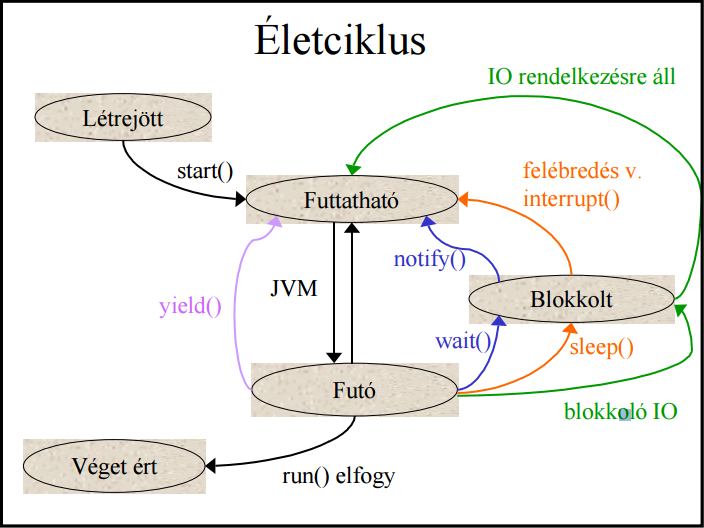
if(a < b && a++ < 10){

Logikai és műveletnél ha az első kiértékelés hamis(tehát a nem kisebb mint b),akkor a többi ki sem értékelődik.

}



**Szálkezelés:**

**package** pack;

**import** java.util.Timer;

**import** java.util.TimerTask;

**public** **class** Remind {

Timer timer;

**public** Remind(**int** seconds){

timer = **new** Timer();

timer.schedule(**new** RemindTask(), seconds \* 1000);

}

**class** RemindTask **extends** TimerTask{

@Override

**public** **void** run() {

System.***out***.println("Time is out!!!!");

timer.cancel();

}

}

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

**new** Remind(15);

System.***out***.println("Munka ütemezve");

}

}

A program kimenete,hogy kiírja a konzolra,hogy Munka ütemezve, majd 15 másodperc múlva azt írja ki ,hogy Time is out!!!!

A run() metódus tartalmazza a szál végrehajtási kódját.A schedule() metódus az időzítő,ami két paramétert vár, az első a TimerTask objektum, a másik az idő ezredmásodpercben.

Másik megoldás lehet az idő megadására,hogy a dátumot adjuk meg:

**package** pack;

**import** java.util.Calendar;

**import** java.util.Date;

**import** java.util.GregorianCalendar;

**import** java.util.Timer;

**import** java.util.TimerTask;

**public** **class** Remind {

Timer timer;

**public** Remind(**int** seconds){

timer = **new** Timer();

Date ido;

Calendar cal = Calendar.*getInstance*();

cal.set(Calendar.***MINUTE***, 1);

ido = cal.getTime();

timer.schedule(**new** RemindTask(), ido);

}

**class** RemindTask **extends** TimerTask{

@Override

**public** **void** run() {

System.***out***.println("Time is out!!!!");

timer.cancel();

}

}

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

**new** Remind(15);

System.***out***.println("Munka ütemezve");

}

}

Ebben a példában a futtatáskor lekérd dátumhoz(Calendar.getInstance) hozzáadunk plusz 1 percet és akkor indítjuk a szálat.

**Daemon Thread:** Valójában ez olyan szálat jelöl, ami valami háttérszolgáltatást biztosít (pl. kapcsolatok fogadása amíg az alkalmazásszerver fut), nem pedig egy jól meghatározott, mindenképpen elvégzendő feladatot lát el. A Java program automatikusan kilép ha már csak daemon-szálak futnak az alkalmazásban, szemben a normál szálakkal, amelyek befejezéséig futva marad a program akkor is, ha a főszál már befejeződött.A Garbage Collector is egy daemon szál.Ezek akkor kapnak processzor időt,amikor más szálak nem tartanak rá igényt.

Az ilyen szál képes tovább élni azután is,hogy amelyik szál elindította őt befejeződik, a sima szál megáll.

**Végrehajtási szál**: utasítások sora + CPU regiszterek + megfelel ő verem. A Java program indításakor létrejön egy els ődleges szál, amely a kés őbbiekben további szálakat hozhat létre (és így tovább).

**Időzített szálak leállítása:**

-A cancel() metódus meghívásával.

-Az időzítő szálát daemon szállá kell alakítani.(new Timer(true)).Ha a java program csak daemon szálakat tartalmaz,akkor kilép.

-El kell távolítani a Timer objektumhoz tartozó mutatókat,és ekkor a szál is megszűnik.

-a System.exit metódust hívjuk meg,ami leállítja a szálat,és ezzel együtt a programot is.

**Szálak példányosítása:**

**package** pack;

**public** **class** SimpleThread **extends** Thread {

**public** SimpleThread(String str) {

**super**(str);

/\*A konstruktor paraméterként egy stringet vár,amivel hivatkozhatunk

majd a szálra.Az ősosztály konstruktora beállítja a szál nevét.

\*/

}

/\*A run() metódusban történik a tényleges munka.\*/

**public** **void** run() {

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println(i + " " + getName());

/\*getName() metódussal kérdezhető le a szál neve\*/

**try** {

/\*Itt egy másodpercen belüli véletlen szám generűlódik ,és addig altatás

következik be erre a szálra.\*/

*sleep*((**long**)(Math.*random*() \* 1000));

} **catch** (InterruptedException e) {}

/\*a sleep metódussal\*/

}

System.***out***.println("KÉSZ! " + getName());

}

}

A futtató osztály:

**package** pack;

**public** **class** TwoThreadsTest {

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

**new** SimpleThread("Jamaica").start();

**new** SimpleThread("Fiji").start();

}

}

A main() metódus azonnal két szálat hoz létre a start() metódus segítségévle.A start() metódus hívja meg run() metódusokat.Ha nem altatnánk(sleep) a szálakat menet közben,akkor nem adódna át a proszesszor a másik szálnak,csak mikor az első teljesen végzett.

**Mikor nem jó ha a Thread osztályból származtatunk:**Ez a megoldás akkor rossz,ha például egy Jframe-ből kell származtatnunk,és szeretnénk külön szálon futtatni,de a java csak egyszeres öröklődést engedélyez.Ezért oldották meg a java fejlesztői,hogy legyen egy úgynevezett Runnable interfész ,amit implementálhatunk és annak segítségével már lehet külön szálon kezelni.A Runnable interfész egyetlen metódust definiál,ami a run().

**package** pack;

**public** **class** SimpleThread **implements** Runnable {

**public** **void** run() {

System.***out***.println("alma");

}

}

**package** pack;

**public** **class** TwoThreadsTest {

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

Thread szal = **new** Thread(**new** SimpleThread());

szal.start();

}

}

Indítani úgy lehet, ha egy Thread objektumnak megadod paraméterként, és arra meghívjuk a start() eljárást.

**Thread osztály részletei:**

-konstruktorok:

public Thread()

Itt paraméter nélküli konstruktornál generálódik egy név a szálnak: Thread-0 a szám változik.

public Thread([Runnable](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) target)

Itt egy olyan osztály objektumát adjuk át,amelyik implementálja a runnable interfacet.

public Thread([String](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html) name)

Itt megadjuk a szál nevét.

public Thread([Runnable](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) target,

[String](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html) name)

stb.....

-Egyéb metódusok:

public static [Thread](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html) currentThread()

Visszaadja azt a Thread objektumot,ami jelenleg él.

public class MyThread extends Thread {

public void run( ) {

if (Thread.currentThread( ) != this)

throw new IllegalStateException(

"Run method called by incorrect thread");

... main logic ...

Ez az if elágazás akkor él,ha ez a szál életbe lép,pedig nem ez a jelenleg aktuálisan végrehajtandó szál,akkor ugyanis IllegalStateExceptiont dob.Tehát ez akkor dobódik,ha nem ez a szál az aktuális,mégis meghívódik a run metódusa.

public static void sleep(long millis)

throws [InterruptedException](http://docs.oracle.com/javase/7/docs/api/java/lang/InterruptedException.html)

Adott időnyi várakozás.Felfüggesztheti saját magát ezzel a szál egy időre.Más szálakra nem használhatja,csak saját magára.

public void start()

Szál indítása.Egy szál sosem indítható egynél többször.

public final boolean isAlive()

Még él-e a szál.Ez akkor ad vissza igazat,ha elindítottik start() metúdussal és meghívódott a run() metódusa.

public final void suspend()

Szál felfüggesztése.

public final void resume()

Szál újraindítása suspend után.

public final void setPriority(int newPriority)

prior lehet 1-10, fontosságot jelöl. OS függő, hogy pontosan milyen hatása van. Időosztásos (time slicing) rendszereken nincs gond vele, egyébként egy "önző" szál teljesen befoglalhatja a CPU-t.

public final void join(long millis)

throws [InterruptedException](http://docs.oracle.com/javase/7/docs/api/java/lang/InterruptedException.html)

Ez a metódus mindaddig vár, amíg az a szál, amelyikre meghívtuk, be nem fejeződik. Neve onnan származik, hogy a hívó szál addig várakozik, amíg a specifikált szál nem *csatlakozik (join)* hozzá. Olyan formája is van ennek a metódusnak, amelyben megadható az a maximális idő, hogy legfeljebb meddig akar várakozni a másik szálra.

**package** pack;

**public** **class** SimpleThread **extends** Thread {

**public** SimpleThread(String nev){

**super**(nev);

}

**public** **void** run() {

System.***out***.println(**this**.getName());

**try** {

**this**.*sleep*(5000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**package** pack;

**public** **class** SimpleThread1 **extends** Thread{

**public** SimpleThread szal;

**public** SimpleThread1(String nev,SimpleThread elso){

**super**(nev);

**this**.szal = elso;

}

**public** **void** run() {

**try** {

szal.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println(**this**.getName());

}

}

**package** pack;

**public** **class** TwoThreadsTest {

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

SimpleThread t1 = **new** SimpleThread("elso");

SimpleThread1 t2 = **new** SimpleThread1("masodik",t1);

t1.start();

t2.start();

}

}

A szal.join() –val megvárjuk míg az a szál mindent meg nem csinál ,ami a run() metódusában van.

a join() metódus segítségével, ami lehet ővé teszi, hogy egy szál, miel őtt elkezdene műveleteket végezni valamilyen adatokkal, ha szükséges, megvárja, hogy egy másik szál befejezze az illet ő adatokkal kapcsolatos, már elkezdett műveletek végrehajtását

public final void setDaemon(boolean on)

setDaemon(): daemon szál készítése (akkor terminál, ha minden más, nem daemon szál is már terminált).

**Szál terminálása egyszerű módszer:**

**private** **volatile** **boolean** done = **false**;

**private** **volatile** **boolean** done = **false**;

**public** **void** run( )

**while**(done!=**true**) {

nextCharacter( );

**try** {

Thread.*sleep*(getPauseTime( ));

} **catch** (InterruptedException ie) {

**return**;

}

}

}

**public** **void** setDone(){

**this**.done=**true**;

}

Ugyebár egy szál terminálódik,ha végrehajtódik a run() metódusa teljesen.Ezzel a módszerrelmanuálisan leállíthatjuk,hisz ha nem megy rá a while ciklusra,akkor nem altat,és nem indul újra a run() metódus,és végig ér a run() metódus,tehát terminálódik.Ekkor un. dead állapotba kerül.A Garbage Collector csak a dead állapotban lévő szál objektumokat szabadíthatja fel.

public void interrupt()

Ez a metódus ha meghívódik egy objektumra,akkor ez azt váltja ki,hogy a blokkoló metódusok(sleep,wait,join),melyek blokkolják a szálat,ezek megdobnak InterruptedExceptiont.

Ez a metódus megoldás lehet a setDone() fenti megoldás alternatívájára is.

run() metódus belseje:

**while**(!isInterrupted()){

**for**(**int** i=1; i<=5; i++){

**try**{

Thread.*sleep*(100);

}**catch**(InterruptedException e){

**return**;

}

**if**(*commonMemory*> 5 ){

**this**.interrupt();

}

}

}

Az isInterrupted() akkor ad igazat,ha meghívtuk az objektumra az interrupted metódust.Ha alvás közben mondjuk a mainból kiadunk erre a példányra egy interruptot,akkor a sleep rögtön kivételt dob,és úgy írtam meg a catch blokkot,hogy visszatérjen a run metódusból(return).

**Main mint fő szál:**

static Thread currentThread()

Bár a fő szálat automatikusan létrehozza a virtuális gép, amikor a program elindul, mégis vezérelhető ez a szál a Thread objektumon keresztül. Ennek érdekében egy referenciát kell létrehoznia erre a szálra a currentThread metódus hívásával, amely a Thread osztály public static metódusa.

Ez a metódus visszaad egy referenciát arra a szálra, amelyben hívták. Amint megvan a referenciája a fő szálra, vezérelheti azt. Kezdjünk egy példával:

package pack;

class TwoThreadsTest {

public static void main(String args[]) {

Thread t = Thread.currentThread();

System.out.println("Current thread: " + t);

// Change the name of the thread

t.setName("My Thread");

System.out.println("After name change: " + t);

try {

for(int n = 5; n > 0; n--) {

System.out.println(n);

Thread.sleep(1000);

}

} catch (InterruptedException e) {

System.out.println("Main thread interrupted");

}

}

}

A program által generált kimenet:

Current thread: Thread[main,5,main]

After name change: Thread[My Thread,5,main]

5

4

3

2

1

A kiírt információ rendre: a szál neve, prioritása és a szálcsoport neve. Defaultként a fő szál neve **main**. Prioritása 5, ami szintén a default prioritás, s annak a szálcsoportnak a neve, amelybe beletartozik, szintén **main**. Egy szálcsoport (thread group) olyan adatstruktúra, amely egy szálcsoport valamennyi száljának állapotát együttesen vezérli. Ezt a folyamatot egy speciális runtime környezet kezeli

**Runnable interfész:**

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.");

}

}

**Sysnchronized blokk alkalmazása:**

**public** **class** SimpleThread **extends** Thread {

**public** **static** **int** *commonMemory* = 0;

**public** SimpleThread(String name){

**super**(name);

}

**public** **void** run(){

**int** tempMemory = 0;

**synchronized** (**this**.getClass()) {

**for**(**int** i=1; i<=5; i++){

tempMemory = *commonMemory*;

System.***out***.println(Thread.*currentThread*().getName() + " - Read - Value= " + tempMemory);

**try**{

Thread.*sleep*(100);

}**catch**(InterruptedException e){

**return**;

}

*commonMemory* = ++tempMemory;

System.***out***.println(Thread.*currentThread*().getName() + " - Write - Value= " + tempMemory);

}

}

}

}

A fenti kód értelmezése:

A statikus commonMemory-t úgy képzeljük el mint a közös raktár.Ebből olvasnak értéket és ezt írják a szálak.Jelen esetben két szálat fogunk létrehozni.A main szálban két darab SimpleThread példányt fogunk létrehozni,ami azt jelenti,hogy még két szál fog futni mellette.Ez a SimpleThread szál a run() metódusában azt csinálja,hogy a saját belső memóriájába(tempMemory) beolvassa a közös memória értékét(tehát olvas), majd terminálja magát 0,1 másodpercre, majd a commonMemorynak ami a közös raktár,értékül adja a saját megnövelt memóriaértékét(tehát most ír).Közben minden művelet után ki írja a konzolra,hogy mit csinált,és melyik szál csinálta.

Mi lehet a probléma?

Könnyen belátható,hogy végeredményként 10-nek kell a közös memóriában lennie,hisz mind a két szál egy ötször körbefutó for ciklusban növeli az értéket.Viszont mi van akkor,ha a szál1 kiolvassa,hohy az érték a commonMemoryban például 1,mjad sleepel és megkapja a cpu-t a szál 2 és ő is kiolvassa ,hogy 1 az érték,majd szal1 megkapja a cpu-t és módosít 2 –re,majd szal2 is módosít 2-re.Tehát a szal1 módosítása hiába való,kiesik.Bár a fenti megoldás orvosolja a problémát azáltal,hogy sychronized blokkba tettük azon részt,ami konkurenciát okozna.

A synchronized paraméterként azt az osztályt kapja,amin a konkurens végrehajtás nem lehetrséges,ezért használjuk itt mivel ezen az osztályon van a static változó amit nem lehet oszottan kezelni a this.getClass()-t.

Mivel a synchronized blokk magába foglalja a for ciklust is,ezért a szal1 végrehajthatja a teljes for ciklust és csak utánna,adódik át a cpu a szál2-nek.

Az alábbi megoldás csak a ciklus szakaszára érvényes,és ez is tökéletes megoldás.Itt csak az a kikötés,hogy a ciklus magot tutira végrehajthassa a szál.De elvehetődik a vezérlés a szál1-től valamely ciklus végén.Tehát itt nem biztos,hogy a szál1 fogja 5-ig és a szál2 fogja 10-ig növelni a változó értékét.

**public** **void** run(){

**int** tempMemory = 0;

**for**(**int** i=1; i<=5; i++){

**synchronized** (**this**.getClass()) {

tempMemory = *commonMemory*;

System.***out***.println(Thread.*currentThread*().getName() + " - Read - Value= " + tempMemory);

**try**{

Thread.*sleep*(100);

}**catch**(InterruptedException e){

**return**;

}

*commonMemory* = ++tempMemory;

System.***out***.println(Thread.*currentThread*().getName() + " - Write - Value= " + tempMemory);

}

}

}

A synchronized blokk előnyei:

**public** **class** SimpleThread {

**public** **static** Integer *commonMemory* = 0;

**public** SimpleThread(){

}

**public** **void** doit(){

**int** tempMemory = 0;

**for**(**int** i=1; i<=5; i++){

tempMemory = *commonMemory*;

System.***out***.println(Thread.*currentThread*().getName() + " - Read - Value= " + tempMemory);

**try**{

Thread.*sleep*(100);

}**catch**(InterruptedException e){

**return**;

}

*commonMemory* = ++tempMemory;

System.***out***.println(Thread.*currentThread*().getName() + " - Write - Value= " + tempMemory);

}

}

}

**public** **class** VThread **extends** Thread{

**public** SimpleThread smp;

**public** VThread(String name,SimpleThread smp){

**super**(name);

**this**.smp = smp;

}

**public** **void** run() {

**synchronized**(smp){

**this**.smp.doit();

}

}

}

**class** TwoThreadsTest {

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

SimpleThread sm = **new** SimpleThread();

VThread t1 = **new** VThread("elso",sm);

VThread t2 = **new** VThread("masodik",sm);

t1.start();

t2.start();

**try** {

t1.join();

t2.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Végeredmény: " + SimpleThread.*commonMemory*);

System.***out***.println("end of program");

}

}

A fent látható 3 osztály alkot egy programot jelen esetben.synchronized blokkot ha this kulcsszóval használjuk egy metódusban,azzal azt érjük el,mintha a metódus synchronized metódus lenne.

Előnye a blokknak,hogy kisebb kódra bontható a szinkronizált rész.

Olyan objektumok is szinkronizálhatóak,melyek nem úgy lettek megírva,hogy tartalmaznak synchronized metódusokat.

A fenti példában látható,hogy nem az osztályba írtunk szonkronizált metódust,hanem az osztályból(SimpleThread) készített példányra lockoltunk.

Metódus szinkronizációjával egyenértekű a következő blokos megoldás:

public void ratesz(int osszeg){

synchronized(this){

...

}

}

**Szálak közötti kommunikáció:**Itt arról van szó,hogy egy szál a másikat értesíteni tudja arról,hogy folytassa munkálatatit vagy pihenjen keveset.

Hibátlan, de alacsony hatékonyságú megoldást biztosíthatunk, ha a várakozási sorban bevezetünk egy  szemafort, amit a termelő igaz, a fogyasztó pedig hamis állapotba állít adat beírásakor, illetve kiolvasásakor. A szemafor folytonos lekérdezese (polling) azonban tetemes CPU időt használ fel.

A tökéletes megoldás:

Amint már említettük, a többszálúság alkalmazása helyettesíti az eseménykezelő hurkokon alapuló programozást a feladatot diszkrét logikai egységekre bontva. A szál további előnye: kiküszöböli a pollingot, ami felesleges CPU időt igényelne. Tekintsük erre a klasszikus termelő-fogyasztó együttműködést. A termelőt és a fogyasztót is egy-egy szál valósítsa meg. S tegyük fel, hogy a termelőnek mindaddig várakoznia kell, amíg a fogyasztó fel nem dolgozta a termelt adatot. Egy pollingos rendszerben a termelő jelentős CPU időt fog elvesztegetni annak lekérdezésével, hogy a fogyasztó befejezte-e a feladatát. Ugyanez érvényes a fordított irányban is, azaz a fogyasztó is tetemes CPU időt fogyaszthat a pollinggal, arra várakozva, hogy a termelő előállítson egy adatot számára.

A polling elkerülésére Java bevezetett egy elegáns interprocess kommunikációs mechanizmust a **wait()**, **notify()** és **notifyAll()** metódusokkal. Ezek a metódusok mint **final** vannak implementálva az**Object**-ben, tehát minden osztály tartalmazza őket. Mindegyik fenti metódus csak **synchronized** metódusból hívható. Funkciójuk a következő:

* **wait()**azt üzeni annak a szálnak, amelyből hívták, hogy függessze fel működését ("aludjon"), amíg egy másik szál nem szóllítja meg a **notify()** metódussal. Csak szinkronizált metódusból vagy blokkból hívható meg.
* **notify()** véletlenszerűen kiválasztva felébreszt egy szálat azon szálak közül, amelyek a **wait()**-et hívták ezen az objektumon. Csak szinkronizált metódusból vagy blokkból hívható meg.
* **notifyAll()** az összes szálat felébreszti, amely a **wait()**-et hívta ugyanazon objektumon. A felébresztett szálak közül a legnagyobb prioritású fog futni. Csak szinkronizált metódusból vagy blokkból hívható meg.

Ezek a metódusok az **Object** osztályon belül az alábbiak szerint vannak deklarálva:

final void wait()

final void notify()

final void notifyAll()

A következő példában megvalósítom a a termelő,fogyasztó és a raktár szerkezetet:

Alapgondolat:A Raktar osztály tartalmaz egy Listát,amibe jelen esetben integer számokat pakolhatunk.Illetve tartalmaz egy logikai változót,ami azt jelzi ,hogy rakva van-e a raktárba vagy nincs.Itt azt feltételezzük,hogy a termelő egy integert termelhet,és azt ki is kell vennie a feldolgozónak.Tehát egyszerre egy betevés és egy kivevés lehet.Ne legyen olyan,hogy berakok 2-őt és kiveszek egyetm majd berakok 3-at és kiveszek egyet stb....

A Raktar osztály két szinkronizált metódust tartalmaz,az egyik amivel hozzáadhatunk a raktárhoz elemet, a másik amivel kiolvashatunk a raktárból elemet(ugyebár ki nem vesszük belőle,marad a listában).

Csak szinkronizált metódusból adható ki a wait() notify() notifyAll() metódusok.

A Termelo szal az add() metódust hívogatja,hisz ő hozzáad, a Fogyaszto pedig a kivesz() metódust hivogatja.

Mivel mind a termelő és mind a fogyasztó,ugyanazon objektumon végezne módosításaokat,hisz mindkettőnek ugyan azt a raktár referenciát adtuk értékül,ezért kellenek a további intézkedések:

Ha épp a termelő még hozzáadna a raktárhoz,de már van ott valami,akkor kiad egy wait() utasítást erre az objektumra(ezt az objektumon belül definiáltuk),majd a másik szál jut szóhoz és majd az fogja éleszteni erre az objektumra a termelő szálat.

**class** Raktar {

**public** List<Integer> raktar = **new** ArrayList<Integer>();

**public** **boolean** vanBenne = **false**;

/\*Ez a boolean változó hamis,ha nincs a raktárban semmi.\*/

**public** **synchronized** **void** add(**int** szam){

**if**(vanBenne == **true**){

**try** {

wait();

/\*Ha a raktárban van valami,akkor a hozzáadó metódus nem tud még

valamit hozzáadni,ezért ki kell adni a wait() utasítást,

és lemondani a cpu-ról.\*/

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**if**(raktar.size() < 100){

raktar.add(szam);

System.***out***.println("Berakva: " + szam);

vanBenne = **true**;

notify();

/\*Itt amikor hozzáadtunk a raktárhoz, a vanBenne logikai változót true-ra

állítjuk,hogy ne tegyünk bele mégegyszer,ha esetleg az ütemező ennek a szálnak

adná a kövi cpu időt.És felébresztjük a másik szálat a várakozásból.

Igen itt a nitify elég,hisz csak egy másik szál várakozik erre az objektumra

ami a Raktar.\*/

}**else**{

notify();

}

}

**public** **synchronized** **void** kivesz(){

**if**(vanBenne == **false**){

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**if**(raktar.size() < 100){

System.***out***.println("Kiveve: " + raktar.get(raktar.size()-1));

vanBenne = **false**;

notify();

}**else**{

notify();

}

}

}

**class** Termelo **implements** Runnable {

**private** Raktar raki;

**public** Thread szal;

**private** **int** szam = 0;

**public** Termelo(Raktar raki){

szal = **new** Thread(**this**,"Termelo");

szal.start();

**this**.raki = raki;

}

@Override

**public** **void** run() {

**while**(**true**){

raki.add(++szam);

**if**(szam == 100){

**return**;

}

}

}

}

**class** Fogyaszto **implements** Runnable {

**private** Raktar raki;

**public** Thread szal;

**public** Fogyaszto(Raktar raki){

szal = **new** Thread(**this**,"Fogyaszto");

szal.start();

**this**.raki = raki;

}

@Override

**public** **void** run() {

**while**(**true**){

raki.kivesz();

**if**(raki.raktar.size()==99){

/\*Ezzel a megoldással kiugrok a while ciklusból,ha már megvolt a 100 elem,

és mivel nincs más a run()-ban ,ezért ez a szál dead szál lesz.\*/

**return**;

}

}

}

}

**class** SimpleThread1 {

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

Thread ez = Thread.*currentThread*();

Raktar r = **new** Raktar();

Fogyaszto masodik = **new** Fogyaszto(r);

Termelo elso = **new** Termelo(r);

**try** {

masodik.szal.join();

elso.szal.join();

/\*Mivel ezek az osztályok nem a Thread osztályból származnak,ezért nem hívható

meg a referenciájukra a join(),viszont az osztályon belüli Thread objektumra már

igen,tehát így kell joinolni interfész esetén.\*/

} **catch** (InterruptedException e) {

e.printStackTrace();

}

/\*Fontos,mindíg a main szálnak kell legutoljára terminálódnia,ezért bevárja az összes

többi szálat,majd úgy ér véget.\*/

System.***out***.println("vege a proginak");

}

}

**Deadlock problematika:**Itt az a helyzet,hogy két szál egymásra vár.Az egyik lefoglal egy erőforrást t1-et majd a másik lefoglalja t2-őt majd az első szeretné lefoglalni t2-t is de nem tudja mert már foglalt,a második pedig t1-et szeretné lefoglalni de már azis foglalt ezért a végtelenségig várnak egymásra.

public class TestThread {

public static Object Lock1 = new Object();

public static Object Lock2 = new Object();

public static void main(String args[]) {

ThreadDemo1 T1 = new ThreadDemo1();

ThreadDemo2 T2 = new ThreadDemo2();

T1.start();

T2.start();

}

private static class ThreadDemo1 extends Thread {

public void run() {

synchronized (Lock1) {

System.out.println("Thread 1: Holding lock 1...");

try { Thread.sleep(10); }

catch (InterruptedException e) {}

System.out.println("Thread 1: Waiting for lock 2...");

synchronized (Lock2) {

System.out.println("Thread 1: Holding lock 1 & 2...");

}

}

}

}

private static class ThreadDemo2 extends Thread {

public void run() {

synchronized (Lock1) {

System.out.println("Thread 2: Holding lock 1...");

try { Thread.sleep(10); }

catch (InterruptedException e) {}

System.out.println("Thread 2: Waiting for lock 2...");

synchronized (Lock2) {

System.out.println("Thread 2: Holding lock 1 & 2...");

}

}

}

}

}

A kimenet:

Thread 1: Holding lock 1...

Thread 2: Holding lock 2...

Thread 1: Waiting for lock 2...

Thread 2: Waiting for lock 1...

A fenti kódban beütött a deadlock:

A Demo1 synchronized blokkal lockolja a Lock1 objektumot, majd kap egy sleepet, amikor ugyanis a Demo2 szál lockolja a Lock2 objektumot majd ő is sleeppel alszik.Ezután a Demo1 kapja újra a cpu-t ,de ekkor a következő dolga az lenne,hogy lefoglalja a Lock2 objektumot,viszont nem tudja mert az már lockolva van,majd elveszti a cpu-t és megkapja a Demo2, aki viszont a Lock1 objektumot szeretné zárolni,viszont azt meg a Demo1 zárolja és így a végtelenségig várnake egymásra.

Megoldás:

public class TestThread {

public static Object Lock1 = new Object();

public static Object Lock2 = new Object();

public static void main(String args[]) {

ThreadDemo1 T1 = new ThreadDemo1();

ThreadDemo2 T2 = new ThreadDemo2();

T1.start();

T2.start();

}

private static class ThreadDemo1 extends Thread {

public void run() {

synchronized (Lock1) {

System.out.println("Thread 1: Holding lock 1...");

try { Thread.sleep(10); }

catch (InterruptedException e) {}

System.out.println("Thread 1: Waiting for lock 2...");

synchronized (Lock2) {

System.out.println("Thread 1: Holding lock 1 & 2...");

}

}

}

}

private static class ThreadDemo2 extends Thread {

public void run() {

synchronized (Lock1) {

System.out.println("Thread 2: Holding lock 1...");

try { Thread.sleep(10); }

catch (InterruptedException e) {}

System.out.println("Thread 2: Waiting for lock 2...");

synchronized (Lock2) {

System.out.println("Thread 2: Holding lock 1 & 2...");

}

}

}

}

}

Kimenet:

Thread 1: Holding lock 1...

Thread 1: Waiting for lock 2...

Thread 1: Holding lock 1 & 2...

Thread 2: Holding lock 1...

Thread 2: Waiting for lock 2...

Thread 2: Holding lock 1 & 2...

Értelmezés:A Demo1 zárolja a Lock1 objektumot,majd amikor sleep() miatt elveszti a cpu-t ,a Demo2 is a Lock1 objektumot zárolná,viszont nem tudja ezért elveszti a cpu-t ,majd a Demo1 zárolja a Lock2-t is és végrehajt mindent,és a blokk végén felszabadítja az erőforrásokat,és ezért a Demo2 is zökkenőmentesen végig tudja csinálni a procedúrát.

**Deadlock szinkronizált metódusok miatt:**

Az alábbi példában szemléltetésre kerül az,hogy miként alakulhat ki deadlock szinkronizált metódusok használatba vétele miatt.

C és D osztályok ugyan azt a viselkedést produkálják.Készítsünk mindből egy példányt.A C osztályból készítünk egy példányt illetve a D osztályból is.Mindkét objektumnak vagy egy kiir() szinkronizált metódusa, amely kezdetben kiír egy szöveget majd sleeppel átadja a cpu-t egy másik szálnak,majd a másik osztály példányának meghívná a szintén szinkronizált bejut metódusát.

Program menete:Készítünk egy példányt a C osztályból és a D osztályból is.

Majd elindulnak a szálak.Először az első szál, ami meghívja a c objektum kiir() metódusát amiben kiír egy szöveget majd sleeppel terminálja magát,majd a második szál is meghívja a D objektum kiir() metódusát,majd az is kiír valamit majd sleepet tol.Ez után az első szálon lesz megint a sor,ami viszont a megkapott D objektumnak használná a bejut() metódusát,viszont azt nem tudja,mert még a második szál mindíg blokkolja azt az objektumot szinkronizált metódusok végrehajtására,mert még a kiir() metódust nem hajtotta végre,hisz a felénél sleepet tolt.Az első szál elveszti a cpu-t majd a második szál folytatódik a kiirí() metódusnál,viszont az meg a C objektum bejut() metódusát használná,amit szintén nem lehet.Eredmény deadlock.Megoldás lehet az ha a bejut() metódusokat nem szinkronizálta csináljuk.Ekkor ugyanis a nem szinkronizált metódusokhoz bármennyi szál konkurensen hozzáférhet.

A main szálban látható,hogy maszek megoldást használok.Az a lényeg,hogy C objektumnak meg kell kapnia egy D objektumot, D obketumnak pedig meg kell kapnia egy C objektumot,és így tudják hívni annak bejut() metódusát.Mivel először valamelyiket létre kell hozni és csak úgy adható át a másiknak,ezért a C objektumnak van setD metódusa amivel majd a később létrehozott D objektumot megkapja.

Ha a kikommentezett részt kivennénk a programból,akkor nem lenne deadlock,mert nem ugyan annak az objektumnak a szinkronizált metódusát hívogatnánk.

Érthetően:Van egy C osztály ,aminek vannak szinkronizált metóduasi.Ha két különbözö objektumot is létrehozok belőle,akkor két külön szálon,annak a két metódusnak nyugodtan hívhatok szinkronizált metódust konkurensen,hisz nem egy memóriaterületek.

**class** C **extends** Thread{

D d;

**public** C(String name){

**super**(name);

}

**public** **void** run() {

kiir();

}

**public** **void** setD(D d){

**this**.d = d;

}

**public** **synchronized** **void** kiir(){

System.***out***.println("Thread name: " + Thread.*currentThread*().getName() +

"Start the execution");

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("" + Thread.*currentThread*().getName() + "Várakozik.....");

d.bejut();

}

**public** **synchronized** **void** bejut(){

System.***out***.println("Bejutottam a C beldelyebe!!!!!!");

}

}

**class** D **extends** Thread{

C c ;

**public** D(String name,C c){

**super**(name);

**this**.c = c;

}

**public** **void** run() {

kiir();

}

**public** **synchronized** **void** kiir(){

System.***out***.println("Thread name: " + Thread.*currentThread*().getName() +

"Start the execution");

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println("" + Thread.*currentThread*().getName() + "Várakozik.....");

c.bejut();

}

**public** **synchronized** **void** bejut(){

System.***out***.println("Bejutottam a D beldelyebe!!!!!!");

}

}

**class** TestThread {

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

C c = **new** C("C");

D d = **new** D("D",c);

/\*c.setD(new D("F",c));\*/

c.setD(d);

c.start();

d.start();

**try** {

c.join();

d.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("vege");

}

}

**Szálak közötti kommunikáció PipedInputStream és PipedOuputStream segítségével:**

A csövek kiválóan alkalmasak végrehajtási szálak közötti kommunikáció megszervezésére.Az egyik szál adatokat írhat a kimeneti adatfolyamra, és ezeket a másik szál elolvashatja a megfelelő bemeneti adatfolyamon.

A következő program azt csinálja,hogy van két szál, az egyik egy szöveges file olvasását végzi majd out csatornára írja amit kiolvasott, a másik szál pedig az input csőből kilvassa az adatokat és beírja a másolat.txt szöveges állományba.

Mivel ebben az esetben az olvasott adatot kell eljuttatni valahova,ezért az out cső csatlakozik az input csőhöz,hisz ki szeretnénk belőle olvasni majd.

**public** **class** Writes **implements** Runnable{

**public** Thread szal;

**private** FileOutputStream out;

**private** PipedInputStream incso;

**public** Writes(String utvonal,PipedInputStream incso) **throws** FileNotFoundException {

**this**.szal = **new** Thread(**this**,"Író");

out = **new** FileOutputStream(utvonal);

**this**.incso = incso;

**this**.szal.start();

}

@Override

**public** **void** run() {

**int** c;

**try** {

/\*if(incso.available() == 0){

try {

this.szal.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}\*/

**while**((c = incso.read()) != -1){

**try** {

out.write(c);

out.flush();

} **catch** (IOException e) {

e.printStackTrace();

}

}

} **catch** (IOException e) {

e.printStackTrace();

}**finally**{

**if**(out != **null**){

**try** {

out.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

**if**(incso != **null**){

**try** {

incso.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

}

**public** **class** Reades **implements** Runnable{

**public** Thread szal;

**private** FileInputStream in;

**private** PipedOutputStream out;

**public** Reades(String utvonal,PipedOutputStream out) **throws** FileNotFoundException {

**this**.szal = **new** Thread(**this**,"Olvaso");

in = **new** FileInputStream(utvonal);

**this**.out = out;

**this**.szal.start();

}

@Override

**public** **void** run() {

**int** c;

**try** {

**while**((c = in.read()) != -1){

out.write(c);

}

} **catch** (IOException e) {

e.printStackTrace();

}**finally**{

**if**(in != **null**){

**try** {

in.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

**if**(out != **null**){

**try** {

out.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

}

**class** Fut{

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

PipedInputStream in = **new** PipedInputStream();

PipedOutputStream out = **new** PipedOutputStream(in);

Writes w = **new** Writes("./masolat.txt",in);

Reades r = **new** Reades("./alma.txt",out);

**try** {

r.szal.join();

w.szal.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("A másolás megtörtént!!!!!");

}

}

**Többszálú programozás a Jávában   
/ Java lépésről-lépésre © Gál Tibor /**

**Multitasking és multithreading**

**A Java thread modellje**

**Szál prioritások**

**A Thread osztály és a Runnable iterfész**

A Java többszálú programozási (multithreading) rendszere a **Thread** osztályra, ennek metódusaira, illetve a **Runnable** interfészre épül. Egy új szál létrehozására a programnak vagy ki kell terjeszteni a **Thread**osztályt, vagy implementálnia kell a **Runnable** interfészt.

A **Thread** osztály több metódust definiál a szálak menedzselésére. A fontosabbak, melyeket használni fogunk (vagy melyeket a forgalomból való kivonás előtt gyakran használtak, de használatuk hibás működéshez vezethetett) a következők:

|  |  |
| --- | --- |
| **Metódus** | **Jelentés** |
| getName | A szál nevének lekérdezése |
| getPriority | A szál prioritásának lekérdezése |
| isAlive | Annak meghatározása, hogy a szál még fut-e |
| join | Várakozás, amig egy szál befejeződik |
| interrupt | A szál futásának megszakítása |
| suspend | Egy szál futásának felfüggesztése. Deprikált! |
| resume | Egy felfüggesztett szál végrehajtásának felszabadítása. Deprikált! |
| run | Szál végrehajtásának belépési pontja |
| sleep | Egy szál végrehajtásának felfüggesztése bizonyos időre |
| start | Egy szál elindítása a run metódusának hívásával |
| stop | Egy szál futásának megállítása. Deprikált! |

**A main thread**

Amikor egy Java program elindul, akkor már egy szál fut. Ez rendszerint programjának *main thread*-je, mivel ez az egyetlen metódus, amit közvetlenül hív a virtuális gép, azaz amelyet közvetlenül végrehajt. A fő szál két szempontból is fontos:

* Ez az a szál, amelyből a gyerekszálak létrehozhatók.
* Ennek a szálnak kell legutoljára befejeznie a futását. Amikor a fő szál megáll, akkor programja befejeződött.

Bár a fő szálat automatikusan létrehozza a virtuális gép, amikor a program elindul, mégis vezérelhető ez a szál a **Thread** objektumon keresztül. Ennek érdekében egy referenciát kell létrehoznia erre a szálra a**currentThread** metódus hívásával, amely a **Thread** osztály **public static** metódusa. Ennek általános alakja:

static Thread currentThread()

Ez a metódus visszaad egy referenciát arra a szálra, amelyben hívták. Amint megvan a referenciája a fő szálra, vezérelheti azt. Kezdjünk egy példával:

// Controlling the main thread

class CurrentThreadDemo {

public static void main(String args[]) {

Thread t = Thread.currentThread();

System.out.println("Current thread: " + t);

// Change the name of the thread

t.setName("My Thread");

System.out.println("After name change: " + t);

try {

for(int n = 5; n > 0; n--) {

System.out.println(n);

Thread.sleep(1000);

}

} catch (InterruptedException e) {

System.out.println("Main thread interrupted");

}

}

}

A program által generált kimenet:

Current thread: Thread[main,5,main]

After name change: Thread[My Thread,5,main]

5

4

3

2

1

A kiírt információ rendre: a szál neve, prioritása és a szálcsoport neve. Defaultként a fő szál neve **main**. Prioritása 5, ami szintén a default prioritás, s annak a szálcsoportnak a neve, amelybe beletartozik, szintén **main**. Egy szálcsoport (thread group) olyan adatstruktúra, amely egy szálcsoport valamennyi száljának állapotát együttesen vezérli. Ezt a folyamatot egy speciális runtime környezet kezeli, ezzel azonban a továbbiakban nem foglalkozunk.

Ezután nézzük meg egy kicsit közelebbről a programban használt **Thread** metódusait. A **sleep()** metódus hatására annak a szálnak a végrehajtása, amelyben hívtuk, fel lesz függesztve az argumentumként megadott msec időtartamra. A metódus általános formája:

static void sleep(long millisec) throws InterruptedException

Amint a korábbi program mutatta, a szál neve a **setName()** metódussal változtatható. A szál nevét pedig a **getName()** hívásával kaphatjuk meg. Ezek a metódusok a **Thread** osztály metódusai és deklarálásuk a következő:

final void setName(String threadName)

final String getName()

Itt a threadName a szál nevét specifikálja.

**Egy thread létrehozása**

Általánosságban azt mondhatjuk, hogy egy szálat a **Thread** objektum példányosításával hozhatunk létre. A Java ennek végrehajtására két lehetőséget biztosít:

* Egy **Runnable** interfész implementálása
* A **Thread** osztály közvetlen kiterjesztése

Vizsgáljuk meg a fenti lehetőségeket.

**A Runnable implementálása**

Egy szál létrehozásának legegyszerűbb módja az, hogy létrehozunk egy olyan osztályt, amely a **Runnable** interfészt implementálja. Minden objektumon létre lehet hozni egy szálat, ha az implementálja a**Runnable**-t. A **Runnable** implementálásához az osztálynak csak egyetlen metódust, a **run()** metódust kell implementálnia, amely a következőképpen van deklarálva:

public abstract void run()

A **run()** metóduson belül kell megadni azt a kódot, amely az új szálat képezi. Fontos megérteni, hogy ez a **run()** metódus hívhat más metódusokat, használhat más osztályokat és deklarálhat változókat mint a fő szál. Az egyetlen eltérés az, hogy a **run()** meghatároz egy belépési pontot, egy másik konkurensen végrehajtandó szálat a programján belül. Ez a szál akkor fejeződik be, amikor a **run()** visszatér.

Miután létrehozott egy **Runnable**-t implementáló osztályt, létre kell hoznia egy **Thread** típusú objektumot az osztályon belül. A **Thread** több konstruktort definiál. Amelyiket mi most itt használunk, az a következő:

Thread(Runnable threadOb, String threadName)

Ebben a konstruktorban a threadOb a **Runnable** interfészt implementáló osztálynak egy példánya. Ez azt definiálja, hogy hol fog a szál kezdődni. Az új szál nevét pedig a threadName specifikálja.

A létrehozott új szál végrehajtása azonban mindaddig nem fog elkezdődni, amíg a **Thread** osztályban deklarált **start()** metódust meg nem hívjuk. Lényegében a **start()** hajt végre egy call-t a **run()**-ra. A**start** metódus deklarálása:

sunchronized void start()

Tekintsük az alábbi példát, amely létrehoz egy új szálat és elindítja annak végrehajtását:

// Create a second thread.

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.");

}

}

class ThreadDemo {

public static void main(String args[]) {

new NewThread(); // create a new thread

try {

for(int i = 5; i > 0; i--) {

System.out.println("Main Thread: " + i);

Thread.sleep(1000);

}

} catch (InterruptedException e) {

System.out.println("Main thread interrupted.");

}

System.out.println("Main thread exiting.");

}

}

A **NewThread** osztály konstruktorán belül az új **Thread** objektumot az alábbi utasítással hozzuk létre:

t = new Thread(this, "Demo Thread");

Amikor első argumentumként a **this**-t adjuk át, akkor ezzel azt specifikáljuk, hogy az új szálnak a **run()** metódust **ezen** az objektumon kell meghívnia. A **start()** meghívásával elindítjuk a szál végrehajtását a**run()** metóduson. Ennek következtében a gyerekszál **loop** hurokja elindul. A **start()** hívása után a **NewThread** konstruktora visszatér a **main()**-hez. Amikor a fő szál ismét folytatódik, akkor az is belép egy hurokba. Mindkét szál fut, osztottan használva a CPU-t, ameddig a hurkok be nem fejeződnek. A kimenet:

Child thread: Thread[Demo Thread,5,main]

Main Thread: 5

Child Thread: 5

Child Thread: 4

Main Thread: 4

Child Thread: 3

Child Thread: 2

Main Thread: 3

Child Thread: 1

Exiting child thread.

Main Thread: 2

Main Thread: 1

Main thread exiting.

Amint korábban már említettük, egy többszálú programban a fő szálnak kell utoljára befejeződni. Ha a fő szál hamarabb befejeződik mint a gyerekszál, akkor a Java runtime rendszere "fennakadhat". Az előző programban biztosított, hogy a fő szál fejeződik be utoljára, mivel az 1000, a gyerekszál pedig 500 msec időtartamokra van elaltatva. Nemsokára azonban ennél egy biztonságosabb megoldást látunk erre a problémára.

**A Thread osztály kiterjesztése**

Egy szál létrehozásának másik módja az, hogy a **Thread** osztályból származtatunk egy új osztályt, s majd ennek létrehozzuk egy példányát. Az új osztálynak felül kell definiálnia a **run()** metódust, amely a szál belépési pontja lesz. Továbbá hívnia kell a **start()** metódust, hogy a szál végrehajtása elinduljon. Az előző programnak a **Thread** kiterjesztésével megvalósított változatát az alábbiak mutatják:

// Create a second thread by extending Thread

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.");

}

}

class ExtendThread {

public static void main(String args[]) {

new NewThread(); // create a new thread

try {

for(int i = 5; i > 0; i--) {

System.out.println("Main Thread: " + i);

Thread.sleep(1000);

}

} catch (InterruptedException e) {

System.out.println("Main thread interrupted.");

}

System.out.println("Main thread exiting.");

}

}

Ez a program azonos kimenetet produkál mint a megelőző változat. Amint látható, a gyerekszál egy **NewThread** objektum példányosításával lett létrehozva, ami viszont a **Thread**-ből lett származtatva. Figyelje meg a **super()** metódust a **NewThread**-en belül. Ez a konstruktor következő formáját aktivizálja:

public Thread(String threadName)

Itt a threadName specifikálja a szál nevét.

**Több szál létrehozása**

Eddig csak két szálat használtunk, a fő szálat és egy gyerekszálat. Programunk azonban tetszőleges számú szálat létrehozhat, illetve elindíthat. Például a következő program három gyerekszálat hoz létre:

// Create multiple threads.

class NewThread implements Runnable {

String name; // name of thread

Thread t;

NewThread(String threadname) {

name = threadname;

t = new Thread(this, name);

System.out.println("New thread: " + t);

t.start(); // Start the thread

}

// This is the entry point for thread.

public void run() {

try {

for(int i = 5; i > 0; i--) {

System.out.println(name + ": " + i);

Thread.sleep(1000);

}

} catch (InterruptedException e) {

System.out.println(name + "Interrupted");

}

System.out.println(name + " exiting.");

}

}

class MultiThreadDemo {

public static void main(String args[]) {

new NewThread("One"); // start threads

new NewThread("Two");

new NewThread("Three");

try {

// wait for other threads to end

Thread.sleep(10000);

} catch (InterruptedException e) {

System.out.println("Main thread Interrupted");

}

System.out.println("Main thread exiting.");

}

}

Ennek a programnak a kimenete a következő:

New thread: Thread[One,5,main]

New thread: Thread[Two,5,main]

New thread: Thread[Three,5,main]

One: 5

Two: 5

Three: 5

One: 4

Two: 4

Three: 4

One: 3

Two: 3

Three: 3

One: 2

Two: 2

Three: 2

One: 1

Two: 1

Three: 1

One exiting.

Two exiting.

Three exiting.

Main thread exiting.

Amint látható, miután el lettek indítva, mindhárom gyerekszál osztozik a CPU-n. Figyelje meg a **main()**-ben a **sleep(10000)** hívását. Ez a fő szál 10 másodperces elaltatását jelenti, ez biztosítja, hogy utoljára ez a szál fog befejeződni.

**Az isAlive() és a join() használata**

Ahogy korábban már említettük, a főszálnak kell utoljára befejeződnie. Az előző példákban ezt a megfelelő idejű elaltatásokkal biztosítottuk. Ez azonban nem kielégítő megoldás. Jobb lenne, ha meg tudnánk vizsgálni, hogy egy szál befejeződött-e.

A Java két lehetőséget biztosít annak eldöntésére, hogy egy szál befejeződött-e. Az egyik az **isAlive()** metódus hívása. Ez **true** értéket ad vissza, ha az a szál, amelyiken hívtuk, még aktív. Egyébként a visszadott érték **false**. Magának a metódusnak az alakja:

final boolean isAlive() throws InterruptedException

Bár az **isAlive()** néha hasznos lehet, sokkal inkább használjuk a **join()** metódust, melynek alakja:

final void join() throws InterruptedException

Ez a metódus mindaddig vár, amíg az a szál, amelyikre meghívtuk, be nem fejeződik. Neve onnan származik, hogy a hívó szál addig várakozik, amíg a specifikált szál nem *csatlakozik (join)* hozzá. Olyan formája is van ennek a metódusnak, amelyben megadható az a maximális idő, hogy legfeljebb meddig akar várakozni a másik szálra. Az alábbiak a megelőző példa javított változatát szemléltetik a **join()**használatával. Egyúttal a **isAlive()** használatát is szemlélteti a program.

// Using join() to wait for threads to finish.

class NewThread implements Runnable {

String name; // name of thread

Thread t;

NewThread(String threadname) {

name = threadname;

t = new Thread(this, name);

System.out.println("New thread: " + t);

t.start(); // Start the thread

}

// This is the entry point for thread.

public void run() {

try {

for(int i = 5; i > 0; i--) {

System.out.println(name + ": " + i);

Thread.sleep(1000);

}

} catch (InterruptedException e) {

System.out.println(name + " interrupted.");

}

System.out.println(name + " exiting.");

}

}

class DemoJoin {

public static void main(String args[]) {

NewThread ob1 = new NewThread("One");

NewThread ob2 = new NewThread("Two");

NewThread ob3 = new NewThread("Three");

System.out.println("Thread One is alive: "

+ ob1.t.isAlive());

System.out.println("Thread Two is alive: "

+ ob2.t.isAlive());

System.out.println("Thread Three is alive: "

+ ob3.t.isAlive());

// wait for threads to finish

try {

System.out.println("Waiting for threads to finish.");

ob1.t.join();

ob2.t.join();

ob3.t.join();

} catch (InterruptedException e) {

System.out.println("Main thread Interrupted");

}

System.out.println("Thread One is alive: "

+ ob1.t.isAlive());

System.out.println("Thread Two is alive: "

+ ob2.t.isAlive());

System.out.println("Thread Three is alive: "

+ ob3.t.isAlive());

System.out.println("Main thread exiting.");

}

}

A program kimenete most a következő:

New thread: Thread[One,5,main]

New thread: Thread[Two,5,main]

New thread: Thread[Three,5,main]

Thread One is alive: true

Thread Two is alive: true

Thread Three is alive: true

Waiting for threads to finish.

One: 5

Two: 5

Three: 5

One: 4

Two: 4

Three: 4

One: 3

Two: 3

Three: 3

One: 2

Two: 2

Three: 2

One: 1

Two: 1

Three: 1

One exiting.

Two exiting.

Three exiting.

Thread One is alive: false

Thread Two is alive: false

Thread Three is alive: false

Main thread exiting.

Amint látható, a **join()**-ból való visszatérés után a szálak befejeződtek.

**Szálak felfüggesztése és továbbindítása**

Néha hasznos lehet egy szál futásának felfüggesztése. Például egy szeparált szál felhasználható az idő kijelzésére. Ha viszont a felhasználó ezt nem akarja, akkor fel lehet függeszteni ezt a szálat. Bármiről is van szó, egyszerű a szál felfüggesztése. Ugyanakkor, ha felfüggesztettünk egy szálat, akkor a továbbindítása is egyszerű.

A szálak felfüggesztésére a Java a **suspend()** és a **resume()** metódusokat vezette be. Alakjuk a következő:

 final void suspend()  
 final void resume()

A fenti metódusok használata során azonban néha hibás működés lépett fel, ezért ezeket a metódusokat kivonták a forgalomból (deprikáltak). Az átaluk megvalósított funkció más úton való megvalósításához azonban mégsem felesleges használatuk bemutatása egy rövid példán:

// Using suspend() and resume().   
class NewThread implements Runnable {   
  String name; // name of thread   
  Thread t;

  NewThread(String threadname) {   
    name = threadname;   
    t = new Thread(this, name);   
    System.out.println("New thread: " + t);   
    t.start(); // Start the thread   
  }

  // This is the entry point for thread.   
  public void run() {   
    try {   
      for(int i = 15; i > 0; i--) {   
        System.out.println(name + ": " + i);   
        Thread.sleep(200);   
      }   
    } catch (InterruptedException e) {   
      System.out.println(name + " interrupted.");   
    }   
    System.out.println(name + " exiting.");   
  }   
}

class SuspendResume {   
  public static void main(String args[]) {   
    NewThread ob1 = new NewThread("One");   
    NewThread ob2 = new NewThread("Two");

    try {   
      Thread.sleep(1000);   
      ob1.t.suspend();   
      System.out.println("Suspending thread One");   
      Thread.sleep(1000);   
      ob1.t.resume();   
      System.out.println("Resuming thread One");   
      ob2.t.suspend();   
      System.out.println("Suspending thread Two");   
      Thread.sleep(1000);   
      ob2.t.resume();   
      System.out.println("Resuming thread Two");   
    } catch (InterruptedException e) {   
      System.out.println("Main thread Interrupted");   
    }

    // wait for threads to finish   
    try {   
      System.out.println("Waiting for threads to finish.");   
      ob1.t.join();   
      ob2.t.join();   
    } catch (InterruptedException e) {   
      System.out.println("Main thread Interrupted");   
    }

    System.out.println("Main thread exiting.");   
  }   
}

Bár a fent két metódus használata igen kényelmesnek látszik, de problémát jelenthet, hogy a szál felfüggesztése vagy megállítása aszinkron módon, egy külső szálból hajtódik végre. Ennek következtében hibás működés vagy deadlock léphet fel, ezért a suspend(), resume(), továbbá a stop() metódusok használata a Java 1.1-es változatától kezdve már nem javasolt (deprecated metódusok). A helyes megoldás úgy valósítható meg, hogy a külső szálból csak jelezzük a felfüggesztési vagy megállítási igényt, de annak pontos végrehajtási idejét magára a szálra bizzuk.

A stop() használata azért problémás, mert azonnal megszakítja a szál minden tevékenységét, így az objektum inkonzisztens állapotban maradhat. Gondoljunk egy közepén megszakított banki tranzakcióra. A pénz leemelése mondjuk már megtörtént, de az új számlára rátétele nem, amikor a stop fellépett! Kívülről nem lehet eldönteni, mikor biztonságos egy szál megállítása, helyesen az csak a szálban magában dönthető el. Ezért, ha kívülről meg akarunk állítani egy szálat, akkor ezt csak "megüzenjük neki", s majd a szál maga végzi el az alkalmas állapotban a megállítást. Másként megfogalmazva, egy flag-et állítunk be, amelyet a szál periódikusan vizsgál. Például:

public class MyThread extends Thread {

private boolean stopRequested;

public void run(){

while(!stopRequested && me van elvegzendo munka) {

a meg elvegzendo munka vegrehajtasa

}

public void requestStop() {

stopRequested = true;

}

A fenti program biztosítja a run() metódus számára, hogy maga döntse el, mikor lehet a szálat befejezni. A fenti megoldás mindaddig jól működik, amíg a szál nincs blokkolt állapotban. Ez utóbbi esetben csak a blokkolás megszünte után fog befejeződni. De megszakítással kikényszeríthetjük a szálat a blokkolt állapotból, ezt használja ki a módosított stopRequest metódus:

public void requestStop() {

stopRequested = true;

interrupt();

}

A stopRequested megvizsgálható a szál catch részében az InterruptedException esetén:

try {

wait();

} catch (InterruptedException e) {

if(stopRequested)

return; // elhagyja a run metodust

}

A szálak futásának biztonságos felfüggesztéséhez mellőzzük a suspend() használatát, helyette bevezetünk egy suspendRequested változót, melyet a run() fő hurokjában periódikusan megvizsgálunk. Például:

while (suspendRequested)

speep(1);

Jobb megoldás azonban a később tárgyalandó wait() és notify() használata.

class MyThread extends Thread {

public void requestSuspend() {

suspendRequested = true;

}

private synchronized void checkSuspended()

throws InterruptedException {

while(suspendRequested)

wait();

}

public synchronized void requestResume() {

suspendRequested = false;

notify();

}

public void run() {

while(meg van munka) {

checkSuspend();

hatralevo munka elvegzese;

}

}

privat boolean suspendRequested;

}

A wait(), notify() funkcióját lásd később. A synchronized használatára pedig azért van szükség, mert a fenti két metódus csak szinkronizált metódusokban vagy blokkokban használható.

**A szálak prioritása**

A szálak prioritása a **setPriority()** metódussal állítható. Ez a **Thread** osztály metódusa, s általános alakja:

final void setPriority(int level)

Itt level a prioritási szintet jelenti a hívó szál számára. A szint értékének a **MIN\_PRIORITY** és a **MAX\_PRIORITY** között kell lenni. Jelenleg ez 1 és 10. A default prioritási szint **NORM\_PRIORITY**, ami jelenleg 5. Ezek a prioritások mint a **Thread** osztály **final** változói vannak definiálva.

A jelenlegi prioritás a **getPriority()** metódussal állapítható meg, amelynek alakja:

final int getPriority()

A prioritás használatával azonban vigyáznunk kell, mivel a Java implementációk a különböző operációs rendszerek alatt radikálisan eltérően viselkednek a szálak prioritását illetően. A Windows 95 úgy-ahogy az elvártak szerint viselkedik, de pl. a Solaris esetén már problémák vannak a preemptive viselkedéssel. Ezért a platformfüggetlen viselkedés érdekében célszerű, ha a szálakat úgy használjuk, hogy azok önként lemondjanak a vezérlési jogról.

Az alábbi példa a szálak prioritására mutat be egy programot.

// Demonstrate thread priorities.

class clicker implements Runnable {

volatile int click = 0;

Thread t;

private boolean running = true;

public clicker(int p) {

t = new Thread(this);

t.setPriority(p);

}

public void run() {

while (running) {

fclick();

}

}

public void fclick() {

click++;

}

public void stop() {

running = false;

}

public void start() {

t.start();

}

}

class HiLoPri {

public static void main(String args[]) {

Thread.currentThread().setPriority(Thread.MAX\_PRIORITY);

clicker hi = new clicker(Thread.NORM\_PRIORITY + 2);

clicker lo = new clicker(Thread.NORM\_PRIORITY - 2);

lo.start();

hi.start();

try {

Thread.sleep(10000);

} catch (InterruptedException e) {

System.out.println("Main thread interrupted.");

}

lo.stop();

hi.stop();

// Wait for child threads to terminate.

try {

hi.t.join();

lo.t.join();

} catch (InterruptedException e) {

System.out.println("InterruptedException caught");

}

System.out.println("Low-priority thread: " + lo.click);

System.out.println("High-priority thread: " + hi.click);

}

}

A fenti program Windows 95 alatt a következő eredményt adta:

Low-priority thread: 388872

High-priority thread: 9757630

Azaz a kis prioritású szál a CPU 4%-át, a nagy prioritású pedig 96%-át kapta.

**Szinkronizáció - I.**

Ha kettő vagy több szál férhet hozzá ugyanahhoz az erőforráshoz, akkor biztosítani kell, hogy egyidejűleg csak az egyik szál férhessen hozzá. Az ezt biztosító eljárást szinkronizációnak nevezzük. A Java nyelvi szintű szinkronizációt biztosít a **synchronized** kulcsszó használatával.

Ha egy metódust kiegészítünk a **synchronized** prefixszel, s egy szál belép ebbe a metódusba, akkor minden más szál várakozni fog ennek a metódusnak ( és minden más synchronized metódusnak) a hivásakor ugyanazon objektum esetén, amíg a metódusba belépett szál azt el nem hagyja. Amikor a szál visszatér a szóbanforgó metódusból, akkor ezzel közvetve felszabadítja a többi szál számára a hozzáférést.

A szinkronizáció jobb megértése céljából elöször tekintsünk egy szinkronizációt nem tartalmazó programot. Ez három osztályt tartalmaz. Az első, a **Callme** egyetlen metódust, a **call()** metódust tartalmazza. Ez a metódus egy **msg** nevű **String**típusú paramétert vesz át, majd kinyomtatja szögletes zárójelek között. A záró szögletes zárójel előtt azonban 1 másodpercre elaltatjuk a jelenlegi szálat a**Thread.sleep(1000)** hívásával.

A következő, **Caller** osztály konstruktora létrehoz egy referenciát a **Callme** egy példányára és egy **Stringre**, melyeket a **target** és **msg** változókban tárol. A konstruktor emellett létrehoz egy új szálat is, amely ennek az objektumnak a **run()** metódusát hívja. Ez a szál azonnal elindítódik. A **Caller** **run()** metódusa hívja a **Callme** **target** példányán a **call()** metódust, átadva annak az **msg** stringet. Végül a **Synch**létrehozza a **Callme** egy példányát, valamint a **Caller** három példányát, melyek mindegyike eltérő stringet specifikál. Az egyes **Caller** objektumok a **Callme** azonos példányát kapják meg.

// This program is not synchronized.

class Callme {

void call(String msg) {

System.out.print("[" + msg);

try {

Thread.sleep(1000);

} catch(InterruptedException e) {

System.out.println("Interrupted");

}

System.out.println("]");

}

}

class Caller implements Runnable {

String msg;

Callme target;

Thread t;

public Caller(Callme targ, String s) {

target = targ;

msg = s;

t = new Thread(this);

t.start();

}

public void run() {

target.call(msg);

}

}

class Synch {

public static void main(String args[]) {

Callme target = new Callme();

Caller ob1 = new Caller(target, "Hello");

Caller ob2 = new Caller(target, "Synchronized");

Caller ob3 = new Caller(target, "World");

// wait for threads to end

try {

ob1.t.join();

ob2.t.join();

ob3.t.join();

} catch(InterruptedException e) {

System.out.println("Interrupted");

}

}

}

A program kimenete a következő:

[Hello[Synchronized[World]

]

]

Amint látható a kimenetek egymásba csúsztak. A **sleep()** hívásával a **call()** metódus lehetővé tette az átkapcsolást egy másik szálra. Ebben a programban semmi nem gátolja meg, hogy ugyanazon objektum, ugyanazon metódusát egyidejűleg hívják. Esetünkben a **sleep()**-et használtuk, hogy a probléma evidens és reprodukálható legyen.

A probléma megoldása egyszerű. A **call()** metódust egészítsük ki egy **synchronized** prefixszel, azaz a program ezen része legyen a következő:

class Callme {

synchronized void call(String msg) {

...

Ekkor programunk kimenete a következő lesz:

[Hello]

[Synchronized]

[World]

Amikor egy metódus vagy a metódusok egy csoportja egy objektum belső állapotát megváltoztatja, akkor használnia kell a **synchronized** kulcsszót a versenyhelyzet elkerülésére. Ne felejtse el, ha egy szál belépett egy szinkronizált metódusba egy példányon, akkor más szál nem tud már belépni ugyanennek a példánynak egyetlen szinkronizált metódusába. Természetesen ezen példány nem szinkronizált metódusai továbbra is hívhatók.

**Szinkronizáció - II.**

A **synchronized** metódusok alkalmazása egyszerű, de nem mindig lehetséges. Ugyanis, ha egy olyan objektumot akar szinkronizálni, melyet nem többszálú végrehajtásra terveztek meg, akkor ez az osztály nem tartalmaz **synchronized** metódusokat. S ha ezt az osztályt egy harmadik személy hozta létre, s nem áll rendelkezésre a forráskód, akkor nehezen tudja utólag kiegészíteni a **synchronized** metódusokkal. Hogyan szinkronizálható ekkor ez az objektum? A megoldás szerencsére egyszerű: Az ezt az objektumot hívó metódusokat egy **synchronized** blokkban kell elhelyezni. A **synchronized** utasítás általános formája:

Synchronized(object) {

// szinkronizálandó utasítások

}

Itt az object egy referencia a szinkronizálandó objektumra. Ha csak egyetlen utasítást akar szinkronizálni, akkor a kapcsos zárójelre nincs is szükség. A szinkronizált blokk biztosítani fogja, hogy a hivatkozott objektum metódusának hivása szinkronizált módon fog végbemenni. A következő példa az előző példa alternatív megoldása.

// This program uses a synchronized block.

class Callme {

void call(String msg) {

System.out.print("[" + msg);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println("Interrupted");

}

System.out.println("]");

}

}

class Caller implements Runnable {

String msg;

Callme target;

Thread t;

public Caller(Callme targ, String s) {

target = targ;

msg = s;

t = new Thread(this);

t.start();

}

// synchronize calls to call()

public void run() {

synchronized(target) { // synchronized block

target.call(msg);

}

}

}

class Synch1 {

public static void main(String args[]) {

Callme target = new Callme();

Caller ob1 = new Caller(target, "Hello");

Caller ob2 = new Caller(target, "Synchronized");

Caller ob3 = new Caller(target, "World");

// wait for threads to end

try {

ob1.t.join();

ob2.t.join();

ob3.t.join();

} catch(InterruptedException e) {

System.out.println("Interrupted");

}

}

}

**Szálak közötti kommunikáció**

Az előző példákban feltétel nélkül blokkoltuk a többi szál felől a hozzáférést bizonyos metódusokhoz. Bár ez hatékony, mégis egy más szinten sokkal hatékonyabb kommunikáció érhető el a szálak között.

Amint már említettük, a többszálúság alkalmazása helyettesíti az eseménykezelő hurkokon alapuló programozást a feladatot diszkrét logikai egységekre bontva. A szál további előnye: kiküszöböli a pollingot, ami felesleges CPU időt igényelne. Tekintsük erre a klasszikus termelő-fogyasztó együttműködést. A termelőt és a fogyasztót is egy-egy szál valósítsa meg. S tegyük fel, hogy a termelőnek mindaddig várakoznia kell, amíg a fogyasztó fel nem dolgozta a termelt adatot. Egy pollingos rendszerben a termelő jelentős CPU időt fog elvesztegetni annak lekérdezésével, hogy a fogyasztó befejezte-e a feladatát. Ugyanez érvényes a fordított irányban is, azaz a fogyasztó is tetemes CPU időt fogyaszthat a pollinggal, arra várakozva, hogy a termelő előállítson egy adatot számára.

A polling elkerülésére Java bevezetett egy elegáns interprocess kommunikációs mechanizmust a **wait()**, **notify()** és **notifyAll()** metódusokkal. Ezek a metódusok mint **final** vannak implementálva az**Object**-ben, tehát minden osztály tartalmazza őket. Mindegyik fenti metódus csak **synchronized** metódusból hívható. Funkciójuk a következő:

* **wait()**azt üzeni annak a szálnak, amelyből hívták, hogy függessze fel működését ("aludjon"), amíg egy másik szál nem szóllítja meg a **notify()** metódussal. Csak szinkronizált metódusból vagy blokkból hívható meg.
* **notify()** véletlenszerűen kiválasztva felébreszt egy szálat azon szálak közül, amelyek a **wait()**-et hívták ezen az objektumon. Csak szinkronizált metódusból vagy blokkból hívható meg.
* **notifyAll()** az összes szálat felébreszti, amely a **wait()**-et hívta ugyanazon objektumon. A felébresztett szálak közül a legnagyobb prioritású fog futni. Csak szinkronizált metódusból vagy blokkból hívható meg.

Ezek a metódusok az **Object** osztályon belül az alábbiak szerint vannak deklarálva:

final void wait()

final void notify()

final void notifyAll()

A következő program, amely nem használja a fenti metódusokat, hibásan implementálja a fogyasztó/termelő folyamatot. Négy osztályt tartalmaz. **Q** a szinkronizálandó várakozási sor, **Producer** szál alapú objektum, amely a várakozási sor számára adatokat termel, **Consumer** szál alapú objektum, amely a várakozási sorból adatokat vesz ki, **PC** pedig egy olyan osztály, amely a fenti három osztály egy-egy példányát létrehozza.

// An incorrect implementation of a producer and consumer.

class Q {

int n;

synchronized int get() {

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

return n;

}

synchronized void put(int n) {

this.n = n;

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

}

}

class Producer implements Runnable {

Q q;

Producer(Q q) {

this.q = q;

new Thread(this, "Producer").start();

}

public void run() {

int i = 0;

while(true) {

q.put(i++);

}

}

}

class Consumer implements Runnable {

Q q;

Consumer(Q q) {

this.q = q;

new Thread(this, "Consumer").start();

}

public void run() {

while(true) {

q.get();

}

}

}

class PC {

public static void main(String args[]) {

Q q = new Q();

new Producer(q);

new Consumer(q);

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

}

}

Jóllehet a put() és a get() metódusok a **Q**-n szinkronizálva vannak, semmi nem fogja megakadályozni a fogyasztót, hogy egy értéket kétszer is kivegyen onnan. Ezért hibás kimenetet kaphat (a pontos kimenet a proceszor sebességétől függ):

Put: 1

Got: 1

Got: 1

Got: 1

Got: 1

Put: 2

Put: 3

Put: 4

Put: 5

Put: 6

Got: 6

Amint látható, a termelő elhelyezett egy 1-est, a fogyasztó szála elindult, s ezt négyszer is kivette. Ezután a termelő "feléledt", s 2-től 6-ig adatokat tett be a várakozási sorba, anélkül hogy azokat a fogyasztó kivehette volna.

Hibátlan, de alacsony hatékonyságú megoldást biztosíthatunk, ha a várakozási sorban bevezetünk egy tele szemafort, amit a termelő igaz, a fogyasztó pedig hamis állapotba állít adat beírásakor, illetve kiolvasásakor. A szemafor folytonos lekérdezese (polling) azonban tetemes CPU időt használ fel. Ezért alacsony hatékonyságú ez, az alábbiakban bemutatott megoldás:

//Correct, however, low performance solution

class Q1 {

int n;

boolean tele = false;

synchronized int get() {

if(!tele) return -1;

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

tele = false;

return n;

}

synchronized int put(int n) {

if(tele) return -1;

this.n = n;

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

tele = true;

return 1;

}

}

class Producer1 implements Runnable {

Q1 q;

Producer1(Q1 q) {

this.q = q;

new Thread(this, "Producer").start();

}

public void run() {

int i = 0;

while(true) {

if(q.put(i)!=-1) i++;

}

}

}

class Consumer1 implements Runnable {

Q1 q;

Consumer1(Q1 q) {

this.q = q;

new Thread(this, "Consumer").start();

}

public void run() {

while(true) {

q.get();

}

}

}

class PC1 {

public static void main(String args[]) {

Q1 q = new Q1();

new Producer1(q);

new Consumer1(q);

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

}

}

A helyes és hatékony megoldás a **wait()** és **notify()** mindkét irányú alkalmazásával érhető el, melyet az alábbi program szemléltet:

// A correct implementation of a producer and consumer.

class Q2 {

int n;

boolean valueSet = false;

synchronized int get() {

if(!valueSet)

try {

wait();

} catch(InterruptedException e) {

System.out.println("interruptedException caught");

}

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

valueSet = false;

notify();

return n;

}

synchronized void put(int n) {

if(valueSet)

try {

wait();

} catch(InterruptedException e) {

System.out.println("interruptedException caught");

}

this.n = n;

valueSet = true;

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

notify();

}

}

class Producer2 implements Runnable {

Q2 q;

Producer2(Q2 q) {

this.q = q;

new Thread(this, "Producer").start();

}

public void run() {

int i = 0;

while(true) {

q.put(i++);

}

}

}

class Consumer2 implements Runnable {

Q2 q;

Consumer2(Q2 q) {

this.q = q;

new Thread(this, "Consumer").start();

}

public void run() {

while(true) {

q.get();

}

}

}

class PC2 {

public static void main(String args[]) {

Q2 q = new Q2();

new Producer2(q);

new Consumer2(q);

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

}

}

A szinkronizált **get()**-en belül hívjuk a **wait()**-et. Ez felfüggeszti a működést mindaddig, amíg a **Producer** nem jelzi, hogy rendelkezésre áll az adat. Amikor ez megtörténik, akkor a **get()** újraéled. Amint megkapta az adatot, a **get()** hívja a **notify()**-t. Ez közli a **Producer**-rel, hogy újabb adatot tehet a várakozási sorba. A **put()**-on belül a **wait()** felfüggeszti a végrehajtást, amíg a **Consumer** el nem fogyasztja az adatot. Amikor a végrehajtás továbbindítódik, akkor a következő adat elhelyeződik a sorban, s hívódik a **notify()**. Ez közli a **Consumer**-rel, hogy újabb adat rendelkezésre áll ...

A most már helyesen működő program kimenete:

Put: 1

Got: 1

Put: 2

Put: 2

Put: 3

Got: 3

**Deadlock**

A többszálú rendszerekben könnyen előfordulhat, hogy keresztben egymásra várakozó szálak miatt a program végrehajtása végtelen időre elakad, azaz ún.deadlock lép fel. A deadlock egy bonyolult hiba, amit nehéz megkeresni, mivel indeterminisztikusan lép fel, s több mint két szálat is érinthet.

Az alábbi programban a végrehajtás folyamán deadlock lép fel, amelyből csak a CTRL C-vel lehet kilépni.

// An example of deadlock.

class A {

synchronized void foo(B b) {

String name = Thread.currentThread().getName();

System.out.println(name + " entered A.foo");

try {

Thread.sleep(1000);

} catch(Exception e) {

System.out.println("A Interrupted");

}

System.out.println(name + " trying to call B.last()");

b.last();

}

synchronized void last() {

System.out.println("Inside A.last");

}

}

class B {

synchronized void bar(A a) {

String name = Thread.currentThread().getName();

System.out.println(name + " entered B.bar");

try {

Thread.sleep(1000);

} catch(Exception e) {

System.out.println("B Interrupted");

}

System.out.println(name + " trying to call A.last()");

a.last();

}

synchronized void last() {

System.out.println("Inside A.last");

}

}

class Deadlock implements Runnable {

A a = new A();

B b = new B();

Deadlock() {

Thread.currentThread().setName("MainThread");

Thread t = new Thread(this, "RacingThread");

t.start();

a.foo(b); // get lock on a in this thread.

System.out.println("Back in main thread");

}

public void run() {

b.bar(a); // get lock on b in other thread.

System.out.println("Back in other thread");

}

public static void main(String args[]) {

new Deadlock();

}

}

A program kimenete pedig:

MainThread entered A.foo

RacingThread entered B.bar

MainThread trying to call B.last()

Racing Thread trying to call A.last()

A program értelmezése:A classnak és B classnak is két szinkronizált metódusa van.A szinkronizált metódusokra az jellemző,ha egy objektumon már valameily szál haszál egy szinkronizált metódust,akkor semelyik más szál nem használhat addig szinkronizált metódust azon az objektumon,csak ha már az első szál végzett és nem blokkol tovább.Itt a problémát az okozza ,hogy Az A objektum foo metódusa elindul majd kap egy sleepet, ami után a B objektum bar metódusa hívódik meg ami szintén szinkronizált, majd ez is kap egy sleepet, tehát látható,hogy az A objektum blokkolja a szinkronizált metódusokat, illetve a B objektum is blokkolja a rajta végezhető szinkronizált metódusokat.Mivel a sleepek után az A objektum folytatná azzal,hogy az említett B objektum egy szinkronizált metódusát hívná,de nem tudja, ezért B objektumú szál kapja a cpu-t, de ő pedig az A objektum egy másik szinkronizált metódusát hívná, ami szintén nem lehetséges,ezért ezek elkezdtek egymásra várni.Megoldás lehet ,ha például a last metódusokat nem szinkronizáltra definiáljuk,mivel a nem szinkronizált metódusokhoz bármely szál hozzáférhet konkurensen.

**Animáció**

**Első példa**

Az alábbi példában egy pirosan kitöltött kört mozgatunk átlósan a kereten. A kör pozicióját egy szálban állítjuk, s a paint() metódus mindig az aktuális pozícióban rajzolja ki a kört.

import java.awt.\*;

import java.awt.event.\*;

class animate extends Frame implements Runnable, WindowListener {

Thread t;

int x, xp;

animate(String s) {

super(s);

addWindowListener(this);

t = new Thread(this, "animate");

t.start();

}

public void windowClosing(WindowEvent e) {

this.dispose();

}

public void windowActivated(WindowEvent e) {}

public void windowClosed(WindowEvent e) {}

public void windowDeactivated(WindowEvent e) {}

public void windowDeiconified(WindowEvent e) {}

public void windowIconified(WindowEvent e) {}

public void windowOpened(WindowEvent e) {}

public void paint(Graphics g) {

g.setColor(Color.white);

g.fillRect(xp,xp,20,20);

g.setColor(Color.red);

g.fillOval(x,x,20,20);

}

public void run() {

xp=0;

try {

while(true) {

x=0;

for(int i = 0; i < 80; i++) {

x+=2;

repaint();

t.sleep(40);

xp=x;

}

}

} catch (InterruptedException e) {

System.out.println("Main thread interrupted.");

}

}

public static void main(String args[]) {

animate f = new animate("Animation Frame");

f.setLocation(50,50);

f.setSize(200,200);

f.setVisible(true);

}

}

**Első alternatíva**

Az előző példában zavaró villogás (flikker) lép fel az animáció során. Ennek az az oka, hogy a repaint() végrehajtásakor a virtuális gép az update() metódust hívja meg, ami először törli a képernyőt majd hívja a paint() metódust. Jobb lenne, ha a teljes képernyő törlése helyett csak azt a részt kellene törölni, ami feltétlenül szükséges. Ez megoldható, ha átírjuk az update() metódust. Azaz a szükséges törlést magunk elvégezzük a paint() metódusban, s az update() metódust átdefiniáljuk, hogy csak a paint() metódust hívja. Ezt mutatja be a következő megoldás.

import java.awt.\*;

import java.awt.event.\*;

class animate1 extends Frame implements Runnable, WindowListener {

Thread t;

int x, xp;

animate1(String s) {

super(s);

addWindowListener(this);

t = new Thread(this, "animate");

t.start();

}

public void windowClosing(WindowEvent e) {

this.dispose();

}

public void windowActivated(WindowEvent e) {}

public void windowClosed(WindowEvent e) {}

public void windowDeactivated(WindowEvent e) {}

public void windowDeiconified(WindowEvent e) {}

public void windowIconified(WindowEvent e) {}

public void windowOpened(WindowEvent e) {}

public void paint(Graphics g) {

g.setColor(Color.white);

g.fillRect(xp,xp,20,20);

g.setColor(Color.red);

g.fillOval(x,x,20,20);

}

public void update(Graphics g) {

paint(g);

}

public void run() {

xp=-20;

try {

while(true) {

x=-20;

for(int i = 0; i < 80; i++) {

x+=2;

repaint();

t.sleep(40);

xp=x;

}

}

} catch (InterruptedException e) {

System.out.println("Main thread interrupted.");

}

}

public static void main(String args[]) {

animate1 f = new animate1("Animation Frame");

f.setLocation(50,50);

f.setSize(200,200);

f.setVisible(true);

}

}

**Második alternatíva**

Az előbbi megoldásban a villogás mértéke csökkent, de nem szünt teljesen meg. A legjobb megoldást az ún. kettős pufferelés adja. Ekkor nem közvetlenül a képernyőre, hanem a háttérbe rajzolunk, s a háttérből átmásoljuk a képet az előtérbe. Az átmásolás sokkal gyorsabb mint a rajzolás, ez biztosítja a jobb minőséget.

import java.awt.\*;

import java.awt.event.\*;

class animate2 extends Frame implements Runnable, WindowListener {

Thread t;

int x;

animate2(String s) {

super(s);

addWindowListener(this);

t = new Thread(this, "animate");

t.start();

}

public void windowClosing(WindowEvent e) {

this.dispose();

}

public void windowActivated(WindowEvent e) {}

public void windowClosed(WindowEvent e) {}

public void windowDeactivated(WindowEvent e) {}

public void windowDeiconified(WindowEvent e) {}

public void windowIconified(WindowEvent e) {}

public void windowOpened(WindowEvent e) {}

Image offScreenImage;

Graphics offScreenGraphics;

public void paint(Graphics g) {

if(offScreenGraphics==null) {

offScreenImage=createImage(200,200);

offScreenGraphics = offScreenImage.getGraphics();

}

offScreenGraphics.setColor(Color.white);

offScreenGraphics.fillRect(0,0,200,200);

offScreenGraphics.setColor(Color.red);

offScreenGraphics.fillOval(x,x,20,20);

g.drawImage(offScreenImage,0,0,this);

}

public void update(Graphics g) {

paint(g);

}

public void run() {

try {

while(true) {

x=-20;

for(int i = 0; i < 200; i++) {

repaint();

x++;

t.sleep(20);

}

}

} catch (InterruptedException e) {

System.out.println("Main thread interrupted.");

}

}

public static void main(String args[]) {

animate2 f = new animate2("Animation Frame");

f.setLocation(50,50);

f.setSize(200,200);

f.setVisible(true);

}

}

**További példák**

A következő példában a mozgó gömb megállítható és továbbindítható. Emellett, az ablak nagyítását, kicsinyítését mindig követi a gömb úthossza, az mindig középen eseik lefelé.

import java.awt.\*;

import java.applet.\*;

import java.awt.event.\*;

public class myFrame extends Frame

implements Runnable, ActionListener {

Thread myThread;

int ypos, hmax;

boolean animation;

Button startB, stopB;

myCanvas mC;

Panel p;

public myFrame(){

super();

ypos = 0;

hmax =100;

animation = false;

this.setLayout(new BorderLayout());

mC = new myCanvas();

this.add("Center",mC);

p = new Panel();

p.setLayout(new FlowLayout());

this.add("South",p);

startB = new Button("Start");

stopB = new Button("Stop");

p.add(startB);

p.add(stopB);

startB.addActionListener(this);

stopB.addActionListener(this);

this.addWindowListener(new WindowAdapter() {

public void windowClosing(WindowEvent we) {

System.exit(0);

}

} );

if(myThread == null){

myThread = new Thread(this, "Falling");

myThread.start();

}

}

public void actionPerformed(ActionEvent e) {

if(e.getSource() == startB)

animation = true;

else

animation = false;

}

public void run(){

try {

while (myThread != null) {

if(animation) {

if(ypos > hmax) ypos=0;

ypos++;

mC.repaint();

}

myThread.sleep(40);

}

} catch (InterruptedException e) {

}

}

public static void main(String args[]){

myFrame f = new myFrame();

f.setSize(200, 200);

f.setLocation(0,0);

f.setVisible(true);

}

class myCanvas extends Canvas {

Image offScreenImage;

Dimension offScreenSize;

Graphics offScreenGraphics;

public void paint (Graphics g) {

Dimension d = this.getSize();

if((this.offScreenImage == null) ||

(d.width != this.offScreenSize.width)

||(d.height != this.offScreenSize.height)) {

this.offScreenImage =

this.createImage(d.width, d.height);

this.offScreenSize = d;

this.offScreenGraphics =

this.offScreenImage.getGraphics();

}

hmax = d.height;

this.offScreenGraphics.clearRect(0, 0, d.width, d.height);

this.offScreenGraphics.drawRect(1,1,d.width-2, d.height-2);

this.offScreenGraphics.fillOval(d.width/2 - 25,ypos,50,50);

g.drawImage(this.offScreenImage, 0, 0, null);

}

public void update(Graphics g) {

paint(g);

}

}

}

A következő példa egy képet mozgat a képernyőn. Mivel paramétereket olvas be a html lapról, csak a thread4.htm fájllal futtatva működik.

import java.awt.\*;

import java.applet.\*;

public class thread4 extends Applet implements Runnable {

Thread animate;

Image kep;

int position=0;

String picName,picSizex,picSizey;

int picx,picy;

public void init(){

setLayout(null);

picName = getParameter("Name");

picSizex = getParameter("Width");

picSizey = getParameter("Height");

picx = Integer.valueOf(picSizex).intValue();

picy = Integer.parseInt(picSizey);

kep = getImage(getCodeBase(),picName);

resize(picx,picy);

}

public void start(){

if(animate == null){

animate = new Thread(this, "Animation");

animate.start();

}

}

public void run(){

while (animate != null) {

position=position+1;

repaint();

try {

animate.sleep(20);

} catch (InterruptedException e) {}

}

}

public void paint(Graphics g) {

update(g);

}

Dimension offDimension;

Image offImage;

Graphics offGraphics;

public void update(Graphics g){

Dimension d=size();

if ((offGraphics == null)

|| (d.width != offDimension.width)

|| (d.height != offDimension.height)) {

offDimension = d;

offImage = createImage(d.width, d.height);

offGraphics = offImage.getGraphics();

offGraphics.setColor(getBackground());

}

offGraphics.fillRect(0,0,picx,picy);

offGraphics.drawImage(kep,picx-position,0,this);

offGraphics.drawImage(kep,2\*picx-position,0,this);

if(position>2\*picx){position=picx;}

g.drawImage(offImage, 0, 0, this);

}

}

http://www.theorphys.elte.hu/fizinf/HaloAdat/tananyag/java/A\_SZALAK/index.html