Wątki

Wątek jest pojedynczym sekwencyjnym strumieniem sterowania wewnątrz programu. Mówiąc o wątkach i ich wzajemnych relacjach bierze się pod uwagę: kto je stworzył, kiedy powinny zostać uruchomione lub zatrzymane, jaką funkcje realizują, etc.

Metody tworzenia wątków.

Istnieją dwie metody tworzenia wątków:
1.Utworzyć podklasę klasy Thread, przysłaniając metodę run własną metodą, a następnie utworzyć składnik tej nowej podklasy i wywołać metodę run.

```
class A extends Thread {
        public A(String name) {super(name);}
        public void run() {
          System.out.println(„My name is ” + getName());
        }
}
class B {
        public static void main(String[] args) {
          A a = new A(„mud”);
          a.start();
        }
}
```

2. Zaimplementować interfejs (uruchomieniowy) Runnable w klasie z publiczną metodą run, stworzyć składnik tej klasy, przekazać referencje do tego obiektu do konstruktora Thread. Zaletą jest tu możliwość rozszerzenia (extends) klas zdefiniowanych przez użytkownika.

```
class C extends ....  implements Runnable {
        public void run() {
          System.out.println(„Wątek o nazwie  ” + Thread.currentThread().getName());
        }
}
class B {
        public static void main(String[] args) {
          C c = new C();
          Thread t = new Thread(c, „mud too”);
          t.start();
        }
}
```

Możliwe stany wątków:

tworzony (new), gotowy (runnable), wykonywany (running), zawieszony (suspended), zablokowany (blocked), zawieszony-zablokowany (suspended-blocked), zakończony (dead). Przejście pomiędzy stanami możliwe jest dzięki wywołaniu którejś z funkcji:

* yield, sleep (metody statyczne w klasie Thread, stosują się więc tylko do bieżąco wykonywanego wątku);
* resume, stop, suspend, start, join (metody składowe (instancyjne), które mogą wywołać dowolne wątki na dowolnym obiekcie typu Thread);
* wait(może być bez parametrów i z parametrami: (long milisec) oraz (long milisec, int nanosec)), notify, notifyAll (muszą być wywoływane z wnętrza bloku synchronized).

wait(), wait(timeout), wait(timeout,nanoseconds) –milisec,nanosec.

Tabela stanów wątków.

Current	New State

State	runnable	running	suspended	blocked	suspended-blocked	dead
new	start					
runnable		scheduled	suspend			stop
running	time slice ends, yield		suspend	blocking IO, sleep, wait, join		stop, run ends
suspended	resume					stop
blocked	IO completes, sleep expires, notify, notifyAll, join completes				suspend	stop
suspenden-blocked			IO completes	resume		stop

W Javie nowo tworzony wątek dziedziczy priorytet wątku, który go stworzył. Wątki obdarzane są priorytetami, w zakresie od MIN_PRIORITY do MAX_PRIORITY (stałe zdefiniowane w klasie Thread, standardowo w Javie priorytety mają wartości od 1 do 10, przy czym normalnie dla wątków 5). Standardowo wątkom przyznawany jest priorytet NORM_PRIORITY, ale można też użyć setPriority, getPriority, aby go zmienić.

```
public class TestPriorytetow {
      public static void main(String argv[]) {
        C c = new C();
          Thread t1 = new Thread(c, "pierwszy watek");
          Thread t2 = new Thread(c, "drugi watek");
          t2.setPriority(t1.getPriority()+1);
          t1.start();
          t2.start();
}
```

W każdej chwili, gdy wiele wątków gotowych jest do działania, runtime system wybiera ten wątek (runnable) który ma najwyższy priorytet i go wykonuje. Tylko w przypadku, gdy działający wątek zatrzymuje się (po stop), oddaje sterowanie (po yield), lub przestaje być wątkiem działającym (not runnable), wątek o niższym priorytecie zacznie się wykonywać. W przypadku wątków o tym samym priorytecie, scheduler wybiera jeden z nich na zasadzie round-robin. Wybrany wątek będzie działał dopóty, dopóki nie stanie się prawdziwe co najmniej jedno ze zdań:

- Wątek o wyższym priorytecie staje się gotowym do wykonania (runnable).
- Bieżący wątek oddaje sterowanie (po yield) lub jego metoda run się kończy.
- W systemie w podziałem czasu skończy się okres przydzielony dla wątku

Gdy któreś z powyższych zdań okaże się prawdziwe, wtedy zacznie działać wątek o niższym priorytecie. Jednak reguła ta może okazać się fałszywa, gdy scheduler został tak zaimplementowany, aby nie doprowadzać do zagłodzenia wątków o niskich priorytetach.

Uwaga: przy pracy z wątkami należy pamiętać o ich synchronizacji i zabezpieczeniu zmiennych współdzielonych. Zobacz opis do słów: volatile, synchronized.

Podział czasu. Standardowo procesor przydzielany jest wątkom na okres 100 ms. Dzieje się tak jednak tylko dla Windows95/NT. W przypadku JDK1.1 dla systemu Solaris 2.x nie było zaimplementowanego podziału czasu. Aby sprawdzić, z jakim schedulerem ma się do czynienia, wystarczy uruchomić następujący przykład:

```
class C extends ....  implements Runnable {
        public void run() {
          System.out.println(„Wątek o nazwie  ” + Thread.currentThread().getName());
           // aby zapewnic przełaczanie wątków niezależnie od schedulera można w tym
miejscu wywołać
           // Thread.yield();  lub równoważnie Thread.currentThread().yield();
            }
}

public class TestSchedule raWatkow {
      public static void main (String argv[]) {
        DzialajacyWatek dw = new DzialajacyWatek();
        new Thread(dw, "pierwszy watek").start();
        new Thread(dw, "drugi watek").start();
        new Thread(dw, "trzeci watek").start();
        }
}
```

Jeśli na ekranie wyświetlany będzie cały czas komentarz „Wątek o nazwie pierwszy wątek”, znaczy to, że nie ma przełączania wątków. W takim przypadku można zaimplementować własny scheduler, który uruchamiany będzie z najwyższym priorytetem i który będzie dokonywał przełączeń pomiędzy procesami wykorzystując metodę nup(). Wątek taki powinien być uruchomiony przed wszystkimi innymi wątkami oraz powinien wywoływać metodę setDeamon(true).

```
public class Scheduller implements Runnable {
  private int timeSlice = 0; // milliseconds
  private Thread t = null;

  public Scheduller(int timeSlice) {

        this.timeSlice = timeSlice;
        t = new Thread(this);
        t.setPriority(Thread.MAX_PRIORITY);
        t.setDaemon(true);      t.start();
  }
  public void run() {
        int napping = timeSlice;
        while (true) {
                try{
                        t.sleep(napping);
                } catch(InterruptedException e){}
        }
  }
}
```

Poniżej zamieszczony jest kolejny przykład na uruchamianie wątków. W przykładzie tym przechwytywany jest wyjątek, wykorzystana jest metoda join() oraz currentThread().

```
public class TestUruchamianiaWatkow {
      public static void main (String argv[]) {
        C c = new C();
        while (true){
```

```
        Thread t = new Thread(c, “wątek pierwszy”);
        System.out.println("operator new Thread() wykonany" + (t == null ? "blednie" ;
"poprawnie") + ".");
        t.start();
        try {
            t.join(); // czeka aż wątek zakończy wykonywanie metody run()
           }
         catch (InterruptedException ignored){}
       }
      }
}
```

Jeden wątek może przerwać inny wątek przez wywołanie metody składowej interrupt przerywanego wątku. Jeśli przerywany wątek jest w zablokowany po sleep, join lub wait, metody te wysyłają obiekt InterruptedException zamiast się normalnie kończyć. Jeśli wątek nie jest zablokowany, to ustawia się po jego przerwaniu odpowiednia logiczna flaga. Flagę można sprawdzić metodą składową isInterrupted (zwracającą logiczną wartość). Dla wygody w klasie Thread zamieszczono statyczną metodę interrupted, która wywołuje currentThread().isInterrupted(). Wywołanie to resetuje flagę (czego nie robi isInterrupted).

Do sprawdzania „żywotności” wątków służą metody isDeamon, isAlive, getName. isAlive zwraca false dla nowego wątku albo wątku zakończonego (dead thread), zwraca true dla wątków ruchomionych metodą start i nie zatrzymanych (tj. wątków gotowych do działania lub jak i wątków niegotowych (runnable or not runnable), przy czym nie można rozróżnić wątku nowego od skończonego jak również wątku gotowego od wątku niegotowego. Ponadto do oceny stanu wątków można posłużyć się mechanizmem wyjątków. W poniższym przykładzie przechwytywany jest wyjątek ThreadDeath.

```
public class TestZatrzymania {
      public static void main(String argv[]) {
        Thread t = new Thread(new DzialajacyWatek());

        try {
           t.start();
           metodaMogacaZatrzymacWatek();
        } catch (ThreadDeath aTD) {
           // tu jest miejsce na zareagowanie na przerwanie watku
           throw aTD; // przekaz blad dalej
        }
}
```

Wątki można grupować w pewne zbiory. Do tworzenia zbiorów wyjątków służy klasa ThreadGroup.

Dostęp do pamięci. Zgodnie z definicją języka Java, wpisywanie danych do komórek pamięci nie musi odbywać się w kolejności, w jakiej zadeklarowane to zostało w programie. Aby taką kolejność zachować, deklaruje się nadpisywaną przez wątki zmienną słowem volatile

```
class D {
  static int x = 0, y = 0;
  static void a() {x = 3, y = 4; }
  static int b() {int z = y; z += x; return z;}
}
```

Jeśli różne wątki wywołają a() i b(), to zwracane wartości przez b mogą być następujące: 0, 3, 4, 7 (gdzie 4 jest zwrócone, jeśli a zapisze w y wartość 4 i będzie to widoczne dla b wcześniej niż zapis wartości 3 w x. Jeśli obie zmienne byłyby zadeklarowane jako

```
        static volatile int x =0, y =0;
```

to b nigdy nie zwróciłoby 4.

Aktywne czekanie:

sensownie	raczej źle
while (buffer[putIn].occupied) Thread.currentThread().yield();	while (buffer[putIn].occupied);

Można też aktywne czekanie zaimplementować następująco

```
while (condition) try {wait();} catch (InterruptedException e) {}
```

i używać notifyAll.

Semafory. W standardzie Javy nie wyróżniono semaforów. Dlatego trzeba je symulować przy użyciu monitorów (które mieszczą się już w standardzie tego języka). Pewną „namiastką" semaforów binarnych jest blok synchronizacji, służący zabezpieczaniu sekcji krytycznej.
Przykład

synchronized (obj) { *critical section* }	obj może tu być dowolnym obiektem.

Jeśli wszystkie sekcje krytyczne znajdują się w metodach pojedynczego obiektu, wtedy blok synchronizacji używa this (referencje do tego obiektu)

```
synchronized (this) { ... }
```

Ponadto, jeśli ciało metody stanowi blok synchronizacji z this, tj.

```
type method ( ... ) { synchronized (this) {critical section} }
```

kompilator Javy pozwala na użycie zapisu

```
synchronized type method ( ... ) {critical section}
```

Jeśli wzajemne wykluczanie dotyczy wielu różnych klas bądź obiektów, wtedy obiekt wspólny dla synchronizowanych bloków musi zostać stworzony na zewnątrz klasy i musi być przekazany do ich konstruktorów.

private Object mutex = null;
....
mutex = this ;
....
public void run(){
for (int m = 1; m <= M; m++)
synchronized (mutex) { sum = fn (sum, m); } }
	mutex traktowany jest tu jako zmienna warunkowa, na której dokonuje się podnoszenia i opuszczania.

Uwaga: semafory binarne można używać do wzajemnego wykluczania jak i do synchronizacji procesów. Użycie bloku synchronizacji umożliwia tylko to pierwsze.

synchronized – zabezpiecza przed jednoczesnym wykonaniem fragmentu kodu zawierającego dany obiekt przez współbieżnie działające wątki.

// zabezpieczenie przypisania w metodzie print względem obiektu TryPointPrinter

```
public class TryPointPrinter {
        public void print (Point p) {
                float safeX, safeY;

                synchronized(this) {
                        safeX = p.x();
                        safeY = p.y();
                }
                System.out.println("x =", + safeX + ", y=" + safeY );
        }
}
```

// zabezpieczenie przypisania w metodzie print względem obiektu p

```
public class TryPointPrinter {
	public void print (Point p) {
		float safeX, safeY;

		synchronized(p) {
			safeX = p.x();
			safeY = p.y();
		}
		System.out.println("x =", + p.x() + ", y=" +p.y() );
	}
}
```

// bezpiecznie zadeklarowana klasa Point

```
public class Point {
	private float x, y;
	public float x() { return x; }
	public float y() { return y;}
	public synchronized void setXY (float newX, float newY) {
		x = newX;
		y = newY;
	}
}
```

// źle, zmienna statyczna dostępna dla wszystkich obiektów LicznikStatyczny

```
public class LicznikStatyczny {
	private static int licznik;

	public synchronized void zwiększLicznik() {
		licznik ++;
	}
}
```

// dobrze, bo teraz synchronizacja na właściwym obiekcie - klasie

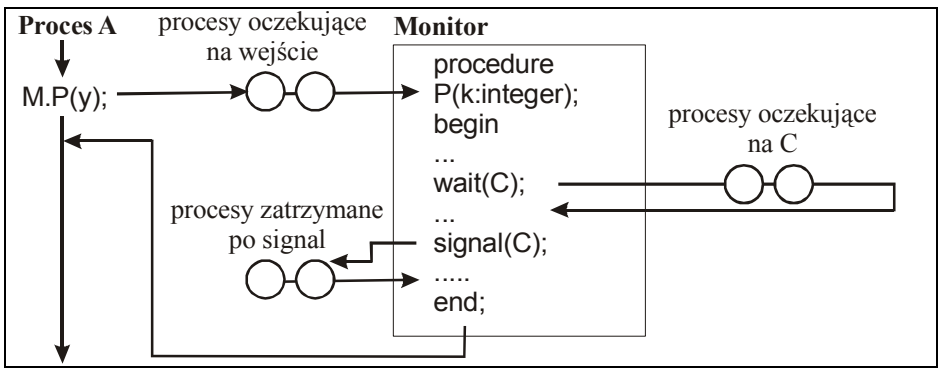
```
public class LicznikStatyczny {
	private static int licznik;

	public void zwiększLicznik() {
		synchronized ( getClass()) {
		licznik ++;
		}
	}
}
```

// jeśli zmienna statyczna jest publicznie dostępna, można ją zabezpieczyć następująco (choć jest to w złym stylu):

```
		synchronized (Class.forName("LicznikStatyczny")) {
			LicznikStatyczny.licznik ++;
		}
```

Monitory.



Generalnie dyscyplina sygnałów w monitorze może być opisana zachowaniem: signal_and_exit, signal_and_wait, signal_and_continue. Kolejkowanie wątków nie musi być typu FIFO. Wątki, które wykonały signal nie muszą wyjść z monitora, wątki wznawiane po signal wcale nie muszą wykonać się przed wątkami czekającymi na skorzystanie z monitora.

W monitorach Javy zaimplementowane jest dyscyplina signal_and_continue. Do wysyłania sygnałów służą metody notify lub notifyAll, oczekiwanie zaś realizuje się metodą wait. Każdy monitor w Javie ma pojedynczą nienazwaną anonimową zmienną warunkową, na której odbywa się oczekiwanie i wysyłanie sygnałów. Ta zmienna odpowiada obiektowi, na którym odbywa się synchronizacja (synchronized). Dlatego też wait(), notify(), notifyAll() mogą być wywołane tylko wewnątrz synchronizowanej metody.

Zazwyczaj monitory blokują wszystkie zaimplementowane w nich metody. Monitory Javy umożliwiają synchronizację części metod. Metody niezsynchronizowane mogą mieć formę publicznego dostępu i wywoływać metody zsynchronizowane, które są prywatne.

bbmo.java

```
class BoundedBuffer {       // designed for a single producer thread
                            // and a single consumer thread
  private int numSlots = 0;
  private double[] buffer = null;
  private int putIn = 0, takeOut = 0;
  private int count = 0;

  public BoundedBuffer(int numSlots) {
    if (numSlots <= 0) throw new IllegalArgumentException("numSlots<=0");
    this.numSlots = numSlots;
    buffer = new double[numSlots];
    System.out.println("BoundedBuffer alive, numSlots=" + numSlots);
  }

  public synchronized void deposit(double value) {
    while (count == numSlots)
      try {
        wait();
      } catch (InterruptedException e) {
        System.err.println("interrupted out of wait");
      }
    buffer[putIn] = value;
    putIn = (putIn + 1) % numSlots;
    count++;                  // wake up the consumer
    if (count == 1) notify();  // since it might be waiting
    System.out.println(" after deposit, count=" + count
      + ", putIn=" + putIn);
  }

  public synchronized double fetch() {
    double value;
    while (count == 0)
      try {
        wait();
      } catch (InterruptedException e) {
        System.err.println("interrupted out of wait");
      }
    value = buffer[takeOut];
    takeOut = (takeOut + 1) % numSlots;
    count--;                        // wake up the producer
    if (count == numSlots-1) notify(); // since it might be waiting
    System.out.println(" after fetch, count=" + count
      + ", takeOut=" + takeOut);
    return value;
  }
}
```

bbpc.java

```
import Utilities.*;

class Producer extends MyObject implements Runnable {

  private int pNap = 0; // milliseconds
  private BoundedBuffer bb = null;

  public Producer(String name, int pNap, BoundedBuffer bb) {
    super(name);
    this.pNap = pNap;
    this.bb = bb;
  }

  public void run() {
    double item;
    int napping;
    while (true) {
      napping = 1 + (int) random(pNap);
      System.out.println("age=" + age() + ", " + getName()
        + " napping for " + napping + " ms");
      CurrentThread. sleep(napping);
      item = random();
      System.out.println("age=" + age() + ", " + getName()
        + " produced item " + item);
      bb.deposit(item);
      System.out.println(getName() + " deposited item " + item);
    }
  }
}

class Consumer extends MyObject implements Runnable {

  private int cNap = 0; // milliseconds
  private BoundedBuffer bb = null;

  public Consumer(String name, int cNap, BoundedBuffer bb) {
    super(name);
    this.cNap = cNap;
    this.bb = bb;
  }

  public void run() {
    double item;
    int napping;
    while (true) {
      napping = 1 + (int) random(cNap);
      System.out.println("age=" + age() + ", " + getName()
        + " napping for " + napping + " ms");
      CurrentThread.sleep(napping);
      System.out.println("age=" + age() + ", " + getName()
        + " wants to consume");
      item = bb.fetch();
      System.out.println(getName() + " fetched item " + item);
    }
  }
}

class ProducerConsumer extends MyObject {

  public static void main(String[] args) {

    // parse command line arguments, if any, to override defaults
```

```
    GetOpt go = new GetOpt(args, "Us:p:c:R:");
    go.optErr = true;
    String usage = "Usage: -s numSlots -p pNap -c cNap -R
runTime";
     int ch = -1;
     int numSlots = 20;
     int pNap = 3;       // defaults
     int cNap = 3;       // in
     int runTime = 60;   // seconds
     while ((ch = go.getopt()) != go.optEOF) {
       if      ((char)ch == 'U') {
         System.out.println(usage);  System.exit(0);
       }
       else if ((char)ch == 's')
         numSlots = go.processArg(go.optArgGet(), numSlots);
       else if ((char)ch == 'p')
         pNap = go.processArg(go.optArgGet(), pNap);
       else if ((char)ch == 'c')
         cNap = go.processArg(go.optArgGet(), cNap);
       else if ((char)ch == 'R')
         runTime = go.processArg(go.optArgGet(), runTime);
       else {
         System.err.println(usage);  System.exit(1);
       }
     }
     System.out.println("ProducerConsumer: numSlots=" +
numSlots + ", pNap="
        + pNap + ", cNap=" + cNap + ", runTime=" + runTime);

     // create the bounded buffer
     BoundedBuffer bb = new BoundedBuffer(numSlots);

     // start the Producer and Consumer threads
     Thread producer = new Thread(new
Producer("PRODUCER", pNap*1000, bb));
     Thread consumer = new Thread(new Consumer("Consumer",
cNap*1000, bb));
     producer.start();
     consumer.start();
     System.out.println("All threads started");

     // let them run for a while
     nap(runTime*1000);
     System.out.println("age()=" + age()
       + ", time to stop the threads and exit");
     producer.stop();
     consumer.stop();
     System.exit(0);
   }
}
```

Grupowanie wątków

Każdy wątek w Javie jest członkiem grupy wątków (*thread group*). Grupy umożliwiają dokonywania operacji na wielu wątkach równocześnie (np. można uruchomić lub zawiesić wszystkie wątki należące do tej samej grupy pojedynczym wywołaniem odpowiedniej metody). Do grupowania wątków służy klasa ThreadsGroup w pakietu java.lang.
W czasie działania programu wątki umieszczane są w grupach w czasie ich tworzenia. Decyzję, do jakiej grupy przypisać wątek, pozostawia się programiście. Może on zdecydować, aby wątek trafił do grupy standardowej (default group) lub też do grupy zdefiniowanej i utworzonej w programie. Po przydzieleniu wątka do grupy nie można już później tego przypisania zmienić.

Standardowa grupa wątków (The Default Thread Group)

Jeśli wywołanie konstruktora przy tworzeniu nowego wątku nie zawierało specyfikacji grupy, nowy wątek zostanie przypisany do tej samej grupy co grupa wątku, w którym go stworzono (current thread group , current thread).

Podstawową grupą wątków jest grupa nazywana main. Grupę tę tworzy system podczas pierwszego uruchomienia aplikacji Javy. Jeśli programista nie zmodyfikuje sposobu tworzenia nowych obiektów, wszystkie wątki umieszczone zostaną w tej samej grupie.

Uwaga: Jeśli nowy wątek został stworzony wewnątrz apletu, jego grupą niekoniecznie musi być main. O tym, do jakiej grupy wątek zostanie przypisany decydować będzie przeglądarka, w której aplet uruchomiono.

Tworzenie grupy wątków

Wątki przypisywane są do grup podczas ich tworzenia. W klasie Thread istnieją trzy konstruktory, które to umożliwiają:

public Thread(ThreadGroup *group*, Runnable *runnable*)
public Thread(ThreadGroup *group*, String *name*)
public Thread(ThreadGroup *group*, Runnable *runnable*, String *name*)

Każdy z tych konstruktorów tworzy nowy wątek, inicjalizuje go odpowiednio do wartości parametrów Runnable i String, oraz przypisuje wątek do podanej grupy.
W poniższym przykładzie zadeklarowana i stworzona jest nowa grupa wątków, do której przypisany zostaje nowo stworzony wątek:

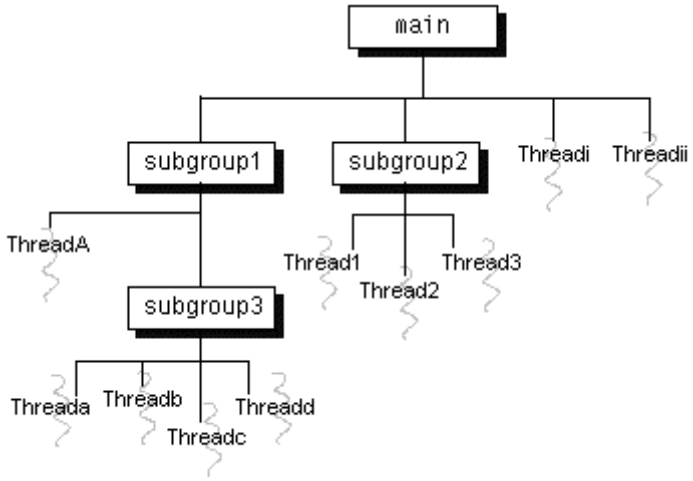
```
ThreadGroup myThreadGroup = new ThreadGroup("My Group of Threads");
Thread myThread = new Thread(myThreadGroup, "a thread for my group");
```

Parametr ThreadGroup przekazany do konstruktora wątku nie musi być grupą zdefiniowaną przez użytkownika. Może to być grupa stworzona przez *Java runtime system,* lub grupa stworzona przez aplikację, w której uruchomiony został aplet (np. przez przeglądarkę).
Aby dowiedzieć się, do jakiej grupy wątek należy, wystarczy wywołać metodę getThreadGroup

```
theGroup = myThread.getThreadGroup();
```

Klasa ThreadGroup

Służy do grupowania dowolnej liczby wątków.
ThreadGroup może zawierać nie tylko same wątki, ale także inne grupy wątków. Najwyżej w hierarchii grup w aplikacjach Javy stoi grupa o nazwie main. Można tworzyć wątki i grupy wątków w podgrupach grupy main tworząc „drzewiasta" strukturę.



Metody klasy ThreadGroup można podzielić na:

- Metody zarządzania kolekcjami (*Collection Management Methods*) - służą do zarządzania kolekcjami wątków i podgrupami, które grupa zawiera
- Metody operujące na grupie - służą do ustawiania lub odczytywania atrybutów grupy (obiektu ThreadGroup)
- Metody operujące na wszystkich wątkach wewnątrz grupy - są to metody, które wykonują operacje jak start i resume na wszystkich wątkach i podgrupach wątków

- Metody ograniczania dostępu – związane są z menadżerem bezpieczeństwa, dotyczą ograniczania dostępu do wątku opartego o członkostwo (przynależność do grupy).

Metody zarządzania kolekcjami wątków:

activeCount - liczba aktywnych wątków w grupie. Często używana w połączeniu z metodą enumerate, aby otrzymać tablicę referencji do wszystkich aktywnych wątków w grupie.

Przykład (tworzenie tablicy aktywnych wątków i wydrukowywanie ich nazw):

```
public class EnumerateTest {
   public void listCurrentThreads() {
      ThreadGroup currentGroup =
               Thread.currentThread().getThreadGroup();
      int numThreads = currentGroup.activeCount();
      Thread[] listOfThreads = new Thread[numThreads];

      currentGroup.enumerate(listOfThreads);
      for (int i = 0; i < numThreads; i++)
         System.out.println("Thread #" + i + " = " +
                              listOfThreads[i].getName());
   }
}
```

Mamy też do dyspozycji activeGroupCount oraz list.

Metody operujące na grupie wątków:

Klasa ThreadGroup dostarcza wielu atrybutów, które można ustawiać i odczytywać, a dotyczą one grupy jako całości. Atrybuty te zawierają: maksymalny priorytem, jaki może mieć wątek należący do grupy, parametr mówiący o tym, czy grupa jest grupą typu demon ("daemon" group); nazwa grupy, ojciec grupy.
Metody, które ustawiają i odczytują parametry klasy ThreadGroup operują na poziomie grupy. Dzięki nim można odczytać lub zmienić atrybut obiektu klasy ThreadGroup, nie wpływając na żaden z wątków wewnątrz grupy. Metody operujące na poziomie grupy są następujące:

getMaxPriority and setMaxPriority
getDaemon and setDaemon
getName
getParent and parentOf
toString

Jeśli na przykład setMaxPriority zmieni maksymalny priorytet wątków w grupie, to zmieniony zostanie atrybut tylko obiektu reprezentującego grupę, nie zmienione natomiast zostaną priorytety samych wątków wewnątrz grupy.
W przykładzie poniżej tworzona jest grupa wątków:

```
public class MaxPriorityTest {
   public static void main(String[] args) {

      ThreadGroup groupNORM = new ThreadGroup(
                        "A group with normal priority");
      Thread priorityMAX = new Thread(groupNORM,
                     "A thread with maximum priority");

      // set Thread's priority to max (10)
      priorityMAX.setPriority(Thread.MAX_PRIORITY);

      // set ThreadGroup's max priority to normal (5)
      groupNORM.setMaxPriority(Thread.NORM_PRIORITY);
```

```
        System.out.println("Group's maximum priority = " +
                           groupNORM.getMaxPriority());
        System.out.println("Thread's priority = " +
                           priorityMAX.getPriority());
    }
}
```

Kiedy tworzony jest obiekt ThreadGroup groupNORM, dziedziczy on maksymalny priorytet grupy wątków, z której się wywodzi (grupy ojcowskiej). W tym przypadku jest to maximum (MAX_PRIORITY) ustalone przez Java runtime system. Następnie tworzony jest wątek priorityMAX, któremu przypisany zostaje maksymalny priorytet dopuszczany przez Java runtime system. W następnej kolejności obniżony zostaje priorytet dla grupy wątków do wartości (NORM_PRIORITY). Wywołanie w tym miejscu metody setMaxPriority nie wpływa na priorytet wątku priorityMAX. Mamy więc sytuacje, w której wątek priorityMAX ma priorytet 10, który jest wyższy od maksymalnego priorytetu grupy groupNORM do której należy.
Wyjściem programu jest:

```
Group's maximum priority = 5
Thread's priority = 10
```

Tak więc metoda setMaxPriority wpływa tylko na te wątki w grupie, które stworzone zostały przed jej wywołaniem. Ma ona natomiast wpływ na wszystkie nowe wątki stworzone w grupie, wszystkie podgrupy oraz na wątki, dla których wywołana została metoda setPriority.
Podobnie zmiana statusu daemon będzie dotyczyła tylko nowych wątków w grupie oraz kolejnych podgrup. Status „deamon" grupy wątków oznacza, że grupa zostanie zniszczona jeśli wszystkie wątki w grupie zakończą się.

Metody operujące na wszystkich wątkach wewnątrz grupy:
Zalicza się do nich trzy metody, mogące zmienić stan wszystkich wątków w grupie i podgrupach jednocześnie:
resume
stop
suspend

Metody ograniczania dostępu:
Klasa ThreadGroup sama w sobie nie daje żadnych możliwości, aby ograniczyć prawa dostępu wątkom z jednej grupy do metod wątków z innej grupy. Aby wprowadzić takie ograniczenia klasy Thread oraz ThreadGroup współpracują z menadżerami bezpieczeństwa(SequrityManager).

Obie klasy, tj. klasa Thread oraz klasa ThreadGroup, posiadają metodę checkAccess. Metoda ta wywołuje metodę checkAcces bieżącego menadżera bezpieczeństwa. Jeśli dostęp jest niedozwolony, metoda checkAccess zgłasza wyjątek SecurityException. W przeciwnym razie checkAccess po prostu kończy swe działanie.
Poniżej wymieniona jest lista metod klasy ThreadGroup, które wywołują metodę checkAccess klasy ThreadGroup, przed własnym wykonaniem (są to metody o tzw. regulowanym dostępie).

ThreadGroup(ThreadGroup *parent*, String *name*)
setDaemon(boolean *isDaemon*)
setMaxPriority(int *maxPriority*)
stop
suspend
resume
destroy

A oto lista metod w klasie Thread, które wywołują checkAccess przed własnym wykonaniem:

konstruktory, które definiuja grupę wątku
stop
suspend

resume
setPriority(int *priority*)
setName(String *name*)
setDaemon(boolean *isDaemon*)

Samodzielna aplikacja Javy nie posiada menadżera bezpieczeństwa przez domniemanie (default). Nie ma więc żadnych ograniczeń na dostęp i modyfikację parametrów dotyczących wątków w grupie. Zmian mogą dokonywać jedne wątki względem drugich, niezależnie od tego, z jakiej grupy pochodzą.
Aby zaimplementować własny menadżer bezpieczeństwa dostępu, wystarczy stworzyć klasę pochodna klasy SecurityManager, przysłonić odpowiednie metody w tej klasie i następnie zainstalować SecurityManager jako bieżącego menadżera bezpieczeństwa dla uruchamianej aplikacji.

Użycie klas Timer i TimerTask

Klasy timerów wprowadzone zostały w wersji Javy 1.3. Zdefiniowano je w pakiecie java.util. Klasa Timer służy do szeregowania (sheduling) instancji klasy nazywanej TimerTask.

Uwaga, w pakiecie javax.swing.Timer isnieje również klasa Timer. Jest ona użyteczniejsza od klasy z pakietu java.util.Timer przy projektowaniu GUI. Przy projektowaniu GUI można też posłużyć się klasą SwingWorker. Służy ona do przeprowadzania zadań w wątku biegnącym w tle, pozwala również na odświeżenie GUI, kiedy zadanie się skończy.

Przykład:
Reminder.java

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

/**
 * Simple demo that uses java.util.Timer to schedule a task to execute
 * once 5 seconds have passed.
 */

public class Reminder {
   Timer timer;

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

   class RemindTask extends TimerTask {
      public void run() {
         System.out.println("Time's up!");
         timer.cancel(); //Terminate the timer thread
      }
   }

   public static void main(String args[]) {
      System.out.println("About to schedule task.");
      new Reminder(5);
      System.out.println("Task scheduled.");
   }
}
```

Po uruchomieniu programu wyświetlone zostaną komunikaty:

Task scheduled.
Five seconds later, you see this:
Time's up!

Program ten pokazuje prosty przykład implementacji, w której jakieś zadanie ma zostać wykonane przez wątek Timera w określonym czasie. Na początek zaimplementowana jest klasa RemindTask, która dziedziczy po klasie TimerTask. Metoda run w zaimplementowanej klasie zawiera kod zadania.
Program zaczyna się od stworzenia wątku Timera. Następnie tworzony jest obiekt zawierający metodę run (new RemindTask()) i przekazany jest on do Timera (timer.schedule(new RemindTask(), seconds*1000);). W podanym przykładzie metoda schedule wywołana jest z parametrem mówiacym o opóźnienieniu, z jakim wykonane ma zostać zadania. Można też użyć metody schedule, w której czas uruchomienia zadania podany jest bezwzględnie:

```
//Get the Date corresponding to 11:01:00 pm today.
Calendar calendar = Calendar.getInstance();
calendar.set(Calendar.HOUR_OF_DAY, 23);
calendar.set(Calendar.MINUTE, 1);
calendar.set(Calendar.SECOND, 0);
Date time = calendar.getTime();

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

Zatrzymywanie Timerów:
Domyślnie program działa tak długo, jak długo działa wątek timera. Wątek timera można przerwać na trzy sposoby:

1. przez wywołanie metody cancel na timerze. Wywołanie tej funkcji może nastąpić w dowolnym miejscu programu (np. w metodzie run).
2. przez stworzenie wątku timera jako wątku-demona (deamon): jeśli wszystkie pozostawione przez program wątkami są demonami, program kończy się. Aby stworzyć demona, w konstruktorze Timera należy umieścić argument true: new Timer (true).
3. gdy wszystkie zadania przeznaczone do uruchomienia zakończyły się, usunięcie wszystkich referencji do timera (obiektu klasy Timer) zakończy jego działanie (choć może nie natychmiast).
4. przez wywołanie metody System.exit. Ten sposób stosowany jest, gdy w użyciu są klasy AWT.

Wykonywanie zadań wielokrotnie:

```
schedule(TimerTask task, long delay, long period) // mogą pojawić się opóźnienia
schedule(TimerTask task, Date time, long period)
scheduleAtFixedRate(TimerTask task, long delay, long period) // opóźnienia są redukowane
scheduleAtFixedRate(TimerTask task, Date firstTime, long period)
```

```
timer.schedule(new RemindTask(),
        0,      //initial delay
        1*1000);  //subsequent rate
```

Tworzenie własnych menadżerów bezpieczeństwa

Każda aplikacja Javy może mieć własny menadżer bezpieczeństwa. Menadżer ten jest obiektem, który przez cały czas działania aplikacji jest odpowiedzialny za sprawowanie pieczy nad wykonywaniem niebezpiecznych operacji, jak. np. odczytywanie czy zapisywanie plików. W pakiecie java.lang umieszczono abstrakcyjna klasę SecurityManager, która dostarcza programowego interfejsu oraz częściowej implementacji dla wszystkich menadżerów bezpieczeństwa Javy.

Domyślnie aplikacja Javy nie posiada żadnego menadżera bezpieczeństwa. Znaczy to, że po jej uruchomieniu system nie wprowadza żadnych ograniczeń na wykonywanie przez nią operacji. Aby ograniczyć swobodę aplikacji należy zaimplementować i jej przypisać własny menadżer bezpieczeństwa.

Uwaga: W istniejących przeglądarkach oraz aplikacjach do uruchamiania apletów menadżerowie bezpieczeństwa stanowią integralną ich część. W chwili uruchomienia danej przeglądarki zaczyna istnieć związany z nią menadżer bezpieczeństwa. Wszystkie aplety uruchamiane wewnątrz przeglądarki będą musiały podlegać restrykcjom związanego z nią menadżera bezpieczeństwa.

Bieżący menadżer bezpieczeństwa obowiązujący w danej aplikacji można uzyskać przez wywołanie metody getSecurityManager() z klasy System:

```
SecurityManager appsm = System.getSecurityManager();
```

Jeśli po wywołaniu metoda getSecurityManager()zwróci wartość null, znaczyć to będzie, że z bieżącą aplikacją nie jest związany żaden menadżer bezpieczeństwa. W takim przypadku, tj. dopóki nie zostanie stworzony własny menadżer bezpieczeństwa, w aplikacji nie będzie można korzystać z żadnych jego metod.

Jeśli menadżer bezpieczeństwa istnieje, to można się do niego odwoływać w celu uzyskania zezwolenia na wykonanie pewnych operacji. I faktycznie, wiele klas z pakietów Javy własnie to robi. Na przykład metoda System.exit(), przerywająca działania iterpretara Javy, zwraca się do menadżera bezpieczeństwa o zezwolenie na wykonanie operacji wywołując jego metodę checkExit():

```
SecurityManager security = System.getSecurityManager();
if (security != null) {
   security.checkExit(status);
}
. . .
// code continues here if checkedExit() returns
```

Metoda checkExit() wykona się normalnie, jeśli menadżer bezpieczeństwa zezwali na wykonanie metody exit(). Jeśli jednak zezwolenie takie nie zostanie udzielone, metoda checkExit() zgłosi wyjątek SecurityException. Tak więc ostatecznie w obsłudze wyjątku można zareagować na brak zezwolenia do wykonania danej operacji.

W klasie SecurityManager zdefiniowanych jest wiele różnych metod to weryfikacji zezwoleń na wykonywanie innych operacji. Na przykład metoda checkAccess() służy do weryfikacji operacji na wątkach.

Implementacja menadżera bezpieczeństwa

Aby napisać własny menadżer bezpieczeństwa należy na początek stworzyć klasę będącą pochodną klasy SecurityManager. W podklasie klasy SecurityManager należy następnie przesłonić te metody klasy SecurityManager, które będą wykorzystane w aplikacji do zapewnienia bezpieczeństwa wykonywanych operacji.

W poniższym przykładzie omówione zostaną szczegóły związanych z zapewnieniem bezpieczeństwa wykonywania operacji na plikach.

Domyślnie przy próbie otwarcie pliku do czytania wywoływana jest metoda checkRead() bieżącego menadżera bezpieczeństwa. Podobnie przy próbie otwarcia pliku do zapisu wywoływana jest jego metoda checkWrite(). Jeśli menadżer bezpieczeństwa zezwala na wykonanie danej operacji, wywołanie odpowiedniej metody check*XXX*() kończy się normalnie. W przeciwnym przypadku check*XXX*() zgłosza wyjątek SecurityException.

Aby wprowadzić własne ograniczenia na dostęp do systemu plików należy stworzyc menadżera bezpieczeństwa będącego potomkiem klasy SecurityManager. Menadżer ten powinien przysłaniać metody checkRead() oraz checkWrite() klasy bazowej, które odpowiedzialne są za weryfikacje praw dostępu do pliku. Klasa SecurityManager dostarcza trzy wersje metody checkRead() oraz dwie wersje klasy checkWrite(). Każda z nich powinna zostać przysłonięta.

Niech więc w naszej alikacji otwieranie pliku do oczytu bądź zapisu zabezpieczone będzie hasłem wprowadzanym przez użytkownika. Podanie poprawnego hasła pozwoli na wykonanie danej operacji. Jeśli hasło nie będzie poprawne, dostęp do plików zostanie zabroniony. Realizację takiej strategii zapewni menadżer bezpieczeństwa PasswordSecurityManager zadeklarowany następująco:

```
class PasswordSecurityManager extends SecurityManager {
private: String password;
   . . .
}
```

Występujący w tej deklaracji prywatny parametr password przechowywać będzie hasło umożliwiające wykonywanie operacji na plikach. Hasło to inicjalizowane będzie w konstruktorze menadżera bezpieczeństwa:

```
PasswordSecurityManager(String password) {
   super();
   this.password = password;
}
```

Do zadawania pytań o hasło i jego weryfikacji przeznaczona zostanie metoda accessOK(). Będzie ona zadeklarowana jako prywatna metoda klasy PasswordSecurityManager. Metoda ta zwróci true, jeśli użytkownik wprowadzi poprawne hasło, w przeciwnym wypadku wartością zwracaną będzie false.

```
private boolean accessOK() {
   int c;
   DataInputStream dis = new DataInputStream(System.in);
   String response;

   System.out.println("What's the secret password?");
   try {
      response = dis.readLine();
      if (response.equals(password))
         return true;
      else
         return false;
   } catch (IOException e) {
      return false;
   }
}
```

Na koniec w klasie PasswordSecurityManager umieszczone zostaną trzy metody checkRead()oraz dwie metody checkWrite() przysłaniające swoje odpowiedniki z klasy SecurityManager.

```
public void checkRead(FileDescriptor filedescriptor) {
```

```
    if (!accessOK())
        throw new SecurityException("Not a Chance!");
}
public void checkRead(String filename) {
    if (!accessOK())
        throw new SecurityException("No Way!");
}
public void checkRead(String filename, Object executionContext) {
    if (!accessOK())
        throw new SecurityException("Forget It!");
}
public void checkWrite(FileDescriptor filedescriptor) {
    if (!accessOK())
        throw new SecurityException("Not!");
}
public void checkWrite(String filename) {
    if (!accessOK())
        throw new SecurityException("Not Even!");
}
```

Wszystkie metody check*XXX*() korzystają z metody accessOK(), która umożliwia pobranie hasła od użytkownika i jego weryfikację. Dla niepoprawnego hasła check*XXX*() zgłosi wyjątek SecurityException. Dla hasła poprawnego check*XXX*() kończy się normalnie. Uwaga: SecurityException jest wyjątkiem runtime i dlatego metody, w których on występuje nie muszą być deklarowane z użyciem słowa kluczowego throws.

Uwaga: W przeglądarkach obowiązuje zazwyczaj taka implementacja, w której aplety odczytane z sieci nie mogą czytać ani pisać w lokalnym systemie plików, chyba że użytkownik zatwierdzi takie operacje.

W klasie SecurityManager istnieje szereg zdefiniowanych metod check*XXX*() służących weryfikacji praw dostępu dla różnorakich operacji. Aby zaimplementować własny menadżer bezpieczeństwa nie trzeba ich wszystkich przesłaniać. Wystarczy, że zaimplementowane zostaną tylko te metody, które są potrzebne. Niemniej w domyślnej implementacji klasy SecurityManager wszystkie metody check*XXX*() zgłaszaja wyjątki SecurityException. W innych słowach, klasa SecurityManager z definicji wprowadza ograniczenia dla wszystkich operacji, w których uwzględnia się warunki bezpieczeństwa. Dlatego może okazać się, że we własnej implementacji menadżera bezpieczeństwa trzeba będzie przysłonić wiele z metod check*XXX*() aby otrzymać zadowalające rozwiązanie.
Wszystkie metody check*XXX*() klasy SecurityManager działają w ten sam sposób:

- jeśli dostęp jest dozwolony, po prostu kończą się
- jeśli dostęp jest zabroniony, zgłaszaja wyjątek SecurityException.

Własne metody przysłaniające metody check*XXX*() z klasy SecurityManager powinny zachowywać się w ten sam sposób.

Podsumowując, napisanie własnego menadżera bezpieczeństwa ogranicza się do wykonania dwóch kroków:

- stworzenia klasy dziedziczącej po klasie SecurityManager
- przysłonienia wybranych metod

Problemem w implementacji może okazać się wybór metody do przysłonięcia. W poniższym zezstawieniu wymieniono obiekty, na których można przeprowadzać różnego rodzaju operacje (pierwsza kolumna) oraz zestaw metod z klasy SecurityManager odpowiedzialnych za kontrolę bezpieczeństwa wykonania tych operacji.

Operacja na	Dostępne metody
sockets	checkAccept(String *host*, int *port*)

	checkConnect(String *host*, int *port*) checkConnect(String *host*, int *port*, Object *executionContext*) checkListen(int *port*)
threads	checkAccess(Thread *thread*) checkAccess(ThreadGroup *threadgroup*)
class loader	checkCreateClassLoader()
file system	checkDelete(String *filename*) checkLink(String *library*) checkRead(FileDescriptor *filedescriptor*) checkRead(String *filename*) checkRead(String *filename*, Object *executionContext*) checkWrite(FileDescriptor *filedescriptor*) checkWrite(String *filename*)
system commands	checkExec(String *command*)
interpreter	checkExit(int *status*)
package	checkPackageAccess(String *packageName*) checkPackageDefinition(String *packageName*)
properties	checkPropertiesAccess() checkPropertyAccess(String *key*) checkPropertyAccess(String *key*, String *def*)
networking	checkSetFactory()
windows	checkTopLevelWindow(Object *window*)

Generalnie wywołania metod check*XXX*() zaimplementowane są w pakietach klas Javy, z którymi wiążą się zagadnienia bezpieczeństwa. Jeśli więc korzysta się z tych klas, metod check*XXX*() nie trzeba jawnie wywoływać. System sam wywoła odpowiednią metodę w odpowiednim czasie. Na przykład metoda checkAccess(ThreadGroup *g*) wywoływana jest w chwili tworzenia nowej grupy watków (obiektu klasy ThreadGroup), ustawiania statusu grupy jako demona, zatrzymania (wykonania stop), itp. Kiedy więc tworzy się własny menadżer bezpieczeństwa, należy pamiętać w jakich sytuacjach nastąpi niejawne wywołanie metody check*XXX*(), a kiedy wywołanie to będzie jawne. Ponadto, w zależności od przyjętej strategii bezpieczeństwa można dokonac wyboru, które z metod check*XXX*() należy przesłonić, a które nie.

Instlacja menadżera bezpieczeństwa

Aby we własnej aplikacji można było skorzystać z zaimplementowanego menadżera bezpieczeństwa, należy go zainstalować. Do tego celu służy metoda setSecurityManager() z klasy System. Menadżer bepieczeństwa może być w aplikacji zdefiniowany tylko raz, tj. metoda System.setSecurityManager() może być wywołana tylko raz w przeciągu całego czasu działania aplikacji. Kolejne wywołania metody System.setSecurityManager() w aplikacji powodowałyby zgłaszanie wyjątku SecurityException.
W poniższym przykładzie pokazano aplikacje, które instaluje obiekt klasy PasswordSecurityManager jako bieżący menadżer bezpieczeństwa, a następnie sprawdza, czy nowy menadżer zezwala na wykonanie operacji otwarcia do odczytu i otwarcia do pisania wykonywanych na plikach.

```
import java.io.*;

public class SecurityManagerTest {
   public static void main(String[] args) throws Exception {

      BufferedReader buffy = new BufferedReader(
                                    new InputStreamReader(System.in));

      try {
         System.setSecurityManager(
                           new PasswordSecurityManager("ThisIsPass", buffy));
      } catch (SecurityException se) {
         System.err.println("SecurityManager already set!");
      }
```

```
        BufferedReader in = new BufferedReader(new FileReader("inputtext.txt"));
        PrintWriter out = new PrintWriter(new FileWriter("outputtext.txt"));
        String inputString;
        while ((inputString = in.readLine()) != null)
            out.println(inputString);
        in.close();
        out.close();
    }
}
```

Instalacja menadżera bezpieczeństwa odbywa się na początku metody main:

```
try {
    System.setSecurityManager(new PasswordSecurityManager("ThisIsPass"));
} catch (SecurityException se) {
    System.out.println("SecurityManager already set!");
}
```

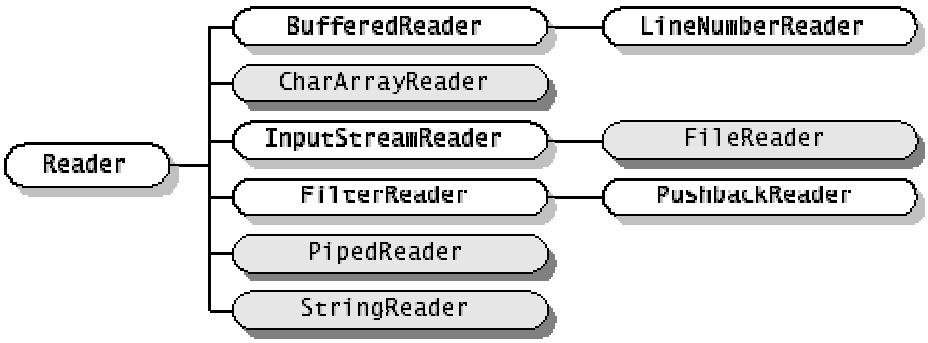
(Metoda setSecurityManager wywołana jest z argumentem będącym nowo stworzonym obiekt klasy PasswordSecurityManager z hasłem "ThisIsPass"). W dalszej części programu w bloku try..catch otwierane są dwa pliki. Wywołanie konstruktorów FileInputStream oraz FileOutputStream niejawnie powoduje wywołanie metod checkAccess() zainstalowanego menadżera bezpieczeństwa PasswordSecurityManager.

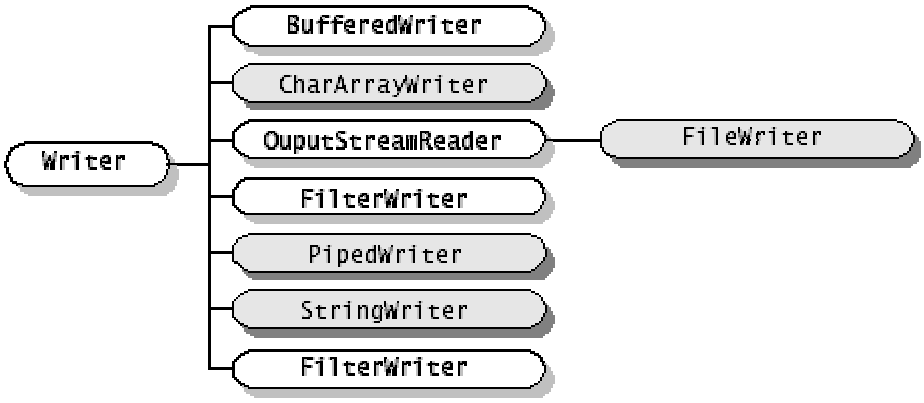
```
try {
    DataInputStream fis = new DataInputStream(
                          new FileInputStream("inputtext.txt"));
    DataOutputStream fos = new DataOutputStream(
                           new FileOutputStream("outputtext.txt"));
    String inputString;
    while ((inputString = fis.readLine()) != null) {
        fos.writeBytes(inputString);
        fos.writeByte('\n');
    }
    fis.close();
    fos.close();
} catch (IOException ioe) {
    System.err.println("I/O failed for SecurityManagerTest.");
}
```

Po uruchomieniu powyższego przykładu użytkownik zostanie zapytany dwukrotnie o hasło: raz przy otwieraniu pierwszego z plików, drugi raz przy otwieraniu drugiego z plików. Jeśli wprowadzone hasło okaże się fałszywe, zostanie zgłoszony wyjątek i aplikacja zakończy się.

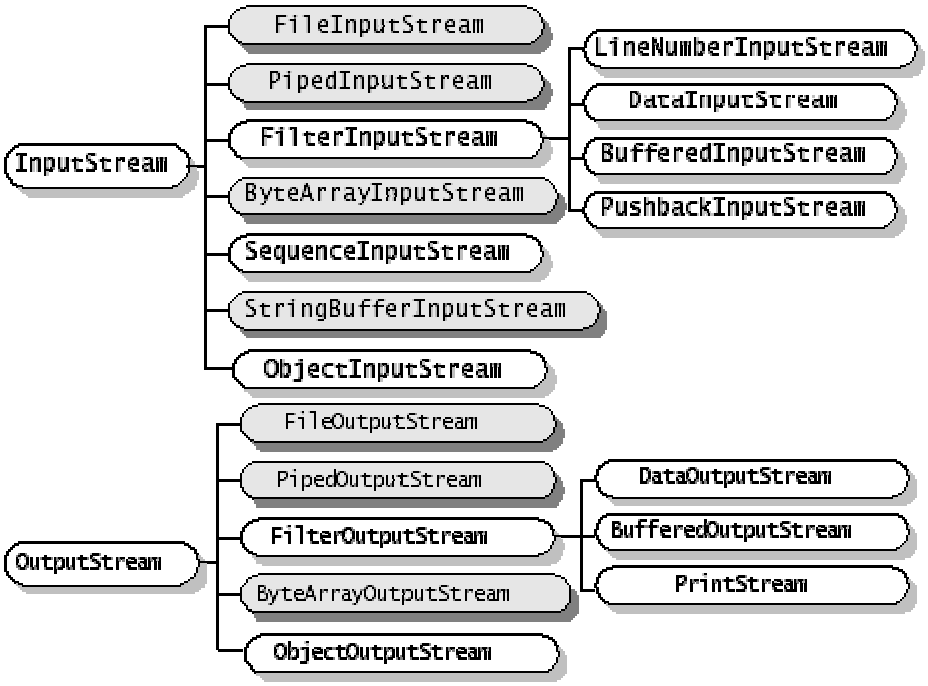
Strumienie

Strumienie znaków





Strumienie bajtów



Reader i InputStream dostarczają podobnych metod służących do odczytu danych:

Reader (czytanie znaków i tablic znaków)	InputStream (czytanie bajtów i tablic bajtów)
int read() int read(char cbuf[]) int read(char cbuf[], int offset, int length)	int read() int read(byte cbuf[]) int read(byte cbuf[], int offset, int length)

Writer i OutputStream dostarczają podobnych metod służących do zapisu danych:

Writer (pisanie znaków i tablic znaków)	IOutputStream (pisanie bajtów i tablic bajtów)
int write(int c) int write(char cbuf[]) int write(char cbuf[], int offset, int length)	int write(int c) int write(byte cbuf[]) int write(byte cbuf[], int offset, int length)

Ponadto Reader i InputStream dostarczają metod:

- do zaznaczania położenia w strumieniu i resetowania położenia (mark() oraz reset() w powiązaniu z markSupported())
- opuszczania pewnej ilości danych (skip() – ile bajtów opuścić),
- określających ilość dostępnych danych (available() z InputStream) – ile bajtów dostępnych bez blokowania strumienia oraz ready() z Reader - czy są jakieś znaki do odczytania),
- zamykających jawnie strumień (close())

Domyślnie implementacje metod markSupported() oraz reset() w klasach InputStream i Reader działają w identyczny sposób: markSupportred() zwraca false, reset() zgłasza wyjątek IOException. Jednakże mark() klasy InputStream nie wykonuje żadnych czynności, zaś mark() klasy InputStream zgłasza wyjątek IOException.

Przykład:

```
import java.io.*;

public class CopyBytes {
   public static void main(String[] args) throws IOException {
      File inputFile = new File("farrago.txt");
      File outputFile = new File("outagain.txt");

      FileInputStream in = new FileInputStream(inputFile);
      FileOutputStream out = new FileOutputStream(outputFile);
      int c;

      while ((c = in.read()) != -1)
        out.write(c);

      in.close();
      out.close();
   }
}
```

```
DataInput d=new DataInputStream(new BufferedInputStream(new FileInputStream("farrago.txt")));

try {
while (true) {
 byte b = (byte) d.readByte();
...
}
} catch (EOFException e) {
... //koniec pliku
}

String linia;
while ((linia=d.readLine()) != null) { ... }
```

Class java.util.StringTokenizer

```
public static String replace(String line, String find_key,
        String replace_key)
{
        StringBuffer result = new StringBuffer();

        for (StringTokenizer tokenizer =
                new StringTokenizer(line, " \t\n,.:;\"", true);
                tokenizer.hasMoreElements();  )
        {
                String word = (String) tokenizer.nextElement();

                if (word.equals(find_key))
                        result.append(replace_key);
                else
                        result.append(word);
        }

        return result.toString();
}
```