Київський національний університет імені Тараса Шевченка

О.В. Галкін, Л.О. Катеринич, О.С. Шкільняк

ПРОГРАМУВАННЯ НА JAVA 8

Навчальний посібник

Київ 2017

УДК 004.43

Галкін О.В., Катеринич Л.О., Шкільняк О.С. Програмування на Java 8: Навчальний посібник для студентів факультету комп’ютерних наук та кібернетики. – К.: ЛОГОС, 2017. – 186 с.

Рецензенти:

Єршов С.В., доктор фіз.-мат. наук, с.н.с.,
учений секретар Інституту кібернетики імені В.М. Глушкова НАН України

Омельчук Л.Л., кандидат фіз.-мат. наук, доцент,
доцент кафедри теорії та технології програмування
Київського національного університету імені Тараса Шевченка

Рекомендовано до друку вченою радою
факультету комп’ютерних наук та кібернетики
11 вересня 2017 року
протокол № 1

В посібнику розглядаються основи мови програмування Java 8, концепції об’єктно-орієнтованого програмування та важливі аспекти застосування супутніх технологій: сервлети, JSP, Hibernate, JDBC.

Посібник призначений головним чином для студентів спеціальності “Інженерія програмного забезпечення”, а також для всіх, хто цікавиться сучасними технологіями програмування.

© О.В. Галкін, Л.О. Катеринич, О.С. Шкільняк, 2017

ЗМІСТ

РОЗДІЛ 1 7

1.1. ОСНОВИ МОВИ JAVA 7
1.1.1. Приклад простої програми 7
1.1.2. Змінні 7
1.1.3. Коментарі 8
1.1.4. Іменовані константи 8
1.2. КЛАСИ ТА ОБ'ЄКТИ 9
1.2.1. Простий клас 9
1.2.2. Поля 9
1.2.3. Керування доступом 10
1.2.4. Створення об'єктів 10
1.2.5. Конструктори 10
1.2.6. Методи 12
1.2.7. Посилання this 13
1.2.8. Перевантаження методів 13
1.2.9. Статичні члени 14
1.2.10. Блоки статичної ініціалізації 14
1.2.11. Статичні методи 15
1.2.12. Збір сміття 15
1.2.13. Метод finalize 15
1.2.14. Метод main 16

РОЗДІЛ 2 17

2.1. СПАДКУВАННЯ 17
2.2. ПОРЯДОК ВИКЛИКУ КОНСТРУКТОРІВ У ПОХІДНИХ КЛАСАХ 18
2.3. ПЕРЕВИЗНАЧЕННЯ МЕТОДІВ І ПРИХОВУВАННЯ ПОЛІВ 19
2.4. КЛЮЧОВЕ СЛОВО SUPER 20
2.5. ОГОЛОШЕННЯ МЕТОДІВ І КЛАСІВ ІЗ КЛЮЧОВИМ СЛОВОМ FINAL 20
2.6. КЛАС OBJECT 21
2.7. АБСТРАКТНІ КЛАСИ ТА МЕТОДИ 22
2.8. ДУБЛЮВАННЯ ОБ'ЄКТІВ 22

РОЗДІЛ 3 24

3.1. ПАКЕТИ ТА ІНТЕРФЕЙСИ 24
3.2. ОПЕРАТОР IMPORT 24
3.3. ІНТЕРФЕЙСИ 25
3.4. РЕАЛІЗАЦІЯ ІНТЕРФЕЙСУ 26
3.5. КОНФЛІКТИ ІМЕН 26
3.6 ФУНКЦІОНАЛЬНІ ІНТЕРФЕЙСИ ТА ЛЯМБДА ВИРАЗИ 27
3.7. ОПЕРАТОРИ ТА ВИРАЗИ 31
3.7.1. Ідентифікатори 31

3.7.2. Символи ... 31
3.7.3. Оголошення змінних ... 32
3.7.4. Масиви ... 32
3.7.5. Ініціалізація масивів ... 33
3.7.6. Явне перетворення типів ... 33
3.7.7. Арифметичні оператори ... 34
3.7.8. Побітові оператори ... 35
РОЗДІЛ 4 ... 36
4.1. ОПЕРАТОР IF-ELSE ... 36
4.2. ОПЕРАТОР SWITCH ... 37
4.3. ЦИКЛИ В JAVA ... 38
4.4. ОПЕРАТОР BREAK ... 39
4.5. ОПЕРАТОР CONTINUE ... 40
4.6. ОПЕРАТОР RETURN ... 40
4.7. ВИКЛЮЧЕННЯ ... 40
4.8. ОПЕРАТОР THROW ... 41
4.9. УМОВА THROWS ... 41
4.5. ОПЕРАТОРИ TRY, CATCH ТА FINALLY ... 41
4.7. ВКЛАДЕНІ КЛАСИ ТА ІНТЕРФЕЙСИ ... 43
4.7.1. Статичні вкладені класи ... 43
4.7.2. Нестатичні вкладені класи ... 43
4.7.3. Розширення внутрішніх класів ... 44
4.7.4. Локальні внутрішні класи ... 46
4.7.5. Анонімні внутрішні класи ... 46
4.7.6. Вкладеність в інтерфейсах ... 47
РОЗДІЛ 5 ... 48
5.1. ШАБЛОНИ В JAVA ... 48
5.2. ПОТОКИ В JAVA ... 48
5.2.1. Створення потоків ... 48
5.2.2. Використання об'єктів Runnable ... 50
5.2.3. Синхронізація ... 51
5.2.4. Методи synchronized ... 51
5.2.5. Оператори synchronized ... 53
5.2.6. Методи wait, notifyall і notify ... 54
5.2.7. Додаткові зауваження щодо методів wait, notify і notifyall ... 57
5.2.8. Планування потоків ... 58
РОЗДІЛ 6 ... 62
6.1. ВЗАЄМНЕ БЛОКУВАННЯ ПОТОКІВ ... 62
6.2. ЗАВЕРШЕННЯ ВИКОНАННЯ ПОТОКУ ... 63
6.3. КОРЕКТНЕ ЗАВЕРШЕННЯ РОБОТИ ПОТОКУ ... 64
6.4. ОЧІКУВАННЯ ЗАВЕРШЕННЯ РОБОТИ ПОТОКУ ... 65
6.5. ПОТОКИ – ДЕМОНИ ... 66

6.6. КВАЛІФІКАТОР VOLATILE 67
6.7. КЛАС THREADGROUP 68
6.8. МЕТОД STOP() 71
6.9. ЗМІННІ THREADLOCAL 71
6.10. НАЛАГОДЖЕННЯ ПОТОКІВ 71
РОЗДІЛ 7 73
7.1. КЛАСИ - ОБОЛОНКИ 73
7.2. РЕФЛЕКСІЯ 75
7.3. КЛАС CLASS 76
РОЗДІЛ 8 82
8.1. КЛАС MODIFIER 82
8.2. КЛАС FIELD 82
8.3. КЛАС METHOD 83
8.4. КЛАС CONSTRUCTOR 85
8.5. КЛАС ACCESSIBLEOBJECT 86
8.6. КЛАС ARRAY 87
8.7. КЛАС PACKAGE 88
8.8. КЛАС PROXY 88
8.9. ЗАВАНТАЖЕННЯ КЛАСІВ 89
РОЗДІЛ 9 93
9.1. ПІДГОТОВКА КЛАСУ ДО ВИКОРИСТАННЯ 93
9.2. ЗАВАНТАЖЕННЯ ДОДАТКОВИХ РЕСУРСІВ 93
9.3. СЕРІАЛІЗАЦІЯ 95
9.3.1. Серіалізація об’єктів Java 95
9.3.2. Підготовка класів до серіалізації 96
9.3.3. Порядок серіалізації та десеріалізації 98
9.3.4. Настроювання механізму серіалізації 99
9.3.5. Контроль версій об'єктів 101
9.3.6. Типи PutField і GetField 102
9.3.7. Інтерфейс Externalizable 103
РОЗДІЛ 10 104
10.1. ФОРМАТ СЕРІАЛІЗОВАНОГО ОБ'ЄКТА 104
10.2. XML 107
10.3. DTD 107
10.4. ОПИС ЕЛЕМЕНТА 108
10.5. ОПИС АТРИБУТІВ 108
10.6. ВИЗНАЧЕННЯ СУТНОСТІ 109
10.7. XLINK 112
10.8. ПРОСТІ ПОСИЛАННЯ 112
10.9. АТРИБУТИ, ЩО ВИЗНАЧАЮТЬ ПОВЕДІНКУ ПОСИЛАНЬ 113
10.10. РОЗШИРЕНІ ПОСИЛАННЯ 114

РОЗДІЛ 11 ..... **116**
11.1. СТОРІНКИ СТИЛІВ XML 116
11.2. XML-АНАЛІЗАТОРИ 121
РОЗДІЛ 12 ..... **123**
12.1. SAX-АНАЛІЗАТОРИ 123
12.2. ДЕРЕВОВИДНА МОДЕЛЬ DOM 127
12.3. ТЕХНОЛОГІЯ RMI 132
РОЗДІЛ 13 ..... **136**
13.1. JDBC 136
13.2. МЕТАДАННІ 142
13.3. ПУЛ З'ЄДНАНЬ 143
13.4. ПАТЕРН DATA ACCESS OBJECT 144
13.4.1. Рагальні відомості про патерн DAO 144
13.4.2. Стратегії для DAO 146
РОЗДІЛ 14 ..... **151**
14.1. РОБОТА З БАЗАМИ ДАНИХ З ВИКОРИСТАННЯМ HIBERNATE 151
14.2. ЗАПИТИ В HIBERNATE 158
14.2.1. HQL 158
14.2.2. Відношення в Hibernate 158
14.2.3. Criteria 162
14.2.4. Запити з використанням SQL 164
14.2.5. Інша реалізація Hibernate JPA 164
РОЗДІЛ 15 ..... **167**
15.1. СЕРВЛЕТИ 167
15.2. ІНТЕРФЕЙСИ SERVLETREQUEST I HTTPSERVLETREQUEST 170
15.3. ІНТЕРФЕЙСИ SERVLETRESPONSE I HTTPSERVLETRESPONSE 172
15.4. ІНТЕРФЕЙС SERVLETCONFIG 173
15.5. РОЗГОРТАННЯ СЕРВЛЕТА 174
15.6. ВИДОБУВАННЯ ІНФОРМАЦІЇ ІЗ ЗАПИТУ 175
РОЗДІЛ 16 ..... **177**
16.1. JSP 177
16.2. ДИРЕКТИВИ 178
16.3. ОГОЛОШЕННЯ 180
16.4. СКРИПТЛЕТИ 180
16.5. ВИРАЗИ 181
16.6. НЕЯВНІ ОБ'ЄКТИ 182
16.7. СТАНДАРТНІ ЕЛЕМЕНТИ ACTION 183
РЕКОМЕНДОВАНА ЛІТЕРАТУРА **186**

РОЗДІЛ 1
1.1. ОСНОВИ МОВИ JAVA

1.1.1. Приклад простої програми

Програми мовою Java будуються на основі класів. Клас, як і у випадку C++ містить у собі члени двох видів: поля й методи. Полями називаються данні, які належать або самому класу, або його об'єктам; значення полів визначають стан об'єкта або класу. Методами називаються послідовності операторів, що виконують деякі дії з полями для зміни стану об'єкта.

Розглянемо приклад простої програми мовою *Java*:

```
class HelloWorld {
  public static void main(String[] args){
  System.out.println("Hello, world");
}}
```

Дана програма виводить на екран:

```
Hello, world.
```

Єдиним параметром методу *main* є масив об'єктів *String*, які являють собою аргументи програми з командного рядка, що використовується для запуску. У наведеному вище прикладі main містить усього один оператор, що викликає метод *println*.

1.1.2. Змінні

Змінні в Java оголошуються аналогічно C++. Розглянемо приклад:

```
/** Вивід чисел Фібоначі < 50 */
class Fibonacci {
public static void main(String[] args) {
  int lo = 1; int hi = 1; //оголошення змінних
  System.out.println(lo);
  while (hi < 50) {
    System.out.println(hi);
    hi = lo + hi;  // Значення змінної hi
    lo = hi - lo; /* Нове значення lo дорівнює старому hi,
    тобто   сумі за винятком старого lo */
  }
}}
```

В Java є наступні примітивні типи даних:

- *boolean* одне із двох значень: *true* або *false*;

- *char* 16-розрядний символ у кодуванні Unicode 1.1;
- *byte* 8-розрядне ціле (зі знаком);
- *short* 16-розрядне ціле (зі знаком);
- *int* 32-розрядне ціле (зі знаком);
- *long* 64-розрядне ціле (зі знаком);
- *float* 32-розрядне із плаваючою крапкою (IEEE 754-1985);
- *double* 64-розрядне із плаваючою крапкою (IEEE 754-1985).

1.1.3. Коментарі

В Java передбачені коментарі трьох видів - всі вони зустрічаються в попередньому прикладі.

Текст, що записаний за символами // аж до кінця рядка, ігнорується компілятором; теж саме стосується й тексту, що знаходиться між символами /* та */.

Коментар третього типу зустрічається на самому початку програми, між символами /** і */. Коментар, що починається із двох зірочок, це коментар, що документує.

1.1.4. Іменовані константи

Константами називаються фіксовані значення - наприклад, 12, 17.9 або "String like this". Щоб створити іменовану константу в Java, необхідно вказати в її оголошенні ключові слова *static* і *final* і задати початкове значення:

```
class CircleStuff {
  static final double p = 3.1416;
}
```

1.2. КЛАСИ ТА ОБ'ЄКТИ

1.2.1. Простий клас

Основними компонентами класу є поля (дані) і методи (код для роботи з ними). Розглянемо простий клас *Body*, призначений для зберігання відомостей про небесні тіла:

```
class Body {
   public long idNum;
   public String nameFor;
   public Body orbits;
   public static long nextID = 0;
}
```

Насамперед оголошується ім'я класу. Оголошення класу в мові Java створює ім'я типу, так що надалі посилання на об'єкти цього типу можуть оголошуватися в такий спосіб:

```
Body mercury;
```

Подібне оголошення вказує на те, що *mercury* є посиланням на об'єкт класу *Body*. Спочатку посилання має значення *null*, і об'єкт, на який посилається *mercury*, не існує доти, поки ви не створите його явно; щодо цього Java відрізняється від інших мов програмування, у яких об'єкти створюються при оголошенні змінних.

1.2.2. Поля

Змінні класу називаються полями; прикладами можуть служити поля *nameFor* і *orbits*, що входять у клас *Body*. Наявність у кожного об'єкта окремого екземпляра полів означає, що його стан унікальний.

Іноді буває необхідно, щоб один екземпляр поля спільно використовувався всіма об'єктами класу. Такі поля оголошуються із ключовим словом *static* і називаються статичними полями, або змінними класу. При оголошенні в класі поля *static* всі об'єкти цього класу розділяють одну й ту саму копію статичного поля.

1.2.3. Керування доступом

Для керування доступом до полів і методів класу з інших класів, а також для керування успадкуванням їх у підкласах члени класів можуть оголошуватися з одним із чотирьох атрибутів доступу:

- Відкритий *(public*): до членів класу завжди можна звертатися з будь-якого місця, у якому доступний сам клас; такі члени успадковуються в підкласах;
- Закритий (*private*): доступ до членів класу здійснюється тільки із самого класу;
- Захищений (*protected*): до даних членів дозволяється доступ з підкласів і з функцій, що входять у той же пакет. Такі члени успадковуються підкласами;
- Пакетний: доступ до членів, оголошеним без вказівки атрибута доступу, здійснюється тільки з того ж пакета. Такі члени успадковуються підкласами пакета.

1.2.4. Створення об'єктів

Для першої версії класу *Body* створення та ініціалізація об'єктів, що представляють небесні тіла, відбувається в такий спосіб:

```
Body sun = new Body();
sun.idNum = Body.nextID++;
sun.nameFor = “Sol”;
sun.orbits = null; // Сонце є центром Сонячної системи
Body earth = new Body();
earth.idNum = Body.nextID++;
earth.nameFor = “Earth”;
earth.orbits = sun;
```

1.2.5. Конструктори

Кожний знову створений об'єкт має деякий вихідний стан. Значення полів можуть ініціалізуватися при їхньому оголошенні - іноді цього буває досить (наприклад поле цілого типу ініціалізується нулем). Однак досить

часто для визначення вихідного стану простої ініціалізації даних виявляється недостатньо; наприклад, можуть знадобитися які-небудь вихідні дані, або ж виконувані операції не можуть бути представлені у вигляді простого присвоювання.

Для тих випадків, коли простої ініціалізації недостатньо, використовуються конструктори. Ім'я конструктора збігається з ім'ям класу, що він ініціалізує. Конструктори, подібно методам, можуть одержувати один або кілька параметрів, однак вони не є методами й не можуть повертати ніякого значення. Параметри конструктора (якщо вони є) вказуються в дужках за ім'ям типу при створенні об'єкта оператором new. Розглянемо приклад:

```
class Body {
   public long idNum;
   public String name = “”;
   public Body orbits = null;
   private static long nextID = 0;
   Body() { idNum = nextID++; }
  }
```

У цьому випадку доданий конструктор за замовчуванням для класу *Body*. Можна також додати конструктор з параметрами:

```
Body(String bodyName, Body orbitsAround) {
 this();
 name = bodyName;
 orbits = orbitsAround;
}
```

Видно, що з одного конструктора можна викликати інший конструктор цього ж класу - для цього першим виконуваним оператором повинен бути виклик *this()*. Це називається “явним викликом конструктора”. Якщо для виклику конструктора необхідні параметри, вони можуть передаватися. Тоді створення об'єктів відбувається в такий спосіб:

```
Body sun = new Body(“Sol”, null);
Body earth = new Body(“Earth”, sun);
```

Якщо не оголошено жодного конструктора, тоді створюється без аргументний конструктор за умовчанням, який не робить нічого. Цей конструктор створюється автоматично лише в тих випадках, коли немає

жодних інших конструкторів, — існують класи, для яких без аргументний конструктор працюватиме невірно.

1.2.6. Методи

Методи класу звичайно містять код, що аналізує стан об'єкта й змінює його. Виклик методу являє собою операцію, виконувану з об'єктом за допомогою посилання на нього з використанням оператора:

```
reference.method(parameters)
```

Кожний метод викликається з певною кількістю параметрів. Java не підтримує методів, у яких допускається змінне число параметрів. Кожний параметр має строго певний тип - примітивний або посилальний. Крім того, методи мають тип значення, що повертається, що вказується перед їхнім ім'ям.

Розглянемо приклад методу із класу *Body*:

```
public String toString() {
  String desc = idNum + " (" + name + ")";
  if (orbits != null)
  desc += " orbits " + orbits.toString();
  return desc;
}
```

У цьому методі виконується конкатенація об'єктів *String* за допомогою операторів + і +=. Спочатку утвориться рядок, що містить ідентифікатор і назву об'єкта. Якщо дане небесне тіло обертається навколо іншого, то до неї приєднується рядок з описом центра обертання, для чого викликається метод *toString* відповідного об'єкта. Послідовність рекурсивних викликів продовжує будувати ланцюжок тіл, що обертаються навколо один одного, поки не буде знайдене тіло, що не має центра обертання.

Існує кілька способів, якими вдається домогтися повернення методом декількох значень: можна повертати посилання на об'єкт, у якому ці значення зберігаються у вигляді полів; приймати як параметри посилання на об'єкти, у яких повинні зберігатися результати; нарешті, можна повертати масив з результатами. Розглянемо приклад:

```
class Permissions { public boolean canDeposit,
  canWithdraw, canClose; }
```

Метод, що заповнює ці поля має вигляд:

```
class Account {
public Permissions permissionsFor(Person who) {
Permissions perm = new Permissions();
perm.canDeposit = canDeposit(who);
perm.canWithdraw = canWithdraw(who);
perm.canClose = canClose(who);
return perm; }  // ... визначення методу canDeposit()
   }
```

Якщо метод не повертає ніякого значення, то на місці типу, що повертається, ставиться ключове слово *void*.

1.2.7. Посилання *this*

Посилання *this* в Java має ту ж функцію, що й у C++. Спеціальне посилання на об'єкт *this* можуть також застосовуватися усередині нестатичних методів; вона вказує на поточний об'єкт, для якого був викликаний даний метод. *this* частіше всього використовується для передачі посилання на поточний об'єкт як параметр для інших методів.

Наприклад:

```
Service.add(this);
```

this неявно додається в початок кожного посилання на поле або метод, якщо тільки програміст не вказав посилання на інший об'єкт. Наприклад, присвоєння значення полю *str* у наступному класі:

```
class Name {
public String str;
Name() { str = "";}
}
```

рівносильно наступному:

```
this.str = "";
```

1.2.8. Перевантаження методів

Мова Java допускає перевантаження методів. Перевантаження здійснюється за сигнатурою методів. Розглянемо приклад:

```
public Body orbitsAround() { return orbits;}
public void orbitsAround(Body around)
{ orbits = around;}
```

Перевантаження операторів в Java не допускається.

1.2.9. Статичні члени

Для статичного члена створюється всього один екземпляр, загальний для всього класу, замість побудови його копій у кожному об'єкті класу. У випадку статичних змінних (змінних класу), це рівно одна змінна, незалежно від того, скільки об'єктів було створено на основі класу (навіть якщо жодного).

Ініціалізація статичних полів класу відбувається до того, як вони використовуються або запускається кожний з його методів.

Розглянемо приклад:

```
class Value {
  public static double UNSET = 10.2;
  private double V;
  public void unset() { V = UNSET; }
  ...
}
```

1.2.10. Блоки статичної ініціалізації

Клас також може містити блоки статичної ініціалізації, які присвоюють значення статичним полям або виконують іншу необхідну роботу. Статичний ініціалізатор виявляється найбільш корисним у тих випадках, коли простої ініціалізації в оголошенні поля недостатньо.

Розглянемо приклад:

```
class Primes {
  protected static int[] knownPrimes = new int[4];
static {
   knownPrimes[0] = 2;
   for(int i = 1; i < knownPrimes.length; i++)
   knownPrimes[i] = nextPrime(); }
}
```

Статична ініціалізація усередині класу виконується в порядку з ліва на право і зверху донизу.

Якщо статичнийініціалізатор із класу *X* викликає метод із класу *Y*, а статичний ініціалізатор класу *Y* викликає метод із класу *X* для завдання своїх статистичних величин, то коли *Y*, у свою чергу звернеться до методу з *X*, те останній буде виконуватися без завершення статичної ініціалізації. Всі статичні поля для яких ініціалізація не була виконана будуть мати значення за замовчуванням.

1.2.11. Статичні методи

Статичні методи викликаються для цілого класу. Подібні методи також називаються методами класу. Статичний метод може виконувати завдання, загальні для всіх об'єктів класу, наприклад, повертати наступний серійний номер або що-небудь у цьому роді.

За межами класу статичні методи звичайно викликаються через ім'я класу, а не через посилання:

```
prime = Primes.nextPrime();
knownCnt = Primes.knownPrimes.length;
```

1.2.12. Збір сміття

Java виконує весь збір програмного сміття автоматично. Це означає, що пам'ять, що займається об'єктом, що не використовується, може бути повернута в систему. Об'єкт є “тим, що не використовується”, коли на нього відсутні посилання в статичних даних і в будь-якій зі змінних виконуваного в даний момент методу, коли не вдається знайти посилання на нього за допомогою відстеження полів і елементів масивів статичних даних і змінних методів.

1.2.13. Метод *finalize*

Клас може реалізувати метод з ім'ям *finalize*, що виконується перед знищенням об'єкта або при завершенні роботи віртуальної машини. Метод

finalize дає можливість використовувати видалення об'єкта для звільнення інших, не пов'язаних з Java ресурсів. Він оголошується в такий спосіб:

```
protected void finalize() throws Throwable {
  super.finalize();
  // ...
}
```

Роль методу *finalize* стає особливо істотної при роботі із зовнішніми стосовно Java ресурсами, які занадто важливі, щоб можна було чекати етапу збору сміття. Наприклад, відкриті файли (число яких звичайно обмежено) не можуть чекати завершальної фази *finalize* - немає ніякої гарантії, що об'єкт, що містить відкритий файл, буде знищений збирачем сміття до того, як витратяться всі ресурси по відкриттю файлів. Розглянемо приклад:

```
public class ProcessFile {
   private Stream File;
   public ProcessFile(String path) {
       File = new Stream(path);   }
     // ...
   public void close() {
      if (File != null) {
             File.close();
             File = null;  }        }
   protected void finalize() throws Throwable {
   super.finalize();
   close();    }
}
```

Метод *finalize* може “воскресити” об'єкт, знову роблячи його об’єктом, що використовується - скажімо, включаючи його в статичний список об'єктів. Однак, подібні дії не рекомендуються.

1.2.14. Метод *main*

При запуску програми на Java система знаходить і запускає метод main цього класу. Метод *main* повинен бути оголошений як public, static і void (тобто не повертає ніякого значення), і йому повинен передаватися один аргумент типу *String[]*. Розглянемо приклад:

```
  class Echo {
     public static void main(String[] args) {
         for (int i = 0; i < args.length; i++)
           System.out.print(args[i]);
           System.out.println();
   }}
```

РОЗДІЛ 2
2.1. СПАДКУВАННЯ

Спадкування в Java має той же зміст, що й у C++. Однак спадкування в Java здійснюється за допомогою ключового слова *extends*. Розглянемо приклад:

```
class Attr {
  private String name;
  private Object value = null;
  public Attr(String nameOf) { name = nameOf; }
  public Attr(String nameOf, Object valueOf) {
    name = nameOf; value = valueOf; }
  public String nameOf() { return name; }
  public Object valueOf() { return value; }
  ...
class ColorAttr extends Attr {
  private ScreenColor myColor;
  public ColorAttr(String name, Object value) {
    super(name, value); //виклик конструктора суперкласу
    decodeColor(); }
  public ColorAttr(String name) {
    this(name, "transparent"); // виклик першого конструктора }
  public ColorAttr(String name, ScreenColor value) {
    super(name, value.toString());
    myColor = value; }
  public Object valueOf(Object newValue) {
    // спочатку виконати метод valueOf() суперкласу
    Object retval = super.valueOf(newValue);
    decodeColor();
    return retval;
 }}
```

Ієрархія класів для нашого приклада виглядає у такий спосіб (Рис.2.1.) (клас *Object* присутній у будь-якій ієрархії):

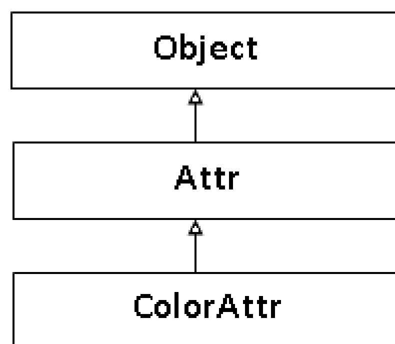


Рис.2.1. Ієрархія класів

2.2. ПОРЯДОК ВИКЛИКУ КОНСТРУКТОРІВ У ПОХІДНИХ КЛАСАХ

При створенні об'єкта всі його поля набувають вихідні значення за замовчуванням залежно від їхнього типу. Потім відбувається виклик конструктора. Кожний конструктор виконується за три фази:

1. Виклик конструктора суперкласу;
2. Присвоєння значень полям за допомогою ініціалізаторів;
3. Виконання тіла конструктора.

Розглянемо приклад:

```
 class X {
   protected int xMask = 0x00ff;
   protected int fullMask;
   public X()
      { fullMask = xMask; }
   public int mask(int orig)
     { return (orig & fullMask);
 }}

class Y extends X {
   protected int yMask = 0xff00;
   public Y()
      { fullMask |= yMask;
 }}
```

Розглянемо значення полів на кожному етапі:

1. Присвоєння значень полям за замовчуванням:

```
xMask=0, fullMask=0, yMask=0.
```

2. Виклик конструктора класу *Y*:

```
xMask=0, fullMask=0, yMask=0.
```

3. Виклик конструктора *X*:

```
xMask=0, fullMask=0, yMask=0.
```

4. Ініціалізація полів *X*:

```
xMask= 0x00ff, fullMask=0, yMask=0.
```

5. Виконання конструктора X

```
xMask= 0x00ff, fullMask= 0x00ff, yMask=0.
```

6. Ініціалізація полів *Y*:

```
xMask= 0x00ff, fullMask= 0x00ff, yMask= 0xff00.
```

7. Виконання конструктора *Y*:

```
xMask= 0x00ff, fullMask= 0x00ff, yMask= 0xffff
```

2.3. ПЕРЕВИЗНАЧЕННЯ МЕТОДІВ І ПРИХОВУВАННЯ ПОЛІВ

Методи в Java як і в C++ можна перевизначити. Розширений клас може змінити права доступу до методів, що є успадкованими з суперкласу, але лише в тому випадку, якщо він розширює їх. Метод, оголошений у суперкласі як *protected*, може бути повторно представлений як *protected* (цілком звичайна ситуація) або *public*, але не як *private*. Обмежувати доступ до методів у порівнянні із суперкласом безглуздо, оскільки таке обмеження дуже легко обійти: досить перетворити посилання в супертип з більшими правами доступу й використовувати її для виклику методу.

При виклику методу для деякого об'єкта його реалізація вибирається залежно від фактичного типу об'єкта. Розглянемо наступний приклад:

```
class SuperShow {
  public String str = "SuperStr";
  public void show() {
    System.out.println("Super.show: " + str);
}}

class ExtendShow extends SuperShow {
  public String str = "ExtendStr";
  public void show() {
    System.out.println("Extend.show: " + str); }

public static void main(String[] args) {
    ExtendShow ext = new ExtendShow();
    SuperShow sup = ext;
    sup.show(); // викликається show із класу ExtendShow
    ext.show(); // викликається show із класу ExtendShow
    System.out.println("sup.str = " + sup.str);//друкується
    SuperStr System.out.println("ext.str = " + ext.str);
    //друкується ExtendShow
}}
```

Якщо існуючий метод одержує параметр типу *SuperShow* і звертається до *str* через посилання на об'єкт-параметр, він завжди буде одержувати *SuperShow.str*, навіть якщо методу насправді був переданий об'єкт типу *ExtendShow*. Якби класи були спроектовані так, щоб для доступу до рядка застосовувався спеціальний метод, то в цьому випадку був би викликаний перевизначений метод, що повертає *ExtendShow.str*. Це ще одна із причин, по

якій визначення класів із закритими даними, доступ до яких здійснюється за допомогою методів, є кращим.

2.4. КЛЮЧОВЕ СЛОВО SUPER

Ключове слово *super* може використовуватися у всіх нестатичних методах класу. При доступі до полів або виклику методів ключове слово super являє собою посилання на поточний об'єкт як екземпляр суперкласу.

У виклику виду *super.method* завжди використовується реалізація *method* із суперкласу, а не його перевизначена реалізація, що перебуває десь нижче в ієрархії класів. Розглянемо приклад:

```
class That {
  /** повернути ім'я класу */
   protected String nm() { return ”That”;
}}

class More extends That {
   protected String nm() { return ”More”; }
   protected void printNM() {
   That sref = super;
   System.out.println(”this.nm() = ” + this.nm());
   //Викликається nm() з More
   System.out.println(”sref.nm() = ” + sref.nm());
   //Викликається nm() з More
   System.out.println(”super.nm() = ”+super.nm());
   //Викликається nm() з That
}}
```

2.5. ОГОЛОШЕННЯ МЕТОДІВ І КЛАСІВ ІЗ КЛЮЧОВИМ СЛОВОМ FINAL

Якщо метод оголошений з атрибутом *final*, це означає, що жоден розширений клас не зможе перевизначити даний метод з метою змінити його поведінку. Інакше кажучи, дана версія методу є остаточною.

Подібним чином можуть оголошуватися класи:

```
final class NoExtending {...}
```

Клас, позначений з атрибутом *final*, не може мати спадкоємців, а всі його методи також неявно є *final*.

2.6. КЛАС OBJECT

Всі класи є явними або неявними розширеннями класу Object і, таким чином, успадковують його методи. Розглянемо методи класу Object:

- **public boolean equals(Object obj)** - порівнює об'єкт-одержувач із об'єктом, на який вказує посилання obj; повертає true, якщо об'єкти рівні між собою, і false у противному випадку. Якщо потрібно з'ясувати, чи вказують два посилання на той самий об'єкт, можна зрівняти їх за допомогою операторів == і !=, а метод equals призначений для порівняння значень. Реалізація методу equals, прийнята в Object за замовчуванням, припускає, що об'єкт дорівнює лише самому собі.
- **public int hashCode()** - повертає хеш-код для даного об'єкта. Кожному об'єкту може бути привласнений деякий хеш-код, що використовується при роботі з хеш-таблицями. За замовчуванням повертає значення, що є унікальним для кожного об'єкта. Воно використовується при збереженні об'єктів у таблицях Hashtable.
- **protected Object clone() throws CloneNotSupportedExcepti** - повертає дублікат об'єкта. Дублікатом є новий об'єкт - копія об'єкта, для якого викликався метод clone.
- **public final Class getClass()** - повертає об'єкт типу Class, що відповідає класу даного об'єкта.
- **protected void finalize() throws Throwable** - кінцеві операції з об'єктом, здійснювані під час збирання сміття. Методи hashCode і equals повинні перевизначитися, якщо необхідно іншу концепцію рівності об'єктів, що відрізняється від прийнятої в класі Object. За замовчуванням вважається, що два різних об'єкти не рівні між собою, а їх хеш-коди не повинні збігатися.

2.7. АБСТРАКТНІ КЛАСИ ТА МЕТОДИ

Абстрактний клас необхідний, коли деяка поведінка характерна для більшості або всіх об'єктів даного класу, але деякі аспекти мають сенс лише для обмеженого кола об'єктів, що не є складовими суперкласу. В Java такі класи оголошуються із ключовим словом *abstract*, і кожний метод, не реалізований у класі, також оголошується як *abstract*. При цьому в класі спадкоємці не обов'язково реалізовувати всі абстрактні методи базового класу. Розглянемо приклад:

```
abstract class Benchmark {
abstract void benchmark();
public long repeat(int count) {
  long start = System.currentTimeMillis();
  for (int i = 0; i < count; i++) benchmark();
  return(System.currentTimeMillis() - start);}}

class MethodBenchmark extends Benchmark {
  public void benchmark() { ....}
  public static void main(String[] args) {
  int count = Integer.parseInt(args[0]);
  long time = new MethodBenchmark().repeat(count);
  System.out.println(count + ” methods in ” + time + ”
  milliseconds”);
}}
```

2.8. ДУБЛЮВАННЯ ОБ'ЄКТІВ

Метод *Object.clone* допомагає робити дублювання об'єктів. При дублюванні вертається новий об'єкт, вихідний стан якого копіює стан об'єкта, для якого був викликаний метод *clone*. Всі наступні зміни, внесені в об'єкт-дубль, не змінюють стани вихідного об'єкта.

Найпростіша можливість створити клас, що дублюється - оголосити про реалізацію в ньому інтерфейсу *Cloneable*:

```
class MyClass extends AnotherClass implements Cloneable {...}
```

Метод clone в інтерфейсі *Cloneable* має атрибут *public*, отже, метод *MyClass.clone*, успадкований від *Object*, також буде *public*. Розглянемо приклад:

```
public class IntegerStack implements Cloneable { private int[]
buffer;
private int top;
```

```
public IntegerStack(int maxContents) {
buffer = new int[maxContents];
top = -1;}
public void push(int val) {
 buffer[++top] = val; } }
```

Якщо копіювати об'єкт класу *IntegerStack* прямо, наприклад:

```
 IntegerStack first = new IntegerStack(2);
 first.push(2);
 first.push(9);
 IntegerStack second = (IntegerStack)first.clone();
```

то змінюючи *second* ми міняємо *first*.

Правильне копіювання - це перевизначення методу *clone* у класі *IntegerStack*, тобто у класі *IntegerStack* визначаємо метод *clone*:

```
public Object clone() {
  try {
  IntegerStack nObj =(IntegerStack)super.clone();
  nObj.buffer = (int[])buffer.clone();
  return nObj; }
catch (CloneNotSupportedExeption e) {
// Не може відбутися - метод clone() підтримується класом
IntegerStack, так і масивами
throw new InternalError(e.toString());}}
```

Можна вимагати, щоб метод *clone* підтримувався у всіх спадкоємцях даного класу, - для цього необхідно перевизначити метод *clone* так, щоб у його сигнатуру не входило оголошення про ініціювання виключення *CloneNotSupportedException*. У результаті підкласи, у яких реалізується метод *clone*, не зможуть ініціювати виключення *CloneNotSupportedException*, оскільки методи підкласу не можуть вводити нові виключення.

Аналогічно, якщо метод *clone* буде завжди ініціювати виключення *CloneNotSupportedException*, то в цьому випадку це еквівалентно тому, що клас не підтримує клонування.

РОЗДІЛ 3
3.1. ПАКЕТИ ТА ІНТЕРФЕЙСИ

Пакет (package) - це якийсь контейнер, що використовується для того, щоб ізолювати імена класів (у C++ аналог пакета - це простір імен). Для створення пакета використовується ключове слово *package*, що повинне стояти на початку файлу (якщо такого слова немає, то класи у файлі потрапляють у безіменний простір імен).

Якщо оголосити клас, що належить певному пакету, наприклад, *package java.awt.image;* то й вихідний код цього класу повинен зберігатися в каталозі *java/awt/image*. Варто відзначити, що каталог, який транслятор Java буде розглядати як кореневий для ієрархії пакетів, можна задавати за допомогою змінної оточення *CLASSPATH*. За допомогою цієї змінної можна також задати кілька кореневих каталогів для ієрархії пакетів (через ; як у звичайному *PATH*).

Якщо, наприклад, написаний клас *Myclass.java* і розміщений у пакеті test, тоді, після компіляції, цей клас можна запустити java test.Myclass.

3.2. ОПЕРАТОР IMPORT

Після оператора *package* у файлі звичайно йдуть оператори *import*. Загальна форма оператора *import* така:

```
import пакет1 [.пакет2].(ім'я_класу|*);
```

Тут пакет1 - ім'я пакета верхнього рівня, пакет2 - це необов'язкове ім'я пакета, вкладеного в перший пакет і відділене крапкою. І, нарешті, після вказівки шляху в ієрархії пакетів, вказується або ім'я класу, або метасимвол зірочка. Зірочка означає, що, якщо Java-транслятору буде потрібно який-небудь клас, для якого пакет не визначено явно, він повинен переглянути весь вміст пакета із зірочкою замість ім'я класу. У наведеному нижче фрагменті коду показані обидві форми використання оператора *import* :

```
import java.util.Date
import java.io.*;
Розглянемо приклад:
```

```
package p1;
public class Protection {
 int n = 1;
 private int n_pri = 2;
 protected int n_pro = 3;
 private protected int n_pripro = 4;
 public int n_pub = 5;
   public Protection() {
      System.out.println("base constructor");
      System.out.println("n="+n);
      System.out.println("n_pri="+n_pri);
      System.out.println("n_pro="+n_pro);
      System.out.println("n_pripro="+n_pripro);
      System.out.println("n_pub="+n_pub);
} }
```

Клас *Protection* належить пакету *p1*.

3.3. ІНТЕРФЕЙСИ

Інтерфейси Java створені для підтримки динамічного вибору методів під час виконання програми. Інтерфейси схожі на класи, але на відміну від останніх у інтерфейсів немає змінних, а в оголошеннях методів відсутня реалізація. Клас може мати будь-яку кількість інтерфейсів. Інтерфейс також відрізняється й від абстрактного класу. В Java якщо клас реалізує (успадковує) інтерфейс, то він повинен реалізувати всі методи оголошені в інтерфейсі, для абстрактного класу це не обов'язково.

В оголошенні інтерфейсу використовується ключове слово interface. В інтерфейсі можна також повідомляти константи, тобто визначення інтерфейсу має вигляд:

```
interface ім'я {
тип_результату ім'я_методу1(список параметрів);
тип ім'я_ final-змінної = значення;
 }
```

(модифікатор *final* можна не вказувати, він буде доданий автоматично).

Розглянемо приклад:

```
interface Callback {
void callback(int param); }
```

Інтерфейси допускають розширення. Розглянемо приклад:

```
interface FloorWax{
double f();}
interface DessertTopping{
```

```
int Myconst=10;
double f1();}
interface Shimmer extends FloorWax, DessertTopping {
//розширення інтерфейсу
double amazingPrice();
}
```

3.4. РЕАЛІЗАЦІЯ ІНТЕРФЕЙСУ

Реалізація інтерфейсу здійснюється за допомогою ключового слова ***implements***. Таким чином, клас, що реалізує інтерфейс, буде мати вигляд:

```
      class ім'я_класу [extends суперклас] [implements
      інтерфейс0 [, інтерфейс1...]]
      { тіло класу }
```

Розглянемо приклад:

```
     class Client implements Callback { void callback(int p) {
       System.out.println("callback called with " + p);
       } }
```

Метод *callback* інтерфейсу, що визначений раніше, викликається через змінну - посилання на інтерфейс:

```
class TestIface {
public static void main(String args[])
{ Callback c = new client();
 c.callback(42); } }
```

3.5. КОНФЛІКТИ ІМЕН

Якщо два методи відрізняються тільки типом ініційованих виключень, метод класу, що реалізує інтерфейс зобов'язаний відповідати обом оголошенням з однаковими сигнатурами (кількістю й типом параметрів), але може ініціювати свої виключення. Однак методи в межах класу не повинні відрізнятися тільки складом ініційованих виключень.

Розглянемо приклад:

```
interface X {
void setup() throws SomeExeption;
}
interface Y {
void setup();}
class Z implements X, Y {
public void setup() {  ... }
}
```

Клас *Z* може містити єдину реалізацію, що відповідає *X.setup* і *Y.setup*. Метод може ініціювати менше виключень, чим оголошено в його суперкласі, тому при оголошенні *Z.setup* необов'язково вказувати, що в методі ініціюється виключення типу *SomeException*. *X.setup* тільки дозволяє використовувати дане виключення.

3.6 ФУНКЦІОНАЛЬНІ ІНТЕРФЕЙСИ ТА ЛЯМБДА ВИРАЗИ

Синтаксис лямбда виразу має вигляд:

параметри -> {тіло функції}.

Типи параметрів можна опускати. Наприклад:

```
p -> return p.getGender() == Person.Sex.MALE
                  && p.getAge() >= 18
                  && p.getAge() <= 25
```

Розглянемо наступний код:

```
interface MyInterface{
  public void func(int a);
}
public class JavaApplication106 {
    public static void f(MyInterface m){
        m.func(20);
    }
    public static void main(String[] args) {
        f(p->{int c=2*p; System.out.println(c);});
    }
 }
```

В цьому випадку лямбда функція нічого не повертає, тому тут return не потрібен.

Наведений код є еквівалентним такому:

```
interface MyInterface{
  public void func(int a);
}
class Impl implements MyInterface{
  public void func(int a){
     int c=2*a;
    System.out.println(c);
   }
}
public class JavaApplication106 {
    public static void f(MyInterface m){
        m.func(20);
    }
    public static void main(String[] args) {
```

```
        MyInterface obj=new Impl();
        f(obj);
    }
 }
```

Звернемо увагу на помилку компіляції в наступному прикладі:

```
interface MyInterface{
  public void func(int a);
  public void g();
}
public class Main {
    public static void f(MyInterface m){
        m.func(20);
    }
    public static void main(String[] args) {
        f(p->{int c=2*p; System.out.println(c);});
        //Помилка: MyInterface не є функціональним інтерфейсом
    }
 }
```

Лямбда функція може приймати декілька параметрів:

```
interface MyInterface{
  public int func(int a,int b);
}
public class JavaApplication106 {
    public static void f(MyInterface m){
        System.out.println(m.func(20,30));
    }
    public static void main(String[] args) {
        f((p,p1)->{int c=p+p1; return c;});
    }
 }
```

Наведемо приклад використання лямбда функцій для реалізації функціональних інтерфейсів:

```
class A {
    int a;
    public int getA(){
        return a;
    }
}
interface MyInterface{
  public int func(A a);
}
public class Main {
    public static void main(String[] args) {
        MyInterface m=p->{return p.getA();};
        A pa=new A();
        pa.a=200;
        System.out.println(m.func(pa));
    }
}
```

Розглянемо замикання в лямбда виразах. Ми можемо використвувати в лямбда виразах змінні з охоплюючої області видимості:

```
class A {
    int a;
    public int getA(){
        return a;
    }
}
interface MyInterface{
  public int func(A a);
}
public class Main {
    static int b=100;
    public static void main(String[] args) {
        int c=30;
        MyInterface m=p->{return p.getA()+b+c;};  //вірно
        A pa=new A();
        pa.a=200;
        System.out.println(m.func(pa));
    }
}
```

Поглянемо, як виглядатимуть посилання на методи. Розпочнемо з посилань на нестатичні методи:

```
class A {
    int a;
    public int getA(){
        return a;
    }
}
interface MyInterface{
  public int func(A a);
}
class B{
   public int mymethod(A pa){
       return pa.a+200;
   }
}
public class Main {
    static int b=100;
    public static void main(String[] args) {
        int c=30;
        B pb=new B();
        MyInterface m=pb::mymethod;
        A pa=new A();
        pa.a=200;
        System.out.println(m.func(pa));
    }
 }
```

Тепер розглянемо приклад такого посиланя на статичні методи:

```
class A {
    int a;
    public int getA(){
        return a;
    }
}
interface MyInterface{
  public int func(A a);
}
class B{
   public static int mymethod(A pa){
       return pa.a+200;
   }
}
public class JavaApplication106 {
    static int b=100;
    public static void main(String[] args) {
        int c=30;
        MyInterface m=B::mymethod;
        A pa=new A();
        pa.a=200;
        System.out.println(m.func(pa));
    }
}
```

Приклад посилання на метод екземпляру з довільного об’єкту визначеного типу:

```
String[] stringArray = { "Barbara", "James", "Mary", "John",
"Patricia", "Robert", "Michael", "Linda" };
Arrays.sort(stringArray, String::compareToIgnoreCase);
```

В нашому випадку виклик метода виглядатиме як

```
a.compareToIgnoreCase(b)
```

В Java 8 можна використовувати посилання на конструктори. Розглянемо відповідний приклад.

```
interface MyInterface{
  public A func();
}
class A{
    int x;
    public A(){
        this.x=100;
    }
    public int getX(){
        return this.x;
    }
}
public class JavaApplication106 {
   public static void  f(MyInterface m){
        A mm=m.func();
```

```
        System.out.println(mm.getX());
    }
    public static void main(String[] args) {
        f(A::new);
    }
}
```

3.7. ОПЕРАТОРИ ТА ВИРАЗИ

Всі програми Java написані в Unicode - 16-розрядному наборі символів. Перші 256 символів Unicode являють собою набір Latin-1, а основна частина перших 128 символів Latin-1 відповідає 7-розрядному набору символів ASCII.

3.7.1. Ідентифікатори

Ідентифікатори Java, що використовуються для іменування оголошених у програмі величин (змінних і констант) і міток, повинні починатися з букви, символу підкреслення (_) або знака долара ($), за якими ідуть букви або цифри в довільному порядку. Імена змінних в Java можна писати буквами національних алфавітів.

3.7.2. Символи

Деякі службові символи в Java:

- \n перехід на новий рядок (\u000A);
- \t табуляція (\u0009);
- \\ зворотна коса риса (\u005C);
- \' апостроф (\u0027);
- \" лапки (\u0022);
- \ddd символ у восьмеричному поданні, де кожне d відповідає цифрі від 0 до 7.

Восьмеричні символьні константи можуть складатися із трьох або менше цифр і не можуть перевищувати значення \377 (\u00ff). Символи, представлені в шестнадцатеричном виді, завжди повинні складатися із чотирьох цифр.

3.7.3. Оголошення змінних

При оголошенні змінних вказується тип, рівень доступу й інші атрибути ідентифікатора. Оголошення складається із трьох частин: спочатку наводиться список модифікаторів, за ним тип, і на завершення список ідентифікаторів (аналогічно C++). Локальні змінні можуть оголошуватися без модифікаторів, наприклад:

```
float[] x, y;
```

Поля з модифікатором final повинні ініціалізуватися при оголошенні.

3.7.4. Масиви

Робота з масивами в Java аналогічно C++. Елементи масиву можуть мати примітивний тип або бути посиланнями на об'єкти, у тому числі й посиланнями на інші масиви. Масив оголошується в такий спосіб:

```
int[] ia = new int[3];
```

у цьому випадку масив має три елементи.

Розмір масиву можна одержати з поля *length*. Розглянемо приклад:

```
for (int i =0; i < ia.length; i++) System.out.println(i + ": " +
ia[i]);
```

Масиви завжди є неявним розширенням класу *Object*. Можна створювати масиви класів. Розглянемо приклад:

```
 class A{
...
  public A(int a, int b){...... }
A[] mas=new A[10];
 for(int i=0;i<mas.length;i++)
  mas[i]=new A(3,2);
```

В Java можна використовувати багатомірні масиви:

```
float[][] mat = new float[4][4];
...
for (int y = 0; y << mat.length; y++) {
for (int x = 0; x << mat[y].length; x++)
System.out.println(mat[x][y] + " ");
System.out.println();
}
```

Перший (лівий) розмір масиву повинен задаватися при його створенні. Інші розміри можуть вказуватися пізніше:

```
float[][] mat = new float[4][];
for (int y = 0; y < mat.length; y++)
mat[y] = new float[4];
```

3.7.5. Ініціалізація масивів

Щоб ініціалізувати масив, необхідно задати значення його елементів у фігурних дужках після його оголошення. Розглянемо приклад:

```
  String[] dangers = { "Lions", "Tigers", "Bears" };
```

Для багатомірних масивів маємо:

```
double[][] identityMatrix = {
 { 1.0, 0.0, 0.0, 0.0 },
 { 0.0, 1.0, 0.0, 0.0 },
 { 0.0, 0.0, 1.0, 0.0 },
 { 0.0, 0.0, 0.0, 1.0 }
};
```

3.7.6. Явне перетворення типів

В Java можливі явні перетворення типів. Розглянемо приклад:

```
 double d=5.6;
 long l=(double) d;
```

Розглянемо приведення типів у класах:

```
 class A{...};
 class B extends A{...};
 class test{
  public static void main(String[] args){
   A obj=new B();
   if(obj instanceof B) B bobj=(B) obj;
 //Якщо приведення спрацює, то bobj вказує
 //туди ж, куди й obj, у противному випадку буде
 //викинуте виключення ClassCastException
... }}
```

Перетворення типів впливають на перевантаження методів. Розглянемо приклад. Нехай є ієрархія класів (Рис.3.1):

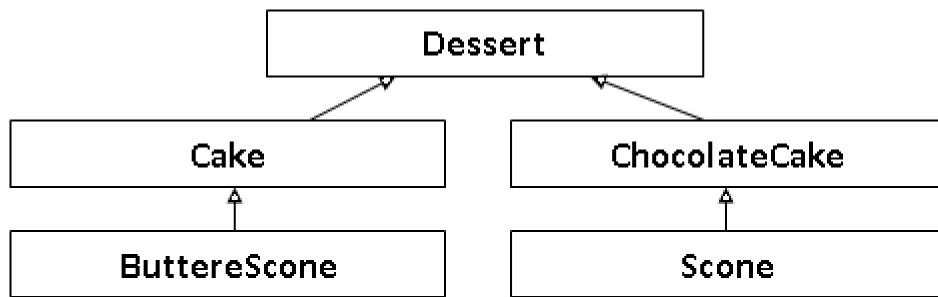


Рис.3.1. Діаграмма класів для приклада

Припустимо, є кілька перевантажених методів:

```
void f(Dessert d, Scone s){...};
void f(Cake c, Dessert d){...};
void f(ChocolateCake cc, Scone s){...};
```

Розглянемо виклики:

```
f(dessertRef,sconeRef);//викликається void
f(Dessert d, Scone s);
f(chocolateCakeRef, dessertRef);//викликається
void f(Cake c, Dessert d){...};
f(chocolateCakeRef, butteredsconeRef);
//викликається void f(ChocolateCake cc, Scone s);
f(cakeRef,sconeRef);//error.
```

3.7.7. Арифметичні оператори

Java підтримує сім арифметичних операторів, які працюють із будь-якими числовими типами:

+ додавання

- віднімання

* множення

/ ділення

% залишок

Java також підтримує унарний мінус (-) для зміни знака числа. Знак може бути змінений оператором наступного виду:

```
val = -val;
```

Крім того, є й унарний плюс — наприклад, +3. Унарний плюс включено для симетрії, без нього було б неможливо записувати константи виду +2.0.

3.7.8. Побітові оператори

& - побітове ТА;

| - побітове АБО;

^ - операція XOR;

<< - зсув бітів вліво із заповненням позицій праворуч нулями;

>> - зсув бітів вліво із заповненням позицій ліворуч значенням старшого (знакового) біта;

>>> - зсув бітів вправо із заповненням позицій ліворуч нулями.

РОЗДІЛ 4
4.1. ОПЕРАТОР IF-ELSE

В Java є стандартний набір операторів відношення і умовних операторів:

> більше;

>= більше або рівно;

< менше;

<= менше або рівно;

== рівно;

!= не рівно.

Усі ці оператори повертають логічні значення. Унарний оператор ! інвертує логічну величину, і, отже, *!true* — це те ж саме, що й *false*. Перевірка логічних величин звичайно проводиться безпосередньо — якщо *x* і *y* відносяться до логічного типу, то можна написати:

```
if (x || !y) {...}
```

ніж

```
if (x == true || y == false) {...}.
```

Результати логічних операцій можуть поєднуватися за допомогою операторів && і ||, які означають "логічне И" і "логічне АБО" відповідно. По можливості ці оператори намагаються обійтися без обчислення свого другого операнда. Наприклад:

```
if (w && x) {// зовнішнє "якщо"
  if (y || z) {// внутрішнє "якщо"
    // ... // тіло внутрішнього "якщо"
}}
```

Внутрішнє "якщо" виконується лише в тому випадку, якщо оба значення (*w* і *x*) рівні *true*. Якщо *w* рівно *false*, то *x* не обчислюється. Тіло внутрішнього "якщо" виконується, якщо хоча б одне зі значень, *y* або *z*, рівно *true*. Якщо *y* рівно *true*, *z* не обчислюється.

Це правило часте використовується в багатьох програмах на Java для забезпечення правильності або ефективності роботи. Наприклад, даний метод робить безпечним наступний фрагмент:

```
if (ix >>= 0 && ix << array.length && array[ix] != 0) { ...}
```

Спочатку виконується перевірка діапазону — тільки в тому випадку, якщо змінна *x* лежить у припустимих межах, вона буде використана для доступу до елемента масиву array.

Два посилання на об'єкт завжди можуть перевірятися на рівність. Вираження *ref1==ref2* рівно *true*, якщо обидва посилання вказують на той самий об'єкт або обидва вони рівні *null*, навіть якщо посилання відносяться до різних оголошених типів. А якщо ні, то вертається *false*.

З об'єктами *String* оператори рівності працюють не так, як можна було б очікувати. Для заданих об'єктів *str1* і *str2*, що відносятся до типу *String*, вираз *str1==str2* перевіряє, чи вказують ці два посилання на той самий об'єкт *String*. Оператор == не перевіряє, чи збігається вміст цих рядків. Рівність вмісту визначається методом *String.equals*.

Умовний оператор дозволяє всього в одному виразі обрати одне із двох значень на підставі логічної змінної. Наприклад:

```
value = (usersetit ? usersvalue : defaultvalue);
```

рівнозначно наступному:

```
if (usersetit) value = usersvalue;else value = defaultvalue;
```

Головна відмінність між операторами *if* і *?:* полягає в тому, що останній має власне значення.

4.2. ОПЕРАТОР SWITCH

Оператор *switch* має вигляд:

```
 switch (цілочисельний вираз){
   case мітка1: оператор1,оператор2,...,break;
   case мітка2: оператор1,оператор2,...,break;
   case мітка3: оператор1,оператор2,...,break;
   ...
   default:...
}
```

Даний оператор повністю аналогічний умовному операторові з C++.

4.3. ЦИКЛИ В JAVA

1. **Цикл while** виглядає в такий спосіб:

```
while (логічний вираз)
           {оператори;}
```

На початку роботи оператора обчислюється логічний вираз, і якщо воно рівно *true*, то виконується оператор (який, зрозуміло, може являти собою блок); це відбувається доти, поки логічний вираз не стане рівним *false*.

Цикл *while* виконується нуль або більше раз, оскільки логічний вираз може виявитися рівним *false* при його першому обчисленні.

Іноді буває потрібно, щоб тіло циклу свідомо було виконано хоча б один раз, і тому в мові Java також передбачена конструкція *do-while*:

```
do-while:
do {оператори;}
  while (логічний вираз);
```

Тут логічний виражз обчислюється після оператора. Цикл виконується, поки вираз залишається рівним *true*. Оператор, що є тілом циклу *do-while*, майже завжди являє собою блок.

2. **Цикл for.**

Оператор *for* використовується для виконання циклу за значеннями з певного діапазону. Він виглядає в такий спосіб:

```
for (ініціалізація; логічний вираз; збільшення)
   оператор
```

Такий запис еквівалентний:

```
{
     ініціалізація;
       while (логічний вираз) {
               оператори;
               збільшення;
        }
 }
```

за тим виключенням, що збільшення завжди виконується, якщо в тілі циклу зустрічається оператор continue (див. розділ “Оператор continue”).

Звичайне застосування циклу *for* — почергове присвоєння змінної значень із деякого діапазону, поки не буде досягнутий кінець цього діапазону.

Вираз ініціалізації та збільшення в циклі for можуть являти собою список значень, що відокремлюються комою. Обчислення цих виразів, як і в інших операторах, відбувається з ліва на право. Наприклад, щоб за допомогою двох індексів перебрати всі елементи масиву в протилежних напрямках, можна скористатися наступним виразом:

```
for ( i = 0, j = arr.length - 1; j >>= 0; i++, j--) {...}
```

Користувач сам вибирає діапазон значень змінної циклу. Наприклад, цикл *for* часто застосовується для перебору елементів зв'язного списку або значень, що входять у математичну послідовність.

Усі вирази в заголовку циклу *for* є необов'язковими. Якщо пропустити ініціалізацію або збільшення, то відповідна частина циклу просто не виконується. Відсутній логічний вираз вважається завжди рівним *true*. Отже, для створення нескінченного циклу можна скористатися наступним записом:

```
for (; ; )
     оператори;
```

Мається на увазі, що цикл буде перерваний іншими засобами — скажімо, описаним нижче оператором *break* або ініціалізацією виключення.

4.4. ОПЕРАТОР BREAK

Оператор *break* застосовується для виходу з будь-якого блоку. Існує різновид оператора *break* з міткою. Розглянемо приклад:

```
private float[][] Matix;
public boolean workOnFlag(float flag) {
 int y, x;
 boolean found = false;
 search: //search це мітка
 for (y = 0; y < Matrix.length; y++) {
   for (x = 0; x < Matrix[y].length; x++) {
      if (Matrix[y][x] == flag) {
           found = true;
           break search; } } } //оператор search з міткою.
  if (!found) return false;
            return true;}
```

4.5. ОПЕРАТОР CONTINUE

Оператор *continue* здійснює перехід у кінець тіла циклу й обчислює значення керуючого логічного виразу. Цей оператор часто використовується для пропуску деяких значень у діапазоні циклу, які повинні ігноруватися.

```
int i=0;
while (i<10) {
...
 if (i==5) continue;
//у випадку якщо умова i==5 істина, оператори нижче
виконуватися не будуть
}
```

Існує оператор *continue* з міткою.

4.6. ОПЕРАТОР RETURN

Оператор *return* завершує виконання методу й передає керування в місце його виклику. Якщо метод не повертає ніякого значення, достатньо написати:

```
 return;
```

Оператор *goto* у Java відсутній.

4.7. ВИКЛЮЧЕННЯ

Виключення у Java являють собою об'єкти. Всі типи виключень (тобто всі класи, об'єкти яких ініціалізуються як виключення) повинні розширювати клас мови Java, що називається *Throwable*, або один з його підкласів. Однак, за згодою, нові типи виключень розширюють клас *Exception*, що є спадкоємцем *Throwable*.

Розглянемо приклад виключення:

```
public class List_full extends Exception{
  private String attrName;
  public List_full(String n){attrName=n;}
  public String getMessage(){
  return attrName;}
 }
```

4.8. ОПЕРАТОР THROW

Виключення генерується інструкцією *throw*:

```
throw вираз;
```

Розглянемо приклад:

```
class List{
   Node* first;
   int size;
   int max;
 class Node{
      int value;
      Node* next;}
 ...
 public void add(int el) throws List_full {
  ...
   if(size>max) throw new List_full("List is full");
 ...
} };
```

4.9. УМОВА THROWS

Виключення, що перевіряються і викидаються методом, оголошуються в умові *throws*, що може містити список значень, відокремлюваних друг від друга комами:

```
  public void f() throws Exception1,Exception2
     {.....};
```

При перевизначенні успадкованого методу або реалізації методу абстрактного класу необхідно забезпечити сумісність пропозиції методу що буде перевизначене.

Існує наступне правило - при перевизначені або реалізації не дозволяється задавати в пропозиції throws нового методу більшу кількість виключень, що об'являються, а ніж у вихідному коді.

Якщо в оголошенні методу стоїть native, то в цьому методі теж можна повідомляти виключення, однак реалізація таких native - методів перебуває поза компетенцією компілятора Java.

4.5. ОПЕРАТОРИ TRY, CATCH TA FINALLY

Щоб перехопити виключення, необхідно помістити фрагмент програми в оператор *try*. Базовий синтаксис оператора *try* виглядає в такий спосіб:

```
 try{
...}
catch ( тип-виключення ідентифікатор)
{...}
catch ( тип-виключення ідентифікатор)
{...}
finally {...}
```

Розглянемо приклад:

```
  public void f(){
   List* l=new List(...);
try{
   l.add(10);
}
 catch(List_full e){
   System.out.println(e.getMessage());}
}
```

Пропозиція *finally* оператора *try* дозволяє виконати деякий фрагмент програми незалежно від того, відбулося виключення чи ні. Звичайно робота такого фрагмента зводиться до "очищення" внутрішнього стану об'єкта або звільненню "необ'єктних" ресурсів (наприклад, відкритих файлів), що зберігаються в локальні змінні.

Розглянемо приклад:

```
public boolean searchFor(String file, String word) throws
StreamException{
Stream input = null;
try {
   input = new Stream(file);
   while (!input.eof())
   if (input.next() == word) return true;
     return false;}
   finally {
     if (input != null) input.close();
 }}
```

Якщо створення об'єкта оператором *new* закінчиться невдало, то *input* збереже своє вихідне значення *null*. Якщо ж виконання *new* буде успішним, то *input* буде містити посилання на об'єкт, що відповідає відкритому файлу. Під час виконання умови *finally* потік *input* буде закриватися лише в тому випадку, якщо він попередньо був відкритий. Незалежно від того, чи виникло виключення при роботі з потоком чи ні, умова *finally* забезпечує закриття файлу.

4.7. ВКЛАДЕНІ КЛАСИ ТА ІНТЕРФЕЙСИ

4.7.1. Статичні вкладені класи

Вкладений клас або інтерфейс, оголошений у вигляді статичного члена зовнішнього класу поводиться точно так, як і звичайний зовнішній клас. Ім'я вкладеного типу задається у вигляді:

```
Им’я_зовнішнього_типу.Им’я_вложеного_типу
```

Розглянемо приклад:

```
public class BankAccount  {
 private long number;
 private long balance;
 public static class Permissions  {
  public boolean canDeposit, canwithdraw, canclose;
}
...}
```

Створити об'єкта класу *Permissions* можна в такий спосіб:

```
 BankAccount.Permissions perm=new BankAccount.Permissions();
```

У вкладених статичних класів можуть бути будь-які модифікатори доступу: *public*, *protected*, *private*.

Не існує будь-яких обмежень, пов'язаних з можливістю розширення вкладеного статичного класу - клас може бути успадкований будь-яким іншим класом, що володіє необхідними правами доступу. При цьому, зрозуміло, похідний клас не здатний успадкувати ті привілеї доступу до членів зовнішнього класу, якими наділений вкладений клас.

Вкладені інтерфейси завжди "статичні", хоча відповідний модифікатор static може бути опущений.

4.7.2. Нестатичні вкладені класи

Нестатичні вкладені класи прийнято називати внутрішніми класами. Об'єкт внутрішнього класу завжди асоціюється з відповідним об'єктом зовнішнього класу. Розглянемо приклад:

```
public class BankAccount {
 private long number;
 private long balance;
 private Action lastAct;
```

```
public class Action  {
   private String act;
   private long amount;
   Action(String act,   long amount)  {
   this.act = act;
   this.amount = amount;
  }
  public String toString {
   return number + ": " + act + " " + amount;
  }}
  public void deposit(long amount) {
   balance += amount;
   lastAct = new Action("прихід", amount);
 }
 public void withdraw(long amount) { balance -= amount;
   lastAct = new Action("расход", amount);
}}
```

Об'єкт внутрішнього класу за замовчуванням одержує у своє розпорядження посилання *this* на поточний зовнішній об'єкт. У методі *deposit* рядок, у якому створюється об'єкт *Action*, можна переписати у вигляді:

```
lastAct = this.new Action("прихід", amount);
```

Місце this у такому разі може зайняти, якщо це необхідно, посилання на будь-який інший об'єкт класу BankAccount. Розглянемо приклад:

```
public void transfer(BankAccount other,  long amount)  {
  other.withdraw(amount) ;
  deposit(amount);
  lastAct = this.new Action("перевод",   amount);
  other.lastAct = other.new Action("перевод",  amount); }
```

Варто відзначити, що повне посилання на поточний об'єкт зовнішнього класу *BankAccount* має вигляд:

```
BankAccount.this.number=10;
```

4.7.3. Розширення внутрішніх класів

Внутрішні класи допускають успадкування. Єдина вимога полягає в тому, що об'єкти розширеного класу повинні зберігати зв'язок з об'єктами вихідного зовнішнього класу або класу, похідного від нього. Наприклад:

```
class Outer { // зовнішній  клас
    class  inner {....}  // внутрішній  клас
 }
class ExtendedOuter extends Outer {
```

```
// Розширений зовнішній клас class Extendedlnner extends inner
{....}
// розширений внутрішній клас
 public inner ref = new Extendedlnner();
}
```

Поле *ref* ініціалізується при створенні об'єкта класу *ExtendedOuter*. У ході процесу побудови екземпляра класу *Extendedlnner* викликається його конструктор за замовчуванням без параметрів, що, у свою чергу, за допомогою посилання *super* неявно звертається до конструктора за замовчуванням класу *Inner*. Конструктор *Inner* вимагає наявності об'єкта *Outer*, до якого треба "прив'язатися", - у цій ролі виступає поточний об'єкт класу *ExtendedOuter*.

Якщо зовнішнім стосовно розширеного внутрішнього класу служить інший клас, не похідний від *Outer*, або якщо внутрішній клас у результаті розширення перестає бути внутрішнім, для забезпечення коректності виклику конструктора класу *Inner* за допомогою посилання *super* повинна бути надана додаткове явне посилання на об'єкт *Outer*. Наприклад:

```
 class Unrelated extends Outer.inner {
 public Unrelated(Outer  ref)  {
    ref.super();
  }}
```

Варто відзначити, що не можна користуватися синтаксисом побудови внутрішнього класу для створення необхідного об'єкта класа *Outer*:

```
   Outer  ref = new Outer();
   Unrelated u = ref.new Unrelated();  //error
```

При звертанні у вкладених класах до полів базових класів їх варто постачати посиланням *this* або *super*. Розглянемо приклад:

```
 class A{
   int x;
  class B extends C{
     void increment(){x++;}//error, правильно this.x++;
}}
```

4.7.4. Локальні внутрішні класи

У Java дозволяється повідомляти вкладені класи усередині блоків коду, таких як тіло методу, конструктора тощо. Єдиний модифікатор, що дозволений уживати в оголошенні локального класу - це *final*.

Код локального внутрішнього класу має право доступу до усіх змінних, які належать контексту того ж блоку - локальним змінним, параметрам методу тощо. Єдине обмеження - локальна змінна або параметр методу стають доступними для локального класу, якщо вони позначені як *final*. Розглянемо приклад

```
public static Iterator walkThrough(final  Object[]   objs)   {
class  iter implements  iterator {
    private int pos = 0;
    public boolean hasNext() { return(pos < objs.length);}
    public Object next() throws NoSuchElementException {
    if (pos >= objs.length)
     throw new  NoSuchElementException();
     return objs[pos++];
    }
    public void remove() {
     throw new UnsupportedOperationException();
 }}
return new lter();
}
```

4.7.5. Анонімні внутрішні класи

Анонімний внутрішній клас - це клас без імені. Розглянемо приклад:

```
public static iterator walkThrough(final  Object[]  objs){
  return new lterator()  { //далі йде опис анонімного класу
  private int pos = 0;
  public boolean hasNext()  {return (pos < objs.length); }
  public Object next()  throws NoSuchElementException  {
   if (pos >= objs.length)
   throw new NoSuchElementException();
   return objs[pos++]; }
  public void  remove()   {
   throw new UnsupportedOperationException(); }
 }}
```

4.7.6. Вкладеність в інтерфейсах

Можна використовувати вкладені класи усередині інтерфейсів. Розглянемо приклад:

```
  interface A{
     class B{
        private int a=10;
    ...
     }  ...}
```

За допомогою вкладених класів можна, в інтерфейсі, створити поле, що допускає зміну. Розглянемо приклад:

```
interface A{
    class B{
      private int x=0;
      public int get(){return x;}
      public void set(int n){x=n;}
    }
 B obj=new B();
}
```

У будь-якому коді, що реалізує інтерфейс A можливий спільний доступ до даних за допомогою *obj*.

РОЗДІЛ 5
5.1. ШАБЛОНИ В JAVA

В Java можна використовувати шаблони класів. Розглянемо приклад класу - шаблона:

```
 public class Pair<T, S> {
     public Pair(T f, S s) { first = f; second = s; }
     public T getFirst() { return first; }
     public S getSecond() { return second; }
     public String toString() {
       return "(" + first.toString() + ", " +
       second.toString() + ")"; }
    private T first;
    private S second;
}
```

Використання даного класу має вигляд:

```
…
Pair<String, String> grade440 = new Pair <String,
String>("mike", "A");
Pair<String, Integer> marks440 = new Pair <String,
Integer>("mike", 100); System.out.println("grade:" +
grade440.toString()); System.out.println("marks:" +
marks440.toString());
…
```

Однак варто відмитити, що конкретизація шаблона примітивним типом в Java неможлива, тобто:

```
Pair<int, int> grade440 = new Pair <int, int>(5, 10);
//error
```

5.2. ПОТОКИ В JAVA

Потоки в Java дозволяють розпаралелити виконання програми. Потік може працювати незалежно від інших потоків.

5.2.1. Створення потоків

Потоки представлені класом у стандартних бібліотеках Java. Щоб створити новий потік виконання, необхідно створити об'єкт *Thread*:

```
   Thread worker = new Thread();
```

Після того як об'єкт-потік буде створений, можна задати його конфігурацію й запустити.

У поняття конфігурації потоку входить вказівка вихідного пріоритету, ім'я тощо. Коли потік готовий до роботи, варто викликати його метод *start*. Метод *start* породжує новий виконуваний потік на основі даних об'єкта класу *Thread*, після чого завершується. Метод *start* також викликає метод *run* нового потоку, що приводить до активізації останнього.

Вихід з методу *run* означає припинення роботи потоку. Потік можна завершити і явно, за допомогою виклику *stop* його виконання може бути припинено методом *suspend*.

Стандартна реалізація *Thread.run* не робить нічого. Тому необхідно або розширити клас *Thread*, щоб включити в нього новий метод *run*, або створити об'єкт *Runnable* і передати його конструкторові потоку.

Насамперед розглянемо розширення класу *Thread*. Розглянемо приклад, що створює два потоки, які виводять слова "*ping*" і "*PONG*" з різною частотою.

```
class PingPong extends Thread {
  String word;
  int delay; // тривалість паузи
  PingPong(String whatToSay, int delayTime) {
   word = whatToSay;
   delay = delayTime;
  }
 public void run() {
    try {
      for (;;) {
     System.out.print(word + " ");
     sleep(delay);
 }}
 catch (InterruptedException e) { return; } }
 public static void main(String[] args) {
 new PingPong("ping", 33).start(); // 1/30 секунди
 new PingPong("PONG", 100).start(); // 1/10 секунди
}}
```

Тут метод run працює в нескінченному циклі, виводячи вміст поля *word* і роблячи паузу на *delay* мікросекунд. Метод *PingPong.run* не може ініціювати виключень, оскільки цього не робить переобумовлений їм метод *Thread.run*. Відповідно, необхідно перехопити виключення *InterruptedException*, що може ініціалізуватися методом *sleep*.

Розглянемо інший метод створення потоку - створення об'єкта *Runnable* і передачі його конструкторові потоку.

5.2.2. Використання об'єктів *Runnable*

Потік служить абстракцією поняття виконавця — суб'єкта, здатного до виконання яких-небудь корисних дій. План роботи, що підлягає виконанню, описується за допомогою інструкцій методу *run*. Щоб якась мета була досягнута, необхідні виконавець і план роботи: інтерфейс *Runnable* абстрагує поняття роботи й дозволяє призначити її виконавцеві - об'єкту потоку. У складі інтерфейсу *Runnable* оголошений єдиний метод:

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

Клас *Thread* реалізує інтерфейс *Runnable*, оскільки потік сам по собі здатний визначати план роботи, що підлягає виконанню.

Реалізація інтерфейсу *Runnable* у багатьох випадках представляється більше простим рішенням, чим розширення класу *Thread*.

Розглянемо той же самий приклад, але тут напишемо реалізацію інтерфейсу *Runnable*:

```
public class RunPingPong implements Runnable{
     private String word;
     private int delay;
  RunPingPong(string whatToSay, int delayTime)
    { word = whatToSay; delay = delayTime;}
  public void  run() {
    try {
      for  (;;){
      System.out.print(word + " ");
      Thread.sleep(delay);} }
 catch (interruptedException e) {
    return;  }}
 public static void main(String[] args){
    Runnable ping = new RunPingPong(“ping”, 33);
    Runnable pong = new RunPingPong(“PONG”, 100);
    new Thread(ping).start();
    new Thread(pong).start(); }
```

Існує чотири перевантажені версії конструктора класу *Thread*, що дозволяють передати як параметр об'єкт *Runnable*:

- **public Thread(Runnable target)** - створює новий об'єкт *Thread*, що використовує метод run об'єкта *target*.
- **public Thread(Runnable target, String name)** - створює новий об'єкт *Thread* з заданим ім'ям *name*, що використовує метод *run* об'єкта *target*.
- **public Thread(ThreadGroup group, Runnable target)** - створює новий об'єкт *Thread* у зазначеному об'єкті *ThreadGroup*, що використовує метод *run* об'єкта *target*.
- **public Thread(ThreadGroup group, Runnable target, String name)** - створює новий об'єкт *Thread* із заданим ім'ям *name* у зазначеному об'єкті *ThreadGroup*, що використовує метод *run* об'єкта *target*.

5.2.3. Синхронізація

Коли два потоки повинні скористатися тим самим об'єктом, виникає небезпека, що накладення операцій приведе до руйнування даних. Тому потоки повинні синхронізувати свій доступ до критичних секцій. Для цього в умовах багатопоточності використовується блокування об'єкта. Коли об'єкт заблокований деяким потоком, тільки цей потік може працювати з ним.

5.2.4. Методи *synchronized*

Щоб клас міг використовуватися в багатопоточному середовищу, необхідно оголосити відповідні методи з атрибутом *synchronized*. Якщо деякий потік викликає метод *synchronized*, то відбувається блокування об'єкта (Рис.5.1.). Виклик методу *synchronized* того ж об'єкта іншим потоком буде припинений до зняття блокування.

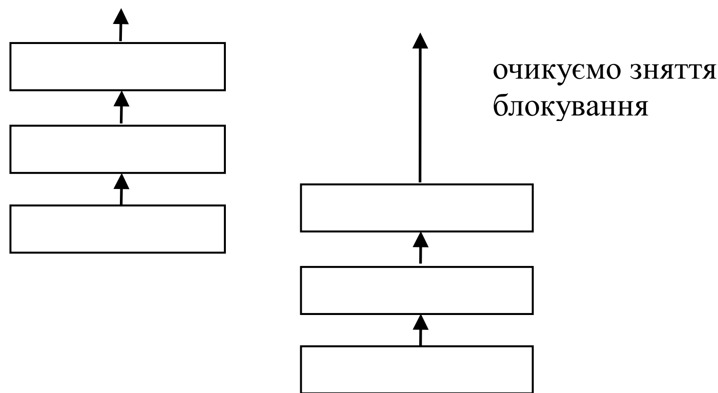


Рис.5.1. Синхронізація потоків

Синхронізація приводить до того, що виконання двох потоків стає взаємно виключаючим за часом. Розглянемо приклад:

```
class Account {
   private double balance;
  //дане поле захищене від будь-яких несинхронних дій
   public Account(double initialDeposit) {
   balance = initialDeposit; }
   public synchronized double getBalance()
   { return balance; }
   public synchronized void deposit(double amount) {
   balance += amount;
}}
```

Конструктор не зобов'язаний бути *synchronized*, оскільки він виконується тільки при створенні об'єкта, а це може відбуватися тільки в одному потоці для кожного знову створюваного об'єкта.

Два потоки не можуть одночасно виконувати синхронізовані статичні методи одного класу. Блокування статичного методу на рівні класу не відбиваються на об'єктах останнього - можна викликати синхронізований метод для об'єкта, поки інший потік заблокував весь клас у синхронізованому статичному методі. В останньому випадку блокуються тільки синхронізовані статичні методи.

Якщо синхронізований метод перевизначається в розширеному класі, то новий метод не зобов'язаний бути синхронізованим. Метод суперкласу при цьому залишається синхронізованим, так що несинхронність методу в розширеному класі не скасовує його синхронізованого поводження в суперкласі. Якщо в несинхронізованому методі використовується

конструкція *super.method()* для звертання до методу суперкласу, то об'єкт блокується на час виклику до виходу з методу суперкласу.

5.2.5. Оператори *synchronized*

Оператор *synchronized* дозволяє виконати синхронізований фрагмент програми, що здійснює блокування об'єкта, не жадаючи від програміста виклику синхронізованого методу для даного об'єкта. Оператор *synchronized* складається із двох частин: вказівки блокируемого об'єкта й оператора, виконуваного після одержання блокування. Загальна форма оператора *synchronized* виглядає у такий спосіб:

```
synchronized (вираження){
оператори}
```

Розглянемо приклад:

```
public static void abs(int[] values) {
  synchronized (values) {
  //доступ до масиву values блокується з боку інших потоків
  for (int i = 0; i < values.length; i++) {
  if (values[i] < 0)
  values[i] = - values[i]; }
}}
```

Оператори *synchronized* мають переваги перед *synchronized* методами:

- вони дають можливість визначення синхронізованої ділянки коду, що охоплює тільки деякий фрагмент тіла методу.
- *synchronized* - оператори дозволяють синхронізувати об'єкти, відмінні від *this*, і дають можливість створювати найрізноманітніші схеми синхронізації.

Одна з ситуацій, що досить часто зустрічаються, пов'язана з необхідністю забезпечення вищого рівня інтенсивності конкурентного доступу до коду класу за рахунок зменшення розмірів блокованих областей коду. Розглянемо приклад:

```
class SeparateGroups {
   private double aval =0.0;
   private double bval = 1.1;
   protected object lock = new object();
   protected Object lock = new Object;
```

```
   public double get() {
    synchronized (lock) { return aval;} }
   public void set(double val)  {
    synchronized  (lock)  { aval  = val;} }
   public double get(){
    synchronized  (lock)  { return bval;} }
   public void set(double val)  { synchronized  (lock){
     bval  = val;
}}
   public void reset(){
    synchronized (lock) {
    synchronized (lock) {
    aval = bval =0.0;
}}}}
```

Ще одна із часто виникаючих ситуацій, у яких зручно використовувати інструкції *synchronized*, пов'язана з необхідністю синхронізації коду зовнішнього об'єкта при звертанні до нього із внутрішнього об'єкта:

```
public class Outer {
   private int data;
   private class inner {
     void setOuterData(){
       synchronized  (Outer.this){data = 12;}
}}}
```

Для завдання блокируемого об'єкта можна використовувати літерал типу *class* для поточного класу. Такий підхід застосуємо й у тому випадку, якщо слід запобігти доступу до статичних даних з боку нестатичного коду. Розглянемо приклад:

```
class Body {
   public long idNum;
   public String nameFor;
   public Body orbits;
   public static long nextID = 0;
 Body(){
  synchronizes(Body.class){
    idNum=nextID++;}
}}
```

5.2.6. Методи *wait, notifyall* і *notify*

Механізм блокування вирішує проблему з можливим впливом декількох потоків, однак хотілося б, щоб потоки могли обмінюватися інформацією один з одним.

Із цією метою застосовуються метод очікування *wait()*, що дозволяє призупинити виконання потоку до того моменту поки не буде виконана певна умова й методи оповіщення *notify* і *notifyall*.

Існує стандартний зразок коду, який важливо враховувати при використанні методів очікування та оповіщення. Потоку, що очікує виконання деякої умови, завжди слід виконувати дії, подібні таким:

```
synchronized void dowhenCondition(){
    while(!умова) wait()}
//виконати те, що необхідно, якщо умова дорівнює true }
```

При використанні методів очікування та оповіщення варто дотримуватися наступних правилам.

1. Всі функції по забезпеченню взаємодії потоків повинні виконуватися в рамках синхронізованого коду. Якщо ця вимога не задовольняється, стан об'єкта не може вважатися стабільним. Наприклад, якщо метод не оголошений як *synchronized*, після виконання блоку *while* не можна твердо гарантувати, що перевіряєма умова залишиться рівна *true*, оскільки інший потік може змінити ситуацію.
2. Один з важливих аспектів методу *wait* полягає в тому, що при припиненні виконання потоку він атомарним образом звільняє блокування об'єкта.
3. Умова очікування повинна завжди перевірятися циклічно. Не досить перевірити її тільки один раз, - після того як умова виконана, воно може змінитися знову. Інакше кажучи, не можна заміняти *while* на *if*.

Методи оповіщення, у свою чергу, викликаються синхронізованим кодом і змінюють одну або кілька умов дозволу, яких можуть очікувати будь-які інші потоки. Код оповіщення звичайно виглядає наступним чином:

```
synchronized void changeCondition(){
//змінити деяке значення, використовуване у виразіі умови
очікування
notifyAll(); // або  notify();
}
```

Метод *notifyAll* сповіщає всі потоки, що очікують, а *notify* вибирає для цього тільки один потік.

Потоки можуть очікувати виконання умов (можливо, різних), що стосуються одного й того самого об'єкта. Якщо умови дійсно різні, для оповіщення всіх потоків, що очікують, варто завжди використовувати метод *notifyAll* замість *notify*.

Метод *notify* дозволяє трохи підвищити ефективність коду й може застосовуватися тільки в тих випадках, коли:

- всі потоки очікують виконання тієї самої умови;
- саме більше один потік набуде переваги через виконання умови;

У всіх інших ситуаціях слід використовувати *notifyAll*. Розглянемо кілька прикладів.

Приклад 1.

```
class Cell {//  Елемент черги
   Cell   next;
   Object item;
   Cell(Object item)   {this.item  = item;} }
class Queue {
private Cell head, tail;
//елементи в "голові" і "хвості" черги
public synchronized void add(Object o) {
  Cell p = new Cell(o);
  if (tail == null)
   head = p;
  else
   tail.next = p;
   p.next = null;
   tail = p;
  notifyAll();//  у чергу доданий елемент }
public synchronized object take()
throws interruptedException{
while(head==null){
 wait(); // Чекати повідомлення про додавання елемента
 Cell p = head;
//запам'ятати елемент, що займає місце в "голові" черги
 head = head.next; // Видалити елемент із "голови" черги
 if (head == null);// перевірити, чи не порожня черга
 tail  = null;
 return p.item;}}
```

Приклад 2.

```
class MyThread implements Runnable
 {
   private static long time1=0;
   private static Object obj=new Object();
   private int flag;
```

```
   public MyThread(int flag1){
     flag=flag1;}
  synchronized public static void setTime(){
    time1++;
    synchronized(obj){obj.notifyAll();}
 }
synchronized public void write() throws InterruptedException
   {if (flag==0)  {
   System.out.println(time1);}
   if(flag==1){ long time2=time1;
     while(time 1-time2<15)
       {synchronized(obj) {obj.wait();}}
   System.out.println("Hello1"); }
   if(flag==2){ long time2=time1;
      while(time 1-time2<7){
      synchronized(obj){obj.wait();
  }}
  System.out.println("Hello2");
}}
public void run(){
try{
    for(;;){
    if (flag==0){ setTime();}
    write();
    Thread.sleep(1000);}}
catch(InterruptedException e){
    System.out.println("error");
     return;
}}
 public static void main(String[] args){
 Runnable th1=new MyThread(0);
 Runnable th2=new MyThread(1);
 Runnable th3=new MyThread(2);
 new Thread(th1).start();
 new Thread(th2).start();
 new Thread(th3).start();
}}
```

5.2.7. Додаткові зауваження щодо методів *wait, notify* і *notifyall*

Розглянемо докладніше методи *wait*, *notify* і *notifyAll*:

- **public final void wait(long timeout) throws InterruptedException** - виконання потоку припиняється до одержання повідомлення або по закінченню заданого інтервалу часу *timeout*. Значення *timeout* задається в миллисекундах. Якщо воно дорівнює нулю, то очікування не переривається по тайм-ауту, а триває до одержання повідомлення.

- **public final void wait(long timeout, int nanos) throws InterruptedException** - аналог попереднього методу з можливістю більше точного контролю часу; інтервал тайм-ауту являє собою суму двох параметрів: *timeout* (у миллисекундах) і *nanos* (у наносекундах, значення в діапазоні 0-999999).
- **public final void wait() throws InterruptedException** - еквівалентно *wait(0).*
- **public final void notify()** - посилає повідомлення рівно одному потоку, що очікує виконання деякої умови. Потоки, які поновлюються лише після виконання даної умови, можуть викликати одну з різновидів *wait*. При цьому обрати поток, що сповіщається, неможливо, тому дана форма *notify* використовується лише в тих випадках, коли точно відомо, які потоки очікують подій, які це події й скільки триває очікування.
- **public final void notifyAll()** - посилає повідомлення всім потокам, що очікують виконання деякої умови. Звичайно потоки стоять, поки якийсь інший потік не змінить деяку умову. Використовуючи цей метод, керуючий умовою потік сповіщає всі потоки, що очікують, про зміну умови. Потоки, які відновляються лише після виконання даної умови, можуть викликати одну з різновидів *wait*.

Всі ці методи реалізовані в класі *Object*. Проте вони можуть викликатися тільки із синхронізованих фрагментів, з використанням блокування об'єкта, у якому вони застосовуються.

5.2.8. Планування потоків

Відносно потоків в Java даються лише загальні гарантії. Як кількісний показник важливості виконуваних функцій потоку ставиться у відповідність пріоритет (priority), значення якого використовується системою для визначення того, який з потоків повинен виконуватися в кожний момент часу.

У системі з *N* процесорами одночасно може виконуватися *N* високопріоритетних потоків. Потокам, що володіють більше низькими значеннями пріоритету, ресурси процесорів звичайно віддаються тільки в тому випадку, коли більше важливі потоки блоковані. Щоб запобігти ймовірності зависання, система вправі надавати ресурси низькоприоритетним потокам і в інші моменти часу - у зв'язку з так званим старінням пріоритетів (priority aging), - але програми не в змозі серйозно використовувати таку можливість.

Визначення розкладу пріоритетного обслуговування потоків з перериваннями входить у компетенцію конкретної віртуальної машини Java. Найчастіше твердих гарантій поводження системи відносно планування завдань не існує - можна тільки очікувати, що перевага в тім або іншому випадку буде віддано потоку, що володіє більше високим пріоритетом.

Вихідне значення пріоритету потоку відповідає пріоритету потоку-"батька". Величина пріоритету може бути змінена за допомогою виклику методу *setPriority* з аргументом з інтервалу, що задається значеннями іменованих констант *MIN_PRIORITY* і *MAX_PRIORITY*, визначених у складі класу *Thread*.

Пріоритету потоку, пропонованому за замовчуванням, відповідає константа *NORM_PRIORITY*. Пріоритет працюючого потоку допускається змінювати в будь-який момент. Якщо пріоритет потоку знижується, система може передати обчислювальні ресурси іншому потоку, оскільки вихідний втратить "членство" у групі потоків з найвищими пріоритетами. Щоб одержати поточне значення пріоритету потоку, варто скористатися методом *getPriority*.

Існують методи в класі *Thread* які керують плануванням потоків у системі.

- **public static void sleep(long millis) throws InterruptedException** - припиняє роботу потоку як мінімум на зазначене число миллисекунд. “Як

мінімум" означає, що не існує гарантій поновлення роботи потоку точно в зазначений час.

- **public static void sleep(long millis, int nanos) throws InterruptedException** - припиняє роботу потоку як мінімум на зазначене число миллисекунд і додаткове число наносекунд. Значення інтервалу в наносекундах лежить у діапазоні 0-999999.
- **public static void yield()** - потік передає керування, щоб дати можливість працювати й іншим потокам, що виконуються. Планувальник потоків вибирає новий потік серед потоків, що виконуються, у системі. При цьому може бути викликаний потік, що тільки що поступився керування, якщо його пріоритет виявиться найвищим.

Розглянемо приклад.

```
class Babble extends Thread {
// передавати керування іншим потокам?
static boolean doYield;
// кількостей повторів при виводі String word;
static int howOften;
Babble(String whatToSay) { word = whatToSay; }
public void run() {
   for (int i = 0; i < howOften; i++) {
   System.out.println(word);
   if (doYield) yield(); // передати керування іншому потоку }}
public static void main(String[] args) {
   howOften = Integer.parseInt(args[1]);
   doYield = new Boolean(args[0]).booleanValue();
   Thread cur = currentThread();
   cur.setPriority(Thread.MAX_PRIORITY);
   for (int i = 2; i < args.length; i++)
   new Babble(args[i]).start(); }}
```

Коли потоки працюють, не передаючи керування один одному, їм приділяються більші кванти часу - звичайно цього буває досить, щоб закінчити вивід у монопольному режимі. Наприклад, при запуску програми із присвоєнням *doYield* значення *false*:

```
Babble false 2 Did DidNot
```

результат буде виглядати в такий спосіб:

```
Did
Did
DidNot
DidNot
```

Якщо кожний потік передає керування після чергового *println*, то інші потоки також одержать можливість працювати. Якщо присвоти doYield значення true:

```
Babble true 2 Did DidNot
```

то інші потоки також зможуть виконуватися між черговими виводами й, у свою чергу, будуть уступати керування, що приведе (як варіант) до наступного:

```
Did
DidNot
Did
DidNot
```

РОЗДІЛ 6
6.1. ВЗАЄМНЕ БЛОКУВАННЯ ПОТОКІВ

Якщо є два потоки й два об'єкти, що підлягають блокуванню, виникає небезпека виникнення взаємоблокування - кожний з потоків володіє блокуванням одного об'єкта й очікує звільнення іншого об'єкта. Якщо об'єкт *X* володіє *synchronized* методом, що викликає *synchronized* метод об'єкта *Y*, а *Y*, у свою чергу, також має синхронізований метод, що звертається до *synchronized* методу об'єкта *X*, два потоки можуть перебувати в стані очікування взаємного завершення, щоб опанувати блокуванням, і жоден з них не виявиться здатний продовжити роботу. Така ситуація називається клінчем.

Розглянемо приклад:

```
class Friends {
  private Friends partner;
  private String name;
  public Friends(String name){ this.name = name; }
  public synchronized void hug(){
  Systern.out.println(Thread.currentThread(). getName()+"в" +
name + ".hug() намагається  викликати" + partner.name +
".hugBack()") ;
  partner.hugBack();}
  private synchronized void hugBack(){
  System.out.println(Thread.currentThread().getName()+ " в  " +
name + ".hugBack()"); }
  public void becomeFriend(Friends partner)  {
  this.partner = partner;
}}
```

Далі можливий наступний сценарій:

```
public static void main(string[]  args)  {
  final Friends jareth = new Friends("jareth");
  final Friends cory = new Friends("cory");
  jareth.becomeFriend(cory); cory.becomeFriend(jareth);
  new Thread(new Runnable(){
  public void run(){ jareth.hug();}},"thread1").start();
  new Thread(new Runnable(){
  public void run(){ cory.hug();}},"thread2").start();
}
```

У такий спосіб є наступний сценарій:

1. *Thread 1* викликає *synchronized* метод *jareth.hug()*; тепер *thread 1* володіє блокуванням об'єкта *jareth*.

2. *Thread 2* викликає s*ynchronized* метод *cory.hug()*; тепер *thread 2* володіє блокуванням об'єкта cory;
3. *jareth.hug()* викликає *synchronized* метод *cory.hugBack()*; *thread 1* припиняє виконання, переходячи в стадію очікування можливості захвата блокування *cory* (якої в цей момент володіє *thread 2*);
4. *cory.hug()* викликає *synchronized* метод *jareth.hugBack()*; *thread 2* припиняє виконання, переходячи в стадію очікування можливості захвата блокування *jareth* (якої в цей момент володіє *thread 1*).

Програма після запуску встигне вивести:

```
Thread 1 в jareth.hug()  намагається  викликати cory.hugBack()
Thread 2  в cory.hug() намагається  викликати jareth.hugBack()
```

Після програма зависає. Зрозуміло, не виключено, що повезе, і один з потоків зуміє виконати код *hug* цілком ще до моменту старту другого потоку.

6.2. ЗАВЕРШЕННЯ ВИКОНАННЯ ПОТОКУ

Про потік, що приступився до роботи, говорять як про діючий (alive), і метод *isAlive* такого потоку повертає значення *true*. Потік продовжує залишатися діючої доти, поки не буде зупинений у результаті виникнення одного із трьох можливих подій:

- метод *run* завершив виконання нормальним образом;
- робота методу *run* перервана;
- викликано метод *destroy* об'єкта потоку.

Повернення з методу *run* за допомогою *return* або в результаті природного завершення коду - це нормальний спосіб закінчення виконання потоку.

Виклик методу *destroy* об'єкта потоку - це зовсім радикальний крок. У цьому випадку потік "помирає" раптово, незалежно від того, що саме він виконує в цей момент, і не звільняє ні однієї із захоплених блокувань, тому інші потоки можуть залишитися в стані нескінченного очікування.

Багатьма java-машинами метод *destroy* не підтримується, і його виклик тягне викидання виключення типу *NoSuchMethodError*, здатного припинити роботу потоку-ініціатора, а не того потоку, завершення якого передбачалося.

6.3. КОРЕКТНЕ ЗАВЕРШЕННЯ РОБОТИ ПОТОКУ

Може виникнути ситуація, коли потік створюється для досягнення певної мети, а потім його виконання необхідно перервати перш, ніж він вирішить поставлене завдання.

Для завершення потоку викликається метод *interrupt*, і код відповідного потоку повинен сам стежити за подією переривання й відповідати за його виконання.

Розглянемо приклад:

Потік 1:

```
thread2.interrupt();
```

Потік 2:

```
while(! Interrupted()){
// щось робиться
...}
```

Метод *interrupt* сам по собі не примушує потік припиняти свою діяльність, хоча часто перериває сплячий режим або режим очікування потоку, що виконує відповідно функції *sleep* або *wait*.

До механізму переривання роботи потоку мають відносини наступні методи:

- **interrupt()** - посилає потоку повідомлення про переривання;
- **isInterrupted()** - перевіряє, чи була перервана робота потоку викликом методу *interrupt*;
- **interrupted()** - статичний метод, що перевіряє, чи виконувалося переривання потоку, і очищающий "стан переривання" потоку. Останнє може бути очищено тільки самим потоком "зовнішніх" способів скасування повідомлення про переривання, посланого потоку, не існує.

Переривання за допомогою методу *interrupt* звичайно не впливає на працездатність потоку, але деякі методи, такі як *sleep* і *wait*, будучи перерваними, відоброжають виключення типу *interruptedException*.

Інакше кажучи, якщо потік у момент переривання його роботи за допомогою interrupt виконує один із цих методів, вони генерують виключення *InterruptedException*. У цьому випадку стан переривання потоку очищається, тому код, що обробляє виключення *InterruptedException*, звичайно повинен виглядати в такий спосіб:

```
void tick(int count,long pauseTime){
    try {
      for (int i  =0; i< count;i++)  {
        System.out.println('...');
        Thread.sleep(pauseTime);
      }}
    catch(interruptedException e){
      Thread.currentThread().interrupt();
 }}
```

Метод *tick* виводить на екран символ крапки count раз, "засипаючи" після кожної операції на період часу, якій дорівнює значенню *pauseTime*, вираженому в миллисекундах.

Якщо робота потоку переривається за допомогою *interrupt* у момент виконання їм методу *tick*, метод *sleep* викидає виключення типу *InterruptedException*. Керування передається із циклу *for* у пропозицію *catch*, де повідомлення про переривання потоку посилає заново.

6.4. ОЧІКУВАННЯ ЗАВЕРШЕННЯ РОБОТИ ПОТОКУ

Потік здатний чекати завершення роботи іншого потоку, використовуючи один з різновидів методу *join*. Розглянемо приклад:

```
class CalcThread extends Thread {
  private double  result;
  public void run()  { result = calculate(); }
  public double getResult(){return  result; }
  public double calculate() {
   //   ...   обчислити значення поля  result } }
 class  showjoin  {
    public static void main(string[]   args)   {
    CalcThread calc = new CalcThread();
    calc.start() ;
```

```
    ...
    try{
      calc.join();
      System.out.println ("The result is " + calc.getResult());}
    catch (InterruptedException e){
      System.out.println("the thread is interrupted");}}
    ...
}
```

Вихід з *join* означає, що робота методу *CalcThread.run* завершена й значення result може бути використане в поточному потоці. Метод *join* має три форми:

1. **public final void join(long millis) throws InterruptedException** - очікує завершення виконання потоку або закінчення заданого періоду часу, вираженого в миллисекундах, залежно від того, що відбудеться раніше. Нульове значення параметра означає завдання нескінченного проміжку часу. Якщо робота потоку переривається в момент очікування, викидається виключення типу *InterruptedException*;
2. **public final void join(long millis, int nanos) throws InterruptedException** - більш чутлива версія методу. Величина інтервалу очікування складається із двох складових: *millis* (вираженої в миллисекундах) і *nanos* (у наносекундах). Знову, сумарне нульове значення параметра означає нескінченне очікування. Величина *nanos* повинна перебувати в проміжку 0-999999;
3. **public final void join() throws InterruptedException** - метод аналогічний першому варіанту за умови join(0). Внутрішня реалізація методу join може бути виражена в наступних термінах: *while(isAlive()) wait()*.

6.5. ПОТОКИ – ДЕМОНИ

Існують два види потоків - користувальницькі (user) і потоки - демони (daemon). Наявність користувальницьких потоків зберігає додаток у працюючому стані. Коли виконання останнього з користувальницьких потоків завершується, діяльність всіх демонів переривається й додаток фінішує.

Переривання роботи демонів схоже на виклик методу destroy - воно відбувається раптово й не залишає потокам ніяких шансів для виконання завершальних операцій, - тому демони обмежені у виборі функціональних можливостей.

Для додання потоку статусу демона необхідно викликати метод *setDaemon(true)*, визначенний у класі *Thread*. Перевірити приналежність потоку до категорії демонів можна за допомогою методу *isDaemon()*. За замовчуванням статус демона успадковується потоком від потоку-"батька" у момент створення й після старту не може бути змінений; спроба виклику *setDaemon(true)* під час роботи потоку приводить до відображення виключення типу *IllegalThreadStateException*.

Метод *main* за замовчуванням породжує потоки зі статусом користувальницьких.

6.6. КВАЛІФІКАТОР VOLATILE

Мова гарантує, що операції читання й запису будь-яких значень, крім стосовних до типів *long* або *double*, завжди виконуються атомарним образом - відповідна змінна в будь-який момент часу буде містити тільки те значення, що збережено певним потоком, але не якусь суміш результатів декількох різних операцій запису. Однак, атомарний доступ не гарантує, що потік завжди зможе мати саму останню версію значення, збереженого в змінній. Кваліфікатор *volatile* повідомляє компілятору, що значення змінної може бути змінене в непередбачений момент.

Розглянемо приклад:

```
currentvalue = 5;
 for  (;;)   {
  display.showValue(currentvalue);
  Thread.sleep(1000);  // заснути на одну секунду }
```

Якщо метод *showValue* сам по собі не має можливість зміни значення *currentvalue*, компілятор вільний висунути припущення про те, що усередині циклу for це значення можна трактувати як незмінне, і використовувати ту саму константу 5 на кожній ітерації циклу при виклику *showValue*. Але якщо

вміст поля *currentvalue* у ході виконання циклу піддається відновленню за допомогою інших потоків, припущення компілятора виявиться невірним. Тому правильно оголосити змінну *currentvalue*, використовуючи *volatile*:

```
      volatile int currentvalue = 5;
```

6.7. КЛАС THREADGROUP

Потоки можуть поєднуватися в групи потоків по міркуваннях поліпшення керованості й безпеки. Одна група потоків може належати іншій групі, становлячи ієрархію з основною групою на верхньому рівні. Потоки, що ставляться до групи, можуть управлятися одноразово іншими словами можна перервати роботу відразу всіх потоків групи або встановити для них єдине максимальне значення пріоритету виконання.

Об'єкти груп можуть бути використані також для завдання верхньої границі значень пріоритетів потоків, що відносятся до групи. Після виклику методу *setMaxPriority* з передачею йому відповідного найбільшого припустимого значення пріоритету будь-яка спроба завдання значення, що перевищує встановлений поріг, зводиться до підвищення пріоритету потоку тільки до величини максимального рівня. Розглянемо приклад:

```
static synchronized void maxThread(Thread thr, intpriority){
 ThreadGroup grp = thr.getThreadGroup();
 thr.setPriority(priority);
 grp.setMaxPriority(thr.getPriority()  - 1);
}
```

Клас **ThreadGroup** підтримують наступні конструктори й методи:

- **public ThreadGroup(String name)** - створює новий об'єкт класу *ThreadGroup*, що належить тій групі потоків, до якої відносится й потік-"батько". Як і у випадку об'єктів потоків, імена груп не використовуються виконуючою системою безпосередньо, але як параметр *name* ім'я групи може бути передане значення *null*;
- **public ThreadGroup(ThreadGroup parent, String name)** - створює новий об'єкт класу *ThreadGroup* із зазначеним ім'ям *name* у складі "батьківської"

групи потоків *parent*. Якщо в якості *parent* передане значення *null*, викидається виключення типу *NullPointerException*;

- **public final String getName()** - повертає рядок -ім'я поточної групи потоків;
- **public final ThreadGroup getParent()** - повертає посилання на об'єкт "батьківської" групи потоків або *null*, якщо такого немає (останнє можливо тільки для групи потоків верхнього рівня ієрархії);
- **public final void setDaemon(boolean daemon)** - надає поточному об'єкту групи потоків статус приналежності до категорії груп-демонів
- **public final boolean isDaemon()** - повертає статус приналежності поточного об'єкта групи потоків до категорії груп-демонів;
- **public final void setMaxPriority(int maxpri)** - установлює верхню границю пріоритетів виконання для поточної групи потоків;
- **public final int getMaxPriority()** - повертає раніше задане значення верхньої границі пріоритетів виконання для поточної групи потоків;
- **public final boolean parentOf(ThreadGroup g)** - перевіряє, чи є поточна група "батьківської" стосовно групи g або збігається із групою *g*;
- **public final void checkAccess()** - виводе виключення типу *SecurityException*, якщо потоку не дозволено впливати на параметри групи потоків; у противному випадку просто повертає керування;
- **public final void destroy()** -знищує об'єкт групи потоків. Група не повинна містити потоків, інакше метод викидає виключення типу *IllegalThreadStateException*. Якщо в складі групи є інші групи, вони також не повинні містити потоків. Не знищує об'єкти потоків, що належать групі;
- **public int activeCount()** - повертає приблизну кількість діючих (активних) потоків групи, включаючи й ті потоки, які належать вкладеним групам. Кількість не можна вважати точним, оскільки в момент виконання методу воно може змінитися, - одні потоки "вмирають", а інші створюються.

Потік уважається діючої, якщо метод *isAlive* відповідного об'єкта *Thread* повертає значення *true*;

- **public int enumerate(Thread[] threadslnGroup,boolean recurse) -** заповнює масив *threadslnGroup* посиланнями на об'єкти діючих потоків групи, беручи до уваги розмір масиву, і повертає кількість збережених посилань. Якщо значення параметра *recurse* дорівнює *false*, враховуються тільки ті потоки, які належать безпосередньо поточній групі, а в противному випадку - ще й потоки, що ставляться до всіх вкладених груп;
- **public int enumerate(Thread[] threadslnGroup)** - метод аналогічний попередньому за умови *enumerate(threadsInGroup true)*;
- **public int activeGroupCount()** - подібний до методу *activeCount*, але підраховує кількість груп, включаючи вкладені;
- **public int enumerate(ThreadGroup[] groupslnGroup, boolean recurse) -** подібний до відповідного варіанта методу *enumerate* для підрахунку потоків, але заповнює масив *groupslnGroup* посиланнями на об'єкти вкладених груп потоків;
- **public int enumerate(ThreadGroup[] groupslnGroup)** - метод аналогічний попереднього за умови *enumerate(groupslnGroup, true)*;
- **public void uncaughtException(Thread thr, Throwable exc) -** викликається, коли потік *thr* у поточній групі генерує виключення ехс, що далі не обробляється.

У класі *Thread* існує два статичних методи, що дозволяють обробляти дані про групу, до якої належить поточний потік:

- **public static int activeCount()** - повертає кількість діючих потоків у групі, до якої відноситься поточний потік;
- **public static int enumerate(Thread[] threadslnGroup)** - метод аналогічний виклику *enumerate (threadslnGroup)* об'єкта групи, який належить поточний потік.

6.8. МЕТОД STOP()

Виклик методу *stop* приводить до виникнення у відповідному потоці асинхронного виключення типу *ThreadDeath*. Об'єкти типу *ThreadDeath* можуть бути виловлені точно так само, як і інші, а якщо виключення не піддається обробці, наслідку такої бездіяльності згодом приведуть до аварійного завершення роботи потоку.

6.9. ЗМІННІ THREADLOCAL

Клас *ThreadLocal* надає можливість мати єдину логічну змінну, що володіє незалежними значеннями в контексті кожного окремого потоку. У складі об'єкта *ThreadLocal* є методи *set* і *get*, які дозволяють відповідно привласнювати й зчитувати значення змінної для поточного потоку.

Розглянемо приклад:

```
public class Operations  {
  static class user {
    static final  User unknown_user = new user(); };
  boolean canChange(User u){return true; }
  private static ThreadLocal   users = new ThreadLocal()   {
  protected Object initialvalue(){
   return User.unknown_user;} }
  private static  user currentuser(){
    return (User)users.get();}
  public static void setuser(User newuser)  {
   users.set(newuser); }
  public void setvalue(int newvalue)  {
   User user = currentuser();
   if (!canChange(user))  throw new securityException();
   // ... змінити значення ... }
... }
```

Коли потік припиняє існування, значення, установлені для цього потоку в змінних *ThreadLocal*, недосяжні й можуть бути знищені збирачем сміття, якщо які-небудь посилання на них відсутні.

6.10. НАЛАГОДЖЕННЯ ПОТОКІВ

У складі класу *Thread* є кілька методів, які можна використовувати в процесі налагодження коду багатопоточного додатка:

- **public String toString()** - повертає строкове подання вмісту об'єкта потоку, що включає його найменування, значення пріоритету виконання й ім'я відповідної групи потоків;
- **public static void dumpStack()** - виводить на консоль дані трасування для поточного потоку.

  У класі *ThreadGroup* існую також методи для налагодження потків:
- **public String toString()** - повертає строкове значення вмісту об'єкта групи потоків, що включає його найменування й значення пріоритету виконання;
- **public void list()** -виводить на консоль інформацію про об'єкт *ThreadGroup*, що включає результати викликів *toString* для кожного з потоків, що належать групі, і всіх вкладених груп.

РОЗДІЛ 7

7.1. КЛАСИ - ОБОЛОНКИ

Кожному простому типу в Java відповідає клас-оболонка. Класи-оболонки виконують дві основні функції.

Перша складається в підтримці контейнера методів і змінних, що відносяться до певного типу (наприклад, методів перетворення рядків і констант, що задають границі інтервалу припустимих значень). Розглянемо приклад коду, що перевіряє чи можна використовувати *float* замість *double*:

```
double aval  = Math.abs(value);
if (Float.MAX_VALUE >= aval && aval  >= Float.Ml_VALUE)
  return fasterFloatCalc((float) value);
else
  return slowerDoubleCalc(value);
```

Друга функція класів-оболонок полягає в забезпеченні можливості створення об'єктів для зберігання значень простих типів і використання їх у контексті класів, які "вміють" звертатися тільки з посиланнями на об'єкти типу *Object*.

Клас-оболонка визначає незмінний об'єкт для подання значення відповідного типу. Після створення об'єкта, значення, що у ньому зберігається, змінити не можна. Так, наприклад, об'єкт, створений у результаті виконання інструкції *new Integer(1)*, буде завжди містити значення 1 - у складі класу *Integer* немає методів, які дозволили б його замінити.

Перелічимо деякі класи-оболонки:

1. Number - це абстрактний клас - оболонка, наслідуваний всіма класами – оболонками *Short*, *Byte*, *Integer*, *Long*, *Float*, *Double*;

2.Short;

3. Byte;

4. Integer;

5.Long;

6. Float;

7. Double;

8. Boolean;

9. Character;

10. Void.

У кожному класі-оболонці визначені наступні конструктори:

- Приймаючий як параметр значення простого типу, що створює об'єкт, відповідний клас-оболонку; наприклад, конструктори *Character(char)* і *Integer(int)*;
- Перетворюючий зміст єдиного параметра типу *String* у вихідне значення об'єкта (за винятком класу *Character*, у складі якого такого конструктора немає), наприклад *new Float("6.02e23")*; для числових типів рядок параметра повинен містити подання величини в десятковій системі числення - якщо перетворення неприпустимо, відображається виключення типу ***NumberFormatException***.

У кожному із класів-оболонок визначені методи:

- **public static Type valueOf(String str)** - повертає об'єкт заданого типу *Type*, що містить значення, перетворене з рядка *str*. Для числових типів мається на увазі, що рядок містить значення аргументу в системі числення з основою 10. Наприклад, *Float.valueOf ("6.02e23")* і *Integer.valueOf("16").* Виклик методу рівнозначний використанню оператора *new*, що має на увазі звертання до конструктора з параметром типу *String*. Якщо вміст *str* не може бути перетворено, відображається виключення типу *NumberFormatException*;
- **public String toString()** - перевизначена версія методу *Object.toString()*, що забезпечує одержання строкового значення вмісту об'єкта класу-оболонки;
- **public type typevalue()** - повертає значення простого типу *type*, що відповідає вмісту поточного об'єкта класу-оболонки. Наприклад, new *Integer(6).intValue()* повертає 6;
- **public int compareTo(Type other)** - повертає значення, менше (більше) нуля або рівне нулю, якщо вміст поточного об'єкта відповідно менше

(більше) значення *other* того ж типу *Type* або дорівнює йому. Метод не визначений у класі *Boolean*.

- **public int compareTo(Object obj)** - якщо *obj* відноситься до того ж типу, що й поточний об'єкт, метод рівнозначний *compareTo(Type)*. У противному випадку відображається виключення типу *ClassCastException*. Метод не визначений у класі *Boolean*;
- **public boolean equals(Object obj)** - повертає *true*, якщо поточний об'єкт і об'єкт, переданий як аргумент, відносяться до одного типу й містять однакові значення. Наприклад, для двох об'єктів *x* и *y* типу *Integer* вираз *x.equals (y)* дорівнює *true*, якщо тільки *x.intvalue() == y.intValue()*. Якщо значення *obj* не належить тому ж типу, що й поточний об'єкт, або дорівнює *null*, метод повертає *false*;
- **public int hachCode()** - повертає хеш-код поточного об'єкта, заснований на його вмісті.

7.2. РЕФЛЕКСІЯ

Пакет *Java.lang.reflect* містить набір метакласів рефлексії (reflection), що дозволяють докладно досліджувати будь-який тип Java. З їхньою допомогою можна створити повнофункціональний броузер типів або додаток, що інтерпретує користувацький код і перетворить його у фактичні інструкції оголошення класів, створення об'єктів, виклику методів і т.д. Всі типи знаходяться в пакеті *Java.lang.reflect*, за винятком класів *Class* і *Package*, які є частиною пакета *Java.lang*.

В основі механізму рефлексії є об'єкт типу *Class*. Він дозволяє одержати повний список членів конкретного класу, простежити ієрархію базових типів (реалізованих інтерфейсів і розширюваних класів) і витягти інформацію про самий клас (наприклад, про використовувані в його оголошенні модифікаторах - *public*, *abstract*, *final* тощо) і про пакет, у якому він міститься.

Механізм рефлексії дозволяє створювати код, що виконує ті ж дії, які можна здійснити за допомогою коду, написаного безпосередньо. Знаючи ім’я певного класу (якого в момент написання програми може і не бути в наявності), вдається одержати відповідний об'єкт типу *Class* і використовувати його для створення об'єктів цього класу.

7.3. КЛАС CLASS

Кожному типу Java - простим типам, класам, інтерфейсам і масивам - відповідає власний об'єкт класу *Class*. Існує також спеціальний об'єкт *Class*, що представляє службове слово *void*. Об'єкти *Class* можуть використовуватися для одержання інформації про типи й якщо тип відноситься до посилальних типів - для створення нових об'єктів цього типу.

Існує чотири способи одержання об'єкта *Class*:

1. Застосування методу *getClass*;
2. Використання літерала типу *Class* (ім'я класу, супроводжуваного *суффиксом.class*, - наприклад, *String.class*);
3. Застосування статичного методу *Class.forName* із вказівкою повного ім'я класу, що включає назву пакета, якому він належить;
4. Одержання даних за допомогою методів рефлексії, які повертають об'єкти *Class* для вкладених класів і інтерфейсів.

Розглянемо приклад (програма відображає на екрані відомості про ієрархію типів для типу, значення якого передається як параметр командного рядка):

```
public class TypeDesc {
  public static void main(String[] args)
 { TypeDesc desc = new TypeDesc();
   for(int i=0; i< args.length; i++){
try {
 Class startclass = Class.forName(args[i]);
   desc.printType(startClass,0,basic); }
catch (ClassNotFoundException e){
  System.err.println(e);  // Якщо виникла помилка } } }
private static String[]
  basic= {"class", "interface" },
  supercl= {"extends", "implements"},
```

```
  iFace= { null, "extends"};
private void  printType(
   Class type, int depth, String[] labels){
   if(type==null) return;//Завершення рекурсії
   //вивід інформації про поточний тип
for (int i =0; i < depth; i++)
  System.out.print("  ");
   System.out.print(labels[type.islnterface()? 1:0] + " ");
  System.out.printIn(type.getName());
// вивести інформацію про всі інтерфейси, що реалізовані
поточним класом
Class[] interfaces = type.getlnterfaces();
   for (int i =0; i < interfaces.length; i++)
   printType(interfaces[i], depth + 1,
   type.islnterface()? iFace : supercl);
   // виконати рекурсію по базових класах
   printType(type.getSuperclass(), depth + 1, supercl);
}}
```

Результат роботи програми при завданні в якості параметра рядка *java.util.HashMap* буде:

```
class java.util.HashMap
  implements java.util.Map
  implements java.lang.Cloneable
  implements java.io.Serializable
  extends java.util.AbstractMap
  implements java.util.Map
  extends java.lang.object
```

У складі класу *Class* визначений ряд простих методів-запитів, що дозволяють перевірити тип об'єкта *Class*:

- **public boolean islnterface()** -повертає *true*, якщо об'єкт *Class* представляє інтерфейс;
- **public boolean isArray()** - повертає *true*, якщо об'єкт *Class* представляє масив;
- **public boolean isPrimitive()** - повертає *true*, якщо об'єкт *Class* представляє один із простих типів або службове слово *void*;
- **public Class[] getlnterfaces()** - повертає масив об'єктів *Class*, елементи якого відповідають інтерфейсам, реалізованим поточним типом. Якщо інтерфейсів, базових стосовно типу, не існує - через те, наприклад, що тип не реалізує інтерфейси безпосередньо або є простим, - вертається масив нульової довжини;

- **public Class getSuperclass()** - повертає об'єкт *Class*, що відповідає класу, базовому стосовно поточного типу. Метод повертає *null*, якщо поточний об'єкт *Class* представляє клас *Object*, інтерфейс, простий тип або службове слово *void* (всі вони не мають базові класи). Якщо поточним типом є масив, вертається об'єкт *Class* для класу *Object*;
- **public int getModifiers()** - повертає модифікатори, які використовуються в оголошенні типу, у вигляді кодованого цілочисельного значення. Значення повинне декодуватися за допомогою констант і методів класу *Modifier*. До числа модифікаторів типу відносяться модифікатори доступу (*public*, *protected* і *private*) і ознаки *abstract*, *final* і *static*. Для зручності факт приналежності типу до категорії інтерфейсів також кодується, для чого застосовується константа *INTERFACE*;
- **public Class getComponentType()** - повертає об'єкт класу *Class*, що представляє компонентний тип масиву, якій описується поточним об'єктом *Class*. Якщо поточний об'єкт *Class* не відповідає масиву, вертається *null*. Нехай, наприклад, заданий масив значень *int*; тоді метод *getClass* поверне об'єкт *Class*, для якого метод-запит *isArray* поверне значення *true*, а метод *getComponentType* - об'єкт *int.class*.
- **public static Class forName(String name, boolean initialize, ClassLoader loader) throws ClassNotFoundException** - повертає об'єкт *Class*, що відповідає названому по імені класу або інтерфейсу, за допомогою заданого завантажника класів. Використовуючи повне ім'я класу або інтерфейсу, передане як параметр *name*, метод виконує спробу виявити й завантажити клас або інтерфейс. Для завантаження класу або інтерфейсу застосовується заданий завантажник класів. Якщо значення параметра *loader* дорівнює *null*, використовується системний завантажник класів. Клас ініціалізується тільки за умови, що значення параметра *initialize* дорівнює *true* і раніше ініціалізація класу не виконувалася. Якщо класс з заданим іменем не може бути знайдений, відображається виключення типу *ClassNotFoundException*.

Імена, передані в метод *forName* повинні бути повними, тобто містити найменування пакета й власне ім'я класу, як, наприклад, *java.uti.HashMap* або *java.lang.Object*. Для подання імен типів масивів використовується спеціальна система позначень - символ [супроводжується спеціальним кодом, що відповідає певному компонентному типу масиву. Компонентні типи кодуються відповідно до наступної схеми.

Код	**Тип**
B	byte
C	char
D	double
F	float
I	int
J	long
L *Ім'я класу*	class або interface
S	short
Z	boolean

Наприклад, масив елементів типу *int* одержить ім'я *[I*, а масив об'єктів типу *Object* - *[Ljava.lang.Object*; Багатомірний масив - це масив, компонентний тип якого також є масивом; тому, наприклад, тип, оголошений як *int[][]*, одержить ім'я *[[I* .

- **public Constructor[] getConstructors();**
- **public Field[] getFields();**
- **public Method[] getMethods();**
- **public Class[] getClasses();**

За допомогою даних методів можна одержати відомості про *public-члени*, які або оголошені в поточному класі (або інтерфейсі), або успадковані.

- **public Constructor[ ];**
- **getDeclaredConstructors();**
- **public Field[ ] getDeclaredFields();**

- **public Method[ ] getDeclaredMethods();**
- **public Class[ ] getDeclaredClasses();**

  За допомогою даних методів можна одержати дані про члени (не обов'язково позначених як *public*) певної категорії, які оголошені в поточному класі (або інтерфейсі), але не успадковані.
- **public Field getField(String name);**
- **public Field getDeclaredField(String name);**

  За допомогою даних методів можна одержати відомості про конкретний член класу. Перший метод знаходить *public-поле*, оголошене в поточному класі або успадковане від базових типів, а другий повертає тільки таке поле (не обов'язково позначене як *public*), що належить поточному класу. Якщо поле із заданим ім'ям не може бути знайдено, видається виключення типу *NoSuchFieldException*.
- **public Method getMethod(String name, Class[] parameterTypes);**
- **public Method getDeclaredMethod(Stringname, Class[] parameterTypes);**

  За допомогою даних методів можна отримати інформацію про конкретні методи класу, у масиві *parameterTypes* передаються типи параметрів методу.
- **public Constructor getConstructor(Class[] parameterTypes);**
- **public Constructor getDeclaredConstructor(Class[] parameterTypes).**

  Аналогічно попереднім методам.

Перед виконанням кожного з розглянутих вище методів здійснюється перевірка повноважень доступу - всі методи взаємодіють із менеджером безпеки, установленим у системі. Якщо менеджер безпеки не підключений, дозволяється використання всіх методів без яких-небудь обмежень. Звичайно менеджери безпеки дозволяють будь-якому коду викликати методи для одержання інформації про *public-члени* типу - це відповідає правилам нормального рівня доступу, прийнятим у мові. Однак можливості звертання

до членів, не позначеним ознакою public, як правило, обмежене - одержання подібних даних дозволено коду, що володіє привілеями усередині системи.

Варто відмітити, що менеджери безпеки здатні розрізняти тільки рівні доступу *public* і *не-public* - для розпізнавання членів, що мають ознаку доступу рівня пакета або *protected*, інформації не досить. Якщо доступ заборонений, виводиться виключення типу *SecurityException*.

Розглянемо приклад:

```
import Java.lang.reflect.*;
 public class ClassContents {
     public static void main(String[] args){
       try {
         Class з = class.forName(args[0]);
         System.out.println(c);
         printMembers(c.getFields());
         printMembers(c.getConstructors()) ;
         printMembers(c.getMethods());}
      catch(ClassNotFoundException e)  {
         System.out.println("Невідомий клас:   " + args[0]);
}}
 private static void printMembers(Member[] mems){
          for(int i=0; i< mems.length;  i++)  {
          if (mems[i].getDeclaringClass() == Object.class)
          continue;
          String decl  = mems[i].toString();
          System.out.print("   ");
          System.out.println(decl);
}}}
```

РОЗДІЛ 8

8.1. КЛАС MODIFIER

Клас *Modifier* кодує всі модифікатори, використовувані в оголошеннях типу, у вигляді констант: *ABSTRACT, FINAL, INTERFACE, NATIVE, PRIVATE, PROTECTED, PUBLIC, STATIC, STRICT, SYBCHRONIZED, TRANSIDENT, VOLATILE*. Кожній з констант відповідає метод-запит виду *isMod(int modifier)* (тут *Mod* одне з вище наведених імен, наприклад, *isPublic*), що повертає *true*, якщо модифікатор *mod* присутній в оголошенні типу.

Розглянемо приклад. Нехай є оголошення поля:

```
public static final int s=10;
```

тоді значення повертається методом *getModifiers* відповідного об'єкта класу *Field* буде мати вигляд:

```
Modifier.PUBLIC | Modifier.STATIC | Modifier.FINAL
```

Модифікатор *strictfp* представляється константою *STRICT*.

Методи-запити можна використовувати в наступній формі:

```
Modifier.isPrivate(field.getModifiers());
```

це еквівалентно наступній умові:

```
(field.getModifiers()&Modifier.PRIVATE)!=0
```

8.2. КЛАС FIELD

У складі класу *Field* реалізовані методи, що дозволяють запитувати інформацію про тип поля, а також зчитувати та змінювати його значення.

Розглянемо деякі методи класу *Field*:

- **getType()** - повертає об'єкт класу *Class*, що відповідає типу поточного поля. Наприклад, для поля типу *int*, одержимо *int.class*.
- Методи **set** і **get** - дозволяють зчитувати поточне значення поля, а також задавати нове. Розглянемо приклад:

```
public static void printsnortField(Object o,  String name)
throws NoSuchFieldException, IllegalAccessException
  Field field = o.getClass() .getField(name);
  Short value = (Short) field.get(o); System.out.println(value);
}
```

Тобто метод *get* повертає покажчик, на яке посилається відповідне поле або об'єкт класу-оболонки.

Приклад використання методу *set* має вигляд:

```
public static void setShortField(Object o, String name,short nv)
throws NoSuchFieldException,   IllegalAccessException
{  Field field = o.getClass().getField(name) ;
   field.set(o,new Short(nv)); }
```

Для збереження *nv* у полі даного об'єкта необхідно використовувати класи оболонки.

Існують також методи, що мають вид *getPrimitiveType* (наприклад, *getInt*) і *setPrimitiveType*. Ці методи можна використовувати для зміни полів у класі, що мають примітивний тип. Наприклад, *field.setShort(o,nv)*.

8.3. КЛАС METHOD

Засоби класу *Method* - дозволяють одержувати повну інформацію, що стосується оголошень методів певного класу, і при необхідності викликати ці методи в контексті заданих об'єктів.

Розглянемо методи класу *Method*:

- **public Class getReturnType()** - повертає об'єкт *Class*, що відповідає типу значення, що повертається поточним методом. Якщо замість типу значення, що повертається, в оголошенні методу зазначене службове слово *void*, розглянутий метод поверне об'єкт *void.class*;
- **public Class[] get ParameterTypes()** - повертає масив об'єктів *Class*, що відповідають типам параметрів, які задані в оголошенні поточного методу. Об'єкти заносяться в масив у тому порядку, у якому параметри перераховані в оголошенні методу. Якщо метод не має параметрів, вертається порожній масив;
- **public Class[] getExceptionTypes()** - повертає масив об'єктів *Class*, що відповідають типам виключень, які задані в пропозиції *throws* оголошення поточного методу. Об'єкти заносяться в масив у тому порядку, у якому найменування типів виключень перераховані в оголошенні методу;
- **public Object invoke(Object onThis, Object[] args) throws IllegalAccessException,IllegalArgumentException,**

InvocationTargetException - викликає метод, обумовлений поточним об'єктом *Method*, у контексті об'єкта *onThis* із завданням значень аргументів, переданих масивом *args*. Для нестатичних методів вибір реалізації здійснюється на підставі фактичного типу об'єкта, обумовленого параметром *onThis*. Для статичних методів *onThis* не приймається до уваги й може дорівнювати *null*. Довжина масиву *args* повинна збігатися із числом параметрів в оголошенні методу, а типи об'єктів-елементів масиву повинні допускати присвоювання відповідним типам параметрів методу - у противному випадку буде відображатися виключення *IllegalArgumentException*. Якщо в складі об'єкта, обумовленого параметром *onThis*, немає типу, членом якого є поточний метод, відображається виключення *IllegalArgumentException*. Якщо значення *onThis* дорівнює *null* і метод не статичний, генерується виключення типу *NullPointerException*. Якщо виконання викликуваного методу завершується аварійно, відображаеться виключення типу *InvocationTargetException*.

Розглянемо приклад. Викличемо засобами рефлексії метод *str.indexOf(".", 8),* тоді маємо:

```
Throwable failure;
try {
 Class strClass = str.getClass();
 Method index = strClass.getMethod("indexOf", new Class[]
{string.class,  int.class } );
Object  result = index.invoke(str, new object[] { ".",   new
lnteger(8)} );
return  ((Integer) result).intvalue();}
  catch  (NoSuchMethodException e)  { failure = e; }
  catch  (invocationTargetException e) {
    failure = e.getTargetException(); }
  catch  (illegalAccessException e)  {failure = e;}
throw failure;
```

8.4. КЛАС CONSTRUCTOR

Для створення нових екземплярів (об'єктів) типу може бути використаний метод *newInstance* об'єкта *Class*, що відповідає цьому типу. Метод викликає конструктор без аргументів, що належить типу, і повертає посилання на знову створений об'єкт класу *Object*, що повинен бути явно перетворений до необхідного типу.

Розглянемо приклад.

```
static double[]  testData = { 0.3,1.3 e-2, 7.9, 3.17 };
 public static void main(String[] args){
try {
    for  (int arg = 0; arg < args.length; arg++){
     String name = args[arg];
     Class classFor = Class.forName(name);
     SortDouble sorter = (SortDouble)classFor.newInstance();
     SortMetrics metrics = sorter.sort(testData);
     System.out.println(name + ": " + metrics);
     for  (int i  =0;  i  < testData.length; i++)
     System.out.println(" " + testData[i]); } }
catch(Exception e)
     { System.err.println(e);
}}
```

Метод *newInstance* при некоректному застосуванні здатний відображати велику кількість об'єктів виключень різних типів:

- **InstantiationException** - якщо клас, об'єкт якого повинен бути створений, не має конструктора без аргументів, або визначений як абстрактний, або в дійсності є інтерфейсом, або виконання процедури створення об'єкта переривається за будь-яких інших причин;
- **IllegalAccessException** - якщо клас або його конструктор без аргументів недоступні;
- **SecurityException** - якщо діюча політика безпеки забороняє створення нових об'єктів;
- **ExceptionInInitializerError** - відображається при ініціалізації класу.

У класі *Constructor* визначені й інші методи:

- **public Class[] getParameterTypes***()* - повертає масив об'єктів *Class*, що відповідають типам параметрів, які задані в оголошенні поточного конструктора;

- **public Class[] getExceptionTypes()** - повертає масив об'єктів *Class*, що відповідають типам виключень, які задані в пропозиції *throws* оголошення поточного конструктора;
- **public Object newlnstance(Object[] args) throws InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException** - використовує конструктор, що представляється поточним об'єктом *Constructor*, для створення й ініціалізації нового екземпляра класу, у якому конструктор оголошений, з передачею заданих аргументів. Повертає посилання на знову створений і ініціалізований об'єкт. Довжина масиву *args* повинна збігатися із числом параметрів в оголошенні конструктора, а типи об'єктів-елементів масиву повинні допускати присвоювання відповідним типам параметрів конструктора - у противному випадку буде відображатися виключення *IllegalArgumentException*.

8.5. КЛАС ACCESSIBLEOBJECT

Класи *Field*, *Constructor* і *Method* є похідними від класу AccessibleObject, що дає можливість дозволяти або забороняти перевірку ознак доступу рівня мови, таких як *public* і *private*.

Клас *AccessibleObject* має методи:

- **public void setAccessible(boolean flag)** - встановлює прапор доступу до об'єкта у відповідності зі значенням аргументу: *true* означає, що об'єкт більше не підкоряється правилам доступу, установлюваним на рівні мови (і буде завжди доступний), а *false* змушує об'єкт підтримувати заданий рівень доступу. Якщо повноважень по зміні прапора доступу недостатньо, відображається виключення типу *SecurityException;*
- **public static void setAccessible(AccessibleObject[] array, boolean flag)** - дозволяє встановити прапор доступу до об'єктів, переданим у вигляді масиву. Якщо в процесі обробки чергового об'єкта викидається виключення типу *SecurityException*, об'єкти, розташовані в масиві раніше,

зберігають знову задані значення рівня доступу, а всі інші об'єкти залишаються в колишньому стані;

- **public boolean isAccessible()** - повертає поточне значення прапора доступу до об'єкта.

8.6. КЛАС ARRAY

Клас *Array* використовується для створення масиву засобами рефлексії. Для створення масивів використовуються дві форми методу *newInstance*:

- **public Object newlnstance(Class compType, int length)** - повертає посилання на новий масив типу *compType* заданої довжини *length*.
- **public Object newinstance(Class compType, int[] dim) -** повертає посилання на новий багатомірний масив типу *compType*, розмірності якого задані значеннями елементів масиву-параметра *dimensions*. Якщо масив *dim* порожній або має довжину, що перевищує припустиме число размірностей (звичайно 255), відображається виключення типу *IllegalArgumentException*.

Розглянемо приклади.

Приклад 1. Сформуємо масив типу *byte*:

```
byte[] ba =  (byte[]) Array.newlnstance(byte.class,   13);
```

Це рівнозначно:

```
  byte[] ba = new byte[13];
```

Приклад 2.

```
int[] dims = {4, 4};
double[][] matrix =(double[][])
Array.newlnstance(double.class,  dims);
```

Це рівнозначно:

```
double[][]  matrix = new double[4][4];
```

Клас *Array* має методи *get* і *set*. Нехай заданий масив xa значень типу int; тоді вираженню *xa[i]* буде відповідати:

```
Integer n=Array.get(xa, i)
```

Надати значення елементу масиву можна так: *xa[i] = 23;* - це те ж саме, що *Array.set(xa, i, new Integer(23))*.

8.7. КЛАС PACKAGE

Виклик методу *getPackage* класу *Class* дозволяє одержати об'єкт класу *Package*, що містить опис пакета, у складі якого визначений клас (сам клас *Package* розміщений у пакеті *java.lang*).

Метод *getName* об'єкта *Package* повертає повне ім'я поточного пакета.

8.8. КЛАС PROXY

Клас *Proxy* дозволяє динамічно створювати класи, що реалізують один або декілька інтерфейсів.

Розглянемо задачу ведення журналу звертань до об'єкта, що зберігає інформацію про виклики методів об'єкта і дозволяє у випадку, непередбаченої ситуації відновити хід подій.

Модель, реалізована класом *Proxy*, пердбачає виклик методу *Proxy.getProxyClass* з передачею йому посилань на завантажник класів і масив об'єктів, що представляють інтерфейси, - у результаті буде отриманий об'єкт *Class* для типу *Proxy*.

Об'єкти *Proxy* володіють одним конструктором, що приймає як параметр об'єкт класу, похідного від *InvocationHandler*.

Для створення об'єкта *Proxy* варто викликати метод newInstance об'єкта *Constructor* з параметром типу *InvocationHandler*.

```
public class DebugProxy implements InvocationHandler {
   private final Object obj;    // об'єкт
   private final   List methods;      // методи, що викликаються
   private final   List history;      // журнал обігів
   private DebugProxy(Object obj)  {
      this.obj = obj;
      methods = new ArrayList;
      history = Collections.unmodifiableList(methods); }
   public static synchronized Object proxyFor(Object obj)  {
   Class objclass = obj.getclass();
   return Proxy.newProxyinstance(objclass.getClassLoader(),
objclass.getinterfaces(), new DebugProxy(obj)); }
   public Object
   invoke(Object proxy,  Method method,  Object[]  args) throws
Throwable
    {methods.add(method);
 try {
     return method.invoke(obj,   args); }
```

```
 catch(invocationTargetException e)  {
     throw e.getTargetException(); } }
public List getHistory() {return history; }
```

Якщо необхідно налаштувати *Proxy* для заданого об'єкта, варто викликати *ProxyFor*:

```
Object proxyObj  =  DebugProxy.proxyFor(realObj);
```

Таким чином, об'єкт *proxyObj* реалізує ті ж інтерфейси, що й об'єкт *realObj*. Одержати журнал можна за допомогою коду:

```
DebugProxy h = (DebugProxy)
Proxy.getlnvocationHandler(proxyObj);
List history = h.getHistory();
for(int i= 0; i< history.size(); i++)
Systern.out.println(history.get(i)) ;
```

Якщо не користуватися *newProxylnstance*, то код буде мати вигляд:

```
Class proxyClass = Proxy.getProxyClass(
     objclass.getclassLoader(),
     objClass.getInterfaces();
     Constructor ctor = proxyClass.getConstructor(new Class[]
{InvocationHandler.class});
     Object[] ctorArgs = new Object[] {
     new DebugProxy(obj)};
     Object proxyobj = ctor.newInstance(ctorArgs);
}
```

8.9. ЗАВАНТАЖЕННЯ КЛАСІВ

Виконуюча система завантажує класи в міру виникнення необхідності в них. Функціональні особливості процедур завантаження класів істотно залежать від реалізації віртуальних машин Java, але в більшості випадків для відшукання класів, які адресуються програмою, але не завантажених виконуючою системою, застосовується механізм перегляду шляхів пошуку класів.

Щоб створити додаток, що у стані завантажувати класи способами, відмінними від передбачених за замовчуванням, варто скористатися об'єктом класу *ClassLoader*, здатним одержати байт-код реалізації потрібного класу й завантажити його в середовище виконуючої системи.

Клас *ClassLoader* є абстрактним класом. Для створення власного завантажника класів, необхідно створити клас-спадкоємець від *ClassLoader* і перевизначити метод:

- **protected Class findClass(String name) throws ClassNotFoundException** - яий знаходить байт-код класу із заданим ім'ям *name* і завантажує дані в середовище віртуальної машини, повертаючи об'єкт *Class*, що представляє знайдений клас. Об'єкт-звантажник здатний делегувати повноваження по завантаженню класів "батьківському" завантажнику класів (*parent class loader).* "Батьківський" завантажник класів може бути заданий як аргумент конструктора класу *ClassLoader. protected ClassLoader()* створює об'єкт *ClassLoader*, неявно використовуючи в якості "батьківського" завантажника класів системний завантажник (який може бути отриманий за допомогою виклику методу *getSystemClassLoader*);
- **protected ClassLoader(ClassLoader parent)** - створює об'єкт *ClassLoader*, використовуючи заданий "батьківський" завантажник класів. Основним у складі класу *ClassLoader* є метод *loadClass public class loadClass(String name) throws ClassNotFoundException* повертає об'єкт *Class* для класу із заданим ім'ям і при необхідності завантажує цей клас. Якщо клас не може бути завантажений, відображається виключення типу *ClassNotFoundException*.

Схема завантаження класів, що пропонується методом *loadClass* за замовчуванням і звичайно не перевизначена, виглядає так:

1. Перевірити за допомогою виклику методу *findLoadedClass* класу *Class Loader*, чи не завантажувався заданий клас раніше; у складі *ClassLoader* передбачена таблиця об'єктів *Class* для всіх класів, завантажених засобами поточного завантажника класів; якщо клас був завантажений колись, метод *findLoadedClass* поверне посилання на існуючий об'єкт *Class*;
2. Якщо клас не завантажувався, викликається *loadClass* "батьківського" завантажника класів; якщо поточний завантажник не володіє "батьком", використовується системний завантажник класів;

3. Якщо клас усе ще не завантажений, викликається метод *findClass*, що виконує пошук і завантаження класу.

Розглянемо приклад.

```
 class  PlayerLoader extends ClassLoader {
    public Class findClass(String name) throws
ClassNotFoundException{
 try {
  byte[] buf = bytesForClass(name);
  return defineClass(name, buf, 0, buf.length); }
catch    (IOException e)  {
  throw new classNotFoundException(e.toString()); } }
//  ... Оголошення методу bytesForclass і інших методів
}
```

Метод *findClass* звичайно виконує дві функції. По-перше, він повинен виявити байт-код заданого класу й зберегти його в масиві типу *byte* - цей обов'язок у прикладі покладений на метод *bytesForClass*. По-друге, він використовує прикладний метод *defineClass*, щоб виконати фактичне завантаження класу, обумовленого байт-кодом.

Метод *defineClass* має вигляд **protected final Class defineClass(String name, byte[] data, int offset, int length) throws ClassFormatError** - повертає об'єкт *Class* для класу із заданим ім'ям *name*; бінарне подання класу передається у вигляді масиву *data*. Для завантаження класу використовуються тільки байти, що містяться в елементах масиву *data* з індексами від *offset* до *offset+length*. Якщо байти із зазначеного проміжку не задовольняють необхідному формату опису класу, відображається об'єкт виключення типу *ClassFormatError*. Метод відповідальний за збереження посилання на об'єкт *Class* для завантаженого класу в таблиці завантажених класів, що переглядається методом *findLoadedClass*.

Розглянемо метод *bytesForClass*.

```
protected byte[]  bytesForClass(String name) throws
lOException, ClassNotFoundException
{
   FileInputStream in = null;
try {
    in = streamFor(name + ".class");
    int length = in.available();   // число доступних байтів
    if (length == 0) throw new
```

```
   ClassNotFoundException(name);
   byte[]  buf = new byte[length];
   in.read(buf);   // Зчитування байтів
   return buf; }
 finally {  if (in!=null) in.close();
}}
```

РОЗДІЛ 9

9.1. ПІДГОТОВКА КЛАСУ ДО ВИКОРИСТАННЯ

Процес підготовки класу до використання має три стадії:

1. завантаження; одержання байт-коду, що реалізує клас, і створення об'єкта *Class*;
2. зв'язування; перевірка того, чи задовольняє байт-код синтаксису мови, виділення необхідної пам'яті для зберігання статичних даних і (ця функція не обов'язкова) всіх посилань, що місяться в коді класу, із завантаженням, якщо це необхідно, класів, що відповідають таким посиланням;
3. ініціалізація; ініціалізація базового класу (з його попереднім завантаженням і зв'язуванням, якщо це необхідно) і обчислення всіх виразів і блоків статичної ініціалізації класу.

Об'єкт *Class*, що повертається методом *defineClass*, служить тільки меті подання завантаженого класу - клас усе ще залишається не зв'язаним. Виконання процедури зв'язування може бути примусово спровоковано викликом методу *resolveClass*. Цей метод має вигляд *protected final void resolveClass(Class c).* Зв'язує клас, якщо той ще не зв'язаний.

Віртуальна машина, приступаючи до ініціалізації класу, перевіряє, чи виконане його зв'язування. Клас повинен бути ініціалізован перш, ніж може бути створений об'єкт класу, або викликаний статичний метод класу, або здійснений доступ до статичного поля класу. Особливості процесу залежать від конкретної реалізації віртуальної машини Java.

9.2. ЗАВАНТАЖЕННЯ ДОДАТКОВИХ РЕСУРСІВ

Класи є основними ресурсами, яких потребує програма, але деяким класам необхідні також додаткові ресурси, такі як фрагменти тексту, зображення або дані мультимедиа.

Розглянемо приклад завантаження додаткових ресурсів:

```
String b = "Str1.fl";
InputStream in;
```

```
ClassLoader loader = this.getClass() .getClassLoader() ;
 if (loader!= null)
    in = loader.getResourceAsStream(b);
 else
   in = ClassLoader.getSystemResourceAsStream(b);
```

Системні ресурси пов'язані із системними класами, а такі можуть не мати завантажники. Статичний метод *getSystemResourceAsStream* повертає об'єкт *InputStream* потоку уведення для ресурсу із заданим ім'ям.

Реалізація методу *getResourceAsStream*, пропонована класом *ClassLoader*, передбачає повернення значення *null*. Задача перевизначення методу з підтримкою потрібного алгоритму пошуку ресурсів покладають на похідний клас завантажника. Звичайно це значить, що ресурси будуть знайдені в тих же місцях і у той же спосіб, що й файли класів.

Наприклад, для класу *PlayerLoader* (визначений в попередній лекції) маємо:

```
public  InputStream getResourceAsStream
String name) {
 try
     {return streamFor(name); }
 catch (IOException e)
     { return null; }
}
```

Два інших методи класу *ClassLoader*, що мають відношення до проблеми пошуку й завантаження ресурсів - *getResource* і *getSystemResource*, - повертають об'єкти класу *URL*.

Метод *getContents* об'єкта *URL*, що повертається методами завантажника класів, дозволяє одержати об'єкт, що представляє вміст, що відповідає адресі *URL*.

Існують також методи:

- **public Enumeration findResources(String name)** - повертає об'єкт *Enumeration*, що дозволяє переглядати об'єкти *URL* для всіх ресурсів, що відповідають заданому імені. Реалізація методу, пропонована за замовчуванням класом *ClassLoader*, повертає об'єкт *Enumeration* з нульовим числом ресурсів;

- **public Java.net.URL findResource(String name)** - повертає об'єкт *Java. net.URL* для ресурсу із заданим ім'ям або *null*, якщо такий не знайдений. Якщо існує кілька ресурсів, що відповідають заданому імені, правила вибору одного з них визначаються конкретною реалізацією. Реалізація методу, пропонована за замовчуванням класом *ClassLoader*, повертає значення *null*.

9.3. СЕРІАЛІЗАЦІЯ

Серіалізація – це процес збереження стану об'єкта в послідовність байтів; десеріалізація є процесом відновлення об'єкта з цих байтів.

Якщо двом компонентам Java необхідно спілкуватися один з одним, то їм необхідний механізм для обміну даними. Є кілька способів реалізувати цей механізм.

Перший спосіб – розробити власний протокол і передати об'єкт. Це означає, що одержувач повинен знати протокол, який використовує відправник для відтворення об'єкта, що ускладнює розробку сторонніх компонентів. Отже, повинен бути універсальний і ефективний протокол передачі об'єктів між компонентами. Для цього і створена серіалізація, компоненти Java використовують цей протокол для передачі об'єктів.

9.3.1. Серіалізація об’єктів Java

Клас об'єкта, що серіалізується, повинен реалізувати інтерфейс *java.io.Serializable*:

```
import java.io.Serializable;
 class TestSerial implements Serializable {
  public byte version = 100;
  public byte count = 0;
}
```

Інтерфейс *Serializable* – це інтерфейс-маркер; у ньому не задекларовано жодного методу.

Він говорить механізму серіалізації, що клас може бути серіалізовано.

Серіалізація робиться викликом методу

```
writeObject() класу java.io.ObjectOutputStream
```

Розглянемо приклад:

```
public static void main(String args[])
throws IOException
{
     FileOutputStream fos = new FileOutputStream("temp.out");
     ObjectOutputStream oos = new ObjectOutputStream(fos);
     TestSerial ts = new TestSerial();
     oos.writeObject(ts);
     oos.flush();
     oos.close();
 }
```

Відтворити об'єкт із файлу можна в такий спосіб:

```
public static void main(String args[]) throws IOException {
     FileInputStream fis = newFileInputStream("temp.out");
     ObjectInputStream oin = new ObjectInputStream(fis);
     TestSerial ts = (TestSerial) oin.readObject();
     System.out.println("version="+ts.version);
 }
```

Відновлення об'єкта відбувається за допомогою виклику методу *oin.readObject().* У методі відбувається зчитування набору байтів з файлу й створення точної копії оригінального об'єкту. *oin.readObject()* може прочитати будь-який серіалізований об'єкт, тому необхідно отриманий об'єкт приводити до конкретного типу.

9.3.2. Підготовка класів до серіалізації

За замовчуванням процес серіалізації полягає в серіалізації кожного поля об'єкта, що не позначений як *transient* або *static,* також даний процес припускає, що всі поля-об'єкти, що підлягають серіалізації, повинні вказувати на типи, які у свою чергу підтримують можливість серіалізації.

Крім того, потрібно, щоб клас, базовий відносно розглянутого, або мав конструктор без параметрів (щоб останій міг бути викликаний у процесі десеріалізації), або сам у свою чергу забезпечував реалізацію інтерфейсу *Serializable* (у такому випадку реалізація того ж інтерфейсу в похідному класі буде надлишковим рішенням).

Розглянемо приклад:

```
public class Name implements Serializable {
   private String name;
   private long id;
   private transient boolean hashSet = false;
   private transient int hash;
   private static long nexti = 0;
   public Name(String name)   {
   this.name = name;
   synchronized  (Name.class)   { id = nextl++;} }
   public int hashCode() {
   if (lhashset) {hash = name.hashCode() ; hashSet = true;}
   return hash;}
    ...
   }
```

Об'єкт класу *Name* може бути збережений у потоці *ObjectOutputStream* і безпосередньо, за допомогою виклику методу *writeObject*, і побічно, якщо цей об'єкт адресується з іншого об'єкта, що підлягає серіалізації. Вміст полів *name* і *id* виводиться в потік; поля *nextID*, *hashSet* і *hash*, у процес серіалізації залучені не будуть, оскільки *nextID* оголошено як *static*, а два інших поля позначені модифікатором *transient*. Поле *hash* містить хеш-код, що може бути заново перелічений на підставі вмісту *name*, немає ніяких причин витрачати на його серіалізацію додаткові обчислювальні ресурси й час.

Схема десеріалізації, пропонована за замовчуванням, передбачає зчитування з потоку байтових даних, збережених у процесі серіалізації. Статичні поля класу залишаються в недоторканності - у ході завантаження класу будуть виконані всі звичайні процедури ініціалізації, і статичні поля одержать необхідні вихідні значення. Кожному полю *transient* у відновленому об'єкті привласнюється значення за замовчуванням, що відповідає типу цього поля. При десеріалізації об'єкта класу *Name* полям name і *id* знову створеного об'єкта будуть привласнені ті ж значення, що й у вихідному об'єкті, вміст статичного поля *nextID* залишиться недоторканим, а *transient-поля hashSet* і *hash* одержать відповідні значення за замовчуванням (*false* і *0*)- якщо значення *hashSet* дорівнює *false*, величина *hash* буде перераховано заново.

Будь-яка спроба здійснення серіалізації об'єкта, що не підтримує цю можливість, призводить до виключення типу *NotSerializableException*.

9.3.3. Порядок серіалізації та десеріалізації

Серіалізація об'єктів виконується в порядку за спаданням по деревоподібній ієрархії типів - від першого із класів, що підтримують інтерфейс *Serializable*, до класів більш приватних типів.

Розглянемо наступну ієрархію типів (Рис. 9.1.):

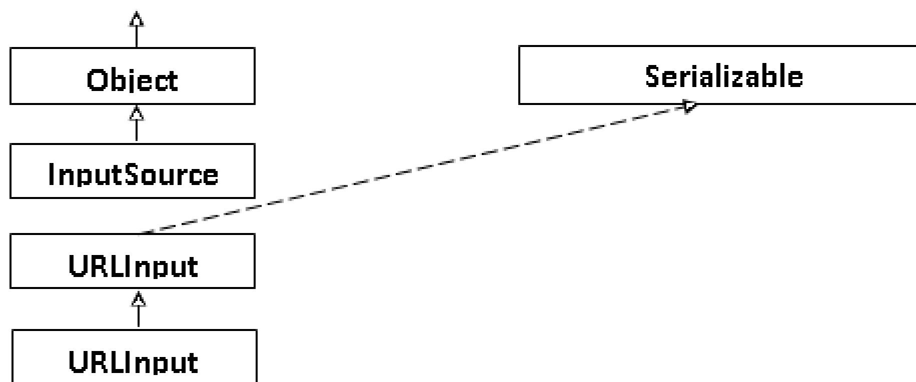


Рис. 9.1. Серіалізація класів

Розглянемо порядок десеріалізації для об'єкта *HTTPInput*:

1. Потік *ObjectlnputStream* спочатку виділяє пам'ять для нового об'єкта, а потім знаходить в ієрархії типів найближчий із класів, що підтримує інтерфейс *Serializable*, тобто *URLInput*;
2. Потік викликає конструктор без параметрів для класу, базового стосовно знайденого (найближчого із числа тих, які не підтримують інтерфейс *Serializable*), тобто *InputSource*.

Примітка. Якщо повинен бути збережений інший стан частини об'єкта, що відноситься до базового класу (*InputSource*), відповідальність за серіалізацію й десеріалізацію цього стану покладається на *URLInput*. Якщо ж базовий клас, що не забезпечує серіалізацію, має власний значимий стан, то необхідно відступити від схеми серіалізації, пропонованої за замовчуванням, і здійснити додаткове настроювання найближчого із класів, що підтримують механізм серіалізації.

3. Після того, як найближчий із класів, що підтримують механізм серіалізації, завершує відновлення даних тої підмножини полів, які

відностяться до базового класу, він приступає до відтворення свого власного стану, зчитуючи дані з потоку;

4. Потік *ObjectInputStream* здійснює перегляд ієрархії об'єктів у спадному порядку й виконує десеріалізацію кожного об'єкта за допомогою викликів *readObject*.

Якщо в ході десеріалізації об'єкта будуть знайдені посилання на інші раніше серіалізовані об'єкти, вони об'єкти піддаються десеріалізації в тому порядку, у якому вони виявляються. Так, якщо об'єкт *URLInput*, скажемо, мав би посилання на об'єкт *HashMap*, адресована хеш-таблиця разом з усім її вмістом була би піддана десеріалізації раніше, ніж частина, що залишилася, об'єкта *URLInput* як такого.

Перш ніж будуть виконані будь-які з розглянутих операцій десеріалізації, відповідні класи повинні бути завантажені, інакше метод *readObject* видасть виключення *ClassNotFoundException*.

9.3.4. Настроювання механізму серіалізації

Для деяких класів звичайні засоби десеріалізації можуть виявитися неадекватними або неефективними.

У цьому випадку необхідно перевизначити методи *readObject* і *writeObject*. Подібні перевизначені варіанти методів *writeObject* і *readObject* викликаються тільки в контексті об'єктів відповідних класів, і методи несуть відповідальність винятково за стан об'єкта класу як такого, у тому числі й у частині, успадкованої від базового класу, що не підтримує механізм серіалізації. Якщо в складі класу реалізовані власні варіанти методів *writeObject* і *readObject*, вони не повинні звертатися до однойменних методів базового класу.

Як приклад розглянемо поліпшений варіант класу *Name*:

```
public class BetterName implements Serializable {
    private String name;
    private long id;
    private transient int hash;
```

```
    private static long nextl = 0;
    public BetterName(String name)  {
       this.name = name;
       synchronized  (BetterName.class)  { id = nextl++;}
       hash = name.hashCode(); }
    private void writeObject(ObjectOutputStream out) throws
lOException
        {out.writeUTF(name); out.writeLong(id); }
    private void readObject(Objectlnputstream in) throws
lOException, ClassNotFoundException
  {name = in.readUTF();
   id = in. readLong() ;
   hash = name.hashCode(); }
...}
```

Існує обмеження, яке варто взяти до відома, приступаючи до настроювання механізму серіалізації: не можна привласнювати значення полю *final* у тілі методу *readObject*, оскільки поля *final* можуть бути про-ініціалізовані тільки в блоках ініціалізації або конструкторах.

Набір інструкцій виклику "низькорівневи" методів *read* і *write* класів *Datalnput* і *DataOutput* можуть бути замінені єдиним звертанням до відповідного методу, що реалізує механізм серіалізації або десеріалізації, що передбачений за замовчуванням.

```
private void writeObject(ObjectOutputStream out) throws
lOException
   { out.defaultWriteObject();}
private void readObject(Objectlnputstream in) throws
lOException, ClassNotFoundException
   {in.defaultReadObject();
    hash = name.hashcode();
}
```

Зустрічаються ситуації, коли об'єкт не може бути належним чином проініціалізованим доти, поки граф, частиною якого він є, не буде десеріалізовано повністю.

Можна змусити потік *ObjectInputStream* викликати метод визначений користувачем, якщо звернутися до методу *registerValidation* потоку й передати йому посилання на об'єкт класу, що реалізує інтерфейс *ObjectlnputValidation*. По завершенні десеріалізації об'єкта верхнього рівня графа буде викликаний метод *validateObject* передбаченого користувачем

об'єкта *ObjectlnputValidation*, що зможе виконати всі необхідні операції ініціалізації й контролю цілісності даних.

При серіалізації вміст об'єкта зберігається в потоці, а потім у ході десеріалізації відновлюється копія того ж об'єкта. Однак, можуть зустрітися класи, для яких подібна схема дій не підходить. Якщо, наприклад, об'єкти якогось класу, як передбачається, повинні бути унікальними для кожної віртуальної машини, у ході десеріалізації об'єкта, можливо, буде потрібно замінити його рівноцінним об'єктом, що вже існує в середовищі віртуальної машини. Подібний механізм можна реалізувати шляхом визначення методів *writeReplace* і *readResolve*:

- **<*модифікатор доступу*> Object writeReplace() throws ObjectStreamException** - повертає об'єкт, що у процесі серіалізації замінить поточний об'єкт. Може бути повернутий будь-який об'єкт, включаючи поточний;
- **<*модифікатор доступу*> Object readResolve() throws ObjectStreamException** - повертає об'єкт, що у процесі десеріалізації замінить поточний об'єкт. Може бути повернутий будь-який об'єкт, включаючи поточний.

Наприклад, метод *readResolve* може забезпечити пошук об'єкта, рівноцінного тому, що отриманий у результаті десеріалізації. Якщо такий існує, він буде повернутий, а інакше варто зареєструвати поточний об'єкт (для майбутнього використання в контексті *readResolve*) і повернути *this*.

9.3.5. Контроль версій об'єктів

Реалізація класів може бути змінена, якщо подібне відбувається в проміжку часу між серіалізацією і десеріалізацією об'єкта цього класу, потік *ObjectInputStream* здатний виявити факт внесення змін.

При збереженні об'єкта разом з ним записується унікальний ідентифікатор номера версії (*serial version unique identifier*, *UID*), 64-бітове

значення типу *long*. За замовчуванням ідентифікатор створюється у вигляді хеш-коду, побудованого на основі інформації про імена класу, його членів і базових інтерфейсів; зміна таких даних служить сигналом про можливу несумісність версій класу.

При введенні даних про об'єкт із потоку *ObjectInputStream* зчитується також й ідентифікатор номера версії. Потім відбувається спроба завантаження відповідного класу.

Якщо необхідний клас не знайдений або ідентифікатор завантаженого класу не збігається з тим, що зчитаний з потоку, метод *readObject* відображає виключення типу *InvalidClassException*. Якщо успішно завантажені всі потрібні класи й виявлений збіг всіх ідентифікаторів, об'єкт може бути десеріалізаван.

Існує можливість зберегти сумісність класу, що має додаткові несуттєві нововведення, з даними раніше серіалізованих об'єктів. Для цього варто примусово оголосити в складі класу спеціальне поле, що містить значення ідентифікатора номера версії, наприклад:

```
private static final long serialversionUl = -
1307795172754062330L;
```

9.3.6. Типи *PutField* і *GetField*

Розглянемо приклад, коли в процесі розвитку класу може знадобитися доступ до первинних версій структури полів класу.

```
public class  Rectangle implements Serializable {
   private static final  long
   serialVersionUID = -1307795172754062330L;
   transient private double x, y;
   transient private double width, height;
   private void  readObject(Objectlnputstream in) throws
lOException, ClassNotFoundException
  { ObjectinputStream.GetField fields;
    fields = in.readFields();
    x = fields.get("x1", 0.0);
    в = fields.get("y1", 0.0);
    double x2 = fields.get("x2", 0.0);
    double y2 = fields.get("y2", 0.0);
    width = (x2  - x);
```

```
    height = (y2  - y);}
private void writeObject(ObjectOutputstream out) throws
lOException
   {ObjectOutputStream.PutField fields = out.putFields();
    fields.put("x1", x);
    fields.put("y1",y);
    fields.put("x2", x + width);
    fields.put("y2",  в + height);
    out.writeFields(); }
    private static final ObjectStreamField[]
serialPersistentFields = {new objectStreamField("x1",
Double.TYPE), new ObjectStreamField("y1", Double.TYPE),
new ObjectStreamField("x2",Double.TYPE),
new ObjectStreamField("y2",Double.TYPE),
}; }
```

Для того, щоб виконуюча система в процесі десеріалізації була здатна визначити, до якого саме типу ставляться значення, збережені засобами об'єкта *PutField*, варто позначити серіалізуємі поля, оголосивши спеціальний масив об'єктів типу *ObjectStreamField*.

9.3.7. Інтерфейс *Externalizable*

Інтерфейс *Externalizable* є похідним від *Serializable*. Клас, що реалізує інтерфейс *Externalizable*, здобуває всі повноваження по керуванню станом об'єкта, що підлягає серіалізації, але при цьому несе й сугубу відповідальність за коректну обробку даних, що відносяться до базових класів, забезпечення контролю версій тощо.

У складі інтерфейсу *Externalizable* визначені два методи:

```
public interface Externalizable extends Serializable {
 void writeExternal(ObjectOutput out) throws lOException;
 void  readExternal(Objectlnput in) throws lOException,
 ClassNotFoundException;
}
```

Методи викликаються в процесі серіалізації й десеріалізації об'єкта відповідно.

Класам, похідним від класу, що реалізує інтерфейс *Externalizable*, часто необхідно викликати відповідний метод базового класу перш ніж буде здійснена серіалізація або десеріалізація їхнього власного стану – на відміну від класів, що реалізують схему серіалізації, прийняту за замовчуванням.

РОЗДІЛ 10

10.1. ФОРМАТ СЕРІАЛІЗОВАНОГО ОБ'ЄКТА

Розглянемо вигляд серіалізованого об'єкта *TestSerial*, клас *TestSerial* має вигляд:

```
class parent implements Serializable {
  int parentVersion = 10;}
class contain implements Serializable{
  int containVersion = 11;}
public class SerialTest extends parent implements Serializable {
  int version = 66;
  contain con = new contain();
  public int getVersion() {
  return version;  }
public static void main(String args[]) throws IOException {
  FileOutputStream fos = newFileOutputStream("temp.out");
  ObjectOutputStream oos = new  ObjectOutputStream(fos);
  SerialTest st = new SerialTest();
  oos.writeObject(st);
  oos.flush();
  oos.close();
}}
```

У файлі *temp.out* серіалізований об'єкт *st* матиме вигляд:


```
AC ED 00 05 73 72 00 0A 53 65 72 69 61 6C 54 65
73 74 05 52 81 5A AC 66 02 F6 02 00 02 49 00 07
76 65 72 73 69 6F 6E 4C 00 03 63 6F 6E 74 00 09
4C 63 6F 6E 74 61 69 6E 3B 78 72 00 06 70 61 72
65 6E 74 0E DB D2 BD 85 EE 63 7A 02 00 01 49 00
0D 70 61 72 65 6E 74 56 65 72 73 69 6F 6E 78 70
00 00 00 0A 00 00 00 42 73 72 00 07 63 6F 6E 74
61 69 6E FC BB E6 0E FB CB 60 C7 02 00 01 49 00
0E 63 6F 6E 74 61 69 6E 56 65 72 73 69 6F 6E 78
70 00 00 00 0B
```


Розглянемо, що являють собою байти в сериализованном об'єкті:

- AC ED: STREAM_MAGIC. Говорить про те, що використовується протокол серіалізація.
- 00 05: STREAM_VERSION. Версія серіалізації.
- 0x73: TC_OBJECT. Позначення нового об'єкта.
- 0x72: TC_CLASSDESC. Позначення нового класу.
- 00 0A: Довжина ім'я класу.
- 53 65 72 69 61 6c 54 65 73 74: SerialTest, ім'я класу.
- 05 52 81 5A AC 66 02 F6: SerialVersionUID, ідентифікатор класу.

- 0x02: Різні прапори. Цей специфічний прапор говорить про те, що об'єкт підтримує серіалізацію.
- 00 02: Число полів у класі. алгоритм записує поле int version = 66:
- 0x49: Код типу поля. 49 це «I», що закріплено за int.
- 00 07: Довжина ім'я поля.
- 76 65 72 73 69 6F 6E: version, ім'я поля.
- Далі алгоритм записує наступне поле, *contain con = new contain();*
- 0x74: TC_STRING. Позначає новий рядок.
- 00 09: Довжина рядка.
- 4C 63 6F 6E 74 61 69 6E 3B: Lcontain;, Канонічне JVM позначення (тому що це об'єкт).
- 0x78: TC_ENDBLOCKDATA, Кінець опціонального блоку даних для об'єкта.

Потім іде опис класу *parent*:

- 0x72: TC_CLASSDESC. Позначення нового класу
- 00 06: Довжина ім'я класу.
- 70 61 72 65 6E 74: parent, ім'я класу.
- 0E DB D2 BD 85 EE 63 7A: SerialVersionUID, ідентифікатор класу.
- 0x02: Різні прапори. Цей прапор позначає що клас підтримує серіалізацію.
- 00 01: Число полів у класі.
- Далі йде опис полів класу parent, клас має одне поле, *int parentVersion = 100.*
- 0x49: Код типу поля. 49 позначає «I», що закріплене за Int.
- 00 0D: Довжина ім'я поля.
- 70 61 72 65 6E 74 56 65 72 73 69 6F 6E: parentVersion, ім'я поля.
- 0x78: TC_ENDBLOCKDATA, кінець опційного блоку даних для об'єкта.
- 0x70: TC_NULL, позначає те що більше немає суперкласів, тому що ми досягли верха ієрархії класів.

Тепер будуть записані фактичні дані асоційовані з об'єктом:

- 00 00 00 0A: 10, значення parentVersion.
- 00 00 00 42:66, значення version в SerialTest.

Далі записана інформація про об'єкт класу *contain. contain con = new contain().*

Для цього робиться опис класу *contain*.

- 0x73: TC_OBJECT, позначає новий об'єкт.
- 0x72: TC_CLASSDESC, позначає новий клас.
- 00 07: Довжина ім'я класу.
- 63 6F 6E 74 61 69 6E: contain, ім'я класу.
- FC BB E6 0E FB CB 60 C7: SerialVersionUID, ідентифікатор цього класу.
- 0x02: Різні прапори. Цей прапор позначає що клас підтримує серіалізацію.
- 00 01: Число полів у класі. Далі йде опис єдиного поля класу conatin, int containVersion = 11;
- 0x49: Код типу поля. 49 позначає «I», що закріплене за Int.
- 00 0E: Довжина ім'я поля.
- 63 6F 6E 74 61 69 6E 56 65 72 73 69 6F 6E: containVersion, ім'я поля.
- 0x78: TC_ENDBLOCKDATA, кінець опционального блоку даних для об'єкта.

Далі перевіряється, чи має *contain* батьківський клас. Якщо має, то алгоритм починає запис цього класу; але в цьому випадку суперкласу в contain немає, і алгоритм записує *TC_NULL.* *0x70: TC_NULL.*

Наприкінці записуються фактичні дані асоційовані з об'єктом класу conatin:

- 00 00 00 0B: 11, значення containVersion.

У майбутньому серіалізація буде замінена форматом XML.

10.2. XML

Мова розмітки XML (Extensible Markup Language) була розроблена W3C. Головною перевагою XML є сумістність даних, представлених у цьому форматі, з різними додатками. Мова XML була розроблена на базі універсальної мови розмітки SGML. Мова HTML, як мова розмітки гіпертекстових документів, також з’явилась з SGML.

Основна ідея XML - це текстове подання за допомогою тегів, структурованих у вигляді дерева даних. Деревоподібна структура добре описує бізнес-об'єкти, конфігурацію, структури даних тощо. Крім того, подання даних у вигляді XML легке для читання.

10.3. DTD

Для опису структури XML-документа використовується DTD (Document Type Definition). DTD визначає, які теги (елементи) можуть використовуватися в XML - документі, як ці елементи зв'язані (наприклад, указувати на те, що елемент *<book>* включає дочірні елементи *<price>* і *<author>),* які атрибути має той або інший елемент.

Будувати DTD для XML-документа не обов'язково, програми-аналізатори будуть обробляти XML-Файл і без DTD. Але в цьому випадку автор повинен правильно його сформувати.

Для того щоб сформувати DTD, можна створити або окремий файл і описати в ньому структуру документа, або включити DTD-опис безпосередньо в документ XML. У першому випадку маємо:

```
<?xml version="1.0" standalone="yes" ?>
<! DOCTYPE journal SYSTEM "book.dtd">
```

У другому випадку опис елемента міститься в XML-Документ:

```
<?xml version="1.0" ?>
...
<! DOCTYPE book [
<!ELEMENT book (price, author)>
...
]>
```

10.4. ОПИС ЕЛЕМЕНТА

Елемент в DTD описується за допомогою дескриптора *!ELEMENT*, у якому вказується назва елемента і його вміст.

Наприклад, визначимо елемент *<book>* у якого є дочірні елементи *<price>* і *<author>*, тоді одержимо опис:

```
<!ELEMENT price PCDATA>
<!ELEMENT author PCDATA>
<!ELEMENT book (price, author)>
```

У цьому випадку були визначені два елементи *price* і *author* і описаний їхній уміст за допомогою маркера *PCDATA*.

Маркер *PCDATA* (parseable character data) указує аналізатору, що елементи можуть містити будь-яку інформацію, з якої може працювати програма-аналізатор.

Крім маркера *PCDATA*, є також такі маркери:

- **EMPTY -** елемент порожній,
- **ANY** - вміст документа спеціально не описується.

При описі елемента *<book>,* було зазначено, що він складається з дочірніх елементів *<price>* і *<author>*. Можна розширити цей опис за допомогою символів '+', '*', '?', що використовуються для вказівки кількості входжень елементів. Так, наприклад, *<!ELEMENT book (price, author+, caption?)>* означає, що елемент *book* містить один і тільки один елемент *price*, трохи (мінімум один) елементів *author* і необов'язковий елемент *caption* (символ * указує на те, що в складі елемента може міститися будь-яка, у тому числі й нульова кількість елементів). Якщо існує кілька варіантів складових елементів, то використовується символ '|'. Наприклад: *<!ELEMENT book (PCDATA | body)>* у цьому випадку елемент *book* може містити або дочірній елемент *body*, або *PCDATA*.

10.5. ОПИС АТРИБУТІВ

Атрибути елементів описуються за допомогою дескриптора *!ATTLIST*, усередині якого задаються ім'я атрибута, тип значення, додаткові параметри:

```
<!ATTLIST article
id ID #REQUIRED
about CDATA #IMPLIED
type (actual | review | teach ) 'actual' >
```

У цьому випадку в елемента *<article>* визначаються три атрибути: *id*, *about*, *type*. Існує декілька можливих значень атрибута, це:

- **CDATA** – значенням атрибута є будь-яка послідовність символів;
- **ID** – визначає унікальний ідентифікатор елемента в документі;
- **IDREF** (**IDREFS**) – значенням атрибута буде ідентифікатор (список ідентифікаторів), певний у документі;
- **ENTITY** (**ENTITES**) – містить ім'я зовнішньої сутності (кілька імен, розділених комами);
- **NMTOKEN** (**NMTOKENS**) – слово (кілька слів, розділених пробілами).

Значенням атрибута також може бути перерахування. Набір припустимих значень записують в круглі дужки (значення1|значення2|значение3), потім вказується значення за замовчуванням 'значення1' .

Значення за замовчуванням можуть бути наступними:

#REQUIRED – означає, що значення повинне бути присутнім у документі;

#IMPLIED – означає, що якщо значення атрибута не задано, то додаток повинен використовувати своє власне значення за замовчуванням;

#FIXED – означає, що атрибут може приймати лише одне значення, те, що зазначене в DTD.

Якщо в документі атрибуту не буде привласнено ніякого значення, то його значення буде дорівнює заданому в DTD.

10.6. ВИЗНАЧЕННЯ СУТНОСТІ

Сутність являє собою деяке визначення, чий зміст може бути повторно використане в документі. Описується сутність за допомогою дескриптора *!ENTITY*:

```
<!ENTITY company 'Sun Microsystems'>
...
```

```
<sender>&company;</sender>
...
```

Програма-аналізатор, що буде обробляти файл, автоматично підставить значення Sun Microsystems замість *&company*.

В XML включено кілька внутрішніх визначень:

< – символ <;

> – символ >;

& – символ &;

&apos – символ апострофа ‘;

" – символ подвійних лапок “.

Тобто оголошення міститься між послідовностями символів *"<!ENTITY" і ">"* і може бути представлене в одному з перерахованих нижче форматів.

Внутрішній примітив. Ім'я примітива ставиться у відповідність значенню, що використовується у XML-документі. Якщо в складі документа зустрічається вираження *&ім'я_примітива*, воно заміняється значенням, пов'язаним з ім'ям.

```
<! ENTITY ім'я_примітива "значення_примітива">
```

Зовнішній примітив XML. Ім'я примітива ставиться у відповідність XML-документ, URL якого вказується при визначенні примітива. Якщо в складі документа зустрічається вираження *&ім'я_примітива;,* воно заміняється вмістом XML-документа. У процесі читання документа видбувається його розбір *<!ENTITY ім'я_примітива SYSTEM " URL">*.

Зовнішній двійковий примітив. Ім'я примітива ставиться у відповідність набір двійкових даних із заданим URL. Якщо при обробці документа зустрічається вираз *&ім'я примітива;,* воно заміняється даними, тип яких зазначений в оголошенні примітива. Так, якщо в оголошенні заданий тип GIF87A, це означає, що URL указує на двійковий файл у форматі GIF. У процесі читання розбір змісту зовнішнього двійкового примітива не вирконується й він може бути використаний лише в атрибуті елемента *<!ENTITY ім'я_примітива SYSTEM "URL" NDATA тип_даних>*.

Приклад:

```
<!ENTITY logotype SYSTEM "/image.gif" NDATA GIF87A>
```

Примітив параметра. Ім'я примітива зв'язується зі значенням, що може бути використано лише в складі DTD (але не в XML-документі). Перед ім'ям такого примітива вказується символ "%", а посилання на значення примітива задаються в DTD у вигляді %ім'я примітива DTD).

<!ENTITY % ім'я примітива DTD "значення_примітива_DTD">

Нехай існує XML-Документ, що містить дані адресної книги:

```
<?xml version="1.0"?>
   <!DOCTYPE notepad SYSTEM "notepad.dtd">
   <notepad>
    <note login="rom">
     <name>Valera Romanchik</name>
     <tel>217819</tel>
     <url>http://www.bsu.by</url>
   <address>
     <street>Main Str., 35</street>
     <city>Kiev</city>
    <country>UKR</country>
   </address>
 </note>
<note login="goch">
  <name>Igor Blinov</name>
  <tel>430797</tel>
   <url>http://bsu.iba.by</url>
  <address>
   <street>Deep Forest, 7</street>
    <city>Polock</city>
    <country>VCL</country>
  </address>
  </note>
</notepad>
Тоді DTD файл буде мати вигляд:
<?xml version="1.0" encoding=" UTF-8"?>
  <!ELEMENT notepad (note+)>
  <!ELEMENT note (name,tel,url,address)>
  <!ELEMENT address (street,city,country)>
  <!ATTLIST note login ID #REQUIRED>
  <!ELEMENT name PCDATA>
  <!ELEMENT tel PCDATA>
  <!ELEMENT street PCDATA>
  <!ELEMENT city PCDATA>
  <!ELEMENT country PCDATA>
  <!ELEMENT url PCDATA>
```

10.7. XLINK

Гіпертекстові засоби XML базуються на понятті ресурсу. Ресурсами вважаються фрагменти даних, що адресуються (наприклад, файли, зображення й додатки). Зв'язки або посилання відображають відносини між ресурсами. Локальним ресурсом називається ресурс, що включається до складу поєднуючого XML-елементу. Вилучений ресурс - це фрагмент даних, на який вказує XML-посилання.

Поєднуючими елементами називаються XML-елементи, до складу яких входять посилання. Для реалізації посилання використовується атрибут з ім'ям *xlink:type*.

Стандартні значення атрибута *xlink:type*, які визначені специфікацією *XLink*, перераховані нижче:

- **Simple** - створює простий поєднуючий елемент, що визначає зв'язок між локальним і віддаленим ресурсом;
- **Extended** - створює розширений поєднуючий елемент, що визначає зв'язок між декількома ресурсами;
- **Locator -** елемент, що визначає адрес ресурсу. У складі разширеного посилання типу *extended* повинен бути заданий хоча б один елемент *locator;*
- **Arc** - визначає правила переходу між посиланнями, що знаходяться в розширеному сполучному елементі (типу *extended*);
- **Title -** *з*адає символьний опис розширеного посилання;
- **Resource -** визначає локальний ресурс для зовнішнього посилання. Якщо зовнішнє посилання використовує елемент типу resource, воно називається внутрішнє (inline).

10.8. ПРОСТІ ПОСИЛАННЯ

Просте посилання встановлює зв'язок між локальним і зовнішнім ресурсами. Формат простого посилання має вигляд:

```
<ім'я_елемента xlink:type="simple" xlink:href="URL”>
...</ім'я_елемента>
```

або

```
<ім'я елемента xlink:type="simple" xlink:href="URL"/>
```

Для створення простого посилання використовується значення *simple* атрибута *xlink:type*. Як значення атрибута *xlink:href* вказується URL ресурсу. Наприклад:

```
<homepage xlink:type="simple"
xlink:href="http://www.google.com">Google Home </homepage>
```

Прості посилання багато в чому нагадують посилання, що використовуються в HTML-документах. Сполучний елемент, побудований за допомогою простого посилання, може відображатися в програмах перегляду (наприклад, в Web-Броузерах), а шляхом активізації посилання (наприклад, по клацанню миші) можна звернутися віддаленого ресурсу.

10.9. АТРИБУТИ, ЩО ВИЗНАЧАЮТЬ ПОВЕДІНКУ ПОСИЛАНЬ

Крім атрибутів, що використовуються для зв'язування ресурсів, існують атрибути, що дозволяють визначити додаткові характеристики поеднаних елементів:

- **xlink:title -** атрибут задає символьне ім'я віддаленого ресурсу;
- **xlink:role -** атрибут визначае призначення віддаленого ресурсу. Значення атрибута xlink:role використовується при обробці документа. У складі поєднуючого елемента можуть також бути присутнім атрибути, що задають поводження посилання;
- **xlink:show -** задає порядок відображення ресурсу, на який вказує посилання. Значення *xlink: show="new"* повідомляє про те, що віддалений ресурс повинен відображатися в новому вікні. Якщо задано значення *xlink:show="replace",* віддалений ресурс заміщає вміст поточного вікна. Значення **xlink:show="embed"** говорить про те, що ресурс, на який вказує посилання, повинен вбудовуватися в поточний документ.

- **xlink:actuate -** задає умови активації посилання. Наприклад, якщо в документі задане значення *xlink:actuate="onLoad",* це означає, що додаток повинен завантажити ресурс відразу після розбору посилання. Значення *link:actuate="onRequest"* вказує на те, що ресурс повинен завантажуватися після того, як користувач активує посилання (наприклад, клацне на ньому мишею).

10.10. РОЗШИРЕНІ ПОСИЛАННЯ

Розширені посилання можуть використовуватися для створення множинних зв'язків. Якщо один зі зв'язків указує на локальний елемент, посилання називається внутрішнім. Якщо всі зв'язки вказують на віддалені ресурси, посилання називається зовнішнім.

Для ідентифікації розширеного посилання використовується значення extended атрибута *xlink:type.* Локальний ресурс вказується в складі розширеного посилання як елемент із атрибутом *xlink:type="resource".*

Віддалений ресурс, що перебуває за межами розширеного посилання (наприклад, у складі іншого XML-документу), представляється за допомогою підлеглого елемента з атрибутом *xlink:type="locator".* Для ідентифікації ресурсу в елементі такого типу використовується URL, що задається як значення атрибута *xlink:href*. Крім того, в елементі типу *locator* можуть бути присутнім інші стандартні атрибути, що визначають семантику й поводження посилання.

Розглянемо приклад документа, що містить три посилання на віддалені ресурси, що описує різновиди продукту. Значення атрибута *xlink:role* можуть бути використані для вибору ресурсу що найбільш відповідає інтересам замовника:

```
  <sweaters xlink:type="extended" xlink:title="BeeShirts.com
Sweaters>
  <sweaterAd xlink:type="resource" xlink:role="sweaters">
BeeShirts has sweater styles to suit you </sweaterAd> <default
xlink:type="locator"
xlink:href="http://www/beeshirts.com/sweaters/standart/"
xlink:role="standard"/>
```

```
<option
xlink:type="locator"
xlink:href="http://www.beeshirts.com/sweaters/calsic/"
xlink:title="Classic Sweater Style"
xlink:role="classic"/>
<option
  xlink:type="locator"
xlink:href="http://www.beeshirts.com/sweaters/sports/"
  xlink:title»"Sporty Sweater Style"
  xlink:role="sports"/>
</sweaters>
```

Оскільки в розширеному посиланні задаються зв'язки з різними ресурсами, існує кілька варіантів переходу від одного ресурсу до іншого. Правила переходу задаються за допомогою атрибута *xlink:type*, що належать підлеглому елементу, що перебуває в складі посилання. Значенням атрибута в цьому випадку є “arc”. В arc-елементі, як і в інших елементах, можуть бути присутнім атрибути *xlink:show* і *xlink:actuate*, що задають поведінку посилання. Крім того, в елементі даного типу вказуються атрибути *xlink:from* і *xlink:to*, значення яких повинні відповідати значенням атрибута *xlink:role* інших елементів.

Приклад:

```
<sweaters xlink:type="extended" xlink:title="BeeShirts.com
Sweaters">
...
<selection
    xlink:type="arc"
    xlink:from="sweaters" xlink:to="classic"
xlink:show="replace" xlink:actuate="onRequest"/>
</sweaters>
```

РОЗДІЛ 11
11.1. СТОРІНКИ СТИЛІВ XML

Мова XSL дозволяє перетворювати XML-документи в документи інших типів, а також задає порядок їхнього форматування.

Принципи перетворення XML-документів описані в розділі XSL Transformations (XSLT) специфікації XSL. Засоби, обумовлені XSLT, дозволяють конвертувати XML-код в HTML, PDF, ASCII-Текст і інші типи документів.

Для форматування документів у мові XSL передбачені спеціальні об'єкти й властивості. Вони повідомляють програму про те, як необхідно розмістити елементи XML-документа на сторінці.

Елементи, що використовуються для форматування, мають стандартні імена, перед якими вказується префікс простору імен *fo*:

Додаток, що підтримує XML і XSL, читає документ і пов'язаний з ним лист стилів, а потім перетворює документ за правилами заданим за допомогою XSL-виразів. XSL-додаток представляє вміст XML-документа у вигляді деревоподібної структури, а потім по цій структурі будує цільовий документ. Зокрема, цільовим може бути неструктурований документ наприклад HTML-сторінка або текстовий файл.

Сторінки стилів XSL, як і документ, з яким вони зв'язуються, представляються у форматі XML. Сторінки стилів містять шаблони, що описують деревоподібну структуру вихідного документа.

Кореневий елемент листа стилів XSL з ім'ям *xsl:stylesheet* містить атрибут *xmlns:xsl.* Елементи *xsl:template* з атрибутами визначають правила, за якими устанавлюється відповідність між елементами XML і шаблонами листа стилів.

Інструкції з обробки фрагментів, що відповідають шаблонам, задаються за допомогою підлеглих елементів, що входять до складу *xsl:template*.

Імена цих підлеглих елементів починаються із префікса *xsl:*. Інші компоненти елементів *xsl:template* представляють XML-вирази та дані, що використовуються при формуванні вихідного XML-документу.

Формат листа стилів XSL має вигляд:

```
<?xml version="1.0"?>
<xsl:stylesheet   xmlns:xsl="http://www.w3.org/TR/ WD-xsl”>
<xsl:template match="ім'я_елемента">
<!— Дії для елемента з ім'ям ім'я_елемента —>
<!— вибір підлеглого елемента з ім'ям підлеглий_елемент —>
<xsl: apply-template select="підлеглий_елемент"/>
<!- Вказівка стилю ---і> </xsl:template>
</xsl:stylesheet>
```

Посилання на лист стилів XSL, що знаходяться в XML-документі, має такий вигляд.

```
<?xml version="1.0"?>
<? xml-stylesheet type="text/xml‟ href="XSLdocumentName.xsl"?>
...
```

Розглянемо приклад з використанням XML. Створимо XML файл *my.xml*.

```
<?xml version="1.0"?>
   <steps>
       <step>
           <name>Step1</name>
      </step>
      <step>
          <name>Step2</name>
     </step>
     <step>
        <name>Step3</name>
    </step>
 </steps>
```

На екрані браузера маємо (Рис.11.1):

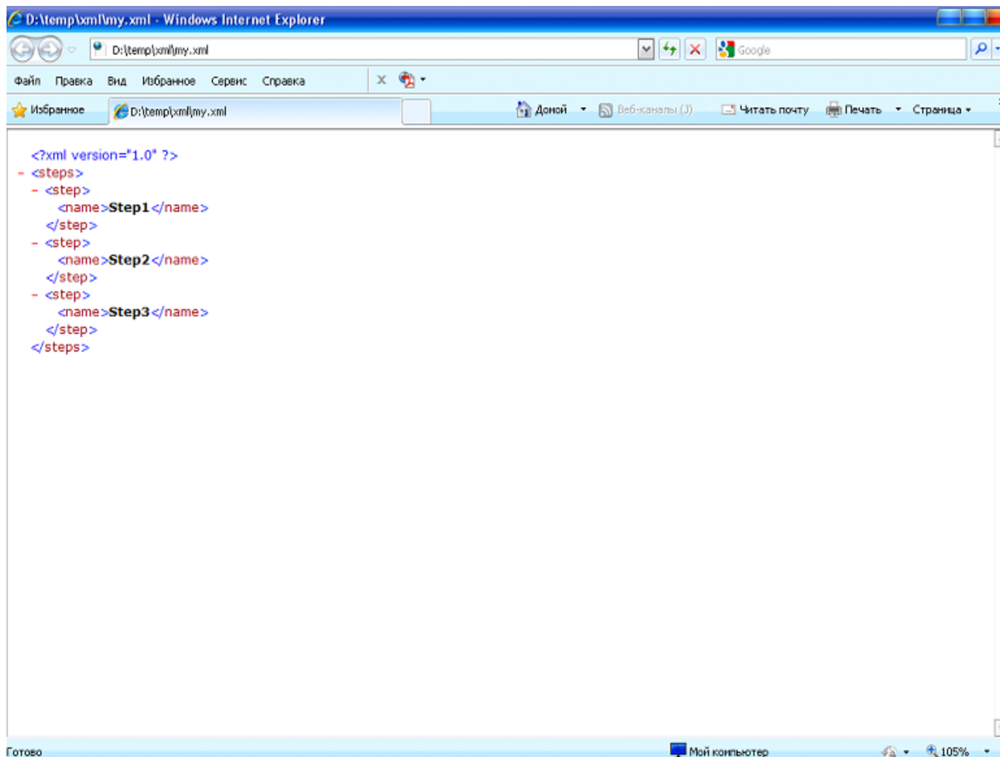


Рис.11.1 Відображення XML в броузері

При відображенні xml файлу можна використовувати технологію CSS. Тоді у файл *my.xml* буде мати вигляд:

```
<?xml version="1.0"?>
<? xml-stylesheet type="text/css" href="my.css"?>
 <steps>
     <step>
     <name>Step1</name>
     </step>
     <step>
     <name>Step2</name>
     </step>
     <step>
     <name>Step3</name>
     </step>
</steps>
```

Файл *my.css* буде мати вигляд:

```
step
  {
      COLOR: blue;
      DISPLAY: block
  }
```

На екрані одержимо (Рис. 11.2):

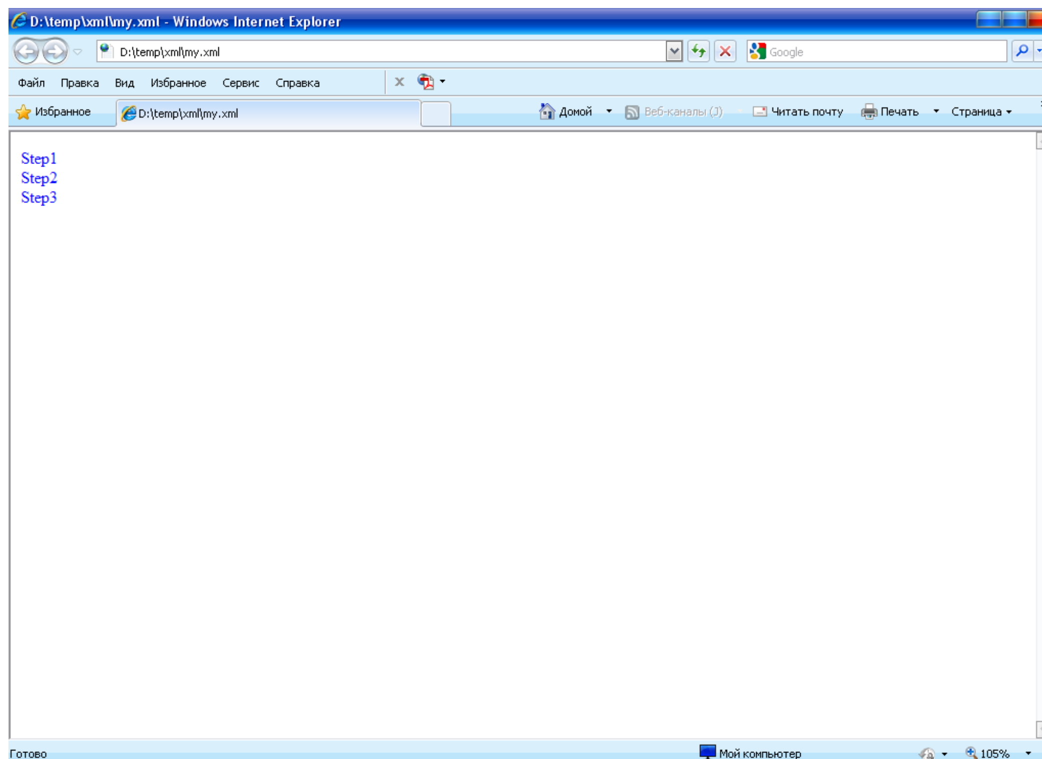


Рис.11.2. XML з CSS

XML файл із DTD визначенням має вигляд:

```
<?xml version="1.0"?>
    <!DOCTYPE step
     [
         <!ENTITY home "www.firststeps.ru">
     ]
    >
  <steps>
        <step>
           <name>Step1</name>
            <author>&home;</author>
       </step>
   <step>
          <name>Step2</name>
          <author>&home;</author>
   </step>
   <step>
      <name>Step3</name>
      <author>&home;</author>
   </step>
  </steps>
```

У браузері цей файл має вигляд (Рис.11.3):

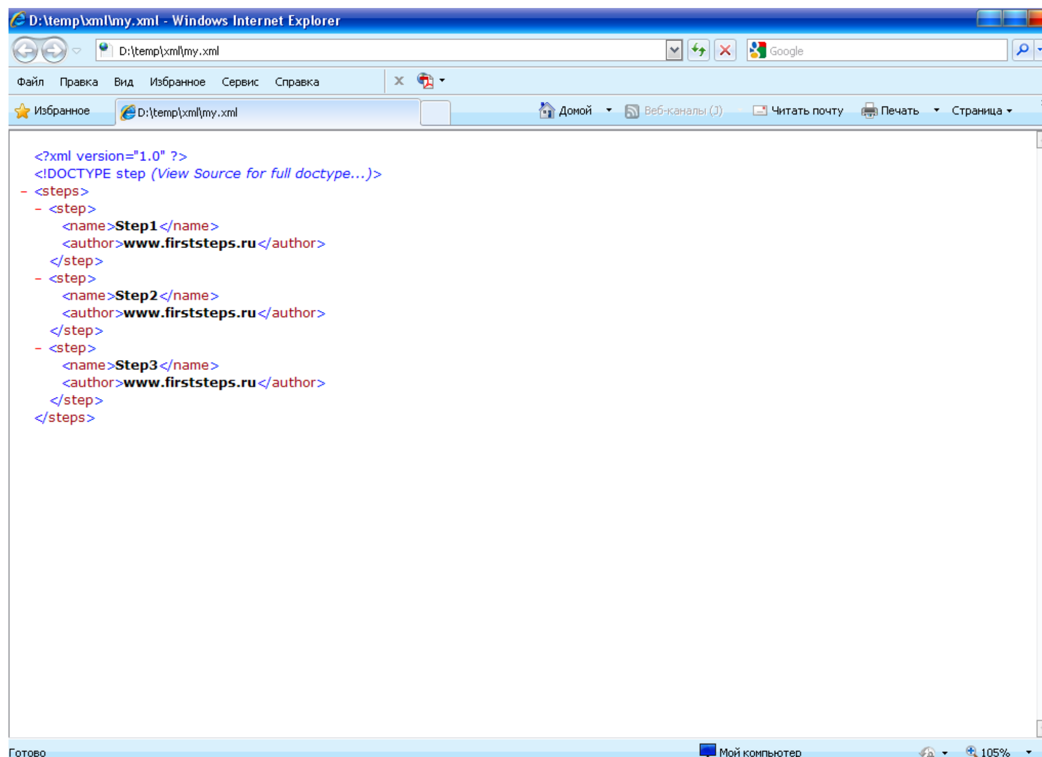


Рис.11.3 XML з DTD

Розглянемо використання замість *css xsl*. Для цього в xml файлі зробимо посилання на файл *my.xsl*.

```
<?xml version="1.0"?>
<? xml-stylesheet type="text/xsl"  href="my.xsl"?>
  <note>
            <head>Step1</head>
            <head>Step2</head>
   </note>
```

Файл *my.xsl* має вигляд:

```
 <?xml version="1.0"?>
   <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/ WD-xsl">
   <xsl:template match="/">
    <html>
     <body>
      <H1>Hello</H1>
      <span style="color:red"> <xsl: apply-templates/>  </span>
    </body>
  </html>
</xsl:template>
<xsl:template match="head">
   <h1>Hello2</h1>
   <xsl: apply-templates> </xsl: apply-templates>
</xsl:template>

<xsl:template match="note">
  <h1>Hello1</h1>
   <xsl: apply-templates> </xsl: apply-templates>
```

```
</xsl:template>

<xsl:template match="text()">
  <h1>Hello3</h1>
   <xsl: value-of select="."/>
</xsl:template>

</xsl:stylesheet>
```

Браузер відобразить наступне (Рис. 11.4.):

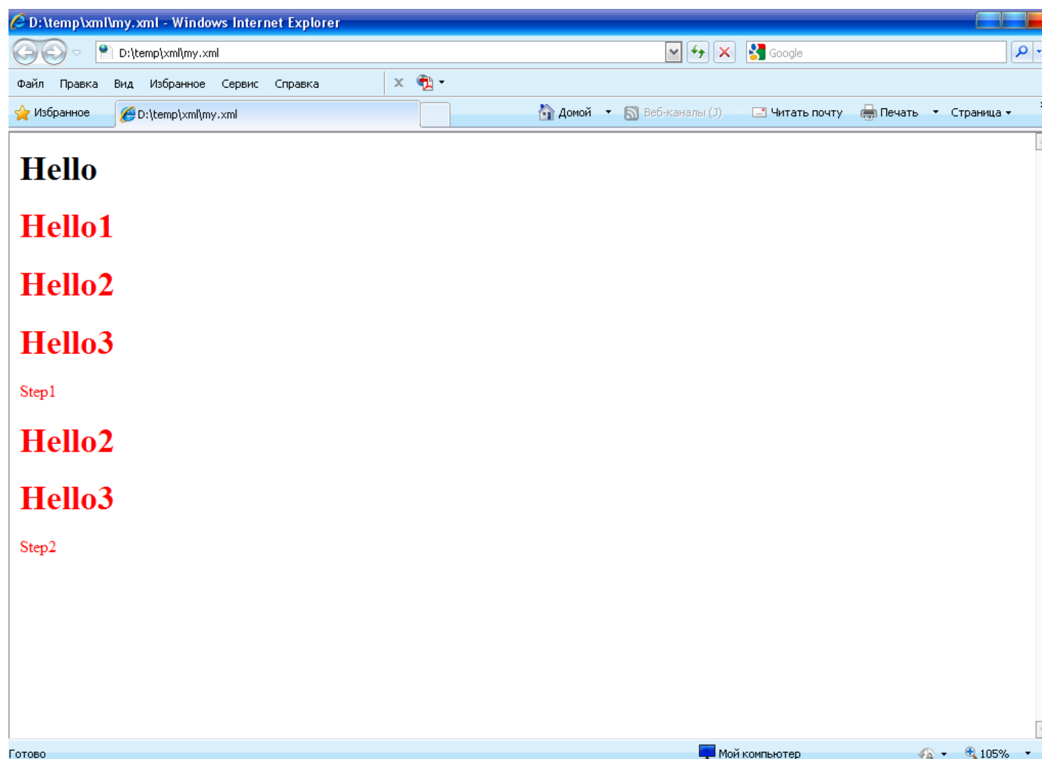


Рис. 11.4. XML з XSL

11.2. XML-АНАЛІЗАТОРИ

Кожна програма, що працює з XML, використовує аналізатор, що являє собою деякий компонент, що перебуває між додатком і файлами XML.

Документи XML можуть бути або *well-formed*, або *valid*. Документи *wellformed* складені відповідно до синтаксичних правил побудови XML-Документів. Документи не тільки сформовані синтаксично правильно, але й додержуються деякої структури, що описана в DTD.

Відповідно є валідні й невалідні аналізатори. І ті, і інші перевіряють XML-документ на відповідність синтаксичним правилам, але тільки валідні

аналізатори знають, як перевірити XML-Документ на відповідність структурі, описаної в DTD.

Ніякого зв'язку між видом аналізатора й видом XML-документа немає. Валідний аналізатор може розібрати XML-документ, для якого немає DTD, і, навпаки, невалідний аналізатор може розібрати XML-документ, для якого є DTD.

Існує два види взаємодії додатка й аналізатора: використовувати модель, засновану на поданні вмісту файлу XML у вигляді дерева об'єктів, або подійну модель.

Аналізатори, які будують деревоподібну модель, - це DOM-аналізатори (Dynamic Object Model). Аналізатори, які генерують події, - це SAX-аналізатори (Simple API for XML).

У першому випадку аналізатор будує в пам'яті дерево об'єктів, що відповідає XML-документу. Далі вся робота ведеться саме із цим деревом.

У другому випадку аналізатор працює в такий спосіб: коли відбувається аналіз документа, аналізатор генерує події, пов'язані з різними ділянками XML-файлу, а програма, що використовує аналізатор, вирішує, як реагувати на ці події. Так, аналізатор буде генерувати події про те, що він зустрів початок документа або його кінець, початок елемента або його кінець, символьну інформацію усередині елемента тощо.

DOM-Аналізатори варто використовувати тоді, коли потрібно знати структуру документа й може знадобитися змінювати цю структуру або використовувати інформацію з XML-файлу кілька разів.

SAX-Аналізатори використовуються тоді, коли потрібно витягти інформацію про декілька елементів з XML-файлу або коли інформація з документа потрібна тільки один раз.

РОЗДІЛ 12
12.1. SAX-АНАЛІЗАТОРИ

SAX API визначає ряд подій, які будуть сгенеровані при розборі документів:

- **startDocument** – подія, що сигналізує про початок документа;
- **endDocument** – подія, що сигналізує про завершення документа;
- **startElement** – дана подія буде сгенерировано, коли аналізатор повністю обробить вміст відкриваючого тегу, включаючи його ім'я й всі атрибути, що втримуються;
- **endElement** – подія, що сигналізує про завершення елемента;
- **characters** – подія, що сигналізує про те, що аналізатор зустрів символьну інформацію усередині елемента;
- **warning**, **error**, **fatalError** – ці події сигналізують про помилки при розборі XML-документа.

У пакеті *org.xml.sax.helpers* знаходиться клас *DefaultHanlder*, що має методи для обробки всіх вищевказаних подій.

Для створення додатка, якій обробляє XML файл необхідно:

1. Створити клас, суперкласом якого буде *DefaultHandler*, і перевизначити методи, відповідальні за обробку подій, що цікавлять.
2. Створити об'єкт-парсер класу *org.xml.parsers.SAXParser*.
3. Викликати метод *parse()*, якому як параметри передати імена файлу, що розбирається, і екземпляр створеного на першому кроці класу.

Розглянемо приклад: розбір документа notepad.xml, що має вигляд:

```
<?xml version="1.0"?>
   <!DOCTYPE notepad SYSTEM "notepad.dtd">
<notepad>
 <note login="rom">
       <name>Valera Romanchik</name>
       <tel>217819</tel>
      <url>http://www.bsu.by</url>
  <address>
     <street>Main Str., 35</street>
     <city>Kiev</city>
     <country>UKR</country>
  </address>
```

```
 </note>
 <note login="goch">
 <name>Igor Blinov</name>
  <tel>430797</tel>
   <url>http://bsu.iba.by</url>
 <address>
  <street>Deep Forest, 7</street>
  <city>Polock</city>
  <country>VCL</country>
</address>
</note>
</notepad>
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import java.net.URL;
import java.net.MalformedURLException;
import java.util.Vector;
    interface ConstNote
{
       int NAME = 1, TEL = 2, URL = 3,
       STREET = 4, CITY = 5, COUNTRY = 6;
 }
class DocHandler extends DefaultHandler implements ConstNote
   {
     Vector notes = new Vector();
     Note curr = new Note();
     int current = -1;
     public Vector getNotes() { return notes; }
     public void startDocument() {
       System.out.println("parsing started");
     }
     public void endDocument(){System.out.print("");}
     public void startElement(String uri, String localName,
String qName, Attributes attrs) {
      if (qName.equals("note")) {
       curr = new Note();
       curr.setLogin(attrs.getValue(0));}
     if (qName.equals("name"))  current = NAME;
      else if (qName.equals("tel")) current = TEL;
              else if (qName.equals("url")) current = URL;
                      else if (qName.equals("street"))
                          current = STREET;
                             else if (qName.equals("city"))
                                 current = CITY;
                                else if(qName.equals("country"))
                                    current = COUNTRY;
}
public void endElement(String uri, String localName, String
qName)
{ if (qName.equals("note")) notes.add(curr);}
```

```
public void characters(char[] ch, int start, int length) {
String s = new String(ch, start, length);
   try{
     switch (current) {
       case NAME: curr.setName(s); break;
       case TEL: curr.setTel(Integer.parseInt(s)); break;
       case URL: try { curr.setUrl(new URL(s));}
                 catch (MalformedURLException e) {}; break;
       case STREET:curr.address.setStreet(s); break;
       case CITY: curr.address.setCity(s); break;
      case COUNTRY: curr.address.setCountry(s);break;} }
   catch (Exception e) { System.out.println(e);} } }
public class MyParserDemo {
  public static void main(String[] args) {
  try { SAXParser parser =
        SAXParserFactory.newInstance().newSAXParser();
          DocHandler dh = new DocHandler();
           Vector v;
            if (dh != null) parser.parse("notepad.xml", dh);
            v = dh.getNotes();
            for (int i = 0; i < v.size(); i++)
      System.out.println(((Note) v.elementAt(i)).toString());
   }
  catch (Exception e) { e.printStackTrace();} } }
```

У результаті на консоль буде виведена наступна інформація:

```
      parsing started
      rom
      Valera Romanchik 217819 http://www.bsu.by
      address:Main Str., 35 Kiev UKR
      goch
      Igor Blinov 430797 http://bsu.iba.by
      address:Deep Forest, 7 Polock VCL
```

Клас *Note* має вигляд:

```
import java.net.URL;
 class Note {
       private String name, login;
       private int tel;
       private URL url;
       public Address address = new Address();
   public void setAddress(Address address)
     { this.address = address; }
   public void setLogin(String login)
     { this.login = login;}
   public void setName(String name)
     { this.name = name; }
   public void setTel(int tel)
     { this.tel = tel; }
public String toString() {
     return login + " " + name + " " + tel + " "
     + url + "\n\t address:" + address.street + " "
```

```
    + address.city + " " + address.country; }
class Address {
     String street, city, country;
public void setCity(String city) { this.city = city;}
public void setCountry(String state) { this.country = state; }
public void setStreet(String street){ this.street = street;}
 }
 public void setUrl(URL url) { this.url = url;} }
```

Розглянемо докладніше інтерфейс методу *startElement*.

```
void startElement(String uri, String localName, String qName,
Attributes attrs)
```

Тут:

Кваліфіковане ім'я або qName - це комбінація інформації простору імен, якщо воно існує, і властиво ім'я елемента. *qName* також містить у собі знак двокрапка (:), якщо він є - наприклад, *revised:response*;

URI простору імен - наприклад, в *revised:respo*nse це *revised*;

Локальне ім'я localName - це дійсне ім'я елемента, таке, як *note*. Якщо документ не надає інформацію про постори імен, парсер не зможе визначити, якою частиною *qName* є *localName*;

Будь-які атрибути attrs - атрибути для елементів передаються як коРОЗДІЛ об'єктів, так як показано в попередній лекції.

Розглянемо приклад створення простору імен в XML файлі. Нехай є XML файл:

```
  <?xml version="1.0"?>
<surveys xmlns="http://www.nicholaschase.com/surveys/"
xmlns:revised="http://www.nicholaschase.com/surveys/revised/">
<response username="bob">
 <question subject="appearance">A</question>
 <question subject="communication">B</question>
 <question subject="ship">A</question>
 <question subject="inside">D</question>
 <question subject="implant">B</question>
 <revised:question subject="appearance" revised:subject="looks">
     D</revised:question>
   <revised:question
subject="communication">A</revised:question> <revised:question
subject="ship">A</revised:question> <revised:question
subject="inside">D</revised:question> <revised:question
subject="implant">A</revised:question> </response> <response
username="sue"> ...
```

Варто пам'ятати, що атрибути ніколи не знаходяться у просторі імен за замовчуванням. Метод *startElement* у цьому випадку буде мати вигляд:

```
public void startElement( String namespaceURI, String localName,
String qName, Attributes atts) throws SAXException
 {
if (namespaceURI == "http://www.nicholaschase.com/surveys/")
    {  if (localName == "question")
       { thisQuestion = atts.getValue("subject"); }
    }
   thisElement = localName; } //thisElement є полем класу.
```

Список *Attributes* має методи, які дозволяють визначати простір імен атрибута, це методи, *getURI()* і *getQName*.

Парсер Java за замовчуванням не буде повідомляти про значення локальних імен, якщо спеціально не включена обробка просторів імен:

```
...
try {
SAXParserFactory spfactory =

SAXParserFactory.newInstance();
spfactory.setValidating(true);
spfactory.setNamespaceAware(true);
SAXParser saxParser = spfactory.newSAXParser(); ...}
 catch(...){...}
```

setNamespaceAware(boolean awareness) – включення інформованості про простір імен;

setValidating(boolean validating) - включення перевірки коректності DTD.

12.2. ДЕРЕВОВИДНА МОДЕЛЬ DOM

DOM (Dynamic object model) являє собою деякий загальний інтерфейс для роботи зі структурою документа. Одна із цілей розробки полягала в тім, щоб код, написаний для роботи з будь-яким DOM-аналізатором, міг працювати й з будь-яким іншим DOM-аналізатором.

DOM-аналізатор будує дерево, що представляє вміст XML-документа, і визначає набір класів, які представляють кожний елемент в XML-документі (елементи, атрибути, сутності, текст тощо).

В Java включена підтримка DOM (пакет org.w3c.dom).

Основним об'єктом DOM є *Node* – деякий загальний елемент дерева. Більшість DOM-об'єктів успадковано саме від *Node*.

Node визначає ряд методів, які використовуються для роботи з деревом:

- **getNodeType()** – повертає тип об'єкта (елемент, атрибут, текст, CDATA тощо), список значень, що повертаються, мають вигляд:

ELEMENT_NODE	1
ATTRIBUTE_NODE	2
TEXT_NODE	3
CDATA_SECTION	4
ENTITY_REFERENCE_NODE	5
ENTITY_NODE	6
PROCESSING_INSTRUCTION_NODE	7
COMMENT_NODE	8
DOCUMENT_NODE	9
DOCUMENT_TYPE_NODE	10
DOCUMENT_FRAGMENT_NODE	11
NOTATION_NODE	12

- **getParentNode()** – повертає об'єкт, що є батьком поточного вузла *Node*;
- **getChildNodes()** – повертає список об'єктів, що є дочірніми елементами;
- **getFirstChild()**, **getLastChild()** – повертає перші й останній дочірні елементи;
- **getAttributes()** – повертає список атрибутів даного елемента;
- **getNodeName() і getNodeValue()** - використовується для знаходження ім'я й значення кожного атрибута;
- **setNodeValue() -** встановлює значення вузла;
- **hasChildNodes()** – повертає *true* якщо існують дочірні вузли;
- **getChildNodes() -** повертає дочірні елементи (повертає об'єкт класу **NodeList**);

Інтерфейс *NamedNodeMap:*

- **getLength()** - повертає кількість елементів;
- **item(int)** – витягає елемент із зазначеним індексом. Якщо вузол не містить атрибутів, то вертається null.

Інтерфейс *Document:*

- **getElementsByTagName(String) -** як параметр задається ім'я елемента, метод повертає об'єкт класу *NodeList* (серед його методів також присутні *getLength()* і *item(int)*, як в *NamedNodeMap*), що містить посилання на всі елементи із заданим ім'ям.

Інтерфейс *Element***:**

- **getElementsByTagName(String)** - робота цього методу ідентична вищерозглянутому, з тією різницею, що цей метод шукає елементи тільки серед вкладених елементів поточного вузла. Будемо використовувати аналізатор XML4J від IBM).

```
import org.w3c.dom.Element;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.Text;
import org.apache.xerces.parsers.DOMParser;
import java.net.URL;
import java.util.Vector;
public class MyDOMDemo {
  public static String getValue(Element e, String name) {
    NodeList nList = e.getElementsByTagName(name);
    Element elem = (Element) nList.item(0);
    Text t= (Text) elem.getFirstChild();
    return t.getNodeValue(); }

public static void main(String[] args) {
   Document doc = null;
   DOMParser parser = new DOMParser();
   Vector entries = new Vector();
  try {
    parser.parse("notepad.xml");
    doc = parser.getDocument();
    Element root = doc.getDocumentElement();
    NodeList noteList = root.getElementsByTagName("note");
   Element noteElem;
for (int i = 0; i <noteList.getLength(); i++) {
   noteElem = (Element) noteList.item(i);
```

```
    Myclass e = new Myclass();
    NodeList list = noteElem.getChildNodes();
    Node log = noteElem.getAttributes().item(0);
    e.setLogin(log.getNodeValue());
    e.setName(getValue(noteElem, "name"));
    e.setTel(Integer.parseInt(getValue(noteElem, "tel")));
    e.setUrl(new URL(getValue(noteElem, "url")));
    Element n =
(Element)noteElem.getElementsByTagName("address").item(0);
    e.address.setStreet(getValue(n, "street"));
    e.address.setCountry(getValue(n, "country"));
    e.address.setCity(getValue(n, "city"));
    entries.add(e);}}
  catch (Exception e) {System.out.println(e);}
  for (int i = 0; i < entries.size(); i++)
   System.out.println(((Note)
  entries.elementAt(i)).toString());}}
```

XML-Документи можна не тільки читати, але й корегувати. Розглянемо приклад:

```
import org.jdom.*;
import org.jdom.input.SAXBuilder;
import org.jdom.output.XMLOutputter;
import java.util.*;
import java.io.FileOutputStream;
public class JDOMChanger {
   static void lookForElement(String name, String element,
String content, String login) {
     SAXBuilder builder = new SAXBuilder();
  try {
      Document document = builder.build(name);
      Element root = document.getDocumentElement();
      NodeList c = root.getChildNodes();
  for (int i=0; i<c.getLenght(); i++) {
    Element e = (Element) c.item(i);
  if (e.getAttributeValue("login").equals(login)) {
    e.getChild(element).setText(content);}}
 XMLOutputter serializer = new XMLOutputter();
serializer.output(document, new FileOutputStream(name));
   System.out.flush();}
  catch (Exception e) { System.out.println(e);}}
public static void main(String[] args) {
   String name = "notepad.xml";
JDOMChanger.lookForElement(name, "tel", "09", "rom");
  }}
```

Розглянемо перетворення файлу *notepad.xml* в html файл із використанням *notepad.xsl,* що має вигляд:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:template match="/">
```

```
   <html>
      <head><title>Notepad Contents</title></head>
    <body>
     <table border="1">
     <tr>
      <th>Login</th>
      <th>Name</th>
      <th>Street</th>
     </tr>
    <xsl: for-each select="notepad/note">
      <tr>
       <td><xsl: value-of select="@login"/></td>
       <td><xsl: value-of select="name"/></td>
      <td><xsl: value-of select="address/street"/></td>
    </tr>
   </xsl: for-each>
</table>
 </body></html>
</xsl:template>
</xsl:stylesheet>
```

Відповідний код буде мати вигляд:

```
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamSource;
public class SimpleTransform {
   public static void main(String[] args) {
   try {
 TransformerFactory tFact =

TransformerFactory.newInstance();
 Transformer transformer = tFact.newTransformer(new
StreamSource("notepad.xsl"));
  transformer.transform( new StreamSource("notepad.xml"), new
StreamResult("notepad.html"));} catch (TransformerException e)
{ e.printStackTrace();
}}}
```

У результаті отримаємо HTML-документ наступного виду:

```
<html><head>
<META http-equiv=" Content-Type" content="text/html;
    charset= UTF-8">
  <title>Notepad Contents</title>
    </head>
   <body>
   <table border="1">
   <tr>
    <th>Login</th><th>Name</th><th>Street</th>
    </tr><tr>
```

```
    <td>rom</td><td>Valera Romanchik</td><td>Main Str., 35</td>
    </tr><tr>
    <td>goch</td><td>Igor Blinov</td><td>Deep Forest, 7</td></tr>
</table></body></html>
```

12.3. ТЕХНОЛОГІЯ RMI

Як правило, програми, які використовують RMI складаються із двох різних програм: сервера й клієнта. Програма-сервер при цьому створює об'єкти, робить посилання на ці об'єкти доступними й очікує, поки клієнти не почнуть викликати методи цих об'єктів, а програма-клієнт одержує посилання на один або кілька об'єктів на сервері й викликає їхні методи. Технологія RMI надає механізм, за допомогою якого сервер і клієнт спілкуються й передають один одному інформацію. Такі додатки називають розподіленими об'єктними додатками.

Розподілені об'єктні програми повинні здійснювати наступне:

- **Знаходження вилучених об'єктів.** Програми можуть використовувати різні механізми одержання посилань на вилучені об'єкти. Наприклад, програма може зареєструвати свої вилучені об'єкти за допомогою апарата простих імен - реєстру RMI (RMI registry). Як альтернатива, програма може передавати й повертати посилання на вилучені об'єкти усередині інших вилучених викликів.
- **Спілкування з віддаленими об'єктами.** Деталі взаємозв'язку між вилученими об'єктами керуються RMI. Для програміста віддалене спілкування виглядає майже так само, як і виклик звичайних методів в Java.
- **Завантаження визначень класів, об'єкти яких передаються.** Оскільки RMI дозволяє передавати об'єкти, він також надає механізми завантаження визначень класів і передачі інформації об'єктів.

На рис.12.1 показана структура RMI додатка:

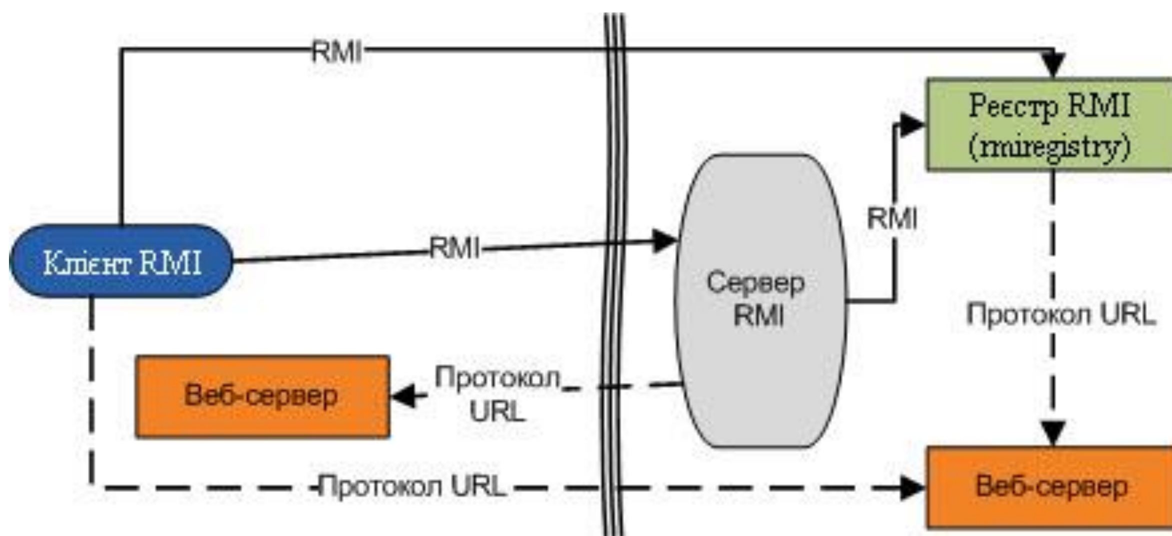


Рис.12.1. Структура RMI

Сервер викликає реєстр для того, щоб зв'язати ім'я з вилученим об'єктом. Клієнт шукає вилучений об'єкт за його іменем в реєстрі сервера й потім викликає його метод.

Реалізація RMI програми включає наступні кроки:

1. Визначення віддалених інтерфейсів. Віддалений інтерфейс визначає методи, які можуть бути віддалено викликані клієнтом. Клієнт програмується під віддалений інтерфейс, а не під класи-реалізації цих інтерфейсів. Проектування таких інтерфейсів включає визначення типів об'єктів, які будуть використовуватися як параметри й значень, що повертаються, для цих методів. Якщо які-небудь із цих інтерфейсів ще не існують, то їх теж потрібно визначити;

2. Реалізація віддалених об'єктів. Віддалені об'єкти повинні реалізувати один або кілька віддалених інтерфейсів. Клас віддаленого об'єкта може включати реалізації інших інтерфейсів і методів, які будуть доступні тільки локально. Якщо який-небудь локальний клас використовується як параметр або значення, що повертається, хоча б одного із цих методів, то вони повинні бути теж реалізовані;

3. Реалізація клієнтів. Клієнти, що використовують віддалені об'єкти, можуть бути реалізовані в будь-який момент після того, як визначені

віддалені інтерфейси, і навіть після того, як віддалені об'єкти були «задеплоени» (deployed).

Розглянемо приклад RMI додатка:

1. **Визначення віддалених інтерфейсів:**

```
  import java.rmi.*;
     public interface AddServerIntf extends Remote{
     double add (double d1, double d2) throws RemoteException;
}
```

2. Реалізація вилучених об'єктів:

```
import java.rmi.*;
import java.rmi.server.*;
/* всі вилучені об'єкти повинні розширювати UnicastRemoteObject,
що забезпечує функціональні можливості на віддалених машинах */
public interface AddServerImpl extends UnicastRemoteObject
implements AddServerIntf{
    public AddServerImpl() throws RemoteException{
    // конструктор серверної реалізації
    }
public double add (double d1, double d2) throws RemoteException{
    return d1+d2; // реалізація методу
}}
```

Реєстрація RMI об'єкта в реєстрі

```
  import java.rmi.*;
  import java.net.*;
   public class AddServer{
     public static void main (String[] args){
   try{
    // створюємо екземпляр «оброблювача»
   AddServerImpl ASI = new AddServerImpl ();
   // метод rebind обновляє RMI-Реєстр і
  // зв'язує ім'я("Plus") з об'єктним посиланням(ASI)
   Naming.rebind ("Plus", ASI); }
  catch(Exception e){ e.printStackTrace ()}}}
```

3. Реалізація клієнтів:

```
  import java.rmi.*;
    public class AddClient{
     public static void main (String[] args){
      // вимагає трьох параметрів командного рядка
      // перший параметр - IP-Адреса або ім'я сервера
     // другі й третій - доданки
     if (args.length != 3) System.exit(1);
     try{       // plusURL - рядок із синтаксисом URL
    String plusURL = "rmi://" + args[0] + "/Plus";
    // одержуємо посилання на віддаленуу реалізацію
     AddServerIntf plus = (AddServerIntf)Naming.lookup(plusURL);
     System.out.println("The first number is: " + args[1]);
```

```
     System.out.println("The second number is: " + args[2]);
    double d1 = Double.valueOf(args[1]).doubleValue();
    double d2 = Double.valueOf(args[2]).doubleValue();
   // Безпосереднє звертання до віддаленогоо методу:
    System.out.println("The sum is: " + plus.add(d1,d2));
    }catch(Exception e){e.printStackTrace () }}}
```

Щоб генерувати заглушки й кістяки, потрібно використовувати інструмент, називаний компілятором RMI:

```
  …\>rmic AddServerImp
```

Дана програма генерує два нових файли:

```
AddServerImpl_Skel.class (кістяк)
```

і

```
AddServerImpl_Stub.class (заглушка)
```

На машинах сервера та клієнта необхідно встановити наступне:

На машині клієнта:

```
AddClient.class, AddServerImpl_Stub.class і AddServerIntf.class
```

На машині сервера:

```
AddServerIntf.class, AddServerImpl.class,
AddServerImpl_Skel.class і AddServer.class
```

Запуск сервера:

```
>java AddServer
```

Запуск клієнта:

```
>java AddClient server 8 9
```

РОЗДІЛ 13
13.1. JDBC

JDBC - це стандартний прикладний інтерфейс (API) мови Java для організації взаємодії між програмою і СУБД. Ця взаємодія здійснюється за допомогою драйверів JDBC, що забезпечують реалізацію загальних інтерфейсів для конкретних СУБД і конкретних протоколів. В JDBC визначаються чотири типи драйверів:

Тип 1 – драйвер, що використовує інший прикладний інтерфейс, зокрема ODBC, для роботи із СУБД (так званий JDBC-ODBC – міст). Стандартний драйвер першого типу *sun.jdbc.odbc.JdbcOdbcDriver* входить в JSDK;

Тип 2 - драйвер, що працює через нативні бібліотеки (тобто клієнта) СУБД;

Тип 3 - драйвер, що працює по мережному й незалежному від СУБД протоколу із проміжним Java-сервером, що, у свою чергу, підключається до потрібної СУБД;

Тип 4 - мережний драйвер, що працює прямо з потрібною СУБД і не потребує установки native-бібліотек.

Перевага природньо віддається другому типу, однак якщо програма виконується на машині, на якій не передбачається установка клієнта СУБД, те вибір виконується між третім і четвертим типами. Причому четвертий тип працює прямо із СУБД по її протоколу, тому можна припустити, що драйвер четвертого типу буде більше ефективним у порівнянні із третім типом з погляду продуктивності. Перший же тип, як правило, використовується рідко, тобто в тих випадках, коли в СУБД немає свого драйвера JDBC, зате є драйвер ODBC.

JDBC надає інтерфейс для розробників, що використовують різні СУБД. За допомогою JDBC відсилаються SQL-запити тільки до реляційних баз даних (БД), для яких існують драйвери, що знають спосіб спілкування з реальним сервером бази даних.

Розглянемо звичайну послідовність дій при роботі з базою даних:

Завантаження класу драйвера бази даних при відсутності екземпляра цього класу (для *JavaDB*):

```
String driverName = "org.apache.derby.jdbc.EmbeddedDriver";
```

Після цього виконується власне завантаження драйвера у пам’ять:

```
Class.forName(driverName);
```

і стає можливим з'єднання із СУБД.

Ці дії можна виконати, імпортуючи бібліотеку й створюючи об'єкт явно (наприклад для *DB2*):

```
import COM.ibm.db2.jdbc.net.DB2Driver;
```

а потім, new *DB2Driver();*

2. Установка з'єднання із БД має вигляд:

```
String connectionURL="jdbc:derby:NeuronNetwork;create=true";
Connection conn = DriverManager.getConnection(connectionURL,
"login", "password");
```

У результаті буде повернутий об'єкт *Connection* і буде одне встановлене з'єднання із БД *NeuronNetwork*.

Клас *DriverManager* надає можливість для керування набором драйверів баз даних. Методу *getConnection()* необхідно передати тип і фізичне місце розташування БД, а також логин і пароль для доступу. За допомогою методу *registerDriver()* драйвера реєструються, а методом *getDrivers()* можна одержати список всіх драйверів.

3. Створення об'єкта для передачі запитів:

```
Statement st = conn.createStatement();
```

Об'єкт класу *Statement* використовується для виконання запиту без його попередньої підготовки. Можуть застосовуватися також оператори для виконання підготовлених запитів і збережених процедур *PreparedStatement* і *CallableStatement.* Створений об'єкт можна використовувати для виконання запиту.

Розглянемо приклад використання *CallableStatement*. Нехай є SQL функція:

```
CREATE OR REPLACE FUNCTION FUNC_GET_ALL_USERS RETURN
SYS_REFCURSOR AS rcs SYS_REFCURSOR;
```

```
BEGIN
    OPEN rcs FOR
         SELECT * FROM USERS;
         RETURN rcs;
END;
Connection connect = null;
      CallableStatement procedure = null;
    try{
    connect = .......;
        try{
procedure = connect.prepareCall("{ ? = CALL
FUNC_GET_ALL_USERS()}");//виклик для функції
   procedure.registerOutParameter(1, Types.OTHER);
   procedure.execute();
   ResultSet rs = procedure.getResultSet();
   while(rs.next()){
     System.out.println(rs.getString("nickname")); } }
     finally{ connect.close();
 }}
     catch(SQLException e)
     { System.out.println(e);
}
```

Очевидно, що можна написати SQL функції, яким передаються деякі параметри, у цьому випадку маємо:

```
CallableStatement cstmt = conn.prepareCall( "{call
getTestData(?, ?)}");
cstmt.registerOutParameter(1,java.sql.Types.TINYINT);
cstmt.registerOutParameter(2,java.sql.Types.DECIMAL, 3);
//3 - до-у знаків після десяткової коми
cstmt.executeQuery();
byte x = cstmt.getByte(1);
BigDecimal n = cstmt.getBigDecimal(2,  3);
```

Якщо параметри sql функції є inout, то тоді одержуємо:

```
CallableStatement cstmt = con.prepareCall( "{call
reviseTotal(?)}");
cstmt.setByte(1, 25);
cstmt.registerOutParameter(1, java.sql.Types.TINYINT);
cstmt.executeUpdate();
byte x = cstmt.getByte(1);
```

Значення, повернуте у вихідний параметр, може бути *NULL*. При цьому методи *getXXX* повертають *null*, *0* або *false*, залежно від типу даних. Єдиним способом довідатися, чи повернула процедура *0*, *false* або *NULL*, є виклик методу *wasNull*, що повертає.

Розглянемо приклад, двовимірний масив на основі бази даних.

```
import java.sql.*;
import java.util.*;
```

```
//тут зберігається ім'я масиву, до-у рядків і до-у стовпців.
class MasSize  {
  private int columns;
  private int rows;
  private String name;
  public void setName(String n){name=new String(n);}
  public void setColumns(int n){columns=n;}
  public void setRows(int n){rows=n;}
  public String getName(){return name;}
  public int getColumns(){return columns;}
  public int getRows(){return rows;}
  public boolean equals(Object o){
     MasSize o1=(MasSize) o;
     if(name.equals(o1.name)) return true;
     else  return false; }  };

public class DB{
    private String name;  //ім'я бази даних
    private LinkedList tables=new LinkedList();
    //список імен масивів у базі даних
    private Connection conn;
    public DB(String preambula, String n, String driver){
      name=new String(n);
      String connectionURL=preambula+" "+name+";create=true";
  try{   Class.forName( driver);   }
    catch(java.lang.ClassNotFoundException e){
    System.out.println("In function DB");
    e.printStackTrace();}
  try {
     conn = DriverManager.getConnection(connectionURL);}
     catch (Exception e){System.out.println("In function
     DB");
     e.printStackTrace();
   }}

public void createMas(String name_of_massive, int
number_of_rows, int number_of_columns){
  try {   //додаємо ім'я масиву в список імен масивів
          MasSize m=new MasSize();
          m.setName(name_of_massive);
          m.setColumns(number_of_columns);
          m.setRows(number_of_rows);
          if(!tables.contains(m)) tables.add(m);
        else  { System.out.println("Massive with such name
          already exists"); return;   }
       //створення таблиці
  String createString="CREATE TABLE "+name_of_massive+" (
  NUMBER int generated by default as identity (START WITH 1,
  INCREMENT BY 1), "+"w0 DOUBLE )";
  Statement stmt = conn.createStatement();
    stmt.executeUpdate(createString);
  for(int i=1; i< number_of_columns; i++)  {
```

```
        //додавання стовпців у таблицю
        String stradd="ALTER TABLE "+name_of_massive+" ADD
        "+"w"+String.valueOf(i)+" DOUBLE";
        stmt.executeUpdate(stradd);    }
      for(int j=0;j<number_of_rows;j++) {
         //додавання рядків у таблицю
         String str_query="insert into "+name_of_massive+" ( w0
         )"+" values  "+"( ?)";
         PreparedStatement psInsert =
         conn.prepareStatement(str_query);
         psInsert.setDouble(1,0.0);
         psInsert.executeUpdate();}}
      catch (Exception e) {
         System.out.println("In function createMas");
         e.printStackTrace();
   }}

public void set(String name_of_massive, int row, int column,
double value){
    //зміна значення елемента з координатами row і column на
значення value
   try{
    //перевірка чи існує масив з ім'ям name_of_massive
        MasSize m=new MasSize();
        m.setName(name_of_massive);
        m.setColumns(0);
        m.setRows(0);
       if(!tables.contains(m)) {System.out.println("there is no
such massive");
             return; }
     String str="update "+name_of_massive+" set

"+"w"+String.valueOf(column)+"=?"+" where

NUMBER="+String.valueOf(row+1);
    PreparedStatement psInsert = conn.prepareStatement(str);
     psInsert.setDouble(1,value);
    psInsert.executeUpdate();}
 catch(Exception e){  System.out.println("In function set");
         e.printStackTrace();} }

 public double get(String name_of_massive, int row, int column){
     //одержання елемента масиву з координатами  row і column
   try{
    MasSize m=new MasSize();
         m.setName(name_of_massive);
         m.setColumns(0);
         m.setRows(0);
        if(!tables.contains(m)){ System.out.println("there is no
such massive");
          return -1; }
```

```
    String str="select "+"w"+String.valueOf(column)+" from
"+name_of_massive+" where NUMBER="+String.valueOf(row+1);
     Statement stmt2 = conn.createStatement();
    ResultSet rs = stmt2.executeQuery(str);
     rs.next();
    double d=rs.getDouble(1);
     rs.close(); return d;}
   catch(Exception e){ System.out.println("In function get");
     e.printStackTrace(); return -1;}}

   public int Ncolumns(String name_of_massive) {
     // кіл-у стовпців у масиві  name_of_massive
  try{
      MasSize m=new MasSize();
        m.setName(name_of_massive);
        m.setColumns(0);
        m.setRows(0);
       if(!tables.contains(m)) {System.out.println("there is no
such massive");
         return -1; }
       MasSize m1=(MasSize) tables.get(tables.indexOf(m));
        return m1.getColumns();}
  catch(Exception e){ System.out.println("In function
Ncolumns");
       e.printStackTrace(); return -1;}}

   public int Nrows(String name_of_massive){
     // кіл-у рядків у масиві  name_of_massiv
   try{
      MasSize m=new MasSize();
        m.setName(name_of_massive);
        m.setColumns(0);
        m.setRows(0);
      if(!tables.contains(m)) { System.out.println("there is no
such massive");
      return -1;    }
      MasSize m1=(MasSize) tables.get(tables.indexOf(m));
      return m1.getRows();}
   catch(Exception e){System.out.println("In function
Ncolumns");
     e.printStackTrace(); return -1;}}

  public void delete_mas(String name_of_massive){
    //видалення масиву name_of_massive з бази даних
   try{
      MasSize m=new MasSize();
        m.setName(name_of_massive);
        m.setColumns(0);
        m.setRows(0);
     if(!tables.contains(m)) { System.out.println("there is no
such massive");
          return; }
```

```
     String str="drop table "+name_of_massive;
     Statement stmt2 = conn.createStatement();
     stmt2.executeUpdate(str);}
 catch(Exception e){ System.out.println("In function
delete_mas");
    e.printStackTrace();}}}
Використання даного класу може мати такий вигляд:
class m1{
public static void main(String[] args){
DB
data=newDB("jdbc:derby:","NeuronNetwork","org.apache.derby.jdbc.
EmbeddedDriver");
         data.createMas("mas",3,3);
         data.set("mas",1,1,3.56);
         double d=data.get("mas",1,1);
         System.out.println(d);
         data.set("mas",1,2,4.78);
         d=data.get("mas",1,2);
         System.out.println(d);
         int r=data.Ncolumns("mas");
         int c=data.Nrows("mas");
         System.out.println("rows= "+r);
         System.out.println("cols= "+c);
         data.set("mas",1,2,5.5);
         d=data.get("mas",1,2);
         System.out.println(d);
}}
```

13.2. МЕТАДАННІ

За допомогою методів інтерфейсів *ResultSetMetaData* і *DatabaseMetaData* можна одержати список таблиць, визначити типи, властивості й кількість стовпців БД.

Одержати об'єкт *ResultSetMetaData* можна в такий спосіб:

```
ResultSetMetaData rsMetaData= rs.getMetaData();
```

де *rs* має тип *ResultSet.*

Розглянемо деякі методи інтерфейсу *ResultSetMetaData*:

int getColumnCount() – повертає число стовпців набору результатів об'єкта *ResultSet;*

String getColumnName(int column) – повертає ім'я зазначеного стовпця об'єкта *ResultSet*;

int getColumnType(int column) – повертає тип даних зазначеного стовпця об'єкта *ResultSet.*

Одержати об'єкт *DatabaseMetaData* можна в такий спосіб:

```
DatabaseMetaData dbMetaData = conn.getMetaData();
```

де conn об'єкт класу Connection.

Розглянемо деякі методи інтерфейсу *DatabaseMetaData:*

String getDatabaseProductName() – повертає назву СУБД;

String getDatabaseProductVersion() – повертає номер версії СУБД;

String getDriverName() – повертає ім'я драйвера JDBC;

String getUserName() – повертає ім'я користувача БД;

String getURL() – повертає місцезнаходження джерела даних;

ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types) – повертає набір типів таблиць, доступних для даної БД;

catalog - ім'я каталогу в БД;

schemaPattern - ім'я схеми (схема-безліч іменованих об'єктів, що використовується для класифікації об'єктів БД);

tableNamePattern – ім'я таблиці (або таблиць);

types – список табличних типів.

До повернутого об'єкта типу *ResultSet* застосовується метод *getMetaData().*

13.3. ПУЛ З'ЄДНАНЬ

З'єднання із БД є дорогою (по необхідних ресурсах) операцією. Тому еефективним способом зменшення кількості з’єднань є організація пула використовуваних з'єднань, які не закриваються фізично, а зберігаються в черзі та надаються повторно для інших запитів.

Доступ, що розділяється до джерела даних можна організувати шляхом оголошення статичної змінної типу *DataSource* з пакета *javax.sql*. Джерело даних типу DataSource - це компонент, що представляє з'єднання з додатком СУБД.

Клас *InitialContext*, як частина JNDI API, забезпечує роботу з каталогом іменованих об'єктів. У цьому каталозі можна зв'язати об'єкт джерела даних *DataSource* з деяким ім'ям (не тільки з ім'ям БД, але й взагалі з кожним), попередньо створивши об'єкт *DataSource.*

Потім створений об'єкт можна одержати за допомогою методу *lookup()* за його ім'ям. Методу *lookup()* передається ім'я, що завжди починається з імені кореневого контексту.

```
javax.naming.Context ct = new javax.naming.InitialContext();
DataSource ds = (DataSource)ct.lookup("java:jdbc/mybd");
Connection cn = ds.getConnection("name", "pass");
```

Після виконання запиту з'єднання завершується і його об'єкт вертається назад у пул викликом:

```
cn.close();
```

Деякі виробники СУБД для полегшення створення пула з'єднань визначають власний клас на основі інтерфейсу *DataSource*. У цьому випадку пул з'єднань може бути створений, наприклад, у такий спосіб:

```
import COM.ibm.db2.jdbc.DB2DataSource;
...
DB2DataSource ds = new DB2DataSource();
ds.setServerName("//localhost:6061/mybd");
Connection cn = ds.getConnection("db2adm",  "pass");
```

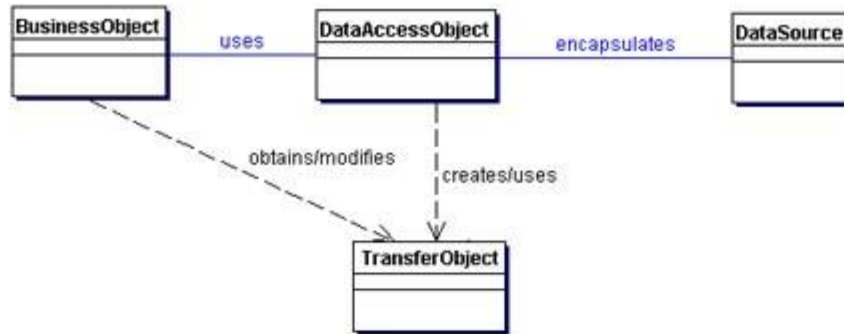
Драйвер визначається автоматично в об'єкті *DB2DataSource*.

13.4. ПАТЕРН DATA ACCESS OBJECT

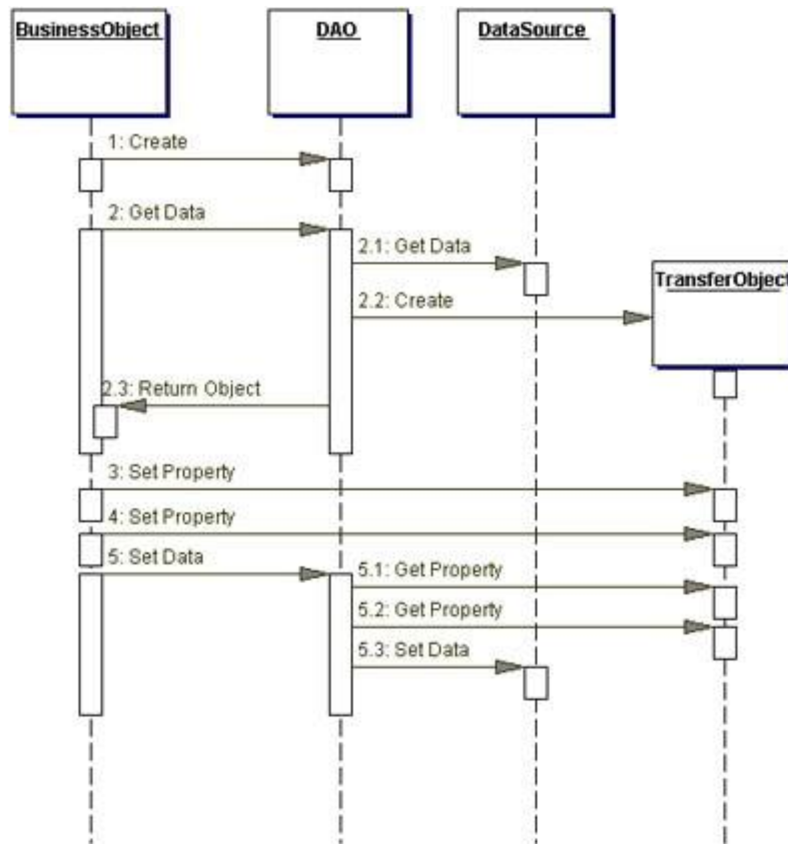
13.4.1. Рагальні відомості про патерн DAO

Data Access Object (DAO) використовується для абстрагування й інкапсулювання доступу до джерела даних. Бізнес-компоненти, що використовують DAO, працюють з простішим інтерфейсом, який надається об'єктом DAO своїм клієнтам. DAO повністю приховує деталі реалізації джерела даних від клієнтів. Оскільки при змінах реалізації джерела даних інтерфейс, що представляється DAO, не змінюється, цей патерн дає можливість DAO приймати різні схеми сховищ не впливаючи на клієнти або бізнес-компоненти.

Діаграма класів для DAO виглядає так:



Діаграма послідовності, що показує взаємодії між учасниками в цьому патерні має наступний вигляд:



BusinessObject

BusinessObject представляє клієнта даних. Це об'єкт, який потребує доступу до джерела даних для отримання та збереження даних. BusinessObject може бути реалізований як сесійний компонент, компонент управління даними чи інший Java-об'єкт, сервлет або допоміжний компонент.

DataAccessObject

DataAccessObject є первинним об'єктом цього патерну. DataAccessObject абстрагує реалізацію доступу до даних для BusinessObject,

яка використовується, забезпечуючи прозорий доступ до джерела даних. BusinessObject передає також відповідальність за виконання операцій завантаження і збереження даних до об'єкта DataAccessObject.

DataSource

DataSource представляє реалізацію джерела даних. Джерелом даних може бути база даних, XML-репозиторій і не тільки, зокрема інша система (традиційна/мейнфрейм), служба (B2B-служба або система обслуговування кредитних карт), або будь-якої репозиторій (LDAP).

TransferObject

TransferObject є об'єктом, який використовується для передачі даних. DataAccessObject може використовувати TransferObject для повернення даних клієнту. DataAccessObject може також приймати дані від клієнта в об'єкті TransferObject для їх оновлення в джерелі даних.

13.4.2. Стратегії для DAO

Стратегія Automatic DAO Code Generation

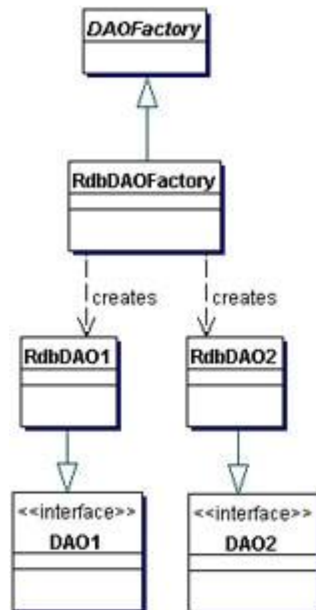
Оскільки BusinessObject відповідає конкретному DAO, є можливість встановити взаємовідношення між BusinessObject, DAO реалізаціями, що застосовуються (наприклад, таблиці в RDBMS). Після встановлення взаємовідношень з'являється можливість написати просту утиліту для генерації коду, що залежить від програми, яка може генерувати код для всіх об'єктів DAO, потрібних додатку. Метадані для генерації DAO можуть визначатися розробником в файлі-дескрипторі. В якості альтернативи генератор коду може автоматично проаналізувати базу даних і надати необхідні для доступу до неї об'єкти DAO.

Стратегія Factory for Data Access Objects

Патерн DAO можна зробити дуже гнучким застосувавши патерни Abstract Factory та Factory Method. Зокрема, згадана стратегія може бути реалізована з використанням патерну Factory Method для генерації декількох

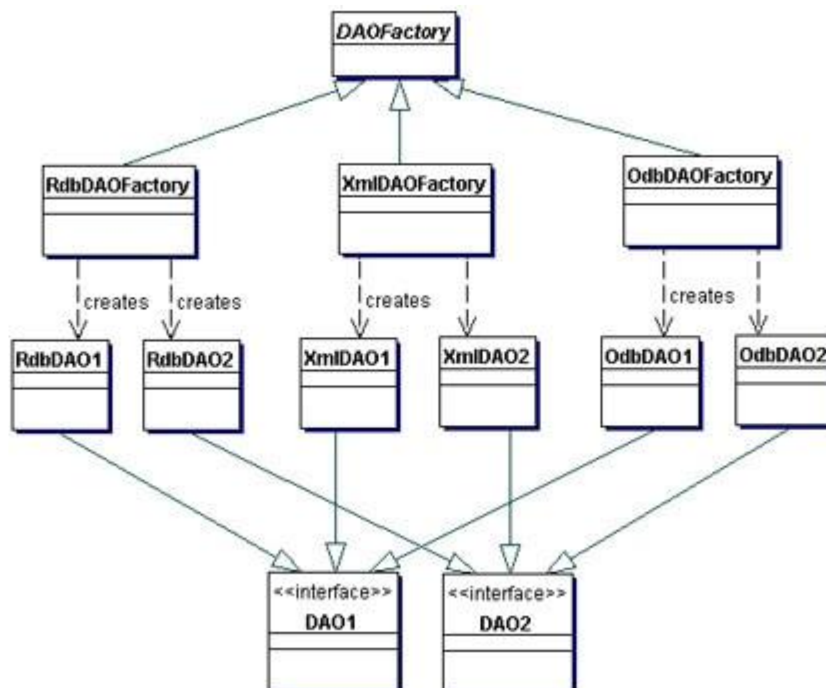
потрібних додатку об'єктів DAO тоді, коли застосоване сховище даних не змінюється при переході від однієї реалізації до іншої.

Тоді діаграма класів виглядатиме так:

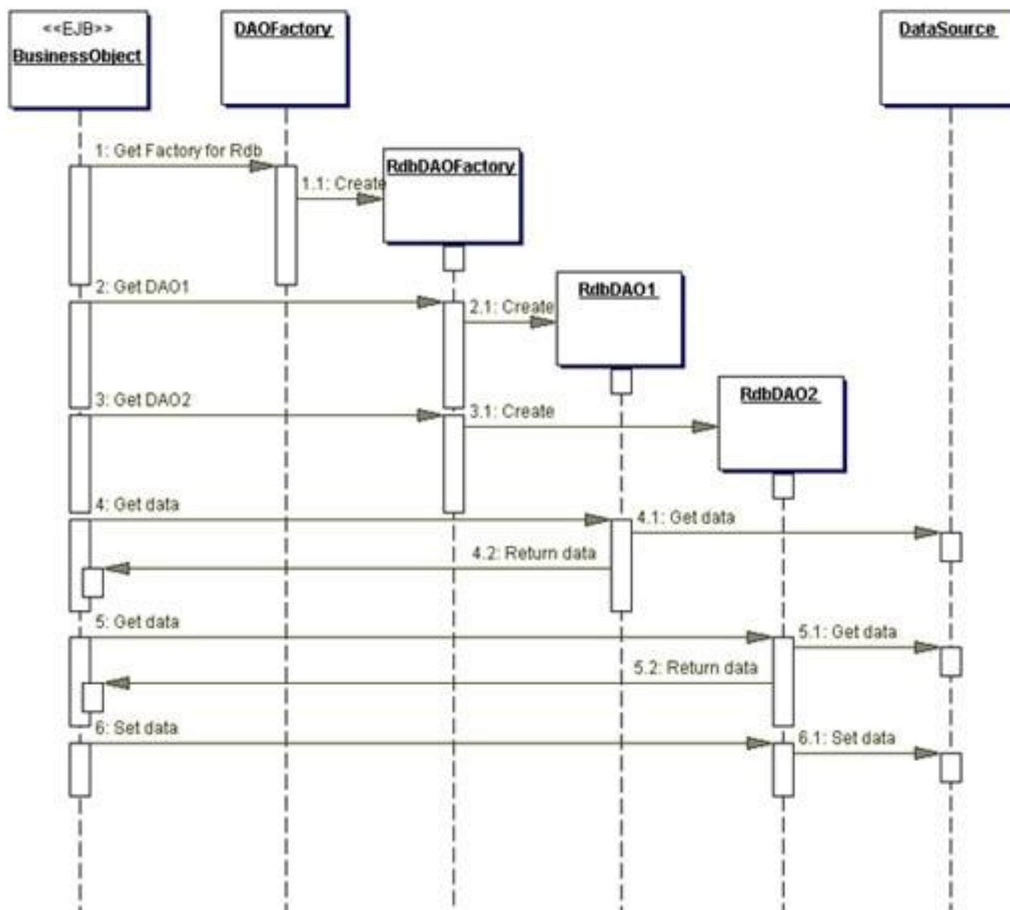


У випадку можливості змін у застосованому сховищі даних при переході від однієї реалізації до іншої, стратегія може бути реалізована із застосуванням патерну Abstract Factory.

Отже, діаграма класів прийме вигляд:



Опис взаємодій для цієї стратегії подано на наступній діаграмі послідовності:



Розглянемо приклад коду для патерна DAO.

```
// DAO Factory
public abstract class DAOFactory {
//список DAO типів, що підтримуються фабрикою
 public static final int CLOUDSCAPE = 1;
 public static final int ORACLE = 2;
 public static final int SYBASE = 3;
...
//методи для створення DAO
 public abstract CustomerDAO getCustomerDAO();
 public abstract AccountDAO getAccountDAO();
 public abstract OrderDAO getOrderDAO();
...
public static DAOFactory getDAOFactory( int whichFactory) {
   switch (whichFactory) {
      case CLOUDSCAPE: return new CloudscapeDAOFactory();
      case ORACLE : return new OracleDAOFactory();
      case SYBASE : return new SybaseDAOFactory();
      ...
      default : return null;
} } }
public class CloudscapeDAOFactory extends DAOFactory {
 public static final String DRIVER=
                          "COM.cloudscape.core.RmiJdbcDriver";

 public static final String DBURL=
```

```
      "jdbc:cloudscape:rmi://localhost:1099/CoreJ2EEDB";
//метод для створення з'єднання з Cloudscape
public static Connection createConnection() {…….}
 public CustomerDAO getCustomerDAO() {
                        return new CloudscapeCustomerDAO(); }

 public AccountDAO getAccountDAO() {
                        return new CloudscapeAccountDAO(); }

 public OrderDAO getOrderDAO() {
                        return new CloudscapeOrderDAO(); }
... }
```

Інтерфейс для CustomerDAO виглядатиме так:

```
public interface CustomerDAO {
   public int insertCustomer(...);
   public boolean deleteCustomer(...);
   public Customer findCustomer(...);
   public boolean updateCustomer(...);
   public RowSet selectCustomersRS(...);
   public Collection selectCustomersTO(...);
   ...
}
```

Імплементація інтерфейсу CustomerDAO

```
public class CloudscapeCustomerDAO
                                  implements CustomerDAO {
  public CloudscapeCustomerDAO() {…. }
  public int insertCustomer(...) {
//метод повертає номер новоствореного Customer'а або -1 при
помилці
 }
  public boolean deleteCustomer(...) { …….. }
  ………………………
}
```

Розглянемо клас Customer Transfer Object:

```
public class Customer implements java.io.Serializable {
  int CustomerNumber;
  String name;
  String streetAddress;
  String city;
  ...
//методи get та set
  ... ...
}
```

Нарешті, розглянемо приклад використання DAO Factory:

```
…………………………………
DAOFactory cloudscapeFactory =
                                DAOFactory.getDAOFactory(
                       DAOFactory.DAOCLOUDSCAPE);
CustomerDAO custDAO =
                cloudscapeFactory.getCustomerDAO();

int newCustNo = custDAO.insertCustomer(...);

//створення об'єкта Customer
Customer cust = custDAO.findCustomer(...);

//зміна значень об'єкта Customer
cust.setAddress(...);
cust.setEmail(...);
custDAO.updateCustomer(cust);
………………………………………………………….
```

РОЗДІЛ 14

14.1. РОБОТА З БАЗАМИ ДАНИХ З ВИКОРИСТАННЯМ HIBERNATE

Hibernate – це механізм відображення в реляційній базі даних об'єктів Java.

На практиці найбільшу популярність набули реляційні моделі баз даних, хоча в сучасних методологіях програмування користується популярністю об'єктно-орієнтоване програмування.

Для їхнього стикування розроблені численні технології, специфікації та фреймворки для маппінгу об'єктів на таблиці реляційних баз даних.

У Java розробників є багато технологій для роботи з даними. Це може бути просто серіалізація об'єктів, JDBC, JDO, JPA та ряд інших.

Java Persistence API (JPA), що є складовою частиною Hibernate, поєднує в собі простоту серіалізації об'єктів з можливістю роботи з даними на рівні об'єктно-орієнтованої моделі. При цьому залишається можливість комбінування доступу до даних як у JDBC на рівні реляційних даних.

Розглянемо приклади використання Hibernate. У базі даних створимо таблицю Student з трьома полями:

1) id - ідентифікатор

2) name - ім'я студента

3) age - його вік

Іншими словами, виконаємо запит:

```
CREATE TABLE Student (id NUMBER (10) NOT NULL,
name varchar (100) NOT NULL,
age NUMBER (3) NOT NULL,
CONSTRAINT pk_Student PRIMARY KEY (id));
```

Тепер створимо пакет logic. У ньому опишемо клас-сутність, який будемо зберігати в БД.

```
@Id
    @GeneratedValue (generator="increment")
    @GenericGenerator (name="increment", strategy = "increment")
    @Column(name="id")
```

```
    public Long getId() {
        return id;
    }

    @Column(name="name")
    public String getName(){
        return name;
    }

    @Column(name="age")
    public Long getAge(){
        return age;
    }

    public void setId(Long i){
        id = i;
    }

    public void setName(String s){
        name = s;
    }

    public void setAge(Long l){
        age = l; } }
```

Анотації тут використовуються для Mapping Java класів з таблицями бази даних. Простіше кажучи для того, щоб Hibernate знав, що даний клас є сутністю, об'єкти цього класу ми будемо зберігати в базі даних. Використані тут анотації мають наступний смисл:

@Entity - вказує на те, що даний клас є сутністю;

@Table - задає ім'я таблиці, в якій будуть зберігатися об'єкти класу;

@Id - позначає поле id;

@GeneratedValue (generator = "increment");

@GenericGenerator (name = "increment", strategy = "increment") - вказує на те, як буде генеруватися id (в даному випадку за зростанням);

@Column - позначає ім'я колонки, що відповідає даному полю.

Варто відзначити також, що всі класи-сутності повинні обов'язково мати геттери, сеттери і конструктор за замовчуванням. Тепер створимо головний конфігураційний файл hibernate.cfg.xml і помістимо його в папку

bin проекту. З цього файлу Hibernate буде брати всю необхідну йому інформацію:

```
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
 <hibernate-configuration>
   <session-factory>
    <property
name="connection.driver_class">oracle.jdbc.driver.OracleDriver</
property>
    <property
name="connection.url">jdbc:oracle:thin:@localhost:1521:MyDB</pro
perty>
    <property name="connection.username">Your_Login</property>
    <property
name="connection.password">Your_Password</property>
    <property name="connection.pool_size">10</property>
    <property
name="dialect">org.hibernate.dialect.OracleDialect</property>
    <property name="show_sql">true</property>
    <property name="hbm2ddl.auto">update</property>
    <property
name="hibernate.connection.autocommit">false</property>
    <property
name="current_session_context_class">thread</property>

    <mapping class="logic.Student" />

  </session-factory>
</hibernate-configuration>
```

Створимо пакет util, а в ньому клас HibernateUtil, який буде відповідати за обробку даного xml файлу і встановлення з'єднання з базою даних:

```
package util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static SessionFactory sessionFactory = null;

    static {
        try {
                //creates the session factory from
hibernate.cfg.xml
                sessionFactory = new
Configuration().configure().buildSessionFactory();
        } catch (Exception e) {
              e.printStackTrace();
```

```
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

Тепер залишилося розібратися зі взаємодією додатка з базою даних. Для класу-сутності, визначимо інтерфейс StudentDAO з пакета DAO, що містить набір необхідних методів:

```
package DAO;

import java.sql.SQLException;
import java.util.List;
import logic.Student;

public interface StudentDAO {
    public void addStudent(Student student) throws SQLException;
//додати студента

    public void updateStudent(Student student) throws
SQLException; //оновити студента
    public Student getStudentById(Long id) throws SQLException;
 //отримати студента за id
    public List getAllStudents() throws SQLException;
 //отримати всіх студентів
    public void deleteStudent(Student student) throws
SQLException; //видалити студента
}
```

Тепер визначимо реалізацію цього інтерфейсу в класі SudentDAOImpl в пакеті DAO.Impl:

```
package DAO.Impl;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import javax.swing.JOptionPane;
import org.hibernate.Session;
import util.HibernateUtil;
import DAO.StudentDAO;
import logic.Student;

public class StudentDAOImpl implements StudentDAO {
        public void addStudent(Student stud) throws SQLException
{
            Session session = null;
            try {
                session =
```

```
HibernateUtil.getSessionFactory().openSession();
                session.beginTransaction();
                session.save(stud);
                session.getTransaction().commit();
            } catch (Exception e) {
                JOptionPane.showMessageDialog(null,
e.getMessage(), "Ошибка I/O",

JOptionPane.OK_OPTION);
            } finally {
                if (session != null && session.isOpen()) {
                    session.close();
                } }
      }
public void updateStudent(Student stud) throws SQLException {
            Session session = null;
            try {
                session =
HibernateUtil.getSessionFactory().openSession();
                session.beginTransaction();
                session.update(stud);
                session.getTransaction().commit();
            } catch (Exception e) {
                JOptionPane.showMessageDialog(null,
e.getMessage(), "Ошибка I/O",

JOptionPane.OK_OPTION);
            } finally {
                if (session != null && session.isOpen()) {
                    session.close();
                }
            }
      }


public Student getStudentById(Long id) throws SQLException {
            Session session = null;
            Student stud = null;
            try {
                session =
HibernateUtil.getSessionFactory().openSession();
                stud = (Student) session.load(Student.class,
id);
            } catch (Exception e) {
                JOptionPane.showMessageDialog(null,
e.getMessage(), "Ошибка I/O",

JOptionPane.OK_OPTION);
            } finally {
                if (session != null && session.isOpen()) {
                    session.close();
                }
            }
```

```
            return stud;
      }

public List<Student> getAllStudents() throws SQLException {
            Session session = null;
            List<Student> studs = new ArrayList<Student>();
            try {
                session =
HibernateUtil.getSessionFactory().openSession();
                studs =
session.createCriteria(Student.class).list();
            } catch (Exception e) {
                JOptionPane.showMessageDialog(null,
e.getMessage(), "Ошибка I/O",

JOptionPane.OK_OPTION);
            } finally {
                if (session != null && session.isOpen()) {
                    session.close();
                }
            }
            return studs;
      }

public void deleteStudent(Student stud) throws SQLException {
            Session session = null;
            try {
                session =
HibernateUtil.getSessionFactory().openSession();
                session.beginTransaction();
                session.delete(stud);
                session.getTransaction().commit();
            } catch (Exception e)
{ JOptionPane.showMessageDialog(null, e.getMessage(), "Ошибка
I/O", JOptionPane.OK_OPTION);
            } finally {
                if (session != null && session.isOpen()) {
                    session.close();
                }
            }
      }
}
```

Створимо клас Factory в пакеті DAO, до якого будемо звертатися за нашими реалізаціями DAO, від яких і будемо викликати необхідні нам методи:

```
package DAO;
import DAO.Impl.StudentDAOImpl;
```

```
public class Factory {
      private static StudentDAO studentDAO = null;
      private static Factory instance = null;

      public static synchronized Factory getInstance(){
            if (instance == null){
              instance = new Factory();
            }
            return instance;
      }

      public StudentDAO getStudentDAO(){
            if (studentDAO == null){
              studentDAO = new StudentDAOImpl();
            }
            return studentDAO;
      }
}
```

Метод *main* буде мати вигляд:

```
import DAO.Factory;
public class Main {
public static void main(String[] args) throws SQLException {
        //Створюемо двох студентів
        Student s1 = new Student();
        Student s2 = new Student();
        //Ініціалізуемо їх
        s1.setName("Ivanov Ivan");
        s1.setAge(21l);
        s2.setName("Petrova Alisa");
        s2.setAge(24l);
       //Зберігаемо в бд, id будуть сгенеровані автоматично
        Factory.getInstance().getStudentDAO().addStudent(s1);
        Factory.getInstance().getStudentDAO().addStudent(s2);
        //виведемо всіх студентів з бд
        List<Student> studs =
Factory.getInstance().getStudentDAO().getAllStudents();
        for(int i = 0; i < studs.size(); ++i) {
                System.out.println("Name student : " +
studs.get(i).getName() + ", Age : " +

studs.get(i).getAge() +",  id : " + studs.get(i).getId());

System.out.println("=============================");

        }
    }
}
```

14.2. ЗАПИТИ В HIBERNATE

Запити повертають набір даних з бази даних, що задовольняють заданій умові. Бібліотека Hibernate пропонує три види запитів до БД:

1) HQL
2) Criteria
3) SQL

14.2.1. HQL

Hibernate дозволяє виконувати запити на мові HQL (The Hibernate Query Language – мова запитів Hibernate), яка багато в чому схожа на мову SQL, з тією різницею, що є повністю об'єктно-орієнтованою.

Якщо запит за допомогою SQL проводився методом createSQLQuery, то в HQL буде просто createQuery.

Простий приклад:

```
List <Student> studs = (List <Student>) session.createQuery (
"from Student order by name "). list ()
```

Select на початку запиту можна не вказувати.

Оскільки HQL – об'єктно-орієнтована мова, то значення полів можна вибрати і так:

```
List <String> names = (List <String>) session.createQuery (
"select  stud.name from Student stud order by name "). list ();
```

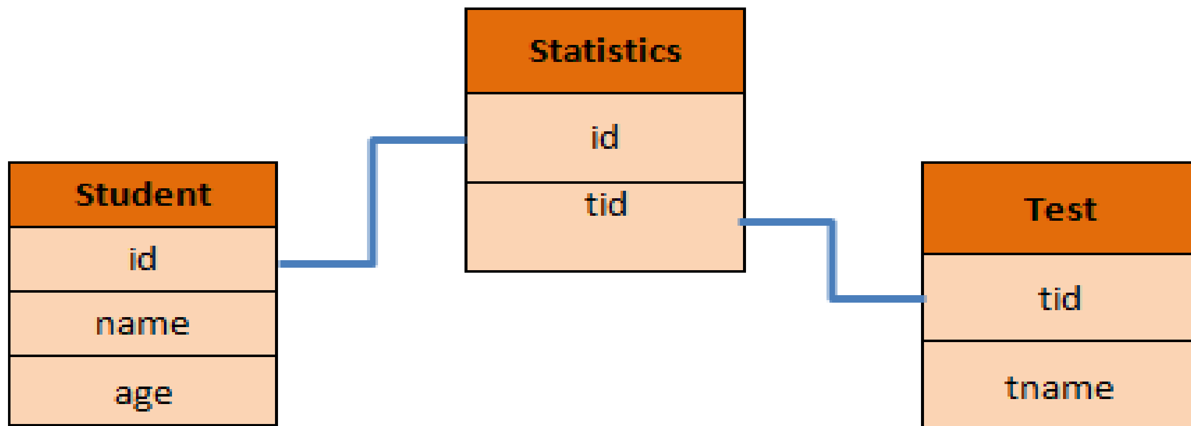
А можна і так:

```
List result = session.createQuery ( "select new list (stud,
name, stud.age)  from Student as stud "). list ();
```

Мова HQL відносно складна, але надає дуже багато можливостей.

14.2.2. Відношення в Hibernate

Крім таблиці Student, створимо ще дві таблиці Test і Statistics. Вони будуть пов'язані наступним чином:



Таблиця Statistics служить для зв'язку таблиць Student і Test, щоб уникнути відносини багато до багатьох.

Створимо ці дві таблиці:

```
CREATE TABLE Test (tid NUMBER (10) NOT NULL,
tname varchar (100) NOT NULL, CONSTRAINT pk_Test PRIMARY KEY
(tid)); CREATE TABLE Statistics (stid NUMBER (10) NOT NULL,
id NUMBER (10) NOT NULL,
tid NUMBER (10) NOT NULL,
CONSTRAINT pk_Statistics PRIMARY KEY (stid),
CONSTRAINT fk_Student FOREIGN KEY (id) REFERENCES Student (id),
CONSTRAINT fk_Test FOREIGN KEY (tid) REFERENCES Test (tid));
```

Також в файл hibernate.cfg.xml додамо маппінг нових класів:

```
<Mapping class = "logic.Test" /> <mapping class =
"logic.Statistics" />
```

Розглянемо код.

Створюємо в пакеті logic класи-сутності:

```
Test
package logic;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinTable;
import javax.persistence.Table;
import javax.persistence.JoinColumn;
```

```
import org.hibernate.annotations.GenericGenerator;

@Entity
@Table(name="Test")
public class Test {

    private Long tid;
    private String tname;

    public Test(){
        tname = null;
    }

public Test(Test s){
        tname = s.getTName();
    }

    @Id
    @GeneratedValue(generator="increment")
    @GenericGenerator(name="increment", strategy = "increment")
    @Column(name="tid")
    public Long getTid() {
        return tid;
    }

    @Column(name="tname")
    public String getTName(){
        return tname;
    }

    public void setId(Long i){
        tid = i;
    }

    public void setTName(String s){
        tname = s;
    }
}
Statistics
package logic;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

import org.hibernate.annotations.GenericGenerator;

@Entity
@Table(name="Statistics")
public class Statistics {
```

```
    private Long stid;
    private Long id;
    private Long tid;

    public Statistics(){
    }
@Id
    @GeneratedValue(generator="increment")
    @GenericGenerator(name="increment", strategy = "increment")
    @Column(name="stid")
    public Long getStid(){
        return stid;
    }

    @Column(name="id")
    public Long getId(){
        return id;
    }

    @Column(name="tid")
    public Long getTid(){
        return tid;
    }
}
```

Класи TestDAO і TestDAOImpl створюються аналогічно як для сутності Student.

Залишилося тільки показати Hibernate, як ці таблиці між собою пов'язані.

У Hibernate для цього передбачені наступні види анотацій: @OneToOne, @OneToMany, @ManyToOne, @ManyToMany.

Наприклад, щоб зв'язати таблиці Student і Statistics зв'язком багато до одного, слід додати в клас Student наступної код:

```
private Statistics stat;
@ManyToOne
@JoinTable (name = "id")
public Statistics getStat ()
{
return stat;
}
```

У класі Statistics аннотуємо зв'язок один до багатьох з класом Test:

```
 private Set <Test> tests = new HashSet <Test> (0); @OneToMany
@JoinTable (name = "id") public Set <Test> getTests () {return
tests;  }
```

Оскільки таблиця Statistics є не просто таблицею, пов'язаною зі Student і Test, а розбиває небажаний зв'язок багато до багатьох, ми також можемо показати це Hibernate. Просто замість того, щоб окремо позначати зв'язок в кожній таблиці, ми позначимо весь зв'язок в одній, наприклад, в таблиці Test додавши код:

```
 private Student stud;
 @ManyToOne
 @JoinTable (name = "Statistics",    joinColumns = @JoinColumn
(name = "tid"),   inverseJoinColumns = @JoinColumn (name =
"id"))
 public Student getStud ()
{
return stud;
}
```

С допомогою параметра name анотації @JoinTable ми позначаємо сполучну таблицю,

joinColumns = @JoinColumn - вказуємо через який ключ пов'язані таблиці Test і Statistics,

inverseJoinColumns = @JoinColumn - вказуємо, через які ключі пов'язані вже Statistics і Student.

Як цей зв'язок позначали в класі Student:

```
private Test test; @ManyToOne @JoinTable (name = "Statistics",
joinColumns = @JoinColumn (name = "id"), inverseJoinColumns =
@JoinColumn (name = "tid")) public Test getTest () {return test;
}
```

14.2.3. *Criteria*

Об'єкт Criteria створюється за допомогою методу createCriteria екземпляра класу Session:

```
Criteria crit = session.createCriteria(Student.class);
//створюємо критерій запиту
crit.setMaxResults(50); //обмежуемо число результатів
List studs = crit.list(); //розміщуємо результати у список
```

В даному прикладі був створений критерій запиту на основі класу Student. Звуження вибірки здійснюється наступним чином:

```
List studs = session.createCriteria(Student.class)
    .add( Expression.like("name", "Ivanov%") )
    .add( Expression.between("age", 18, 25) )
    .list();

List studs = session.createCriteria(Student.class)
    .add( Expression.like("name", "_van%") )
    .add( Expression.or(
        Expression.eq( "age", new Integer(20) ),
        Expression.isNull("age")
    ) ).list();

List studs = session.createCriteria(Student.class)
    .add( Expression.in( "name", new String[] { "Ivanov Ivan",
                                                         "Petrov
Petia", "Zubin Egor" } ) )
        .add( Expression.disjunction()
        .add( Expression.isNull("age") )
        .add( Expression.eq("age", new Integer(20) ) )
        .add( Expression.eq("age", new Integer(21) ) )
        .add( Expression.eq("age", new Integer(22) ) )
    ) ).list();
```

Expression.like - вказує шаблон, де '_' - будь-який один символ, '%' - будь-яку кількість символів Expression.isNull - значення поля одно NULL. Expression.between - 'age' - ім'я поля, 18 - мінімальне значення зазначеного поля, 25 - його максимальне значення Expression.in - вказує діапазон значень конкретного поля Expression.disjunction, Expression.or - диз'юнкція (OR) - об'єднує в собі кілька інших виразів оператором АБО. Expression.eq - визначає рівність поля якомусь значенню. Результати також можна відсортувати:

```
List studs = sess.createCriteria(Student.class)
    .add( Expression.like("name", "Iv%")
    .addOrder( Order.asc("name") ) //за зростанням
    .addOrder( Order.desc("age") ) //за спаданням
    .list();
```

Також є можливість запиту за даними примірника класу:

```
Student s = new Student();
s.setName("Ivanov Ivan");
s.setAge(20l);
List results = session.createCriteria(Student.class)
```

```
    .add( Example.create(s))
    .list();
```

Поля об'єкта, що мають значення null або є ідентифікаторами, будуть ігноруватися. Example також можна налаштовувати:

```
Example example = Example.create(s)
    .excludeZeroes()           // виключає поля з нульовими
значеннями
    .excludeProperty("name")  // виключає поле "name"
    .ignoreCase()// задає незалежне від регістру порівняння
рядків
    .enableLike();    // використовує like для порівняння рядків
List results = session.createCriteria(Student.class)
    .add(example)
    .list();
```

14.2.4. Запити з використанням SQL

Hibernate дозволяє створювати запити на рідному для бази даних діалекті SQL. Виглядати це буде приблизно так:

```
sess.createSQLQuery ( "select * fromStudent"). addEntity
(Student.class) .list (); sess.createSQLQuery ( "select id,
name, age from Student").addEntity (Student.class) .list ()
```

У запитах також можна вказувати параметри:

```
Query query = session.createSQLQuery ( "select * from Student
where name like? "). addEntity (Student.class);
List result = query.setString (0," Ivan% "). list ();
 query = session.createSQLQuery (" select * from Student where
name like : Name "). AddEntity (Student.class);
List result = query.setString (" name "," Ivan% "). List ();
```

У першому випадку за допомогою query.setString вказується порядковий номер параметра (?) і значення типу String, яке замість нього буде підставленим. Якщо значення типу Long, то буде setLong, якщо Date, то setDate і так далі. У другому випадку ім'я параметра задано явно, тому значення задається параметром по імені.

14.2.5. Інша реалізація Hibernate JPA

Розглянемо приклад роботи з базою даних з використанням об'єктно-орієнтованої моделі.

Нехай маємо СУБД MySQL. Припустимо, що створена база даних з ім'ям mybase, а в ній є наступна таблиця, з ім'ям mytable (ID Name Age).

Створимо клас mytable, який співпадає із записом в таблиці:

```
import java.io.Serializable;
import javax.persistence.*;
@Entity
@Table(name= "mytable")
public class mytable implements Serializable {
private static final long serialVersionUID = 1L;
@Id
@Column(name= "id")
@GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
@Column(name= "name", length=64)
   private String name;
@Column(name= "age", length=64)
   private Byte age;

public mytable() {};
public mytable(String name,Byte b){
   this.age=b;
   this.name=name;
}
public Long getId() {return id; }
public void setId(Long id) {this.id = id;}
public void setname(String name){
  this.name=name;
};
public void setAge(Byte age){
  this.age=age;
};
public String toString() {
   return "testproject.mytable[id=" + id + "]";
}
}
```

Hibernate для своєї роботи вимагає спеціальний файл конфігурації (він використовується під час компіляції), який називається persistence.xml. Файл має вигляд:

```
<?xml version="1.0" encoding="UTF-8" ?>
 <persistence version="2.0"
xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
 <persistence-unit name="JavaApplication3PU"
```

```
transaction-type="RESOURCE_LOCAL">
  <provider>
     org.eclipse.persistence.jpa.PersistenceProvider
 </provider>
 <class>mytable</class>

<properties>
  <property name="javax.persistence.jdbc.url"
           value="jdbc:mysql://localhost:3306/mybase" />
  <property name="javax.persistence.jdbc.password"

value="qwertyui" />
  <property name="javax.persistence.jdbc.driver"
                                  value="com.mysql.jdbc.Driver"
/>
  <property name="javax.persistence.jdbc.user"

value="alex" />
  <property name="eclipselink.ddl-generation"

value="create-tables" />
</properties>
</persistence-unit>
</persistence>
```

Тоді робота з базою даних буде такою:

```
import javax.persistence.*;
public class Main {
  public static final EntityManagerFactory emf =

Persistence.createEntityManagerFa9999ctory(

"JavaApplication3PU");
  public static EntityManager em = emf.createEntityManager();
  public static void main(String[] args) {
    mytable te=new mytable("aaaa",new Byte((byte)34));
    te.setname("bbb");
    te.setage(new Byte((byte)56));
    em.getTransaction().begin();
    em.persist(te);
    em.getTransaction().commit();
  }
}
```

РОЗДІЛ 15

15.1. СЕРВЛЕТИ

Сервлети - компоненти програм Java Platform Enterprise Edition, що виконуються на стороні сервера, здатні обробляти клієнтські запити й динамічно генерувати відповіді на них. Найбільше поширення одержали сервлети, що обробляють клієнтські запити по протоколу HTTP.

Зараз сервлети підтримуються більшістю Web-Серверів і є частиною платформи Java EE.

Всі сервлети реалізують загальний інтерфейс *Servlet*. Для обробки HTTP-запитів можна скористатися в якості базового клас абстрактним класом *HttpServlet.* Класи, що відносяться до технології сервлетів знаходяться у пакеті *javax.servlet.* Життєвий цикл сервлета починається з його завантаження у пам’ять контейнером сервлетов при старті або у відповідь на перший запит. Далі відбуваються ініціалізація, обслуговування запитів і завершення існування.

Першим викликається метод *init().* Він дає сервлету можливість ініціалізувати дані й підготуватися для обробки запитів.

Після цього сервлет можна вважати запущеним, він чекає запитів від клієнтів. Запит, що з'являється, обслуговується методом *service()* сервлета, а всі параметри запиту упаковуються в об'єкт *ServletRequest*, що передається як перший параметр методу *service().* Другий параметр методу – об'єкт *ServletResponse*. В цей об'єкт упаковуються вихідні дані в процесі формування відповіді клієнтові. Кожний новий запит приводить до нового виклику методу *service().* Метод *service()* повинен вміти обробляти відразу кілька запитів, тобто бути синхронізований для виконання у багатопоточних середовищах. Якщо ж потрібно уникнути множинних запитів, сервлет повинен реалізувати інтерфейс *SingleThreadModel,* що не містить жодного методу та тільки вказує серверу про однопотокову природу сервлета.

Після завершення виконання сервлета контейнер сервлетів викликає метод *destroy(),* у тілі якого необхідно розмістити код звільнення зайнятих сервлетом ресурсів.

Інтерфейсом *Servlet* передбачена реалізація ще двох методів: *getServletConfig()* і *getServletInfo().* Перший повертає об'єкт типу *ServletConfig,* що містить параметри конфігурації сервлета, а другий – рядок, що описує призначення сервлета.

При розробці сервлетів як базовий клас у більшості випадків використовують не інтерфейс *Servlet,* а клас *HttpServlet,* відповідальний за обробку запитів http (Рис.15.1).

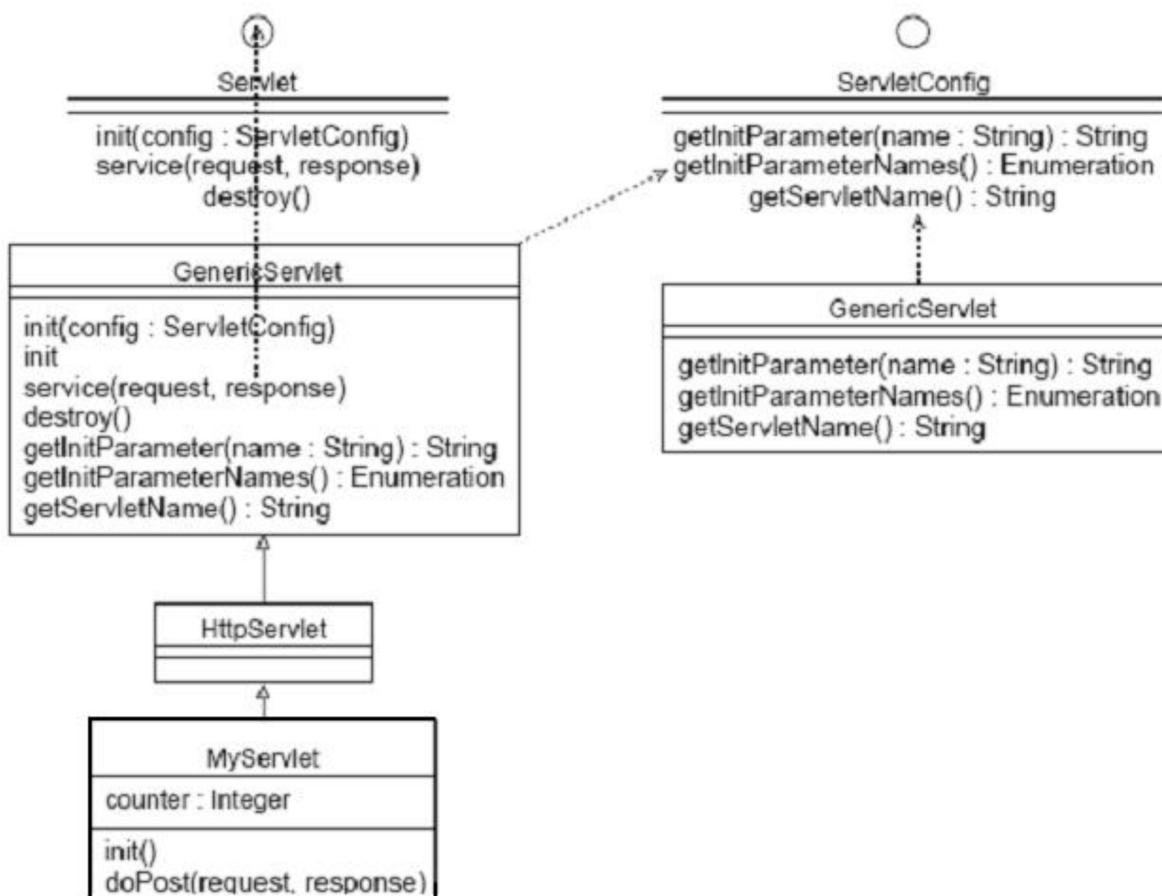


Рис.15.1. Діаграмма класів для servlet

Клас *HttpServlet* має реалізований метод *service(),* що є диспетчером для інших методів, кожний з яких обробляє методи доступу до ресурсів.

У специфікації HTTP визначені наступні методи: *GET, HEAD, POST, PUT, DELETE, OPTIONS і TRACE.*

Найбільше часто вживаються методи *GET* і *POST*, за допомогою яких на сервер передаються запити, а також параметри для їхнього виконання.

При використанні методу *GET* (за замовчуванням) параметри передаються як частина URL, значення можуть вибиратися з полів форми або передаватися безпосередньо через URL. При цьому запити кэшируются й мають обмеження на розмір. При використанні методу *POST (method=POST)* параметри (поля форми) передаються у вмісті HTTP-запиту й упаковані згідно з полем заголовка *Content-Type*. За замовчуванням у форматі:

```
<ім’я>=<значення>&<ім’я>=<значення>&...
```

До завдань методу *service()* класу *HttpServlet* входить аналіз отриманого через запит методу доступу до ресурсів і виклик методу, ім’я якого подібно з назвою методу доступу до ресурсів, але перед ім’ям додається префікс *do*: *doGet()* або *doPost()*. Крім цих методів можуть використовуватися методи: *doHead(), doPut(), doDelete(), doOptions() і doTrace().*

Очевидно, розробник повинен перевизначити потрібний метод, розмістивши в ньому функціональну логіку.

Розглянемо приклад:

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class MyServlet extends HttpServlet {
public MyServlet() {
 super();
...
}
public void init() throws ServletException {
...
}
public void doGet(HttpServletRequest request,
HttpServletResponse response)
```

```
    throws ServletException, IOException
{response.setContentType(«text/html»);
    PrintWriter out = response.getWriter();
    out.print(«This is «);
    out.print(this.getClass().getName());
   out.print(«, using the GET method»);}
public void doPost(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException
   {response.setContentType(«text/html»);
    PrintWriter out = response.getWriter();
    out.print(«This is «);
    out.print(this.getClass().getName());
   out.print(«, using the POST method»);}
public void destroy() {
   super.destroy();
...
}}
```

15.2. ІНТЕРФЕЙСИ SERVLETREQUEST І HTTPSERVLETREQUEST

Потік даних надходить від клієнта у вигляді закодованого й упакованого запиту. Викликаючи методи інтерфейсу *ServletRequest,* можна одержувати певний набір даних, посланих клієнтом. Так, метод *getCharacterEncoding()* визначає символьне кодування запиту, а методи *getContentType()* і *getProtocol()* – MIME-тип запиту, що прийшов, а також назву та версію протоколу відповідно.

Інформацію про ім’я сервера, що прийняв запит, і порт, на якому запит був “почутий” сервером, видають методи *getServerName()* і *getServerPort().*

Можна також довідатися дані й про клієнта, від імені якого прийшов запит. Його IP-адреса вертається методом *getRemoteAddr(),* а його ім’я – методом *getRemoteHost().*

Якщо необхідно прямий доступ до вмісту отриманого запиту,то можна викликати метод *getInputStream()* або *getReader().* Перший повертає посилання на об’єкт класу *ServletInputStream,* а другий – на *BufferedReader.* Після цього можна читати будь-який байт із отриманого запиту, використовуючи техніку роботи з потоками Java.

Якщо, звертаючись до сервера, клієнт крім універсальної адреси задав параметри, сервлету то їх можна довідатися за допомогою наступних методів:

getParameter() - повертає значення параметра за його іменем або *null*, якщо параметра з таким ім’ям немає;

getParameterValues() - повертає масив рядків, а саме всі значення параметра за його іменем, причому параметр може мати кілька значень;

getParameterNames() - повертає об’єкт типу *Enumeration*, що дозволяє довідатися імена всіх присланих параметрів.

Інтерфейс *HttpServletRequest* є похідним від інтерфейсу *ServletRequest* і використовується для одержання інформації в HTTP-сервлетах.

В інтерфейсі *HttpServletRequest* є додаткові методи:

getCookies() - повертає масив *cookies*;

getQueryString() - повертає рядок запиту *http*;

getRemoteUser() - використовується для одержання ім’я користувача, що виконав запит.

Розглянемо приклад:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class RequestClass extends HttpServlet {
public void doGet(HttpServletRequest req,HttpServletResponse
resp)
throws IOException, ServletException
{ resp.setContentType(«text/html»);
  PrintWriter out = resp.getWriter();
  out.println(«<HTML><HEAD>»);
  out.println(«<TITLE>Request Information </TITLE>»);
  out.println(«</HEAD><BODY>»);
  out.println(«<H3>Request Information </H3>»);
  out.println(«<BR>Method: « + req.getMethod());
  out.println(«<BR>Request URI: « +
  req.getRequestURI());
  out.println(«<BR>Protocol: « +
  req.getProtocol());
  out.println(«<BR>PathInfo: « +
  req.getPathInfo());
  out.println(«<BR>Remote Address: « +
  req.getRemoteAddr());
```

```
  out.println(«</BODY></HTML>»);
  out.close();
}}
```

На екрані одержимо:

```
Request Information
Method: GET
Request URI: /FirstProject/RequestClass
Protocol: HTTP/1.1
PathInfo: null
Remote Address: 127.0.0.1
```

15.3. ІНТЕРФЕЙСИ SERVLETRESPONSE І HTTPSERVLETRESPONSE

Дані, що генеруються сервлетами пересилаються серверу-контейнеру за допомогою об'єктів, що реалізують інтерфейс *ServletResponse*, а сервер, у свою чергу, пересилає відповідь клієнтові, що ініціював запит.

В інтерфейсі *ServletResponse* оголошені наступні методи:

setContentType() - задає MIME-тип документів, що генеруються;

getOutputStream() - повертає посилання на потік *ServletOutputStream;*

getWriter() - поверне посилання на потік типу *PrintWriter*.

В інтерфейсі *HttpServletResponse,* що успадковує інтерфейс *ServletResponse,* оголошені наступні методи:

addCookie() - пересилає cookie на клієнтську станцію;

sendError(int sc, String msg) - повідомляє про помилки, що виникли як параметр; передається код помилки і при необхідності текстове повідомлення;

setDateHeader() - додає в заголовок повідомлення параметри;

setAttribute(String name, Object ob) - метод встановлює значення атрибутів компонентів, що є внутрішніми параметрами для передачі інформації між компонентами додатка, наприклад від сервлета до сторінки JSP або іншому сервлету;

getAttributeNames(), getAttribute(String name) - витягають передану компонентом інформацію.

Розглянемо приклад:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class RequestHeader extends HttpServlet {
public void doPost(HttpServletRequest request,
HttpServletResponse response)
 throws IOException, ServletException
    {response.setContentType(«text/html»);
    PrintWriter out = response.getWriter();
    Enumeration e = request.getHeaderNames();
    while (e.hasMoreElements()) {
    String name = (String)e.nextElement();
   String value = request.getHeader(name);
   out.println(name + « = « + value);
}}}
```

Html-файл буде мати вигляд:

```
<HTML>
<HEAD><TITLE>index.html</TITLE></HEAD>
<BODY>
<H2>ЗАПУСК СЕРВЛЕТА</H2>
<FORM action=»RequestHeader»  method=»POST»>
<INPUT type=»submit» value=»Execute»>
</FORM>
</BODY>
</HTML>
```

У браузері одержимо:

```
accept = image/gif, image/ x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/msword, application/ x-shockwave-flash, */*
referer = http://localhost:8080/FirstProject/index.html
accept-language = en-us
content-type = application/ www-form-urlencoded
accept-encoding = gzip, deflate
user-agent = Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
host = localhost:8080
content-length = 0
connection = Keep-Alive
cache-control = no-cache
```

15.4. ІНТЕРФЕЙС SERVLETCONFIG

Інтерфейс *ServletConfig* має наступні методи:

getServletName() - дозволяє одержати ім’я сервлета;

getInitParameterNames() - метод повертає імена параметрів ініціалізації у вигляді *Enumeration;*

getInitParameter(String n) - дозволяє одержати значення конкретного параметра;

getServletContext() - повертає посилання на ***ServletContext,*** використовуючи всю інформацію про середовище, у якій запущений і виконується сервлет.

15.5. РОЗГОРТАННЯ СЕРВЛЕТА

```
Import java.io.*;
import java.util.Locale;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class FirstServlet extends HttpServlet {
private volatile int count;
public void init()
throws ServletException {
super.init();
count = 0;
}
public void doGet(HttpServletRequest req, HttpServletResponse
res)
throws ServletException, IOException {performTask(req, res);}
public void doPost( HttpServletRequest req, HttpServletResponse
res)
throws ServletException, IOException
{performTask(req, res);}
public void performTask(HttpServletRequest req,
HttpServletResponse res)
throws ServletException, IOException
{String title = «First Servlet»;
 PrintWriter out = res.getWriter();
 res.setContentType(«text/html; charset=Cp1251»);
 out.println(«<HTML><HEAD><TITLE>»+ title + «</TITLE>» +
 «</HEAD><BODY><H2>This page is generated « + «by
 FirstServlet<H2><H3>This is its «+ ++count+ «
 execution</H3></BODY></HTML>»);
 out.close();
}}
```

Розглянемо розгортання сервлета на сервері Tomcat. У каталозі, де встановлений Web-сервер будуть наступні підкаталоги:

/bin – містить *startup, shutdown* та інші виконувані файли;

/conf – містить конфігураційні файли, зокрема конфігураційний файл контейнера сервлетів *server.xml;*

/server – містить класи;

/logs – містить log-файли;

/webapps – в цьому каталозі знаходяться

папки, що містять сервлети та інші компоненти додатка.

У каталог */webapps* необхідно помістити папку */FirstProject.* Папка */FirstProject* повинна містити каталог */WEB-INF,* у якому містяться підкаталоги:

/classes – містить клас сервлета *FirstServlet.class;*

/lib – містить бібліотеки класів (якщо вони є), упаковані в JAR-файли (архіви java);

/src – містить вихідний файл сервлета ***FirstServlet.java***, а також ***web.xml*** – конфігураційний файл додатка.

У файлі ***web.xml*** необхідно прописати ім'я й шлях до сервлету. Наприклад:

```
< web-app>
<servlet>
< servlet-name>FirstServletname</ servlet-name>
< display-name>FirstServletdisplay</ display-name>
< servlet-class>test.com.FirstServlet </ servlet-class>
</servlet>
< servlet-mapping>
< servlet-name>FirstServletname</ servlet-name>
< url-pattern>/FirstServlettest</ url-pattern>
</ servlet-mapping>
</ web-app>
```

Викликати сервлет можна набравши в браузері:

```
http://localhost:8080/FirstProject/FirstServlettest
```

15.6. ВИДОБУВАННЯ ІНФОРМАЦІЇ ІЗ ЗАПИТУ

Нехай є наступна форма:

```
<HTML>
<BODY>
<FORM action="testform" >
<H2> Name :
<INPUT type="text" name="p0" value="0"> <BR>
Credit: <INPUT type="text" name="p1" value="0"> <BR>
Price: <INPUT type="text" name="p2" value="0"»> <BR>
Adress:<TEXTAREA name="Adress" rows=3 cols=20>
```

```
</TEXTAREA> <BR>
<INPUT type="submit" value="Submit"> <BR>
</FORM>
</BODY>
</HTML>
```

Для обробки даних, отриманих з полів форми, використовується сервлет:

```
package test.com;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class FormRequest extends HttpServlet {
public void doGet(HttpServletRequest req,
 HttpServletResponse resp)
 throws ServletException, IOException
 { performTask(req, resp);}
 public void performTask(HttpServletRequest req,
HttpServletResponse resp) {
try {
  ing val[] = Str new String[3];
int rest;
for (int i = 0; i < val.length; i++)
 val[i] = req.getParameter("p" + i);
  int c = Integer.valueOf(val[1]).intValue();
  int p = Integer.valueOf(val[2]).intValue();
  rest = c - p;
  PrintWriter out = resp.getWriter();
  resp.setContentType("text/html; charset=Cp1251");
  out.println("<HTML><HEAD>");
  out.println("<TITLE>FormRequest</TITLE>");
   out.println("</HEAD><BODY><BR>");
  out.println("<TABLE BORDER=3><TR><TD>");
   out.println("Name</TD><TD>Credit</TD><TD>Price</TD><TD> Rest
");
   out.println("</TD></TR><TR>");
   for (int i = 0; i < val.length; i++)
   out.println("<TD>" + val[i] + "</TD>");
   out.println("<TD>" + rest + "</TD></TR>");
   out.println("Adress:" +req.getParameter("Adress"));
   out.println("</TABLE></BODY></HTML>");
   out.close();} catch (Throwable e) {e.printStackTrace();}}}
```

РОЗДІЛ 16

16.1. JSP

Технологія Java Server Pages (JSP) була розроблена компанією Sun Microsystems, для створення сторінок з динамічним змістом.

Сторінка JSP забезпечує поділ динамічної та статичної частин сторінки, результатом чого є можливість зміни дизайну сторінки, не торкаючись динамічного змісту.

Вміст Java Server Pages (теги HTML, теги JSP і скрипти) переводиться в сервлет код-сервером. Цей процес відповідає за трансляцію як динамічних, так і статичних елементів, оголошених усередині файлу JSP.

Процеси, виконувані з файлом JSP при першому виклику або при його зміні:

1. Браузер робить запит до сторінки JSP;

2. **JSP-engine** аналізує зміст файлу JSP;

3. **JSP-engine** створює тимчасовий сервлет з кодом, заснованим на вихідному тексті файлу JSP, при цьому контейнер транслюють оператори Java у метод *_jspService().* Якщо немає помилок компіляції, то цей метод викликається для безпосередньої обробки запиту. Отриманий сервлет відповідає за виконання статичних елементів JSP, визначених під час розробки додатково до створення динамічних елементів.

4. Отриманий текст компілюється у файл **.class*.

5. Викликаються методи *init()* і *service()* (*doGet()* або *doPost()*), і сервлет логічно виконується.

6. Сервлет встановлений. Комбінація статичного HTML і графіки разом з динамічними елементами, визначений в оригіналі JSP, пересилаються браузеру через вихідний потік об’єкта відповіді *ServletResponse*.

Наступні виклики файлу JSP просто викличуть сервісний метод сервлета. Сервлет використовується доти, поки сервер не буде зупинений і сервлет не буде вивантажений вручну або поки не буде змінений файл JSP.

Робочий цикл JSP має вигляд (Рис. 16.1):

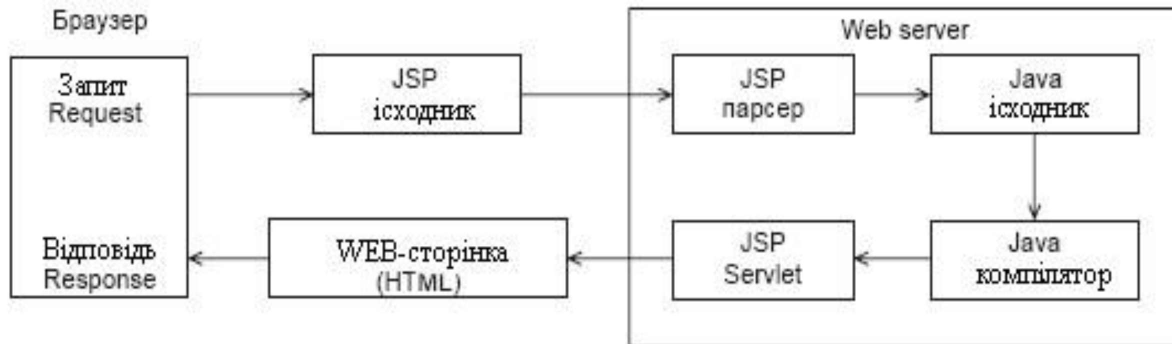


Рис. 16.1. Робочий цикл JSP

JSP-код Java розміщується у спеціальних тегах, які вказують контейнеру, щоб він використовував цей код для генерації сервлета або його частини. У такий спосіб підтримується документ, що одночасно містить і сторінку, і код Java, що управляє цією сторінкою.

Статичні частини HTML-сторінок відсилаються у вигляді рядків у метод *write()*. Динамічні частини включаються прямо в код сервлета.

JSP складається зі стандартних HTML-тегів, JSP-тегів і користувацьких JSP-тегів. Основні теги JSP:

```
< %@ директива %>
< %! оголошення %>
< % скриптлет %>
< %= одиничний Java-вираз %>
< % --JSP-коментар ---%>
<!-- HTML-коментар -->
```

16.2. ДИРЕКТИВИ

Директиви використовуються для установки параметрів серверної сторінки JSP і мають загальний вигляд:

```
< %@ директива ім'я=значення %>
```

Розглянемо приклад:

```
< %@ page language="java" contentType="text/html;
charset=Cp1251" pageEncoding="Cp1251"  errorPage="errorjsp.jsp"
info="директива info"
```

```
 import="java.util.*"  %>
```

Параметр *language* директиви *page* визначає мову що використовується, поки він тільки один. У параметр *info* можна розміщати інформацію про дану сторінку, яку можна одержати, використовуючи метод *getServletInfo()*. Параметр *import* описує пакети й типи, доступні середовищу виконання сценаріїв.

Параметр *contentType* специфікує декодування символів і MIME-тип JSP-відповіді. Директива *taglib* підключає бібліотеки користувацьких тегів.

Директива *include* дозволяє включати в код даної сторінки JSP інші документи допустимих типів.

Приклад:

```
<! ---іприклад # 1 : включення в код умісту іншої сторінки JSP :
jsp01.jsp ---і>
<HTML>
<HEAD><TITLE>jsp01.jsp</TITLE></HEAD>
<BODY>
<H1>First JSP</H1>
< %@ include file="jsp02.jsp"%>
</BODY></HTML>
<! ---і jsp02.jsp ---і>
<HTML><HEAD>
<TITLE>jsp02.jsp</TITLE>
</HEAD>
<BODY>
<P> jsp02.jsp was included in jsp01.jsp</P>
</BODY>
</HTML>
```

Запуск JSP:

```
http://127.0.0.1:8080/FirstProject/jsp01.jsp
```

Параметр *errorPage* вказує на сторінку, перехід до якої буде здійснений у випадку виникнення помилки в поточній сторінці.

```
<! ---і jsp03.jsp ---і>
<HTML><HEAD>
< %@ page language="java"  contentType="text/html;
charset=Cp1251"  pageEncoding="Cp1251" errorPage="errorjsp.jsp"
%>
</HEAD>
<BODY><FORM>
<INPUT type="checkbox"  name="check" value="yesError">
<INPUT type="submit" name="error" value="ERROR"></FORM>
```

```
 < % if("yesError".equals(request.getParameter("check"))) throw
new Exception("yesError"); %>
</BODY>
</HTML>
Сторінка, викликувана при помилках, може мати статичний вигляд
<! ---ierrorjsp.jsp ---i>

<HTML>
<HEAD>
<TITLE>errorjsp.jsp</TITLE>
</HEAD>
<BODY><P>Exception Generated!</P>
</BODY>
</HTML>
```

Роль найпростішої JSP може зіграти звичайна HTML-сторінка, перейменована з расширением *.jsp*.

16.3. ОГОЛОШЕННЯ

Блок оголошень містить змінні Java і методи, які викликаються в expression-блоці. Оголошення не повинне робити запис у вихідний потік *out* сторінки, але може бути використане в скриптлетах і виразах. Наприклад:

```
< %! private int getCount = 0;
  private String getString(){return "РЯДОК";}
  String text = new String("Слово"); %>
```

16.4. СКРИПТЛЕТИ

JSP підтримує вживлення Java-коду в скриптлет-блок. Скриптлети звичайно використовують маленькі блоки коду й виконуються під час обробки запиту клієнта. Коли всі скриптлети збираються воєдино в тім порядку, у якому вони записані на сторінці, вони повинні являти собою правильний код мови програмування. Контейнер поміщає код Java у метод *_jspServlet()* на етапі трансляції.

Розглянемо приклад:

```
<! ---isimple.jsp ---i>
<HTML>
<HEAD>
< %@ page language="java"  contentType="text/html;
charset=Cp1251" pageEncoding=" ISO-8859-5" %>
```

```
</HEAD>
<BODY>
<FORM>
11 * 11 - 9 =
<INPUT type="text" name="text" size="2">
<br><br>
<INPUT type="submit" name="submit" value="Відповідь">
  </FORM>
< % if(request.getParameter("text") != null) {
  if ("112".equals(request.getParameter("text"))){ %>
    <br> Вірно!
    < % } else { %>
    <br> ПОМИЛКА!
    < % } }%>
</BODY>
</HTML>
```

16.5. ВИРАЗИ

В якості виразів використовуються оператори мови Java, які обчислюються, після чого результат обчислення перетворюється в рядок *String* і посилає в потік *out,* як у випадку:

```
< %= text + new String("1") %>
< %= Runtime.getRuntime().totalMemory() %>
```

Перший вираз до рядка *text* приєднує знову створений рядок і відправляє результат у потік *out.* Другий вираз визначає кількість вільної пам'яті.

Розглянемо приклад:

```
<! ---isimplecount.jsp ---i>
<HTML>
<HEAD>
<%@ page language="java" contentType="text/html; charset=Cp1251"
pageEncoding="Cp1251" import="java.util.*"%>
<TITLE>simplecount.jsp</TITLE>
</HEAD>
< %! long localTime = System.currentTimeMillis(); Date localDate
= new Date(); int hitCount = 0; %>
<BODY>
 <H2> Дата завантаження <%= localDate %> </H2>
 <H2> Сьогодні <%= new Date()%> </H2>
 <H3> Сторінка працює
 < %=(System.currentTimeMillis() - localTime)/1000 %>
 секунд </H3>
```

```
 <H3> Сторінку відвідали <%= ++hitCount %> раз, починаючи с <%=
 localDate %> </H3>
 < % System.out.println("Сторінку відвідали (на консоль): " +
 hitCount); System.out.println("до побачення"); %>
</BODY>
</HTML>
```

16.6. НЕЯВНІ ОБ'ЄКТИ

JSP-сторінка завжди має доступ до многих функціональних можливостей сервлета, що створюється Web-контейнером за замовчуванням.

Неявний об'єкт:

request – представляє запит клієнта. Звичайно об'єкт є екземпляром класу, що реалізує інтерфейс *javax.servlet.http.HttpServletRequest.* Для протоколу, відмінного від HTTP, це буде об'єкт реалізації інтерфейсу *javax.servlet.ServletRequest.* Область видимості в межах сторінки;

response – представляє відповідь клієнтові. Звичайно об'єкт є екземпляром класу, що реалізує інтерфейс *javax.servlet.http.HttpServletResponse.* Для протоколу, відмінного від HTTP, це буде об'єкт реалізації інтерфейсу *javax.servlet.ServletResponse.* Область видимості в межах сторінки;

pageContext – визначає контекст JSP-сторінки й надає доступ до неявних об'єктів. Об'єкт класу *javax.servlet.jsp.PageContext.* Область видимості в межах сторінки;

session – створюється контейнером для протоколу HTTP і є екземпляром класу *javax.servlet.http.HttpSession,* надає інформацію про сесію клієнта, якщо така була створена. Область видимості в межах сесії;

application – контейнер, у якому виконується JSP-сторінка, є екземпляром класу *javax.servlet.ServletContext.* Область видимості в межах програми;

out – містить вихідний потік сервлета. Інформація, що посилається в цей потік, передається клієнтові. Об'єкт є екземпляром класу *javax.servlet.jsp.JspWriter*. Область видимості в межах сторінки;

config – містить параметри конфігурації сервлета і є екземпляром класу *javax.servlet.ServletConfig*. Область видимості в межах сторінки;

page – посилання *this* для поточного екземпляра даної сторінки є об'єктом *java.lang.Object.* Область видимості в межах сторінки;

exception – становить собою виняток одного з підкласів класу *java.lang.Throwable,* що передає сторінці повідомлення про помилки та доступно тільки на ній.

16.7. СТАНДАРТНІ ЕЛЕМЕНТИ ACTION

Найбільш використовуваними є стандартні теги версії JSP 2.0:

jsp:useBean – дозволяє використовувати екземпляр компонента JavaBean. Якщо екземпляр із зазначеним ідентифікатором не існує, то він буде створений з областю видимості *page* (сторінка), *request* (запит), *session* (сесія) або *application* (додаток). Оголошується, як правило, з атрибутами *id* (ім'я об'єкта), *scope* (область видимості), *class* (повне ім'я класу), *type* (за замовчуванням *class*).

```
<jsp:useBean id=”ob” scope=”session” class=”test.MyBean” />
```

Створено об'єкт *ob* класу *MyBean,* і надалі через це ім'я можна викликати доступні методи класу.

```
package test;
public class MyBean {
private String info = "немає інформації";
public String getInfo() { return info;}
public void setInfo(String s) {info = s;}}
```

jsp:setProperty – дозволяє встановлювати значення полів зазначеного в атрибуті *name* об'єкта:

```
<jsp:setProperty name=”ob” property=”info” value=”привіт” />
```

jsp:getProperty – одержує значення полів зазначеного об'єкта, перетворить його в рядок і відправляє в неявний об'єкт *out*:

```
<jsp:getProperty name="ob" property="info" />
```

jsp:include – дозволяє включати файли в генерируемую сторінку при запиті сторінки:

```
<jsp:include page="відносний URL" flush="true"/>
```

jsp:forward – дозволяє передати запит іншій сторінці:

```
<jsp:forward page="відносний URL"/>
```

jsp:plugin – заміщається тэгом *<OBJECT>* або *<EMBED>*, залежно від типу броузера, в якому буде виконуватися апплет, що підключається або JavaBean;

jsp:params – групує параметри усередині тегу *jsp:plugin*;

jsp:param – додає параметри в об'єкт запиту, наприклад в елементах *forward, include, plugin*;

jsp:fallback – вказує вміст, що буде використовуватися браузером клієнта, якщо підключається модуль, що, не зможе запуститися. Використовується усередині елемента *plugin*;

jsp:text – містить текстову інформацію.

Розглянемо приклад.

```
<! ---i зчитування інформації й генерація відповіді : index.jsp
---i>
<HTML>
<HEAD>
< %@ page language="java" contentType="text/html;
charset=Cp1251" pageEncoding="Cp1251" %>
<META http-equiv=" Content-Type" content="text/html;
charset=Cp1251">
<TITLE>index.jsp</TITLE>
< %@ page import="java.util.*" %>
<HEAD>
<BODY>
<H1>Форма для заповнення</H1>
<H3>
<% request.setCharacterEncoding("Cp1251"); Enumeration flds =
request.getParameterNames();
if(!flds.hasMoreElements()) { %>
<FORM method="POST" action="index.jsp">
< % for(int i = 0; i < 10; i++) { %>
Field<%=i%>: <INPUT type="text" size="20" name="Поле<%=i%>"
value="Значення<%=i%>"><BR>
< % } %>
```

```
<INPUT TYPE=submit name=submit value="Переслати дані">
</FORM>
< %} else { while(flds.hasMoreElements()) {
String field = (String)flds.nextElement();
String value = request.getParameter(field); %>
<LI><%= field %> = <%= value %></LI>
< % } } %>
</H3>
</BODY>
</HTML>
```

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. К. Арнольд, Дж. Гослинг, Д. Холмс. Язык программирования Java, 3-е изд. – К.: Вильямс, 2001. – 624 с.
2. Б. Эккель. Философия Java, 4-е изд. – СПб: Питер, 2016. – 1168 с.
3. Д. Блох. Java. Эффективное программирование, 2-е изд. – М.: Лори, 2014. – 440 с.
4. С. Макконнелл. Совершенный код, 2-е изд. – Русская редакция, 2017. – 896 с.
5. К.С. Хорстманн, Г. Корнелл. Библиотека профессионала. Java 2. Том 1. Основы, 9-изд. – М.: Вильямс, 2015. – 864 с.
6. К.С. Хорстманн, Г. Корнелл. Библиотека профессионала. Java 2. Том 2. Тонкости программирования, 9-е изд. – М.: Вильямс, 2013. – 1008 с.
7. И.Н. Блинов, В.С. Романчик. Java. Методы программирования. – Минск: “Четыре четверти”, 2013. – 896 с.
8. П. Ноутон, Г. Шилдт. Java 2. Наиболее полное руководство. – СПб.: БХВ-Петербург, 2008. – 1072 с.
9. Б.У. Перри. Java сервлеты и JSP. Сборник рецептов. – М: Кудиц-пресс, 2009. – 768 с.
10. М. Холл. Сервлеты и JavaServer Pages. – СПб.: Питер, 2002. – 496 с.