BABEȘ-BOLYAI UNIVERSITY
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Java 8 Concurrency

Authors:
Sergiu BREBAN

January 6, 2017

Contents

List of Figures

Chapter 1

General Java Concurrency

1.1 Basic concurrency concepts

Concurrency and parallelism are similar concepts. Concurrency generally refers to a situation when you have more than one task in a single processor with a single core, and the task scheduler from the operating system level quickly switches between tasks, such they appear as running simultaneosly. Parallelism is when you have more than one task that run simultaneously on a different processor, core inside a processor or even a different computer. Another definition shows that parallelism refers to having different instances of the same task running at the same time over different parts of a data set. [2]

1.1.1 Syncronization

Syncronization is defined as the coordination of two or more tasks to get the desired results. It is of two kinds:

- Control syncronization, when on task depends on another task result
- Data access syncronization, when multiple tasks have access to a shared variable and only one can access it at any given time

A **critical section** is a piece of code that is executed by a single task at any time, because it access a shared resource. Mutual exclusion is a mechanism used to make this possible.

From a theoretical point of view, the popular mechanism used to get synchronization in a concurrent system are:

- Semaphore
- Monitor

A piece of code is thread safe if it can be used in concurrent applications without problems, by using a syncronization mechanism, a nonblocking compare and swap primitive or immutable data. An **immutable object** is thread-save, because the value of its attributes cannot be modified after initialisation, so you have to create a new one if you want to change it.

An **atomic operation** is an operation that appers as occcuring simultaneosly to the other tasks, and is implemented using syncronization mechanisms, and an **atomic variable** is a variable whose value is got and set using atomic operations.

The concurent tasks have two methods to communicate with each other. One is **shared memory**, and is used when the tasks read and write data from the same memory area, and **message passing**, used when tasks are running on different computers and nedd to comunicate with each other.

1.1.2 Concurrent applications possible problems

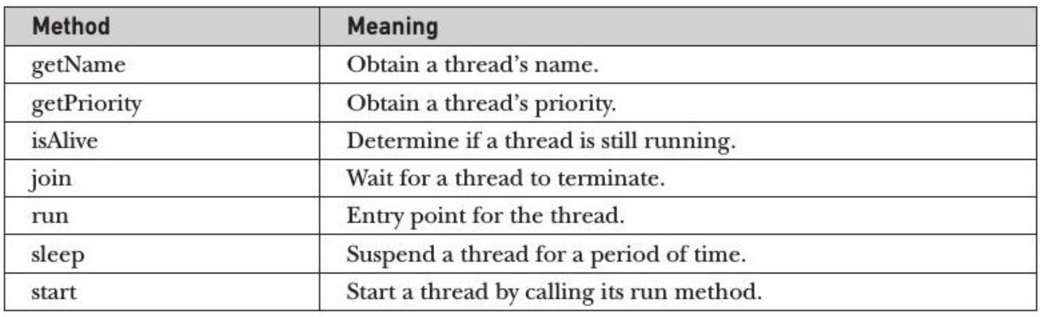
Some of the problems that appear in concurrent applications, caused by the wrong use of syncronization mechanism, are [2]:

- **Data race (race condition)**, when multiple tasks try to write a shared variable without using any syncronization mechanisms
- **Deadlock** occurs when two or more tasks wait for the others to free a shared resource, so they are blocked indefinitely and none will get the resource
- A **livelock** occurs when two tasks are always changing their states due to the actions of the other, they are in a loop, unable to continue
- **Resource starvation** appears when a task never gets the resource it needs to continue, and **fairness** is the solution for that problem
- **Priority inversion** appears when a resource is holden by a low-priority task and a high-priority task needing it cannot continue its execution

1.2 Traditional Java Concurrency Model

1.2.1 The Java Thread Model

Java uses threads to enable its entire environment to be anynchronous, reducing inefficiency by preventing the waste of CPU cycles. This multithreading system is build using the **Thread** class with its methods and the **Runnable** interface. A thread of execution is encapsulated in a **Thread** instance. You have to extend the **Thread** class or to implement the **Runnable** interface to create a new thread. Some of the methods defined in the **Thread** class to help you manage threads are presented in 1.1 [4]



Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

FIGURE 1.1: Thread class methods

1.2.2 The Main Thread

A thread begins running when a Java program starts, and it is usually called the main thread. It is used to spawn other child threads and it is the last thread finishing execution and performs some shutdown actions.

1.2.3 Creating a thread

Implementing Runnable To implement Runnable, a class needs to implement only the **run()** method. Inside this method you define the code that constitues the new thread. The thread will end when run() returns. You have to instantiate a Thread object from within the class that implements Runnable.

```java
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
```

Extending Thread Another way to create a thread is to create a class that extends Thread and instantiate it. The entry point of the new thread is the **run()** method, which has to be overridden. To begin execution of the thread, the new class has to call the start() method.

```java
class NewThread extends Thread {

  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
```

isAlive() and join() You will often want the main thread to finish last. This can be accomplished by calling sleep(), but the most useful methods are isAlive(), whic returns true if the thread is still alive, or false otherwise, and join(), which waits until the thread on which is called terminates.

```
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
ob1.t.join();
ob2.t.join();
ob3.t.join();
```

1.2.4 Thread priorities

The thread scheduler uses thread priorities to decide when a thread should be allowed to run. To set the priority of a thread, we can use the **setPriority()** method from the Thread class. The values must be in the range `MIN_PRIORITY and MAX_PRIORITY`, currently these values being 1 and 10. The current priority of a thread can be obtained using the **getPriority()** method.

1.3 Syncronization

1.3.1 Using syncronized Methods

The easiest syncronization method in Java is to use the monitor associated by each object. To enter this monitor, you have to call all method associated with the **synchronized** keyword. All the other threads that want to call the syncronized method while other thread is inside it have to wait for it to finish.

1.3.2 Using syncronized Statement

To access an object of a class that does not use syncronized methods, you just have to put the calls to that methods inside a syncronized block. The general form of a syncronized block is:

```
syncronized(objRef) {
   //statements to be syncronized
}
```

1.3.3 Using Syncronization Objects

To enable handling of several difficult syncronization situations easily, the following syncronization objects are supported: **Semaphore, CountDownLatch, CyclicBarrier, Exchanger and Phaser**. All these classes are located in the java.util.concurrent package, which defines some core features to support advanced syncronization and interthread communication methods.

Semaphore

CountDownLatch

CyclicBarrier

Exchanger

Phaser

Chapter 2

Java 8: Streams and Lambda expressions

2.1 Definiție

O rețea Bayesiană este un model probabilistic grafic care reprezintă un set de variabile aleatoare și dependețele condiționale dintre ele sub forma unui graf orientat aciclic (DAG).

Fiecare nod al grafului reprezintă o variabilă aleatoare, iar arcele dintre noduri reprezintă dependențele probabilistice dintre nodurile conectate.

Clasificatorul naiv Bayes presupune că toate variabilele sunt independent condiționate, în contrast cu rețelele Bayesiene, care permite condiționarea independentă aplicată la subseturi de variabile.

Aceste modele probabilistice, ca modelul naiv Bayes sau modelele logistice de regresie sunt diferite de alte modele de reprezentare, cum ar fi arborii de decizie, prin faptul că produc estimări probabilistice în loc de clasificări exacte.

Pentru fiecare clasă de valori, estimează probabilitatea ca o instanță dată să aparțină acelei clase. Aceste estimări probabilistice sunt mai utile decât simple predicții, deoarece pot fi clasate, iar costul acestora poate fi minimizat.

2.2 Deducția rețelei Bayesiene

Deducția rețelei Bayesiene se realizează în 3 pași:

- Deducția de variabile neobservate; rețeaua poate fi folosită pentru a oferi informații probabilistice despre relațiile dintre variabile.
- Învățarea probabilităților condiționale; pentru fiecare nod specificarea distribuției probabilităților condiționate de părinții nodului respectiv.
- Învățarea structurii rețelei și construirea grafului.

2.3 Învățarea rețelei Bayesiene

Algoritmul pentru construirea rețelei Bayesiene are două componente: o funcție pentru evaluarea rețelei în funcție de date și o metodă de a genera toate rețelele posibile și a o selecta pe cea mai bună.

După definirea structurii grafului care reprezintă rețeaua, calculul probabilităților condiționale este ușor de realizat, necesitând doar calculul frecvențelor relative a combinațiilor de atribute asociate din setul de date.

2.4 Reprezentarea datelor

Pentru reprezentarea datelor, un format foarte răspândit este formatul ARFF, datele fiind stocate într-un fișier cu formatul .arff. Acest format permite definirea atributelor și a valorilor posibile pentru fiecare atribut, cât și un set de instanțe, cu valori specificate pentru fiecare atribut.

Exemplu de fișier .arff:

```
@relation skiing
@attribute temperature { hot, mild, cool }
@attribute windy { true, false }
@attribute outlook { sunny, overcast }
@attribute snowCover { low, medium, high }
@attribute rainfall { sleet, rain, snow, none }
@attribute ski? { yes, no }
@data
hot, false, overcast, medium, none, no
hot, true, sunny, high, none, no
hot, true, sunny, high, sleet, no
mild, false, sunny, high, none, yes
mild, false, sunny, low, none, no
cool, true, sunny, medium, none, yes
cool, true, overcast, medium, snow, no
```

Acest set de date reprezintă starea unei pârtii de ski, caracterizată de 5 atribute, cu valorile corespunzătoare:
temperature { hot, mild, cool }
windy { true, false }
outlook { sunny, overcast }
snowCover { low, medium, high }
rainfall { sleet, rain, snow, none }

2.5 Algoritm de învățare a structurii

Odată reprezentate datele, avem nevoie de algoritmi specifici pentru a construi și inițializa structura rețelei Bayesiene.

Un exemplu de structură pentru setul de date despre starea pârtiei de ski poate fi observat în Figura 2.1.

Unul dintre cei mai eficienți algoritmi pentru construirea structurii este algorimul K2, iar pentru calcul costului unui graf în cadrul algorimului de căutare, funcția de scor K2, propusă de Cooper and Herskovits (1992). [1]

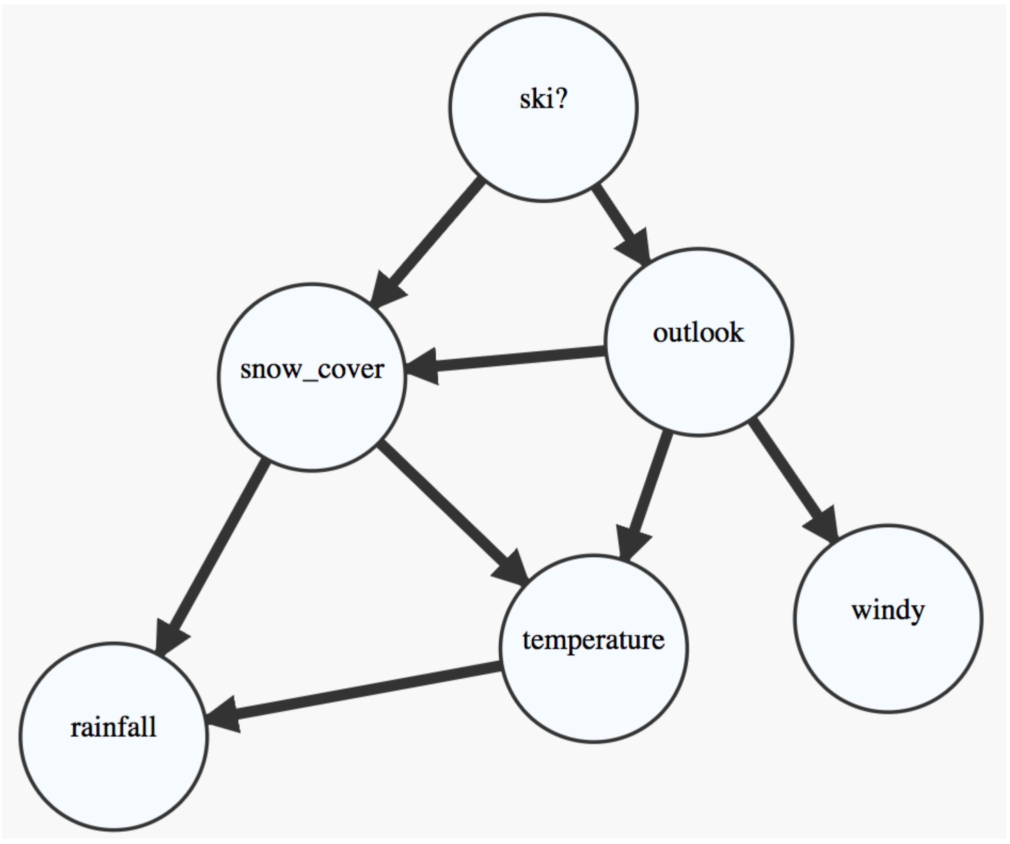


FIGURE 2.1: Strucura unei rețele bayesiene

2.5.1 Algoritmul K2

Algoritmul K2 începe cu o ordine dată a atributelor (noduri) și încearcă să adauge o muchie de la nodurile procesate la cel curent, astfel încât scorul rețelei să fie maxim.

Numărul maxim de părinți poate fi restricționat pentru un nod (la 2 de exemplu) pentru a evita overfitting-ul. În funcția de scor K2 și în calcului tabelelor probabilităților condiționale se poate folosi estimatorul Laplace. [3]

Bibliography

[1] Alexandra M Carvalho. “Scoring functions for learning Bayesian networks”. In: *Inesc-id Tec. Rep* (2009).

[2] Javier Fernandez Gonzalez. *Mastering concurrency programming with Java 8: master the principles and techniques of multithreaded programming with the Java 8 concurrency API*. Community experience distilled. Birmingham: Packt Publ., 2016.

[3] Carolina Ruiz. “Illustration of the K2 algorithm for learning Bayes net structures”. In: *Department of Computer Science, WPI* (2005).

[4] Herbert Schildt and Danny Coward. *Java: The Complete Reference, Ninth Edition*. McGraw-Hill Education, 2014.