MULTITHREADING

Процедура извршавања кôда у посебној нити:

1. Сместити кôд у `run()` метод класе која имплементира интерфејс `Runnable`. Интерфејс има само тај метод:

   ```
   public interface Runnable{
      void run();
   }
   ```

   Класа се просто имплементира:

   ```
   class MojaKlasa implements Runnable{
      public void run(){
            // kod
      }
   }
   ```

2. Креирати објекат класе:

   ```
   Runnable r = new MojaKlasa();
   ```

3. Креирати `Thread` објекат од `Runnable`:

   ```
   Thread t = new Thread(r);
   ```

4. Покренути (стартовати) нит:

   ```
   t.start();
   ```

Могуће је дефинисати нити и као потклasу класе `Thread`, затим конструисати објекат те поткласе и позвати његов метод `start()`. Међутим, овај начин се више не препоручује.

УПОЗОРЕЊЕ: Не позивати метод `run()` класе `Thread` или објекта `Runnable`. Директним позивањем метода `run()` заправо се његово тело извршава у текућој нити - не започиње се нова нит. Звати метод `start()` класе `Thread`. Тиме се креира нова нит у којој се извршава метод `run()`.

`java.lang.Thread`

`Thread(Runnable target)`
креира нову нит која позива метод `run() target`-а

`void start()`
започиње ову нит, позива метод `run()`.
Метод моментално враћа контролу текућој нити. Нова нит се извршава конкурентно.

`void run()`
позива метод `run()` придруженог `Runnable` објекта.

`java.lang.Runnable`

`void run()`
мора бити предефинисан (overriden), а у телу се наводе наредбе које треба извршити у нити

Прекидање нити

Нит се завршава када се заврши њен метод `run()`, извршавањем наредбе `return`, након извршавања последње наредбе у телу метода или уколико се деси избацивање изузетка који се не ухвати унутар метода.

Метод `interrupt()` се може користити како би се затражио завршетак нити.

Када се за нит позове метод `interrupt()`, постави се тзв. `interrupted status` нити. То је `boolean` флег који поседује свака нит. Свака нит требало би повремено да проверава да ли је тај њен флег постављен.

Како би се проверило да ли је `interrupted status` постављен, најпре се позива статички метод `Thread.currentThread()` за добијање текуће нити, а затим метод `isInterrupted()`:

```
while(!Thread.currentThread().isInterrupted() && more work to do){
   do more work
}
```

Међутим, уколико је нит блокирана, није могуће проверити њен `interrupted status`.
Ту је од значаја `InterruptedException`. Када се метод `interrupt()` позове за нит која је блокирала на позиву попут `sleep()` или `wait()`, блокирајући позив се завршава избацивањем изузетка типа `InterruptedException`.

Није обавезно да нит која је „прекинута“ мора да се заврши. Прекид просто служи да привуче њену пажњу. „Прекинута“ нит може да одлучи како ће реаговати на прекид. Неке нити су веома важне, па могу руковати изузетком и наставити своје извршавање. Али, прилично је уобичајено да нит интерпретира прекид као захтев за завршетком. У том случају, метод `run()` је следећег облика:

```
public void run(){
    try {
           …
       while(!Thread.currentThread().isInterrupted() && more work to do){
                do more work
       }
    } catch(InterruptedException e) {
           // thread was interrupted during sleep or wait
    } finally {
           cleanup, if required
    }

    // exiting the run method terminates the thread
}
```

Провера `isInterrupted()` није ни неопходна ни корисна уколико се позива метод `sleep()` (или неки други метод који је могуће прекинути) након сваке итерације посла. Уколико се позове метод `sleep()` када је постављен `interrupted status`, нит „не спава“, већ „очисти“ статус (!) и избаци `InterruptedException`. Према томе, уколико Ваша петља зове `sleep()`, не проверавајте `interrupted status`, већ хватајте `InterruptedException`, овако:

```
public void run(){
   try {
          …
          while(more work to do) {
                do more work
                Thread.sleep(delay);
          }
   }  catch (InterruptedException e) {
          //  thread was interrupted during sleep
   }  finally {
          cleanup, if required
   }
   // exiting the run method terminates the thread
}
```

Постоје два веома слична метода, `interrupted()` и `isInterrupted()`. Метод `interrupted()` је статички метод који проверава да ли је текућа нит прекинута. Даље, позив овог метода „чисти“ `interrupted status` нити. С друге стране, метод `isInterrupted()` је инстанцни метод који се може користити за проверу да ли је произвољна нит прекинута. Његов позив не мења `interrupted status`.

Наићи ћете на доста кода у коме је `InterruptedException` игнорисан на ниском нивоу:

```
void mySubTask(){
   …
   try {
          sleep(delay);
   } catch(InterruptedException e) {} // DON'T IGNORE!
   …
}
```

Не чините то! Уколико не можете смислити ништа паметно што би се могло урадити у `catch` клаузи, још увек имате два разумна избора:

У `catch` клаузи, позовите `Thread.currentThread().interrupt()` како бисте поставили `interrupted status`. У том случају, позивајући метод га може тестирати.

```
void mySubTask(){
   …
   try {
         sleep(delay);
   } catch (InterruptedException e) {
         Thread.currentThread().interrupt();
   }
   …
}
```

Или, што је још боље, декларишите да Ваш метод може избацити `InterruptedException` и изоставите `try`-блок. У том случају позивајући метод (или, у крајњој линији, метод `run()`) га може ухватити.

```
void mySubTask() throws InterruptedException{
   …
   sleep(delay);
   …
}
```

`java.lang.Thread`

`void interrupt()`
шаље нити захтев за прекид. `Interrupted status` нити се поставља на `true`. Ако је нит тренутно блокирана позивом метода `sleep()`, избацује се изузетак типа `InterruptedException`.

`static boolean interrupted()`
тестира да ли је текућа нит (тј. нит која извршава ову наредбу) прекинута. Метод је статички. Позив метода има бочни ефекат - ресетује `interrupted status` текуће нити на `false`.

`boolean isInterrupted()`
тестира да ли је нит прекинута. За разлику од статичког метода `interrupted()`, позив овог метода не мења `interrupted status` нити.

`static Thread currentThread()`
враћа `Thread` објекат који представља нит која се тренутно извршава.

Стања нити

Нит се може налазити у једном од 6 стања:
- new
- runnable
- blocked
- waiting
- timed waiting
- terminated

Како би се утврдило у ком стању је тренутно нит, може се позвати метод `getState()`.

New

Када се нит креира коришћењем оператора `new`, нпр. `new Thread(r)`, нит се још увек не извршава. Таква нит је у стању `new`.

Runnable

Након што се позове метод `start()`, нит прелази у стање `runnable`. Заправо, `runnable` нит може, а не мора да се извршава. На оперативном систему је да дâ нити време за извршавање.

Након што нит започне извршавање, она не мора наставити да се извршава. У ствари, пожељно је да се нити повремено паузирају, како би и друге нити добиле шансу да се извршавају. Детаљи извршавања нити зависе од оперативног система. Системи који имају тзв. „preemptive scheduling“ дају свакој `runnable` нити делић времена за извршавање. Након што се то време потроши, оперативни систем даје другој нити шансу да ради. Приликом избора следеће нити, оперативни систем узима у обзир приоритете нити (више о приоритетима касније).

Сви модерни desktop и сервер оперативни системи користе preemptive scheduling. Међутим, мали уређаји, попут мобилних телефона могу да користе cooperative scheduling. У таквом уређају, нит губи контролу једино када позове метод `yield()` или када блокира или чека.

Blocked и waiting

Нит која је у једном од ова два стања је привремено неактивна. Таква нит не извршава никакав кôд и користи минималне ресурсе. Детаљи њеног реактивирања зависе од тога како је доспела у неактивно стање.

- када нит покуша да добије (унутрашњи, intrinsic) катанац објекта (али не `Lock` из пакета `java.util.concurrent`) који тренутно држи нека друга нит, она прелази у стање `blocked`. Нит постаје одблокирана када се све остале нити одрекну катанца и „распоређивач“ нити допусти нити да га узме.

- када нит чека да друга нит обавести „распоређивача“ нити о неком услову, она је у стању `waiting`. Ово се дешава позивањем метода `wait()` класе `Object` или метода `join()` класе `Thread` или чекањем на `Lock` или `Condition` из пакета `java.util.concurrent`. У пракси, разлика између стања `blocked` и `waiting` није значајна.

- неколико метода поседује параметар `timeout`. Њихово позивање узрокује да нит пређе у стање `timed waiting`. Нит излази из овог стања након истека задатог времена или ако прими одговарајућу нотификацију. Методи са параметром `timeout` су `sleep()` класе `Thread`, `wait()` класе `Object`, `join()` класе `Thread`, `tryLock()` класе `Lock` и `await()` класе `Condition`.

На слици испод приказана су стања у којима се нит може налазити, као и могући прелази из једног стања у друго.

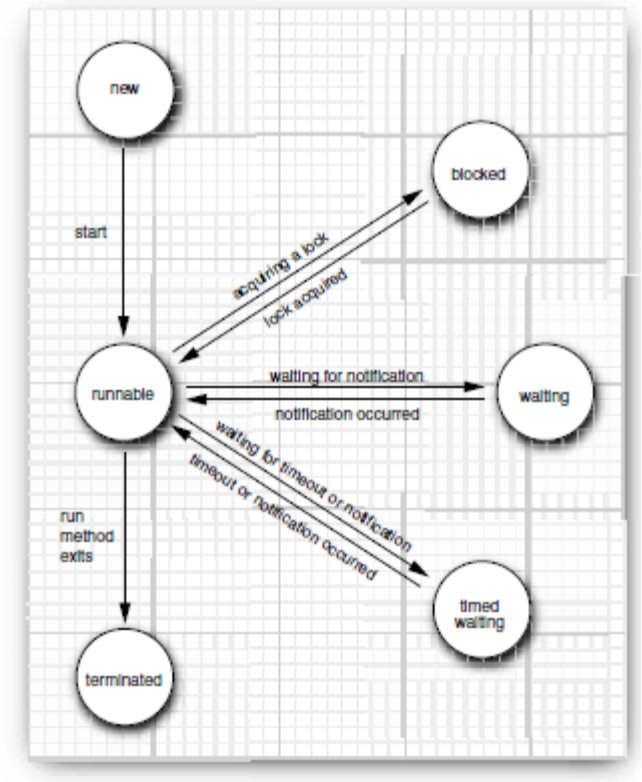


Figure 14–3 Thread states

Када је нит у стању `blocked` или `waiting` (или, наравно, када се заврши), друга нит ће бити распоређена да се извршава.

Terminated

Нит је у стању `terminated` из једног од следећа два разлога:
- умрла је природном смрћу јер је њен метод `run()` завршен нормално
- умрла је изненада јер је неухваћени изузетак завршио метод `run()`.

Посебно, могуће је убити нит позивом метода `stop()`. Међутим, овај метод је застарео (deprecated) и никада га не треба позивати из сопственог кôда.

`java.lang.Thread`

`void join()`
чека да се одређена нит заврши

`void join(long millis)`
чека да одређена нит умре или да прође задати број милисекунди

`Thread.State getState()`
враћа стање текуће нити: једно од NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED

`void stop()`
зауставља нит. Застарео!

`void suspend()`
суспендује извршавање текуће нити. Застарео!

`void resume()`
наставља текућу нит. Валидан је само након што је позван метод `suspend()`. Застарео!

Својства нити

Приоритети нити

Свака нит има свој приоритет. Подразумевано, нит наслеђује приоритет нити која ју је конструисала. Могуће је повећати или смањити приоритет произвољне нити позивом метода `setPriority()`. Могуће вредности су између MIN_PRIORITY (дефинисано као 1 у класи `Thread`) и MAX_PRIORITY (дефинисано као 10). NORM_PRIORITY је дефинисано као 5.

Кад год распоређивач има шансу да одабере нову нит, он бира нити са већим приоритетом. Међутим, приоритети нити су високо зависни од система. Када се виртуална машина ослања на имплементацију нити на host платформи, приоритети Јава нити се мапирају на нивое приоритета host платформе (којих може бити мање или више него у Јави).

Нпр. Windows има 7 нивоа приоритета. Неки од Јава приоритета мапираће се у исти ниво оперативног система. Под Linux-ом се приоритети нити у потпуности игноришу - све нити имају исти приоритет.

Почетници понекад претерују са употребом приоритета нити. Премало је разлога да се приоритети икада користе. Свакако, никада не би требало структуирати своје програме тако да њихово коректно функционисање зависи од приоритета.

ОПРЕЗ: Уколико користите приоритете, чувајте се уобичајене почетничке грешке. Ако имате неколико нити високог приоритета које не постају неактивне, може се десити да се нити нижег приоритета никада не извршавају. Кад год распоређивач треба да одлучи да извршава нову нит, изабраће неку од оних вишег приоритета, чак и ако се тиме нити нижег приоритета потпуно „изгладњују“.

`java.lang.Thread`

`void setPriority(int newPriority)`
поставља приоритет текуће нити. Приоритет мора бити између Thread.MIN_PRIORITY и Thread.MAX_PRIORITY. Користити Thread.NORM_PRIORITY за нормалан приоритет.

`static int MIN_PRIORITY`
минималан приоритет који нит може имати. Вредност је 1.

`static int NORM_PRIORITY`
подразумевани приоритет нити. Вредност 5.

`static int MAX_PRIORITY`
максималан приоритет који нит може имати. Вредност 10.

`static void yield()`
Уколико постоје друге `runnable` нити приоритета бар оноликог колики је приоритет текуће нити, оне ће бити распоређене следеће. Метод је статички.

Демонске нити

Нит се може претворити у демонску позивом
```
t.setDaemon(true);
```

Нема ничег демонског у таквој нити. Просто, нит је демонска онда када нема другу улогу у свом животу осим да служи другима. Примери су нити које шаљу правилне „временске ознаке“ другим нитима или нити које чисте застареле уносе у кешу. Када само демонска нит преостане, ВМ се гаси. Извршавање програма нема сврхе када су све преостале нити демонске.

Почетници који не желе да размишљају о завршним акцијама понекад погрешно користе демонске нити, што може бити опасно. Демонска нит никада не треба да приступа трајном (persistent) ресурсу попут фајла или базе података, пошто се може завршити у било ком тренутку, чак и у сред операције.

`java.lang.Thread`

`void setDaemon(boolean isDaemon)`
означава текућу нит као демонску или корисничку. Овај метод мора се позвати пре него што се нит стартује

Handler-и за неухваћене изузетке

Метод `run()` нити не може избацивати „checked“ изузетке, али може се завршити због „unchecked“ изузетка. У том случају, нит умире.

Међутим, нема `catch`-клаузе ка којој изузетак може пропагирати. Пре него што нит умре, изузетак се прослеђује handler-у за неухваћене изузетке.

Handler мора припадати класи која имплементира интерфејс `Thread.UncaughtExceptionHandler`, који има само један метод:
```
void uncaughtException(Thread t, Throwable e)
```

Почев од Java SE 5.0, могуће је инсталирати handler у било коју нит коришћењем метода `setUncaughtExceptionHandler()`. Такође, могуће је инсталирати подразумевани handler за све нити коришћењем статичког метода `setDefaultUncaughtExceptionHandler()` класе `Thread`. Уколико се не инсталира подразумевани handler, он буде `null`. Међутим, уколико се не инсталира handler за појединачну нит, то буде `ThreadGroup` објекат нити.

Група нити је колекција нити којом је могуће манипулисати одједном. Подразумевано, све нити које креирамо припадају истој групи нити, али могуће је извршити другачија груписања. Почев од Јаве 5.0 не треба користити групе нити у сопственим програмима.

Класа `ThreadGroup` имплементира интерфејс `Thread.UncaughtExceptionHandler`. Њен метод `uncaughtException()` врши следећу акцију:

1. ако група нити има родитеља, позива се метод `uncaughtException()` родитељске групе.
2. иначе, ако метод `Thread.getDefaultExceptionHandler()` врати не-null handler, он се позива
3. иначе, ако је `Throwable` инстанца класе `ThreadDeath`, ништа се не дешава
4. иначе, име нити и запис са стека извршавања за `Throwable` се штамају на `System.err`.

```
java.lang.Thread

static void
setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)
static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()
```
поставља или враћа подразумевани handler за неухваћене изузетке

```
void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)
Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()
```
поставља или враћа handler за неухваћене изузетке. Ако није инсталиран handler, објекат групе нити је handler.

```
java.lang.Thread.UncaughtExceptionHandler

void uncaughtException(Thread t, Throwable e)
```
дефинише се да логује пригодан извештај када се нит заврши неухваћеним изузетком.
Параметри: `t` - нит која је завршена због неухваћеног изузетка, `e` - неухваћени изузетак

```
java.lang.ThreadGroup

void uncaughtException(Thread t, Throwable e)
```
позива овај метод родитељске групе нити ако постоји родитељ, или позива подразумевани handler класе `Thread`, ако постоји подразумевани handler, а иначе штампа запис са стека извршавања на стандардни излаз за грешку. (Међутим, ако је `e ThreadDeath` објекат, не штампа. `ThreadDeath` објекти се генеришу застарелим методом `stop()`.)

Синхронизација

У већини практичних вишенитних апликација, две или више нити приступају истим подацима. Шта се дешава ако две нити имају приступ истом објекту и свака од њих позове метод који мења стање објекта? Као што можете претпоставити, у зависности од редоследа којим је приступано подацима, објекат може бити „покварен“. Таква ситуација се често назива „race condition“.

Race condition пример

Како би се избегло „кварење“ података које дели већи број нити, неопходно је научити синхронизацију приступа. У овој секцији ћемо видети шта се дешава ако се синхронизација не користи (`primer1_bezSinhronizacije`).

У наредном програму симулира се банка са одређеним бројем рачуна. Трансакције које пребацују новац са рачуна на рачун генеришу се случајно. Сваком рачуну одговара једна нит. Свака трансакција пребацује случајну количину новца са текућег рачуна на други, случајно изабрани рачун.

Кôд је праволинијски. Имамо класу `Bank` са методом `transfer()`. Метод пребацује извесну количину новца са једног рачуна на други. Још увек не бринемо о одласку „у минус“. Следи кôд за метод `transfer()` класе `Bank`.

```
public void transfer(int from, int to, double amount){
     // UPOZORENJE: nije bezbedno kada se poziva od strane veceg broja niti
     System.out.println(Thread.currentThread());
     accounts[from] -= amount;
     System.out.printf(" %10.2f from %d to %d", amount, from, to);
     accounts[to] += amount;
     System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
}
```

Следи кôд за класу `TransferRunnable`. Метод `run()` пребацује новац са фиксираног банковног рачуна. У свакој итерацији, метод `run()` изабира случајно рачун и своту, позива метод `transfer()` над објектом који представља банку, а затим спава.

```
class TransferRunnable implements Runnable{
   …
   public void run(){
         try{
               int toAccount = (int)(bank.size() * Math.random());
               double amount = maxAmount * Math.random();
               bank.transfer(fromAccount, toAccount, amount);
               Thread.sleep((int)(DELAY * Math.random());
         }catch(InterruptedException){}
   }
}
```

Када се симулација покрене, не знамо колико новца има на појединачним рачунима у произвољном тренутку. Међутим, знамо да укупна количина новца на свим рачунима треба да остане непромењена јер је све што радимо премештање новца са једног рачуна на други.

На крају сваке трансакције, метод `transfer()` поново израчунава укупну суму и штампа је. Програм се никада не завршава. Притиснути Ctrl + C за завршетак.

Када се погледа типичан излаз из програма, види се да је нешто веома погрешно. Неколико почетних трансакција износ у банци остаје $100,000, што је коректан износ на 100 рачуна, при чему је на сваком иницијално по $1,000. Међутим, након неког времена, укупан износ почиње да

се мења. Може се десити да то буде након краћег или дужег времена. Овакав исход не улива поверење и вероватно не бисте уложили свој тешко стечени новац у овакву банку.

Race condition објашњење

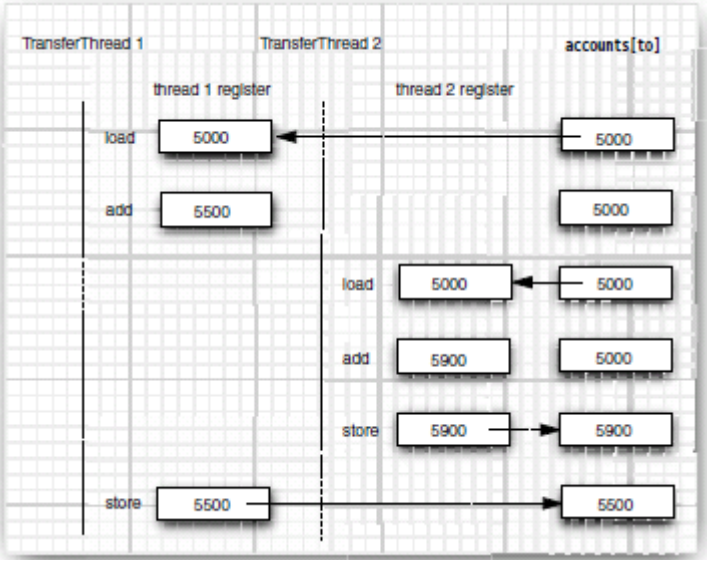
У претходном примеру неколико нити ажурира стања на банковним рачунима. Након неког времена јављају се грешке и извесна количина новца се или губи или спонтано настаје. Овај проблем настаје када две нити симултано покушавају да ажурирају рачун. Претпоставимо да две нити симултано изврше наредбу:

```
accounts[to] += amount;
```

Проблем је што то није атомична операција. Може се процесирати овако (као неколико инструкција, а нит која их извршава може бити прекинута код било које од њих):

1. Учита се `accounts[to]` у регистар `/* load */`
2. Дода се `amount` `/* add */`
3. Врати се резултат у `accounts[to]` `/* store */`

Претпоставимо сада да прва нит изврши кораке 1 и 2, а затим изгуби процесор. Претпоставимо да се друга нит пробуди и ажурира исти елемент низа `accounts`. Потом се прва нит пробуди и доврши свој корак 3. Ова акција пребрише модификацију друге нити. Резултат је да укупна сума више није исправна.



Преплитањем исписа и наредби које ажурирају стање, шансе за грешку су повећане. Уколико се штампање изостави, ризик од грешке је прилично смањен јер свака нит ради веома мало посла пре него што поново заспи, па је мало вероватно да је распоређивач прекине усред израчунавања. Међутим, тај ризик не нестаје у потпуности. Уколико се велики број нити извршава, ипак ће долазити до грешака, чак и ако се уклоне наредбе за испис. Може протећи неколико сати или чак дана до појаве грешке. Мало је горих ствари у животу програмера од грешке која се манифестује једном у неколико дана.

Катанци (Lock Objects)

Почев од Java SE 5.0, постоје два механизма за заштиту блока кôда од конкурентног приступа. Java језик обезбеђује кључну реч `synchronized`, док Java SE 5.0 уводи класу `ReentrantLock`.

Кључна реч `synchronized` аутоматски обезбеђује катанац, као и придружени „услов“, што је чини моћном и погодном за већину случајева у којима је неопходно експлицитно закључавање. Ипак, разумевање кључне речи `synchronized` је једноставније када се прво виде катанци и услови. Пакет `java.util.concurrent` обезбеђује посебне класе за ове фундаменталне механизме.

У основи, заштита блока кôда помоћу `ReentrantLock` изгледа овако:

```
myLock.lock();  // a ReentrantLock object
try{
      critical section
}finally{
      myLock.unlock();
            // make sure the lock is unlocked even if an exception is thrown
}
```

Оваква конструкција гарантује да у сваком тренутку само једна нит може ући у критичну секцију. Оног тренутка када једна нит закључа катанац, ниједна друга нит не може проћи `lock()` наредбу. Када друге нити позову `lock()`, оне се деактивирају све док прва нит не откључа катанац.

ОПРЕЗ: Од кључног је значаја да се операција `unlock()` извршава унутар клаузе `finally`. Уколико кôд унутар критичне секције избаци изузетак, катанац мора бити откључан. Иначе, остале нити ће заувек остати блокиране.

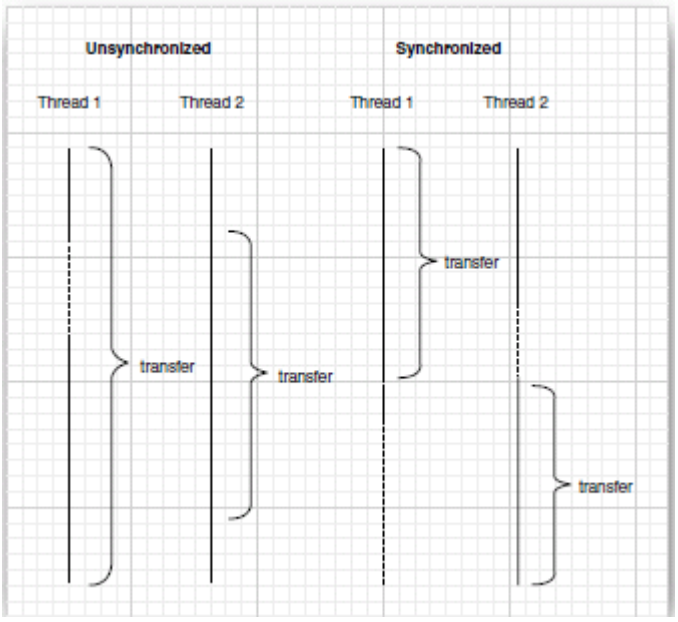
(primer2_katanciUslovi)
Пример коришћења катанца ради заштите метода `transfer()` класе `Bank`.

```
   public class Bank{

      private Lock bankLock = new ReentrantLock();
                        // ReentrantLock implements the Lock interface
      …

      public void transfer(int from, int to, int amount){
            bankLock.lock();
            try{
                  System.out.println(Thread.currentThread());
                  accounts[from] -= amount;
                  System.out.printf(" %10.2f from %d to %d",
                                          amount, from, to);
                  accounts[to] += amount;
                  System.out.printf(" Total Balance: %10.2f%n",
                                          getTotalBalance());
            }finally{
                  bankLock.unlock();
            }
      }
   }
```

Претпоставимо да једна нит позове метод `transfer()` и изгуби процесор пре него што заврши. Претпоставимо да друга нит такође позове метод `transfer()`. Друга нит не може добити катанац и блокира у позиву метода `lock()`. Она се деактивира и мора да чека да прва нит заврши извршавање метода `transfer()`. Када прва нит откључа катанац, друга може наставити своје извршавање.



Када се у програм унесу ове измене, он се може извршавати заувек, а стање у банци никада неће постати погрешно.

Приметимо да сваки `Bank` објекат поседује сопствени `ReentrantLock` објекат. Ако две нити покушају да приступе истом `Bank` објекту, катанац служи за серијализацију приступа. Међутим, ако две нити приступају различитим `Bank` објектима, онда свака нит добија различит катанац и ниједна не блокира. Тако и треба да буде, јер нити се не могу мешати са другима када оне манипулишу различитим инстанцама класе `Bank`.

Катанац се зове „reentrant“, јер нит може у више наврата да добија катанац који већ поседује. Катанац прати тзв. „hold count“, односно угњеждене позиве `lock()` метода. Нит мора да позове `unlock()` за сваки позив `lock()` како би ослободила катанац. Захваљујући овоме, кôд заштићен катанцем може позвати други метод који користи исте катанце.

Нпр. метод `transfer()` позива метод `getTotalBalance()`, који такође закључава `bankLock` објекат, који онда има hold count 2. Када се метод `getTotalBalance()` заврши, hold count је поново 1. Када се и метод `transfer()` заврши, hold count је 0 и нит ослобађа катанац.

Генерално, желећемо да заштитимо блокове кôда који ажурирају дељени објекат или му приступају. Тада смо уверени да се ове операције извршавају у потпуности пре него што друга нит може да користи исти објекат.

ОПРЕЗ: Потребно је бити пажљив да се кôд у критичној секцији не заобиђе избацивањем изузетка. Уколико се избаци изузетак пре краја критичне секције, `finally` клауза ослобађа катанац, али објекат може бити у оштећеном стању.

`java.util.concurrent.locks.Lock`

`void lock()`
дохвата текући катанац, блокира ако нека друга нит већ поседује катанац

`void unlock()`
ослобађа текући катанац

`java.util.concurrent.locks.ReentrantLock`

`ReentrantLock()`
конструише катанац који се може користити за заштиту критичне секције

`ReentrantLock(boolean fair)`
конструише катанац са задатом политиком праведности. Праведан катанац фаворизује нит која је чекала дуже. Међутим, гарантовање праведности може значајно утицати на перформансе. Према томе, подразумевано катанци нису праведни.

ОПРЕЗ: Звучи лепше бити праведан, али праведни катанци су доста спорији од регуларних. Једино у случају да заиста знате шта радите и имате специфичне разлоге због којих је праведност кључна за Ваш програм, једино тада треба користити праведне катанце. Чак и ако их користите, немате гаранцију да је распоређивач нити праведан. Уколико он одлучи да занемари нит која је дуже чекала на катанац, тада та нит не добија шансу да буде праведно третирана од стране катанца.

Услови (Condition Objects)

Често нит уђе у критичну секцију само да би открила да не може да настави даље док неки услов не буде испуњен. За руковање нитима које су добиле катанац, али не могу радити нешто корисно користи се „condition“ објекат. Из историјских разлога, „condition“ објекти се често зову и „condition“ променљиве.

Модификујмо пример са банком. Не желимо да вршимо пренос новца са рачуна који нема довољно средстава. Приметимо да не можемо користити кôд попут

```
if(bank.getBalance(from) >= amount)
   bank.transfer(from, to, amount);
```

Потпуно је могуће да текућа нит буде деактивирана између успешног теста и позива метода `transfer()`. До тренутка када се нит поново активира, стање на рачуну може постати ниже од потребног. Неопходно је обезбедити да ниједна друга нит не може модификовати стање између теста и акције трансфера. То се чини заштитом и теста и акције трансфера катанцем.

```
public void transfer(int from, int to, int amount){
   bankLock.lock();
   try{
         while(accounts[from] < amount){
               // wait
               …
         }
         // transfer funds
         …
   }finally{
         bankLock.unlock();
   }
}
```

Шта радимо када нема довољно новца на рачуну? Чекамо док га нека друга нит не дода. Али ова нит је управо добила ексклузивни приступ `bankLock` катанцу, па ниједна друга нит нема шансу да изврши улагање на рачун. Ту у игру улазе „condition“ објекти.

Сваки катанац може имати један или више придружених condition објеката (услова). Condition објекат се добија позивом метода `newCondition()`. Пожељно је да име објекта асоцира на услов који он представља. Нпр. овде condition објекат представља услов „довољно средстава“ (`sufficientFunds`).

```
class Bank{
   private Condition sufficientFunds;
   …

   public Bank(){
         …
         sufficientFunds = bankLock.newCondition();
   }
}
```

Уколико метод открије да није доступна довољна количина новца, он позива

```
sufficientFunds.await();
```

Текућа нит се онда деактивира и ослобађа катанац. Тиме се допушта да га узме нека друга нит која ће, евентуално, повећати стање на рачуну.

Постоји кључна разлика између нити која чека да добије катанац и нити која је позвала `await()`. Након што нит позове метод `await()`, она постаје део тзв. wait set-а за тај услов. Нит се не реактивира када катанац постане доступан, већ остаје деактивирана све док друга нит не позове метод `signalAll()` за исти услов.

Када друга нит изврши трансфер новца, треба да позове

```
sufficientFunds.signalAll();
```

Овај позив реактивира све нити које чекају на тај услов. Нити се уклоне из wait set-а, оне су поново у стању `runnable` и распоређивач их може активирати. У том тренутку оне ће покушати да поново добију катанац. Чим катанац постане доступан, једна од њих ће га добити и наставити тамо где је стала, а то је завршетак позива `await()`.

У том тренутку, нит поново треба да тестира услов. Нема гаранције да је он сада испуњен - метод `signalAll()` просто даје сигнал нитима које чекају да је услов МОЖДА сада испуњен и да има смисла поново извршити проверу.

Генерално, позив метода `await()` треба да буде у петљи облика:

```
while(!(ok to proceed))
   condition.await();
```

Од кључног је значаја да нека друга нит у неком тренутку позове метод `signalAll()`. Када нит позове `await()`, нема начина да саму себе реактивира. Она се узда у друге нити. Уколико ниједна од њих не реактивира нит која чека, она се никада више неће извршавати. То може водити неугодним deadlock ситуацијама. Уколико су све остале нити блокиране и последња активна позове `await()` не одблокирајући ниједну од осталих нити, она такође блокира. Ниједна нит не остаје да одблокира остале и програм стаје (не завршава се извршавање, али ништа се ни не дешава).

Када је потребно звати `signalAll()`? Увек када се стање објекта промени на начин који може утицати на нити које чекају. Нпр. кад год се стање на рачуну промени, нитима које чекају треба дати нову шансу да провере стање. У нашем примеру, позивамо `signalAll()` када завршимо пренос средстава.

```
public void transfer(int from, int to, int amount){
   bankLock.lock();
   try{
         while(accounts[from] < amount)
               sufficientFunds.await();
         // transfer funds
         …
         sufficientFunds.signalAll();
   }finally{
         bankLock.unlock();
   }
}
```

Позив метода `signalAll()` не активира моментално нит која чека. Он једино одблокира нити које чекају како би се могле надметати за катанац након што га текућа нит ослободи.

Други метод, `signal()`, одблокира само једну, случајно одабрану, нит из wait set-а. То је ефикасније него одблокиравање свих нити, али постоји опасност. Ако случајно изабрана нит открије да и даље не може да настави, она поново постаје блокирана. Ако ниједна друга нит не позове `signal()`, настаје deadlock.

ОПРЕЗ: Нит може звати методе `await()`, `signalAll()` и `signal()` за услов само када поседује катанац за тај услов.

Цена која мора да се плати када се користи механизам синхронизације за заштиту приступа дељеним подацима јесте успорење.

У пракси, исправно коришћење услова може представљати изазов. Пре него што почнете имплементацију сопствених condition објеката, потребно је размотрити употребу неког од конструката описаних касније у делу „Synchronizers“.

`java.util.concurrent.locks.Lock`

`Condition newCondition()`
враћа condition објекат придружен текућем катанцу

`java.util.concurrent.locks.Condition`

`void await()`
ставља текућу нит у wait set за текући услов

`void signalAll()`
одблокирава све нити из wait set-а за текући услов

`void signal()`
одблокирава једну случајно изабрану нит из wait set-а за текући услов

Кључна реч `synchronized`

Пре него што наставимо даље, поновимо најважније о катанцима и условима:
- катанац штити секције кôда, допуштајући да кôд у једном тренутку извршава само једна нит.
- катанац управља нитима које покушавају да уђу у заштићени сегмент кôда
- катанцу може бити придружен један или више condition објеката
- сваки condition објекат управља нитима које су ушле у заштићени сегмент кôда, али не могу да наставе даље.

Интерфејси `Lock` и `Condition` додати су у Java SE 5.0 како би омогућили програмерима висок ниво контроле над закључавањем. Међутим, у већини ситуација толика контрола није неопходна и може се користити механизам који је уграђен у Јаву. Сваки објекат у Јави има унутрашњи/интерни катанац (intrinsic lock). Уколико је метод декларисан са кључном речју `synchronized`, катанац објекта штити читав метод. То значи да нит да би позвала метод мора добити унутрашњи катанац објекта. Другим речима,

```
public synchronized void method(){
   method body
}
```

је еквивалентно са

```
public void method()
{
   this.intrinsicLock.lock();
   try{
         method body
   }finally{
         this.intrinsicLock.unlock();
   }
}
```

Нпр. уместо коришћења експлицитног катанца, можемо просто декларисати метод `transfer()` класе `Bank` као `synchronized`.

Унутрашњи катанац објекта има само један придружени услов. Метод `wait()` додаје нит у wait set, а методи `notifyAll()` и `notify()` одблокиравају нити које чекају. Другим речима, позиви `wait()` и `notifyAll()` еквивалентни су са:

```
intrinsicCondition.await();
intrinsicCondition.signalAll();
```

Методи `wait()`, `notifyAll()` и `notify()` су `final` методи класе `Object`. Методи из `Condition` зову се `await()`, `signalAll()` и `signal()`.

Нпр. класа `Bank` може се имплементирати са (`primer3_synchronized`):

```
class Bank{
     private double[] accounts;

     public synchronized void transfer(int from, int to, double amount)
                                                  throws InterruptedException{
           while(accounts[from] < amount)
                 wait();
                       // wait on intrinsic object lock's single condition
           accounts[from] -= amount;
           accounts[to] += amount;
           notifyAll();  // notify all threads waiting on the condition
     }

     public synchronized double getTotalBalance(){…}
}
```

Као што се може видети, коришћење кључне речи `synchronized` води много концизнијем кôду. Наравно, да би се тај кôд разумео, неопходно је знати да сваки објекат поседује унутрашњи катанац, коме је придружен унутрашњи услов. Катанац управља нитима које покушавају да уђу у `synchronized` метод. Услов управља нитима које су позвале метод `wait()`.

САВЕТ: Синхронизовани методи су релативно праволинијски. Међутим, почетници често „петљају“ са условима. Пре него што користите `wait()/notifyAll()` требало би размотрити употребу неког од конструката описаних касније у делу „Synchronizers“.

Допуштено је и статички метод класе декларисати као `synchronized`. Уколико се такав метод позове, он добија унутрашњи катанац придруженог `Class` објекта. Нпр. да класа `Bank` има статички `synchronized` метод, тада би катанац објекта `Bank.class` био закључан приликом његовог позива. Последица би била да ниједна друга нит не би могла да позове нити тај нити било који други `synchronized` статички метод исте класе.

Унутрашњи катанаци и услови имају извесна ограничења. Између осталог:
- није могуће прекинути нит која покушава да добије катанац
- није могуће задати `timeout` период за покушај добијања катанца
- поседовање само једног услова по катанцу може бити неефикасно.

Да ли у свом кôду користити `Lock` и `Condition` објекте или `synchronized` методе?
Следи препорука:

- најбоље је не користити ни једно ни друго. ☺ У многим ситуацијама може се користити неки од механизама из пакета `java.util.concurrent` који ради читаво закључавање за нас. Нпр. касније ће бити описано како се користи блокирајући ред за синхронизацију нити које раде на заједничком задатку.
- уколико кључна реч `synchronized` „ради“ у Вашој ситуацији, користите је, по сваку цену. Тако пишете мање кôда и имате мање простора да направите грешку.
- користите `Lock`/`Condition` ако имате специфичну потребу за додатном моћи коју Вам ови конструкти пружају.

`java.lang.Object`

Сви доле описани методи могу бити позивани искључиво из `synchronized` метода или блока. Избацују `IllegalMonitorStateException` уколико текућа нит не поседује катанац текућег објекта.

`void notifyAll()`
одблокирава нити које су позвале `wait()` за текући објекат.

`void notify()`
одблокирава једну, случајно изабрану нит од оних које су позвале `wait()` за текући објекат.

`void wait()`
узрокује да нит чека да буде обавештена (notified).

```
void wait(long millis)
void wait(long millis, int nanos)
```
узрокују да нит чека док не буде обавештена или док не истекне задата количина времена.

Параметри: `millis` број милисекунди
`nanos` број наносекунди < 1,000,000

`synchronized` блокови

Као што је већ речено, сваки објекат поседује катанац. Нит може добити катанац позивом `synchronized` метода. Постоји и други механизам за добијање катанца, а то је улазак у `synchronized` блок. Када нит уђе у блок облика:

```
synchronized (obj){
   critical section
}
```

она добија катанац за `obj`.

Понекад можете наићи на ad hoc блокове, попут:

```
public class Bank{
   private double[] accounts;
   private Object lock = new Object();

   …
```

```
            public void transfer(int from, int to, double amount){
                  synchronized(lock) {   // an ad-hoc lock
                        accounts[from] -= amount;
                        accounts[to] += amount;
                  }
                  System.out.println(…);
            }
      }
```

Овде је објекат `lock` креиран искључиво како би се користио катанац који поседује сваки Јава објекат.

Понекад, програмери користе катанац објекта како би имплементирали додатне атомичне операције. Таква пракса позната је као *client-side locking*. Размотримо, на пример, класу `Vector`, листу чији су методи `synchronized`. Претпоставимо да смо стања наше банке сачували у `Vector<Double>`. Следи наивна имплементација метода `transfer()`:

```
public void transfer(Vector<Double> accounts, int from, int to, int amount) {
                                                                  // ERROR
      accounts.set(from, accounts.get(from) - amount);
      accounts.set(to, accounts.get(to) + amount);
      System.out.println(…);
}
```

Методи `get()` и `set()` класе `Vector` су `synchronized`, али то није од помоћи. Сасвим је могуће да нит изгуби контролу унутар метода `transfer()` након што се заврши први позив метода `get()`. Друга нит може тада сачувати другачију вредност на истој позицији. Међутим, можемо употребити катанац:

```
public void transfer(Vector<Double> accounts, int from, int to, int amount) {
      synchronized(accounts){
            accounts.set(from, accounts.get(from) - amount);
            accounts.set(to, accounts.get(to) + amount);
            System.out.println(…);
      }
}
```

Овакав приступ функционише, али у потпуности зависи од чињенице да класа `Vector` користи унутрашњи катанац за све своје мутатор методе. Заправо, да ли је то уопште чињеница? Документација класе `Vector` не обећава тако нешто. Неопходно је пажљиво проучити изворни кôд и надати се да будуће верзије неће увести несинхронизоване мутаторе. Као што можете видети, client-side locking је веома „пипав“ ☺ и генерално се не препоручује.

Концепт монитора (страна 755)

Катанци и услови су моћна средства за синхронизацију нити, али нису веома објектно оријентисани.

... (за практичну употребу ова прича није од значаја)

Volatile поља

Понекад делује да је превише платити цену синхронизације само да би се прочитала или уписала чланица или две инстанце. Након свега, шта може да крене по злу? На жалост, са модерним процесорима и компајлерима, има доста простора за грешку:
- рачунари са већим бројем процесора могу привремено држати вредности из меморије у регистрима или локалним кеш-меморијама. Као последица, нити које се извршавају на различитим процесорима могу видети различите вредности исте меморијске локације!
- компајлери могу преуредити инструкције. Они то неће урадити тако да се промени значење кôда, али праве претпоставку да се вредности из меморије мењају само онда када за то постоје експлицитне инструкције у кôду. Међутим, вредност у меморији може бити промењена од стране друге нити!

Уколико користите катанце како бисте заштитили кôд коме приступа већи број нити, нећете имати ових проблема. Захтева се да компајлери поштују катанце.

Brian Goetz је смислио следећи мото за синхронизацију: „Ако пишете у променљиву из које ће затим моћи да чита друга нит, или читате из променљиве у коју је последња могла да пише друга нит, морате користити синхронизацију."

Кључна реч `volatile` нуди механизам без катанаца за синхронизовани приступ пољу инстанце. Ако декларишете поље као `volatile`, компајлер и виртуална машина узимају у обзир да друга нит може конкурентно ажурирати то поље.

Нпр. претпоставимо да објекат поседује `boolean` флег `done` који једна нит поставља, а друга очитава. Као што је већ речено, могуће је користити катанац:

```
private boolean done;
public synchronized boolean isDone() {return done;}
public synchronized void setDone(){done = true;}
```

Можда није добра идеја користити унутрашњи катанац објекта. Методи `isDone()` и `setDone()` могу блокирати ако друга нит закључа објекат. Уколико то представља проблем, може се користити одвојени катанац, само за ову променљиву, али то изискује превише мучења.

У овом случају, разумно је декларисати поље као volatile:

```
private volatile boolean done;
public boolean isDone() {return done;}
public void setDone() {done = true;}
```

ОПРЕЗ: `volatile` променљиве не обезбеђују никакву атомичност. Нпр. метод
```
public void flipDone()  {done = !done}  // not atomic
```
не гарантује инвертовање вредности поља.

У овом веома једноставном случају, постоји и трећа могућност, а то је коришћење `AtomicBoolean`. Ова класа поседује методе `get()` и `set()` који су гарантовано атомични (као да су `synchronized`). Имплементација користи ефикасне инструкције машинског нивоа које гарантују атомичност без коришћења катанаца. Постоји приличан број омотач-класа (wrapper) у

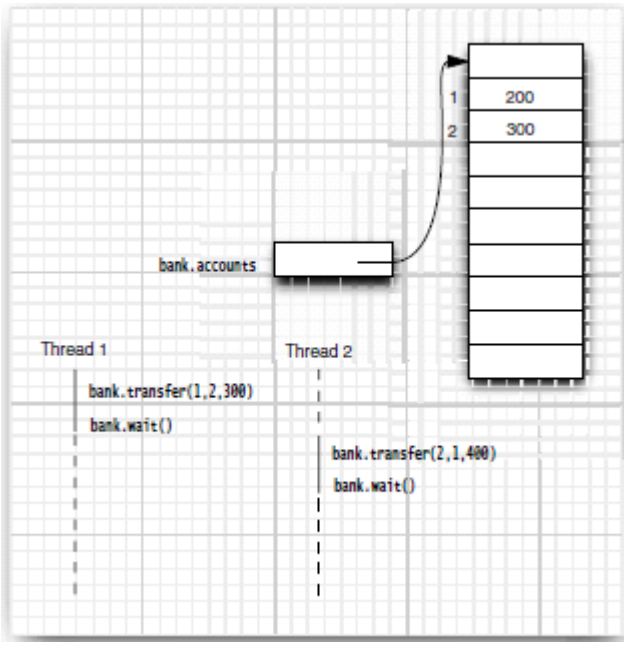
пакету `java.util.concurrent.atomic` за атомичне целе, бројеве у покретном зарезу, низове итд. Ове класе су намењене системским програмерима.

Укратко, конкурентни приступ пољу безбедан је под следећим условима:
- поље је `final` и приступа му се након што се завршио позив конструктора
- сваки приступ пољу заштићен је заједничким катанцем
- поље је `volatile`.

Deadlocks

Катанци и услови не могу решити све проблеме који могу настати у вишенитном програмирању. Размотримо следећу ситуацију:

Рачун 1: $200
Рачун 2: $300
Нит 1: Пренос $300 са Рачун 1 на Рачун 2
Нит 2: Пренос $400 са Рачун2 на Рачун 1



Јасно је да су нити 1 и 2 блокиране. Ниједна не може да настави јер на рачунима 1 и 2 нема довољно средстава.

Да ли је могуће да све нити буду блокиране јер свака чека на још новца? Да и таква ситуација назива се *deadlock*.

У нашем програму deadlock се не може десити из једноставног разлога: износ сваког трансфера је највише $1,000. Како имамо 100 рачуна и укупно $100,000 на њима, бар један од рачуна мора имати бар $1,000 у сваком тренутку. Према томе, нит која преноси новац са тог рачуна може наставити са радом.

Међутим, ако променимо метод `run()` тако што се уклони ограничење од $1,000 по трансакцији, deadlock-ови се могу брзо десити. Поставите `NACCOUNTS` на 10, конструишите сваки `TransferRunnable` са вредношћу `max` постављеном на `2 * INITIAL_BALANCE` и покрените програм. Програм ће радити извесно време, а потом стати.

САВЕТ: Када програм стоји, укуцајте CTRL + \. Добићете листу свих нити. Свака нит има стек извршавања, који говори где је тренутно блокирана.

Још један пример како би се могао изазвати настанак deadlock-а у примеру Банка: учините i-ту нит одговорну за стављање новца на i-ти рачун уместо за узимање новца са њега. У овом случају, постоји шанса да све нити „нападну" један рачун, свака покушавајући да са њега подигне више новца него што на њему има. У `primer3_synchronized` програму, метод `run()` класе `TransferRunnable`, у позиву метода `transfer()` (линија 33) размените `fromAcount` и `toAcount`. Покрените програм и посматрајте како deadlock настаје скоро моментално.

Још једна ситуација у којој deadlock једноставно настаје: промените `signalAll()` у `signal()` у програму `SynchBankTest`. Програм ће у неком тренутку стати. Поново, промените и `NACCOUNTS` у 10 како би се то брже десило. За разлику од `signalAll()` који обавештава све нити које чекају на додавање средстава, метод `signal()` одблокирава само једну нит. Ако та нит не може наставити, све нити могу бити блокиране.

Нпр. размотримо следећу ситуацију која резултира настанком deadlock-а.

Рачун 1: $1,990
На свим другим рачунима: по $990
Нит 1: пребацивање $995 са рачуна 1 на рачун 2
Све остале нити: пребацивање $995 са „свог" рачуна на неки други рачун

→ Јасно, све нити осим нити 1 су блокиране, јер на њима нема довољно новца за жељене трансакције.
Нит 1 наставља. Након трансакције имамо следећу ситуацију:

Рачун 1: $995
Рачун 2: $1985
На свим осталим рачунима: по $990

Потом нит 1 позове метод `signal()` којим се одблокирава случајно одабрана нит. Претпоставимо да је то нит 3. Та нит се буди, открива да нема довољно новца на њеном рачуну и поново позива метод `await()`. Али, нит 1 наставља да се извршава. Генерише се нова случајна трансакција, рецимо:

Нит 1: пребацивање $997 са рачуна 1 на рачун 2.

Онда нит 1 такође позове метод `await()` и све нити су блокиране. Дошло је до `deadlock`-а.

Овде је кривац позив метода `signal()`. Он одблокира само једну нит, а може се десити да не изабере нит која је есенцијална за напредак.

На жалост, не постоји ништа у програмском језику Јава чиме би се ови deadlock-ови избегли или прекинули. Ви морате дизајнирати свој програм тако да обезбедите да се deadlock не може десити.

Lock Testing and Timeouts

Када позове метод `lock()` да добије катанац који поседује друга нит, нит блокира неодређено време. Потребно је бити опрезнији приликом добијања катанца. Метод `tryLock()` покушава да добије катанац и враћа `true` ако успе. Иначе, одмах враћа `false` тако да нит може радити нешто друго.

```
if (myLock.tryLock())
   // now the thread owns the lock
   try{…}
   finally { myLock.unlock();}
else
   // do something else
```

Могуће је позивати метод `tryLock()` и са параметром `timeout`:

```
if(myLock.tryLock(100, TimeUnit.MILLISECONDS))...
```

`TimeUnit` је енумерација са вредностима `SECONDS`, `MILLISECONDS`, `MICROSECONDS` и `NANOSECONDS`.

Метод `lock()` не може бити прекинут. Уколико је нит прекинута док чека да добије катанац, она наставља да буде блокирана док катанац не постане доступан. Ако се деси deadlock, метод `lock()` се никада не завршава.

Међутим, ако се позове метод `tryLock()` са `timeout`-ом, избацује се `InterruptedException` ако се нит прекине док чека. То је очигледно корисна могућност јер допушта програму да разбије deadlock-ове.

Могуће је звати и метод `lockInterruptibly()`. Метод има исто значење као `tryLock()` са бесконачним `timeout`-ом.

Када чекамо на услов, такође можемо задати и `timeout`:

```
myCondition.await(100, TimeUnit.MILLISECONDS)
```

Метод `await()` се завршава ако друга нит активира текућу позивом `signalAll()` или `signal()`, истекне `timeout` или се нит прекине.

Метод `await()` избацује `InterruptedException` ако се нит прекине док чека. У (мало вероватном) случају да бисте радије наставили да чекате, користите метод `awaitUninterruptibly()`.

`java.util.concurrent.locks.Lock`
`boolean tryLock()` покушава да добије катанац без блокирања; враћа `true` ако успе.

`boolean tryLock(long time, TimeUnit unit)`
покушава да добије катанац, блокирајући не дуже од задатог времена; враћа `true` ако успе

`void lockInterruptibly()`
добија катанац, блокира неодређено време. Ако нит буде прекинута, избацује `InterruptedException`.

`java.util.concurrent.locks.Condition`

`boolean await(long time, TimeUnit unit)`
улази у wait set текућег услова, блокирајући док се нит не уклони из wait set-а или не истекне дато време. Враћа `false` ако се метод завршио јер је истекло време, а `true` иначе.

`void awaitUninterruptibly()`
улази у wait set текућег услова, блокирајући док се нит не уклони из wait set-а. Ако се нит прекине, нит не избацује `InterruptedException`.

Read/Write катанци

Пакет `java.util.concurrent.locks` дефинише две класе катанаца, `ReentrantLock`, о којој је већ дискутовано, и `ReentrantReadWriteLock`, која је корисна у случају да већи број нити чита из структуре података, а мањи број њих је модификује. У таквој ситуацији има смисла допустити дељени приступ читачима. Јасно, писач мора имати ексклузивни приступ.
Следе кораци неопходни за коришћење read/write катанаца:

1. конструисање `ReentrantReadWriteLock` објекта:
```
private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
```
2. издвајање read и write катанаца:
```
private Lock readLock = rwl.readLock();
private Lock writeLock = rwl.writeLock();
```
3. коришћење read катанца у свим приступним методима:
```
public double getTotalBalance(){
      readLock.lock();
      try{ … }
      finally{ readLock.unlock(); }
}
```
4. коришћење write катанца у свим мутатор-методима:
```
public void transfer( ... ){
      writeLock.lock();
      try{ … }
      finally{ writeLock.unlock(); }
}
```

`java.util.concurrent.locks.ReentrantReadWriteLock`

`Lock readLock()`
враћа read катанац који може добити већи број читача и ниједан писач

`Lock writeLock()`
враћа write катанац који не може добити ниједан други читач нити писач

Зашто су методи `stop()` и `suspend()` застарели (deprecated)?

Метод `stop()` просто зауставља нит, а метод `suspend()` је блокира док нека друга нит не позове метод `resume()`. Заједничко им је то да покушавају да контролишу понашање дате нити без кооперације нити. Оба су застарела још од Java SE 1.2. Метод `stop()` је инхерентно небезбедан, а искуство је показало да метод `suspend()` често доводи до deadlock-ова.

[Објашњење за метод `stop()`]:
Метод завршава све методе који чекају, укључујући и метод `run()`. Када се нит заустави, она моментално откључава катанце свих објеката које је закључала, што може оставити објекте у неконзистентном стању. Нпр. претпоставимо да се нит `TransferRunnable` заустави у сред пребацивања новца са једног рачуна на други, након скидања са првог рачуна, а пре улагања на други. У том случају објекат који представља банку је оштећен. Након што се катанац ослободи, штета је приметна из других нити које нису стопиране. Када желите да зауставите нит, треба да је прекинете. Прекинута нит може да се заустави онда када је то безбедно (да не остави објекат у неисправном стању, нпр. онаквим какав би се затекао у сред трансакције, што би могло да се деси када би се заустављање дешавало моментално).

Када нит жели да заустави другу нит, нема начина да зна када је безбедно да позове метод `stop()`, а када он води оштећењу објеката. Због тога је он проглашен за deprecated. Треба прекинути нит када желимо да је зауставимо.

[Објашњење за метод `suspend()`]:
За разлику од метода `stop()`, метод `suspend()` неће оштетити објекте. Међутим, ако суспендујемо нит која поседује катанац, тај катанац је недоступан све док се нит не настави (resume). Ако нит која је позвала метод `suspend()` покуша да добије тај исти катанац, настаје deadlock. Суспендована нит чека да је наставе, а нит која ју је суспендовала чека на катанац.

Ова ситуација често се дешава са графичким корисничким интерфејсима. Претпоставимо да имамо графичку симулацију наше банке. Дугме на коме пише Pause суспендује нити за трансфер, а дугме на коме пише Resume их наставља:

```
pauseButton.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e){
         for(int i=0; i<threads.length; i++)
               threads[i].suspend(); // Don't do this
   }
});
resumeButton.addActionListener(…);
         // calls resume on all transfer threads
```

Претпоставимо да метод `paintComponent()` исцртава графикон сваког рачуна, при том позивајући метод `getBalances()` како би дохватио низ стања.

И акција за дугме и реисцртавање се дешавају у истој нити, тзв. event dispatch thread (нит за обраду догађаја). Размотримо следећи сценарио:

1. једна од нити за трансфер добије катанац за објекат `bank`
2. корисник кликне на дугме Pause

3. све нити за трансфер се суспендују; једна од њих још увек поседује катанац за објекат `bank`
4. метод `paintComponent()` позива метод `getBalances()`
5. метод покушава да добије катанац за објекат `bank`

Програм се замрзава.
Нит за обраду догађаја не може да настави јер катанац поседује једна од суспендованих нити. Према томе, корисник не може да кликне на дугме Resume, па се нити никада неће наставити.

Када желите безбедно да суспендујете нит, уведите променљиву `suspendRequested` и тестирајте је на безбедном месту у свом методу `run()` - негде где Ваша нит не закључава објекте који су потребни другим нитима. Када Ваша нит установи да је променљива `suspendRequested` постављена, треба да чека док не постане поново доступна.

Следећи кôд имплементира описани дизајн:

```
private volatile boolean suspendRequested = false;
private Lock suspendLock = new ReentrantLock();
private Condition suspendCondition = suspendLock.newCondition();

public void run(){
   while(…){
         …
         if(suspendRequested){
               suspendLock.lock();
               try{
                     while(suspendRequested)
                           suspendCondition.await();
               }finally{
                     suspendLock.unlock();
               }
         }
   }
}

public void requestSuspend(){
   suspendRequested = true;
}

public void requestResume(){
   suspendRequested = false;
   suspendLock.lock();
   try{ suspendCondition.signalAll(); }
   finally {suspendLock.unlock();}
}
```

Блокирајући редови

До сада су описивани градивни блокови ниског нивоа који чине основе конкурентног програмирања у Јави. Међутим, за практично програмирање, желимо да се држимо подаље од конструката ниског нивоа кад год је то могуће. Много је једноставније и безбедније користити структуре вишег нивоа које су имплементирали експерти за конкурентно програмирање.

Многи вишенитни проблеми могу бити елегантно и безбедно формулисани коришћењем једног или већег броја редова. Нити произвођачи убацују производе у ред, а нити потрошачи их отуда узимају. Ред омогућује да једна нит безбедно преда податке другој. Нпр. размотримо наш пример са банком. Уместо да се објекту који представља банку приступа директно, нити за трансфер убацују објекте са инструкцијама за трансфере у ред. Посебна нит уклања инструкције из реда и извршава трансфере. Само та нит има приступ унутрашњости објекта који представља банку. Синхронизација није потребна. (Наравно, имплементатори threadsafe класа за редове морали су да брину о катанцима и условима, али то је њихов проблем, не Ваш.)

Блокирајући ред узрокује да нит блокира када покушате да додате елемент у тренутно пун ред или да уклоните елемент онда када је ред празан. Блокирајући редови су корисно средство за координисање рада већег броја нити. „Worker“ нити могу периодично стављати међурезултате у блокирајући ред. Друге worker нити уклањају међурезултате и даље их модификују. Ред аутоматски балансира обим посла. Ако први скуп нити ради спорије од другог, други скуп блокира чекајући резултате. Ако се први скуп нити извршава брже, ред се напуни док други скуп не „ухвати прикључак“.

Методи за блокирајући ред

Метод	Нормална акција	Акција у специјалним околностима .
`add()`	додаје елемент	избацује `IllegalStateException` ако је ред пун
`element()`	враћа ел. „главу“	избацује `NoSuchElementException` за празан ред
`offer()`	додаје ел. и враћа `true`	враћа `false` ако је ред пун
`peek()`	враћа ел. „главу“	враћа `null` ако је ред празан
`poll()`	уклања и враћа ел. „главу“	враћа `null` ако је ред празан
`put()`	додаје елемент	блокира ако је ред пун
`remove()`	уклања и враћа ел. „главу“	избацује `NoSuchElementException` за празан ред
`take()`	уклања и враћа ел. „главу“	блокира ако је ред празан

Методи за блокирајући ред спадају у три категорије, у зависности од акције коју предузимају када је ред пун или празан. Ако ред користите као средство за управљање нитима, желећете да користите методе `put()` и `take()`. Операције `add()`, `remove()` и `element()` избацују изузетак када покушате додавање у пун ред или дохватите „глава“-елемент празног реда. Наравно, у вишенитном програму, ред може постати пун или празан у било ком тренутку, па ћете уместо претходна три метода желети да користите методе `offer()`, `poll()` и `peek()`. Ови методи просто враћају индикатор неуспеха уместо да избаце изузетак уколико не могу извршити своје задатке.

Методи `poll()` и `peek()` враћају `null` да укажу на неуспех. Према томе, није допуштено убацивати `null` вредности у овакве редове.

Постоје и варијанте метода `offer()` и `poll()` са параметром `timeout`. Нпр. позив

```
boolean success = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

100 милисекунди покушава да убаци елемент на реп (крај) реда. Ако успе, враћа `true`, а иначе враћа `false` када време истекне.

Слично, позив

```
Object head = q.poll(100, TimeUnit.MILLISECONDS);
```

100 милисекунди покушава да уклони главу реда. Ако успе, враћа главу, а иначе враћа `null` када истекне време.

Метод `put()` блокира када је ред пун, а метод `take()` блокира ако је ред празан. Ово су еквиваленти метода `offer()` и `poll()` без параметра `timeout`.

У пакету `java.util.concurrent` постоји неколико варијанти блокирајућих редова. Подразумевано, `LinkedBlockingQueue` нема горње ограничење за капацитет, али максимални капацитет се може опционо задати. `LinkedBlockingDeque` је double-ended верзија. `ArrayBlockingQueue` се конструише са задатим капацитетом и опционим параметром којим се захтева праведност. Ако је задата праведност, предност се даје нитима које дуже чекају. Као и увек, праведност повлачи значајно смањење перформанси и треба је користити једино ако је неопходна због специфичности проблема.

`PriorityBlockingQueue` је ред са приоритетима, не FIFO. Елементи се уклањају редоследом који одговара њиховим приоритетима. Ред је неограниченог капацитета, а дохватање елемента блокира када је ред празан.

Коначно, `DelayQueue` садржи објекте који имплементирају интерфејс `Delayed`:

```
interface Delayed extends Comparable<Delayed>{
   long getDelay(TimeUnit unit);
}
```

Метод `getDelay()` враћа преостало кашњење (delay) објекта. Негативна вредност указује да је delay-период истекао. Елементи се могу уклонити из `DelayQueue` само када њихов delay-период истекне. Неопходно је имплементирати и метод `compareTo()`. `DelayQueue` користи тај метод за сортирање уноса.

Пример (`primer4_BlockingQueue`): коришћење блокирајућег реда за контролисање скупа нити. Програм претражује све фајлове директоријума и његових поддиректоријума, штампајући линије које садрже дату кључну реч.

Нит произвођач енумерише све фајлове у свим поддиректоријумима и смешта их у блокирајући ред. Ова операција је брза и ред би се брзо напунио свим фајловима фајл система да није ограничен.

Такође, покреће се велики број претраживачких нити. Свака претраживачка нит узима фајл из реда, отвара га, штампа све линије које садрже кључну реч и затим узима следећи фајл. За завршавање апликације користи се следећи трик. У циљу означавања краја, нит за енумерацију смешта „лажни“ (dummy) објекат у ред. Када претраживачка нит узме овај објекат, враћа га назад и завршава се.

Приметите да није неопходна експлицитна синхронизација нити. Као механизам синхронизације користи се ред (структура података).

```
java.util.concurrent.ArrayBlockingQueue<E>

ArrayBlockingQueue(int capacity)
ArrayBlockingQueue(int capacity, boolean fair)
```
конструишу блокирајући ред датог капацитета и подешавања праведности. Ред је имплементиран као циркуларни низ.

```
java.util.concurrent.LinkedBlockingQueue<E>
java.util.concurrent.LinkedBlockingDeque<E>

LinkedBlockingQueue()
LinkedBlockingDeque()
```
конструишу неограничени блокирајући ред или double-ended ред (deque), имплементиран као повезана листа

```
LinkedBlockingQueue(int capacity)
LinkedBlockingDeque(int capacity)
```
конструишу ограничени ред или double-ended ред задатог капацитета имплементиран као повезана листа

```
java.util.concurrent.DelayQueue<E extends Delayed>

DelayQueue()
```
конструише неограничени блокирајући ред `Delayed` елемената. Само елементи чији је delay-период истекао могу се уклонити из реда.

```
java.util.concurrent.Delayed

long getDelay(TimeUnit unit)
```
враћа delay текућег објекта, изражен у датим временским јединицама

```
java.util.concurrent.PriorityBlockingQueue<E>

PriorityBlockingQueue()
PriorityBlockingQueue(int initialCapacity)
PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator)
```
конструише неограничени блокирајући ред са приоритетима имплементиран као хип

Параметри:

`initialCapacity`	иницијални капацитет реда са приоритетима. Подразумевано 11
`comparator`	компаратор који се користи за поређење елемената. Ако није задат, класа елемената мора имплементирати интерфејс `Comparable`.

```
java.util.concurrent.BlockingQueue<E>

void put(E element)
```
додаје елемент, блокира по потреби

```
E take()
```
уклања и враћа главу, блокира по потреби

```
boolean offer(E element, long time, TimeUnit unit)
```
додаје дати елемент и враћа `true` ако успе, блокира по потреби док се елемент не дода или време не истекне

```
E poll(long time, TimeUnit unit)
```
уклања и враћа главу, блокира по потреби док елемент не постане доступан или не истекне време. Враћа `null` ако не успе.

```
java.util.concurrent.BlockingDeque<E>

void putFirst(E element)
void putLast(E element)
```
додаје елемент, блокира по потреби

```
E takeFirst()
E takeLast()
```
уклања и враћа главу или реп, блокира по потреби

```
boolean offerFirst(E element, long time, TimeUnit unit)
boolean offerLast(E element, long time, TimeUnit unit)
```
додаје дати елемент и враћа `true` ако успе, блокира по потреби док се елемент не дода или не истекне врме

```
E pollFirst(long time, TimeUnit unit)
E pollLast(long time, TimeUnit unit)
```
уклања и враћа главу или реп, блокира по потреби док елемент не постане доступан или време не истекне. Ако не успе, враћа `null`.

Thread-Safe колекције

Када већи број нити конкурентно модификује структуру података, попут хеш-табеле лако је могуће да се структура оштети. Нпр. једна нит може започети убацивање новог елемента. Претпоставимо да изгуби процесор док је усред рерутирања линкова bucket-а хеш-табеле. Уколико друга нит обилази исту листу, може пратити погрешне линкове и направити пустош избацујући изузетке или тако што бива заробљена у бесконачној петљи.

Дељена структура података може се заштитити катанцем, али обично је једноставније одабрати thread-safe имплементацију. Блокирајући редови разматрани у претходној секцији су thread-safe колекције. У наредним секцијама разматрамо остале thread-safe колекције из Јавине библиотеке.

Ефикасне мапе, скупови и редови

Пакет `java.util.concurrent` располаже ефикасним имплементацијама мапа, уређених скупова и редова: `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet` и `ConcurrentLinkedQueue`.

Ове колекције користе софистициране алгоритме који минимизују сударање допуштајући конкурентни приступ различитим деловима структуре података.
За разлику од већине колекција, метод `size()` за ове колекције не оперише нужно у константном времену. Одређивање тренутне величине овакве колекције обично захтева обилазак.

Колекције враћају *weakly consistent* итераторе. То значи да итератори могу, а не морају одражавати све модификације учињене након њиховог конструисања, али они неће два пута вратити исту вредност и неће избацити `ConcurrentModificationException`.

Супротно овоме, итератор колекције из пакета `java.util` избацује `ConcurrentModificationException` када је колекција промењена након конструисања итератора.

Конкурентна хеш-мапа може ефикасно подржати велики број читача и фиксирани број писача. Подразумевано, претпоставља се да постоји до 16 *симултаних* писач нити. Може их бити много више, али ако више од 16 њих истовремено пише, остале су привремено блокиране. Могуће је задати већи број у конструктору, али је мало вероватно да за тим има потребе.

Класе `ConcurrentHashMap` и `ConcurrentSkipListMap` имају корисне методе за уметање и уклањање асоцијација (парова кључ-вредност). Метод `putIfAbsent()` атомично додаје нову асоцијацију уколико је раније није било. То је корисно за кеш коме приступа већи број нити како би се осигурало да само једна нит додаје ставку у кеш.

```
cache.putIfAbsent(key, value);
```

Супротна операција је `remove()` (која би вероватно требало да се зове removeIfPresent). Позив

```
cache.remove(key, value);
```

атомично уклања кључ и вредност ако су присутни у мапи. Коначно,

```
cache.replace(key, oldValue, newValue);
```

атомично замењује стару вредност новом, где је стара вредност била придружена датом кључу.

```
java.util.concurrent.ConcurrentLinkedQueue<E>

ConcurrentLinkedQueue<E>()
```

конструише неограничени неблокирајући ред коме више нити може безбедно приступати

```
java.util.concurrent.ConcurrentSkipListSet<E>

ConcurrentSkipList<E>()
ConcurrentSkipList<E>(Comparator<? super E> comp)
```

конструишу сортирани скуп коме већи број нити може безбедно приступати. Први конструктор захтева да елементи имплементирају интерфејс `Comparable`.

```
java.util.concurrent.ConcurrentHashMap<K, V>
java.util.concurrent.ConcurrentSkipListMap<K, V>

ConcurrentHashMap<K, V>()
ConcurrentHashMap<K, V>(int initialCapacity)
ConcurrentHashMap<K, V>(int initialCapacity, float loadFactor,
                                                int concurrencyLevel)
```

конструишу хеш мапу којој већи број нити може безбедно приступати.

Параметри	`initialCapacity`	иницијални капацитет ове колекције. Подразумевано 16.
	`loadFactor`	контролише промену величине: уколико просечна попуњеност bucket-а премаши овај фактор, повећа се величина табеле. Подразумевано 0.75
	`concurrencyLevel`	процењени број конкурентних нити писача

```
ConcurrentSkipListMap<K, V>()
ConcurrentSkipListMap<K, V>(Comparator<? super K> comp)
```
конструишу сортирану мапу којој већи број нити може безбедно приступати. Први конструктор захтева да кључеви имплементирају интерфејс `Comparable`.

```
V putIfAbsent(K key, V value)
```
ако кључ `key` није присутан у мапи, придружује му дату вредност `value` и враћа `null`. Иначе враћа постојећу вредност придружену кључу.

```
boolean remove(K key, V value)
```
ако је датом кључу `key` тренутно придружена вредност `value`, уклања дати кључ и вредност и враћа `true`, а иначе враћа `false`.

```
boolean replace(K key, V oldValue, V newValue)
```
ако је кључу `key` тренутно придружена вредност `oldValue`, придружује му вредност `newValue`. Иначе, враћа `false`.

Copy on Write Arrays

`CopyOnWriteArrayList` и `CopyOnWriteArraySet` су thread-safe колекције у којима сви мутатори креирају копију низа. То је погодно када број нити које итерирају преко колекције увелико премашује број нити које је мењају. Када се конструише итератор, он садржи референцу на текући низ. Ако је низ касније измењен, итератор и даље има стари низ, али низ колекције је измењен. Као последица, стари итератор има конзистентан, али потенцијално застарео, поглед коме може приступати без икаквих трошкова синхронизације.

Старије Thread Safe колекције

Класе `Vector` и `Hashtable` су одувек обезбеђивале thread-safe имплементације динамичког низа и хеш табеле. У Java SE 1.2, ове класе су декларисане као застареле (obsolete) и замењене класама `ArrayList` и `HashMap` које нису thread-safe, већ је обезбеђен нови механизам у библиотеци колекција. Свака класа може бити учињена thread-safe помоћу *synchronization wrapper*-а:

```
List<E> synchArrayList = Collections.synchronizedList(new ArrayList<E>());
Map<K, V> synchHashMap = Collections.synchronizedMap(new HashMap<K, V>());
```

Методи резултујућих колекција су заштићени катанцем, обезбеђујући thread-safe приступ.

Треба обезбедити да ниједна нит не приступа структури података преко оригиналних несинхронизованих метода. Најједноставнији начин за то јесте не сачувати референцу на оригинални објекат. Једноставно конструишите колекцију и одмах је проследите wrapper-у, као што је урађено у горњим примерима.

Још увек је потребно користити „client-side“ locking ако желите да итерирате преко колекције док друга нит има могућност да је мења:

```
synchronized(synchHashMap){
   Iterator<K> iter = synchHashMap.keySet().iterator();
   while(iter.hasNext())
         …
}
```

Потребно је користити исти кôд и за collection-based for-петљу јер петља користи итератор. Итератор заправо не успева и избацује `ConcurrentModificationException` ако друга нит модификује колекцију док је итерирање у току. Синхронизација је неопходна како би конкурентно модификовање било поуздано детектовано.

Обично је боље користити колекције дефинисане у пакету `java.util.concurrent` уместо синхронизационих wrapper-а. Посебно, `ConcurrentHashMap` је пажљиво имплементирана тако да јој већи број нити може приступати не блокирајући једна другу, тако што приступају различитим bucket-има. Изузетак је array list која се често модификује. У том случају, синхронизовани `ArrayList` може бити ефикаснији него `CopyOnWriteArrayList`.

```
java.util.Collections

static <E> Collection<E> synchronizedCollection(Collection<E> c)
static <E> List synchronizedList(List<E> c)
static <E> Set synchronizedSet(Set<E> c)
static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)
static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)
static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)
```
конструишу погледе на колекције, чији методи су синхронизовани

Callables and Futures

`Runnable` енкапсулира задатак/посао (task) који се асинхроно извршава; могуће је мислити о њему као о асинхроном методу без параметара и без повратне вредности. `Callable` је сличан са `Runnable`, али враћа вредност. Интерфејс `Callable` је параметризован и има само један метод:

```
public interface Callable<V>{
   V call() throws Exception;
}
```

Типски параметар је тип повратне вредности. Нпр. `Callable<Integer>` представља асинхроно израчунавање које на крају враћа `Integer` објекат.

`Future` „чува“ резултат асинхроног израчунавања. Започнете израчунавање, дате некоме `Future` објекат и заборавите на њега. Власник `Future` објекта може добити резултат када он буде готов.

Интерфејс `Future` има следеће методе:

```
public interface Future{
   V get() throws …;
   V get(long timeout, TimeUnit unit) throws …;
   void cancel(boolean mayInterrupt);
   boolean isCancelled();
   boolean isDone();
}
```

Позив првог метода `get()` блокира док се израчунавање не заврши. Други метод избацује `TimeoutException` ако позив истекне пре но што се израчунавање заврши. Уколико нит која извршава израчунавање буде прекинута, оба метода избацују `InterruptedException`. Ако се израчунавање већ завршило, метод `get()` се завршава одмах (без блокирања).

Метод `isDone()` враћа `false` ако је израчунавање још увек у току, а `true` ако је завршено.

Од израчунавања се може одустати позивом метода `cancel()`. Уколико израчунавање још увек није почело, никада неће почети. Ако је, пак, израчунавање у току, прекида се ако је параметар `mayInterupt true`.

`FutureTask` wrapper је погодан механизам за претварање `Callable` и у `Future` и у `Runnable` - јер имплементира оба ова интерфејса. Нпр.

```
Callable<Integer> myComputation = …;
FutureTask<Integer> task = new FutureTask<Integer>(myComputation);
Thread t = new Thread(task);  // it's a Runnable
t.start();
…
Integer result = task.get();  // it's a Future
```

Пример (`primer5_CallableFuture`), сличан са претходним који тражи фајлове који садрже дату кључну реч. Међутим, сада ћемо просто пребројати колико је таквих фајлова. Према томе, имамо посао који се дуго извршава и враћа цео број - пример за `Callable<Integer>`.

```
class MatchCounter implements Callable<Integer>{
   public MatchCounter(File directory, String keyword){…}
   public Integer call() {…} // vraca broj odgovarajucih fajlova
}
```

Потом конструишемо `FutureTask` објекат од `MatchCounter` и користимо га за започињање нити:

```
FutureTask<Integer> task = new FutureTask<Integer>(counter);
Thread t = new Thread(task);
t.start();
```

Коначно, штампамо резултат:

```
System.out.println(task.get() + "matching files. ");
```

Наравно, позив метода `get()` блокира док резултат не постане доступан.

Унутар метода `call()` рекурзивно се користи исти механизам. За сваки поддиректоријум креира се нови `MatchCounter` и покреће нит за њега. Такође, одлажемо `FutureTask` објекте у `ArrayList<Future<Integer>>`. На крају, просумирамо резултате:

```
for(Future<Integer> result: results)
   count += result.get();
```

Сваки позив метода `get()` блокира док резултат не постане доступан. Наравно, нити се извршавају паралелно, па постоји велика могућност да ће резултати бити доступни отприлике у исто време.

`java.util.concurrent.Callable<V>`

`V call()`
извршава посао који има резултат

`java.util.concurrent.Future<V>`

`V get()`
`V get(long time, TimeUnit unit)`
враћа резултат, блокирајући док не постане доступан или док не истекне дато време. Други метод избацује `TimeoutException` ако не успе.

`boolean cancel(boolean mayInterrupt)`
покушава да откаже извршење текућег посла. Ако је посао већ започет и параметар `mayInterrupt` је `true`, он се прекида. Враћа `true` ако је отказивање посла успело.

`boolean isCanceled()`
враћа `true` ако је посао отказан пре завршетка

`boolean isDone()`
враћа `true` ако је посао завршен нормално, отказивањем или изузетком

`java.util.concurrent.FutureTask<V>`

`FutureTask(Callable<V> task)`
`FutureTask(Runnable task, V result)`
конструише објекат који је и `Future<V>` и `Runnable`

Executors

Конструисање нове нити је донекле скупо јер укључује интеракцију са оперативним системом. Уколико наш програм треба да креира велики број нити које кратко трају, боље је да уместо тога користи *thread pool*. Thread pool садржи известан број „доконих“ (idle) нити које су спремне за извршавање. Када се метод `run()` заврши, нит не умире, већ остаје да опслужи наредни захтев.

Други разлог за коришћење thread pool-а је смањивање броја конкурентних нити. Креирање огромног броја нити може увелико деградирати перформансе, па чак и срушити виртуалну машину. Ако имате алгоритам који креира много нити, требало би да користите „фиксирани“ thread pool који ограничава укупан број конкурентних нити.

Класа `Executors` има известан број статичких factory метода за конструисање thread pool-ова.

Метод	Опис
`newCachedThreadPool`	нове нити се креирају по потреби, доконе се чувају 60 секунди
`newFixedThreadPool`	скуп нити је фиксиран, доконе се чувају неодређено време
`newSingleThreadExecutor`	само једна нит која извршава послове секвенцијално (слично као Swing event dispatch thread)
`newScheduledThreadPool`	фиксирани thread pool за испланирано (scheduled) извршавање; замена за `java.util.Timer`
`newSingleThreadScheduledExecutor`	само једна нит за испланирано извршавање

Thread pools

Метод `newCachedThreadPool()` конструише thread pool који сваки посао извршава одмах, користећи постојећу докону нит ако постоји, а креирајући нову у супротном.

Метод `newFixedThreadPool()` конструише thread pool фиксиране величине. Ако има више послова него доконих нити, неуслужени послови се смештају у ред. Извршавају се када се неки други заврше.

`newSingleThreadExecutor` је дегенерисани pool величине 1: једна нит извршава послове, један за другим.

Ова три метода враћају објекат класе `ThreadPoolExecutor` која имплементира интерфејс `ExecutorService`.

Једним од следећих метода могуће је да се `ExecutorService` објекту проследи `Runnable` или `Callable`:

```
Future<?> submit(Runnable task)
Future<T> submit(Runnable task, T result)
Future<T> submit(Callable<T> task)
```

Када позовете метод `submit()` добијете објекат класе која имплементира интерфејс `Future`, који можете користити за испитивање стања у коме се налази посао.

Први метод `submit()` враћа `Future` чудног изгледа `Future<?>`. Такав објекат се може користити за позиве метода `isDone()`, `cancel()` и `isCanceled()`. Метод `get()` по завршетку просто враћа `null`.

Друга верзија метода `submit()` такође шаље `Runnable`, а метод `get()` објекта `Future` враћа дати објекат `result` по завршетку.

Трећа верзија шаље `Callable`, а враћени `Future` добија резултат израчунавања када је готов.

Када завршите са thread pool-ом, позовите `shutdown()`. Овај метод иницира завршавајућу секвенцу за thread pool. Executor који се гаси не прихвата нове послове. Када се сви послови заврше, нити из pool-а умиру. Алтернативно, може се позвати `shutdownNow()`. Pool тада отказује све послове који још нису почели и покушава да прекине нити које се извршавају.

Укратко, за коришћење thread pool-а потребно је:

1. позвати статички метод `newCachedThreadPool()` или `newFixedThreadPool()` класе `Executors`.
2. позвати метод `submit()` како би се послали `Runnable` или `Callable` објекти
3. ако желите да будете у могућности да откажете посао или ако пошаљете `Callable` објекте, ослоните се на враћене `Future` објекте
4. позовите `shutdown()` када не желите више да шаљете послове

Претходни пример је производио велики број нити које се кратко извршавају, по једну за сваки директоријум. Следећи пример (`primer6_ThreadPool`) уместо таквог приступа користи thread pool.

Из информативних разлога, овај програм штампа највећу димензију thread pool-а за време извршавања. Ова информација није доступна кроз интерфејс `ExecutorService`. Из тог разлога неопходно је кастовање pool објекта у класу `ThreadPoolExecutor`.

```
java.util.concurrent.Executors

ExecutorService newCachedThreadPool()
```
враћа кеширани thread pool који креира нити по потреби и завршава нити које су доконе 60 секунди

```
ExecutorService newFixedThreadPool(int threads)
```
враћа thread pool који користи задати број нити за извршавање послова

```
ExecutorService newSingleThreadExecutor()
```
враћа „егзекутор“ који извршава послове секвенцијално у једној нити

```
java.util.concurrent.ExecutorService

Future<T> submit(Callable<T> task)
Future<T> submit(Runnable task, T result)
Future<?> submit(Runnable task)
```
шаље дати посао на извршење

```
void shutdown()
```
завршава сервис, комплетирајући већ послате послове, али не прихватајући нове

```
java.util.concurrent.ThreadPoolExecutor

int getLargestPoolSize()
```
враћа највећу величину thread pool-а за време живота текућег егзекутора.

Scheduled Execution (планирано извршавање)

Интерфејс `ScheduledExecutorService` поседује методе за планирано или поновљено извршавање послова. То је генерализација `java.util.Timer` која омогућује thread pooling. Методи `newScheduledThreadPool()` и `newSingleThreadScheduledExecutor()` класе `Executors` враћају објекте који имплементирају интерфејс `ScheduledExecutorService`.

Могуће је испланирати да се `Runnable` или `Callable` изврши једном, након задатог иницијалног чекања. Такође, могуће је испланирати да се `Runnable` извршава периодично.

```
java.util.concurrent.Executors

ScheduledExecutorService new ScheduledThreadPool(int threads)
```
враћа thread pool који користи задати број нити за распоређивање послова

```
ScheduledExecutorService newSingleThreadScheduledExecutor()
```
враћа егзекутор који распоређује послове у једној нити

```
java.util.concurrent.ScheduledExecutorService

ScheduledFuture<V> schedule(Callable<V> task, long time, TimeUnit unit)
ScheduledFuture<?> schedule(Runnable task, long time, TimeUnit unit)
```
распоређује дати посао након што истекне дато време

```
ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay, long
                                                        period, TimeUnit unit)
```
распоређује дати посао да се извршава периодично, сваки `period` јединица, након што истекне задати период иницијалног чекања

```
ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay,
                                                  long delay, TimeUnit unit)
```
распоређује дати посао да се извршава периодично, са `delay` јединица између завршетка једног позива и почетка наредног, након што истекне задати период иницијалног чекања.

Контролисање група послова

Видели смо како се executor service користи као thread pool за повећање ефикасности извршавања послова. Понекад executor се користи из више тактичког разлога, просто како би се контролисала група повезаних послова. Нпр. можемо отказати све послове из егзекутора позивом метода `shutdown()`.

Метод `invokeAny()` шаље све објекте из колекције `Callable` објеката и враћа резултат завршеног посла. Није познато који је то посао - вероватно онај који се најбрже завршио. Овај метод има примену у проблему претраге у ком прихватамо произвољно решење. Нпр. претпоставимо да желимо да факторизујемо велики цео број - израчунавање које је неохподно за разбијање RSA шифровања. Могли бисмо да пошаљемо известан број послова, од којих сваки покушава факторизацију коришћењем бројева из различитих опсега. Чим један од ових послова добије одговор, наше израчунавање може да се заустави.

Метод `invokeAll()` шаље све објекте из колекције `Callable` објеката и враћа листу `Future` објеката која представља решења свих задатака (послова). Када су резултати израчунавања доступни, они се могу процесирати на следећи начин:

```
List<Callable<T>> tasks = …;
List<Future<T>> results = executor.invokeAll(tasks);
for(Future<T> result: results)
   processFurther(result.get());
```

Мана оваквог приступа је у томе што се може непотребно чекати ако се деси да први посао дуго траје. Има више смисла да се резултати добијају редоследом којим постају доступни. То се може постићи коришћењем `ExecutorCompletionService`.

Започне се са егзекутором, добијеним на уобичајени начин. Затим се конструише `ExecutorCompletionService`. Пошаљу му се послови. Сервис управља блокирајућим редом `Future` објеката који садржи резултате послатих послова редом којим они постају доступни. Тако, ефикаснија организација претходног израчунавања је следећа:

```
ExecutorCompletionService service = new ExecutorCompletionService(executor);
for(Callable<T> task : tasks)
      service.submit(task);
for(int i=0; i< tasks.size(); i++)
      processFurther(service.take().get());
```

`java.util.concurrent.ExecutorService`

`T invokeAny(Collection<Callable<T>>  tasks)`
`T invokeAny(Collection<Callable<T>>  tasks, long timeout, TimeUnit unit)`
извршава дате послове и враћа резултат једног од њих. Други метод избацује `TimeoutException` ако истекне време

`List<Future<T>> invokeAll(Collection<Callable<T>> tasks)`
`List<Future<T>> invokeAll(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)`
извршава дате послове и враћа резултате свих њих. Други метод избацује `TimeoutException` ако истекне време

`java.util.concurrent.ExecutorCompletionService`

`ExecutorCompletionService(Executor e)`
конструише сервис који прикупља резултате датог егзекутора

`Future<T> submit(Callable<T> task)`
`Future<T> submit(Runnable task, T result)`
шаље посао егзекутору

`Future<T> take()`
уклања следећи готов резултат, блокирајући ако такав не постоји

`Future<T> poll()`
`Future<T> poll(long time, TimeUnit unit)`
уклања следећи готов резултат или `null` ако такав не постоји. Други метод чека задато време.

Synchronizers

Пакет `java.util.concurrent` садржи неколико класа које помажу у управљању скупом нити које међусобно сарађују. Ови механизми имају уграђену функционалност за уобичајене „рандеву“ шеме нити. Ако имамо скуп нити које сарађују пратећи неку од тих шема, можемо просто користити одговарајућу библиотечку класу уместо да покушавамо да се изборимо са ручно израђеном колекцијом катанаца и услова.

Класа	Шта ради?	Када је користити?
`CyclicBarrier`	допушта скупу нити да чека док предефинисани број њих не достигне заједничку баријеру а онда опционо извршава неку акцију	када неки број нити треба да се заврши пре него што њихови резултати могу да се користе
`CountDownLatch`	допушта скупу нити да чека док се број не декрементира на 0	када једна или више нити треба да чека да се деси задати број догађаја
`Exchanger`	допушта да две нити размене објекте када су обе спремне за размену	када две нити раде на две инстанце исте структуре података, једна пунећи једну инстанцу, а друга празнећи другу
`Semaphore`	допушта скупу нити да чека док не добије дозволе да настави	да ограничи укупан број нити које могу приступити ресурсу. Ако је број дозвола 1, користи се да блокира нити док друга нит не да дозволу
`SynchronousQueue`	допушта да нит преда објекат другој нити	да једна нит пошаље објекат другој када су обе спремне, без експлицитне синхронизације

Semaphores

Концептуално, семафор управља извесним бројем дозвола (permits). Да би прошла семафор, нит тражи дозволу позивајући метод `acquire()`. Расположив је само фиксирани број дозвола, чиме се ограничава број нити којима је дозвољено да прођу. Друге нити могу да издају дозволе позивајући метод `release()`. Не постоје стварни објекти дозволе. Семафор просто прати број. Штавише, дозвола не мора бити ослобођена од стране нити која ју је добила. У ствари, свака нит може да изда произвољан број дозвола. ~~Ако се изда већи број од максималног расположивог, семафор је просто постављен на максимални број.~~ Ово генерално чини семафоре и веома флексибилним и потенцијално збуњујућим.

Семафоре је изумео Дијкстра 1968, како би их користио као примитиве за синхронизацију. Дијкстра је показао да семафори могу бити ефикасно имплементирани и да су довољно моћни да реше многе уобичајене проблеме синхронизације нити. Препорука је да се семафори користе само када се њихово понашање добро уклапа у наш проблем синхронизације, не доводећи у опасност свој ментални склоп ☺.

Један једноставан пример је семафор са бројем дозвола 1. Такав семафор се може користити као капија за једну нит која се отвара и затвара другом нити. У наставку постоји пример у коме worker нит производи анимацију. С времена на време worker нит чека да корисник притисне дугме. Ова нит покушава да добије дозволу и мора да чека да клик на дугме изда ту дозволу.

Countdown Latches

latch - брава, квака

`CountDownLatch` омогућује да скуп нити чека док број не достигне нулу. Може се употребити само једанпут: када број достигне 0, не може се изнова инкрементирати.

Користан специјални случај је када је број 1. Тиме се имплементира капија за једнократну употребу. Нити се држе на капији док друга нит не постави број на 0.

Замислимо скуп нити којима су потребни неки иницијални подаци како би радиле. Worker нити се покрену и чекају на капији. Друга нит припрема податке. Када је она спремна, позове метод `countDown()` и све worker нити могу наставити.

Онда се може користити друга брава за проверу када су све worker нити завршиле. Иницијализује се брава бројем нити. Свака worker нит умањи број непосредно пре него што се заврши. Друга нит која убира резултате чека на брави и наставља чим се све worker нити заврше.

Barriers

Класа `CyclicBarrier`. Размотримо известан број нити које раде на деловима израчунавања. Када су сви делови готови, потребно је искомбиновати резултате. Када нит заврши свој део, пустимо је да се суочи са баријером. Након што су све нити дошле до баријере, баријера попушта и нити могу наставити.

Следе детаљи. Најпре се конструише баријера, параметар је број нити које учествују:

```
CyclicBarrier barrier = new CyclicBarrier(nThreads);
```

Свака нит ради неки посао и позива метод `await()` за баријеру на завршетку:

```
public void run(){
   doWork();
   barrier.await();
   …
}
```

Метод `await()` опционо узима `timeout` параметар:

```
barrier.await(100, TimeUnit.MILLISECONDS);
```

Ако било која од нити које чекају на баријери напусти баријеру, баријера „пуца“ (break). (Нит може отићи јер је позвала `await()` са `timeout`-ом или јер је прекинута.) У том случају, метод `await()` свих осталих нити избацује `BrokenBarrierException`. Метод `await()` нити које већ чекају непосредно се завршава.

Могуће је опционо задати тзв. „*barrier action*“ која се извршава када све нити дођу до баријере:

```
Runnable barrierAction = ...;
CyclicBarrier barrier = new CyclicBarrier(nthreads, barrierAction);
```

Акција може да покупи резултате појединачних нити.

Баријера се назива цикличном јер се може изнова користити након што се све нити које чекају ослободе. У том смислу она се разликује од `CountDownLatch`, који се може користити само једном.

Exchangers

`Exchanger` се користи када две нити раде на две инстанце истог бафера. Типично, једна нит пуни бафер, а друга троши његов садржај. Када обе заврше, размене своје бафере.

Synchronous Queues

Синхрони ред је механизам који упарује нит произвођача и потрошача. Када нит позове метод `put()` за `SynchronousQueue`, она блокира док друга нит не позове `take()` и обрнуто. За разлику од случаја са `Exchanger`-ом, подаци се шаљу само у једном смеру, од произвођача ка потрошачу.

Иако класа `SynchronousQueue` имплементира интерфејс `BlockingQueue`, она концептуално не представља ред. Не садржи никакве елементе - метод `size()` увек враћа 0.

Пример (`primer7_animacijaSemafor`): паузирање и настављање анимације

Размотримо програм који ради нешто, ажурира дисплеј, затим чека да корисник погледа резултат и притисне дугме како би се наставило, а онда ради следећу јединицу посла.

Семафор са бројем дозвола 1 може се користити за синхронизацију worker нити и нити за обраду догађаја (event dispatch thread). Worker нит позива метод `acquire()` кад год је спремна за паузу. GUI нит позива `release()` кад год корисник кликне на дугме Continue.

~~Шта се дешава ако корисник више пута кликне на дугме док је worker нит спремна? Пошто је само једна дозвола на располагању, број дозвола остаје 1.~~

Програм анимира алгоритам сортирања. Worker нит сортира низ, заустављаујући се периодично и чекајући да корисник изда дозволу за наставак рада. Корисник се може дивити приказу текућег

стања алгоритма и притиснути дугме Continue како би допустио worker нити да пређе на следећи корак.

Кôд за сортирање овде није од интереса, па се просто позива `Array.sort()`, који имплементира merge sort алгоритам. Како би се алгоритам паузирао, задат је `Comparator` објекат који чека на семафору. Тако, анимација се паузира кад год алгоритам пореди два елемента. Текуће вредности у низу се исцртавају, уз назначавање два елемента који се пореде.

Анимација приказује спајање (merging) мањих сортираних опсега у веће, али није у потпуности прецизна. Алгоритам mergesort користи други низ за привремено смештање вредности, који у овом програму није приказан. Поента овог примера није да залази у алгоритам сортирања, већ да покаже како се користи семафор за паузирање worker нити.

`java.util.concurrent.CyclicBarrier`

`CyclicBarrier(int parties)`
`CyclicBarrier(int parties, Runnable barrierAction)`
конструише цикличну баријеру за задати број учесника. `barrierAction` се извршава када сви учесници позову метод `await()` за баријеру

`int await()`
`int await(long time, TimeUnit unit)`
чека док сви учесници не позову `await()` за баријеру или док не истекне време, у ком случају се избацује `TimeoutException`. Након успеха, враћа индекс приспећа текућег учесника. Први учесник има индекс `parties-1`, а последњи 0.

`java.util.concurrent.CountDownLatch`

`CountDownLatch(int count)`
за задати број, конструише браву са одбројавањем

`void await()`
чека да брава одброји до 0

`boolean await(long time, TimeUnit unit)`
чека да брава одброји до 0 или да време истекне. Враћа `true` ако је број 0,а `false` ако је истекло време

`public void countDown()`
врши одбројавање за текућу браву

`java.util.concurrent.Exchanger<V>`

`V exchange(V item)`
`V exchange(V item, long time, TimeUnit unit)`
блокира док друга нит не позове овај метод, а затим размењује `item` са њом и враћа `item` друге нити. Други метод избацује`TimeoutException` након што истекне задато време.

`java.util.concurrent.SynchronousQueue<V>`

`SynchronousQueue()`
`SynchronousQueue(boolean fair)`
конструише синхрони ред који допушта нитима да предају ставке. Ако је `fair true`, ред фаворизује нити које дуже чекају

`void put(V item)`
блокира док друга нит не позове `take()` да узме овај `item`

`V take()`
блокира док друга нит не позове `put()`. Враћа ставку коју је друга нит обезбедила

`java.util.concurrent.Semaphore`

`Semaphore(int permits)`
`Semaphore(int permits, boolean fair)`
конструише семафор са датим ~~максималним~~ бројем дозвола. Ако је `fair true`, фаворизују се нити које дуже чекају

> *`permits` - иницијални број расположивих дозвола. Број може бити негативан и тада се најпре мора десити одговарајући број позива `release()` да би `acquire()` могао да добије тражени број дозвола.*

`void acquire()`
чека да добије дозволу

`boolean tryAcquire()`
покушава да добије дозволу; враћа `false` ако нема ниједне расположиве

`boolean tryAcquire(long time, TimeUnit unit)`
покушава да добије дозволу унутар задатог времена; враћа `false` ако нема расположивих

`void release()`
ослобађа дозволу

Нити и Swing

Када програм треба да ради нешто временски захтевно, покушајте да ангажујете другу worker нит уместо блокирања корисничког интерфејса.

Међутим, потребно је пазити шта се ради у worker нити јер Swing није thread safe. Уколико се покуша манипулисање елементима графичког интерфејса из већег броја нити, кориснички интерфејс може постати „покварен“.

Када се, након покретања примера (`primer8_invokeLater`), кликне на дугме Bad, покреће се нова нит чији метод `run()` „мучи“ комбо-бокс случајно умећући и уклањајући вредности.

```
public void run(){
   try{
         while(true){
               int i = Math.abs(generator.nextInt());
               if(i % 2 == 0)
                     combo.insertItemAt(new Integer(i), 0);
               else
                     combo.removeItemAt(i % combo.getItemCount());
               sleep(1);
         }
   }catch(InterruptedException e){}
}
```

Кликните на дугме Bad. Кликните на комбо-бокс неколико пута. Померите scrollbar. Померите прозор. Кликните на дугме Bad поново. Наставите да кликћете на комбо-бокс. На крају, требало би да видите извештај о избацивању изузетка.
Шта се дешава? Када се елемент убаци у комбо-бокс, комбо-бокс испаљује догађај да би се ажурирао дисплеј. Тада кôд за дисплеј ступа на сцену, читајући текућу величину комбо-бокса и спремајући се да прикаже вредности. Али worker нит наставља да ради - што повремено резултује смањивањем броја вредности у комбо-боксу. Кôд за дисплеј тада „мисли“ да постоји више вредности у моделу него што их је, тражи непостојеће вредности и избацује `ArrayIndexOutOfBoundsException`.

Ова ситуација се могла избећи тако што би се омогућило програмерима да закључају комбо-бокс објекат док га приказују. Међутим, дизајнери Swing-а одлучили су да не чине никакав напор како би Swing био thread safe из два разлога. Први, синхронизација захтева време, а нико не жели додатно да успорава Swing. Још значајније, Swing тим је проверио искуство које су други тимови имали са thread-safe корисничким алатима. Оно што су открили није било обећавајуће. Програмери који су користили thread-safe алате били су збуњени захтевима за синхронизацијом и често креирали програме склоне појави deadlock-ова.

Извршавање временски захтевних послова

Када се нити користе у комбинацији са Swing-ом, потребно је поштовати два једноставна правила.

- ако се акција извршава дуго, урадити је у посебној worker нити, никада у нити за обраду догађаја
- не „дирати“ Swing компоненте ни у једној другој нити осим у нити за обраду догађаја.

Разлог за прво правило је једноставно разумети. Ако узмете много времена нити за обраду догађаја, чини се да је апликација „мртва“ јер не одговара на догађаје. Посебно, у нити за обраду догађаја никада не треба да се нађу улазно/излазни позиви, који могу блокирати неодређено време и никада не би требало да се зове метод `sleep()` (ако је потребно да се чека одређено време, користити timer догађаје).

Друго правило се често зове „single-thread rule“ за Swing програмирање.

Ова два правила изгледају као да су у конфликту. Претпоставимо да се покрене посебна нит за извршавање временски захтевног посла. Обично желимо да ажурирамо кориснички интерфејс како бисмо указивали на прогрес док наша нит ради. Када се посао заврши, желимо поново да

ажурирамо GUI. Али не смемо „дирати“ Swing компоненте из своје нити. Нпр. ако желимо да ажурирамо линију прогреса или текст на лабели, не можемо просто поставити њихове вредности из своје нити.

Да би се овај проблем решио, могу се користити два метода у свакој нити како би се произвољне акције додале реду догађаја. Нпр. претпоставимо да желимо периодично да ажурирамо лабелу у нити како бисмо указали на прогрес. Не можемо рећи `label.setText()` у својој нити. Уместо тога, користимо методе `invokeLater()` или `invokeAndWait()` класе `EventQueue` како би се тај позив извршио у нити за обраду догађаја.

Ево шта радимо. Сместимо Swing кôд у метод `run()` класе која имплементира интерфејс `Runnable`. Потом се креира објекат те класе и проследи статичком методу `invokeLater()` или `invokeAndWait()`. Нпр. овако би се ажурирао текст лабеле:

```
EventQueue.invokeLater(new Runnable(){
   public void run(){
         label.setText(percentage + "% complete");
   }
});
```

Метод `invokeLater()` непосредно враћа контролу када се догађај пошаље у ред догађаја. Метод `run()` се извршава асинхроно. Метод `invokeAndWait()` чека док се метод `run()` заиста не изврши.

У ситуацији ажурирања лабеле са прогресом, метод `invokeLater()` је подеснији. Корисници би радије желели већи прогрес него да имају најпрецизнији индикатор прогреса.

Оба метода извршавају метод `run()` у нити за обраду догађаја. Нема креирања нове нити.

Пример демонстрира како се користи метод `invokeLater()` за безбедно модификовање садржаја комбо-бокса. Ако се кликне на дугме Good, нит умеће и избацује бројеве. Међутим, модификација се заправо дешава у нити за обраду догађаја.

`java.awt.EventQueue`

`static void invokeLater(Runnable runnable)`
узрокује да се метод `run()` објекта `runnable` изврши у нити за обраду догађаја након обраде свих догађаја који чекају

`static void invokeAndWait(Runnable runnable)`
узрокује да се метод `run()` објекта `runnable` изврши у нити за обраду догађаја након обраде свих догађаја који чекају. Позив блокира док се метод `run()` не заврши.

`static boolean isDispatchThread()`
враћа `true` ако је нит која извршава овај метод нит за обраду догађаја

Употреба SwingWorker

Када корисник изда команду чије процесирање траје дуго, желимо да покренемо нову нит која ће одрадити тај посао. Као што смо видели у претходној секцији, та нит треба да користи метод `EventQueue.invokeLater()` за ажурирање корисничког интерфејса.

Неколико аутора је произвело погодне класе да олакша овај програмерски задатак, а једна од њих је нашла своје место у Java SE 6. То је класа `SwingWorker`.

Програм из примера (`primer9_SwingWorker`) има команде за учитавање текстуалног фајла и за одустајање од процеса учитавања. Потребно је пробати програм са дугим фајлом, попут читавог текста „Гроф Монте Кристо“, који је пратећи материјал уз изворни кôд из књиге. Фајл се учитава у посебној нити. Док се фајл чита, ставка менија Open је онемогућена, а ставка Cancel је омогућена. Након читања сваке линије, бројач линија се ажурира у статусној линији. Након што се процес читања заврши, ставка менија Open се поново омогућује, ставка Cancel се онемогућује, а текст у статусној линији се поставља на Done.

Овај пример показује типичне UI активности позадинског посла (background task):
- након сваке јединице посла, ажурира се UI како би показао прогрес
- по завршетку посла, врши се финална измена на UI.

Класа `SwingWorker` чини имплементацију таквог посла једноставном. Предефинише се метод `doInBackground()` тако да ради временски захтеван посао и, с времена на време, позива метод `publish()` да саопшти прогрес. Метод се извршава у worker нити. Метод `publish()` узрокује извршавање метода `process()` у нити за обраду догађаја како би руковао подацима о прогресу. Када се посао заврши, позива се метод `done()` у нити за обраду догађаја тако да се може финиширати ажурирање UI.

Кад год се жели урадити неки посао у worker нити, конструише се нови worker. (Сваки `Worker` објекат је направљен тако да се користи само једном.) Потом се позива метод `execute()`. Овај метод се типично позива у нити за обраду догађаја, али то није обавезно.

Претпоставља се да worker производи резултат неког типа, према томе `SwingWorker<T, V>` имплементира `Future<T>`. Резултат се може добити методом `get()` интерфејса `Future`. Како метод `get()` блокира док резултат не постане доступан, не желимо да га зовемо непосредно након `execute()`. Добра је идеја позвати га само онда када знамо да је посао завршен. Типично, звaћемо метод `get()` из метода `done()`. (не постоји обавеза да се позива `get()`. Понекад је процесирање података о прогресу све што нам је потребно.)

И подаци о прогресу и коначни резултат могу бити произвољног типа. Класа `SwingWorker` има ове типове као своје типске параметре. `SwingWorker<T, V>` производи резултат типа `T` и податке о прогресу типа `V`.

Да би се отказао посао који је у току, користи се метод `cancel()` интерфејса `Future`. Када је посао отказан, метод `get()` избацује `CancellationException`.

Раније је речено да када worker нит позове `publish()`, то узрокује позивање метода `process()` у нити за обраду догађаја. Зарад побољшања ефикасности, резултати неколико позива метода

`publish()` могу бити обједињени у један позив метода `process()`. Метод `process()` прима `List<V>` која садржи све међурезултате.

Применимо овај механизам на читање текстуалног фајла. Испоставља се да је `JTextArea` прилично спор. Дописивање линија из дугог текстуалног фајла траје знатно.

Како би се кориснику приказао учињени прогрес, желимо да у статусној линији приказујемо број прочитаних линија. Тако се подаци о прогресу састоје од текућег броја линије и текуће линије. То упакујемо у тривијалну унутрашњу класу:

```
private class ProgressData{
   public int number;
   public String line;
}
```

Коначни резултат је текст прочитан у `StringBuilder`. Према томе, потребан нам је `SwingWorker<StringBuilder, ProgressData>`.

У методу `doInBackground()` читамо фајл, линију по линију. Након сваке линије позивамо метод `publish()` да објавимо број линије и текст текуће линије.

```
@Override
public StringBuilder doInBackground() throws IOException,
                                             InterruptedException{

      int lineNumber = 0;
      Scanner in = new Scanner(new FileInputStream(file));
      while(in.hasNextLine()){
            String line = in.nextLine();
            lineNumber++;
            text.append(line);
            text.append("\n");
            ProgressData data = new ProgressData();
            data.number = lineNumber;
            data.line = line;
            publish(data);
            Thread.sleep(1); // to test cancellation; no need to do this in
                        // your program
      }
      return text;
}
```

Такође, спавамо по милисекунду након сваке линије како бисмо опуштено могли да тестирамо отказивање, али не бисмо желели да успоравамо сопствене програме спавањем. Ако се та линија закоментарише, видећемо да се „Гроф Монте Кристо“ учита прилично брзо, са само неколико обједињених ажурирања корисничког интерфејса.

Овај програм се може направити тако да се понаша прилично глатко и ажурирањем текстуалне области из worker нити, али то није могуће за већину Swing компонената. Овде је показан генерални приступ у коме се сва ажурирања компонената дешавају у нити за обраду догађаја.

У методу `process()` игноришемо све бројеве линија осим последњег и надовезујемо све линије за јединствено ажурирање текстуалне области.

```
@Override
public void process(List<ProgressData> data){
	if(isCancelled())
		return;
	StringBuilder b = new StringBuilder();
	statusLine.setText("" + data.get(data.size()-1).number);
	for(ProgressData d: data){
		b.append(d.line);
		b.append("\n");
	}
	textArea.append(b.toString());
}
```

У методу `done()`, текстуална област се ажурира комплетним текстом, а ставка менија Cancel се онемогућује.

Приметите како је worker стартован у ослушкивачу догађаја за ставку менија Open.

Ова једноставна техника нам допушта да извршавамо временски захтевне послове задржавајући одзив корисничког интерфејса.

`java.swing.SwingWorker<T, V>`

`abstract T doInBackground()`
предефинишите овај метод тако да извршава позадински посао и врати резултат посла

`void process(List<V> data)`
предефинишите овај метод тако да процесира податке о прогресу у нити за обраду догађаја

`void publish(V… data)`
прослеђује податке о прогресу нити за обраду догађаја. Позивајте овај метод из метода `doInBackground()`

`void execute()`
распоређује текући worker за извршавање у worker нити

`SwingWorker.StateValue getState()`
враћа стање текућег worker-а. То може бити PENDING, STARTED или DONE.

The Single-Thread Rule

Свака Јава апликација почиње методом `main()` који се извршава у main нити. У Swing програму main нит живи кратко. Она распоређује конструкцију корисничког интерфејса у нит за обраду догађаја и потом се завршава. Након конструкције корисничког интерфејса, нит за обраду догађаја процесира нотификације о догађајима, попут позива метода `actionPerformed()` или `paintComponent()`. Друге нити, попут нити која шаље догађаје у ред догађаја се извршавају иза сцене и ове нити су невидљиве програмеру (на апликативном нивоу).

Раније смо увели правило једне нити: „Не дирајте Swing компоненте ни у једној другој нити осим у нити за обраду догађаја“. У овој секцији даље истражујемо то правило.

Постоји неколико изузетака од правила једне нити:
- безбедно је додавати и уклањати ослушкиваче догађаја у произвољној нити. Наравно, методи ослушкивача биће позивани у нити за обраду догађаја.
- мали број Swing метода је thread-safe. Они су специјално означени у API документацији реченицом „This method is thread safe, although most Swing methods are not“.

Најкориснији међу њима су:

```
JTextComponent.setText()
JTextArea.insert()
JTextArea.append()
JTextArea.replaceRange()
JComponent.repaint()
JComponent.revalidate()
```

Сврха метода `revalidate()` је да форсира изглед (layout) компоненте након што се садржај променио. Традиционални AWT има метод `validate()` за форсирање изгледа компоненте. За Swing компоненте се уместо њега просто позива `revalidate()`. Међутим, како би се форсирао изглед за `JFrame`, и даље се мора позивати `validate()` - `JFrame` је `Component`, али не и `JComponent`.

Историјски, правило једне нити допуштало је више. Свакој нити било је допуштено да конструише компоненте, поставља њихова својства и додаје их у контејнере, све док ниједна од компонената није *реализована*. Компонента је реализована ако може примати догађаје за цртање и валидацију. Ово је случај чим се метод `setVisible(true)` или `pack()` позове за компоненту или је компонента додата у контејнер који је реализован.

Та верзија правила једне нити била је погодна. Допуштала је да се GUI креира у методу `main()`, а затим позове `setVisible(true)` за прозор апликације. Није било потребе распоређивати `Runnable` у нит за обраду догађаја.

На жалост, имплементатори неких компонената нису обратили пажњу на суптилности оригиналног правила једне нити. Они су покретали активности у нити за обраду догађаја не мучећи се да провере да ли је компонента реализована. Нпр. ако позовете `setSelectionStart()` или `setSelectionEnd()` за `JTextComponent`, померање курсора се распоређује у нит за обраду догађаја, чак и ако компонента није видљива.

Било је могуће открити и поправити ове проблеме, али Swing дизајнери су изашли лакшим путем. Прогласили су да никада није безбедно приступати компонентама из било које нити осим из нити за обраду догађаја. Према томе, морате да конструишете кориснички интерфејс у нити за обраду догађаја, користећи позив `EventQueue.invokeLater()`.

Наравно, постоји мноштво програма који нису тако пажљиви и живе по старој верзији правила једне нити, иницијализујући кориснички интерфејс у main нити. Такви програми су изложени незнатном ризику да нека од иницијализација корисничког интерфејса узрокује акције у нити за обраду догађаја које су у конфликту са акцијама у main нити. Према томе, треба просто поштовати стриктно правило једне нити.