Багатопоточне програмування

Лекція 7

доц. кафедри Інформатики
Сінельнікова Т.Ф.

ХНУРЕ, Кафедра Інформатики

informatika@kture.kharkov.ua

- Поняття багатопоточності
- Поточна модель Java
- Синхронізація
- Клас Thread и інтерфейс Runnable
- Головний потік
- Реалізація інтерфейсу Runnable
- Розширення Thread
- Вибір підходу
- Створення множинних потоків
- Пріоритети потоків
- Синхронізація
- Використання синхронізованих методів
- Оператор synchronized
- Міжпоточні зв'язки
- Блокування
- Призупинення, відновлення і зупинка потоків в Java
- Використання багатопоточності

Поняття багатопоточності

- Java забезпечує вбудовану підтримку для багатопотокового програмування.
- Багаторівнева програма містить дві і більше частин, які можуть виконуватися одночасно, конкуруючи один з одним.
- Кожна частина такої програми називається потоком, а кожен потік визначає окремий шлях виконання (в послідовності операторів програми).
- Нить - це спеціалізована форма багатозадачності.

Поняття багатопоточності

- Існують дві різні форми багатозадачності. Одна заснована на процесах, а інша - на потоках.
- Процес - це, по суті, що виконується програма.
  Заснована на процесах багатозадачність - це властивість, яка дозволяє вашому комп'ютеру виконувати декілька програм одночасно. Така багатозадачність дає, наприклад, можливість виконувати компілятор Java одночасно з використанням текстового редактора. У багатозадачності, заснованої на процесах, найдрібнішої одиницею диспетчеризації планувальника є програма.
- У багатозадачному середовищі, заснованої на потоках, найдрібнішої одиницею диспетчеризації є потік. Це означає, що окрема програма може виконувати декілька завдань одночасно. Наприклад, текстовий редактор може форматувати текст одночасно з печаткою документа, оскільки ці дві дії виконуються двома окремими потоками.
- Таким чином, багатозадачність, заснована на процесах, має справу з усією картиною, а потокова багатозадачність обробляє деталі.

Поняття багатопоточності

- Багатозадачні потоки вимагають менших накладних витрат у порівнянні з багатозадачними процесами. Процеси - це великовагові завдання, яким потрібні окремі адресні простори. Зв'язки між процесами обмежені і коштують не дешево. Перемикання контексту від одного процесу до іншого також досить дороге завдання.
- З іншого боку, потоки досить легковагі. Вони спільно використовують один і той же адресний простір і кооперативно оперують з одним і тим же великоваговим процесом, міжпоточної зв'язку недорогі, а перемикання контексту від одного потоку до іншого має низьку вартість.

поняття багатопоточності

- Нить дає можливість писати дуже ефективні програми, які максимально використовують CPU, бо час його простою можна звести до мінімуму.
- Це особливо важливо для інтерактивної мережевий середовища, в якому працює Java, тому що час простою є загальним.
- Швидкість передачі даних по мережі набагато менше, ніж швидкість, з якою комп'ютер може їх обробляти.
- У традиційній однопоточному середовищі ваша програма повинна чекати закінчення кожної свого завдання, перш ніж вона зможе перейти до наступного (навіть при тому, що більшу частину часу CPU простоює).
- Нить дозволяє отримати доступ до цього часу простою і краще його використовувати.

Поточна модель Java

- Виконавча система Java багато в чому залежить від потоків, і всі бібліотеки класів розроблені з урахуванням багатопоточності.
- Java використовує потоки для забезпечення асинхронності у всій середовищі.
- Цінність многопоточной середовища краще розуміється по контрасту з її аналогом.
- Однопоточні системи використовують підхід, званий циклом подій з опитуванням (event loop with polling). У цій моделі, єдиний потік управління виконується в нескінченному циклі, опитуючи єдину чергу подій, щоб вирішити, що робити далі. Як тільки цей механізм опитування повертає сигнал готовності мережевого файлу готовий для читання, цикл подій передає управління відповідного обробникові подій. До повернення з цього обробника в системі нічого більше трапитися не може.

Поточна модель Java

- Вигода від багатопоточності Java полягає в тому, що усувається механізм "головний цикл / опитування". Один потік може робити паузу без зупинки інших частин програми. Наприклад, час простою, що утворюється, коли потік читає дані з мережі або чекає введення користувача, може використовуватися в іншому місці.

- Нить дозволяє циклам мультиплікації не діяти протягом секунди між кожним фреймом без примусу робити паузу цілої системи. Коли потоки блокуються в Java-програмі, паузу "тримає" тільки один потік - той, який блокований. Інші продовжують виконуватися.

Поточна модель Java

- Потоки існують в декількох станах.
- Потік може бути в стані виконання.
- Може перебувати в стані готовності до виконання, як тільки він отримає час CPU.
- Виконується потік може бути припинений, що тимчасово пригальмовує його дію.
- Потім призупинений потік може бути продовжений (відновлений) з того місця, де він був зупинений.
- Потік може бути блокований в очікуванні ресурсу. В будь-який момент виконання потоку може бути завершено, що негайно зупиняє його виконання.

Пріоритети потоків

- Java призначає кожному потоку пріоритет, який визначає порядок обробки цього потоку щодо інших потоків.
- Пріоритети потоків - це цілі числа, які визначають відносний пріоритет одного потоку над іншим.
- Високопріоритетний потік не виконується швидше, ніж низькопріоритетним.
- Пріоритет потоку використовується для того, щоб вирішити, коли перемикатися від одного виконується потоку до наступного. Це називається перемиканням контексту.

Пріоритети потоків

- Правила, які визначають, коли перемикання контексту має місце.

- Потік може добровільно відмовитися від управління. Це робиться явно, переходом в режим очікування або блокуванням на виклик введенні / виводі. У цьому сценарії проглядаються всі потоки, і CPU передається самому високопріоритетний потоку, який готовий до виконання.
  Потік може бути припинений більш пріоритетним потоком.
- В цьому випадку займає процесор низькопріоритетним потік тимчасово зупиняється (незалежно від того, що він робить) потоком з більш високим пріоритетом. Даний механізм називається попереджуючої багатозадачністю (preemptive multitasking).

синхронізація

- Оскільки багатопоточність забезпечує асинхронне поведінка ваших програм, повинен існувати спосіб домогтися синхронності, коли в цьому виникає необхідність.
- Наприклад, якщо ви хочете, щоб два потоки взаємодіяли і спільно використовували складну структуру даних типу зв'язного списку, потрібно якимось чином гарантувати відсутність між ними конфліктів.
- Ви повинні втримати один потік від запису даних, поки інший потік знаходиться в процесі їх читання.Для цієї мети Java експлуатує модель синхронізації процесів - монітор.
- Монітор - це механізм управління зв'язком між процесами. Ви можете представляти монітор, як дуже маленький блок, який містить тільки один потік.
- Як тільки потік входить в монітор, всі інші потоки повинні чекати, поки даний не вийде з монітора. Таким чином, монітор можна використовувати для захисту спільно використовуваного (розділяється) ресурсу від керування декількома потоками одночасно.

Синхронізація

- Більшість багатопоточних систем створює монітори як об'єкти, які ваша програма повинна явно отримати і використовувати.
- В Java-системі немає класу з ім'ям Monitor. Замість цього, кожен об'єкт має свій власний неявний монітор, який вводиться автоматично при виклику одного з методів об'єкта.
- Як тільки потік виявляється всередині синхронізованого методу, ніякий інший потік не може викликати інший синхронізований метод того ж об'єкта.

Передача повідомлень

- Після того як ви розділите свою програму на окремі потоки, потрібно визначити, як вони будуть взаємодіяти один з одним.
- Java забезпечує ясний, дешевий шлях для взаємного спілкування двох (чи кількох) потоків через виклики зумовлених методів, якими володіють всі об'єкти.
- Система передачі повідомлень Java дозволяє потоку увійти в синхронізований метод на об'єкті і потім чекати там, поки деякий інший потік явно не повідомить його про вихід.

Клас Thread и інтерфейс Runnable

- Багаторівнева система Java побудована на класі Thread, його методах і пов'язаному з ним інтерфейсі Runnable.
- Thread інкапсулює потік виконання. Так як ви не можете безпосередньо звертатися до внутрішнього стану потоку виконання, то будете мати з ним справу через його повноважного представника - примірник (об'єкт) класу Thread, який його породив.
- Щоб створити новий потік, ваша програма повинна буде або розширювати клас Thread або реалізовувати інтерфейс Runnable.

Клас Thread и інтерфейс Runnable

деякі *методи класу Thread*

Метод	Значення
GetName ()	Отримати ім'я потоку
GetPriority ()	Отримати пріоритет потоку
IsAlive ()	Визначити, чи виконується ще потік
Join ()	Чекати завершення потоку
Run ()	Вказати точку входу в потік
Sleep ()	Призупинити потік на певний період часу
Start ()	Запустити потік за допомогою виклику його методу run ()

Головний потік

- Коли Java-програма запускається, один потік починає виконуватися негайно.
  Він зазвичай називається головним потоком.
- Головний потік важливий з двох причин:

- Це потік, з якого будуть породжені всі інші "дочірні" потоки.
- Це повинен бути останній потік, в якому закінчується виконання. Коли головний потік зупиняється, програма завершується.

Головний потік

- Хоча головний потік створюється автоматично після запуску програми, він може управлятися через Thread-об'єкт. Для організації управління потрібно отримати посилання на нього, викликаючи метод Currentrhread (), який є public static членом класу Thread. Ось його загальна форма:
  static Thread currentThread ()
- Цей метод повертає посилання на потік, в якому він викликається. Як тільки ви отримуєте посилання на головний потік, то можете керувати ним точно так само, як будь-яким іншим потоком.

Головний потік

```
// Управління головним потоком.,
class CurrentThreadDemo {
public static void main(String args[])
{
Thread t = Thread.currentThread();
System.out.println("Поточний потік:
    "+t);
// змінити ім'я потокуt.setName("My
    Thread");
System.out.println("Після зміни імені:
    "+t);
try
{
for(int n = 5; n > 0; n--)
{ System.out.println(n);
    Thread.sleep(1000); } }
catch (InterruptedException e)
{
System.out.println("Головний потік
    завершений"); } } }
```

- Вивод, згенерований цією програмою:

  Поточний потік: Thread [main, 5, main]
  Після зміни імені: Thread [My Thread, 5, main]

  5
  4
  3
  2
  1

Головний потік

- Зверніть увагу на висновки, що використовують t як аргумент printin ().
- За замовчуванням ім'я головного потоку - main. Його пріоритет дорівнює 5, що теж є значенням за замовчуванням і останній ідентифікатор main - ім'я групи потоків, якій належить даний потік.
- Група потоків - структура даних, яка контролює стан сукупності потоків в цілому. Цей процес управляється спеціальною виконавчої (run-time) середовищем і детально тут не обговорюється. Після зміни імені потоку мінлива t знову виводиться на екран.

Головний потік

- Методи класу Thread, які використовуються в програмі.
- Метод sleep () змушує потік, з якого він викликається, призупинити виконання на вказане (в аргументі виклику) число мілісекунд. Його загальна форма має вигляд:
  static void sleep (long milliseconds) throws InterruptedException
- Число мілісекунд інтервалу призупинення визначається в параметрі milliseconds. Крім того, даний метод може викинути виключення типу InterruptedException.
- Метод sleep () має другу форму, яка дозволяє визначати період припинення в частках мілісекунд і наносекунд:
  static void sleep (long milliseconds, int nanoseconds) throws InterruptedException
- Ця друга форма корисна тільки в середовищах, де допускаються короткі періоди часу, вимірювані в наносекундах.

створення потоку

- У самому загальному випадку для створення потоку будують об'єкт типу Thread. В Java це можна виконати двома способами:

- реалізувати інтерфейс Runnable;
- розширити клас Thread, визначивши його підклас.

Реалізація інтерфейсу Runnable

- В Runnable визначений деякий абстрактний (без тіла) модуль виконуваного коду. Створювати потік можна на будь-якому об'єкті, який реалізує інтерфейс Runnable. Для реалізації Runnabie в класі потрібно визначити тільки один метод з ім'ям run ().
  public void run ()
- Усередині run () потрібно визначити код, який утворює новий потік.
- run () може викликати інші методи, використовувати інші класи і оголошувати змінні точно так само, як це робить головний (main) потік.
- Єдина відмінність полягає в тому, що run про встановлює в даній програмі точку входу для запуску іншого, конкуруючого потоку виконання.
- Цей потік завершить своє виконання при поверненні з run ().

Реалізація інтерфейсу Runnable

- Після створення класу, який реалізує Runnable, потрібно організувати об'єкт типу Thread всередині цього класу.
- У класі Thread визначено кілька конструкторів.
- Ми будемо використовувати конструктор наступної форми:
  Thread (Runnable thxeadOb, String threadName)
- Тут threadOb - примірник (об'єкт) класу, який реалізує інтерфейс Runnable. Він визначає, де почнеться виконання нового потоку.
  Ім'я нового потоку визначає параметр threadName.
- Після того як новий потік створений, він не почне виконуватися, поки ви не викликаєте його методом start (), який оголошений в Thread.
  void start ()

Реалізація інтерфейсу Runnable

```
/ / Створення другого потоку,
    class NewThread implements Runnable
    {Thread t;
    NewThread () {
    / / Створити новий, другий потік.
    t = new Threadfthis ("Demo Thread");
    System.out.println ("Дочірній потік:" + t);
    t.startO; / / стартувати потік
    }
    / / Це точка входу в другій потік, public void run () {
    try {
    for (int i = 5; i> 0; i -
      {System.out.println ("Дочірній Потік:" + i);
    Thread.sleep (500);
    }}
    catch (InterruptedException e) {
    System.out.println ("Переривання дочірнього
    потоку.");}
    System.out.println ("Завершення дочірнього потоку.");}}
    class ThreadDemo {
    public static void main (String args [])
    {new NewThreadO; / / створити новий потік
    try {
    for (int i = 5; i> 0; i -) {
    System.out.println ("Головний потік:" + i);
    Thread.sleep (1000);}}
    catch (InterruptedException e) {
    System.out.println ("Переривання головного потоку.");}
    System.out.println ("Завершення головного потоку.");}}
```

- Вивод цієї програми наступний:
  Дочірній потік: Thread [Demo Thread, 5, main]
  Головний потік: 5
  Дочірній Потік 5
  Дочірній Потік 4
  Головний потік: 4
  Дочірній Потік 3
  Дочірній Потік 2
  Головний потік: 3
  Дочірній Потік 1
  Завершення дочірнього потоку.
  Головний потік: 2
  Головний потік: 1
  Завершення головного потоку.

Розширення Thread

- Для генерації потоку другим способом необхідно створити новий клас, який розширює Thread, а потім - примірник цього класу.
- Розширює клас повинен перевизначати метод run (), який є точкою входу для нового потоку.
- Він повинен також викликати start (), щоб почати виконання нового потоку.

Розширення Thread

```
/ / Створює другий потік розширенням
    класу
    / / Thread,
    class NewThread extends Thread {\
    NewThread () {
    / / Створити новий, другий потік
    super ("Demo Thread");
    System.out.println ("Дочірній потік:" +
    this);
    start (); / / стартувати потік
    }
    / / Це точка входу для другого потоку,
    public void run () (try {
    for (int i = 5; i> 0; i -)
      {
    System.out.println ("Дочірній потік:" + i);
    Thread.sleep (500);}}. catch
    (InterruptedException e)
    {
    System.out.println ("Переривання
    дочірнього потоку.");}
    System.out.println ("Завершення
    дочірнього потоку.");}}
```

```
class ExtendThread {
    public static void main (String args [])
    {
    new NewThread (); / / створити
    новий потік
    try {
    for (int i = 5; i> 0; i-) {
    System.out.println ("Головний
    потік:" + i);
    Thread.sleep (1000);}}
    catch (InterruptedException e)
    {
    System.out.println ("Переривання
    головного потоку.");}
    System.out.println ("Завершення
    головного потоку.");
    }
      }
```

Вибір підходу

- Який підхід краще?
- Клас Thread визначає кілька методів, які можуть бути перевизначені похідним класом. З них тільки один повинен бути перевизначений обов'язково - метод run (). Це, звичайно, той же самий метод, який потрібний для реалізації інтерфейсу Runnable.
- Класи повинні розширюватися тільки тоді, коли вони якимось чином поліпшуються або змінюються.
- Так, якщо ви не будете перевизначати ніякого іншого методу класу Thread, то, ймовірно, краще за все просто реалізувати інтерфейс Runnable безпосередньо.

Створення множинних потоків

- До сих пір використовували тільки два потоки - головний і один дочірній.
- Однак ваша програма може породжувати стільки потоків, скільки необхідно.
- Не забувайте використовувати метод sleep ().

Використання методів isAlive () і join ()

- Існують два способи визначення, закінчився чи потік.
- Один з них дозволяє викликати метод isAlive () на потоці. Цей метод визначено в Thread і його загальна форма виглядає так:
  final boolean isAlive ()
- Метод isAlive () повертає true, якщо потік, на якому він викликається - все ще виконується. В іншому випадку повертається false.

Використання методів isAlive () і join ()

- У той час як isAlive () корисний тільки іноді, частіше для очікування завершення потоку викликається метод join () наступного формату:
  final void join () throws InterruptedException
- Цей метод чекає завершення потоку, на якому він викликаний. Його ім'я походить з концепції перекладу потоку в стан очікування, поки вказаний потік не приєднає його.
- Додаткові форми join () дозволяють визначати максимальний час очікування завершення зазначеного потоку.

Пріоритети потоків

- Планувальник потоків використовує їх пріоритети для прийняття рішень про те, коли потрібно вирішувати виконання того чи іншого потоку.
- Теоретично високопріоритетні потоки одержують більше часу CPU, ніж низькопріоритетні.
- На практиці, однак, кількість часу CPU, яке потік отримує, часто залежить від декількох факторів крім його пріоритету. (Наприклад, відносна доступність часу CPU може залежати від того, як операційна система реалізує багатозадачний режим.)
- Високопріоритетний потік може також попереджувати низькопріоритетним (тобто перехоплювати у нього управління процесором).
- Скажімо, коли низькопріоритетним потік виконується, а високопріоритетний потік поновлюється (від очікування на введенні / виводі, наприклад), високопріоритетний потік буде попереджати низькопріоритетним.

Пріоритети потоків

- Теоретично, потоки рівного пріоритету повинні отримати рівний доступ до CPU.
- Для безпеки потоки, які спільно використовують один і той же пріоритет, повинні час від часу поступатися один одному управління.
- Це гарантує, що всі потоки мають шанс виконатися під непріоритетної операційною системою.
- Практично, навіть в непріоритетних середовищах, більшість потоків все ще отримують шанс виконуватися, тому що більшість з них неминуче стикається з деякими блокуючими ситуаціями, типу очікування введення / виведення.
- Коли це трапляється, блокований потік припиняється, а інші можуть продовжуватися.

Пріоритети потоків

- Для установки пріоритету потоку використовуйте метод setPriority (), який є членом класу Thread. Ось його загальна форма:
  final void setPriority (int level)
- де level визначає нову установку пріоритету для викликає потоку.
- Значення параметра level повинно бути в межах діапазону min_priority і max_priority. В даний час ці значення рівні 1 і 10, відповідно.
- Щоб повернути потоку пріоритет, заданий за замовчуванням, визначте norm_priority, який в даний час дорівнює 5.
- Ці пріоритети визначені в Thread як final-змінні.

Пріоритети потоків

- Ви можете отримати поточну установку пріоритету, викликаючи метод getPriority () класу Thread, чий формат має наступний вигляд:
  final int getPriority ()
- Реалізації Java можуть мати радикально різну поведінку, коли вони переходять до планування.

Синхронізація

- Коли кілька потоків потребують доступу до ресурсу, їм необхідний деякий спосіб гарантії того, що ресурс буде використовуватися одночасно тільки одним потоком.
- Процес, за допомогою якого це досягається, називається синхронізацією.
  Ключем до синхронізації є концепція монітора (також звана семафором).
- Монітор - це об'єкт, який використовується для взаємовиключної блокування (mutually exclusive lock), або mutex.

Синхронізація

- Тільки один потік може мати власний монітор в заданий момент.
- Коли потік отримує блокування, кажуть, що він увійшов в монітор. Всі інші потоки намагаються вводити блокований монітор, будуть припинені, поки перший не вийшов з монітора.
- Кажуть, що інші потоки очікують монітор.
- При бажанні потік, який володіє монітором, може повторно вводити той же самий монітор.
- Синхронізувати код можна двома способами. Обидва використовують ключове слово synchronized.

Використання синхронізованих методів

- Синхронізація в Java проста тому, що кожен об'єкт має свій власний неявний пов'язаний з ним монітор.
- Щоб ввести монітор об'єкта, просто викликають метод, який був модифікований ключовим словом synchronized.
- Поки потік знаходиться всередині синхронізованого методу, всі інші потоки, які намагаються викликати його (або будь-який інший синхронізований метод) на тому ж самому примірнику, повинні чекати.
- Щоб вийти з монітора і залишити управління об'єктом наступного очікує потоку, власник монітора просто повертається з синхронізованого методу.

Оператор synchronized

- Хоча визначення синхронізованих методів всередині класів - це прості та ефективні засоби досягнення синхронізації, вони не будуть працювати у всіх випадках.
- Наприклад, ви хочете синхронізувати доступ до об'єктів класу, який не був розроблений для багатопотокового доступу.
- Тобто клас не використовує синхронізовані методи.
- Крім того, цей клас був створений не вами, а третьою особою, і ви не маєте доступу до вихідного коду.
- Таким чином, ви не можете додавати специфікатор synchronized до відповідних методів в класі.
- Рішення даної проблеми дуже просто. Потрібно помістити виклики методів, визначених цим класом всередину синхронізованого блоку.

Оператор synchronized

- Ось загальна форма оператора synchronized:
  synchronized (object) {
  / / Оператори для синхронізації
  }
  де object - посилання на об'єкт, який потрібно синхронізувати.
- Якщо потрібно синхронізувати одиночний оператор, то фігурні дужки можна опустити.
- Блок гарантує, що виклик методу, який є членом об'єкта object, відбувається тільки після того, як поточний потік успішно запровадив монітор об'єкта.

Міжпоточні зв'язки

- Ви можете досягти більш тонкого рівня управління через зв'язок між процесами. Нить замінює програмування циклу подій, розподілом завдань на дискретні і логічні модулі.
- Потоки також забезпечують і друга перевага - вони скасовують опитування.
- Опитування зазвичай реалізується циклом, який використовується для повторюваної перевірки деякої умови.
- Як тільки умова стає істинним, робиться відповідна дія. На цьому втрачається час CPU.
- Наприклад, розглянемо класичну проблему організації черги, де один потік виробляє деякі дані, а інший - їх споживає.
- Припустимо, що, перш ніж генерувати більшу кількість даних, виробник повинен чекати, поки споживач не закінчить свою роботу.
- У системі ж опитування, споживач витрачав би даремно багато циклів CPU на очікування кінця роботи виробника. Як тільки виробник закінчив свою роботу, він змушений почати опитування, витрачаючи багато циклів CPU на очікування кінця роботи споживача. Ясно, що така ситуація небажана.
- Щоб усунути опитування, Java містить витончений механізм межпроцессовой зв'язку через методи wait (), notify () і notifyAll (). Вони реалізовані як final-методи в класі object, тому доступні всім класам

Міжпоточні зв'язки

- wait () повідомляє зухвалому потоку, що потрібно поступитися монітор і переходити в режим очікування ("сплячки"), поки деякий інший потік не введе той же монітор і не викличе notify ();
- notify () "пробуджує" перший потік (який викликав wait ()) на тому ж самому об'єкті;
- notifyAll () пробуджує все потоки, які викликали wait () на тому ж самому об'єкті. Першим буде виконуватися самий високопріоритетний потік.

- Ці методи оголошуються в класі Object в такій формі:
  final void wait () throws XnterruptedException
  final void notify ()
  final void notifyAll ()

Блокування

- Спеціальний тип помилки, яку вам потрібно уникати і яка спеціально ставиться до багатозадачності, це - (взаємне) блокування.
- Вона відбувається, коли два потоки мають циклічну залежність від пари синхронізованих об'єктів.
- Наприклад, припустимо, що один потік вводить монітор в об'єкт x, а інший потік вводить монітор в об'єкт y. Якщо потік в x пробує викликати будь синхронізований метод об'єкту y, це призведе до блокування, як і очікується.
- Однак якщо потік в y, в свою чергу, пробує викликати будь синхронізований метод об'єкту x, то він буде завжди чекати, тому що для отримання доступу до x, він був би повинен зняти свою власну блокування з y, щоб перший потік міг завершитися .

Блокування

- Взаємоблокування - важка помилка для налагодження з двох причин:

  - Взагалі кажучи, вона відбувається дуже рідко, коли інтервали тимчасового квантування двох потоків знаходяться в певному співвідношенні.
  - Вона може включати більше двох потоків і синхронізованих об'єктів. (Тобто блокування може відбуватися через більш хитромудру послідовність подій, що тільки що описано.)

Призупинення, відновлення і зупинка потоків в Java

- Призупинення виконання потоку іноді корисна.
- Наприклад, окремі потоки можуть використовуватися, щоб відображати час дня. Якщо користувач не хоче бачити відображення годин, то їх потік може бути призупинено. У кожному разі припинення потоку - проста справа. Після припинення перезапуск потоку також не складний.
- Механізми припинення, зупинення і відновлення потоків різні для Java 2 і більш ранніх версій Java.

Призупинення, відновлення і зупинка потоків в Java

- До Java 2 для призупинення і перезапуску виконання потоку програма використовувала методи suspend () і resume (), які визначені в класі Thread. Вони мають таку форму:
  final void suspend ()
  final void resume ()
- Клас Thread також визначає метод з ім'ям stop (), який зупиняє потік. Його сигнатура має наступний вигляд:
  void stop ()
- Якщо потік був зупинений, то його не можна перезапускати за допомогою методу resume ().

Призупинення, відновлення і зупинка потоків в Java

- В Java 2 заборонено використовувати методи suspend (), resume () або stop () для управління потоком.
- Потік повинен бути спроектований так, щоб метод run () періодично перевіряв, чи повинен цей потік призупиняти, відновлювати або зупиняти своє власне виконання.
- Це, як правило, виконується застосуванням флажковой змінної, яка вказує стан виконання потоку.
- Поки прапорець встановлений на "виконання", метод run () повинен продовжувати дозволяти потоку виконуватися.
- Якщо ця змінна встановлена на "призупинити", потік повинен зробити паузу. Якщо вона встановлена на "стоп", потік повинен завершитися.

Використання багатопоточності

- Ключем до ефективного використання зазначеної підтримки є скоріше паралельне (одночасне), ніж послідовне мислення.
- Наприклад, коли ви маєте дві підсистеми усередині програми, які можуть виконуватися одночасно, організуйте їх індивідуальними потоками.
- З обережним використанням багатопоточності ви можете створювати дуже ефективні програми.
- Проте необхідні і застереження.
- Якщо ви створите занадто багато потоків, то, навпаки, погіршите ефективність програми.
- Пам'ятайте, що з контекстним перемиканням пов'язані деякі витрати.
- Якщо ви створюєте занадто багато потоків, більше часу CPU буде витрачено на зміни контекстів, ніж на виконання вашої програми.