread Thread[ForkJoinPool-1-worker-2,5,main] and I have got the number 9 is this a prime number? false
I am thread Thread[ForkJoinPool-1-worker-1,5,main] and I have got the number 8 is this a prime number? false
I am thread Thread[ForkJoinPool-1-worker-2,5,main] and I have got the number 4 is this a prime number? false
----Main: The ForkJoinPool has shutted down----
```

-

Σε αυτό το παράδειγμα, δημιουργήσαμε ένα αντικείμενο `ForkJoinPool` και μια κλάση `RecursiveAction`, υποκλάση της `ForkJoinTask` για την εκτέλεση των εργασιών μέσα σε μια δεξαμενή νημάτων `ForkJoinPool`. Για να δημιουργήσουμε ένα αντικείμενο `ForkJoinPool` χρησιμοποιήσαμε τον κατασκευαστή χωρίς παραμέτρους. Έτσι δημιουργήσαμε μία δεξαμενή νημάτων με αριθμό νημάτων ίσο με τον αριθμό των επεξεργαστών του υπολογιστή που εκτελείται το πρόγραμμά μας.

Η κλάση – εργασία `TaskListSeperator` δεν επιστρέφει κάποιο αποτέλεσμα, και για αυτό το λόγο επεκτείνει την κλάση `RecursiveAction`. Στο συγκεκριμένο παράδειγμα εφαρμόσαμε τον αλγόριθμο "διαίρει και βασίλευε" με κρίσιμο μέγεθος (που καθορίζει εάν θα διασπαστεί μια εργασία σε νέες υποεργασίες είτε να επιλυθεί το πρόβλημα της εργασίας με κάποια διαφορετική μεθοδολογία – αλγόριθμο), να είναι ίσο με 1.

Για να εκτελέσουμε τις υποεργασίες που δημιουργούνται από μια εργασία, καλούμε την μέθοδο `invokeAll()`. Η μέθοδος αυτή είναι μία συγχρονισμένη μέθοδος, που σημαίνει ότι η εργασία που έχει

διαιρεθεί σε μικρότερες υποεργασίες θα αναμένει τον τερματισμό της εκτέλεσής τους για να συνεχίσει να εκτελεί το υπόλοιπο του προγράμματος. Ένα πλεονέκτημα που έγκειται σε αυτήν την διαδικασία της αναμονής είναι ότι το νήμα που είναι υπεύθυνο για την εκτέλεση της αρχικής εργασίας η οποία αναμένει για τον τερματισμό της εκτέλεσης των υποεργασιών της, μπορεί να αναλάβει να εκτελέσει κάποια άλλη εργασία που αναμένει ένα νήμα από την δεξαμενή νημάτων. Αυτή η συμπεριφορά της δομής `Fork/Join` προσφέρει μια πιο αποδοτική διαχείριση εργασιών από ότι τα αντικείμενα `Runnable` και `Callable`.

Η μέθοδος `invokeAll()` της κλάσης `ForkJoinTask` παρουσιάζει μια διαφορά μεταξύ της δομής `Fork/Join` και των `Executor`. Στην δομή `Executor`, όλες οι εργασίες πρέπει να σταλούν μέσα σε ένα `executor` για την εκτέλεση και διαχείρισή τους, ενώ στην δομή `Fork/Join`, οι ίδιες οι εργασίες περιλαμβάνουν μεθόδους για την εκτέλεσή τους. Μια από τις μεθόδους αυτές είναι η `invokeAll()` μέθοδος στην κλάση `TaskListSeperator` που επεκτείνεται από την `RecursiveAction` κλάση η οποία επεκτείνει την κλάση `ForkJoinTask`.

Επίσης χρησιμοποιούμε στην `Main` κλάση, την μέθοδο `execute()` της `ForkJoinPool`. Ο τρόπος που λειτουργεί η συγκεκριμένη κλάση είναι να εκτελεί την κάθε εργασία μέσα στην δεξαμενή νημάτων της `ForkJoinPool`. Ο τρόπος λειτουργίας της είναι ασύγχρονος, δηλαδή το νήμα της κλάσης `main` συνεχίζει την εκτέλεσή του ασχέτως με την πορεία της κάθε εργασίας.

Επιπροσθέτως χρησιμοποιήσαμε μερικές μεθόδους της κλάσης `ForkJoinPool` για τον έλεγχο της κατάστασης και εξέλιξης των εργασιών που βρίσκονται εκτελούνται. Συγκεκριμένα σε τακτά χρονικά διαστήματα εκτυπώνουμε τον συνολικό αριθμό νημάτων της δεξαμενής νημάτων `ForkJoinPool` που εκτελούν μια εργασία, τον αριθμό των νημάτων που εκτελούνται παράλληλα, καθώς και τον αριθμό των νημάτων που λογω απόδοσης “έκλεψαν” κάποιες εργασίες από άλλα νήματα που αρχικά τις είχαν αναλάβει (work stealing). Η τεχνική work stealing μπορεί να επιτύχει δυναμική εξισορρόπηση φορτίου, σε περίπτωση που οι υποεργασίες δεν έχουν το ίδιο υπολογιστικό φόρτο.

Πρέπει να σημειωθεί ότι η κλάση `ForkJoinPool` παρέχει και άλλες μεθόδους για την εκτέλεση των εργασιών:

- `execute(Runnable task)`: Είναι μια παραλλαγή της μεθόδου `execute()` που συναντήσαμε στο προηγούμενο παράδειγμα. Σε αυτήν την περίπτωση στέλνουμε μια `Runnable` εργασία σε ένα αντικείμενο `ForkJoinPool`.
- `invoke(ForkJoinTask<T> task)`: Ενώ η μέθοδος `execute()` έχει μια ασύγχρονη συμπεριφορά στα αντικείμενα τύπου `ForkJoinPool`, η μέθοδος `invoke()` έχει μία σύγχρονη συμπεριφορά, ώστε όταν η κλάση που θα την καλέσει θα αναμένει έως ότου η εργασία που έχει δοθεί ως όρισμα τερματίσει την εκτέλεσή της.
- `invokeAll(ForkJoinTask<?>... tasks)`: Αυτή η έκδοση της μεθόδου `invokeAll()` χρησιμοποιεί ένα μεταβλητό αριθμό από εργασίες.
- `invokeAll(Collection<T> tasks)`: Αυτή η έκδοση της μεθόδου `invokeAll()` δέχεται αντικείμενα εργασιών που είναι αποθηκευμένα σε συλλογές δεδομένων όπως `ArrayList`, `LinkedList` κ.α. Πρέπει ωστόσο να οριστεί ο γενικός αφηρημένος τύπος δεδομένων T τύπου `ForkJoinTask` είτε κάποια υποκλάση του (`RecursiveAction`, `RecursiveTask`).

Αν και η κλάση `ForkJoinPool` έχει σχεδιαστεί για την εκτέλεση αντικειμένων τύπου ForkJoinTask, μπορούμε να εκτελέσουμε `Runnable` και `Callable` αντικείμενα άμεσα.

Δημιουργία μιας δεξαμενής νημάτων Fork/Join χωρίς επιστροφή αποτελεσμάτων από τις εργασίες, παράδειγμα #2.

Σε αυτήν την ενότητα για να γίνει πιο κατανοητή η χρήση της δομής `Fork/Join` θα παρουσιαστεί

ένα ακόμη παράδειγμα. Στο παράδειγμα που ακολουθεί έχουμε μία κλάση που υπολογίζει το τετράγωνο ακεραίων αριθμών από μία λίστα ακεραίων αριθμών. Η εργασία – κλάση ασχολείται με τη συγκεκριμένη διαδικασία ονομάζεται `TaskSquareCalculator` και η κλάση `main` απλώς εκτυπώνει τα αποτελέσματα ονομάζεται `MainSquareCalculator`. Ο αλγόριθμος και η όλη διαδικασία είναι η ίδια όπως και στο παραπάνω παράδειγμα, δηλαδή ομοίως ορίζεται μια μεταβλητή που ορίζει το κρίσιμο κομμάτι του κώδικα που είτε θα δημιουργηθούν νέες υποεργασίες που θα κατέχουν ένα μικρότερο τμήμα της λίστας αυτών των ακεραίων αριθμών με μέγεθος όσο θα είναι η τιμή της παραπάνω μεταβλητής. Όσο δεν επιτυγχάνεται αυτός ο στόχος, τόσο δημιουργούνται αυτές οι νέες υποεργασίες με στόχο να επιτύχουμε την την δυνατότητα του αλγορίθμου του “διαίρει και βασίλευε”. Επίσης η διαφορά με το προηγούμενο παράδειγμα είναι ότι ενώ στο προηγούμενο η εκτέλεση των υποεργασιών γινόταν σύγχρονα, στο συγκεκριμένο θα γίνει ασύγχρονα:

-

```
import java.util.concurrent.RecursiveAction;

public class TaskSquareCalculator extends RecursiveAction {

	final int CRITICAL_LIMIT = 3;
	int result;
	int start, end;
	int[] data;

	TaskSquareCalculator(int[] data, int start, int end) {
		this.start = start;
		this.end = end;
		this.data = data;
	}

	@Override
	protected void compute() {
		if ((end - start) < CRITICAL_LIMIT) {
			for (int i = start; i < end; i++) {
				result = data[i] * data[i];
				try {
					System.out.println("I am Thread: "
							+ Thread.currentThread().getName()
							+ " and the Power of " + data[i]
							+ " is " + result);
					Thread.sleep(1000);
				} catch (InterruptedException e) {
					// TODO Auto-generated catch block
					e.printStackTrace();
				}

			}
		} else {
			int mid = (start + end) / 2;
			TaskSquareCalculator leftSubTask = new TaskSquareCalculator(
					data, start,mid);
			TaskSquareCalculator rightSubTask = new TaskSquareCalculator(
					data, mid,end);
			leftSubTask.fork();
			rightSubTask.fork();
			leftSubTask.join();
			rightSubTask.join();
```

```
                }
        }
}
```

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.TimeUnit;

public class MainSquareCalculator {
        public static void main(String[] args) {
                ForkJoinPool pool = new ForkJoinPool();
                int[] data = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
                TaskSquareCalculator task = new TaskSquareCalculator(data, 0,
                                data.length);
                pool.execute(task);

                do {
                        System.out.printf("------------¥n¥tMain Report: Thread Count: %d¥n"
                                        + "¥tMain Report: Thread Steal: %d¥n"
                                        + "¥tMain Report: Parallelism: %d¥n------------¥n"
                                        , pool.getActiveThreadCount()
                                        , pool.getStealCount()
                                        , pool.getParallelism());
                        try {
                                // sleep for 2 seconds until the new main report
                                TimeUnit.MILLISECONDS.sleep(2000);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                } while (!task.isDone());

                pool.shutdown();
                System.out.println("----Main: The ForkJoinPool"
                                + " has shutted down----");

        }
}
```

-

```
------------
        Main Report: Thread Count: 1
        Main Report: Thread Steal: 0
        Main Report: Parallelism: 2
------------
I am Thread: ForkJoinPool-1-worker-4 and the Power of 1 is 1
I am Thread: ForkJoinPool-1-worker-4 and the Power of 2 is 4
I am Thread: ForkJoinPool-1-worker-2 and the Power of 3 is 9
I am Thread: ForkJoinPool-1-worker-5 and the Power of 4 is 16
------------
        Main Report: Thread Count: 5
        Main Report: Thread Steal: 0
        Main Report: Parallelism: 2
------------
```

```
I am Thread: ForkJoinPool-1-worker-5 and the Power of 5 is 25
I am Thread: ForkJoinPool-1-worker-5 and the Power of 6 is 36
I am Thread: ForkJoinPool-1-worker-2 and the Power of 8 is 64
------------
        Main Report: Thread Count: 5
        Main Report: Thread Steal: 0
        Main Report: Parallelism: 2
------------
I am Thread: ForkJoinPool-1-worker-5 and the Power of 7 is 49
I am Thread: ForkJoinPool-1-worker-2 and the Power of 9 is 81
I am Thread: ForkJoinPool-1-worker-2 and the Power of 10 is 100
------------
        Main Report: Thread Count: 4
        Main Report: Thread Steal: 2
        Main Report: Parallelism: 2
------------
----Main: The ForkJoinPool has shutted down----
```

-

Όπως και στο προηγούμενο παράδειγμα διαιρούμε το πρόβλημα σε μικρότερα προβλήματα τα οποία τα αναλαμβάνουν νέες υποεργασίες έως ότου το μέγεθος του προβλήματος γίνει ίσο με την τιμή της κρίσιμης μεταβλητής. Όταν φτάσει το μέγεθος της λίστας σε αυτό το κρίσιμο μέγεθος τότε οι υποεργασίες εκτελούν τους υπολογισμούς τους.

Η διαφορά με το προηγούμενο παράδειγμα βρίσκεται στο τρόπο κλήση και εκτέλεσης των νέων υποεργασιών. Αντί της χρήσης της `invokeAll()` χρησιμοποιούμε τις μεθόδου `fork` και `join` της κλάσης `ForkJoinTask`. Συγκεκριμένα:

1. `ForkJoinTask.fork()`: Η μέθοδος `fork()` καλείται μέσα από μία κλάση που επεκτείνει την `ForkJoinTask` όταν βρίσκεται μέσα σε ένα `ForkJoinPool`. Η διαφορά με την προηγούμενη μέθοδο `invokeAll()` είναι ενώ η κλάση `invokeAll()` όσο δημιουργούνται νέες υποκλάσεις η διαδικασία εκτέλεσής τους είναι συγχρονισμένη, δηλαδή όλες οι εργασίες θα αναμένουν η μία την άλλη ώστε να τερματιστεί η εκτέλεσή τους, ενώ με τη μέθοδο `fork()` η διαδικασία εκτέλεσης των υποεργασιών γίνεται ασύγχρονα, δηλαδή κάθε εργασία εκτελείται παράλληλα με την άλλη και δεν αναμένει η μία την άλλη για το πότε θα ολοκληρώσουν την εκτέλεσή τους.
2. `ForkJoinTask.join()`: Χρησιμοποιούμε τη συγκεκριμένη μέθοδο ώστε κάθε υποεργασία να αναμένει τον τερματισμό της εκτέλεσης της κάθε υποεργασίας που προέρχεται - δημιουργήθηκε από την ίδια γονική εργασία αφού πρώτα έχουν εκτελεστεί παραλλήλως. Η πιο απλά μια υποεργασία θα αναμένει τα “αδέλφια” της υποεργασίες να τερματίσουν την εκτέλεσή τους.

Δημιουργία μια δεξαμενής νημάτων Fork / Join με επιστροφή αποτελεσμάτων από τις εργασίες.

Η δομή Fork/Join παρέχει την δυνατότητα για την εκτέλεση εργασιών τα οποία θα επιστρέφουν κάποιο αποτέλεσμα. Αυτού του είδους οι εργασίες εφαρμόζουν (`implements`) την κλάση `RecursiveTask`. Η `RecursiveTask` κλάση επεκτείνει (`extends`) την κλάση `ForkJoinTask` που εφαρμόζει (`implements`) την διασύνδεση `Future` που παρέχεται από το πακέτου δομών `Executor`.

Στο εσωτερικό κλάσης που υλοποιεί την εργασία που διεκπεραιώνει τον αλγόριθμο “διαίρει και βασίλευε”:

```
Function DivideNConquerExecute(task) {
```

```
      if (problem size > default size) {

           //divide task into smaller subtasks
            tasks=divide(task);

           //then execute them
            DivideNConquerExecute(tasks);

           groupResults();
           return result;

      }
      else {
           //resolve problem using another algorithm;
           return result;

      }
}
```

Εάν η εργασία έχει να επιλύσει ένα πρόβλημα μεγαλύτερο από το προκαθορισμένο κρίσιμο μέγεθος, τότε δημιουργεί νέες εργασίες με διαιρεμένο στη μέση το αρχικό μέγεθος του προβλήματος, έπειτα εκτελούμε αυτές τις υποεργασίες χρησιμοποιώντας τη δομή Fork/Join. Όταν μια υποεργασία έχει ολοκληρώσει την εκτέλεσή της, η αρχική μητρικής της εργασία παραλαμβάνει τα αποτελέσματα του δημιουργήθηκαν και τα ομαδοποιεί – συγκεντρώνει ώστε να υπολογίσει το τελικό αποτέλεσμα.

Στο παρακάτω παράδειγμα θα ασχοληθούμε με τη δομή αυτών των προβλημάτων αναπτύσσοντας μια εφαρμογή η οποία υπολογίζει την ακολουθία Fibonacci. Η παρακάτω υλοποίηση απαρτίζεται από δύο κλάσεις: Την κλάση main και την κλάση – εργασία TaskFibonnaciCalculator.

-

```
import java.util.concurrent.RecursiveTask;

class TaskFibonacciCalculator extends RecursiveTask<Integer> {

        /*
         * origins of this example:
         * http://www.javac.info/jsr166z/jsr166z/forkjoin/RecursiveTask.html
         *
         */

        int input;

        //constructor method
        TaskFibonacciCalculator(int num) {
                this.input = num;
        }

        @Override
        protected Integer compute() {

                if (input == 0) {
                        return 0;
                } else if (input <= 2) {
                        return 1;
```

```
                } else {
                        TaskFibonacciCalculator fcal1 = new TaskFibonacciCalculator(
                                        input - 1);
                        fcal1.fork();
                        TaskFibonacciCalculator fcal2 = new TaskFibonacciCalculator(
                                        input - 2);
                        return fcal2.compute() + fcal1.join();
                }
        }
}
```

```
import java.util.concurrent.ForkJoinPool;

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

                // creating a ForkJoinPool with threads equal to the number
                // of cpu cores (for speed improvement)
                int AVAILABLE_PROCESSORS = Runtime.getRuntime().availableProcessors();
                ForkJoinPool pool = new ForkJoinPool(AVAILABLE_PROCESSORS);

                // generate some fibonacci numbers
                int num_of_fibonacci_numbers = 10;

                for (int i = 1; i <= num_of_fibonacci_numbers; i++) {

                        TaskFibonacciCalculator fibonacciCal =
                                new TaskFibonacciCalculator(i);

                        int fibonacciResult = 0;
                        try {
                                // execute the RecursiveTask inside the ForkJoinPool
                                fibonacciResult = pool.invoke(fibonacciCal);
                                System.out.println("The #" + i + " fibonacci number is "
                                                + fibonacciResult);

                                //then sleep the main thread for 1 second
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }

                }

                //shutting down the ForkJoin pool
                pool.shutdown();
                System.out.println("----Main: The ForkJoinPool"
                                + " has shutted down----");
        } // end of main method
} // end of main class
```

-

```
The #1 fibonacci number is 1
```

```
The #2 fibonacci number is 1
The #3 fibonacci number is 2
The #4 fibonacci number is 3
The #5 fibonacci number is 5
The #6 fibonacci number is 8
The #7 fibonacci number is 13
The #8 fibonacci number is 21
The #9 fibonacci number is 34
The #10 fibonacci number is 55
----Main: The ForkJoinPool has shutted down----
```

-

Η κλάση main δημιουργεί ένα ForkJoinPool αντικείμενο χρησιμοποιώντας το κατασκευαστή του αντικειμένου που μπορούμε να ορίσουμε τον αριθμό των νημάτων να είναι ίσο με τον μέγιστο αριθμό πυρήνων του υπολογιστή που εκτελεί το συγκεκριμένο πρόγραμμα. Στη συνέχεια με την επαναληπτική δομή for-loop υπολογίζουμε την ακολουθία Fibonacci. Η δυναμική του αλγορίθμου “διαίρει και βασίλευε” παρουσιάζεται στον τρόπο του υπολογισμού του κάθε αριθμού της ακολουθίας. Κάθε αριθμός της ακολουθίας χρειάζεται υπολογίζεται ως το άθροισμα των 2 προηγούμενών του αριθμών για να υπολογιστεί. Αυτό επιτυγχάνεται με τον συγκεκριμένο αλγόριθμο, δηλαδή κάθε εργασία για να υπολογίσει ένα νέο αριθμό παράγει 2 νέες υποεργασίες οι οποίες υπολογίζουν το άθροισμα αυτού του αριθμού.

Συντρέχουσες συλλογές δεδομένων

Εισαγωγή

Όταν χρειάζεται να εργαστούμε με δομές δεδομένων σε ένα συντρέχον πρόγραμμα, πρέπει να είμαστε προσεκτικοί με την υλοποίησή τους. Οι περισσότερες δομές δεδομένων δεν είναι κατάλληλες για να συνεργαστούν με συντρέχοντα νήματα επειδή δεν υποστηρίζουν συγχρονισμένη πρόσβαση στα δεδομένα τους. 'Οταν συντρέχουσες εργασίες έχουν μία κοινή δομή δεδομένων που δεν υποστηρίζει μηχανισμούς ελέγχου συντρέχουσας προσπέλασης των δεδομένων της, τότε εμφανίζονται προβλήματα “σ υ ν έ π ε ι α ς δ ε δ ο μ έ ν ω ν”.

Η Java παρέχει συλλογές δομών δεδομένων που μπορούν να χρησιμοποιηθούν στα συντρέχοντα προγράμματά μας χωρίς προβλήματα και ασυνέπειες. Κυρίως η Java παρέχει δύο ειδών συλλογών δεδομένων:

- Ανασταλτικές (Blocking) συντρέχουσες συλλογές δεδομένων: Αυτού του είδους οι συλλογές δεδομένων παρέχουν λειτουργίες για την προσθήκη και αφαίρεση δεδομένων, με την προϋπόθεση εάν η συγκεκριμένη λειτουργία δεν μπορεί αμέσως να εφαρμοστεί, ειδικά σε περιπτώσεις άδειας είτε γεμάτης λίστας, τότε το νήμα της εργασίας που προσπάθησε να εκτελέσει την παραπάνω λειτουργία θα ανασταλεί του έως ότου η συγκεκριμένη λειτουργία μπορεί να διεκπεραιωθεί.
- Μη-Ανασταλτικές (Non Blocking) συντρέχουσες συλλογές δεδομένων: Αυτού του είδους οι συλλογές δεδομένων παρέχουν λειτουργίες για την προσθήκη και αφαίρεση δεδομένων, με την προϋπόθεση εάν η συγκεκριμένη λειτουργία δεν μπορεί αμέσως να εφαρμοστεί, ειδικά σε περιπτώσεις άδειας είτε γεμάτης λίστας δεδομένων, τότε το νήμα της εργασίας που προσπάθησε να εκτελέσει την παραπάνω λειτουργία θα επιστρέψει την τιμή null και θα παρουσιάσει μία εξαίρεση. Επίσης το νήμα αυτό δεν θα ανασταλεί..

Μη-ανασταλτικές (non-blocking) λίστες

Η πιο βασική συλλογή και δομή δεδομένων είναι η λίστα. Μια λίστα έχει ένα πεπερασμένο αριθμό στοιχείων που μπορούμε να προσπελάσουμε, να αφαιρέσουμε και να προσθέσουμε νέα σε οποιαδήποτε θέση. Οι συντρέχουσες λίστες επιτρέπουν στα διάφορα νήματα να προσθέτουν είτε να αφαιρούν στοιχεία της χωρίς να παράγουν ασυνέπεια δεδομένων.

Στο παρακάτω παράδειγμα, υλοποιούμε το μοντέλο Παραγωγού Καταναλωτή με τη χρήση των μη-ανασταλτικών λιστών. Οι μη-ανασταλτικές λίστες παρέχουν λειτουργίες οι οποίες εάν δεν εκτελεστούν αμέσως τότε θα παραχθεί μία εξαίρεση και θα επιστραφεί η τιμή `null`. Η Java έχει την κλάση `ConcurrentLinkedDeque` που υλοποιεί μη-ανασταλτικές συντρέχουσες λίστες.

-

```
public class Item {

	private String description;
	private int itemId;

	public String getDescription() {
		return description;
	}

	public int getItemId() {
		return itemId;
	}

	public Item() {
		this.description = "Default Item";
		this.itemId = 0;
	}

	public Item(String description, int itemId) {
		this.description = description;
		this.itemId = itemId;
	}

}
```

```
import java.util.Random;
import java.util.concurrent.ConcurrentLinkedDeque;

public class ProducerNonBlocking implements Runnable {

	private ConcurrentLinkedDeque<Item> deque;

	public ProducerNonBlocking(ConcurrentLinkedDeque<Item> deque) {
		this.deque = deque;
	}

	@Override
	public void run() {
		Random random = new Random(System.currentTimeMillis());
```

```
                String itemName;
                int value;
                try {
                        for (int i = 1; i < 8; i++) {
                                itemName = "Item" + random.nextInt(20);
                                value = random.nextInt(200);
                                deque.add(new Item(itemName, value));
                                System.out.println("The Producer adds a new item:" + itemName
                                                + " with value: " + value);
                                Thread.currentThread().sleep(250);
                        }
                } catch (InterruptedException ex) {
                        ex.printStackTrace();
                }
        }
}
```

```
import java.util.concurrent.ConcurrentLinkedDeque;

public class ConsumerNonBlocking implements Runnable {

        private ConcurrentLinkedDeque<Item> deque;

        public ConsumerNonBlocking(ConcurrentLinkedDeque<Item> deque) {
                this.deque = deque;
        }

        @Override
        public void run() {
                try {
                        Thread.currentThread().sleep(1000);
                } catch (InterruptedException ex) {
                        ex.printStackTrace();
                }
                Item item;
                while ((item = deque.pollFirst()) != null) {
                        if (item == null) {
                        } else {
                                generateOrder(item);
                        }
                }
        }

        private void generateOrder(Item item) {
                System.out.print("The Consumer bought an item ");
                System.out.print(item.getDescription());
                System.out.println(" with value: "+item.getItemId());
                try {
                        Thread.currentThread().sleep(1000);
                } catch (InterruptedException ex) {
                        ex.printStackTrace();
                }
        }
}
```

```
import java.util.concurrent.ConcurrentLinkedDeque;

public class MainProducerConsumerNonBlocking {

	public static void main(String[] args) {

		ConcurrentLinkedDeque<Item> deque = new ConcurrentLinkedDeque<Item>();

		Thread producerThread = new Thread(new ProducerNonBlocking(deque));
		Thread consumerThread = new Thread(new ConsumerNonBlocking(deque));
		producerThread.start();
		consumerThread.start();
	}
}
```

-

```
The Producer adds a new item:Item7 with value: 57
The Producer adds a new item:Item10 with value: 72
The Producer adds a new item:Item2 with value: 144
The Producer adds a new item:Item8 with value: 26
The Consumer bought an item Item7 with value: 57
The Producer adds a new item:Item19 with value: 15
The Producer adds a new item:Item8 with value: 136
The Producer adds a new item:Item9 with value: 177
The Consumer bought an item Item10 with value: 72
The Consumer bought an item Item2 with value: 144
The Consumer bought an item Item8 with value: 26
The Consumer bought an item Item19 with value: 15
The Consumer bought an item Item8 with value: 136
The Consumer bought an item Item9 with value: 177
```

-

Στο παραπάνω παράδειγμα, χρησιμοποιήσαμε το αντικείμενο ConcurrentLinkedDeque το οποίο είναι παραμετροποιημένο ώστε να δέχεται αντικείμενα τύπου Item η οποία προσομοιώνει ένα στοιχείο. Η δομή δεδομένων ConcurrentLinkedDeque είναι μια μη-ανασταλτική δομή. Για αυτό το λόγο πρέπει να προσέξουμε να μην υπάρξουν περιπτώσεις όπου το νήμα της κλάσης του Καταναλωτή να προσπελάσει μια άδεια λίστα, για αυτό το λόγο υπάρχουν έλεγχοι μέσα στο κώδικα της κλάσης του καταναλωτή ώστε να μην βρεθεί σε αυτήν τη θέση και παρουσιάσει μια εξαίρεση (exception).

Πρέπει να επισημανθεί ότι η διαφορά που υπάρχει μεταξύ του συγκεκριμένου κώδικα και εκείνου που αναπτύχθηκε στους αρχικούς μηχανισμούς διαχείρισης συγχρονισμού και ελέγχου του κρίσιμου τμήματος είναι ότι δεν απαιτείται συγγραφή ειδικού κώδικα για αυτό το φαινόμενο. Δηλαδή οι συγκεκριμένες συλλογές δομών δεδομένων κατέχουν ενσωματωμένους μηχανισμούς συγχρονισμού και ελέγχου για προσπέλαση και πρόσβαση σε κρίσιμα τμήματα του κώδικα όπως είναι τα μοιραζόμενα δεδομένα που διατηρούν.

Ανασταλτικές (blocking) λίστες

Στο παρακάτω παράδειγμα θα μελετήσουμε τη χρήση της “ανασταλτικής (blocking)” λίστας. Η κύρια διαφορά μεταξύ “ανασταλτικής (blocking)” και “μη-ανασταλτικής (non-blocking)” λίστας είναι ότι οι “ανασταλτικές” λίστες έχουν μεθόδους για την εισαγωγή και διαγραφή δεδομένων αλλά αν δεν μπορούν να εκτελέσουν τις εντολές τους αμέσως (σε περιπτώσεις όπως διαγραφής σε άδειας λίστας και προσθήκης στοιχείων σε γεμάτη λίστα), τότε αναστέλλουν το νήμα που κάλεσε αυτές τις λειτουργίες έως

ότου μπορεί η λειτουργία αυτή να εκτελεστεί. Η Java παρέχει την κλάση LinkedBlockingDeque ως “ανασταλτική” λίστα. Στο παρακάτω παράδειγμα προσομοιώνουμε ξανά το φαινόμενο μοιραζόμενης πρόσβασης στοιχείων τύπου Item μεταξύ ενός Παραγωγού και Καταναλωτή.

-

```
public class Item {

	private String description;
	private int itemId;

	public String getDescription() {
		return description;
	}

	public int getItemId() {
		return itemId;
	}

	public Item() {
		this.description = "Default Item";
		this.itemId = 0;
	}

	public Item(String description, int itemId) {
		this.description = description;
		this.itemId = itemId;
	}

}
```

```
import java.util.Random;
import java.util.concurrent.LinkedBlockingDeque;

public class ProducerBlocking implements Runnable {

	private LinkedBlockingDeque<Item> deque;

	public ProducerBlocking(LinkedBlockingDeque<Item> deque) {
		this.deque = deque;
	}

	@Override
	public void run() {

		Random random = new Random(System.currentTimeMillis());
		String itemName;
		int value;

		try {
			for (int i = 0; i < 8; i++) {

				//add a new Item - Product
```

```
                                itemName = "Item" + random.nextInt(20);
                                value = random.nextInt(200);

                                //then insert it to the list
                                deque.put(new Item(itemName, value));

                                System.out.println("The Producer adds a new item:" + itemName
                                                + " with value: " + value);
                                Thread.currentThread().sleep(250);
                        }

                        //the producer after some time closes the store
                        Thread.currentThread().sleep(10000);
                        System.out.println("The Produer closes the Store");
                        System.exit(0);


                } catch (InterruptedException ex) {
                        ex.printStackTrace();
                }

        }

}
```

```
import java.util.concurrent.LinkedBlockingDeque;

public class ConsumerBlocking implements Runnable {

        private LinkedBlockingDeque<Item> deque;

        public ConsumerBlocking(LinkedBlockingDeque<Item> deque) {
                this.deque = deque;
        }

        @Override
        public void run() {
                try {
                        Thread.currentThread().sleep(1000);
                } catch (InterruptedException ex) {
                        ex.printStackTrace();
                }
                Item item;
                try {
                        while ((item = deque.take()) != null) {
                                generateOrder(item);
                        }
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }

        }

        private void generateOrder(Item item) {
                System.out.println("A Consumer bought an item "
                                + item.getDescription() + " with value: " + item.getItemId());
```

```
                try {
                        Thread.currentThread().sleep(1000);
                } catch (InterruptedException ex) {
                        ex.printStackTrace();
                }
        }

}
```

```
import java.util.concurrent.LinkedBlockingDeque;

public class MainProducerConsumerBlocking {

        public static void main(String[] args) {

                LinkedBlockingDeque<Item> deque = new LinkedBlockingDeque<Item>();

                Thread producerThread = new Thread(new ProducerBlocking(deque));
                Thread consumerThread = new Thread(new ConsumerBlocking(deque));
                producerThread.start();
                consumerThread.start();

        } //end of main

}
```

-

```
The Producer adds a new item:Item8 with value: 50
The Producer adds a new item:Item18 with value: 60
The Producer adds a new item:Item1 with value: 74
The Producer adds a new item:Item5 with value: 93
The Producer adds a new item:Item16 with value: 146
The Consumer bought an item Item8 with value: 50
The Producer adds a new item:Item17 with value: 132
The Producer adds a new item:Item1 with value: 31
The Producer adds a new item:Item0 with value: 151
The Consumer bought an item Item18 with value: 60
The Consumer bought an item Item1 with value: 74
The Consumer bought an item Item5 with value: 93
The Consumer bought an item Item16 with value: 146
The Consumer bought an item Item17 with value: 132
The Consumer bought an item Item1 with value: 31
The Consumer bought an item Item0 with value: 151
The Produer closes the Store
```

-

Στο παραπάνω παράδειγμα χρησιμοποιήσαμε την κλάση `LinkedBlockingDeque` στην παραμετροποιημένη μορφή της ώστε να επιστρέφει ώστε τα στοιχεία που θα αποθηκεύονται να είναι αντικείμενα τύπου `Item`. Ο Παραγωγός χρησιμοποιεί την μέθοδο `put()` για να εισάγει αντικείμενο τύπου `Item` στη λίστα. Εάν η λίστα είναι γεμάτη τότε η μέθοδος αυτή αναστέλλει το νήμα εκτέλεσης του αντικειμένου του Παραγωγού έως ότου βρεθεί διαθέσιμος χώρος. Σε αντιδιαστολή ο Καταναλωτής χρησιμοποιεί την μέθοδο `take()` για να αφαιρέσει αντικείμενα τύπου `Item`. Η μέθοδος αυτή αναστέλλει την εκτέλεση του νήματος του Καταναλωτή εάν η λίστα είναι άδεια έως ότου τοποθετηθεί κάποιο αντικείμενο

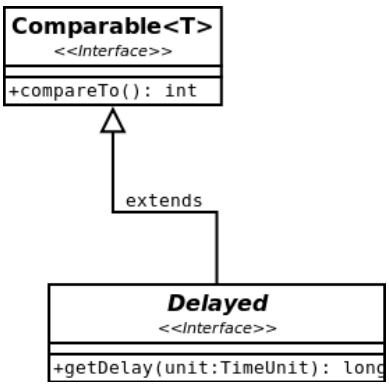
τύπου Item.

Και οι δύο οι παραπάνω μέθοδοι της κλάσης LinkedBlockingDeque παρουσιάζουν μια εξαίρεση τύπου InterruptedException εάν τα διάφορα νήματα που εμπλέκονται στην επεξεργασία των ντικειμένων της κλάσης αυτής διακόψουν την εκτέλεσή τους. Για αυτό το λόγο χρησιμοποιούμε μέσα στο κώδικα τη δομή try...catch.

Χρήση λίστας για ασφαλή συντρέχουσα πολυνηματική πρόσβαση σε δεδομένα που παράγονται με χρονοκαθυστέρηση

Μια ενδιαφέρουσα δομή δεδομένων που παρέχεται από την Java είναι η κλάση DelayedQueue. Σε αυτή τη κλάση μπορούμε να αποθηκεύσουμε στοιχεία τα οποία έχουν ως ιδιαίτερο χαρακτηριστικό (ιδιότητα) την χρονική στιγμή που θα είναι διαθέσιμα προς χρήση. Δηλαδή υπάρχει ένα είδος χρονοκαθυστέρησης . Μέθοδοι της κλάσης που προσπαθούν να προσπελάσουν τα δεδομένα που διατίθενται για μελλοντική πρόσβαση θα απορριφθούν από τη κλάση.

Για να χρησιμοποιήσουμε αυτά τα αντικείμενα και να αποθηκεύουμε δεδομένα στη λίστα τύπου DelayedQueue πρέπει να εφαρμόσουμε (implement) την διασύνδεση Delayed. Αυτή η διασύνδεση μας επιτρέπει να εργαστούμε με αντικείμενα τύπου delayed, ώστε να χρησιμοποιήσουμε δεδομένα ή και αντικείμενα τα οποία θα έχουν μελλοντική καταληκτική ημερομηνία χρήσης και θα αποθηκεύονται σε μια λίστα τύπου DelayedQueue. Ένα ενδεικτικό UML διάγραμμα είναι το εξής:



Η Delayed διασύνδεση υπαγορεύει ρητά και υποχρεωτικά να εφαρμόσουμε (implement) τις παρακάτω μεθόδους:

- compareTo(Delayed o): Η διασύνδεση Delayed επεκτείνει την διασύνδεση Comparable ώστε η συγκεκριμένη μέθοδος να επιτρέπει μια τιμή μικρότερη του μηδενός εάν το αντικείμενο που εκτελεί αυτή τη μέθοδο έχει τιμή - ημερομηνία χρονοκαθυστέρησης μικρότερη από τη τιμή χρονοκαθυστέρησης του αντικειμένου που έχει δοθεί ως όρισμα. Είτε επιστρέφει μια θετική τιμή εάν το αντικείμενο που εκτελεί αυτή τη μέθοδο έχει τιμή - ημερομηνία χρονοκαθυστέρησης μεγαλύτερη από τη τιμή χρονοκαθυστέρησης του αντικειμένου που έχει δοθεί ως όρισμα. Και επιστρέφει την τιμή μηδέν εάν και τα δύο αντικείμενα έχουν την ίδια ακριβώς χρονοκαθυστέρηση.
- getDelay(TimeUnit unit): Αυτή η μέθοδος επιστρέφει τον χρόνο που απομένει έως ότου έλθει ο χρονος ενεργοποίησης για πρόσβαση και προσπέλαση ενός αντικειμένου με χρονοκαθυστέρηση στην πρόσβασή του. Η κλάση TimeUnit χρησιμοποιείται για να οριστεί ο τύπος του χρόνου που θα επιστραφεί. Οι σταθερές που χρησιμοποιούνται είναι: DAYS, HOURS, MICROSECONDS, MINUTES, NANOSECONDS & SECONDS.

Στο παρακάτω παράδειγμα θα μελετήσουμε το τρόπο χρήσης της κλάσης DelayedQueue που αποθηκεύει διάφορα αντικείμενα με την ιδιότητα της χρονοκαθυστέρησης που πρέπει να παρέλθει για να γίνουν ενεργά προς χρήση. Συγκεκριμένα θα προσομοιώσουμε την διαδικασία υποβολής μιας αίτησης. Μία αίτηση θα αναπαρίσταται από την κλάση Application. Κάθε αίτηση υποβάλεται σε ένα “γραφείο” που προσομοιώνεται από την κλάση ApplcationOffice ενώ η κλάση Main υποβάλει αιτήσεις σε διάφορα “γραφεία” και αναμένει το χρόνο που θα εξεταστούν και θα γίνουν δεκτές.

-

```
import java.util.Date;
import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;

public class Application implements Delayed {

        // Declare a private Date attribute named startDate.
        private Date startDate;

        // The constructor of the class.
        public Application(Date startDate) {
                this.startDate = startDate;
        }

        /*
         * Implement the compareTo() method. It receives a Delayed object as its
         * parameter. Return the difference between the delay of the current object
         * and the one passed as parameter.
         */
        @Override
        public int compareTo(Delayed o) {
                long result = this.getDelay(TimeUnit.NANOSECONDS)
                                - o.getDelay(TimeUnit.NANOSECONDS);
                if (result < 0) {
                        return -1;
                } else if (result > 0) {
                        return 1;
                }
                return 0;
        }

        /*
         * Implement the getDelay() method. Return the difference between startDate
         * of the object and the actual Date in TimeUnit received as parameter.
         */
        @Override
        public long getDelay(TimeUnit unit) {
                Date now = new Date();
                long diff = startDate.getTime() - now.getTime();
                return unit.convert(diff, TimeUnit.MILLISECONDS);
        }

}
```

```
import java.util.Date;
import java.util.concurrent.DelayQueue;

public class ApplicationOffice implements Runnable {

	// Declare a private int attribute named id to store a number that
	// identifies this application.
	private int id;

	// Declare a private DelayQueue attribute parameterized with the class
	// named queue.
	private DelayQueue<Application> queue;

	// The constructor of the class.
	public ApplicationOffice(int id, DelayQueue<Application> queue) {
		this.id = id;
		this.queue = queue;
	}

	@Override
	/*
	 * Implement the run() method. First, calculate the date of the application
	 * that will be submited. Then calculate the delay of this application
	 */
	public void run() {
		Date now = new Date();
		Date delay = new Date();
		delay.setTime(now.getTime() + (id * 1000));
		System.out.printf("Application Office: " +
				"an Application #%s is applied by Main thread" +
				", It will be ready in %s¥n", id, delay);

		Application application = new Application(delay);
		queue.add(application);

	}
}
```

```
import java.util.Date;
import java.util.Random;
import java.util.concurrent.DelayQueue;
import java.util.concurrent.TimeUnit;

public class MainApplicationSender {

	public static void main(String[] args) throws Exception {

		// Create a DelayedQueue object parameterized with the Event class.
		DelayQueue<Application> queue = new DelayQueue<Application>();

		// Create an array of five Thread objects to store the applications
		// you're
		// going to apply.
		Thread applicationThread[] = new Thread[5];

		// Create five ApplicationOffice objects, with different IDs.
```

```
                for (int i = 0; i < applicationThread.length; i++) {
                        Random random = new Random();
                        ApplicationOffice applicationSubmit = new ApplicationOffice(random
                                        .nextInt(10), queue);
                        applicationThread[i] = new Thread(applicationSubmit);
                }

                // Launch all the five tasks created earlier.
                for (int i = 0; i < applicationThread.length; i++) {
                        applicationThread[i].start();
                }

                // Wait for the finalization of the threads using the join() method.
                for (int i = 0; i < applicationThread.length; i++) {
                        applicationThread[i].join();
                }

                /*
                 * Write to the console the events stored in the queue. While the size
                 * of the queue is bigger than zero, use the poll() method to obtain an
                 * Event class. If it returns null, put the main thread for 500
                 * milliseconds to wait for the activation of more events.
                 */
                int totalApplications = 0;
                do {
                        int counter = 0;
                        Application application;
                        do {
                                application = queue.poll();
                                if (application != null)
                                        counter++;
                        } while (application != null);
                        if (counter != 0) {
                                System.out.println("---An application is" +
                                                " evaluated and is ready---");
                                totalApplications++;
                        }
                        System.out.printf("At %s %d applications are evaluated" +
                                        " and are ready ¥n",
                                        new Date(), totalApplications);
                        TimeUnit.MILLISECONDS.sleep(500);
                } while (queue.size() > 0);

        }

}
```

-

```
Application Office: an Application #2 is applied by Main thread, It will be ready in Tue May 20 18:24:15 EEST
2014
Application Office: an Application #3 is applied by Main thread, It will be ready in Tue May 20 18:24:16 EEST
2014
Application Office: an Application #6 is applied by Main thread, It will be ready in Tue May 20 18:24:19 EEST
2014
Application Office: an Application #9 is applied by Main thread, It will be ready in Tue May 20 18:24:22 EEST
2014
Application Office: an Application #7 is applied by Main thread, It will be ready in Tue May 20 18:24:20 EEST
2014
```

```
At Tue May 20 18:24:14 EEST 2014 0 applications are evaluated and are ready
At Tue May 20 18:24:14 EEST 2014 0 applications are evaluated and are ready
At Tue May 20 18:24:15 EEST 2014 0 applications are evaluated and are ready
At Tue May 20 18:24:15 EEST 2014 0 applications are evaluated and are ready
---An application is evaluated and is ready---
At Tue May 20 18:24:16 EEST 2014 1 applications are evaluated and are ready
At Tue May 20 18:24:16 EEST 2014 1 applications are evaluated and are ready
---An application is evaluated and is ready---
At Tue May 20 18:24:17 EEST 2014 2 applications are evaluated and are ready
At Tue May 20 18:24:17 EEST 2014 2 applications are evaluated and are ready
At Tue May 20 18:24:18 EEST 2014 2 applications are evaluated and are ready
At Tue May 20 18:24:18 EEST 2014 2 applications are evaluated and are ready
At Tue May 20 18:24:19 EEST 2014 2 applications are evaluated and are ready
At Tue May 20 18:24:19 EEST 2014 2 applications are evaluated and are ready
---An application is evaluated and is ready---
At Tue May 20 18:24:20 EEST 2014 3 applications are evaluated and are ready
At Tue May 20 18:24:20 EEST 2014 3 applications are evaluated and are ready
---An application is evaluated and is ready---
At Tue May 20 18:24:21 EEST 2014 4 applications are evaluated and are ready
At Tue May 20 18:24:21 EEST 2014 4 applications are evaluated and are ready
At Tue May 20 18:24:22 EEST 2014 4 applications are evaluated and are ready
At Tue May 20 18:24:22 EEST 2014 4 applications are evaluated and are ready
---An application is evaluated and is ready---
At Tue May 20 18:24:23 EEST 2014 5 applications are evaluated and are ready
```

-

Στο παραπάνω παράδειγμα υλοποιήσαμε την κλάση `Application`. Αυτή η κλάση έχει μία μοναδική ιδιότητα, τη χρονοκαθυστέρηση για την ενεργοποίηση της πρόσβασή των αντικειμένων της κλάσης και εφαρμόζει (`implement`) τη διασύνδεση `Delayed`, ώστε να αποθηκεύσουμε αντικείμενα τύπου `Application` σε αντικείμενα τύπου `DelayedQueue`.

Η μέθοδος `getDelay()` επιστρέφει τον αριθμό σε msec μεταξύ του τμήματος της χρονοκαθυστέρησης που έχει παρέλθει και την τωρινή ημερομηνία-ώρα. Και οι δύο κλάσεις χρησιμοποιούν αντικείμενα τύπου `Date`.

Η μέθοδος compareTo() επιστρέφει τιμή μικρότερη του μηδενός εάν η χρονοκαθυστέρηση του αντικειμένου που εκτελείται είναι μικρότερη από την χρονοκαθυστέρηση του αντικειμένου που έχει δοθεί ως όρισμα στην μέθοδο. Επιστρέφεται τιμή μεγαλύτερη του μηδενός εάν η χρονοκαθυστέρηση του αντικειμένου που εκτελείται είναι μεγαλύτερη από την χρονοκαθυστέρηση για την ενεργοποίηση του αντικειμένου που έχει δοθεί στην μέθοδο ως όρισμα. Τελικά, επιστρέφει την τιμή μηδέν εάν και οι δύο χρονοκαθυστερήσεις είναι ίσες μεταξύ τους.

Στη κλάση `ApplicationOffice`, κατέχει την ιδιότητα `id`, ώστε όταν ένα `ApplicationOffice` αξιολογεί ένα `Application`, δηλαδή έως ότου παρέλθει η χρονοκαθυστέρησή του και ενεργοποιηθεί για να είναι προσπελάσιμο. Αυτό επιτυγχάνεται στη μέθοδο `run()` της κλάσης `ApplicationOffice` όταν η ίδια δημιουργεί το `Application` που υποβάλλεται και ορίζει η ίδια τη χρονοκαθυστέρηση για την ενεργοποίησή του. Τέλος τοποθετεί το αντικείμενο `Application` στην ουρά τύπου `DelayedQueue`.

Τέλος στη κλάση `main()` δημιουργούμε `ApplicationOffice` αντικείμενα και εκτελούνται από τα αντίστοιχα νήματά τους. Σε τακτά χρονικά διαστήματα η `main` ελέγχει εάν κάποιο `Application` έχει ενεργοποιηθεί η πρόσβασή του χρησιμοποιώντας την μέθοδο `poll()`. Αυτή η μέθοδος ανακτά και διαγράφει το πρώτο αντικείμενο της ουράς. Εάν η ουρά δεν έχει κάποιο αντικείμενο τότε επιστρέφει την τιμή `null`.

Παραγωγή τυχαίων αριθμών σε συντρέχοντα προγράμματα

Στην Java παρέχεται συγκεκριμένη κλάση για την παραγωγή ψευδοτυχαίων αριθμών σε συντρέχοντα προγράμματα. Η κλάση ονομάζεται ThreadLocalRandom και το χαρακτηριστικό της είναι ότι χρησιμοποιείται ως τοπική μεταβλητή σε κάθε νήμα. Κάθε νήμα που χρειάζεται να παράγει τυχαίους αριθμούς έχει και διαφορετική γεννήτρια παραγωγής τυχαίων αριθμών, αλλά όλες αυτές οι γεννήτριες διαχειρίζονται από μία κλάση. Με αυτό το μηχανισμό έχουμε καλύτερη απόδοση από το να χρησιμοποιούμε ένα διαμοιραζόμενο αντικείμενο τύπου Random για την παραγωγή τυχαίων αριθμών σε όλα τα νήματα.

Στο παρακάτω παράδειγμα θα ασχοληθούμε με την κλάση ThreadLocalRandom για να παράγουμε τυχαίους αριθμούς σε μία συντρέχουσα εφαρμογή.

-

```
import java.util.concurrent.ThreadLocalRandom;

public class TaskLocalRandom implements Runnable {

	/*
	 * Implement the constructor of the class. Use it to initialize the
	 * random-number generator to the actual thread using the current() method.
	 */
	public TaskLocalRandom() {
		ThreadLocalRandom.current();
	}

	@Override
	public void run() {
		String name = Thread.currentThread().getName();
		for (int i = 0; i < 10; i++) {
			System.out.printf(
					"I am Thread %s: and generated a new random: %d¥n", name,
					ThreadLocalRandom.current().nextInt(100));
		}
	}

}
```

```
public class MainLocalRandom {

	public static void main(String[] args) {

		Thread threads[] = new Thread[3];

		for (int i = 0; i < 3; i++) {
			TaskLocalRandom task = new TaskLocalRandom();
			threads[i] = new Thread(task);
			threads[i].start();
		}
	}

}
```

```
I am Thread Thread-0: and generated a new random: 55
I am Thread Thread-2: and generated a new random: 54
I am Thread Thread-2: and generated a new random: 10
I am Thread Thread-1: and generated a new random: 53
I am Thread Thread-1: and generated a new random: 82
I am Thread Thread-0: and generated a new random: 34
I am Thread Thread-0: and generated a new random: 35
I am Thread Thread-0: and generated a new random: 68
I am Thread Thread-0: and generated a new random: 56
I am Thread Thread-0: and generated a new random: 58
I am Thread Thread-0: and generated a new random: 77
I am Thread Thread-0: and generated a new random: 39
I am Thread Thread-2: and generated a new random: 16
I am Thread Thread-1: and generated a new random: 41
I am Thread Thread-0: and generated a new random: 4
I am Thread Thread-0: and generated a new random: 84
I am Thread Thread-2: and generated a new random: 0
I am Thread Thread-1: and generated a new random: 72
I am Thread Thread-2: and generated a new random: 47
I am Thread Thread-2: and generated a new random: 99
I am Thread Thread-2: and generated a new random: 45
I am Thread Thread-2: and generated a new random: 48
I am Thread Thread-2: and generated a new random: 92
I am Thread Thread-2: and generated a new random: 45
I am Thread Thread-1: and generated a new random: 72
I am Thread Thread-1: and generated a new random: 28
I am Thread Thread-1: and generated a new random: 58
I am Thread Thread-1: and generated a new random: 90
I am Thread Thread-1: and generated a new random: 33
I am Thread Thread-1: and generated a new random: 53
```

Το χαρακτηριστικό στο παραπάνω παράδειγμα είναι η κλάση `TaskLocalRandom`. Ο κατασκευαστής της κλάσης χρησιμοποιεί την μέθοδο `current()` της κλάσης `ThreadLocalRandom`. Αυτή η μέθοδος επιστρέφει ένα αντικείμενο `ThreadLocalRandom`, ώστε να μπορούμε να παράγουμε τυχαίους αριθμούς χρησιμοποιώντας το αντικείμενο της κλάσης `ThreadLocalRandom`. Ακόμη και αν ένα νήμα που κάνει την κλήση της παραπάνω μεθόδου δεν την χρησιμοποιήσει στον κατασκευαστή της ώστε να συσχετιστεί με ένα αντικείμενο τύπου `ThreadLocalRandom`, όταν κάνει την κλήση για παραγωγή τυχαίων αριθμών από την κλάση `ThreadLocalRandom`, τότε αυτομάτως θα δημιουργηθεί ένα νέο αντικείμενο που θα συσχετιστεί με το νήμα ώστε να παράγονται οι τυχαίοι αριθμοί.

Στη μέθοδο `run()` των νημάτων, κάνουμε κλήση της μεθόδου `current()` για να λάβουμε μια γεννήτρια τυχαίων αριθμών που θα συσχετίζεται με το νήμα που έκανε την κλήση. Με αυτό το τρόπο επιτυγχάνουμε συντρέχουσα εκτέλεση όταν έχουμε μια κοινή γεννήτρια τυχαίων αριθμών για πολλά διάφορα νήματα.

Ατομικές μεταβλητές

Οι ατομικές μεταβλητές στην `Java` παρέχουν έλεγχο συντρέχουσας εκτέλεσης νημάτων σε μεμονωμένες μεταβλητές. Όταν εργαζόμαστε με μια μοιραζόμενη μεταβλητή για πολλά και διάφορα νήματα, σίγουρα θα βρεθούμε αντιμέτωποι με το πρόβλημα της ασυνέπειας δεδομένων της κοινής μεταβλητής.

Για να αποφύγουμε αυτά τα προβλήματα, αλλά και να μη χρησιμοποιήσουμε συνθετότερες δομές συντρέχουσας εκτέλεσης, η `Java` έχει εισάγει τις ατομικές μεταβλητές. Όταν χρησιμοποιείται κοινή μια

ατομική μεταβλητή, τότε η ίδια η κλάση που προέρχεται η μεταβλητή αυτή διαθέτει μηχανισμούς συγχρονισμού και ελέγχου του κρίσιμου σημείου πρόσβασης στη κοινή μεταβλητή από πολλαπλά νήματα με ασφαλή τρόπο και χωρίς περιπτώσεις ασυνέπειας δεδομένων.

Όταν ένα νήμα έχει πρόσβαση σε μία ατομική μεταβλητή, τα υπόλοιπα νήματα που θέλουν να εργαστούν με τη συγκεκριμένη ατομική μεταβλητή θα πρέπει να αναμένουν έως ότου να αποδεσμευτεί η ατομική μεταβλητή. Με άλλα λόγια να γίνει χρήση από κάποιο μηχανισμό συγχρονισμού. Παρόλα αυτά η Java χρησιμοποιεί μια διαφορετική τεχνική που ονομάζεται "Compare and Set". Ο βασικός τρόπος που εκτελείται ο παραπάνω μηχανισμός είναι ότι όταν ένα νήμα έχει πρόσβαση σε μία ατομική μεταβλητή και θέλει με μια λειτουργία - μέθοδο του να τροποποιήσει τη τιμή της, τότε:

1. Αρχικά η συγκεκριμένη λειτουργία λαμβάνει τη τιμή της ατομικής μεταβλητής, τροποποιεί τη τιμή αυτή και την αποθηκεύει σε μία τοπική μεταβλητή.

2. Έπειτα προσπαθεί να αλλάξει τη παλιά τιμή της ατομικής μεταβλητής με τη νέα της τοπικής μεταβλητής. Εάν η παλιά τιμή είναι ακόμη η ίδια με τη νέα, τότε γίνεται η αλλαγή. Διαφορετικά η λειτουργία – μέθοδος ξεκινά ξανά.

Η παραπάνω διαδικασία είναι ( ), δηλαδή τα βήματα 1 και 2 ή εκτελούνται μαζί ή καθόλου. Οι ατομικές μεταβλητές λοιπόν δεν χρησιμοποιούν κλειδώματα ή άλλους μηχανισμούς συγχρονισμού για να προστατέψουν την πρόσβαση στα δεδομένα τους. Όλες οι λειτουργίες τους βασίζονται στη προαναφερθείσα διαδικασία η οποία μας εγγυάται ότι διάφορα και πολλαπλά νήματα μπορούν να διαπραγματευτούν με ατομική μεταβλητή χωρίς να δημιουργούνται ασυνέπειες δεδομένων, όπως επίσης αυξάνεται η απόδοση της εφαρμογής μας, από ότι να είχαμε μια απλή κοινή προστατευμένη με μηχανισμούς συγχρονισμού μεταβλητή.

Η κλάσεις που παρέχονται από τη Java για ατομικές λειτουργίες είναι οι εξής:

1. AtomicLong: Κλάση για ατομικές μεταβλητές τύπου Long.
2. AtomicInteger: Κλάση για ατομικές μεταβλητές τύπου Integer.
3. AtomicBoolean: Κλάση για ατομικές μεταβλητές τύπου Boolean.
4. AtomicReference: Κλάση για ατομικά αντικείμενα τύπου object.

Στο παρακάτω παράδειγμα θα ασχοληθούμε για τη χρήση των ατομικών μεταβλητών προσομοιώνοντας την διαδικασία δοσοληψιών μεταξύ μιας εταιρίας, της τράπεζας και το κοινό λογαριασμό μεταξύ τράπεζας και εταιρίας. Οι λειτουργίες που διαπραγματεύονται τα μέλη της προσομοίωσης είναι η ανάληψη χρημάτων και η κατάθεσή τους. Για αυτό το λόγο θα χρησιμοποιήσουμε την κλάση AtomicLong.

-

```
import java.util.concurrent.atomic.AtomicLong;

public class Account {

        /*
         * Declare a private AtomicLong attribute named balance to store the balance
         * of the account.
         */
        private AtomicLong balance;

        // Implement the constructor of the class to initialize its attribute.
        public Account() {
                balance = new AtomicLong();
```

```
        }

        /*
         * Implement a method named getBalance() to return the value of the balance
         * attribute.
         */
        public long getBalance() {
                return balance.get();
        }

        /*
         * Implement a method named setBalance() to establish the value of the
         * balance attribute.
         */
        public void setBalance(long balance) {
                this.balance.set(balance);
        }

        /*
         * Implement a method named addAmount() to increment the value of the
         * balance attribute.
         */
        public void addAmount(long amount) {
                this.balance.getAndAdd(amount);
        }

        /*
         * Implement a method named substractAmount() to decrement the value of the
         * balance attribute.
         */
        public void subtractAmount(long amount) {
                this.balance.getAndAdd(-amount);
        }

}
```

```
public class Bank implements Runnable {

        // Declare a private Account attribute named account.
        private Account account;

        // Implement the constructor of the class to initialize its attribute.
        public Bank(Account account) {
                this.account = account;
        }

        /*
         * Implement the run() method of the task. Use the subtractAmount() method
         * of the account to make 10 decrements of 1,000 in its balance.
         */
        @Override
        public void run() {
                for (int i = 0; i < 10; i++) {
                        account.subtractAmount(1000);
                }
        }
```

```
}
```

```
public class Company implements Runnable {

	// Declare a private Account attribute named account.
	private Account account;

	// the constructor of the class
	public Company(Account account) {
		this.account = account;
	}

	/*
	 * Implement the run() method of the task. Use the addAmount() method of the
	 * account to make 10 increments of 1,000 in its balance.
	 */
	@Override
	public void run() {
		for (int i = 0; i < 10; i++) {
			account.addAmount(1000);
		}
	}
}
```

```
public class MainCompanyBankAccount {

	public static void main(String[] args) {

		// Create an Account object and set its balance to 1000.
		Account account = new Account();
		account.setBalance(1000);

		// Create a new Company task and a thread to execute it.
		Company company = new Company(account);
		Thread companyThread = new Thread(company);

		// Create a new Bank task and a thread to execute it.
		Bank bank = new Bank(account);
		Thread bankThread = new Thread(bank);

		System.out.printf("Account : Initial Balance: %d¥n", account
				.getBalance());

		// Start the threads.
		companyThread.start();
		bankThread.start();

		/*
		 * Wait for the finalization of the threads using the join() method and
		 * write in the console the final balance of the account.
		 */
		try {
			companyThread.join();
			bankThread.join();
			System.out.printf("Account : Final Balance: %d¥n", account
```

```
                                        .getBalance());
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

-

```
Account : Initial Balance: 1000
Account : Final Balance: 1000
```

-

Το χαρακτηριστικό του παραπάνω παραδείγματος είναι στη κλάση Account. Σε αυτή τη κλάση ορίζουμε την μεταβλητή AtomicLong για να καταχωρούμε το υπόλοιπο του λογαριασμού. Έπειτα δημιουργούμε τις κατάλληλες μεθόδους για ανάληψη και κατάθεση σε λογαριασμό. Επίσης για να δούμε το υπόλοιπο του λογαριασμό χρησιμοποιούμε την μέθοδο getBalance() που χρησιμοποιεί την μέθοδο get() της κλάσης AtomicLong. Για να υλοποιήσουμε τη μέθοδο setBalance() που αλλάζει τη τιμή της ιδιότητας Balance, χρησιμοποιήσαμε τη μέθοδο set() της κλάσης AtomicLong. Για να υλοποιήσουμε τη μέθοδο addAmount() που αυξάνει το υπόλοιπο του λογαριασμού χρησιμοποιούμε τη μέθοδο getAndAdd() της κλάσης AtomicLong που επιστρέφει τη τιμή του υπολοίπου Balance και στη συνέχεια την αυξάνει κατά το ανάλογο ποσό. Τέλος για να υλοποιήσουμε τη μέθοδο subtractAmount() στα πλαίσια της ατομικής μεταβλητής – ιδιότητας Balance, χρησιμοποιήσαμε επίσης τη μέθοδο getAndAdd().

Έπειτα, υλοποιούμε δύο διαφορετικές εργασίες – νήματα

- Τη κλάση Company: που προσομοιώνει μια εταιρία που αυξάνει σε τακτά χρονικά διαστήματα το υπόλοιπο του λογαριασμού.
- Τη κλάση Bank: που προσομοιώνει το τρόπο λειτουργίας της τράπεζας, όπου για διαφόρους λόγους μειώνει το υπόλοιπο του λογαριασμού της εταιρίας.

Η κλάση Main, υλοποιεί όλες τις εργασίες για εκτυπώνει στο τέλος το ανάλογο αποτέλεσμα.

Ατομικοί πίνακες

Κατ' αναλογία με τις ατοιμικές μεταβλητές η Java υποστηρίζει και τους ατομικούς πίνακες. Ως βασικός μηχανισμός συντρέχουσας εκτέλεσης χρησιμοποιείται η τεχνική “compare-and-swap”, που είναι αντίστοιχος με το μηχανισμό “compare-and-set” που εξηγήθηκε παραπάνω για την τροποποίηση της τιμής μιας μεταβλητής. Τα βήματα είναι τα εξής:

1. Πρώτον λαμβάνουμε τη τιμή της πραγματικής μεταβλητής, που χαρακτηρίζεται ως η παλιά τιμή της.
2. Δημιουργούμε μια προσωρινή μεταβλητή όπου θέτουμε με τη νέα τροποποιημένη τιμή που επιθυμούμε.
3. Τέλος αλλάζουμε τη παλιά τιμή της πραγματικής μεταβλητής με τη νέα της προσωρινής, εάν η παλιά τιμή είναι ίση με τη τιμή της προσωρινής μεταβλητής. Η παλιά τιμή μπορεί να είναι διαφορετική από τη πραγματική τιμή της προσωρινής μεταβλητής εάν κάποιο άλλο νήμα έχει ήδη αλλάξει τη τιμή της πραγματικής μεταβλητής. Σε αυτή τη περίπτωση η διαδικασία επαναλαμβάνεται.

Και πάλι τα παραπάνω βήματα είναι αδιαίρετα (atomic). Με αυτό το τρόπο, δεν χρειαζόμαστε συνθετότερους μηχανισμούς συγχρονισμού.

Η Java έχει εισάγει τους ατομικούς πίνακες που παρέχουν ατομικές λειτουργίες για πίνακες ακεραίων ή και long ακεραίων και atomic Reference.

Στο παρακάτω παράδειγμα θα ασχοληθούμε με τη χρήση της κλάσης AtomicIntegerArray για να εργαστούμε με ατομικούς πίνακες.

-

```
import java.util.concurrent.atomic.AtomicIntegerArray;

public class AtomicArrayIncrementer implements Runnable {

	/*
	 * Declare a private AtomicIntegerArray attribute named vector to store an
	 * array of integer numbers.
	 */
	private AtomicIntegerArray vector;

	/*
	 * Implement the constructor of the class to initialize its attribute.
	 */
	public AtomicArrayIncrementer(AtomicIntegerArray vector) {
		this.vector = vector;
	}

	/*
	 * Implement the run() method. Increment all the elements of the array using
	 * the getAndIncrement() method.
	 */
	@Override
	public void run() {
		for (int i = 0; i < vector.length(); i++) {
			vector.getAndIncrement(i);
		}
	}
}
```

```
import java.util.concurrent.atomic.AtomicIntegerArray;

public class AtomicArrayDecrementer implements Runnable {

	/*
	 * Declare a private AtomicIntegerArray attribute named vector to store an
	 * array of integer numbers.
	 */
	private AtomicIntegerArray vector;

	/*
	 * Implement the constructor of the class to initialize its attribute.
	 */
	public AtomicArrayDecrementer(AtomicIntegerArray vector) {
		this.vector = vector;
	}
```

```
        /*
         * Implement the run() method. Decrement all the elements of the array using
         * the getAndDecrement() method.
         */
        @Override
        public void run() {
                for (int i = 0; i < vector.length(); i++) {
                        vector.getAndDecrement(i);
                }
        }
}
```

```
import java.util.concurrent.atomic.AtomicIntegerArray;

public class MainIncrementerDecrementer {

        public static void main(String[] args) {

                /*
                 * Declare a constant named THREADS and assign to it the value 100.
                 */
                final int THREADS = 10;

                System.out.println("Incrementing and decremeting an atomic array "
                                + THREADS + " times¥n");
                waitSomeTime();

                // Create an AtomicIntegerArray object with 10 elements.
                AtomicIntegerArray vector = new AtomicIntegerArray(THREADS);

                // Create an Incrementer task to work with the atomic array
                AtomicArrayIncrementer atomicArrayIncrementer = new AtomicArrayIncrementer(
                                vector);

                // Create a Decrementer task to work with the atomic array
                AtomicArrayDecrementer atomicArrayDecrementer = new AtomicArrayDecrementer(
                                vector);

                // Create two arrays to store 10 Thread objects.
                Thread threadIncrementer[] = new Thread[THREADS];
                Thread threadDecrementer[] = new Thread[THREADS];

                /*
                 * Create and launch 100 threads to execute the Incrementer task
                 */
                for (int i = 0; i < THREADS; i++) {
                        threadIncrementer[i] = new Thread(atomicArrayIncrementer);
                        threadIncrementer[i].start();
                }

                // Wait for the finalization of the threads using the join() method.
                for (int i = 0; i < THREADS; i++) {
                        try {
                                threadIncrementer[i].join();
                        } catch (InterruptedException e) {
                                e.printStackTrace();
```

```
                }
            }

            // print the new incremented atomic vector
            for (int i = 0; i < vector.length(); i++) {
                System.out.println("Vector[" + i + "] : " + vector.get(i));
            }

            System.out.println("¥nIncrement Thread tasks have stopped" +
                    ", decrement Thread tasks "
                    + THREADS + " times ¥nto a atomic array" +
                                " will start¥n");
            System.out.println("Waiting for the decrement Threads");
            waitSomeTime();

            /*
             * Create and launch 100 threads to execute the Decrement task
             */
            for (int i = 0; i < THREADS; i++) {
                threadDecrementer[i] = new Thread(atomicArrayDecrementer);
                threadDecrementer[i].start();
            }

            // Wait for the finalization of the threads using the join() method.
            for (int i = 0; i < THREADS; i++) {
                try {
                    threadDecrementer[i].join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            // print the new incremented atomic vector
            for (int i = 0; i < vector.length(); i++) {
                System.out.println("Vector[" + i + "] : " + vector.get(i));
            }

            System.out.println("¥n----Main: End of the example----");

        } // end of main

        // wait time for the Main thread
        private static void waitSomeTime() {
            try {
                Thread.sleep(3500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

        }

}
```

-

```
Incrementing and decremeting an atomic array 10 times

Vector[0] : 10
Vector[1] : 10
```

```
Vector[2] : 10
Vector[3] : 10
Vector[4] : 10
Vector[5] : 10
Vector[6] : 10
Vector[7] : 10
Vector[8] : 10
Vector[9] : 10

Increment Thread tasks have stopped, decrement Thread tasks 10 times
to a atomic array will start

Waiting for the decrement Threads
Vector[0] : 0
Vector[1] : 0
Vector[2] : 0
Vector[3] : 0
Vector[4] : 0
Vector[5] : 0
Vector[6] : 0
Vector[7] : 0
Vector[8] : 0
Vector[9] : 0

----Main: End of the example----
```

-

Σε αυτό το παράδειγμα ορίσαμε ένα ατομικό και κοινό πίνακα για τα νήματα – εργασίες μας με τύπο AtomicIntegerArray. Εφαρμόσαμε δύο διαφορετικές εργασίες:

- `AtomicArrayIncrementer`: αυξάνει όλα τα στοιχεία του πίνακα κατά μία μονάδα χρησιμοποιώντας τη μέθοδο `getAndIncrement()`.
- `AtomicArrayDecrementer`: μειώνει όλα τα στοιχεία του πίνακα κατά μία μονάδα χρησιμοποιώντας τη μέθοδο `getAndDecrement()`.

Η κλάση `Main` δημιουργεί ένα πίνακα τύπου `AtomicIntegerArray` με στοιχεία που ορίζονται από με τη σταθερή μεταβλητή `THREADS`. Έπειτα εκτελεί τόσες φορές τη τιμή της προαναφερθείσας μεταβλητής τις εργασίες `AtomicArrayIncrementer` για αύξηση της τιμής των στοιχείων κατά μία μονάδα. Στη συνέχεια αναμένει η `Main` την ολοκλήρωση της εκτέλεσης των εργασιών `AtomicArrayIncrementer` και ξεκινά τώρα η μείωση της τιμής των στοιχείων του πίνακα. Αυτό γίνεται με την εργασία `AtomicArrayDecrementer`. Τέλος πρέπει όλα τα στοιχεία του πίνακα να έχουν την τιμή μηδέν.

Επίλογος

Η παρούσα εργασία παρουσιάζει τις βασικές έννοιες του συντρέχοντος προγραμματισμού και τους μηχανισμούς υλοποίησής τους από τη γλώσσα προγραμματισμού Java. Δίνεται ιδιαίτερη έμφαση στην παρουσίαση όλων των εννοιών με βάση συγκεκριμένα παραδείγματα κώδικα, έτοιμα προς εκτέλεση, έτσι ώστε ο αναγνώστης να είναι σε θέση να ελέγξει άμεσα τη κατανόηση των μηχανισμών που παρουσιάζονται.

Λόγω της σημαντικότητας του πεδίου του συντρέχοντος προγραμματισμού στη σύγχρονη Επιστήμη των Υπολογιστών, τόσο στη Java όσο σε άλλες γλώσσες προγραμματισμού αναπτύσσονται διαρκών νέες μέθοδοι και μηχανισμοί υψηλοτέρου επιπέδου. (Δες πχ Microsoft Task Parallel Library, Intel Parallel Studio, OpenMP, Hadoop Map-Reduce, Scala κλπ). Ήδη στη νέα έκδοση της Java που παρουσιάστηκε πρόσφατα έχουν προστεθεί νέοι μηχανισμοί όπως το Project Lambda.

Η παρουσίαση των νέων αυτών εξελίξεων πιθανότατα θα είναι αντικείμενο άλλων εργασιών.

Πηγές

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

http://www.packtpub.com/java-7-concurrency-cookbook/book

http://en.wikipedia.org/wiki/Process_(computing)

http://en.wikipedia.org/wiki/Thread_(computing)

http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html

http://en.wikipedia.org/wiki/Daemon_(computing)

http://stackoverflow.com/questions/2213340/what-is-daemon-thread-in-java

http://en.wikipedia.org/wiki/Semaphore_(programming)

http://www.javacodegeeks.com

http://javarevisited.blogspot.com

http://howtodoinjava.com

http://java.dzone.com

http://en.wikipedia.org/wiki/Taylor_series

http://it.toolbox.com/blogs/lim/how-to-generate-jpeg-images-from-java-41449

Monte Carlo Pi

Markov Chain Monte Carlo

Direct Needle Buffon Pi

http://msdn.microsoft.com/en-us/library/dd460717%28v=vs.110%29.aspx

https://software.intel.com/en-us/intel-parallel-studio-xe

http://hadoop.apache.org/

http://www.scala-lang.org/

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html