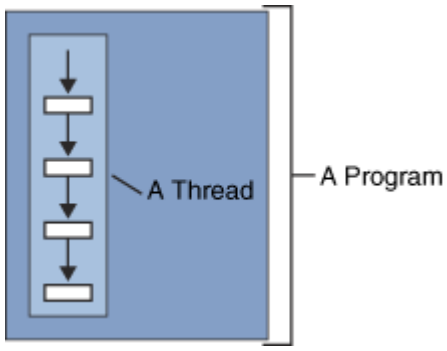
19.Programszálak kezelése

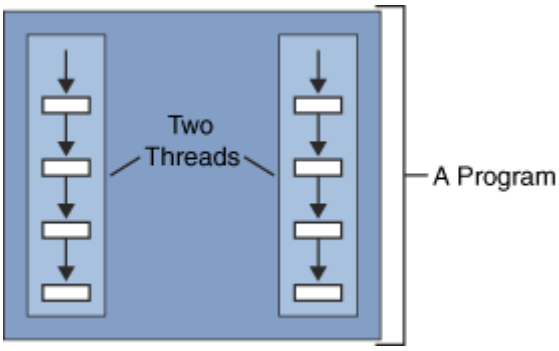
A programok jelentős része soros végrehajtású. Van kezdetük, egy végrehajtandó rész és egy befejezés. A program csak bizonyos időszakaszokban fut, hiszen a processzor már programokat is futtat.

A szál (*thread*) hasonló a soros programhoz. A szálnak is van kezdete, egy végrehajtandó része és egy befejezése. A szál is csak bizonyos időszakaszokban fut. De a szál maga nem egy program, a szálat nem lehet önállóan futtatni. Inkább úgy tekinthetjük, hogy a program részeként fut. A következő ábra megmutatja a kapcsolatot.



Definíció: A szál egy egyedülálló irányító folyamat a programon belül.

Persze nem arra használjuk a szálakat, hogy csak egyet alkalmazunk, inkább több szálat futtatva egy időben különböző feladatokat elvégezve egyetlen programban. A következő kép ezt illusztrálja.



A böngésző programunk is egy példa a többszálú alkalmazásra. Egy általános böngészőben egy oldalon belül egyszerre történik egy kép, applet letöltése, mozgó animáció vagy hang lejátszása, az oldal nyomtatása a háttérben, miközben egy új oldalt tölt be vagy éppen három rendező algoritmus versenyét tekintheti meg.

Néha a szálat könnyűsúlyú processznek nevezik, mert egy teljes programon belül fut, annak lefoglalt erőforrásait és a futtató környezetét használja.

Mint irányító folyamat, a szálnak is kell saját erőforrásokkal rendelkeznie. Rendelkezni kell egy végrehajtó veremmel és a programszámlálóval. Ebben a környezetbe fut a szál kódja. Néhol ezt a környezetet a szál fogalom szinonimájának nevezik.

A szál programozás nem egyszerű. Ha mégis szálakat kell használni, akkor érdemes a magas-szintű szál API-kat használni. Példaképpen, ha a programban egy feladatot többször kell elvégezni, érdemes a *java.util.Timer* osztályt alkalmazni. A *Timer* osztály időzítéses feladatoknál is hasznos lehet. Erre hamarosan látunk példát.

19.1. A *Timer* és a *TimerTask* osztály

Ebben a részben az időzítők használatát tárgyaljuk. A *Timer* osztály a *java.util* csomag része, a *TimerTask* osztály példányai ütemezését végzi. *Reminder.java* egy példa arra, hogy használjuk az időzítőket a késleltetett futtatáshoz:

```
import java.util.Timer;
import java.util.TimerTask;

public class Reminder {
    Timer timer;

    public Reminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(), seconds*1000);
    }

    class RemindTask extends TimerTask {
        public void run() {
            System.out.println("Lejárt az idő!");
            timer.cancel(); //A timer szál megszüntetése
        }
    }

    public static void main(String args[]) {
        new Reminder(5);
        System.out.println("Munka ütemezve.");
    }
}
```

Indulás után ezt látjuk:

```
Munka ütemezve.
```

Öt másodper múlva ezt látjuk:

```
Lejárt az idő!
```

A program egy példa arra, hogyan kell példányosítani és ütemezni a szálakat:

- *TimerTask* leszármazott osztály példányosítása. A *run* metódus tartalmazza a futtatás során végrehajtandó kódot. A példába ezt az leszármazott osztály *RemindTask*-ként neveztük el.
- Létrehozzuk a *Timer* osztály egy példányát.
- Létrehozunk egy időzítő objektumot (*new RemindTask()*).
- Beütemezzük az időzítőt. Ez a példa a *schedule* metódust használja, amelynek paraméterei az időzítő és késleltetés ezredmásodpercben (5000).

Egy másik lehetőség az ütemezésre, hogy az indítási időpontot adjuk meg. Példaképpen a következő kód 23:01-re ütemezi a végrehajtást:

```
Calendar calendar = Calendar.getInstance();
calendar.set(Calendar.HOUR_OF_DAY, 23);
calendar.set(Calendar.MINUTE, 1);
calendar.set(Calendar.SECOND, 0);
Date time = calendar.getTime();

timer = new Timer();
timer.schedule(new RemindTask(), time);
```

19.1.1 Időzített szálak leállítása

Alapesetben a program addig fut, amíg az időzítő szál is fut. Négy leállítási módszer közül választhatunk:

- A *cancel* metódust hívjuk meg. Ezt a program bármelyik pontján megtehetjük, mint a példa *run* metódusában.
- Az időzítő szálát démon szállá kell tenni a következőképpen: *new Timer(true)*. Ha a programban csak démon szálak maradnak, akkor a program kilép.
- Ha befejeződtek az ütemezések, el kell távolítani a *Timer* objektumhoz tartozó mutatókat. Ezzel a szál is megszűnik.
- A *System.exit* metódust hívjuk meg, amely leállítja a programot (benne a szálakat is).

A *Reminder* példa az első módszert használja, a *cancel* metódust hívja meg a *run* metódusból. A démon szálként való létrehozás a példánkban nem működne, mert a programnak futni kell a szál lefutása után is.

19.1.2 Ismételt futtatás

Ebben a példába másodpercenként ismétli a kód futását:

```
public class AnnoyingBeep {
    Toolkit toolkit;
    Timer timer;

    public AnnoyingBeep() {
        toolkit = Toolkit.getDefaultToolkit();
        timer = new Timer();
        timer.schedule(new RemindTask(),
                       0,        //kezdeti késleltetés
                       1*1000);  //ismétlési ráta
    }

    class RemindTask extends TimerTask {
        int numWarningBeeps = 3;
        public void run() {
            if (numWarningBeeps > 0) {
                toolkit.beep();
                System.out.println("Síp!");
                numWarningBeeps--;
            } else {
                toolkit.beep();
                System.out.println("Idő lejárt!");
            //timer.cancel(); // Nem szükséges
                            // mert van System.exit is.
                System.exit(0);   // Leállítja AWT szálat
                            // (és minden mást)
            }
        }
    }
    ...
}
```

Futtatás közben ez lesz a kimenet:

```
Munka ütemezve.
Síp!
Síp!
Síp!
Idő lejárt!
```

Az *AnnoyingBeep* program három paraméteres *schedule* metódust használ, hogy meghatározza a taszk másodpercenkénti indítását. A *Timer* metódus változatai:

- *schedule(TimerTask task, long késleltetés, long gyakoriság)*
- *schedule(TimerTask task, Date idő, long gyakoriság)*
- *scheduleAtFixedRate(TimerTask task, long késleltetés, long gyakoriság)*
- *scheduleAtFixedRate(TimerTask task, Date kezdetiIdő, long gyakoriság)*

A *schedule* metódust akkor használjuk, ha a többszörösen futtatott taszk ismétlési ideje számít, a *scheduleAtFixedRate* metódust, ha az ismétlések időben kötődnek egy pontos időhöz. A példában is a *schedule* metódust alkalmaztuk, amitől 1 másodperces intervallumokban sípol a gép. Ha valamelyik sípszó késik, akkor a következők is késlekednek. Ha úgy döntünk, hogy a program 3 másodperc múlva kilép az első sípszó után – ami azt is eredményezheti, hogy a két sípszó kisebb időközzel szólal meg, ha késlekedés lép fel – akkor a *scheduleAtFixedRate* metódust használjuk.

19.2. Szálak példányosítása

Ha a fenti megközelítés nem alkalmas a feladat ellátására, akkor saját szál példányosítása lehet a megoldás. Ez a fejezet elmagyarázza, hogyan lehet a szál *run* metódusát egyénivé alakítani.

A *run* metódusban van, amit a szál ténylegesen csinál, ennek a kódja határozza meg a futást. A szál *run* metódussal mindent meg lehet tenni, amit a Java programnyelvben le lehet írni. Pl.: prímszámokkal való számolás, rendezés, animáció megjelenítés.

A *Thread* osztály példányosításával egy új szál jön létre, ami alapból nem csinál semmit, de lehetőség van a leszármazott osztályban tartalommal megtölteni.

19.2.1 *Thread* leszármazott és a *run* felülírása

Az első lépés a szálak testreszabásánál, hogy *Thread* osztály leszármazottját létrehozzuk, és az üres *run* metódusát felülírjuk, hogy csináljon is valamit. Nézzük meg a *SimpleThread* osztályt, amely ezt teszi:

```
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
```

```
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("KÉSZ! " + getName());
    }
}
```

A *SimpleThread* osztály első metódusa egy konstruktor, amely paraméterként egy *String*-et fogad. A konstruktor meghívja az ősosztály konstruktorát, ami azért érdekes számunkra, mert az beállítja a szál nevét. Ezt a nevet használjuk a későbbi programban a szál felhasználói azonosítására. A nevet később a *getName* metódussal tudjuk lekérdezni.

A *SimpleThread* osztály következő metódusa a *run*. Itt történik a *Thread* érdemi működése. Ebben az esetben egy tizes *for* ciklust tartalmaz. Minden ciklusban kiírja az iteráció számát, a *Thread* nevét, és utána altatásra kerül véletlenszerűen 1 másodperces határon belül. Ha végzett, a *KÉSZ!* feliratot írja ki a szál nevével együtt. Ennyi a *SimpleThread* osztály. Használjuk fel ezt a *TwoThreadsTest*-ben.

A *TwoThreadsTest* osztály *main* metódusába két *SimpleThread* szálat hozunk létre: Jamaica és Fuji. (Ha valaki nem tudná eldönteni hova menjen nyaralni, akkor használja ezt a programot.)

```
public class TwoThreadsTest {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
```

A *main* metódus azonnal mindkét szálat létrehozza a start metódus meghívásával, amely a *run* metódust hívja meg. Fordítás és futtatás után kibontakozik az úti cél. A kimenet hasonló kell, hogy legyen:

```
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
3 Jamaica
4 Jamaica
4 Fiji
5 Jamaica
5 Fiji
6 Fiji
6 Jamaica
7 Jamaica
7 Fiji
8 Fiji
9 Fiji
8 Jamaica
DONE! Fiji——Look out, Fiji, here I come!
9 Jamaica
DONE! Jamaica
```

Figyeljük meg, hogy a szálak kimenete mennyire független egymástól. Ennek az az oka, hogy a *SimpleThread* szálak egyidejűleg futnak. Mindkét *run* metódus fut, és mindkettőnek van saját kimenete ugyanakkor. Ha a ciklus befejeződik, a szálak leállnak és meghalnak.

19.2.2 *Runnable* interfész példányosítása

A következő *Clock* applet a jelenlegi időt mutatja, és azt másodpercenként frissíti. A frissítés állandó, mivel az óra kijelzése egy saját szálon fut.

Az applet más módszert használ, mint a *SimpleThread*. Itt ugyanis a szülőosztály csak az *Applet* lehet, de így az egyszeres öröklődés miatt a *Thread* már nem lehet szülő. Ezért a példa *Thread* leszármazott létrehozása helyett egy *Runnable* interfészt implementál, ezért a *run* metódust ebben definiálja. A *Clock* létrehoz egy szálat, amelynek a frissítés a célja.

```
import java.awt.*;
import java.util.*;
import java.applet.*;
import java.text.*;

public class Clock extends java.applet.Applet implements
Runnable {
    private volatile Thread clockThread = null;
    DateFormat formatter;      // Formats the date displayed
    String lastdate;           // String to hold date displayed
    Date currentDate;          // Used to get date to display
    Color numberColor;         // Color of numbers
    Font clockFaceFont;
    Locale locale;

    public void init() {
        setBackground(Color.white);
        numberColor = Color.red;
        locale = Locale.getDefault();
        formatter =
            DateFormat.getDateTimeInstance(DateFormat.FULL,
            DateFormat.MEDIUM, locale);
        currentDate = new Date();
        lastdate = formatter.format(currentDate);
        clockFaceFont = new Font("Sans-Serif",
                                 Font.PLAIN, 14);
        resize(275,25);
   }

    public void start() {
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }
```

```
    public void run() {
        Thread myThread = Thread.currentThread();
        while (clockThread == myThread) {
            repaint();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e){ }
        }
    }

    public void paint(Graphics g) {
        String today;
        currentDate = new Date();
        formatter =
             DateFormat.getDateTimeInstance(DateFormat.FULL,
             DateFormat.MEDIUM, locale);
        today = formatter.format(currentDate);
        g.setFont(clockFaceFont);

        // Erase and redraw
        g.setColor(getBackground());
        g.drawString(lastdate, 0, 12);

        g.setColor(numberColor);
        g.drawString(today, 0, 12);
        lastdate = today;
        currentDate=null;
    }

    public void stop() {
        clockThread = null;
    }
}
```

A *Clock* applet *run* metódusa addig nem lép ki a ciklusból, amíg a böngésző le nem állítja. Minden ciklusban az óra kimenete újrarajzolódik. A *paint* metódus lekérdezi az időt, formázza és kijelzi azt.

Kell-e a *Runnable* interfész?

Két módszert ismertettünk a *run* metódus előállítására.

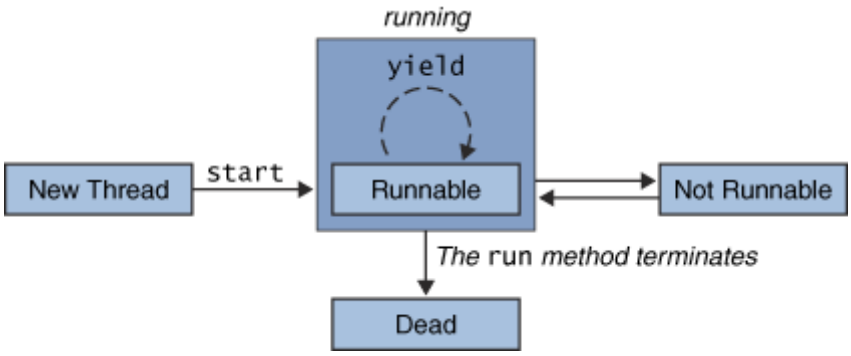
- Leszármazott *Thread* osztály létrehozása és a *run* metódus felülírása.
- Egy olyan osztály létrehozása, amelyik megvalósítja a *Runnable* interfészt és így a *run* metódust is. Ebben az esetbe a *Runnable* objektum szolgáltatja a *run* metódust a szálnak. Ezt feljebb láthatjuk.

Megjegyzés: A *Thread* is megvalósítja a *Runnable* interfészt.

Ahhoz, hogy egy böngészőben futhasson a *Clock* osztálynak az *Applet* osztály leszármazottjának kell lennie. Továbbá a *Clock* applet folyamatos kijelzése miatt egy szálra van szüksége, hogy ne a processzt vegye át, amiben fut. (Néhány böngésző figyelhet arra, hogy az appletek saját szálat kapjanak, elkerülendő, hogy a hibás appletek elvegyék a böngésző szálát. De erre nem építhetünk egy applet írásánál. Az általunk írt appletnek tudnia kell a saját szál létrehozását, ha olyan munkát végez.) De mivel a Java nyelv nem engedélyezi a több osztályokon át való öröklődést, a *Clock* nem lehet egyszerre a *Thread* és az *Applet* osztály leszármazottja. Így a *Clock* osztálynak egy *Runnable* interfészen keresztül kell a szálak viselkedését felvennie.

19.3. Programszál életciklusa

A következő ábra bemutatja a szál különböző állapotait, amibe élete során tud kerülni, és illusztrálja, hogy melyik metódusok meghívása okozza az átmenetet egy másik fázisba. Az ábra nem egy teljes állapotdiagram, inkább csak egy áttekintés a szál életének fontosabb és egyszerűbb oldalairól. A továbbiakban is a *Clock* példát használjuk, hogy bemutassuk a szál életciklusát különböző szakaszaiban.



19.3.1 Programszál létrehozása

Az alkalmazás, melyben egy szál fut, meghívja a szál *start* metódusát, amikor a felhasználó a programot elindítja. A *Clock* osztály létrehozza a *clockThread* szálat a *start* metódusában, ahogy a lenti programkód bemutatja:

```
public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
}
```

Miután a kiemelt utasítás végrehajtódik, a *clockThread* az „új szál” (*New Thread*) állapotban van. Egy szál ebben az állapotában csupán egy üres szál, semmiféle rendszererőforrás nincs hozzárendelve. Ebben az állapotban csak elindítható a szál. A *start* metóduson kívül bármilyen metódus meghívása értelmetlen, és *IllegalThreadStateException* kivételt eredményez. (Alapértelmezetten a rendszer mindig ezt a hibaüzenetet dobja, ha egy olyan metódus kerül meghívásra, amelyik a szál pillanatnyi állapotában nem engedélyezett.)

Érdemes megjegyezni, hogy a *Clock* típusú példány az első paraméter a szál konstruktorban. Ez a paraméter meg kell, hogy valósítsa a *Runnable* interfészt, csak így lehet a konstruktor paramétere. A *Clock* szál a *run* metódust a *Runnable* interfésztől örökli. A konstruktor második paramétere csupán a szál neve.

19.3.2 Programszál elindítása

Most gondoljuk át, mit eredményez a programkódban a *start* metódus:

```
public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
}
```

A *start* metódus létrehozza a szükséges rendszererőforrásokat a szál futtatásához, előkészíti a szálat a futáshoz, és meghívja a szál *start* metódusát. A *clockThread run* metódusa a *Clock* osztályban van definiálva.

Miután a *start* metódus visszatér, a szál fut. Mégis, ennél azért kicsit komplexebb a dolog. Ahogy az előző ábra is mutatja, egy elindított szál a futtatható állapotba kerül. Sok számítógép csak egy processzorral rendelkezik, így lehetetlen, hogy minden futási állapotban lévő szál egyszerre fusson. Ezért a Java futtatórendszernek implementálnia kell egy ütemezési sémát, ami elosztja a processzor erőforrásait az összes futó szál között. Szóval egy bizonyos időpontban, egy futó szál valószínűleg vár, hogy ő kerüljön sorra a CPU ütemezésében.

Még egyszer a *Clock run* metódusa:

```
public void run() {
   Thread myThread = Thread.currentThread();
   while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // The VM doesn't want us to sleep anymore,
            // so get back to work
        }
    }
}
```

A *Clock run* metódusa addig lépked a ciklusban, amíg a *clockThread == myThread* feltétel nem lesz igaz.

A cikluson belül a szál újra kirajzolja az appletet, majd utasítja a szálat, hogy kerüljön alvó állapotba 1 másodpercre. A *repaint* metódus végső soron meghívja a *paint* metódust, ami az aktuális frissítéseket végzi a program megjelenítési felületén. A *Clock paint* metódusa eltárolja az aktuális rendszeridőt, formázza, majd megjeleníti:

```
public void paint(Graphics g) {
    //get the time and convert it to a date
    Calendar cal = Calendar.getInstance();
    Date date = cal.getTime();
    //format it and display it
    DateFormat dateFormatter = DateFormat.getTimeInstance();
    g.drawString(dateFormatter.format(date), 5, 10);
}
```

19.3.3 Programszál nem futtatható állapotba állítása

Egy szál akkor válik nem futtathatóvá, ha a lenti események közül bármelyik is bekövetkezik:

- Meghívásra kerül a *sleep* metódusa.
- A szál meghívja a *wait* metódust, hogy az várjon egy bizonyos feltétel teljesülésére.
- A szál blokkolt egy I/O művelet miatt.

A *clockThread* a *Clock* szálon belül akkor kerül nem futtatható állapotba, mikor a futás metódus meghívja a *sleep*-et a jelenlegi szálban:

```
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
           // A VM nem akarja, hogy tovább aludjunk,
           // szóval irány vissza dolgozni ☺
        }
    }
}
```

Az alatt a másodperc alatt, amíg a *clockThread* alvó állapotban van, a szál nem fut, még akkor sem fog, ha a processzor erőforrásai elérhetővé válnak. Miután letelik az egy másodperc, a szál újra futtathatóvá válik; ha a processzor erőforrásai elérhetővé válnak, a szál újra futni kezd.

Minden egyes nem futtatható állapotba kerüléskor, egy specifikus és megkülönböztethető *exit* metódus téríti vissza a szálat a futtatható állapotba. Egy *exit* hívás után csak a megfelelő feltételek teljesülése esetén fog ismét futni. Például: miután egy szál alvó állapotba lett helyezve, a meghatározott ezredmásodpercek lejárta után újból futtatható állapotba kerül. A következő felsorolás leírja a nem futtatható állapot kilépési feltételeit:

- Ha a szál alvó állapotba lett helyezve, a meghatározott időnek le kell telnie.
- Ha egy szál egy feltételre vár, akkor egy másik objektumnak kell értesítenie a várakozó szálat a feltétel teljesüléséről a *notify* vagy *notifyAll* metódus meghívásával.
- Ha egy szál blokkolva volt egy I/O művelet miatt, az I/O műveletnek be kell fejeződnie.

19.3.4 Programszál leállítása

Habár a *Thread* osztályban rendelkezésre áll a *stop* metódus, ezen metódus alkalmazása nem javallott, mert nem biztonságos, pl. adatvesztés léphet fel. Egy szálnak inkább saját magának kell befejezni a futását úgy, hogy a *run* metódusa természetesen áll le. Például a *while* ciklus a *run* metódusban egy véges ciklus 100-szor fut le, majd kilép:

```
public void run() {
    int i = 0;
    while (i < 100) {
        i++;
        System.out.println("i = " + i);
    }
}
```

A szál a *run* metódusban természetesen „hal meg” mikor a ciklus befejeződik és a *run* metódus kilép.

Nézzük meg, hogyan hajtja végre a *Clock* szál a saját halálát. Nézzük meg újra a *Clock run* metódusát:

```
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            //the VM doesn't want us to sleep anymore,
            //so get back to work
        }
    }
}
```

A kilépési feltétel ebben a *run* metódusban ugyanaz a kilépési feltétele, mint a *while* ciklusnak, mert a ciklus után nincs kód:

```
while (clockThread == myThread) {
```

E feltétel jelzi, hogy a ciklus kilép, ha a jelenleg futásban lévő szál nem egyezik a *clockThread*-el. Mikor lesz ez a helyzet? Akkor, amikor a felhasználó elhagyja az oldalt, az alkalmazás, melyben a szál fut, meghívja a szál *stop* metódusát. Ez a metódus ezután beállítja a *clockThread*-et *null* értékre, így utasítva a fő ciklust, hogy szakítsa meg a futás metódust:

```
public void stop() {    // applets' stop method
    clockThread = null;
}
```

19.3.5 Programszál státusz tesztelése

Az 5.0-ás verzióban került bevezetésre a *Thread.getState* metódus. Mikor ez kerül meghívásra, az alábbi *Thread.State* értékek valamelyike tér vissza:

- *NEW*
- *RUNNABLE*
- *BLOCKED*
- *WAITING*
- *TIMED_WAITING*
- *TERMINATED*

A *Thread* osztályhoz tartozó API tartalmazza az *isAlive* metódust is. Az *isAlive* metódus igaz visszatérési értéket generál, ha a szálat elindították, de még nem lett leállítva. Ha az *isAlive* visszatérési értéke hamis, akkor tudhatjuk, hogy a szál vagy új szál (*NEW*), vagy halott állapotban van (*TERMINATED*). Ha a visszatérési érték igaz, akkor viszont a szál vagy futtatható (*RUNNABLE*), vagy nem futtatható állapotban van.

Az 5.0-ás verziót megelőzően nem lehetett különbséget tenni az új szál vagy a halott szálak között. Ugyancsak nem lehetett megkülönböztetni a futtatható, vagy nem futtatható állapotban lévő száltól.

19.3.6 A processzor használatának feladása

Ahogyan el tudjuk képzelni, CPU-igényes kódok negatív hatással vannak más szálakra, amelyek azonos programban futnak. Vagyis próbáljunk jól működő szálakat írni, ame-

lyek bizonyos időközönként önkéntesen felhagynak a processzor használatával, ezzel megadva a lehetőséget minden szálnak a futásra.

Egy szál önkéntesen abbahagyhatja a CPU használatát a *yield* metódus meghívásával.

19.4. Ellenőrző kérdések

Melyik az az interfész, amelyet megvalósítva több szálú programfutás érhető el?

- *Runnable*
- *Run*
- *Threadable*
- *Thread*
- *Executable*

Mi annak a metódusnak a neve, amellyel a külön programszál futását kezdeményezni tudjuk?

- *init*
- *start*
- *run*
- *resume*
- *sleep*

Mi annak a metódusnak a neve, amellyel a programszál futását le tudjuk állítani?

(Minden helyes választ jelöljön meg!)

- *sleep*
- *stop*
- *yield*
- *wait*
- *notify*
- *notifyAll*
- *synchronized*

Mit tapasztalunk, ha fordítani és futtatni próbáljuk a következő programot?

```
public class Background extends Thread{
    public static void main(String argv[]){
        Background b = new Background();
        b.run();
    }
    public void start(){
        for (int i = 0; i <10; i++){
            System.out.println("Value of i = " + i);
        }
    }
}
```

- Fordítási hiba, mert a *run* metódus nincs definiálva a *Background* osztályban
- Futási hiba, mert a *run* metódus nincs definiálva a *Background* osztályban
- A program lefut, és 0-tól 9-ig ír ki számokat
- A program lefut, de nincs kimenete