CODESPRING

OUR CODE YOUR SUCCESS

Végrehajtási szálak

Runnable, Thread

Simon Károly
simon.karoly@codespring.ro

Végrehajtási szálak Java-ban

CODESPRING
OUR CODE YOUR SUCCESS

- A Java program az operációs rendszer egy folyamatán (process) belül fut. A folyamat adat és kód szegmensekből áll, amelyek egy virtuális címzési térben vannak elhelyezve, és különböző erőforrásokat (memória zónák, fájlok stb.) foglalnak le (melyek a folyamat befejezésekor felszabadulnak).
- A szálak a folyamatok végrehajtó egységei
- Végrehajtási szál: utasítások sora + CPU regiszterek + megfelelő verem
- A szálak a folyamatokhoz hasonlóan viselkednek, ütemezhetőek, az alapvető különbség, hogy egy-egy folyamat címzési tartományán belül helyezkednek el, így bizonyos folyamaton belüli adatok több szál számára láthatóak, ezért szükséges a szálak közötti kommunikáció, a végrehajtási szálak szinkronizálása.
- A Java program indításakor létrejön egy elsődleges szál, amely a későbbiekben további szálakat hozhat létre (és így tovább).
- Azok az applikációk, amelyek több végrehajtási szálon futnak párhuzamosan több feladat végrehajtására képesek.

Végrehajtási szálak létrehozása

CODESPRING
OUR CODE YOUR SUCCESS

- származtatás a java.lang.Thread osztályból
  - `public class Thread extends Object implements Runnable`
- a Runnable interfész megvalósítása(a run() metódust deklarálja, amely így a Thread osztálynak is központi metódusa):
  - ```
    public interface Runnable {
      public abstract void run();
    }
    ```
- A Thread osztály metódusai:
  `start(), run(), sleep(), interrupt(), join(), getName(), setName(), destroy() , isinterrupted()`
- Példa:

```
try {
  Thread.sleep(1000)
} catch(InterruptedException e) {
   //a kivétel kezelése
}
```
```

Szálak létrehozása - példa

CODESPRING
OUR CODE YOUR SUCCESS

```
//Source file: MyThread.java

public class MyThread extends Thread {

  public MyThread(String name) {
       super(name);
  }

  public void run() {
       while(true) {
            try {
                 System.out.println("A szal neve: " + getName());
                 sleep(1000);
            } catch(InterruptedException e) {}
       }
  }


}
```

Szálak létrehozása - példa

```
public class Control {

        private MyThread threads[];

        public Control() {
                threads = new MyThread[2];
                threads[0] = new MyThread("1-es szal");
                threads[1] = new MyThread("2-es szal");
                threads[0].start();
                threads[1].start();
        }

        public static void main(String[] args) {
                Control c = new Control();
        }

}
```

Szálak létrehozása - példa

CODESPRING
OUR CODE YOUR SUCCESS

```
//Source: MyRunnable.java
public class MyRunnable implements Runnable {
      public MyRunnable() { }
      public void run() {
         while(true) {
            System.out.println("A szal neve:" +
                  (Thread.currentThread()).getName() );
            try {
               Thread.sleep(1000);
            } catch(InterruptedException e) {}
         }
      }
}
```

Szálak létrehozása - példa

CODESPRING
OUR CODE YOUR SUCCESS

```
//Source file: Control.java

public class Control {

   public MyRunnable objects[];
   public Thread threads[];

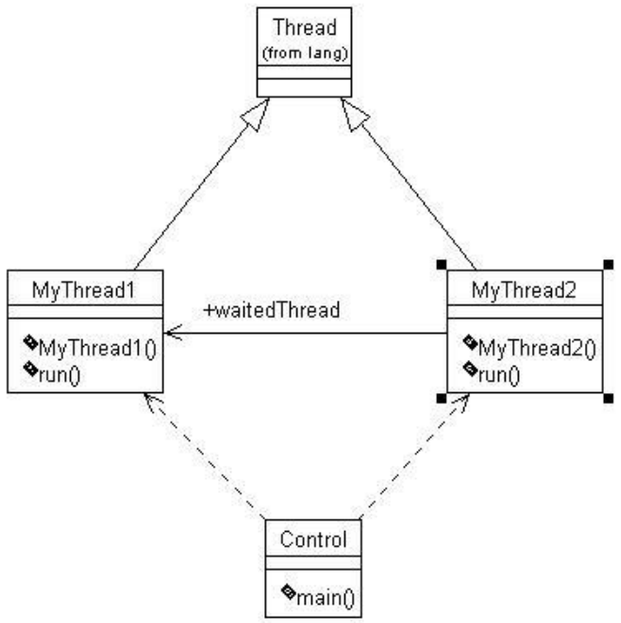
   public Control() {
      objects = new MyRunnable[2];
      objects[0] = new MyRunnable();
      objects[1] = new MyRunnable();
      threads = new Thread[2];
      threads[0] = new Thread(objects[0]);
      threads[1] = new Thread(objects[1]);
      threads[0].start();
      threads[1].start();
   }

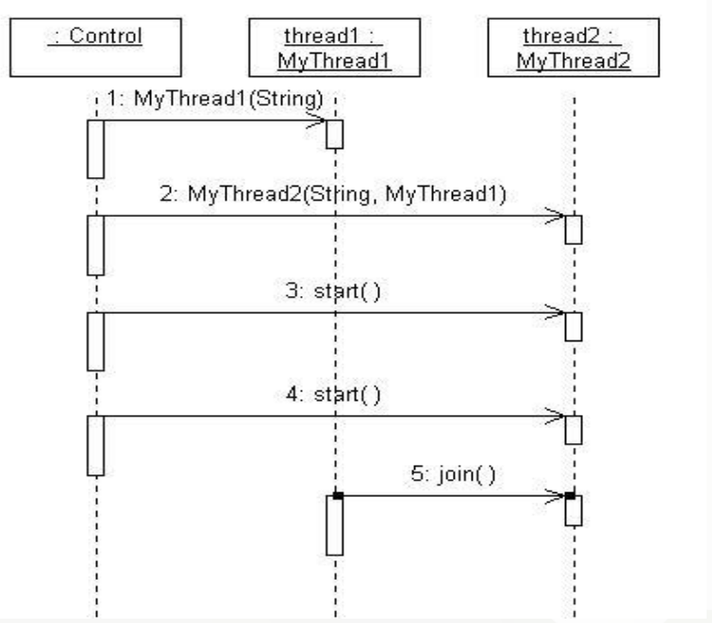
   public static void main(String[] args) {
      Control c = new Control();
   }

}
```

Szálak összekapcsolása

- a join() metódus segítségével, amely lehetővé teszi, hogy egy szál, mielőtt elkezdene műveleteket végezni valamilyen adatokkal, ha szükséges, megvárja, hogy egy másik szál befejezze az illető adatokkal kapcsolatos, már elkezdett műveletek végrehajtását.
- join() – megvárja amíg a másik szál befejezi a műveleteket
- join(int ido) – a megadott ideig vár
- Példa: két szálat hozunk létre, az egyik kiír egy szöveget és vár 5 másodpercet, a másik kiírja a nevét, majd megvárja, amíg az első befejezi a műveletek végrehajtását.





Szálak összekapcsolása

CODESPRING
OUR CODE YOUR SUCCESS

- ```
  //Source file: MyThread1.java

  public class MyThread1 extends Thread {

        public MyThread1(String name) {
          super(name);
        }

        public void run() {
                System.out.println(getName() + "is running");
                for(int i = 0; i < 5; i++) {
                        try {
                                sleep(500);
                        } catch(InterruptedException e){}
                        System.out.println(getName()+" writes " + i);
                }
        }

  }
  ```
  ```

Szálak összekapcsolása

CODESPRING
OUR CODE YOUR SUCCESS

- ```
  //Source file: MyThread2.java

  public class MyThread2 extends Thread {

    public MyThread1 waitedThread;

    public MyThread2(String name, MyThread1 waitedThread){
        super(name);
        this.waitedThread = waitedThread;
    }

    public void run() {
      System.out.println(getName()+" waits for Thread " +
                waitedThread.getName());
      try {
       waitedThread.join();
      } catch(InterruptedException e) {}
      System.out.println(waitedThread.getName() + " has been finished");
      System.out.println(getName() + " has been finished.");
    }

  }
  ```
  ```

Szálak összekapcsolása

CODESPRING
OUR CODE YOUR SUCCESS

- ```
  //Source file: Control.java

  public class Control {

    public static void main(String args[]) {
      MyThread1 thread1 = new MyThread1("First Thread");
      MyThread2 thread2 = new MyThread2("Second Thread", thread1);
      thread1.start();
      thread2.start();
    }

  }
  ```
  ```

Szálak prioritása

CODESPRING
OUR CODE YOUR SUCCESS

- Minden szálhoz egy adott prioritási szint rendelhető hozzá (1-től 10-ig), a setPriority() metódus segítségével (alapértelmezett érték: 5):
  - Thread.MIN_PRIORITY értéke 1
  - Thread.NORM_PRIORITY értéke 5
  - Thread.MAX_PRIORITY értéke 10
- Démon szálak (Daemon Threads): a démon folyamatokhoz hasonló alacsony prioritású szálak, amelyek a háttérben hajtanak végre műveleteket, és automatikusan megsemmisülnek a többi (nem démon) szál leállásakor (gyakorlatilag akkor kapnak processzor időt, amikor más szálak éppen nem tartanak igényt rá)
  - egy szál démon szálba alakítható a setDaemon() metódussal
  - a Java garbage collector is egy démon szál

Szinkronizálás

CODESPRING
OUR CODE YOUR SUCCESS

- Szinkronizálás szükségessége, példa: egy végrehajtási szál kiolvas egy adatstruktúrából egy elemet, hogy elvégezzen vele bizonyos műveleteket, így megváltoztatva annak értékét. Közben egy másik szál is ugyanezt tenné, de amikor kiolvassa az elemet, annak még a régi értékét kapja. A másik szál a műveletek elvégzése után, az eredményeknek megfelelően frissíti az elem értékét, de ezt a frissítést felülírja a második szál saját számításainak végrehajtása után. A végeredmény egy hibás érték.
- Megoldás: az adat zárolása. A második szálnak nem szabadna hozzáférnie az illető elemhez, amíg az első be nem fejezi számításait.
- Több közismert feladat (a vacsorázó filozófusok problémája, az alvó borbély problémája, a dohányzók problémája, a gyártó-fogyasztó probléma stb.), több tervezési minta.
- Két közismert módszer: az E.W. Dijkstra által bevezetett szemaforok alkalmazása, és a C.A.R. Hoare által bevezetett monitorok alkalmazása.
- Java-ban mindkét módszer alkalmazása lehetséges, de általában monitorokat alkalmazunk (ez az alapmechanizmus).

Monitorok

CODESPRING
OUR CODE YOUR SUCCESS

- Monitorok: olyan objektumok, amelyeket több szál is biztonságosan használhat. Fő jellemzőik, hogy metódusaik meghívása a kölcsönös kizárás elvén (mutual exclusion) alapszik, azaz egyszerre csak egyetlen szál férhet hozzá ezekhez a metódusokhoz. Ennek megvalósítása zárak (lock/mutex) segítségével történik.
- A monitorok támogatják a várakozás-értesítés (wait-notify) mechanizmust.
- A gyakorlati megvalósítás Java-ban szinkronizált metódusok (synchronized methods), illetve szinkronizált programblokkok (synchronized statements) alkalmazásával lehetséges.
- Azokat a metódusokat, amelyek törzsében kritikus erőforrásokhoz férünk hozzá, szinkronizáltként határozzuk meg:
- `public synchronized void mySynchronizedMethod()`
- Minden objektumhoz hozzárendelődik egy-egy monitor (monitor, intrinsic lock, monitor lock). Amennyiben egy szál meghív egy szinkronizált metódust, lefoglalja az objektum monitorát és ettől kezdve az objektum zárolt (locked) állapotba kerül. Mindaddig, amíg a monitor fel nem szabadul, más végrehajtási szálak nem férhetnek hozzá az objektum szinkronizált metódusaihoz.

Szinkronizált metódusok

- ```
  public class Counter {
        private long c = 0;
        public synchronized void inc() {
              c++;
        }
  }
  ```
- Ha egy osztály tartalmaz egy szinkronizált metódust, és azt a metódust az osztály egy példányára mutató referencia segítségével egy végrehajtási szál meghívja, lefoglalva az objektum monitorát, akkor a többi szál által egyetlen más szinkronizált metódus sem lesz meghívható a monitor felszabadításáig.
- Ez több esetben hátrányt jelenthet. Például, gondoljunk arra, hogy mi történne, ha az előző példánkat kiegészítenénk olyan módon, hogy két számlálót kezeljen. Bevezetnénk még egy attribútumot, és még egy metódust, amely ennek az értékét növelné. Mivel a metódusok szinkronizáltak, az egyik meghívása az objektum teljes blokkolását eredményezné, így ha egy második szál a másik számlálót szeretné növelni, ezt nem tehetné meg. Mivel a két számláló független egymástól, a teljes blokkolás fölösleges.
- A megoldást a szinkronizált kódblokkok alkalmazása jelenti.
  ```

Szinkronizált programblokkok

CODESPRING
OUR CODE YOUR SUCCESS

- ```
  public class Counter {
        private long c1 = 0;
        private long c2 = 0;
        private Object lock1 = new Object();
        private Object lock2 = new Object();
        public void inc1() {
              synchronized(lock1) {
                    c1++;
              }
        }
        public void inc2() {
              synchronized(lock2) {
                    c2++;
              }
        }
  }
  ```
- Megjegyzés: ha a lock referenciák helyére this-t írtunk volna, a szinkronizált metódusok használatával azonos hatást értünk volna el.
  ```

Felmerülő problémák, hibalehetőségek

CODESPRING
OUR CODE YOUR SUCCESS

- **Holtpont (deadlock):** akkor történik meg, amikor két szál kölcsönösen várakozik a másikra, és emiatt egyik sem haladhat tovább, a program futása holtponthoz érkezik. A leggyakoribb. A helyzetet az a példa illusztrálhatja, amikor két illemtudó személy egyszerre ér oda egy bejárathoz. Ameddig az etikett alapján, a kort és nemet figyelembe véve el tudják dönteni, hogy ki lépjen be elsőnek nincsen probléma. Baj akkor van, ha ilyen jellegű különbségek nincsenek, vagy nem nyilvánvalóak. Mindketten a másik belépésére fognak várni, és így az ajtó előtt, helyben maradnak.
- **Livelock:** olyankor történhet meg, amikor két végrehajtási szál kölcsönösen reagál a másik viselkedésére, „mozdulataira”. Amennyiben a másik cselekvése szintén egy ilyen reakció, megtörténhet, hogy olyan patthelyzetbe kerülnek, amikor egyik sem folytathatja tovább tevékenységét. A helyzetet az a példa illusztrálhatja, amikor két előzékeny személy szembetalálkozik egy szűk folyosón. Mindkettő kitér, hogy elengedje a másikat, de így ismét egymással szembe kerülnek. Ha ezt fogják ismételgetni, megrekednek, egyik sem haladhat keresztül.
- **Kiéheztetés (starvation):** akkor fordul elő, mikor egy „mohó” szál hosszú időn keresztül foglalja egy erőforrás monitorát, és így nem enged hozzáférést másoknak.

Wait-notify mechanizmus

CODESPRING
OUR CODE YOUR SUCCESS

- Gyakran előfordulhat olyan eset, amikor egy végrehajtási szál lefoglal egy monitort, de valamilyen okból kifolyólag még nem végezheti el a feladatai. Például, előzőleg szükséges lehet, hogy más szálak még elvégezzenek bizonyos műveleteket. A wait-notify mechanizmus lényege, hogy az ilyen esetekben a szál átadhatja a monitort más szálnak, és egy várakozási állapotba léphet. A másik szál miután befejezi feladatait, értesíti a várakozó szálakat, átadva a felszabadított monitort.
- A Java nyelvben a megvalósításra az Object ősosztály wait, notify és notifyAll metódusai adnak lehetőséget.
- Ahhoz, hogy egy szál meghívja ezeket a metódusokat egy adott objektumra, birtokolnia kell annak monitorát. A megoldás, hogy a metódusokat csak szinkronizált metódusokon, vagy programblokkokon belül hívjuk meg.
- Amennyiben egy végrehajtási szál meghívja a wait metódust, átengedi a monitort más szálaknak. A döntés meghozatalának alapja általában védett programblokkok (guarded blocks) alkalmazása.

Wait-notify mechanizmus

CODESPRING
OUR CODE YOUR SUCCESS

- Védett programblokkok alkalmazása: a szál mielőtt belekezdene saját feladatainak elvégzésébe, ellenőriz egy adott feltételt, például egy adott kontrollváltozó értékét. Ilyen módon információt szerez arról, hogy az illető műveletek végrehajthatóak-e, vagy még várakozni kell, például arra, hogy előzetesen más szálak még elvégezzenek bizonyos feladatokat. Amennyiben várakoznia kell, a szál meghívja a wait metódust, és átadja a monitort. Miután az a szál, amelyik az előkészítő műveletek elvégzéséért felelős, elfoglalja a monitort, elvégzi az illető műveleteket, módosítja a megfelelő kontrollváltozó értékét, hogy a feltétel teljesüljön, majd meghívja a notifyAll metódust, ezzel értesítve a monitorra várakozó szálakat.
- A notifyAll metódus minden várakozó szálat értesít. Általában ezt a megoldást alkalmazzuk, de helyette az Object osztály notify metódusa is meghívható. Ez a metódus a várakozó szálak közül csak egyetlen szálat fog értesíteni. A „szerencsés” szál kiválasztása nem determinisztikus módon történik, nem tudjuk meghatározni, hogy kit értesítünk. A metódus alkalmazása inkább olyan helyzetekben lehet jó megoldás, amikor nagyon sok végrehajtási szál egymáshoz hasonló műveleteket végez, és tulajdonképpen nem számít, hogy ezek közül melyiket értesítjük.

Wait-notify mechanizmus

CODESPRING
OUR CODE YOUR SUCCESS

- A notifyAll metódus hatása, hogy a monitorra várakozó szálak visszatérnek a wait metódusból, újra futtatható állapotba kerülnek. Ettől függetlenül nem rendelkeznek prioritással más szálakkal szemben, tulajdonképpen csak újra részeseivé válnak a monitor lefoglalásáért folyó versenynek. Következményként, amikor a későbbiekben megszerzik a monitort, újra kell ellenőrizniük a feltételt, meggyőződve arról, hogy az adott pillanatban elvégezhetőek-e a műveletek. Az újraellenőrzés szükségessége azért is természetes, mert egy adott objektum esetében különböző szálak különböző programrészeken belül, különböző okokból hívhatják meg a notifyAll metódust. Az értesítés tulajdonképpen csak azt jelzi, hogy a „feladó” szál elvégzett bizonyos műveleteket, és átengedi a monitort. Nem biztos, hogy pontosan azokat a műveleteket végezte el, amelyekre az értesítés hatására várakozási állapotából visszatért szál várt. A fenti okok miatt a védett programblokkok esetében a feltétel ellenőrzése mindig egy cikluson belül történik.

Wait-notify mechanizmus

CODESPRING
OUR CODE YOUR SUCCESS

- ```
  public synchronized void guardedCode() {
      while(!condition) {
          try {
              wait();
          } catch (InterruptedException e) {}
      }
      …
  }
  ```

- ```
  public synchronized void waitedOperation() {
      …
      condition = true;
      notifyAll();
  }
  ```

Érvénytelenített (deprecated) metódusok

CODESPRING
OUR CODE YOUR SUCCESS

- Egy végrehajtási szál futása akkor áll le, amikor a szál befejezi feladatait, visszatér a run metódusból. Előfordulhatnak olyan esetek, amikor szeretnénk hamarabb leállítani a szálat, vagy szeretnénk felfüggeszteni a futását, majd a későbbiekben folytatni a végrehajtást a felfüggesztés pillanatától. A Thread osztályon belül találhatunk erre a célra használható metódusokat (stop, suspend és resume). A probléma az, hogy, ha megpróbáljuk ezeket alkalmazni, a fordító figyelmeztetni fog (warning), mivel érvénytelennek (deprecated) vannak nyilvánítva.
- A Java fejlődése során gyakran előfordult, hogy egy kiadott verzió valamilyen osztálya biztosított olyan metódusokat, amelyeket a későbbiekben (egy következő verzió megjelenésekor) érvénytelennek nyilvánítottak. A stratégia célja, hogy megőrizzék a kompatibilitást: nem törölhették egyszerűen az adott metódusokat az új kiadásból, mivel akkor az előzőleg megírt programok nem működnének az új platformon. Megoldásként ezeket a problémás metódusokat benne hagyták az új verziókban is, de érvénytelennek nyilvánították, így a fordító csak figyelmeztet, de lefordítja a programot.

Érvénytelenített (deprecated) metódusok

CODESPRING
OUR CODE YOUR SUCCESS

- Ha egy metódust érvénytelennek nyilvánítottak, arra valószínűleg jó okuk volt. Általában olyan problémás metódusokról van szó, amelyek potenciális hibaforrások lehetnek. Ennek megfelelően az általános szabály az, hogy kerüljük ezeknek a metódusoknak a használatát, mivel biztos létezik jobb, biztonságosabb megoldás. A legtöbb esetben az osztályok specifikációján belül (ahol szintén fel van tüntetve az érvénytelenítés ténye) javaslatokat is találunk az érintett metódusok helyettesítésére, és a modern fejlesztői környezetek szintén útmutatást adhatnak.
- A **stop** metódus használata azért nem javasolt, mert meghívásának következményeként a szál által foglalt monitorok azonnal felszabadulnak. Ez akkor is megtörténik, ha a szál még nem fejezte be feladatait (például, a blokkolt adatoknak csak egy részét dolgozta fel), és ebben az esetben könnyen előfordulhat, hogy a leállítás hibás adatokat, és hibás működést eredményez.
- A **suspend** metódus használata holtpont/kiéheztetés előidézéséhez vezethet. Ha használatával felfüggesztjük egy szál futását, az általa foglalt monitorok nem szabadulnak fel, és más szálak nem férhetnek hozzá a blokkolt objektumokhoz. A **resume** metódus érvénytelenítése ennek következménye.

Érvénytelenített metódusok újradefiniálása

CODESPRING
OUR CODE YOUR SUCCESS

```
public class ThreadExample implements Runnable {
    private volatile Thread control;
    public void start() {
      control = new Thread(this);
      control.start();
    }
    public void stop() {
      control = null;
    }
    public void suspend() {}
    public void resume() {}
    public void run() {
      Thread thisThread = Thread.currentThread();
      while (thisThread == control) {
                try{
                          Thread.sleep(1000);
                } catch (InterruptedException ex) {}
                System.out.println("running");
      }
      System.out.println("finished");
    }
}
```

A stop metódus

- A megoldás lényege: a szálon belül egy kontrollváltozót alkalmazunk, és a szál futását ennek értékéhez kötjük. Az osztályunkon belül a megfelelő módon implementáljuk a stop metódust, úgy, hogy annak meghívása a kontrollváltozó értékének megváltoztatásához vezessen. Mielőtt a run metóduson belül használt ciklus következő iterációjába lépnénk, ellenőrizzük ennek a változónak az értékét.
- Ahhoz, hogy megfelelően működjön a megoldás, szükséges, hogy a szál minden használatkor (hozzáféréskor) ellenőrizze a változó értékét, kivédve azt a lehetőséget, hogy más szálak változtatásai problémához vezessenek. Ez egyszerűen megoldható, a változót volatile típusmódosítóval láthatjuk el. Egy másik megoldás az lehetne, ha a változó értékét csak szinkronizált programblokkon belül tennénk megváltoztathatóvá.
- Mi történik, ha a szál osztályon belül egy nagyobb értéket adunk meg a sleep metódus paramétereként (pl. tíz másodpercet). Miután elindítottuk a szálat, és megpróbáltuk leállítani, a tényleges leállás nem fog megtörténni, csak miután a szál visszatért a sleep metódusból. Hasonló lenne a helyzet, ha a szál várakozási állapotban lenne.

Interrupt

- A megoldást az interrupt metódus alkalmazása jelenti. A metódus meghívásának hatására a sleep, wait vagy join metódusok azonnal visszatérnek. A metódus meghívás egy alvó vagy várakozó szál esetében InterruptedException típusú kivételt eredményez, és ezt a megfelelő módon kezelnünk kell.

```
public void run() {
        Thread thisThread = Thread.currentThread();
        while (thisThread == control){
              try {
                    Thread.sleep(5000);
              } catch (InterruptedException ex) {
                    break;
              }
              System.out.println("running");
        }
        System.out.println("finished");
}
```

- `public void stop() {`

```
        Thread tmp = control;
        control = null;
        tmp.interrupt();
}
```

- az interrupt metódus meghívása nem vezet a szál leállításához. Csak egy állapotváltozó (flag) értéke módosul. Az érték lekérdezhető az isInterrupted nevű metódus segítségével (egy példány esetében), vagy a Thread osztály interrupted nevű statikus metódusának segítségével. Az első metódus nem változtat a változó értékén, csak visszatéríti azt. A második metódus meghívása false értékre módosítja az állapotváltozót (az előző értéket térítve vissza). A feladat, hogy a szál leállítását, futásának megszakítását, a változó értékét figyelembe véve megfelelő módon megvalósítsa, a programozóra hárul.

Suspend és resume metódusok

CODESPRING
OUR CODE YOUR SUCCESS

- Megjegyzés: amennyiben egy szál blokkolt állapotban van, például, mert egy I/O művelet esetében bejövő adatokra vár, az interrupt metódus nem „zökkenti ki” ebből az állapotból. Az ilyen esetekre nincs általános recept, a konkrét helyzetnek megfelelő speciális megoldást kell alkalmaznunk. Például interrupt metódushívás helyett lezárhatjuk az I/O kapcsolatot.
- A suspend és resume metódusok helyettesítéséhez a wait-notify mechanizmust alkalmazhatjuk.
- Bevezetünk még egy kontrollváltozót, amely a threadSuspended nevet kapja, és boolean típusú. A szál indításakor értékét false-ra állítjuk. A myThread referenciához hasonlóan ennek a változónak az esetében is volatile típusmódosítót alkalmazunk. A suspend metóduson belül egyszerűen true-ra módosítjuk az értéket.
- A while ciklusunkat kiegészítjük, olyan módon, hogy minden iteráció előtt ellenőrizze a változó értékét, és amennyiben az true, egy szinkronizált programblokkon belül mindaddig várakozzon, amíg az érték nem módosul. A várakoztatásra a wait metódust használjuk.
- A resume metódus egy szinkronizált programblokkon belül false-ra állítja a változó értékét, majd a notifyAll metódus segítségével értesíti a várakozó szálakat.

Suspend és resume metódusok

CODESPRING
OUR CODE YOUR SUCCESS

```
public class ThreadExample implements Runnable {
   private volatile Thread control;
   private volatile boolean threadSuspended;
   public void start() {
     control = new Thread(this);
     threadSuspended = false;
     control.start();
   }
public void stop() {
  Thread tmp = control;
  control = null;
  tmp.interrupt();
}
public void suspend() {
  threadSuspended = true;
}
public void resume(){
  synchronized (this) {
            threadSuspended = false;
            notifyAll();
   }
}
```

Suspend és resume metódusok

```
public void run() {
  Thread thisThread = Thread.currentThread();
  while (thisThread == control) {
            try {
                        if (threadSuspended) {
                                    synchronized (this) {
                                                while(threadSuspended)
                                                            wait();
                                    }
                        }
                        Thread.sleep(1000);
                } catch (InterruptedException ex) {
                        break;
                }
                System.out.println("running");
      }
      System.out.println("finished");
   }
}
```

Példa - GUI

CODESPRING
OUR CODE YOUR SUCCESS

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
public class ControlFrame extends JFrame {
    private JButton startButton;
    private JButton stopButton;
    private JButton suspendButton;
    private JButton resumeButton;
    private JPanel contentPane;
    private ThreadExample myThread;

    public ControlFrame() {
      setTitle("Thread example");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      contentPane = new JPanel();
      startButton = new JButton("Start");
      startButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                          validateControls(false, true, false, true);
                          myThread = new ThreadExample();
                          myThread.start();
                }
      });
      contentPane.add(startButton);
```

Példa - GUI

CODESPRING
OUR CODE YOUR SUCCESS

```
stopButton = new JButton("Stop");
stopButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                        validateControls(true, false, false, false);
                        myThread.stop();
            }
});
contentPane.add(stopButton);
suspendButton = new JButton("Suspend");
suspendButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                        validateControls(false, true, true, false);
                        myThread.suspend();
            }
});
contentPane.add(suspendButton);
resumeButton = new JButton("Resume");
resumeButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                        validateControls(false, true, false, true);
                        myThread.resume();
            }
});
contentPane.add(resumeButton);
validateControls(true, false, false, false);
setContentPane(contentPane);
}
```

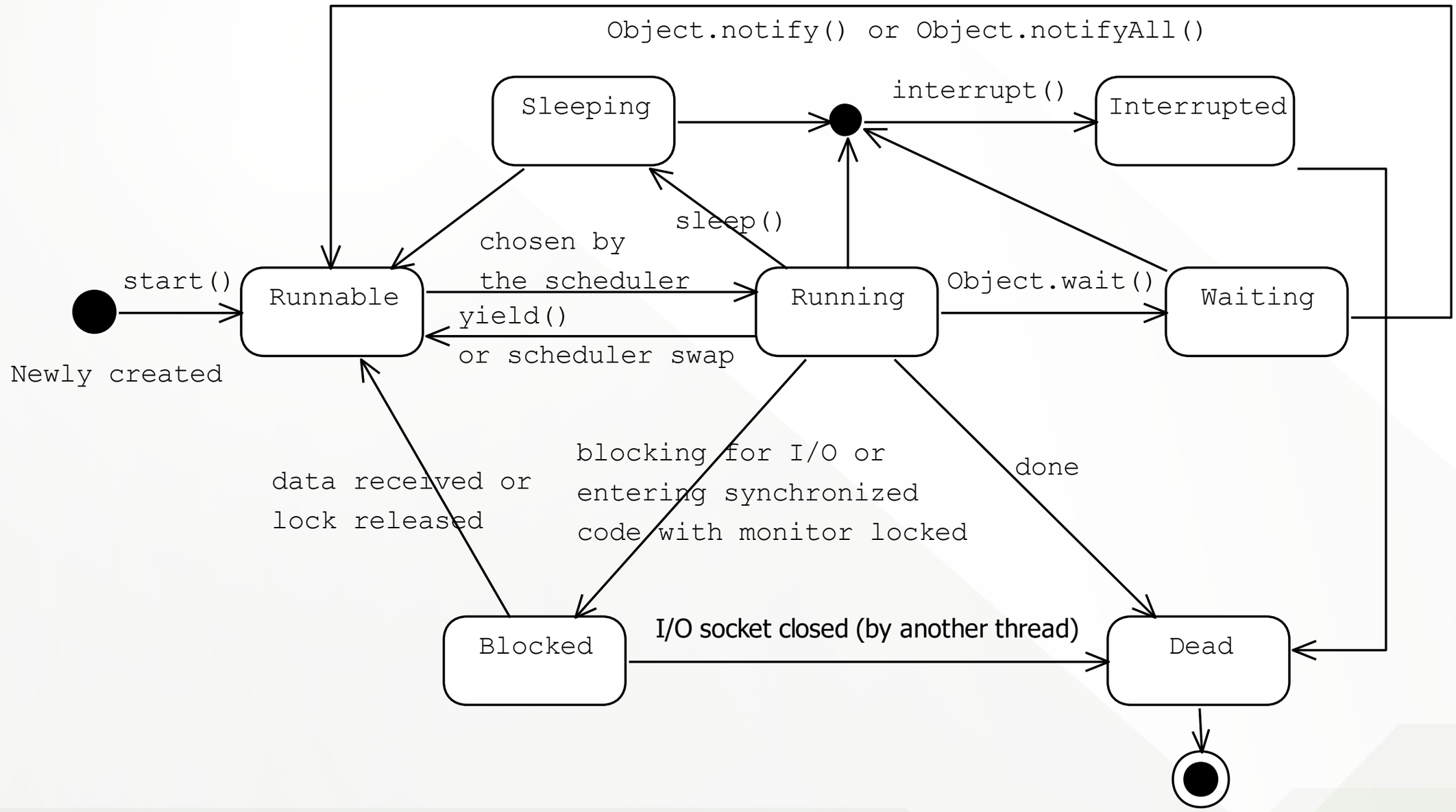
Példa - GUI

```
private void validateControls(boolean start, boolean stop,
                              boolean resume, boolean suspend) {
  startButton.setEnabled(start);
  stopButton.setEnabled(stop);
  resumeButton.setEnabled(resume);
  suspendButton.setEnabled(suspend);
 }

 public static void main(String[] args) {
  ControlFrame te = new ControlFrame();
  te.setBounds(50,50,400,75);
  te.setVisible(true);
 }

}
```

Szálak állapotai



Feladat

- Készítsünk egy egyszerű „autóverseny szimulátort”. Egy SWING eszköztár segítségével megvalósított grafikus felületen belül rajzoljunk ki egy versenypályát, és azon helyezzünk el néhány autót (az autókat egyszerű téglalapok segítségével is ábrázolhatjuk, de kis állományból beolvasott képeket is felhasználhatunk). Arra is lehetőséget adhatunk, hogy a felhasználó meghatározhassa az autók számát (pl. egy szövegmező segítségével). A verseny egy gomb lenyomására indul. Minden autót egy külön végrehajtási szál „irányít”, véletlenszerű időközönként változtatva azok pozícióját (a változtatás mértéke is lehet véletlenszerű). A verseny addig tart, ameddig az első autó beér a célvonalba. Készítsünk egy olyan változatot is, amelynek esetében minden autó beér a célba, majd egy párbeszédablak jelenik meg a beérkezés sorrendjével.