­­МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”

­­­

Контрольна робота №1

з курсу «Об’єктно-орієнтоване програмування»

для студентів базового напрямку 6.08.04 "Комп’ютерні науки"

(заочна форма навчання)

Варіант 14

Виконав студент гр. КНз-2

Чалий Михайло

­­

Львів 2014

# Серіалізація об’єктів у мові Java

**Серіалізація** — процес перетворення будь-якої [структури даних](http://uk.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B8_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85) в послідовність [бітів](http://uk.wikipedia.org/wiki/%D0%91%D1%96%D1%82). Зворотною до операції серіалізації є операція десеріалізації - відновлення початкового стану структури даних з бітової послідовності.

Серіалізація використовується для передачі [об'єктів](http://uk.wikipedia.org/wiki/%D0%9E%D0%B1%27%D1%94%D0%BA%D1%82) по мережі і для збереження їх у [файли](http://uk.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB). Наприклад, потрібно створити розподілений [додаток](http://uk.wikipedia.org/wiki/%D0%94%D0%BE%D0%B4%D0%B0%D1%82%D0%BE%D0%BA), різні частини якого мають обмінюватися даними зі складною структурою. У такому випадку для типів даних, які передбачається передавати, пишеться код, який здійснює серіалізацію і десеріалізацію. Об'єкт заповнюється потрібними даними, потім викликається код серіалізації, в результаті виходить, наприклад, [XML](http://uk.wikipedia.org/wiki/XML)-документ. Результат серіалізації передається приймаючій стороні, наприклад, по [електронній пошті](http://uk.wikipedia.org/wiki/%D0%95%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D0%B0_%D0%BF%D0%BE%D1%88%D1%82%D0%B0) або [HTTP](http://uk.wikipedia.org/wiki/HTTP). Додаток-одержувач створює об'єкт того ж типу і викликає код десеріалізації, в результаті отримуючи об'єкт з тими ж даними, що були в об'єкті програми-відправника.

## Застосування

Серіалізація надає декілька корисних можливостей:

* метод реалізації зберігання об'єктів, який зручніший, ніж запис їх властивостей в текстовий файл на диск і повторна збірка об'єктів читанням файлів;
* метод здійснення віддалених викликів процедур, як, наприклад, у [SOAP](http://uk.wikipedia.org/wiki/SOAP);
* метод розповсюдження об'єктів, особливо в технологіях [компонентно-орієнтованого програмування](http://uk.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D1%96%D1%94%D0%BD%D1%82%D0%BE%D0%B2%D0%B0%D0%BD%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F), таких як [COM](http://uk.wikipedia.org/wiki/COM) і [CORBA](http://uk.wikipedia.org/wiki/CORBA);
* метод виявлення змін у даних, що змінюються з часом.

Для найефективнішого використання даних можливостей необхідно підтримувати незалежність від архітектури. Наприклад, необхідно мати можливість надійно відтворювати серіалізований [потік даних](http://uk.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D1%96%D0%BA_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85), незалежно від [порядку байтів](http://uk.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D1%8F%D0%B4%D0%BE%D0%BA_%D0%B1%D0%B0%D0%B9%D1%82%D1%96%D0%B2), що використовується в цій архітектурі. Це означає, що найбільш проста і швидка процедура прямого копіювання ділянки пам'яті, в якому розміщується структура даних, не може працювати надійно для всіх архітектур. Серіалізація структур даних в архітектурно-незалежний формат означає, що не повинно виникати проблем через різний порядок проходження байтів, механізмів розподілу пам'яті або відмінностей представлення структур даних в мовах програмування.

Будь-якій зі схем серіалізації властиво те, що кодування даних послідовно за визначенням, і вибірка будь-якої частини серіалізованої структури даних вимагає, щоб весь об'єкт був зчитаний від початку до кінця і був відновлений. У багатьох програмах така лінійність корисна, тому що дозволяє використовувати прості інтерфейси введення/виведення загального призначення для збереження і передачі стану об'єкта. У додатках, де важлива висока продуктивність, можливо буде доречніше використовувати складнішу, нелінійну організацію зберігання даних.

## Серіалізація в Java

Для запису/читання об’єктів використовуються потокові класи ObjectOutputStream або ObjectInputStream

Метод класу ObjectInputStream:  
  
           void writeObject(Object o); // записує (заносить) об’єкт в потік  
  
Метод класу ObjectInputStream:  
  
           Object readObject(); // вичитує (вибирає) об’єкт з потоку i повертає посилання на нього

Важливо пам’ятати що в потік можуть бути занесені тільки серіалізовані об’єкти. З іншого боку, об’єкт є серіалізованим, якщо його клас реалізує інтерфейс Serializable.

Відзначимо, що майже всі класи стандартних пакетів Javа реалізують цей інтерфейс. Зокрема, масиви (які є об’єктами спеціальних класів визначених на етапі компіляції) також є серіалізованими. Як приклад наведемо програму, в якій “закріпляються” об’єкти переважно стандартних класів Javа і масив. Напочатку, програма записує до потоку об’єкти – дати, масиву, який містить значення температур і масиву стрінгів, які є описами для значень температури. Далі об’єкти вибираються з потоку і можуть довільним чином бути зміненими.

Лістінг програми з прикладом серіалізації наведено нижче.

import java.io.\*;

import java.util.\*;

class Example

{

public static void main(String args[])

{

Date data = new Date();

int[] temperatura = { 25, 19 , 22};

String[] opis = { "день", "ніч", "вода" };

// Запис

try

{

ObjectOutputStream out = new ObjectOutputStream(

new FileOutputStream("c:\\test.dat")

);

out.writeObject(data);

out.writeObject(opis);

out.writeObject(temperatura);

out.close();

}

catch(IOException exc)

{

exc.printStackTrace();

System.exit(1);

}

// Відкривання (зазвичай в іншій програмі)

try

{

ObjectInputStream in = new ObjectInputStream(

new FileInputStream("c:\\test.ser")

);

Date odczytData = (Date) in.readObject();

String[] odczytOpis = (String[]) in.readObject();

int[] odczytTemp = (int[]) in.readObject();

in.close();

System.out.println(String.valueOf(odczytData));

for (int i=0; i<odczytOpis.length; i++)

System.out.println(odczytOpis[i] + " " +

odczytTemp[i]);

}

catch(IOException exc)

{

exc.printStackTrace();

System.exit(1);

}

catch(ClassNotFoundException exc)

{

System.out.println("Не можна віднайти клас об’єкту");

System.exit(1);

}

}

}

Серіалізація має декілька важливих особливостей:

* при серіалізації не зберігаються статичні поля та поля декларовані зі специфікатором transient; власне специфікатор transient використовується для елементів, які не треба піддавати “закріпленню”,
* повний контроль над способом серіалізації можемо отримати визначивши відповідні методи в класі серіалізованого об’єкту, методи ці повинні мати наступну сигнатуру:  
      private void readObject(java.io.ObjectInputStream stream)  
          throws IOException, ClassNotFoundException;  
    
      private void writeObject(java.io.ObjectOutputStream stream)  
          throws IOException
* цілковитий контроль над форматом і способом серіалізації отримується через реалізацією в класі інтерфейсу Externalizable i визначення методів writeExternal i readExternal

# Механізми розпаралелення мови Java

**Паралельні обчислення** — це форма обчислень, в яких кілька дій проводяться одночасно. Грунтуються на тому, що великі задачі можна розділити на кілька менших, кожну з яких можна розв'язати незалежно від інших.

Є кілька різних рівнів паралельних обчислень: [бітовий](http://uk.wikipedia.org/w/index.php?title=%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%B5%D0%BB%D1%96%D0%B7%D0%BC_%D0%B1%D1%96%D1%82%D0%BE%D0%B2%D0%BE%D0%B3%D0%BE_%D1%80%D1%96%D0%B2%D0%BD%D1%8F&action=edit&redlink=1), [інструкцій](http://uk.wikipedia.org/w/index.php?title=%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%B5%D0%BB%D1%96%D0%B7%D0%BC_%D1%80%D1%96%D0%B2%D0%BD%D1%8F_%D1%96%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%86%D1%96%D0%B9&action=edit&redlink=1), [даних](http://uk.wikipedia.org/w/index.php?title=%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%B5%D0%BB%D1%96%D0%B7%D0%BC_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85&action=edit&redlink=1) та [паралелізм задач](http://uk.wikipedia.org/w/index.php?title=%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%B5%D0%BB%D1%96%D0%B7%D0%BC_%D0%B7%D0%B0%D0%B4%D0%B0%D1%87&action=edit&redlink=1). Паралельні обчислення застосовуються вже протягом багатьох років, в основному в [високопродуктивних обчисленнях](http://uk.wikipedia.org/w/index.php?title=%D0%92%D0%B8%D1%81%D0%BE%D0%BA%D0%BE%D0%BF%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D1%96_%D0%BE%D0%B1%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F&action=edit&redlink=1), але зацікавлення ним зросло тільки недавно, через фізичні обмеження зростання [частоти](http://uk.wikipedia.org/wiki/%D0%A7%D0%B0%D1%81%D1%82%D0%BE%D1%82%D0%B0_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D0%BE%D1%80%D0%B0) процесорів. Оскільки [споживана потужність](http://uk.wikipedia.org/w/index.php?title=%D0%A1%D0%BF%D0%BE%D0%B6%D0%B8%D0%B2%D0%B0%D0%BD%D0%B0_%D0%BF%D0%BE%D1%82%D1%83%D0%B6%D0%BD%D1%96%D1%81%D1%82%D1%8C&action=edit&redlink=1) (і відповідно виділення тепла) комп'ютерами стало проблемою в останні роки, паралельне програмування стає домінуючою парадигмою в [комп'ютерній архітектурі](http://uk.wikipedia.org/w/index.php?title=%D0%90%D1%80%D1%85%D1%96%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BF%E2%80%99%D1%8E%D1%82%D0%B5%D1%80%D0%B0&action=edit&redlink=1), основному в формі [багатоядерних процесорів](http://uk.wikipedia.org/wiki/%D0%91%D0%B0%D0%B3%D0%B0%D1%82%D0%BE%D1%8F%D0%B4%D0%B5%D1%80%D0%BD%D0%B8%D0%B9_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D0%BE%D1%80).

Паралелізм даних — це паралелізм властивий [циклам програм](http://uk.wikipedia.org/wiki/%D0%A6%D0%B8%D0%BA%D0%BB_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%B8), які фокусуються на доставці даних різним обчислювальним вузлам для паралельної обробки. "Розпаралелювання циклів часто приводить до подібних (не обов'язково ідентичних) послідовностей операцій, чи обчислення функцій над елементами великих структур даних. Багато наукових, та інженерних програм проявляють паралелізм даних.

Паралелізм задач — характеристика паралельної програми, яка полягає в тому, що «цілком різні обчислення можуть виконуватись над одими, чи різними даними».Це відрізняє паралелізм задач від паралелізму даних, при якому одне і те ж обчислення виконується над одними і тими ж даними. Паралелізм задач, зазвичай не зростає зі зростанням розміру задачі.

У програмуванні на Java, програмістам зазвичай, приходиться мати справу із окремими нитками. Багатонитковість (multithreading) є важливою можливістю Java. Кожна програма Java має хоча б одну нитку виконання. Виконання програми розпочинається із створенням головної нитки (main thread).

При роботі з нитками, програміст може або в ручну керувати ними, або ж використовувати спеціальний менеджер виконувач (executor). Спочатку, необхідно розглянути перший спосіб управління нитками виконання.

## Створення ниток виконання

Кожна нитка це екземпляр класу Thread. Щоб мати змогу створити нову нитку та оперувати нею, зрозуміло, що необхідний код для виконання у цій нитці. Це можна зробити двома шляхами.

Через реалізацію (implements) інтерфейсу Runnable, що робиться через визначення методу run:

public class HelloRunnable implements Runnable {

public void run() {

System.out.println("Привіт - це нитка!");

}

public static void main(String args[]) {

(new Thread(new HelloRunnable())).start();

}

}

А другий спосіб полягає у створенні підкласу для Thread. Клас Thread уже містить реалізацію інтерфейсу Runnable. Проте його реалізація методу run — порожня. Тож програміст у своєму коді просто заміщає метод run:

public class HelloThread extends Thread {

public void run() {

System.out.println("Привіт - це нитка!");

}

public static void main(String args[]) {

(new HelloThread()).start();

}

}

Як бачимо в обох випадках для створення нової нитки викликається метод Thread.start.

Більш кращим для застосування є перший спосіб, оскільки він дозволяє нам розширити ще якийсь клас у нашому класі (У Java відсутнє множинне успадкування, тож не можливо розширити декілька класів).

## Призупинення та переривання нитки

Виконання нитки можна призупинити за допомогою методу **sleep** класу Thread. Можливо, також, перервати виконання нитки з іншої нитки за допомогою методу **interrupt**. Скориставшись методом **join**, ми можемо задати час очікування на завершення виконання іншої нитки (безкінечно або вказаний час). Наступний приклад демонструє роботу двох ниток. Основна(main) нитка запускає виконання іншої нитки, яка просто виводить важливе повідомлення по одному рядку через кожні 4 секунди. Основна нитка виконання кожної секунди виводить повідомлення, що вона все ще очікує на завершення породженої нитки. Якщо виконання породженої нитки триває більше періоду часу, що заданий у змінній patience(терпіння), то нитка перериває виконання породженої нитки (закінчується терпіння). При перериванні породженої нитки, у ній виникає виняток типу InterruptedException, який обробляється нею за допомогою конструкції try-catch.

package ua.wikibooks.oj;

public class ThreadInteruption {

// Display a message, preceded by

// the name of the current thread

static void threadMessage(String message) {

String threadName =

Thread.currentThread().getName();

System.out.format("%s: %s%n",

threadName,

message);

}

private static class MessageLoop

implements Runnable {

public void run() {

String importantInfo[] = {

"У діброві - дуби,",

"Під дубами - гриби,",

"Трава - між грибами,",

"Хмарки - над дубами."

};

try {

for (int i = 0;

i < importantInfo.length;

i++) {

// Пауза на 4 секунди

Thread.sleep(4000);

// Надруквати повідомлення

threadMessage(importantInfo[i]);

}

} catch (InterruptedException e) {

threadMessage("Виконання задачі не завершене!");

}

}

}

public static void main(String args[])

throws InterruptedException {

// Пауза, у мілісекундах

// перед тим як буде перервано MessageLoop

// thread (по замовчуванню одна година).

long patience = 1000 \* 60 \* 60;

// Якщо наявний аргумент в командному рядку,

// задати терпіння (patience) в секундах

if (args.length > 0) {

try {

patience = Long.parseLong(args[0]) \* 1000;

} catch (NumberFormatException e) {

System.err.println("Аргумент повинен бути цілим числом.");

System.exit(1);

}

}

threadMessage("Старт MessageLoop thread");

long startTime = System.currentTimeMillis();

Thread t = new Thread(new MessageLoop());

t.start();

threadMessage("Чекаю допоки MessageLoop thread не закінчить");

// loop until MessageLoop

// thread exits

while (t.isAlive()) {

threadMessage("Все ще очікую...");

// Почекати 1 секунду

// на закінчення MessageLoop thread

t.join(1000);

if (((System.currentTimeMillis() - startTime) > patience)

&& t.isAlive()) {

threadMessage("Терпіння закінчилось, більше не чекатиму!");

// перервати виконання нитки MessageLoop

t.interrupt();

// Може зайняти певний час

// -- безкінечне очікування

t.join();

}

}

threadMessage("Кінець!");

}

}

При виконанні коду, без задання аргументу, скоріш за все нитка виконання MessageLoop успішно завершить свою роботу, а за нею вже і основна нитка. Якщо ж змінити значення змінної patience на менше, або через аргумент, або прямо в коді, то ми отримаємо перед завершенння програми повідомлення "Виконання задачі не завершене!".

Ось результат з успішним завершення виконання MessageLoop:

main: Старт MessageLoop thread

main: Чекаю допоки MessageLoop thread не закінчить

main: Все ще очікую...

main: Все ще очікую...

main: Все ще очікую...

main: Все ще очікую...

main: Все ще очікую...

Thread-0: У діброві - дуби,

main: Все ще очікую...

main: Все ще очікую...

main: Все ще очікую...

Thread-0: Під дубами - гриби,

main: Все ще очікую...

main: Все ще очікую...

main: Все ще очікую...

main: Все ще очікую...

main: Все ще очікую...

Thread-0: Трава - між грибами,

main: Все ще очікую...

main: Все ще очікую...

main: Все ще очікую...

main: Все ще очікую...

Thread-0: Хмарки - над дубами.

main: Кінець!