Procházka na provázku

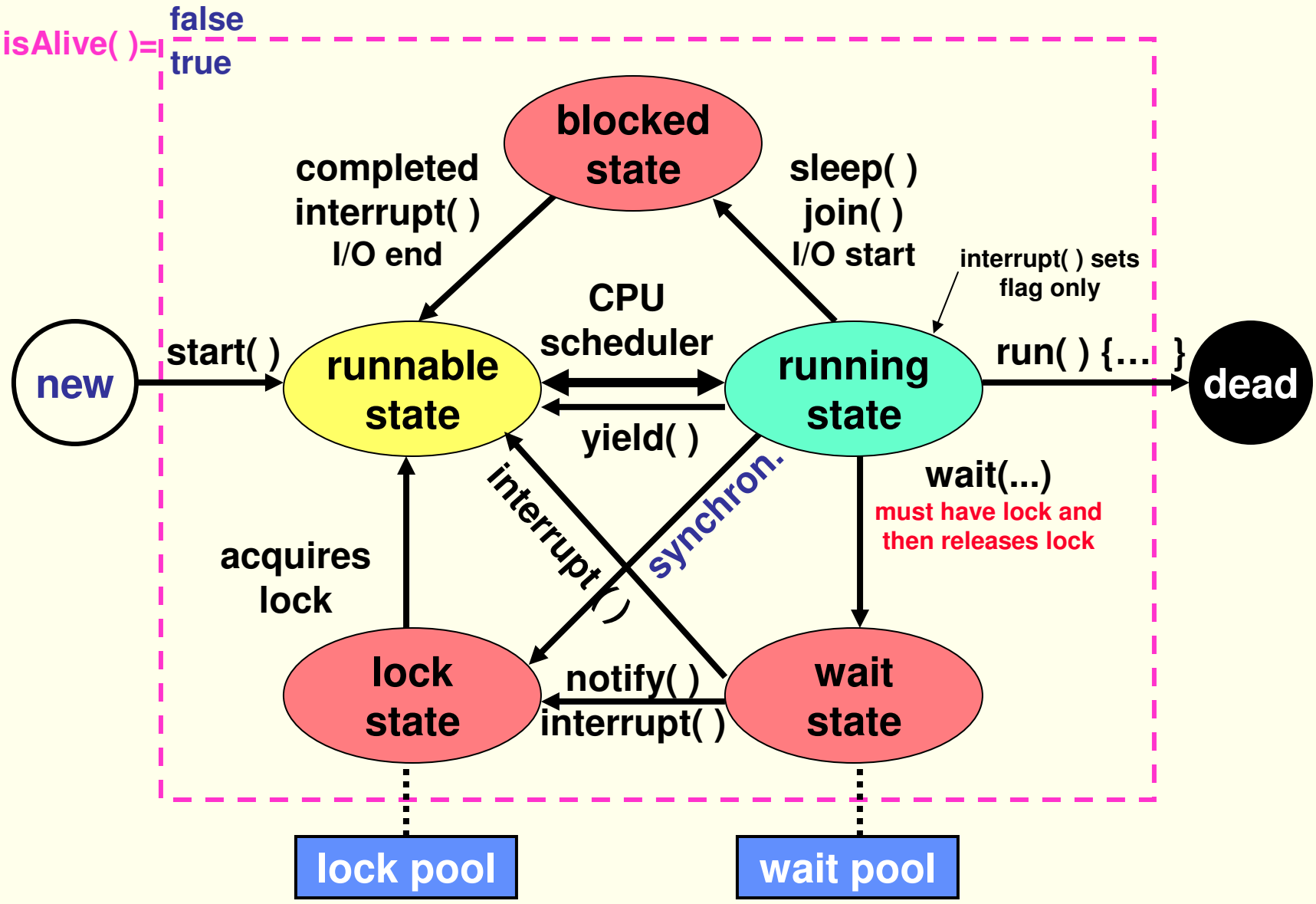
Program s mnoha metodami připomíná knósský labyrint s mnoha sály, jimiž se protlouká statečný Threadeus, jemuž pro šťastný návrat chytrá Ariadna dala klubíčko. Threadeus pokaždé při vstupu do metody vlákno rozvíjí a při návratu z metody svíjí. V metodách Threadeus provádí příkazy.

V jávském labyrintu však může pobíhat najednou i více hrdinů a dokonce i rekurzivně – neb je multithreadový. (Leč v jednom procesoru se v daném okamžiku pohybuje nanejvýš jeden – tak jako v "člověče nezlob se".)

Každý Threadeus má svoje vlákno - druhé konce třímá JVM v roli Ariadny, aby v případě nehody (...Exception) mohla ony smolaře v labyrintu najít a z té šlamastiky vytáhnout. Totiž ani padlí gerojové tam nesmějí zůstat.

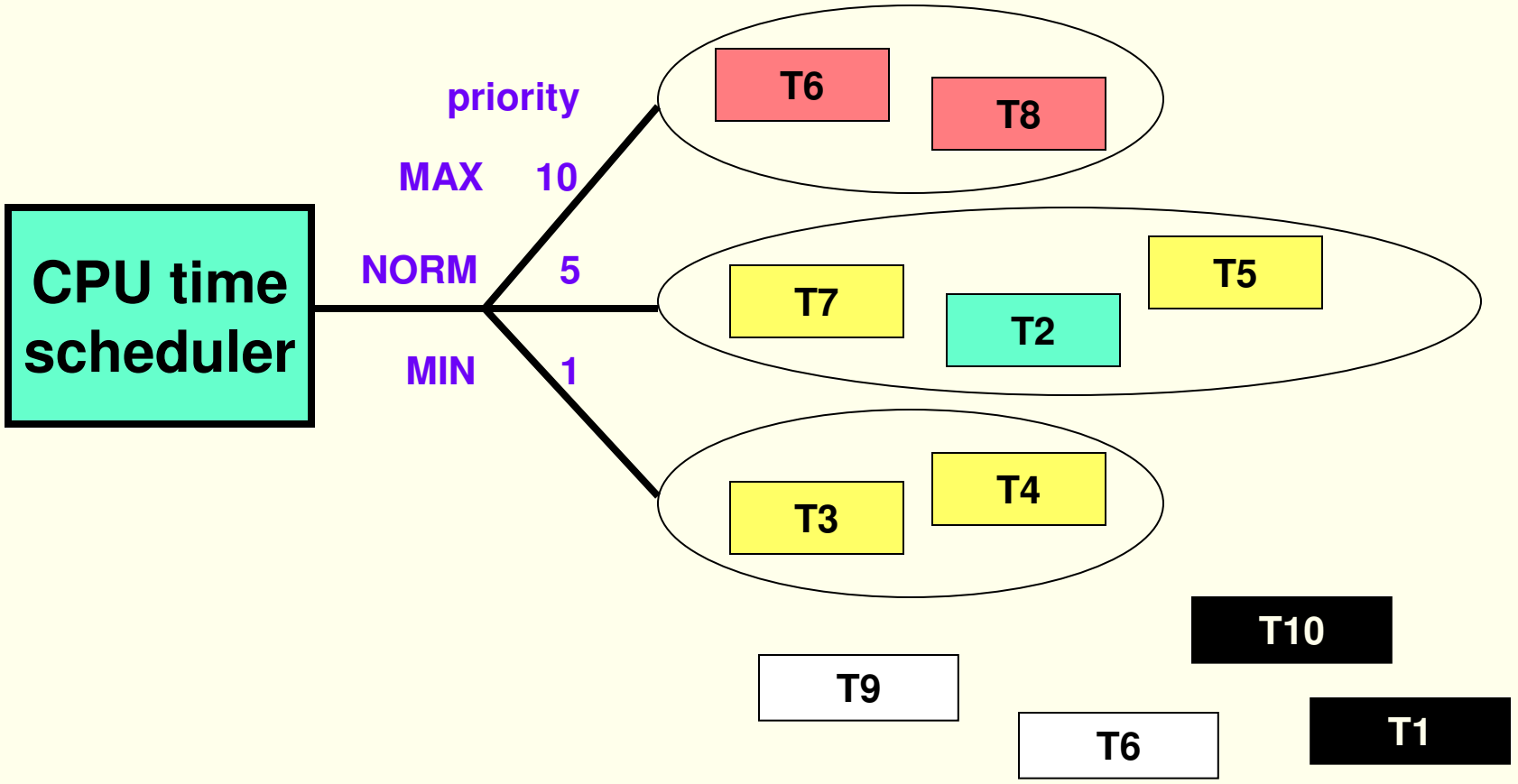
V labyrintu se mohou potulovat také démoni – služební džinové, kteří se však vytratí, není-li tam již žádný živý opravdový hrdina. Také démoni jsou vláknem připoutáni k JVM.

Threadeovy osudy



Přidělování času

JVM scheduler přiděluje CPU čas nespecifikovaným způsobem některému vláknu s nejvyšší prioritou, pokud ono je ve stavu runnable.



Vlákno = thread = lightweight process = task ...

Vlákno není program – je to jakýsi výpočtář, tj. objekt vykonávající činnost.

K provedení výpočtu je nezbytný alespoň jeden výpočtář, který má:

- **k dispozici program – tedy předpis postupu (metody),**
- **přístupná data což jsou atributy tříd, objektů, parametry, proměnné,**
- **čas na práci.**

Vlákno si musí dynamicky pamatovat:

1. **v které metodě se právě nalézá a kam se pak vrátit,**
2. **své lokální proměnné (obé je v aktivačním záznamu na zásobníku),**
3. **adresu aktuálního příkazu v aktuální metodě (program counter).**

Výpočet končí, skončí-li všechna nedémonická vlákna.

Otázka jak je vlákno dlouhé, není nesmyslná: jeho délka se dá chápat jako momentální počet aktivačních záznamů na zásobníku.

Thread a Runnable

JVM scheduler dodává CPU čas tím, že zavolá metodu run() objektů typu java.lang.Thread, zařazených do prioritních front jsou-li ve stavu runnable-running.

V těle metody public void run( ) se definuje požadované chování vlákna – lze volat též metody libovolného objektu či třídy.

Interfejs java.lang.Runnable obsahuje pouze metodu: public void run();

Thread je konkrétní třída implementující Runnable "skoro prázdnou", metodou umožňující předat řízení metodě run v cílovému objektu takto:

```
public void run( )  {  if ( target != null )  target.run();  }
```

V potomcích Threadu se metoda run( ) přepisuje na neprázdnou – definující požadovanou činnost vlákna.

Metoda run( )

Metoda public void run( ) { … } tedy definuje, co má vlákno dělat.
Volá ji JVM a nikoli programátor - ten pouze vlákno odstartuje metodou start( ), čímž se vlákno zařadí do fronty a oživí - isAlive( ) vrací true.

Metoda run( ) typicky obsahuje cyklus, který probíhá, pokud jiné vlákno nenastaví condition na false. Tak je zajištěno, že vlákno dokoná definovaným způsobem (zavrženou metodou stop( ) nikoli).

```
class X implements Runnable {
  boolean condition = true;    // počáteční nastavení
  public void run( ) {
       while ( condition ) {
           ......                        // požadovaná činnost
       }
   }   // smrtící závorka
}
```

Konstruktory

Thread má osm přetížených konstruktorů s různými kombinacemi těchto parametrů:

- **Runnable target - přesměrování dodávky CPU do určeného objektu, který implementuje interfejs Runnable.**
- **String name - jméno vlákna (chybí-li, dosadí se systematické jméno).**
- **ThreadGroup group - přiřazení vlákna do skupiny, změna nemožná. Skupinu vláken lze ovládat najednou.**
- **long stackSize - nastavení velikosti zásobníku.**

Statické metody

- Thread currentThread( ) - odkaz k běžnému vláknu.
- int activeCount( ) - počet aktivních vláken ve skupině běžného vlákna.
- void dumpStack( ) - výpis zásobníku.
- boolean holdsLock(Object o) - test zda vlákno má zámek objektu.
- boolean interrupted( ) - bylo toto vlákno přerušeno? A výmaz příznaku.
- native void sleep(long msec) throws InterruptedException – dočasné uspání. Vyhazuje výjimku při předčasném probuzením jiným vláknem.
- void yield( ) - vlákno přejde ze stavu running do runnable, tj. zřekne se zbytku momentálního přídělu času.
- int enumerate (Thread[ ] tarr) – do určeného pole zkopíruje všechna aktivní vlákna skupiny běžného vlákna a jejích podskupin.

Nestatické metody

- void run( ) - definice funkcionality.
- void start( ) - aktivace, tj. oživení (mrtvé vlákno nelze oživit).
- boolean isAlive( ) - test života.
- int getName( ) / void setName(String name) - getter/setter jména.
- int getPriority( ) / void setPriority(int priority) - getter/setter priority.
- Thread.State getState( ) – stav pro monitoring ne pro synchonizaci.
- ThreadGroup getThreadGroup( ) – skupina do které vlákno patří.
- boolean isDaemon( ) - test démona.
- void setDaemon(boolean on) - nastavení démona/nedémona.
- void interrupt( ) - přerušení vlákna.
- boolean isInterrupted( ) - bylo ono vlákno přerušeno? Příznak nezměněn.
- void join( ) throws InterruptedException - čekání na konec jiného vlákna.
- void join(long msec) throws InterruptedException – s maximem čekání.

Všeobecné, zastaralé a méně důležité metody zde nejsou uvedeny.

Nedémon defaultně vytváří nedémona, démon démona.
Démoničnost je nastavitelná jen před spuštěním dotyčného vlákna.

Způsoby použití

S vlákny lze pracovat dvojím způsobem:

1. **Definovat potomka třídy Thread, v něm přepsat metodu run( ) pro požadovanou funkcionalitu, zkonstruovat objekt a ten aktivovat zděděnou metodou start( ).**
   Toto řešení se hodí jen pro velmi jednoduché případy.

2. **Definovat jakoukoli třídu implementující interfejs Runnable a do ní přepsat metodu run( ) pro požadovanou funkcionalitu. Zkonstruovat příslušný objekt. Zkonstruovat objekt Thread s parametrem Runnable odkazující na příslušný objekt. Nakonec odstartovat objekt Thread.**
   Toto řešení je obecnější, běžící objekt může být složitý, neboť může výhodně dědit od bohaté třídy.

První způsob: Tik

```
class Tik extends Thread {
    public void run( ) {                        // přepsaná metoda Threadu
        try {
            while( true ) {
                System.out.println( " Tik " );
                sleep( 1000 );                  // static method
            }
        }
        catch( InterruptedException ex ) {  }
    }
}
```

spuštění zkráceně: `new Tik( ).start( );`

spuštění rozepsaně:
```
Tik t = new Tik( );
t.start( );
```

Protože Tik je Thread, startem se zařadí do fronty na CPU čas.

Druhý způsob: Tak

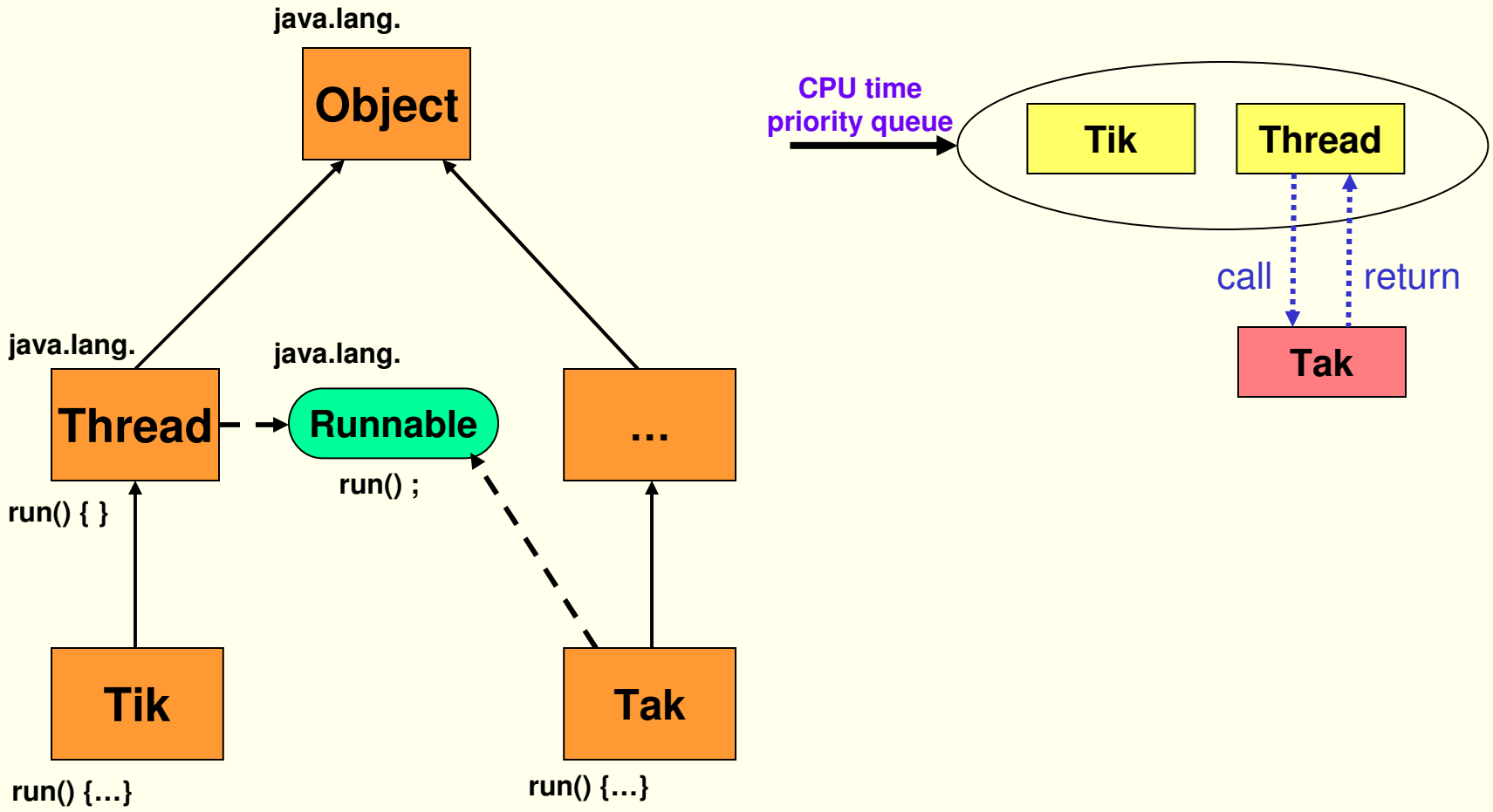
```
class Tak extends HodinkySVodotryskem implements Runnable {
    public void run( ) {                          // přepsaná metoda Runnable
        try {
            while( true ) {
                System.out.println( " Tak " );
                Thread.sleep( 1000 );             // static method
            }
        }
        catch( InterruptedException ex ) {  }
    }
}
```

spuštění zkráceně: `new Thread( new Tak( ) ). start( );`

spuštění rozepsaně:
```
Runnable r = new Tak( );
new Thread( r ). start( );
```

Tak není Thread, avšak má metodu run, kterou zavolá nepřepsaná metoda Threadu, neboť Tak je její target.

Tik versus Tak



Interakce vláken

Vlákna mohou běžet na sobě nezávisle, mohou se vzájemně hledat, testovat, [ne]koordinovaně ovlivňovat anebo mohou spolupracovat.

Metody pro:

hledání: currentThread( ), activeCount( ), getThreadGroup( ), ...

testování: isAlive( ), isDaemon( ), interrupted( ), isInterrupted( ), ...

ovlivňování: start( ), nastavení proměnných, interrupt( ), setPriority( ), ...

navazování: join( )

spolupráci: wait( ), notify( ), notifyAll( )
tyto metody vyžadují synchronizaci pomocí **synchronized** .

Interupt

Metoda t.interrupt( ) sama nezaručuje zastavení vlákna t - záleží na stavu vlákna t.

- **running : žádná vyjímka se nevyhodí, jen mu nastaví jeho příznak přerušení (flag). Je na tomto aby si svůj flag evetuálně zkontroloval a něco udělal. Pokud později přejde do stavu waiting nebo sleeping, předtím než metoda interrupted( ) příznak shodí, vyhodí se InterruptedException a příznak se shodí.**
- **waiting/sleeping : Vlákno přejde do runnable stavu a jakmile poběží vyhodí se InterruptedException. Příznak přerušení není nastaven.**
- **runnable : nic se neděje do té doby než přejde do running stavu – dále viz první bod.**

Join

Metoda t.join(timeout) blokuje běžné vlákno pokud je vlákno t živé, nejdéle však po dobu timeout. Je-li timeout == 0 je doba nekonečná.

Metoda t.join( ) je ekvivalentní t.join(0). To umožňuje běžnému vláknu vyčkat a pak navázat na výsledky činnosti dokončení práce vlákna t. Neodstartované vlákno t je neživé a tedy neblokuje běžné.

Na ukončení vlákna může čekat více vláken.

Vlákna se nijak vzájemně nenavazují (tj. nesloučí se v nějaké jediné - každé samostatně doběhne), navazuje se jejich práce.

Pilný strýc a líný synovec

Vlákno-synovec lajdá (je blokován) do doby, než vlákno-strýc dokončí svůj úkol (vydělat sto milionů) - čili až zemře. Dědictví si však dlouho neužije. Synovec může zjistit strýcův stav metodou isAlive( ).
Je-li nedočkavý, zavolá metodu join(timeout) - leč získá jen část peněz.

```
public class Nephew  {
   static Uncle  u = new Uncle( );
   public static void main( String[ ] args )  throws InterruptedException {
      u.join( );                                                    //  blocked
      System.out.println( u.money );                                //  heritage
    }
}

class Uncle extends Thread {
    int money = 0;
    public Uncle( ) {  this.start( ); }
    public void run( ) {  while ( ( ++money )  < 100000000 );  }    //  work
}
```

Nesynchronizace v hanebné bance

```
public class Test {
  public static void main( String[ ] args ) {
      new Clerk( ).start( );
      new Clerk( ).start( );
      for ( int i=1; true; i++ )  System.out.println( i+ ".:" + Accounts.dump( ) ) ;
  }
}

class Accounts {
   static long a1=0 , a2 = 1000;
   static void move( int x ) { a1 -= x; a2 += x; }                    // zdroj problému
   static String dump( ) {  return a1+ " + " +a2+ " = " +(a1+a2); }
}
class Clerk extends Thread {
   public void run( ) {
       while ( true ) Accounts.move( ( int ) ( ( Math.random( ) - 0.5 ) *100 ) );
    }
}
```

Synchronizace

Synchronizace brání vstupu více vláken do tzv. kritických sekcí, aby jejich nekoordinovaná činnost nezpůsobila chaos v hodnotách atributů.
(Obdobou kritické sekce je autoblok na železnici.)

Klíčové slovo synchronized u metody či bloku definuje kritickou sekci.
Kritických sekcí může být více a mohou být i do sebe vložené.
Mají být co nejkratší, aby nesnižovaly průchodnost programu.

Každý objekt má jediný zámek (intrinsic lock, monitor lock či též monitor).
Vlákno, které zámek získá, vstoupí do kritické sekce a zamkne objekt,
po jejím opuštění se zámek vrátí objektu, čímž se objekt odemkne.
Vlákno může získat více zámků od různých objektů.

Vlákno bez patřičného zámku ke kritické sekci do ní vstoupit nemůže a je přeřazeno ze stavu running do "čekárny" lock poolu.

Vrácený zámek JVM přidělí nedeterministicky některému příslušnému vláknu z lock poolu a přeřadí ho do stavu runnable.

Synchronizace

```
class X {
  ...  synchronized ... method1 ( ... )  {      // Synchronizovaná metoda
        ...
    }

  ...  method2 ( ... ) {                  // Nesynchronizovaná metoda
        ...
        synchronized( o1 ) {              // Synchronizované bloky
            synchronized( o2 ) {          // o1, o2 jsou reference k nějakým objektům
               ...                        //  nebo this nebo Z.class anebo k poli
            }
        }
        ...
    }
}
```

Atributy jejichž hodnoty se mění mají být private.

Statické synchronizované metody využívají zámek třídy.

Konstruktory a inicializátory provádí právě jedno vlákno.

Synchonizace v solidní bance

Solidní banka zjedná nápravu takto:

```
class Accounts {
   static private long a1=0, a2 = 1000;
   static synchronized void move( int x ) { a1 -= x; a2 += x; }          //  OK
   static String dump( ) {  return a1+ " + " +a2+ " = " +(a1+a2); }
}
```

anebo takto:

```
class Accounts {
   static private long a1=0, a2 = 1000;
   static void move( int x ) {
      synchronized ( Accounts.class )  {  a1 -= x; a2 += x;  }          //  OK
    }
    static String dump( ) {  return a1+ " + " +a2+ " = " +(a1+a2); }
}
```

Jak patrno v obou případech se (nevhodně) využívá zámek třídy Accounts.

Spolupráce

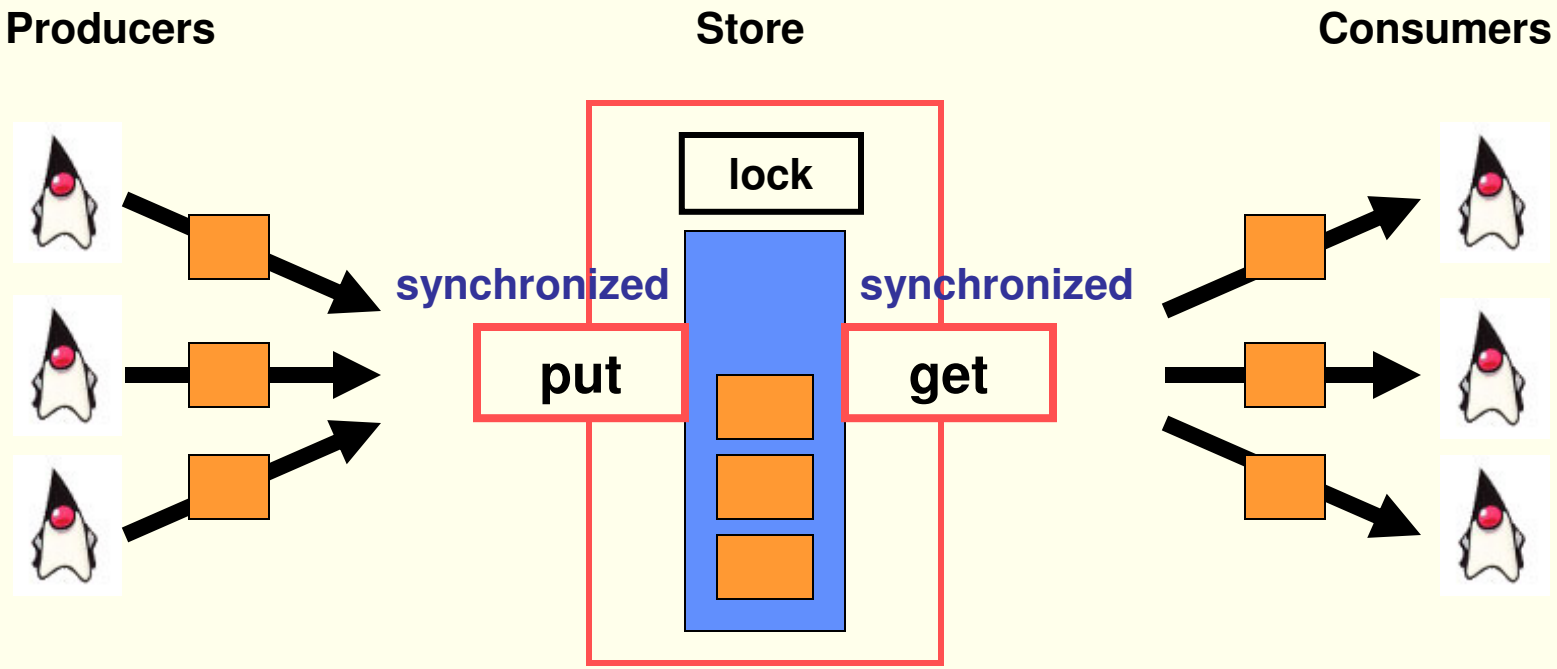
Vlákna spolupracující na nějakém objektu, musejí být koordinována tak, aby nepatřičně nezasahovala do společně přístupných dat.

Příkladem je PC-problém (Producer – Consumer):

- **Producenti se snaží dodat nějaké objekty do společného skladu.**
- **Konzumenti se snaží objekty ze společného skladu odebrat.**
- **Sklad má určitou kapacitu a je-li:**
  - **plný, producenti čekají na volné místo.**
  - **prázdný, konzumenti čekají na dodávku.**
- **Žádný objekt se nesmí ztratit.**
- **Tentýž objekt nesmí být odebrán dvakrát.**
- **Producenti ani konzumenti se navzájem neznají**
- **Producenti a konzumenti musejí být upozorňováni o změnách stavu skladu.**

Sklad musí být dobře střežen: důležité proměnné musejí být private.
Přístup k nim je možný jen přes synchronizované metody.
Přístup do skladu má vždy pouze jeden producent anebo konzument.

PC - problém


Producers
Store
Consumers
lock
synchronized
synchronized
put
get

Metody wait a notify

Koordinovaný přístup ke sdíleným datům se zajišťuje jednak všeobecnými finálními metodami wait a notify a jednak čekárnami wait pool a lock pool. Volat wait a notify může jen vlákno vlastnící zámek k objektu, tj. jen uvnitř kritické sekce, jinak dojde k IllegalMonitorStateException.

- **Čekáním vlákna ve wait poolu na změnu:**
  - **wait( ) - vlákno čeká až ...**
  - **wait(long timeout) – čeká až ... , leč čeká nejdéle zadanou dobu.**

  Obě metody wait odemykají objekt a vracejí zámek.
  Vlákno se převede z wait do lock poolu též zavoláním metody interrupt( ), přičemž se vyhodí výjimka InterruptedException.

- **Upozorněním jiného vlákna, že došlo ke změně podmínek.**
  Vlákno (či vlákna) se z wait do lock poolu se převede metodou:
  - **notify( ) - JVM nedeterministicky převede jedno vlákno.**
  - **notifyAll( ) - JVM převede všechna vlákna.**

Vlákna v lock poolu čekají na přidělení zámků k příslušným objektům.

Producer Consumer Problem

```
public class PCProblem {

   public static void main(String[ ] args) throws Exception {
      Box  b = new Box( );

      Producer p = new Producer(b);
      p.start( );

      Consumer c = new Consumer(b);
      c.start( );

      p.join( );
      c.join( );
      b.print( "END" );
  }

}
```

Producer Consumer Problem

```
class Box {                                          //   Box for one object  only
   private Object x;
   public void print( String r ) {  System.out.println( r+"  Box has "+x ); }
   synchronized void put( Object z ) {               //   z must not be null
      while ( x ! = null )   try { wait( ); }  catch ( InterruptedException ex ) { }
      notifyAll( );
      x = z;
      print( "PUT" );
   }
  synchronized Object get( ) {
      while ( x == null ) try { wait( ); } catch ( InterruptedException ex ) { }
      notifyAll( );
      Object z = x;    x=null;
      print( "GET" );
      return z;
  }
}
```

Producer Consumer Problem

```
class Producer extends Thread {

   Box b;

   Producer( Box b ) { this.b=b; }

   public void run( ) {
      for ( int i = 0; i < 10; i++ ) {
         b.put( new Integer( i ) );
         try {  sleep( ( int )( Math.random( )*1000 ) ); }
         catch ( InterruptedException ex ) { }
      }
   }

}
```

Producer Consumer Problem

```
class Consumer extends Thread {

   Box b;

   Consumer( Box b ) { this.b=b; }

   public void run( ) {
      for ( int i = 0; i < 10; i++ ) {
         System.out.println( ( Integer ) b.get( ) );
         try {  sleep( ( int ) ( Math.random( )*1000 ) ); }
         catch ( InterruptedException ex ) { }
      }
   }

}
```

Uvíznutí (deadlock)

Dva chudí lešetínští kováři mají jedno společné nářadí: kladivo a kleště. Nedohodnou-li se, dojde časem k zlobně umrtvujícímu nicnedělání, tzv. deadlocku čili smrtelnému zaklesnutí.

Přihodí se to takto:

- **První uchopil kladivo a potřebuje ještě kleště.**
- **Mezitím však druhý uchopil kleště a čeká až bude kladivo volné.**

Nikdo to za ně nevyřeší a tak oba čekají a čekají ... čímž živnosti uvíznou a oba ještě více zchudnou - a šafářovic Andulka se nedočká.

Přitom stačí rozumná dohoda - budeš-li potřebovat oba nástroje:

- **Nejdříve uchop kladivo a teprve pak sháněj kleště.**
- **Pracuj.**
- **Pak nejdříve pusť kleště, potom kladivo.**

Ta zaručuje momentálnímu držiteli kladiva, že kleště budou časem volné. Kovářů může být ve městě i více.

Proti uvíznutí

Programátor znemožní uvíznutí takto:

```
synchronized ( hammer ) {
    synchronized ( tongs ) {
        ...     //  work
    }
}
```

Více nářadí lze vložit do synchronizované kolekce či do pole.
(Pole je svou podstatou synchronizovaný objekt.)

```
private Collection tools = new Vector( );
tools.add( hammer );  tools.add( tongs );  tools.add(...);   //  fill tools
synchronized ( tools ) {
    Object x = tools.get(...);
    ...     //  work
}
```

Skupiny vláken

JVM odstartovaná vlákna umístí jako listy stromu, jehož kořen a vnitřní uzly jsou skupiny vláken typu ThreadGroup.

Mrtvá vlákna z tohoto stromu JVM odstraní.

Třída ThreadGroup nabízí metody:

- **ThreadGroup(ThreadGroup parent, String name) – konstruktor.**
- **int activeCount( ) – odhad počtu vláken ve skupině.**
- **int activeGroupCount( ) – odhad počtu skupin ve skupině.**
- **int enumerate(Thread[ ]) – výčet vláken ve skupině.**
- **int enumerate(ThreadGroup[ ]) – výčet skupin vláken ve skupině.**
- **ThreadGroup getParent( ) – reference k nadskupině.**
- **boolean parentOf(ThreadGroup g) – test příslušnosti k podstromu g.**
- **void list( ) – výpis skupiny.**
- **void interrupt( ) – přerušení všech vláken podstromu.**
- **boolean isDaemon( ) – test zda je skupina démonická.**
- **void setDaemon(boolean on) – démonizace všech vláken.**
- **void setMaxPriority(int pri) – nastavení pro další členy skupiny.**
- **void destroy( ) – zrušení prázdného podstromu.**

Časové spouštění úloh

umožňují třídy java.util.Timer a abstraktní java.util.TimerTask.

- **Timer( ) – konstruktor.**
- **Timer(boolean isDaemon) – konstruktor pro démona.**

- **void schedule(...) – naplánuje spuštění úkolu.**
- **void scheduleAtFixedRate(...) – spouštění s pevným intervalem.**

  - tyto metody mají parametry:

  TimerTask task – naplánovaný úkol.

  Date firstTime – datum a čas prvního spuštění.

  long delay – zpoždění pro první spuštění.

  long period – perioda dalších spuštění.

- **void cancel( ) – ukončí činnost objektu i naplánované úkoly.**

Uplynul-li již moment spuštění, spustí se ihned.

Časové spouštění úloh

Abstraktní třída java.util.TimerTask má:

- **protected TimerTask( ) – konstruktor.**
- **public void cancel( ) – ukončí naplánovaný úkol.**
- **public long scheduledExecutionTime( ) – čas posledního spuštění.**
- **public abstract void run( ) – definice činnosti.**

```
Timer t = new Timer( );
t.schedule( new TTA( ), 10000 );
t.scheduleAtFixedRate( new TTB( ), 5000, 500 );

class TTA extends TimerTask{
  public void run( ) { System.out.println( “A” ); }
}
class TTB extends TimerTask{
  public void run( ) { System.out.println( “B” ); }
}
```

Jak JVM spouští aplikaci

Při spuštění java T aa bb ccc systémové vlákno vytvoří ThreadGroup se jménem main a v ní Thread se jménem main s prioritou 5 odstartuje ho.

Vlákno main projde postupně všemi statickými inicializátory (nedojde-li k neodchycené výjimce – a tudíž k ExceptionInInitializerError a ukončení chodu). Následně projde do metody T.main.

```
class ... extends Thread {
   public void run( ) {
      try {
         // call all static initializers
         // call T.main( ... )
      } catch ( Throwable ex ) { ex.printStackTrace(); }
    }
}
```