Олецький О.В.

ПЕРШІ КРОКИ В JAVA

Навчальний посібник для студентів, які навчаються за спеціальностями “Інформатика”, “Програмна інженерія”, “Комп’ютерні науки”, “Прикладна математика”

Київ – 2017

Зміст посібника

ЧАСТИНА 2. НАПИСАННЯ НАЙПРОСТІШИХ КОНСОЛЬНИХ ЗАСТОСУВАНЬ

Розділ 2.1. Основні засоби для створення Java-застосувань

2.1.1 Загальний огляд інструментів для написання Java-програм
2.1.2. Основні утиліти JDK
2.1.3.Компіляція та виконання програм засобами JDK
2.1.4. Налаштування змінних оточення
2.1.5. Програма з кількома класами
2.1.6. Опція -classpath
2.1.7. Створення Java-застосувань засобами Eclipse
2.1.8. Створення Java-застосувань засобами NetBeans
2.1.9. Знайомство з дизасемблером.

Розділ 2.2. Анатомія простого застосування

2.2.1. Основний клас
2.2.2. Клас і файл: назви можуть не співпадати
2.2.3. Метод main та його варіанти
2.2.4. Варіанти Hello, world
2.2.5. Аргументи командного рядка
2.2.6. Програма без методу main
2.2.7. Поняття про пакети

Розділ 2.3. Операції введення-виведення

2.3.1. Загальне поняття про потоки введення-виведення
2.3.2. Стандартні потоки введення-виведення
2.3.3. Введення символів
2.3.4. Класична схема введення рядків
2.3.5. Використання конкатенації для виведення
2.3.6. Введення даних інших типів
2.3.7. Більш нові можливості для введення та виведення
2.3.8. Форматоване виведення
2.3.9. Використання класу Scanner
2.3.10. Деякі особливості та сюрпризи класу Scanner
2.3.11. Використання діалогових вікон для введення
2.3.12. Клас Console

Контрольні запитання

ЧАСТИНА 3. ОГЛЯД ОСНОВНИХ МОЖЛИВОСТЕЙ МОВИ

Вступ

Розділ 3.1. Основні процедурні риси

3.1.1. Коментарі
3.1.2.Змінні та вирази
3.1.3. Найважливіші типи даних
3.1.4. Константи
3.1.5. Надання значення

3.1.6. Основні операції Java
3.1.7. Оператор чи інструкція?
3.1.8. Складені присвоювання
3.1.9. Інкременти та декременти
3.1.10. Умовні оператори
3.1.11. Тернарна операція
3.1.12. Цикли
3.1.13. Цикли: спочатку думаємо, потім кодимо
3.1.14. Оператор варіанту
3.1.15. Функції
3.1.16. Рядки

Розділ 3.2. Класи та об'єкти
3.2.1. Найпростіший приклад
3.2.2. Селектори та модифікатори
3.2.3. Подальший розвиток
3.2.4. Спадкування
3.2.5.Приклад: спадкування методу main()
3.2.5. Клас Object
3.2.6. Елементарний демонстраційний клас
3.2.7. Метод equals()
3.2.8. Метод toString()
3.2.9. "Канонічний" склад класу
3.2.10. Використання IDE для автоматичного конструювання класів
3.2.11. Інтерфейсні типи
3.2.12. Перелічення
3.2.13. Виключення

Розділ 3.3. Примітивні та об'єктні типи
3.3.1. Різниця між примітивними та об'єктними типами
3.3.2. Чи є в Java указники
3.3.3. Низькорівнева реалізація списку
3.3.4. Коли x instanceof Object дорівнює false

Розділ 3.4. Набори даних: масиви та колекції
3.4.1. Поняття про набори даних
3.4.2. Основні риси масивів
3.4.3. Знайомство з класом Arrays: сортування масивів
3.4.4. Динамічне заповнення масивів
3.4.5. Масив з екземплярів класу
3.4.6. Проблема з розміром масивів
3.4.7. Перехід до колекцій
3.4.8. Знайомство з багатовимірними масивами

Контрольні запитання

ЧАСТИНА 4. АНАЛІЗ ЕФЕКТИВНОСТІ, ВІДЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ

Розділ 4.1. Відлагодження програм

4.1.1. Загальне уявлення про відлагодження
4.1.2. Найпростіші прийоми відлагодження
4.1.3. Використання протоколювання для відлагодження
4.1.4. Інструментальні засоби відлагодження

Розділ 4.2. Тестування програм

4.2.1. Роль тестування в програмному проекті
4.2.2. Знайомство з JUnit

Розділ 4.3. Якість та ефективність програм

4.3.1. Поняття про якість програмного забезпечення
4.3.2. Основні показники ефективності
4.3.3. Дослідження часу виконання фрагментів коду
4.3.4. Приклад: порівняння двох способів отримання рядка з примітивного типу
4.3.5. Приклад: порівняння інтерпретації та синхронної компіляції
4.3.6. Поняття про Java Code Conventions

Контрольні запитання

Вправи на програмування

Перелік лістингів програм

ПЕРЕДМОВА

Сьогодні Java є однією з найбільш популярних мов програмування. Вона набула значного поширення в індустрії програмного забезпечення як одна з основних платформ для створення розподілених корпоративних систем та мобільних застосувань. Вважається, що це одна з основних мов, яку повинен знати сучасний фахівець з IT-технологій.

З огляду на це вивчення Java займає помітне місце в підготовці бакалаврів, магістрів та спеціалістів з програмної інженерії, інформатики, комп'ютерних наук, інших споріднених напрямків. Багато університетів включають до своїх навчальних планів нормативні та вибіркові дисципліни, присвячені Java SE, Java EE, мобільним застосуванням. Мова Java використовуються для вивчення інших дисциплін – комп'ютерні алгоритми, обчислювальна математика, паралельні обчислення, інтелектуальні системи та ін. В ряді університетів та навчальних центрів Java взагалі викладається як перша мову програмування. Ставлення автора книги до цього факту є дуже неоднозначним, але факт має місце.

Існує багато гарних книжок з Java, в першу чергу двохтомник К.Хорстманна та Г.Корнелла "Java. Бібліотека професіонала" та Б.Еккеля "Філософія Java". Але все-таки ці книжки важко розглядати як класичні підручники. Вони орієнтовані скоріш на більш-менш грамотних програмістів, ніж на студентів-початківців. Не вистачає розгорнутих учбових прикладів, які ілюструють типові помилки, наслідки цих помилок та процес їх послідовного виправлення. Бракує прикладів, спрямованих на порівняльний аналіз різних засобів розв'язання тієї чи іншої задачі. З іншого боку, існують книжки, спеціально присвячені тонкощам та підводним каменям – але їм не вистачає систематичного викладення основного матеріалу. Недостатня увага приділяється інструментальним засобам – таким, як JUnit, засоби відлагодження та автоматичної генерації коду тощо. Практично не згадується дизасемблер – а в той же час аналіз байт-коду може стати потужним підгрунтям для більш глибокого розуміння ряду особливостей мови (конструювання класів, коваріантність тощо).

Тому планується написання серії навчальних посібників і підручників з Java, спрямованих на заповнення цієї прогалини. Цей посібник – це перша книга цієї серії; вона присвячена основним можливостям мови.

У книзі на інтуїтивному рівні розглядаються базові риси мови, необхідні для того, щоб почати програмувати на Java: організація типового консольного застосування, введення-виведення, основні алгоритмічні конструкції (цикли, умовні оператори тощо), основи об'єктно-орієнтованого програмування (будова типового класу, основні поняття про спадкування), інтерфейсні типи, перелічення, виключення, рядки, масиви. Після обговорення проблем, пов'язаних з масивами, вводиться поняття колекції як більш високорівневого набору даних; наводиться відповідний приклад. Розглядаються основні утиліти SDK, в тому числі дизассемблер: засоби хронометрування програм, а також засоби відлагодження та тестування – зокрема, JUnit.

Лістинги програм, які наводяться в книзі – це перш за все навчальні приклади, які ілюструють типові проблеми та засоби їх вирішення. Крім того, в деяких лістингах наводяться готові рецепти вирішення окремих задач.

До кожної частини наводяться контрольні запитання; в кінці посібника міститься перелік вправ на програмування.

Посібник передбачає певний попередній рівень знань з процедурного та об’єктно-орієнтованого програмування.

ЧАСТИНА 1. БАЗОВІ ЗАСАДИ МОВИ JAVA

Розділ 1.1. Короткий нарис історії мови

1.1.1. З чого все починалося

Мова Java розроблялася під егідою корпорації Sun Microsystems починаючи з 1991 року. Що до цього спонукало? Адже вже у 70-х роках минулого століття існувала величезна кількість мов програмування, і для створення нових мов потрібні були реальна необхідність та нова сильна ідея.

Отже, далекий 1990 рік. Багато нинішніх програмістів ще не народилися. Серед мов програмування домінують C та C++; для підвищення ефективності ряд фрагментів коду доводиться писати на асемблері. Не так давно з'явилася, але вже впевнено заявила про себе нова на той час парадигма об'єктно-орієнтованого програмування. Ще тільки починається масове поширення Інтернету. Графічні користувацькі інтерфейси та технології мультимедіа ще перебувають на початковому етапі розвитку.

У компанії Sun ведеться робота над проектом, метою якого було створення програмованих побутових пристроїв, зокрема контролерів для перемикання телевізійних каналів. Спочатку відповідне програмне забезпечення писалося на C++, але це викликало значні незручності. Пристрої, з якими потрібно було мати справу, дуже відрізнялися за своєю будовою та архітектурою. Для кожної конкретної архітектури розроблене програмне забезпечення доводилося компілювати заново.

Врешті-решт Патрік Ноутон, один з провідних інженерів компанії, вирішив, що з нього досить, і прийняв рішення перейти до іншої компанії. Він написав керівництву компанії листа, в якому пояснив причини свого рішення та піддав критиці те, що відбувалося в компанії, зокрема методи управління проектами.

Показово, що до цього кроку керівництво компанії та колеги Ноутона віднеслися з розумінням. Після деякого обговорення було визнано, що проблема дійсно є серйозною, але вирішити її в рамках існуючих підходів та технологій неможливо. Розробники фактично отримали карт-бланш: пропонуйте будь-які ідеї, але знайдіть вихід.

Було прийнято нестандартне рішення – відмовитися від застосування C++ та створити нову мову програмування. Так народився проект Green.

1.1.2. Зелений дуб та філіжанка кави

Лідерами і основними розробниками проекту Green, спрямованого на створення нової мови, стали Патрік Ноутон та Джеймс Гослінг. До цієї мови були висунуті очевидні вимоги:

1. Платформна незалежність. Власне кажучи, це була основна вимога, заради якої все і робилося. Звичайно, мова не йшла про переносимість програм на рівні початкових кодів – це питання в своїй основі було давно вирішеним. Мова йшла про те, щоб відкомпільовані програмні модулі не залежали від конкретної архітектури та могли працювати на будь-якій апаратній платформі та в будь-якому операційному середовищі. Як основа для рішення була взята концепція ***віртуальної машини***.

2. Компактність. Побутові пристрої – це пристрої недостатньо потужні, і тому виконання програмних модулів не повинно було бути ресурсномістким.

Спочатку нова мова називалася **Oak** (дуб) – на честь дуба, який ріс під вікнами офісу Гослінга. Потім з'ясувалося, що мова програмування з такою назвою уже існує, і нову мову перейменували на **Java**.

Назва Java теж досить показова. Java (Ява) – це назва сорту кави, який вирощується на індонезійському острові Ява. Цей острів, як і вся Індонезія, лежить на межі Індійського та Тихого океанів. В американському сленгу Java – це назва кави взагалі. Тому символ мови Java, який люблять зображати на посібниках – це чашка кави, над якою вирує димок.

Варіанти тут найрізноманітніші – від відверто мінімалістського фірмового логотипу до апетитних натуралістичних фотографій та малюнків.

Мал.1.1. Філіжанка кави – символ Java

Цікаве для східного слов'янина питання – як правильно називати цю мову: Джава чи Ява? Вважається правильною вимова Джава; саме на ній наполягає компанія Sun. Але і проти назви Ява мало хто заперечує – врешті-решт, острів все-таки називається Явою.

1.1.3. Перші розчарування та переломний момент

Перша розробка в рамках проекту Green під назвою "*7" була випущена у 1992 році. Це був пристрій для інтелектуального дистанційного керування.

Була проведена ефектна демонстрація в самій компанії, і всі присутні були в захваті. Але ні цей пристрій, ні подальші зразки (ігрові приставки, засоби для кабельного телебачення) не мали комерційного успіху, і продати їх не вдавалося. Патрік Ноутон безрезультатно налітав на літаку понад 300000 миль, намагаючись продати розроблену технологію.

Так буває з багатьма геніальними ідеями – якщо ідея не грає в одному місці, вона може зіграти в іншому. Успіх прийшов не з тієї сторони, звідки його очікували на початку - він був пов'язаний з Інтернет-технологіями. Тому інколи можна почути, що мова Java розроблялася для створення мережних та веб-застосувань, але це не так. Початковою метою було програмування побутових пристроїв, але все-таки...

Ми знаємо, що 90-і роки ознаменувалися бурхливим розвитком Інтернету, зокрема World Wide Web. За короткі 10 років Інтернет перетворився з суто технічного явища, про яке було відомо лише невеликому колу спеціалістів, на грандіозний соціальний феномен, який багато в чому визначає нинішній розвиток людства.

Але тоді був ще 1994 рік. Компанія Sun спочатку пропустила цей ривок, але швидко зрозуміла, що все ще можна надолужити. Народилася ідея створення ***аплетів***.

Аплет – це програмний модуль, написаний мовою Java, який виконується в середовищі браузера. За допомогою спеціального тегу посилання на клас аплету включається до веб-сторінки. При завантаженні сторінки браузер завантажує і сам аплет.

Виявилося, що мова Java ідеально підходить для цієї мети. Два основні фактори сприяли цьому:

- інтерактивність, тобто можливість взаємодії з користувачем. На відміну від реалій сьогодення, тоді навіть сама така можливість була дивиною – а аплети на

додачу до розвинених засобів користувацького інтерфейсу реалізовували ще й інтерактивну анімацію;
- уже згадана крос-платформність; браузери могли працювати на машинах з різною архітектурою і під управлінням різних операційних систем, і ця риса була дуже доречною.

Для демонстрації цієї технології був потрібний спеціальний браузер, який і був розроблений (він отримав назву HotJava). Аплети були продемонстровані на виставці Sun World'95 у травні 1995 року, і це мало величезний успіх.

При всій повазі до аплетів слід зазначити, що сьогодні ця технологія відходить у минуле. Виникли значні проблеми з застосуванням аплетів (не було гарантії, що в браузері встановлена підтримка сучасної версії Java, яка підтримує нові можливості графічного інтерфейсу). Сьогодні для доставки застосувань на клієнтський комп'ютер пропонується інша технологія – **Java Web Start**. Але у 1995 році аплети свою справу зробили – і почалося тріумфальне розповсюдження Java.

1.1.4. Важливі віхи та версії Java

Перша версія Java (Java 1.0) була випущена на початку 1996 р. Разом з нею був випущений і комплект для розробки застосувань – **Java Developer's Kit (JDK)**.

Ця версія, за загальним визнанням, була ще дуже сирою. Недосконалою була система введення-виведення; модель обробки подій у графічному інтерфейсі теж була дуже невдалою. Деякі виправлення були зроблені у версії 1.1, але справжньою революцією в світі Java став випуск Java 1.2 (1998 р). Ця версія стала називатися **Java 2** – так само, як і її найближчі наступники.

Змін у самій мові не було, але з'явилися нові бібліотеки, і в першу чергу – бібліотека **Swing** для програмування графічного користувацького інтерфейсу, яка є основним засобом для створення desktop-застосувань і сьогодні. Було здійснено розділення на три платформи Java 2:

- **Standard Edition** – стандартний пакет;
- **Enterprise Edition** – пакет для корпоративних застосувань;
- **Micro Edition** – пакет для портативних пристроїв (зокрема, для мобільних телефонів).

Нова революція відбулася у 2004 р. – версія Java 1.5 (або **Java 5**).

На жаль, те, що відбувалося з назвами версій Java, важко охарактеризувати інакше як творче безладдя. Стандартний пакет інструментальних засобів від Sun називався то JDK, то SDK. Версії Java 5 та Java 6 могли називатися відповідно 1.5 та 1.6, а як правильно – мало хто знає. Втім, завжди було зрозуміло, про що йде мова.

Повернемося до Java 5 (або Java 1.5). У цій версії відбулися суттєві зміни в самій мові. Найпомітніші з них – це generics (узагальнені, або параметризовані типи), модифікований for – зручний цикл для перегляду всіх елементів масиву або колекції. Було багато інших нововведень, на які ми будемо звертати увагу на протязі всієї книги.

Версія Java 6, або Java 1.6 (2006 р.), не внесла змін до мови; були додані нові класи.

Лише у 2011 році вийшла довгоочікувана Java 7. За цей час сама компанія Sun була придбана компанією Oracle, але її основні проекти залишилися. У цій версії відбулося ряд синтаксичних змін – але не всі з тих, які були анонсовані.

Нарешті, навесні 2014 року з'явилася Java 8, яка теж внесла ряд мовних новацій. Зокрема, йдеться про ламбда-вирази – зручні можливості для опису функцій.

На протязі всієї книги ми будемо постійно звертати увагу на зміни, які робилися в Java 5, Java 7 та Java 8.

Розділ 1.2. Основні властивості мови Java

1.2.1. Канонічний перелік властивостей

Автори Java розробили певний перелік основних властивостей цієї мови, і сьогодні цей перелік можна вважати канонічним. Отже, мова Java відповідно до цього переліку характеризується наступними рисами:

- простота;
- об'єктна орієнтованість;
- розподіленість;
- надійність;
- безпечність;
- незалежність від архітектури;
- портабельність;
- інтерпретованість;
- продуктивність;
- багатопоточність;
- динамічність.

Дамо деякі коментарі до цього переліку.

1.2.2. Простота

Java розроблялася на основі C++ і багато чого від неї взяла – а процедурні керуючі конструкції Java майже повністю відповідають їх аналогам у C. Насправді Java не така вже й проста, але дійсно спрощена у порівнянні з C++.

Мова C++ має в своєму розпорядженні величезний арсенал різноманітних можливостей. Але таке розмаїття вимагає впевненого володіння відповідними мовними засобами та грамотного їх використання. Розробники Java виключили ті риси C++, які, на їх думку, є або рідко використовуваними і важкими у вивченні, або концептуально сумнівними, або небезпечними з точки зору можливих програмних помилок.

Перелічимо деякі спрощення Java в порівнянні з C++:

- у Java немає множинного спадкування класів. Взагалі, навіть одиночне спадкування далеко не завжди використовується правильно, а з множинним спадкуванням у C++ навіть при грамотному застосуванні пов'язано багато концептуальних проблем;

- з різних варіантів одиночного спадкування класів, притаманних C++, у Java залишилося лише відкрите спадкування;

- у C++ змінні можна розміщувати або на стеку виконання програми, або в динамічній пам'яті. У Java такої свободи немає: екземпляри класів розміщуються в купі, а змінні примітивних типів – на стеку;

- у C++ є широкі можливості для передачі параметрів функцій: або за значенням, або за посиланням, або через указники. У Java параметри функцій завжди передаються за значенням;

- на відміну віх C++, якщо поле класу в Java має об'єктний тип, воно може бути лище указником на екземпляр. Відповідно до цього механізми конструювання об'єктів у Java можуть бути більш простими, ніж у C++; зникає особлива потреба в копіювальних конструкторах (яких у Java в явному вигляді і немає);

- у C++ багато програмних помилок пов'язано з неналежним знищенням об'єктів. У Java збирання сміття (тобто знищення об'єктів, які більше не використовуються), здійснюється автоматично, а операція явного видалення (`delete`) об'єктів взагалі відсутня;

- у Java немає можливості визначати операції в класах; механізм реалізації операцій міг би бути більш простим, ніж у C++, але розробники Java від цього відмовилися;

- в Java немає операцій взяття адреси, розіменування та арифметики указників з огляду на значні небезпеки, пов'язані з їх неправильним використанням;

- в Java немає модифікатора `virtual`; в залежності від їх опису методи класу є або поліморфними, або ні; керувати цим немає можливості.

З іншого боку, Java надає деякі нові можливості, яких немає в C++. Зокрема, в явному вигляді підтримується концепція інтерфейсів. Розширюються можливості, пов'язані з внутрішніми класами та з обробкою виключень (секція `finally`, яка виконується незалежно від того, чи було виключення чи ні). На рівні мови підтримується багатопотоковість. У Java є розвинені засоби рефлексії, тобто аналізу класів на етапі виконання.

Звісно, питання про те, що Java має повністю замінити C++, не ставиться – кожна з цих мов має свою нішу та свою область застосування.

1.2.3. Об'єктна орієнтованість

У 1991 році об'єктно-орієнтована парадигма уже вважалася магістральною, і тому Java просто зобов'язана була стати об'єктно-орієнтованою мовою. На цьому можна було б і зупинитися, але між об'єктними рисами C++ та Java є ряд серйозних відмінностей.

Часто кажуть, що Java є **повністю об'єктно-орієнтованою**. Якщо в C++ ми не хочемо використовувати класи, а хочемо написати програму в чисто процедурному стилі, ми можемо це зробити. У Java це не так. Програмний код завжди має бути інкапсульований до класів. Таким чином, **будь-яка програма**, навіть найпростіша, **повинна бути оформлена як клас** (один або декілька).

Зразу ж розглянемо елементарний приклад – програму, яка виводить на екран рядок `Hello, world.` Якщо це все, що потрібно - то в об'єктно-орієнтованому аналізі, проектуванні і програмуванні для написання такої програми, звісно, немає ніякої необхідності. За своєю суттю програма чисто процедурна, але **в Java ми не можемо обійтися без класів** – поки що вважаємо це необхідною вимогою синтаксису.

Код може мати вигляд:

```
public class HelloWorld {

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

 }

}
```

Лістинг 1.2.3.1. `Hello, world:` навіть найпростіша програма вимагає створення класів

Відкладемо більш детальний аналіз цієї програми на більш пізній час; зараз потрібно сказати про таке.

Ми бачимо один клас, який називається `HelloWorld`. **Будь-яка функція – це метод класу**; ніяких змінних або функцій за межами класів у Java бути не може. Надалі ми будемо вважати терміни "функція" і "метод (класу)" синонімічними, і вживати термін "функція", якщо потрібно буде відтінити процедурні аспекти мови, і "метод" – якщо об'єктно-орієнтовані.

У наведеному класі є лише один метод – метод `main`. Як і в C/C++, це ключовий метод, з якого починається виконання програми; часто кажуть, що це точка входу до програми. Метод `main` може бути як завгодно складним, але в даному випадку все, що він робить – виводить рядок за допомогою виклику методу відповідного класу.

Об'єктні риси Java в своїй основі відповідають загальноприйнятим канонам об'єктно-орієнтованого програмування. Вони в цілому схожі на аналогічні риси C++, але є ряд суттєвих відмінностей; про деякі з них йшлося в попередньому розділі.

1.2.4. Розподіленість.

Уже Java Standard Edition має багато засобів для мережної взаємодії (зокрема, читання веб-ресурсів, обмін даними з віддаленими серверами, підтримка сокетів тощо), а Enterprise Edition та Micro Edition безпосередньо орієнтовані на створення корпоративних та мобільних застосувань, які є розподіленими за своєю природою. Тому Java-технології розглядаються як **платформа для створення розподілених застосувань**. У цьому контексті варто згадати про одну з базових технологій мережної взаємодії на основі Java – **RMI** (**Remote Method Invocation**).

Але розподіленість передбачає не тільки фізичну розподіленість та власне обмін даними по мережі. Не менше значення має і взаємодія між логічними рівнями прикладної програми – зокрема, між рівнем бізнес-логіки та рівнем даних у рамках трьохрівневої клієнт-серверної архітектури. І тут Java надає ряд стандартних засобів:

- розвинена система потоків введення-виведення, зокрема для читання і запису файлів;
- взаємодія з реляційними базами даних (в першу чергу **JDBC**; фактичним стандартом став інструмент об'єктно-реляційного відображення **Hibernate,** хоч він і не входить до стандартної поставки Java);
- читання та модифікація XML-документів.

Крім того, існують пакети для роботи з іншими типами даних (наприклад, JSON).

1.2.5. Надійність

Тут під надійністю мається на увазі захищеність програм від випадкових внутрішніх помилок. Як відомо, основним джерелом програмних помилок у таких мовах, як C/C++ (і Паскаль) є некоректне звернення до пам'яті (пов'язані в першу чергу з неправильним використанням указників та виходом за межі масиву). У Java такі помилки практично виключені. Арифметика указників, операції розіменування та взяття адреси у мові відсутні, так що напряму звернутися до ділянки пам'яті за її адресою неможливо (якщо тільки не викликаються процедури в рідному машинному коді). Виходи за межі масиву жорстко контролюються, і при спробі звернутися до елементу масиву з неіснуючим індексом виникає виключення (помилка часу виконання).

1.2.6. Безпечність.

В першу чергу тут ідеться про захист від злонамірного коду (хоча контроль за використанням пам'яті теж має пряме відношення до безпеки). Ми уже казали, що Java орієнтована на роботу в мережі, а **все, що приходить з мережі, вважається потенційно небезпечним**. Наприклад, легко уявити, що було б, якби аплет, завантажений з довільного сайту, міг би робити що завгодно – наприклад, записати щось на диск або взагалі відформатувати його. Тому Java передбачає певну систему безпеки, яка контролює потенційно небезпечні операції і може їх дозволяти відповідно до тих чи інших політик безпеки.

1.2.7. Незалежність від архітектури комп'ютера.

Незалежність від архітектури, або крос-платформність, нагадаємо, була однією з основних рис, заради якої взагалі створювалася Java. Ця властивість означає, що **відкомпільовані модулі, написані на Java, повинні працювати на будь-якій машині з будь-якою архітектурою та під управлінням будь-якої операційної системи**. Sun каже з цього приводу: "**Write Once - Run Anywhere**" (напиши один раз - запускай де завгодно).

Механізми досягнення крос-платформності спираються на концепції ***віртуальної машини*** та ***байт-коду***. Була розроблена специфікація віртуальної машини Java – абстрактної машини з власним набором команд. Початковий код Java-програми компілюється в послідовність команд цієї абстрактної машини (байт-код), яка інтерпретується на конкретній платформі. Більш детально це буде описано в наступному розділі.

1.2.8. Переносимість.

Для забезпечення переносимості (портабельності) програм з однієї пдатформи на іншу розробники Java намагалися реалізувати принцип, відповідно до якого в мові не повинно бути машинно-залежних рис, або точніше – **жодний аспект специфікації Java не повинен залежати від реалізації**.

Класичне питання на цю тему: який розмір має тип `int` в мові C? Правильною відповіддю вважається `sizeof(int),` оскільки цей розмір може бути різним на різних платформах. У Java оператор `sizeof` взагалі відсутний, і для всіх стандартних типів даних розмір визначається специфікацією.

В ідеалі поведінка програми при перенесенні з однієї машини на іншу не повинна змінюватися. Насправді не все настільки гладенько. Має значення швидкодія процесора, особливо коли паралельно працює багато потоків. У багатопотоковій програмі велике значення має механізм розподілу процесорного часу, який залежить від операційної системи. Різні процесори можуть забезпечувати різну точність при роботі з дійсними числами, і це викликало необхідність в модифікаторі `strictfp`. Можна навести багато інших прикладів на цю тему.

1.2.9. Інтерпретованість

Як ми зазначали, програма на Java компілюється в проміжний байт-код, який далі інтерпретується. Класична інтерпретація має передбачати, що чергова команда байт-коду має бути проаналізована і запущена на виконання.

1.2.10. Продуктивність

Інтерпретація байт-коду є повільним процесом, і тому розробники передбачили можливість перетворення окремих фрагментів проміжного коду в машинний код. Йдеться про механізм **JIT-компіляції**, який буде розглядатися далі.

1.2.11. Багатопотоковість

У Java на рівні мови підтримуються **потоки** – "легкі" процеси, які можуть паралельно працювати, взаємодіяти між собою в рамках однієї програми і, відповідно, використовувати спільний адресний простір.

1.2.12. Динамічність

У Java порівняно легко писати, розповсюджувати та використовувати нові пакети та бібліотеки. Типовим є розповсюдження ***jar-архівів*.**

По суті, jar-архів – це і є ***бібліотека***, яка містить класи та функції для розв'язання тих чи інших задач. Такий архів може бути легко завантажений, зокрема з Інтернету, і підключений до власного програмного проекту. Можна створювати власні бібліотеки. Стандартним інструментом для створення jar-архівів є утиліта **jar**, яка входить до складу JDK.

Бібліотеки тісно пов'язані з пакетами (фізично та логічно об'єднаними групами класів), хоча цей зв'язок не завжди є настільки однозначним.

Розділ 1.3. Крос-платформність, інтерпретація та компіляція

1.3.1. Традиційна схема виконання програм

При традиційному підході до програмування мовами високого рівня (Паскаль, C/C++ та ін..) ***початковий код***, тобто текст програми, написаний однією з таких мов, компілюється в ***об'єктний код*** – послідовність машинних команд, зрозумілих процесору. Механізм роздільної компіляції дозволяє компілювати різні частини програми незалежно. Редактор зв'язків дозволяє скомпонувати окремі відкомпільовані об'єктні модулі в один ***завантажувальний модуль***, який може бути завантажений до пам'яті та безпосередньо виконаний даним процесором.

Так досягається переносимість програм на рівні початкового коду. Якщо потрібно перенести програму на машину з іншою архітектурою та іншим набором машинних команд, початковий код має бути відкомпільований заново.

Але розробники Java ставили перед собою мету досягти переносимості програм не на рівні початкових кодів, а на рівні відкомпільованих модулів.

1.3.2. Віртуальна машина та байт-код

Вимога крос-платформності полягає в тому, що відкомпільовані Java-програми повинні виконуватися на машинах з довільною архітектурою та працювати в будь-якому операційному середовищі. Як основа для досягнення цієї мети (**Write Once – Run Anywhere**), була використана концепція віртуальної машини.

Відповідно до класичного визначення, ***віртуальна машина*** – це сукупність ресурсів, які надаються певною машиною для емуляції роботи деякої іншої машини. Sun розробила специфікацію абстрактної Java-машини з певним набором команд. Початковий код Java-програми компілюється не в "рідні" машинні коди, а в проміжний код - **байт-код.**

Байт-код - це послідовність команд Java-машини. Зрозуміло, що з одного боку цей код є незалежним від конкретної машинної архітектури, а з іншого – конкретна машина повинна надавати засоби для виконання цього коду. Саме це і є основним завданням Java Virtual Machine (JVM) – ***віртуальної машини Java.***

Крім віртуальної машини, для виконання Java-програм потрібна наявність певного оточення, яке називається **Java Runtime Environment (JRE).**

1.3.2. Інтерпретація та JIT-компіляція

JVM забезпечує інтерпретацію та виконання байт-коду на конкретній платформі. Але класична покрокова інтерпретація - це досить повільний процес, і тому з'явився механізм ***JIT-компіляції*** (Just-In-Time-компіляція; інша назва - ***синхронна компіляція***).

Суть JIT-компіляції полягає в наступному. Часто повторювані фрагменти байт-коду транслюються в машинний код, який зберігається в пам'яті і викликається при необхідності. Перш за все це стосується циклів. Але мова йде саме про запам'ятовування окремих фрагментів коду, а не про створення готового до виконання файлу в машинних кодах.

Існують спеціальні механізми, спрямовані на налаштування та оптимізацію режиму синхронної компіляції. Стандартний інтерпретатор працює в комбінованому режимі. Спочатку виконання програми проходить в режимі класичної інтерпретації, в ході якої накопичується необхідна статистика. На основі цього відбувається перехід до подальшої JIT-компіляції.

Розділ 1.4. Проблема захисту

1.4.1. Рівні захисту Java-програм

Захищеність Java-програм – це ще один аспект, на який варто звернути увагу. Розглядаються певні рівні захисту, а саме:

- рівень мови: виключені потенційно небезпечні операції (в першу чергу – арифметика указників); жорсткий контроль за виходом за межі масиву і т.п; про це вже йшлося в розділі 1.2.5;
- ***верифікація***, яка здійснюється при завантаженні класів. Верифікація передбачає перевірку правильності коду (наприклад, відсутність неініціалізованих змінних). Стандартний компілятор Java сам контролює подібні помилки і повинен створювати правильний байт-код, який буде успішно верифіковано. Але необхідність верифікації диктується тим, що цей код можна змінити вручну.
- ***менеджер безпеки*** (SecurityManager), який контролює потенційно небезпечні дії;
- використання **цифрових підписів**.

1.4.2. Поняття про менеджер безпеки

Зупинимося більш детально на аспектах, пов'язаних з використанням менеджеру безпеки. Методи класів. які відповідають за виконання певних потенційно небезпечних дій (наприклад, запис на диск), повинні звертатися до класу **SecurityManager**. Якщо даний код має дозвіл на виконання цієї операції, робота програми продовжується; якщо ж ні – генерується відповідне виключення (помилка часу виконання).

1.4.3. Модель пісочниці

Як базова розглядається модель безпеки, прийнята в Java 1.0. Ця модель відома як модель ***sandbox*** (пісочниця). Застосування, які виконуються під управлінням операційної системи, можуть здійснювати будь-які операції. Але аплети, які завантажуються по мережі та виконуються в середовищі браузера, обмежені правилами "пісочниці".

Існує кілька правил пісочниці, основними з них є наступні:

- аплет не може звертатися до локальної файлової системи;
- аплет не може здійснювати мережних з'єднань, крім того сервера, з якого він був завантажений ("аплет може дзвонити тількі додому");
- аплет не може запускати модулі в машинному коді.

1.4.4. Зміни в моделі безпеки

В моделях безпеки Java 1.1 та Java 2 відбулися суттєві зміни:

- починаючи з Java 1.1, "надійні", підписані аплети можуть отримувати ті ж права, що й застосування;
- починаючи Java 2, самі права стають вибірковими; для їх встановлення використовуються файли ***політик безпеки***.

Розділ 1.5. Технології, платформи, програмні моделі

1.5.1. Загальний огляд Java-технологій

Java – це не тільки мова програмування. На основі цієї мови розвивається ряд технологій, спрямованих на різноманітні практичні застосування. По-перше, технології на основі Java розглядаються як одна з платформ для створення розподілених корпоративних застосувань. По-друге, Java була і залишається однією з основних мов для програмування мобільних пристроїв, спеціалізованих процесорів тощо.

Виділяються наступні платформи Java, орієнтовані на різні практичні застосування і відповідно – на різні технології:

- **Java Standart Edition (SE)** - базові можливості;
- **Java Enterprise Edition (EE)** – бібліотеки для програмування корпоративних, зокрема веб-застосувань;
- **Java Micro Edition (ME)** - для програмування портативних пристроїв – наприклад, мобільних телефонів;
- **Java Card** – для пристроїв, які не мають власного користувацького інтерфейсу – наприклад, для смарт-карт.

Набула значної популярності **JavaFX** – набір інструментальних засобів для створення Rich Internet Applications з розвиненим графічним інтерфейсом користувача.

1.5.2. Програмні моделі

Поняття програмної моделі передбачає певний спосіб організації Java-програми, тобто певний набір ключових методів. Наприклад, для звичайних застосувань таким ключовим методом є метод **`main`** – основна точка входу до програми. При запуску програми віртуальна машина шукає цей метод та викликає його, і з цього починається виконання програми. Якщо метод `main` з потрібною сигнатурою (набором параметрів певного типу) не знайдений, виникає **виключення** – помилка часу виконання, яка призводить до аварійного завершення роботи програми.

Часто кажуть про ***життєвий цикл програми*** та пов'язаний з ним набір методів, які є специфічними для кожної програмної моделі і автоматично викликаються у відповідних ситуаціях; саме ці методи перш за все визначають функціональність програми. У цьому контексті можна сказати, що метод `main` визначає життєвий цикл консольних та графічних застосувань.

Зазвичай виокремлюють такі програмні моделі:

- application – програми, які виконуються під управлінням операційної системи; вони можуть бути консольними або забезпечувати графічний інтерфейс користувача;
- applet – програмні модулі, які виконуються в середовищі браузера;
- midlet – модулі, які виконуються на мобільних телефонах;
- xlet – модулі для портативних пристроїв, які є більш потужними, ніж мобільні телефони;
- servlet – модулі, які виконуються під управлінням веб-сервера або сервера застосувань;
- ejb – модулі, які працюють в рамках компонентної моделі EJB (Enterprise Java Beans).

1.5.3. Java та .NET

Конкуренція між компаніями Sun та Microsoft в кінці 90-х років минулого століття призвело до появи конкуруючої платформи для створення розподілених корпоративних систем, яка розвивається та підтримується Microsoft - **.NET**. Основною мовою .NET стала спеціально розроблена мова програмування - **C#.**

Як і Java, .NET реалізує концепцію байт-коду і теоретично декларує крос-платформність. Але в Microsoft не приховують, що вони розвивають .NET з орієнтацією

саме на Windows. Існує альтернативний проект – **Mono**, орієнтований на забезпечення крос-платформності.

З іншого боку, характерною рисою, "фішкою" .NET є багатомовність (***cross-language***): можливість взаємодії в рамках одного застосування модулів, написаних різними мовами програмування.

Як мова програмування, C# багато в чому схожий на Java, але має ряд суттєвих відмінностей і в багатьох аспектах наближається до C++. Оскільки C# розроблявся пізніше, його розробники мали можливість використовувати досвід розробки Java. Відмітимо деякі з найбільш помітних особливостей C#:

- подібно до C++ і на відміну від Java, у C# в класах можна визначати оператори;

- подібно до C++, C# має модифікатор `virtual`, який робить функцію віртуальною, тобто забезпечує її поліморфність. Крім того, для керування віртуальністю введено модифікатори `new` та `override` (у C++ подібні можливості відсутні);

- передбачений модифікатор `unsafe` (небезпечний), який дозволяє використовувати операції з указниками.

C# має деякі риси, яких немає ні в C++, ні в Java (принаймні, в існуючих версіях), зокрема:

- підтримуються так звані автореалізовані властивості, тобто пов'язані з полями класу стандартні селектори і модифікатори, які не потрібно розписувати явно;

- існує ключове слово для опису делегатів, які часто (хоч і не зовсім точно) називають указниками на функції;

- в останніх версіях C# реалізовано ряд можливостей, характерних для функціонального програмування; зокрема, підтримуються ламбда-вирази – по суті спрощені описи функцій. Подібні можливості почали з'являтися лише у Java 8.

Розділ 1.6. Веб-сторінка Java

Існує офіційна веб-сторінка, присвячена Java – http://www.oracle.com/technetwork/java/index.html, більш відома за своєю старою адресою - http://java.sun.com. На цій сторінці можна знайти дві специфікації – нормативні документи для тих, хто програмує на Java та/або розробляє засоби підтримки для програмування цією мовою.

Специфікація **Java Language Specification (JLS)** визначає основні риси мови, нормативні для будь-якої реалізації. На даний момент не видно суттєвої альтернативи стандартній реалізації від Oracle (JDK), але можна розглядати можливість появи інших реалізацій.

Специфікація **Java Virtual Machine Specification (JVMS)** регламентує принципи роботи віртуальної машини Java, а також містить опис байт-коду.

Крім того, на офіційній сторінці містяться файли для завантаження, документація тощо.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Якою була основна мета створення Java в рамках проекту Green?
2. Чому символом Java стала філіжанка кави?
3. Що таке аплет?
4. Яку роль відіграла технологія аплетів у становленні Java?
5. Який зміст зазвичай вкладається у формулювання “Java є повністю об’єктно-орієнтованою мовою”?
6. Охарактеризуйте поняття крос-платформності та механізм її досягнення в Java.
7. Що таке байт-код?
8. Охарактеризуйте поняття JIT-компіляції. На чому ґрунтується підвищення ефективності JIT-компіляції в порівнянні з класичною інтерпретацією?
9. Охарактеризуйте основні механізми забезпечення надійності та безпеки, реалізовані в Java на рівні основних властивостей мови.
10. Охарактеризуйте суть верифікації, яка здійснюється при завантаженні класу. У чому полягає необхідність верифікації?
11. Для чого призначений менеджер безпеки? Охарактеризуйте механізм його використання.
12. Охарактеризуйте модель пісочниці та основні обмеження на аплети в рамках цієї моделі.
13. Які зміни відбулися в Java 1.1 та Java 2 порівняно з моделлю пісочниці?
14. Перелічіть основні платформи Java.
15. Перелічіть основні програмні моделі Java.
16. Який метод є основною точкою входу для Java-застосувань, які виконуються під управлінням операційної системи?
17. Яка технологія вважається основним конкурентом Java-технологій?
18. Охарактеризуйте специфікації JLS та JVMS.
19. Охарактеризуйте поняття jar-архіву.
20. Який Ви знаєте інструмент для створення власних пакетів і бібліотек?

ЧАСТИНА 2. НАПИСАННЯ НАЙПРОСТІШИХ КОНСОЛЬНИХ ЗАСТОСУВАНЬ

Розділ 2. 1. Основні засоби для створення Java-застосувань

2.1.1. Загальний огляд інструментів для написання Java-програм

У першу чергу слід звернути увагу на **JDK** (**Java Development Kit**) – пакет утиліт від Oracle; здебільшого це утиліти командного рядка. Але, як правило, зручніше працювати з ***інтегрованими середовищами розробки*** (IDE), які суттєво полегшують виконання багатьох етапів та дозволяють значною мірою автоматизувати процес створення проекту в цілому. На момент написання цього розділу найбільш популярними з них є **NetBeans** та **Eclipse,** а також **IDEA**.

Якщо порівнювати NetBeans та Eclipse, їх можливості є більш-менш рівнозначними (принаймні, на початковому етапі вивчення, а також для створення простих застосувань відмінності між цими пакетами не мають особливого значення). Можна сказати, що спеціалісти з Oracle більшою мірою рекомендують NetBeans. Професіональні програмісти більше люблять Eclipse, ніж NetBeans, але NetBeans часто розглядається як пакет, більш зручний для початкового вивчення.

В процесі розробки часто використовуються інші засоби. Деякі з них входять до складу IDE, інші є автономними утилітами або просто класами і методами класів, які можуть включатися до складу прикладних програм. Серед цих засобів відмітимо такі:

- засоби ***тестування,*** в першу чергу **JUnit**;
- засоби ***відлагодження***;
- засоби ***профілювання*** програм, призначені в першу чергу для оцінки часу виконання та пам'яті, що витрачається:
- засоби ***автоматизованої збірки проектів*** (**Maven, Ant**);
- засоби для групової роботи.

2.1.2. Основні утиліти JDK

Найбільш відомими та використовуваними утилітами JDK є:

- **java** – інтерпретатор байт-коду;
- **javac** – компілятор, який на основі початкового тексту створює байт-код;
- **jar** – утиліта для створення архівів;
- **javadoc** – утиліта для автоматизованого створення документації
- **javap** – дизасемблер байт-коду, який дозволяє проаналізувати відкомпільовану програму та подивитися її байт-код у мнемонічному вигляді, зручному для сприйняття людиною.

2.1.3. Компіляція та виконання програм засобами JDK

Напишемо застосування з мінімальною функціональністю, яке ілюструє типову структуру консольних Java-застосувань. Програма, яка виводить на екран статичний текст `Hello, world!`, має вигляд:

```
/* The simplest program*/
public class HelloWorld {

public static void main (String[] args) {


System.out.println("Hello, world!");

}
}
```

Лістинг 2.1.3.1. Проста програма – компіляція та виконання

Ми бачимо, що навіть така проста програма має бути оформлена як клас (в даному випадку **HelloWorld**). Текст цієї програми має бути збережений у файлі **HelloWorld.java**. Java вимагає, щоб загальнодоступні класи (ті, які супроводжуються модифікатором public) зберігалися у файлах, які мають розширення .java, а назва файлу повинна співпадати з іменем класу.

Тут і надалі ми будемо орієнтуватися на використання операційної системи `Windows`.

Запустимо сеанс командного рядка і перейдемо до каталогу, в якому міститься файл з початковим кодом програми. Компіляція програми здійснюється командою

```
javac HelloWorld.java
```

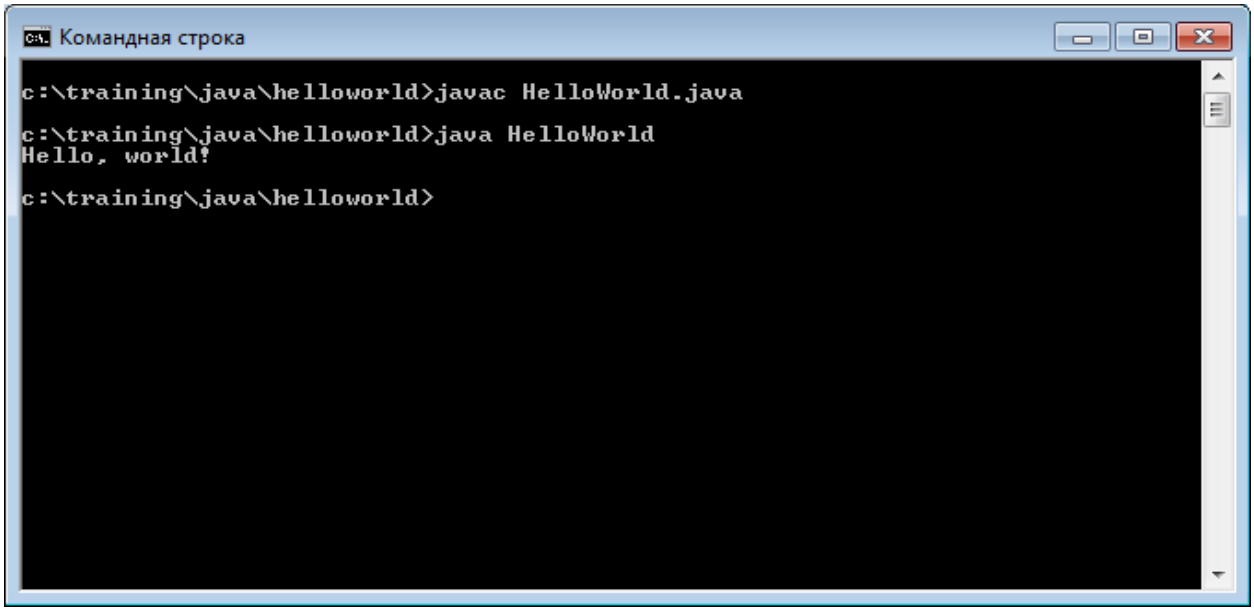
В результаті компіляції утворюється байт-код, який виконується під управлінням віртуальної машини. Технічно байт-код зберігається у файлі з розширенням **.class**; для кожного класу початкового файлу утворюється окремий файл з байт-кодом. У нашому випадку буде створено файл `HelloWorld.class`.

Для виконання програми слід викликати інтерпретатор, тобто запустити віртуальну машину. Наберемо команду

```
java HelloWorld
```

(без розширення).

Ми побачимо наступний результат:



2.1.4. Налаштування змінних оточення

Не завжди все йде настільки гладенько – навіть якщо сама програма написана без помилок. Для нормальної роботи в середовищі JDK необхідно налаштувати змінні ***оточення*** – відповідні змінні операційної системи. Як мінімум, ідеться про змінні **PATH та CLASSPATH.**

Часто може трапитися так, що система відмовляється виконувати, наприклад, компіляцію і повідомляє про те, що не розуміє команди javac. Це означає. що операційна система не може знайти утиліту `javac`, оскільки не знає, в якому каталозі вона міститься.

У цьому випадку утиліту javac можна запустити на виконання, якщо написати її повне ім'я (власну назву разом зі шляхом до утиліти). Але набагато зручніше попередньо вказати шлях до потрібних утиліт у змінній оточення `PATH`; це робиться стандартними засобами операційної системи.

Для того, щоб віртуальна машина могла знайти потрібні класи, каталоги, в яких вони знаходяться, або ж відповідні jar-файли мають бути вказані в змінній оточення `CLASSPATH`.

2.1.5. Програма з кількома класами

Проведемо невеликий експеримент.

Напишемо просту програму, яка складається з двох класів (зараз нам достатньо знати лише те, що один клас може використовувати інший – наприклад, якщо в ньому буде викликаний потрібний метод цього класу).

Початковий код має вигляд

```
public class MainClass {

public static void main (String[] args) {

Informer.inform();

}
}

class Informer {
public static void inform() {
System.out.println("This is very important information");
}
}
```

Лістинг 2.1.5.1. Програма з кількома класами

Тут ми маємо два класи: клас **Informer**, метод **inform()** якого виводить на екран потрібне повідомлення, та клас **MainClass**, який викликає цей метод. Для спрощення ми зробили метод **inform()** статичним – тоді не буде потреби в створенні екземплярів класу Informer; виклик методу здійснюється через ім'я класу (**Informer.inform()).**

Знову-ж таки, клас MainClass повинен містити метод main().

В якому файлі треба зберегти цей текст?

Правильно – **MainClass.java** (за іменем класу).

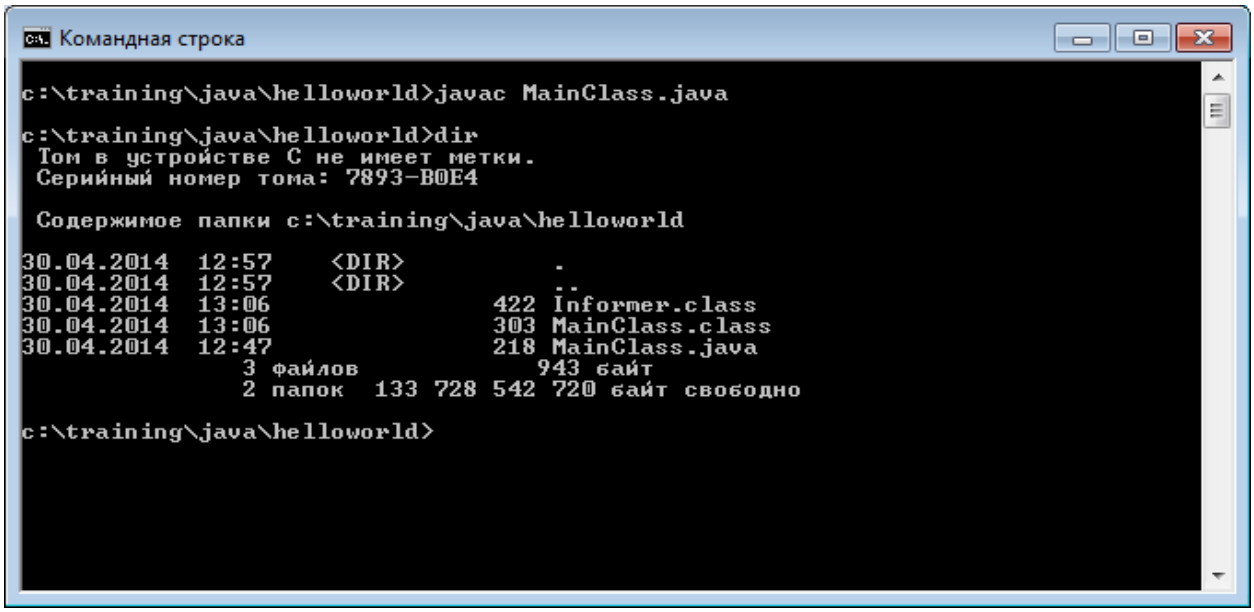
Може виникнути питання – а як же бути з класом Informer, адже він теж має зберігатися у відповідному файлі? Ні, це не обов'язково. Правило щодо відповідності імен файлів і класів стосується лише класів з модифікатором public, а клас Informer не має такого модифікатора і не є публічним.

Звісно, ми **можемо** розмістити кожний клас в окремому файлі (у більшості випадків саме так і потрібно робити).

Як і раніше, запустимо сеанс командного рядка та перейдемо до каталогу, в якому зберігається файл MainClass.java. Відкомпілюємо його так само, як і раніше:

```
javac MainClass.java
```

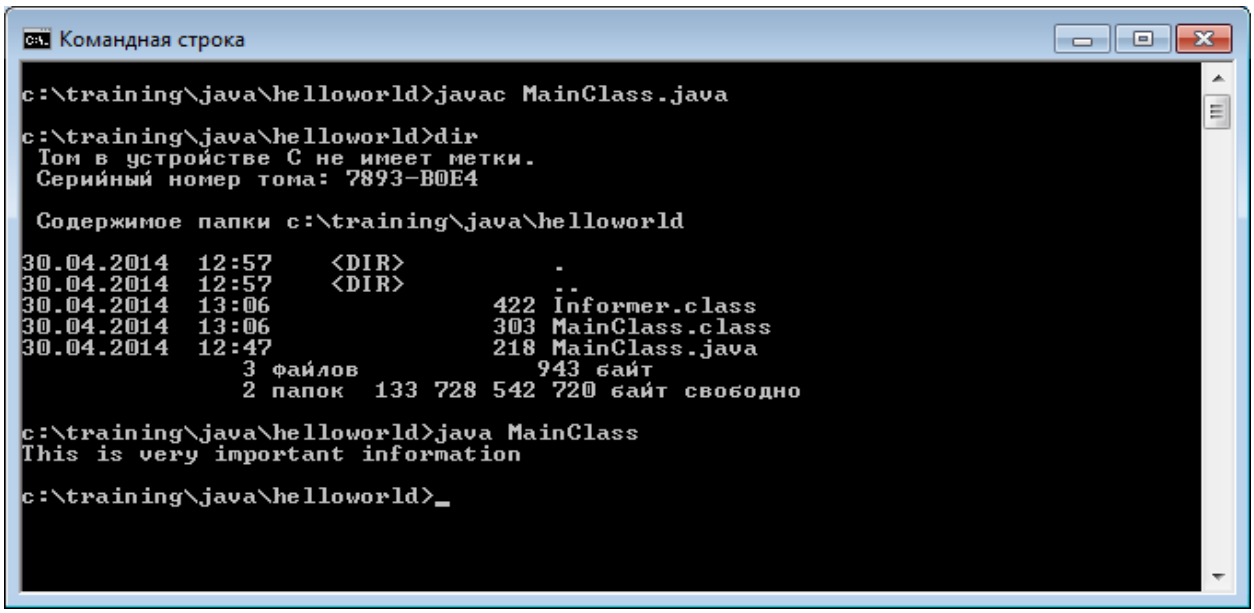
Незважаючи на те, що клас Informer явним чином не компілювався, він все одно буде відкомпільований та буде створений відповідний файл з байт-кодом. Таким чином, буде створено два файли з розширенням .class (MainClass.class та Informer.class) – за кількістю класів у файлі. Ми можемо в цьому переконатися за допомогою команди `dir` (ще раз нагадаэмо, що ми розглядаємо середовище Windows, в Unix-подібних системах аналогом цієї команди є `ls`). В результаті ми побачимо



Виконаємо метод main класу MainClass:

```
java MainClass
```

Як і слід було чекати, ми побачимо



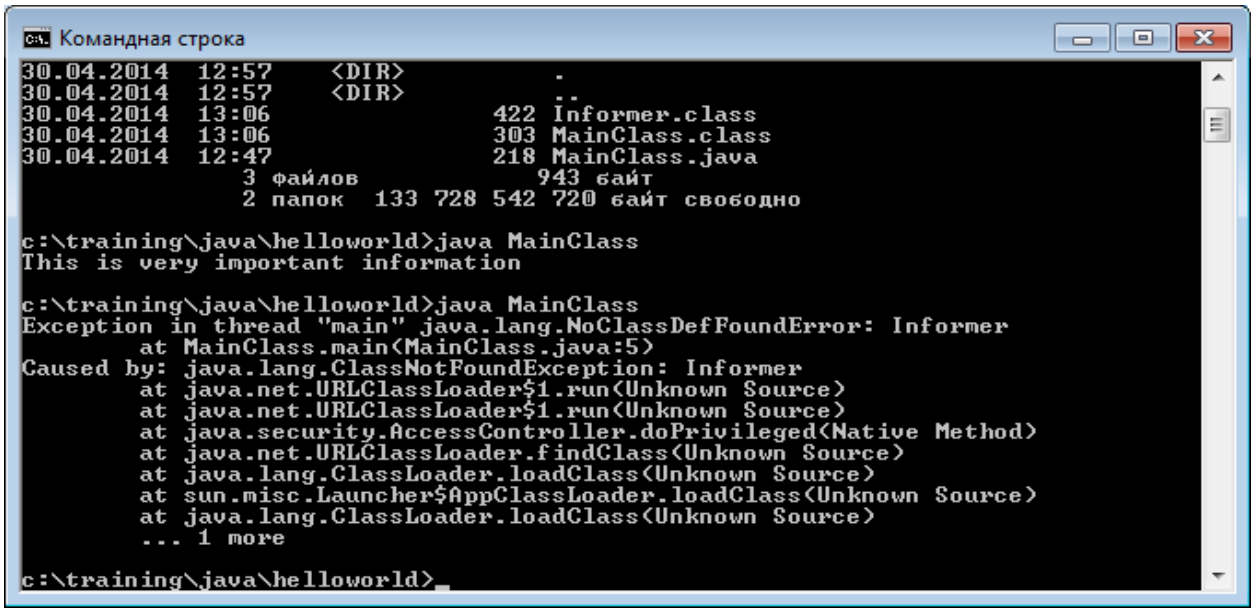
2.1.6. Опція –classpath

Тепер проведемо невеликий експеримент. Створимо підкаталог sub та перемістимо до нього (не скопіюємо) файл Informer.class.

Спробуємо тепер виконати основну програму (MainClass):

```
java MainClass
```

Ми побачимо повідомлення про помилку:

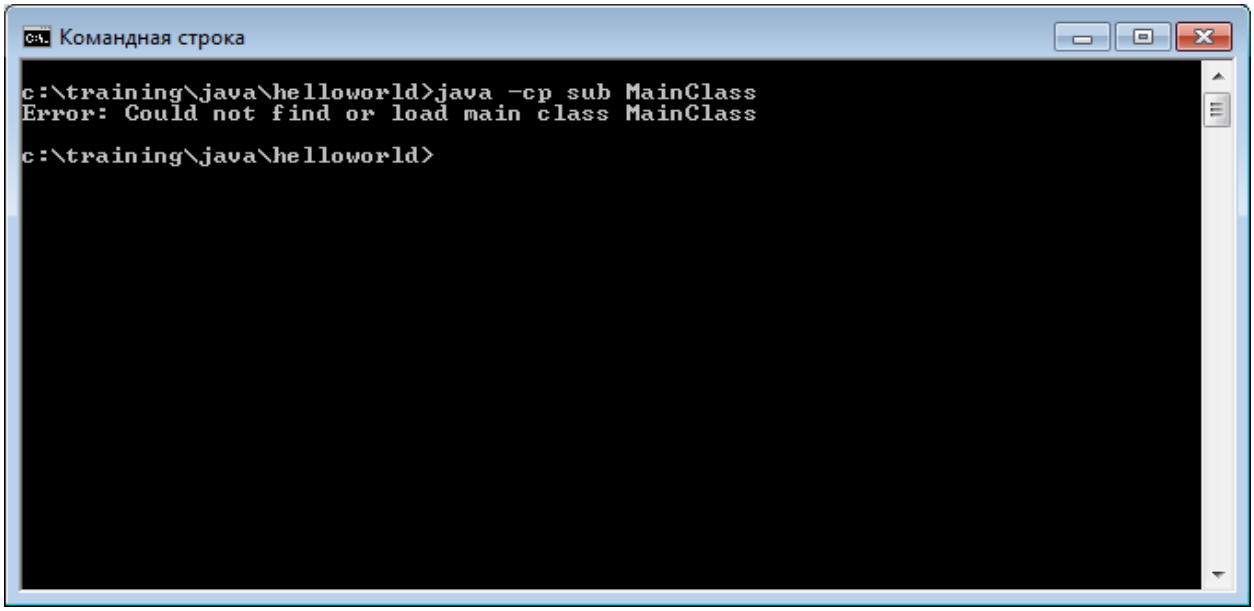


Дійсно, тепер файл Informer.class знаходиться в іншому каталозі, цей каталог не вказаний у змінній CLASSPATH, і тому віртуальна машина не може його знайти.

Для того, щоб все-таки запустити програму один або кілька разів, не обов’язково включати всі потрібні каталоги до CLASSPATH. Можна вказати їх безпосередньо при запуску інтерпретатора за допомогою опції –classpath (або скорочено -cp):

```
java –cp sub MainClass
```

Несподівано бачимо:

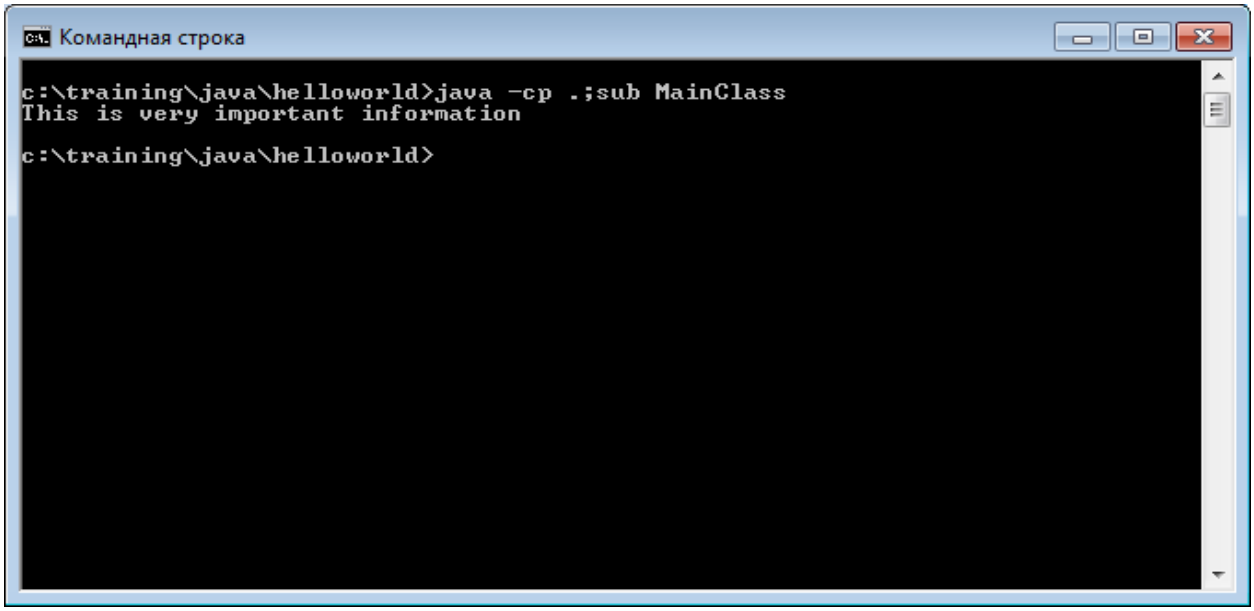


Як же так? Тепер інтерпретатор не знаходить класу MainClass в поточному каталозі?

Це є характерний “підводний камінь”. Справа в тому, що якщо використовується опція –cp, то враховуються тільки ті каталоги, які вказані в цій опції, і тому пошук в поточному каталозі не здійснюється. Вихід є: в опції –cp потрібно вказати і поточний каталог:

```
java –cp .;sub MainClass
```

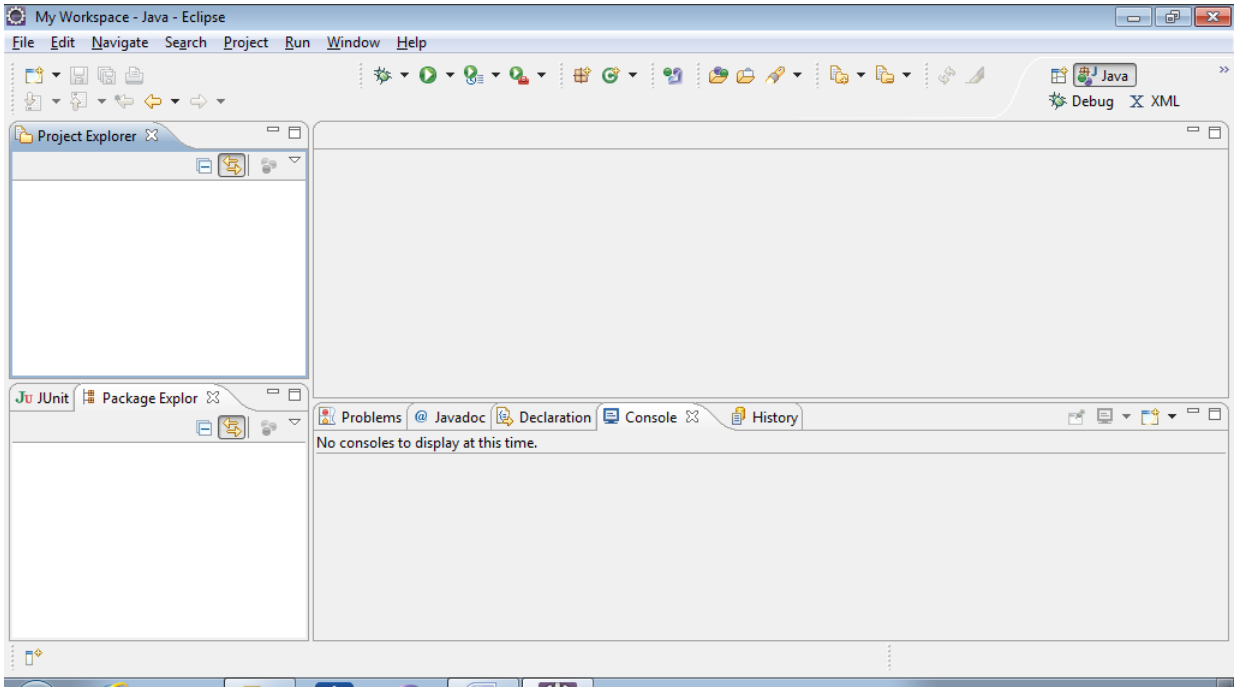
Тепер все правильно:



2.1.7. Створення Java-застосування засобами Eclipse

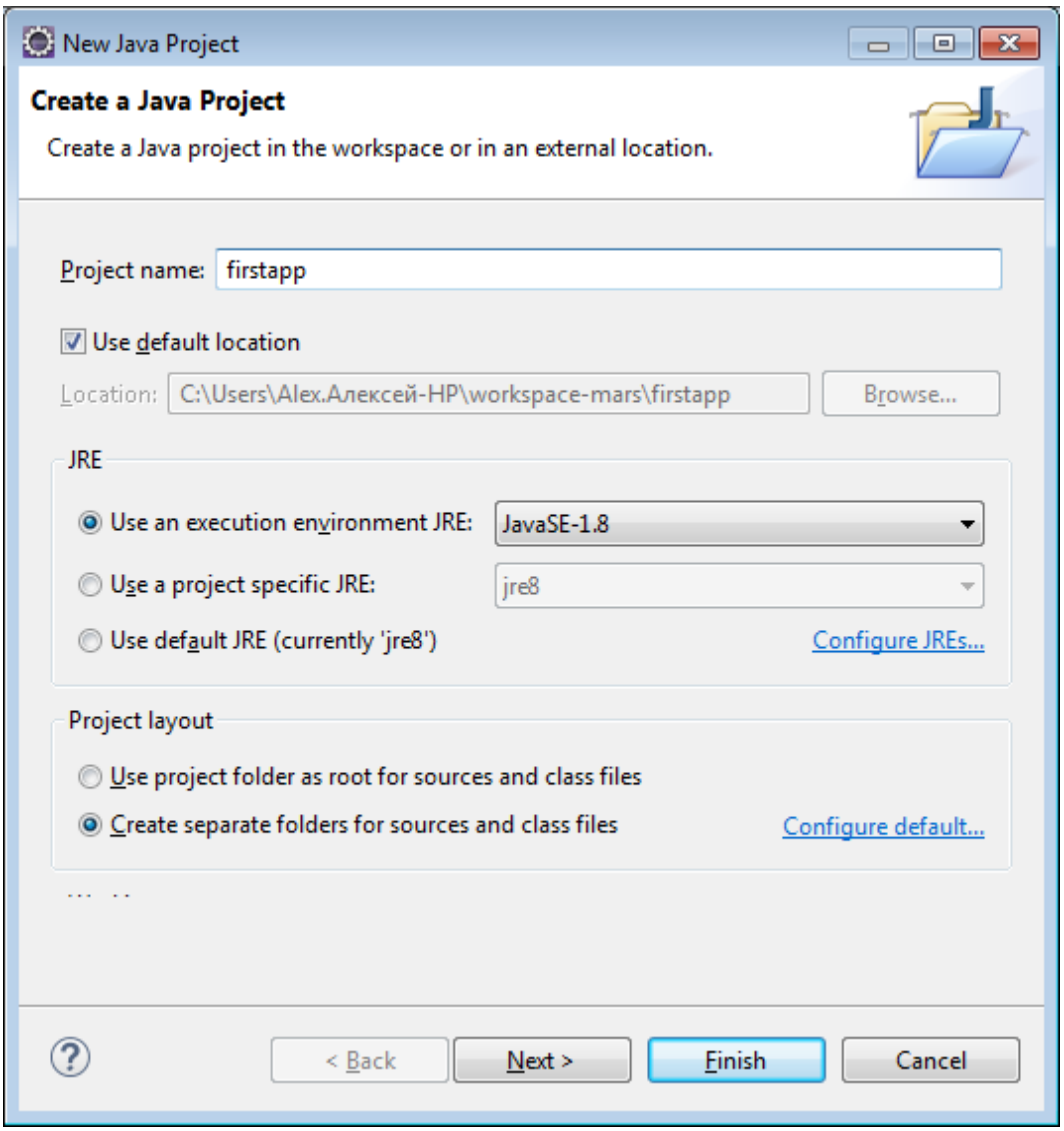
Як уже зазначалося, для професійного програмування на Java використовуються IDE, подібні до Eclipse або NetBeans. Для прикладу візьмемо Eclipse та створимо з його допомогою просте застосування, яке виводить на екран `Hello, world`.

Почнемо зі створення проекту. Запустимо Eclipse. Середовище має мати приблизно такий вигляд:



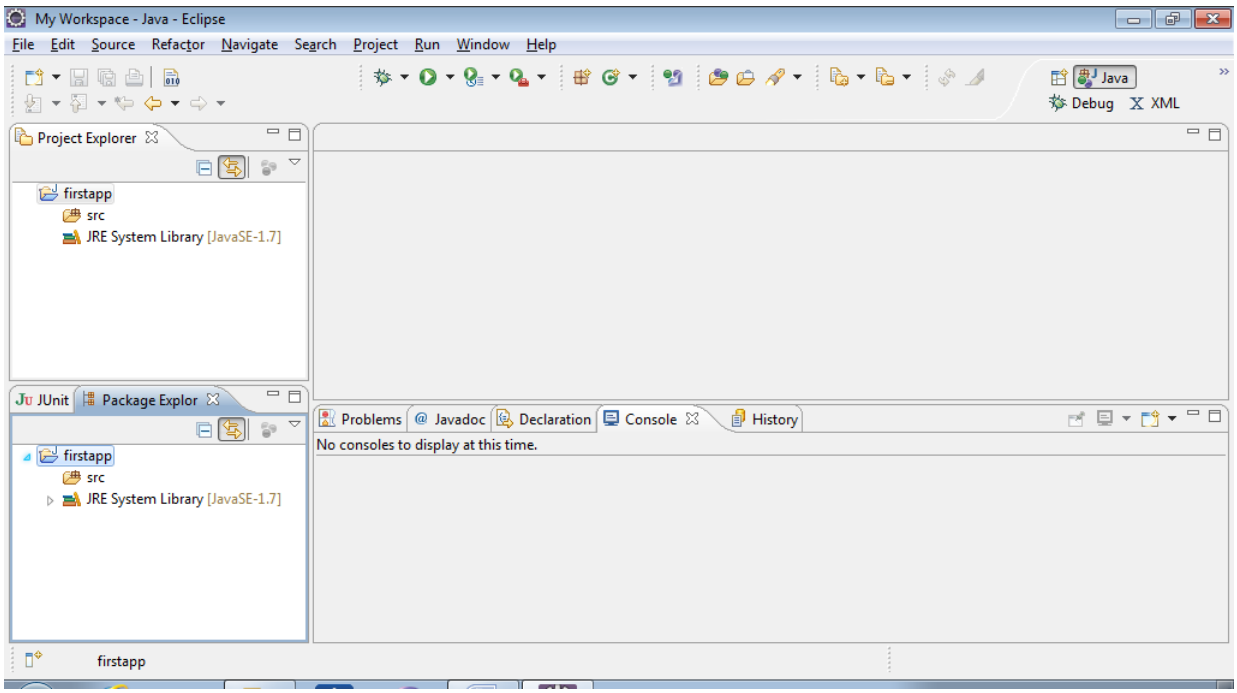
Можна вибирати між різними ***перспективами***, тобто варіантами інтерфейсу.

Виберемо пункт меню File=>New=>JavaProject. Повинно з’явитися вікно, схоже на наступне:



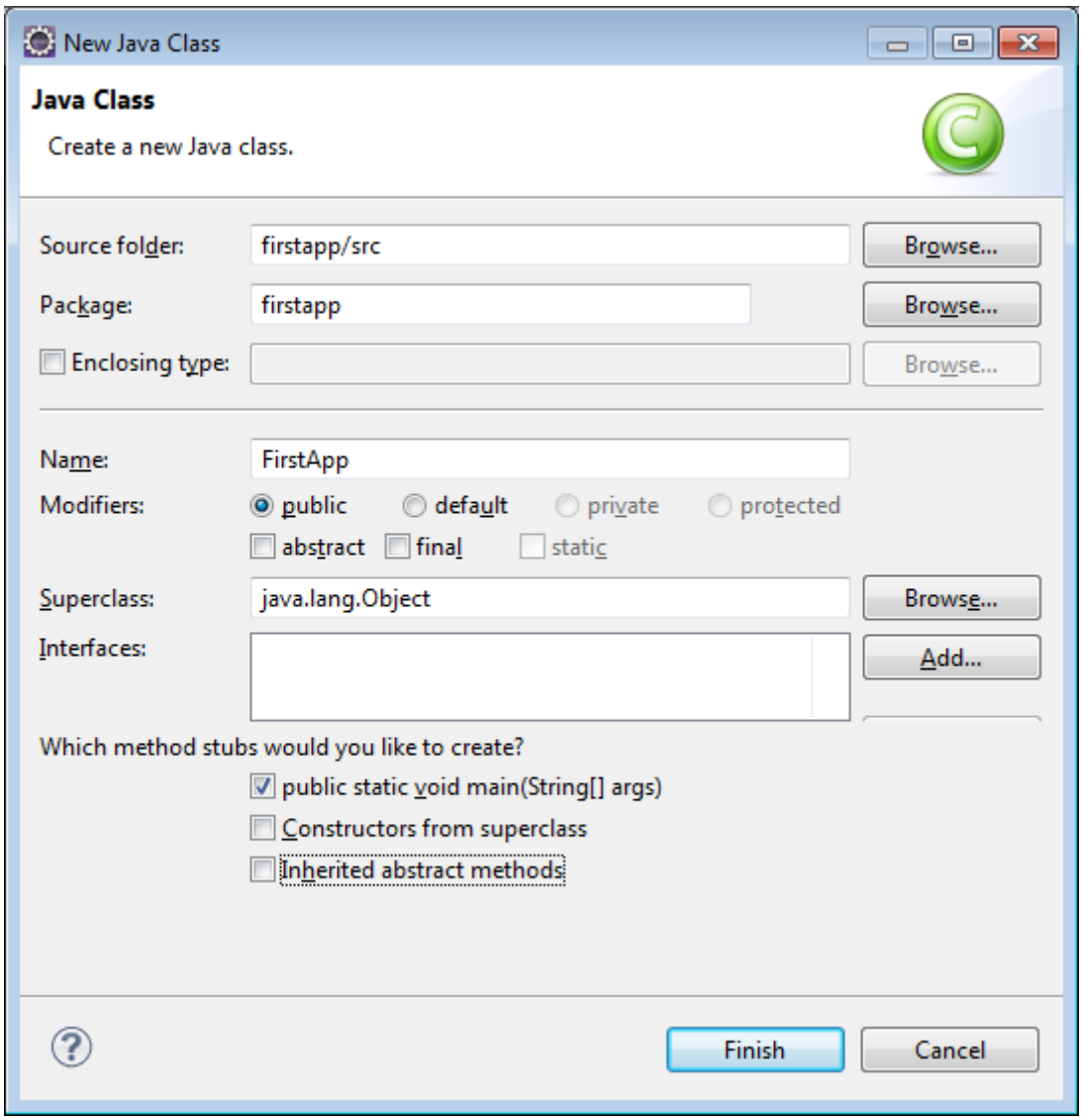
Даємо проекту назву (наприклад, firstapp); вибираємо потрібне JRE. Можна зразу ж підключити додаткові jar-файли та бібліотеки, які можуть використовуватися у проекті. Не будемо цього робити і натиснемо на Finish.

Після цього ми повинні побачити вікно, подібне до наступного:



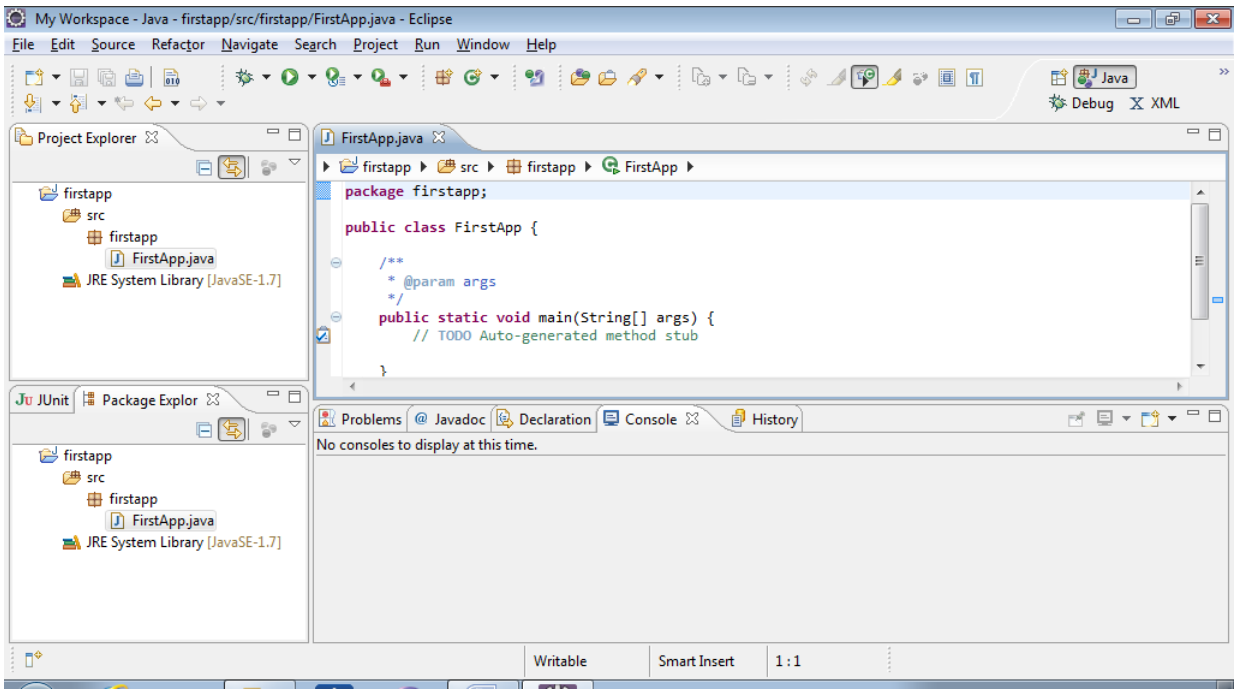
Далі потрібно створити пакет (це можна зробити за допомогою контекстного меню, якщо клацнути правою кнопкою миші на назві проекту). Назвемо його так само, як і проект – first.

Далі створюємо в цьому пакеті новий клас. У відповідній формі пишемо назву класу (нехай це буде FirstApp). Оскільки це головний клас застосування, в якому повинен бути метод main, ставимо відповідну галочку:



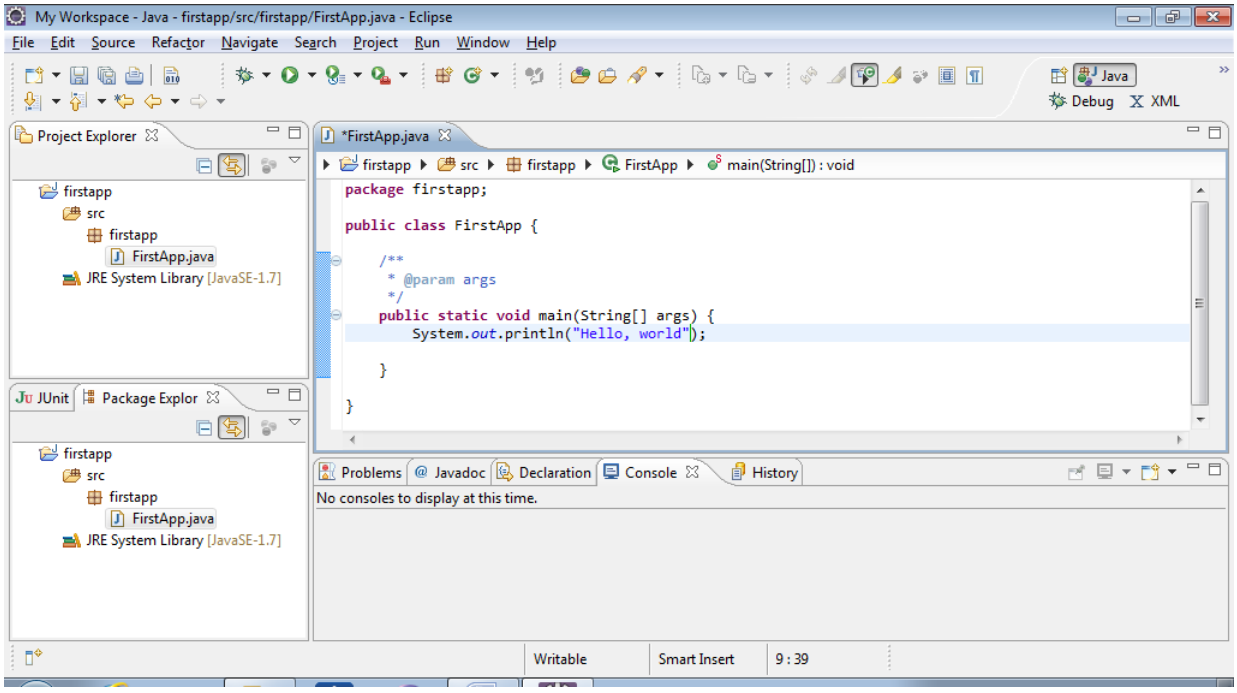
Зверніть увагу на те, що клас має бути в пакеті; створювати класи за межами пакетів дуже не рекомендується.

Ми бачимо заготовку програми, створену майстром Eclipse:

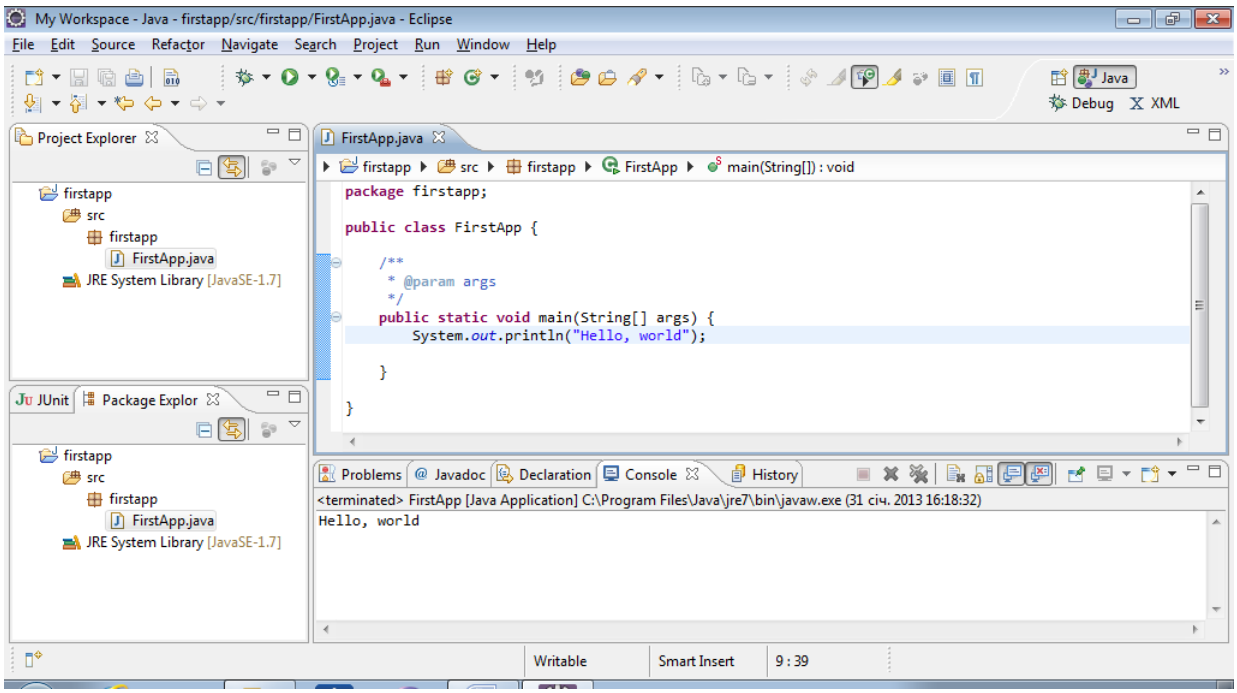


Набираємо текст програми. При цьому зверніть увагу на підказки Eclipse, спрямовані на полегшення та пришвидшення набору.

В результаті отримаємо:



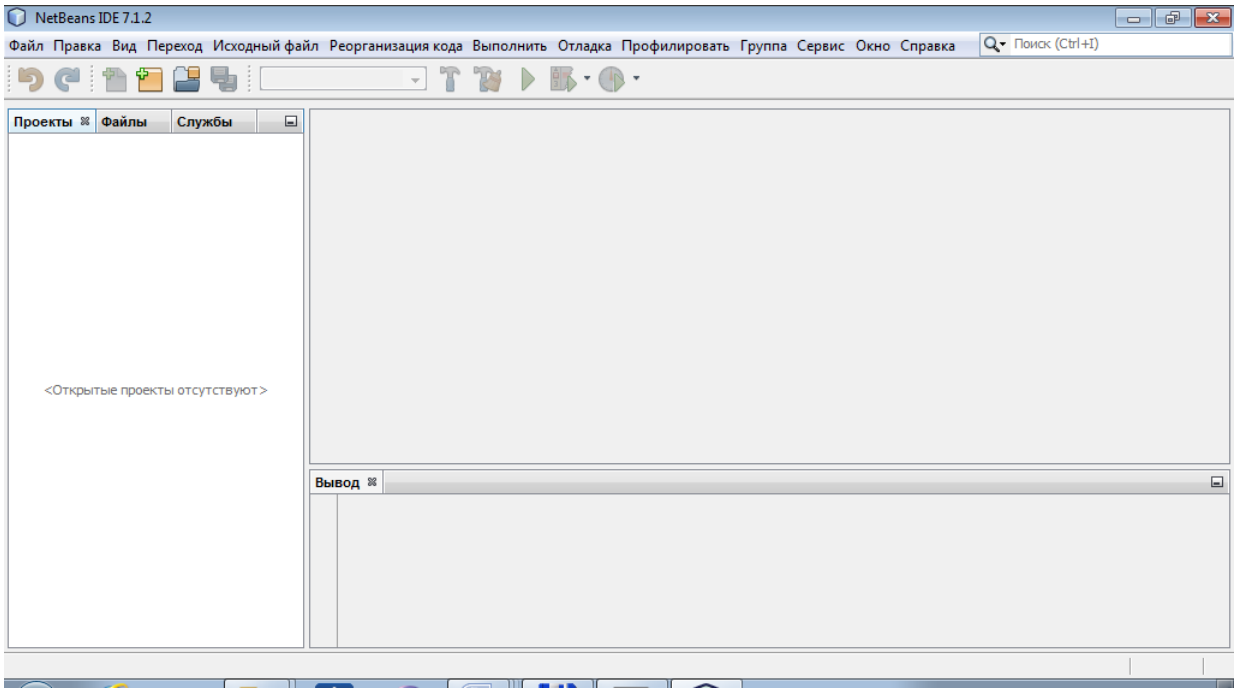
Запускаємо програму на виконання – або натиснувши мишею на зелений трикутник, або за допомогою пункту меню Run. На вкладці Console ми побачимо результат роботи програми:


My Workspace - Java - firstapp/src/firstapp/FirstApp.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Debug
XML
Project Explorer
firstapp
src
firstapp
FirstApp.java
JRE System Library [JavaSE-1.7]
JUnit
Package Explor
FirstApp.java
firstapp > src > firstapp > FirstApp > main(String[]) : void
package firstapp;
public class FirstApp {
/**
* @param args
*/
public static void main(String[] args) {
System.out.println("Hello, world");
}
}
Problems
Javadoc
Declaration
Console
History
<terminated> FirstApp [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (31 січ. 2013 16:18:32)
Hello, world
Writable
Smart Insert
9 : 39

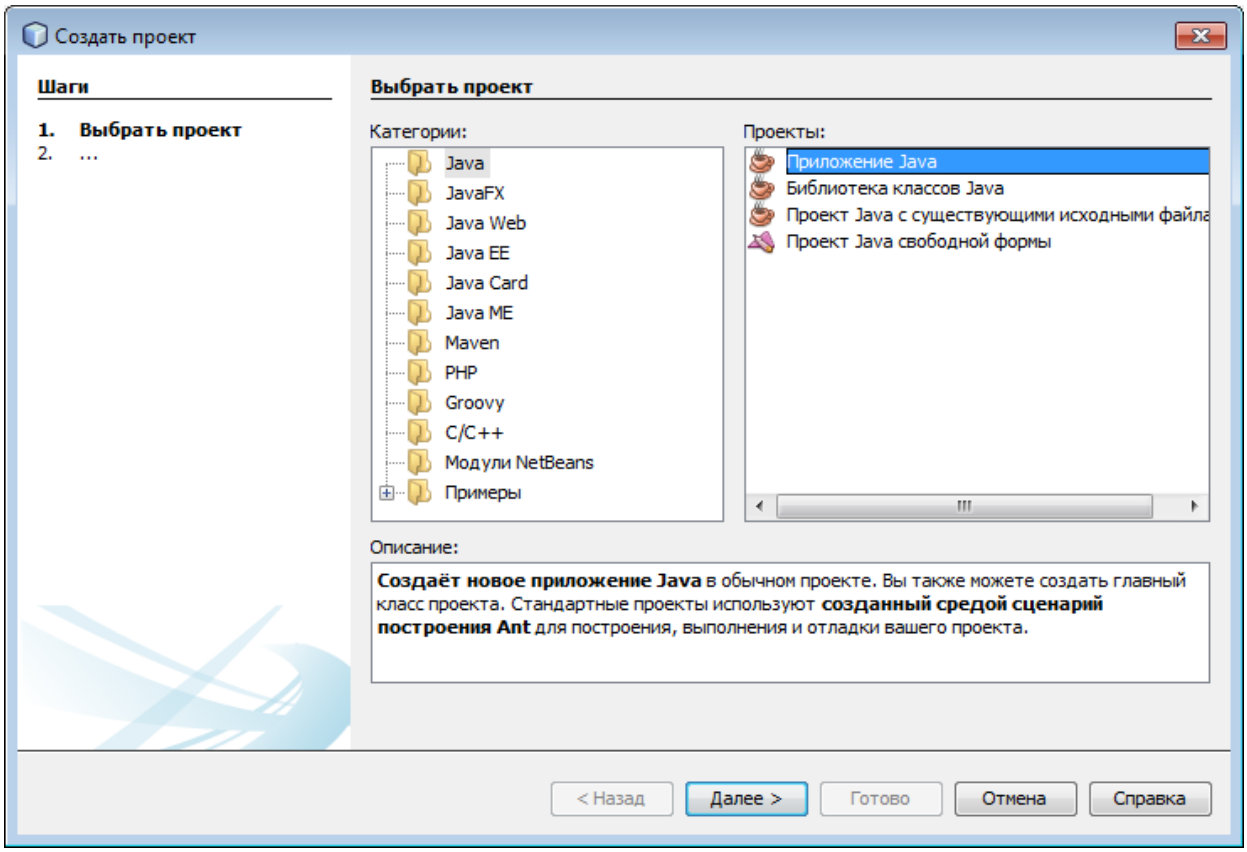
2.1.8. Створення Java-застосування засобами NetBeans

Тепер створимо застосування, яке виводить на екран Hello, world, за допомогою IDE NetBeans.

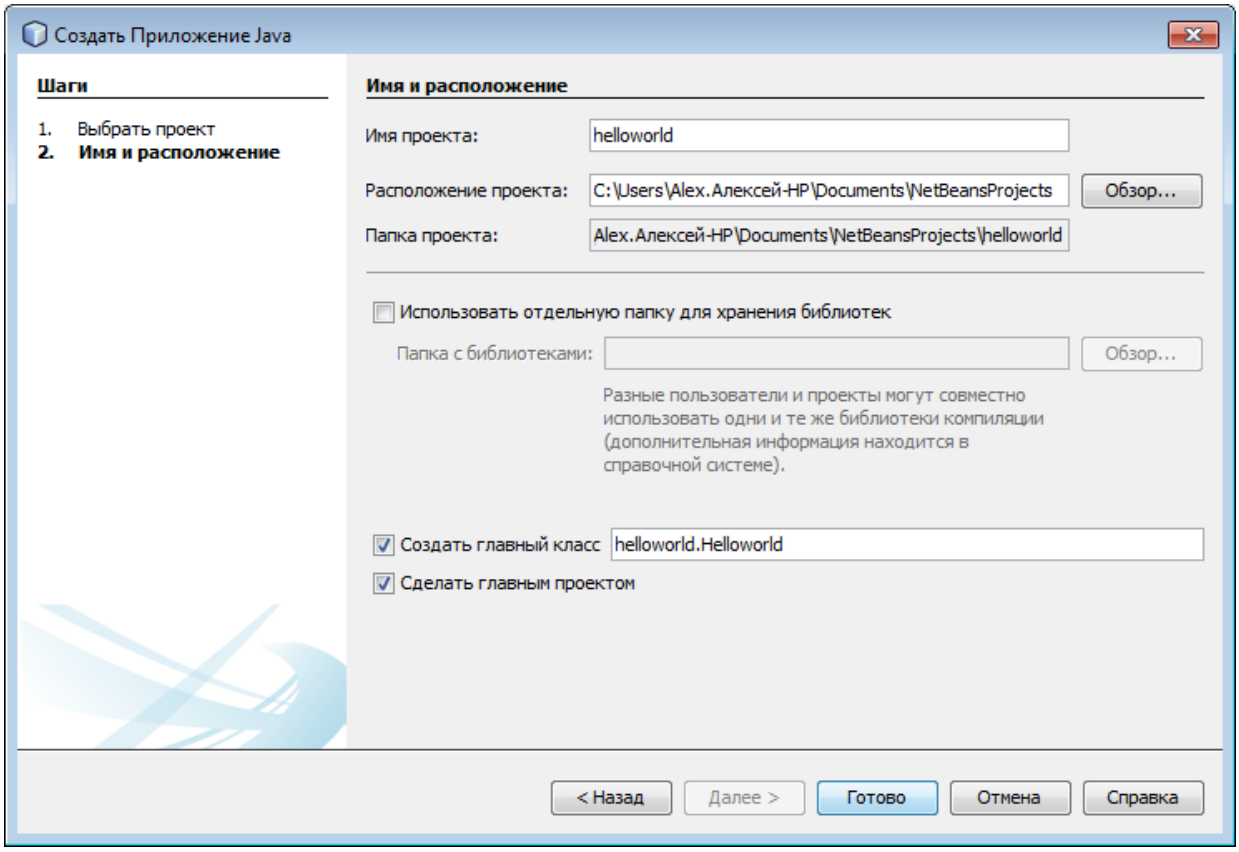
Запустимо NetBeans. Середовище може мати приблизно такий вигляд:



Почнемо створення нового проекту (меню Файл=>Создать проект…). Відкриється діалогове вікно:

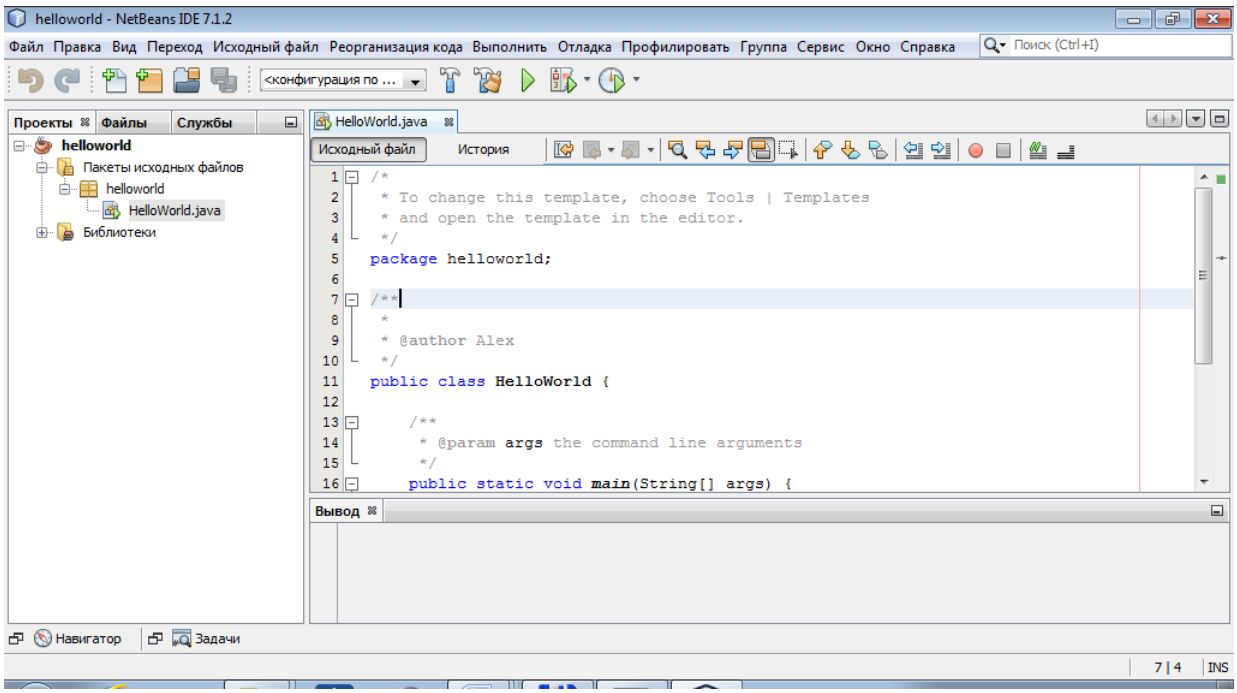


В категорії “Java” виберемо тип проекту: “Приложение Java”. Натиснемо “Далее”. З’явиться вікно:



Тут ми маємо вказати назву проекту (нехай це буде helloworld). NetBeans запропонує зразу створити відповідний основний клас. Можна було б з цим погодитися, але запропонована назва класу Helloworld не дуже відповідає стандартним домовленостям Java Code Conventions щодо імен класів. Тому змінимо її на HelloWorld і натиснемо на кнопку “Готово”.

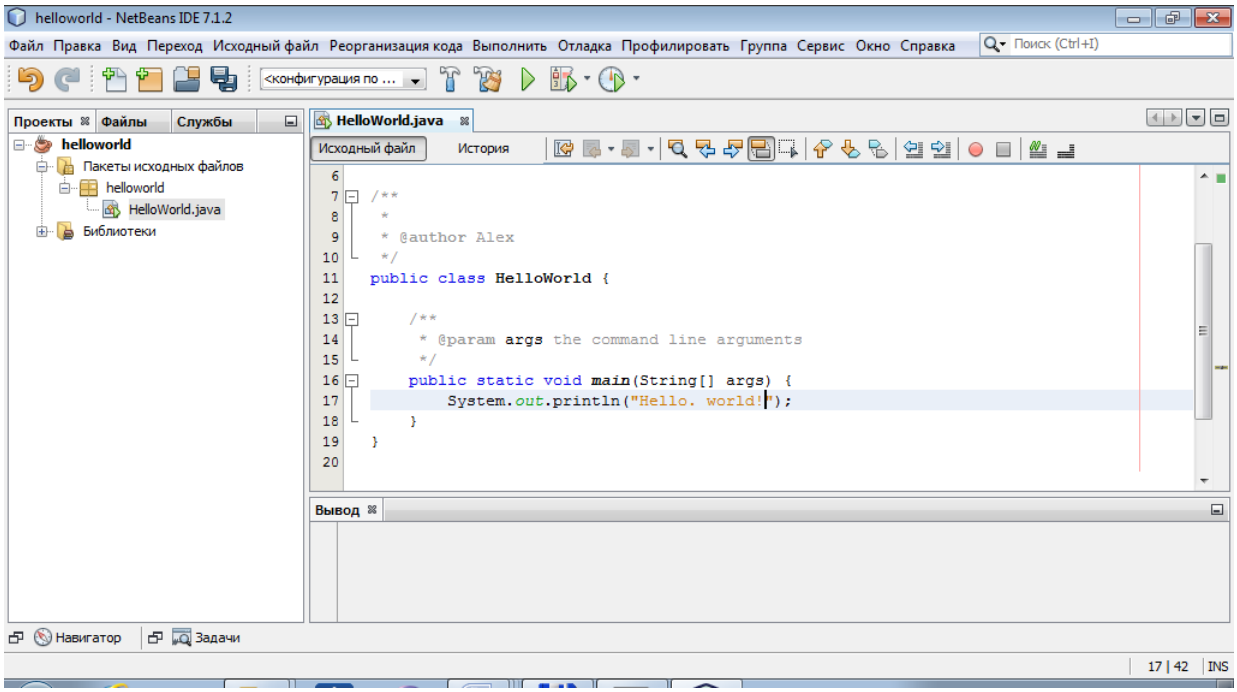
Бачимо заготовку, створену майстром NetBeans:



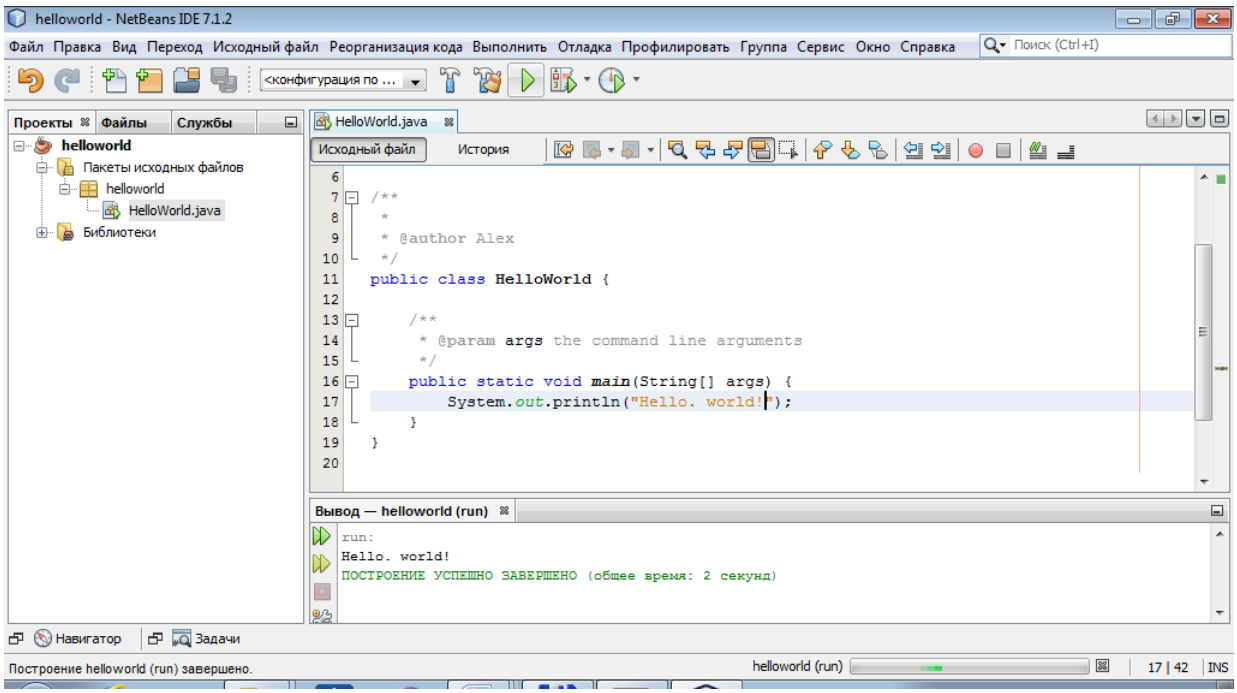
Відмітимо, що на відміну від Eclipse, NetBeans сам створив і пакет, і клас (а заодно і метод main).

Можемо писати код; при цьому не забуваємо про можливість використання клавіатурних комбінацій та підказок від середовища.

В результаті отримаємо:



Запускаємо програму (натискаємо на зелений трикутник або використовуємо меню). Отримуємо результати:


helloworld - NetBeans IDE 7.1.2
Файл Правка Вид Переход Исходный файл Реорганизация кода Выполнить Отладка Профилировать Группа Сервис Окно Справка
Поиск (Ctrl+I)
<конфигурация по ...
Проекты
Файлы
Службы
helloworld
Пакеты исходных файлов
helloworld
HelloWorld.java
Библиотеки
HelloWorld.java
Исходный файл
История
/**
*
* @author Alex
*/
public class HelloWorld {
/**
* @param args the command line arguments
*/
public static void main(String[] args) {
System.out.println("Hello. world!");
}
}
Вывод — helloworld (run)
run:
Hello. world!
ПОСТРОЕНИЕ УСПЕШНО ЗАВЕРШЕНО (общее время: 2 секунд)
Навигатор
Задачи
Построение helloworld (run) завершено.
helloworld (run)
17 | 42
INS

2.1.8 Знайомство з дизасемблером

Аналіз байт-коду, який утворюється внаслідок компіляції Java-програми, часто допомагає більш глибоко розібратися в деталях її роботи. Як основний інструмент для отримання мнемонічного запису байт-коду (дизасемблювання) розглядається утиліта `javap`, яка входить до складу JDK.

Для простоти візьмемо дуже просту програму яка нічого не виводить та не викликає ніяких інших функцій. Ця програма лише встановлює значення двох цілих змінних та підраховує їх суму. Зараз нас цікавить лише можливість подивитися байт-код та отримати початкове уявлення про нього.

Код програми має вигляд

```
public class MainClass {

public static void main (String[] args) {

int a=10;
int b=20;

int s=a+b;

}

}
```

Лістинг 2.1.8.1. Елементарна програма для дизасемблювання

Збережемо цей текст у файлі MainClass.java та відкомпілюємо його:

```
javac MainClass.java
```

Дизасемблюємо байт-код:

```
javap –c MainClass  > bytecode
```

В результаті переадресації дизасембльований код буде збережений у файлі `bytecode`. Він може мати вигляд:

```
Compiled from "MainClass.java"
public class MainClass {
  public MainClass();
    Code:
       0: aload_0
       1:   invokespecial   #1                  //   Method
java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: bipush        10
       2: istore_1
       3: bipush        20
       5: istore_2
```

```
         6: iload_1
         7: iload_2
         8: iadd
         9: istore_3
        10: return
}
```

Лістинг 2.1.8.2. Результат дизасемблювання програми з лістингу 2.1.8.1

Ми бачимо, що метод main() перетворився на послідовність команд Java-машини. Ці команди описані в специфікації JVMS, і для того, щоб розібратися в суті байт-коду, треба звернутися до цієї специфікації.

Розділ 2.2. Анатомія простого застосування

2.2.1. Основний клас

Як ми бачили, в будь-якому застосуванні повинен бути принаймні один клас, причому ім'я публічного класу повинно співпадати з іменем файлу. Наприклад, клас

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

Лістинг 2.2.1.1. Основний клас консольного застосування

має бути збережений у файлі MyClass.java.

2.2.2. Клас і файл: назви можуть не співпадати

Якщо клас не є публічним, співпадіння його імені з іменем файлу не вимагається. Цілком можна написати, наприклад, такий код:

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

Лістинг 2.2.2.1. Основний клас не є публічним

зберегти його у файлі з будь-якою назвою з розширенням .java – наприклад, manyvariants.java та відкомпілювати звичайним чином:

```
javac manyvariants.java
```

Але при цьому все одно створиться файл MyClass.class, і запустити програму, як і слід було очікувати, треба так:

```
java MyClass
```

Таким чином, файл з початковим кодом може мати вигляд:

```
class K1 {
//Якийсь код
}

class K2 {
//Якийсь код
}
…
```

На початку файлу може бути директива **package**, яка свідчить про включення класів до того чи іншого пакету. Крім того, можуть бути директиви **import**, які підключають інші пакети.

2.2.3. Метод main та його варіанти

Основним методом, ***точкою входу*** до програми є метод ***main(),*** який найчастіше має вигляд

```
public static void main(String[] args)
```

Саме цей метод викликається віртуальною машиною і виконується.

В принципі, метод ***main()*** є звичайним методом; зокрема він може бути перевантажений (тобто в класі може бути кілька методів main з різною сигнатурою). Але, незважаючи на це, викликатися буде один з них – той, який має наведену вище сигнатуру. Якщо ж віртуальна машина не знайде потрібного методу, виникне помилка часу виконання – так зване ***виключення.***

Метод ***main*** має бути описаний як ***public*** (загальнодоступний), оскільки в іншому разі віртуальна машина не матиме доступу до нього і, відповідно, не зможе його викликати.

Метод ***main*** має бути описаний як ***static*** (статичний), оскільки цей метод викликається тоді, коли не існує жодного екземпляру класу first.

Оскільки метод нічого не повертає, він повинен бути описаний як ***void*** (зверніть увагу це є одна з "тихих", але важливих відмінностей від C++, стандарт якого вимагає, щоб метод `main()` повертав ***int***.

Модифікатори ***static*** та ***public*** можна переставити місцями.

Для опису параметрів методу main() можна також використати форму функції зі змінною кількістю аргументів (ця можливість існує починаючи з Java 5). Тобто замість

```
public static void main (String[] params)
```

можна написати

```
public static void main (String ... params)
```

Розглянемо невелике контрольне запитання.

Дано кілька варіантів написання методу main. Скільки з них будуть компілюватися:

```
а) public static void main(String...a)
б) public static void main(String.* a)
в) public static void main(String... a)
г) public static void main(String[]... a)
д) public static void main(String...[] a)
```

Відповідь – 3; це варіанти а), в), г). Тут є невеличка пастка: запитання не про те, скільки варіантів можуть бути **виконані**, а про те, скільки будуть **компілюватися**. Варіант г) не є законним з точки зору віртуальної машини, але нормально пропускається компілятором – цей метод приймає змінну кількість масивів, елементами яких є рядки.

2.2.4. Варіанти Hello, world

Повернемося до програми `Hello, world,` яка виводить на екран деякий текст.

```
public class HelloWorld {

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

```
        System.out.println("Hello, world!");

    }

}
```

Лістинг 2.2.4.1. Найпростіший варіант `Hello, world:` виведення літералу

Проілюструємо деякі можливі варіанти цієї програми.

По-перше, у виведенні фігурує ***літерал*** – рядкове значення `Hello, world`. Крім найпростіших випадків, така практика не є особливо рекомендованою, оскільки зловживання літеральними константами в тексті програми знижує її гнучкість і підвищує ризик помилок. Краще надати значення рядка деякій змінній і надалі використовувати цю змінну:

```
public class HelloWorld {


    public static void main(String[] args) {
        String s = "Hello, world!";
        System.out.println(s);

    }

}
```

Лістинг 2.2.4.2. Виведення рядкової змінної

Ще краще (оскільки рядок, з яким ми маємо справу), змінюватися не буде, оголосити `s` як константу. Для цього використаємо ключове слово **`final`**:

```
public class HelloWorld {


    public static void main(String[] args) {
        final String s = "Hello, world!";
        System.out.println(s);

    }

}
```

Лістинг 2.2.2.3.. Виведення константи

Можна також винести змінну `s` за межі методу `main` та зробити її **полем** класу. Але наступний варіант не проходить і призводить до помилки компіляції:

```
public class HelloWorld {

 final String s = "Hello, world!";


 public static void main(String[] args) {

      System.out.println(s);

 }

}
```

Лістинг 2.2.2.4. Виведення поля класу: неправильний варіант

Справа в тому, що метод `main` є **статичним** і може звертатися лише до статичних полів класу. Потрібно зробити поле `s` статичним:

```
public class HelloWorld {

 final static String s = "Hello, world!";


 public static void main(String[] args) {

      System.out.println(s);

 }

}
```

Лістинг 2.2.4.5. Виведення статичного поля: правильно

2.2.5. Аргументи командного рядка

Важливе значення має параметр методу ***main().*** Він є масивом рядків (екземплярів класу **String**), і це дані, які передаються програмі ззовні як ***аргументи командного рядка***. Програма може звертатися до них як до елементів звичайного масиву.

Для прикладу напишемо програму, яка визначає довжини рядків, переданих як аргументи командного рядка.

Створимо файл **StringLength.java** з наступним вмістом:

```
public class StringLength {

public static void main (String[] params) {
```

```
    int paramcount = params.length;
    for (int i=0; i<paramcount; i++) {
    System.out.println("Length      of      "+params[i]+"      is
"+params[i].length());
    }

    }
    }
```

Лістинг 2.2.5.1. Використання аргументів командного рядка

Ми використали “класичну” форму перебору елементів масиву params (щоб підкреслити, що це звичайний масив, ми змінили його традиційну назву). За допомогою звернення

```
    int paramcount = params.length;
```

ми отримуємо довжину цього масиву (змістовно – кількість рядків, які нас цікавлять). Далі перебираємо елементи цього масиву за допомогою циклу for (нумерація елементів масиву, так само як у мові Сі, починається з 0):

```
    for (int i=0; i<paramcount; i++) {
    System.out.println("Length      of      "+params[i]+"      is
"+params[i].length());
    }
```

Існує ще один спосіб перебору елементів масиву – модифікований цикл for, який з’явився у Java 5 (часто його називають циклом foreach, оскільки саме так подібний цикл називається в ряді інших мов програмування). Детальніше про нього буде йтися далі.

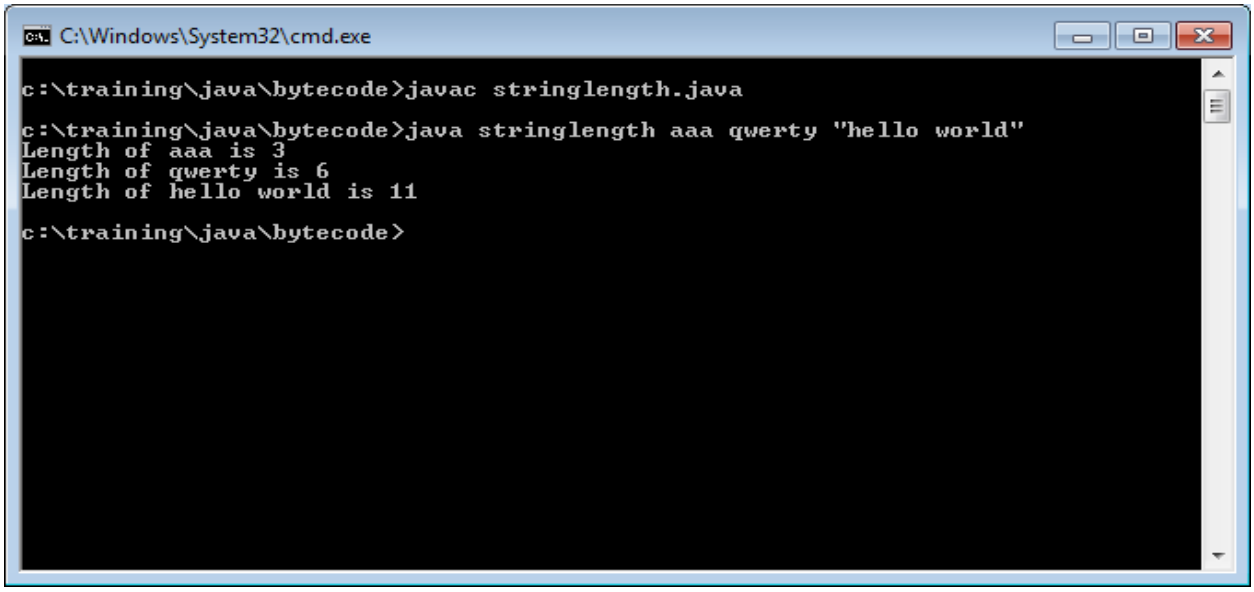
Відкомпілюємо цю програму звичайним чином:

```
javac StringLength.java
```

При запуску слід передати програмі якісь дані. Наприклад, якщо нас цікавлять довжини рядків aaa, qwerty та hello world, програму слід запустити наступним чином:

```
java StringLength aaa qwerty “hello world”
```

Виведення буде мати вигляд:



Звісно, якість програми суттєво зросте, якщо вона буде адекватно реагувати на можливі помилки. Так, у нашому прикладі програма мала б перевірити, чи були їй передані якісь дані, і у випадку, якщо ні – вивести користувачеві відповідні інструкції.

Така модифікація могла б мати вигляд:

```
public class StringLength {

public static void main (String[] params) {


int paramcount = params.length;

if (paramcount == 0) {
System.out.println("Usage: java stringlength str1 ...");
System.exit(0);
}
for (int i=0; i<paramcount; i++) {
System.out.println("Length      of      "+params[i]+"      is
"+params[i].length());
}

}
}
```

Лістинг 2.2.5.2. Перевірка правильності передачі аргументів командного рядка

Виклик `System.exit(0)`, тобто виклик методу ***exit*** класу **System** завершує роботу програми. Замість цього в даному випадку можна було б використати інструкцію ***return***.

Звичайно, наведений приклад є досить ілюстративним. У більш типових ситуаціях аргументи командного рядка використовуються для того, щоб вказати, з якими файлами, мережними ресурсами і т.п. буде працювати програма.

2.2.6. Програма без методу main

Чи можна написати консольне застосування, яке нормально працює, але не використовує методу main? Йдеться не про можливість компіляції (ми уже відмічали, що компілятор не вимагає його наявності; для нього це такий самий метод, як і будь-який інший), а саме про виконання. Така можливість ніколи не була рекомендованою, але вона існувала до Java 6 включно.

В основі лежить механізм ***статичної ініціалізації***. Статична ініціалізація заслуговує на окремий розгляд, але зараз можна сказати, що в класі може бути секція статичної ініціалізації, яка виконується при завантаженні класу - до створення його екземплярів і виклику будь-яких методів.

Програма, яка не використовує main, могла мати вигляд:

```
public class WithoutMain {

static {

System.out.println ("Main is deprecated!");
System.exit(0);
}


}
```

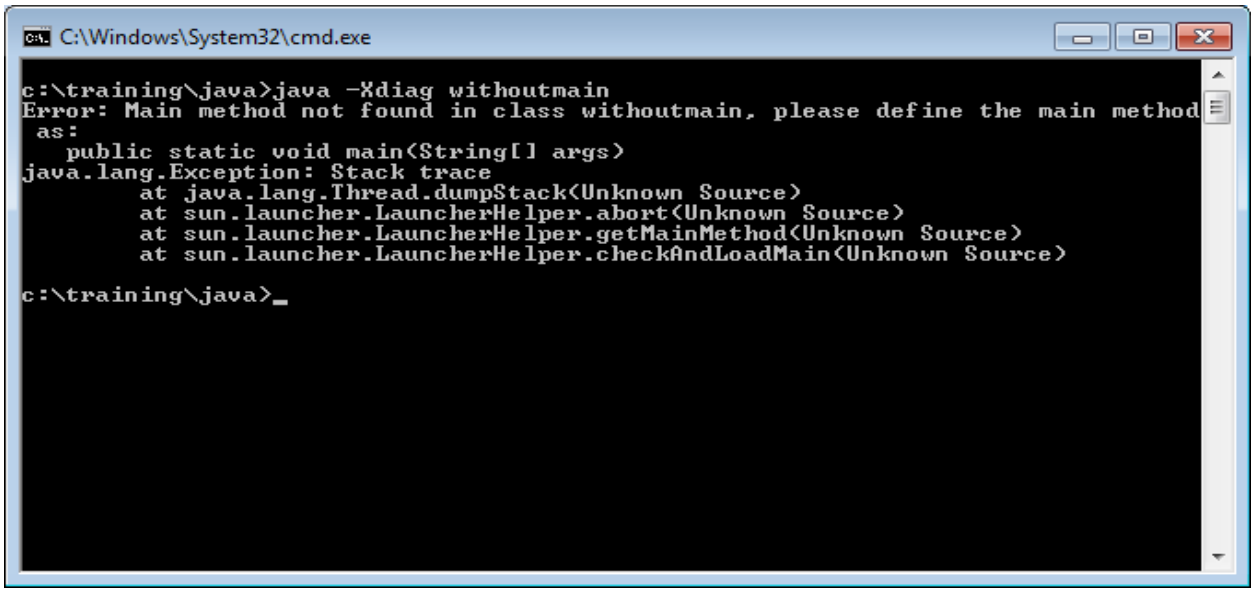
Лістинг 2.2.6.1. Програма без методу main

Ця програма нормально компілювалася, виводила повідомлення

```
Main is deprecated!
```

та припиняла роботу через виклик **System.exit(0)**.

Так було до Java 6 включно. Але в Java 7 виникає помилка часу виконання - віртуальна машина повідомляє про відсутність методу main:



2.2.7. Поняття про пакети

У першому наближенні ***пакети*** – це певним чином організовані групи класів. Кожний клас може бути включений до певного пакету. Наприклад, для включення класу Kl до пакету pak на початку файлу з початковим кодом має міститися інструкція package:

```
package pak;
class Kl {
…
}
```

Повне ім’я класу складається власне з імені класу та імені пакету. Так, у нашому прикладі повне ім’я класу Kl. який міститься в пакеті pak, буде pak.Kl.

Ім’я пакету може бути і більш довгим. Наприклад, клас String міститься в пакеті java.lang, і тому його повне ім’я – java.lang.String.

Зверніть увагу – **клас ідентифікується повним ім’ям**! Так, казати про властивості класу String або будь-якого іншого класу не має сенсу, поки не буде однозначно визначено, до якого саме пакету він належить. Коли йдеться про клас String, практично завжди мається на увазі “стандартний” java.lang.String – але ж це зовсім не обов’язково. Цілком можливо створити власний пакет, в якому буде свій клас String.

Якщо в програмі використовується клас з якогось пакету, потрібно або вказувати його повне ім’я, або імпортувати його за допомогою директиви import:

import pak.Kl;

Можна імпортувати всі класи пакету:

import pak.*;

У пакеті java.lang зібрані найбільш важливі класи, і він імпортується автоматично.

Розділ 2.3. Операції введення-виведення

2.3.1. Загальне поняття про введення-виведення

Програма повинна мати можливість ввести дані, з якими вона працює, з зовнішнього джерела (у найпростішому випадку – з клавіатури) та вивести результати своєї роботи (у найпростішому випадку – на екран). Такі засоби є і в мові Java.

Концептуальною основою системи введення-виведення в Java є поняття потоку. ***Потік*** – це логічна абстракція, яка описує передачу даних від джерела до приймача. Джерела і приймачі даних можуть бути різних типів; дисципліни роботи з ними теж можуть бути різними. Відповідно до цього, Java має велику кількість класів, які описують різні типи потоків.

Для більш чіткого розуміння роботи з потоками як об'єктами відповідних класів потрібно мати уявлення про основні принципи об'єктно-орієнтованого програмування. Але, оскільки введення-виведення має бути присутнім практично в будь-якій програмі, тут ми наведемо найбільш типові операції, не вдаючись до детальних пояснень. Поки що будемо сприймати ці операції введення та виведення як дещо наперед задане.

2.3.2. Стандартні потоки введення-виведення.

Java надає можливість працювати з трьома стандартними потоками (полями класу **java.lang.System**):

- **System.in** – стандартне введення;
- **System.out** – стандартне виведення;
- **System.err** – стандартна помилка; у цей потік спрямовуються повідомлення про помилки, які виникають під час виконання програми.

Потік **System.out** надає зручні засоби для виведення даних різних типів (ми бачили це в попередніх розділах). Але можливості потоку System.in для введення даних значно бідніші – фактично його можна використовувати хіба що для введення окремих символів.

2.3.3. Введення символів

Спробуємо ввести символ з клавіатури:

```
package demobasicio;

import java.io.IOException;

public class DemoBasicIO {

 public static void main(String[] args) {
      System.out.println("Enter a symbol");
      int c = System.in.read();
      System.out.println("You entered symbol "+c);

 }

}
```

Лістинг 2.3.3.1. Введення символу: характерна помилка компіляції

В чому справа? Чому компілятор видає повідомлення про синтаксичну помилку?

Вважається, що при виконанні операцій введення щось може піти не так, і може виникнути помилкова ситуація – виключення; у даному випадку – виключення **IOException**. Це виключення є ***контрольованим***: компілятор слідкує за можливістю його

виникнення. Необхідно або перехопити та обробити це виключення за допомогою інструкції try-catch-finally (див.далі), або явно повідомити компілятору про те, що метод не обробляє це виключення, а ***викидає*** його назовні, за допомогою ключового слова throws, яке додається до опису методу:

```
package demobasicio;

import java.io.IOException;

public class DemoBasicIO {

 public static void main(String[] args) throws IOException {
      System.out.println("Enter a symbol");
      int c = System.in.read();
      System.out.println("You entered symbol "+c);

 }

}
```

Лістинг 2.3.3.2. Введення символу і не зовсім правильне його виведення

Тепер програма компілюється і виконується, але результат є дещо несподіваним. Якщо ми введемо символ `'A'`, на екран виводиться число `65`.

Справа в тому, що символи зберігаються в пам'яті комп'ютера як їх коди, тобто як цілі значення, і результатом методу `System.in.read()` є ціле число. Тому змінна c (результат введення) має цілий тип **int**, і при її виведенні ми бачимо не сам символ, а його код. Для того, щоб все-таки був виведений символ, потрібно здійснити ***приведення*** до символьного типу **char**:

```
package demobasicio;

import java.io.IOException;

public class DemoBasicIO {

 public static void main(String[] args) throws IOException {
      System.out.println("Enter a symbol");
      int c = System.in.read();
      System.out.println("You entered symbol "+ (char) c);

 }

}
```

Лістинг 2.3.3.3. Введення та виведення символу: виправлений варіант

Тепер на при введенні символу `'A'` на екран виводиться саме цей символ.

Але потік **System.in** не надає зручних засобів ні для введення рядків, ні для введення, наприклад, чисел. Тому для цього треба використовувати інші класи потоків.

2.3.4. Класична схема введення рядків

До Java 5 найбільш вживаний спосіб введення рядків, а також даних інших типів полягав у використанні потоку **BufferedReader** з пакету **java.io**. Загальну схему введення на основі цього класу можна охарактеризувати наступним чином.

Спочатку створюється потік введення як екземпляр класу BufferedReader :

```
BufferedReader br=new BufferedReader(new InputStreamReader
(System.in));
```

Далі за допомогою методу readLine можна зчитати черговий рядок (введення закінчується, коли користувач натискає на Enter:

```
String St=br.readLine();
```

Наведемо ілюстративну програму, яка зчитує з клавіатури ім’я користувача та виводить йому привітання.

```
import java.io.*;

public class UsingBufferedReader {

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        System.out.println("Enter your name");
        String name=br.readLine();
        System.out.println("Hello, "+name);
    }
}
```

Лістинг 2.3.4.1. Введення рядків за допомогою класу BufferedReader

2.3.5. Використання конкатенації для виведення

Зверніть увагу на те, як у лістингу 2.3.4.1 формується привітання, що виводиться на екран:

```
System.out.println("Hello, "+name);
```

Метод println() приймає лише один аргумент, який має бути рядком. Тому для формування потрібного виведення дуже типовим було і залишається використання ***конкатенації*** (зчеплення) рядків (оператор +). До появи форматованого виведення (див п.2.3.8) цей спосіб формування виведення був фактично основним.

У даному випадку для формування привітання здійснюється конкатенація двох рядків: незмінної константи `"Hello, "` та змінної `name`, яка вводиться під час роботи програми.

2.3.6. Введення даних інших типів

Метод readLine() класу java.io.BufferedReader дозволяє вводити лише рядки. Тому у випадку, якщо треба ввести дані інших типів (наприклад, числа), можна застосувати наступну схему: ввести дані як рядок, а потім здійснити відповідне перетворення.

Розглянемо простий приклад: програму, яка вводить з клавіатури ціле число та виводить на екран його квадрат. Основна частина програми може мати вигляд

```
    BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
    System.out.println("Enter the integer value");
    String strNumber=br.readLine();
    int number = Integer.parseInt(strNumber);
    System.out.println("The square of "+number+" is
"+number*number);
```

Лістинг 2.3.6.1. Введення чисел

Як і раніше, тут створюється потік br класу BufferedReader; виклик

```
String strNumber=br.readLine();
```

забезпечує введення потрібного числа як рядка.

Перетворення рядка на ціле число здійснюється за допомогою виклику

```
int number = Integer.parseInt(strNumber);
```

Тут використовується метод **parseInt** классу **Integer** – так званої ***об'єктної оболонки*** для цілих чисел. Аналогічним чином можна вводити дані інших простих типів.

Тут заради простоти ми вважаємо, що число, яке вводиться користувачем, завідомо коректне, тобто введений рядок завідомо відповідає форматові цілого числа. Легко зрозуміти, що насправді така коректність зовсім не гарантується. У випадку неправильного введення можуть виникати помилки, які мають належним чином оброблятися.

2.3.7. Бідьш нові можливості для введення та виведення

Починаючи з Java 5, з'явилися більш зручні засоби введення-виведення.

По-перше, з'явився клас **Scanner** з пакету java.util, методи якого забезпечують можливість введення даних різних типів: рядків, цілих чисел та ін.

По-друге, з'явився метод ***printf(),*** добре знайомий програмістам на C/C++. Він дозволяє відформатувати те, що виводиться, за рахунок задання у форматному рядку специфікаторів формату.

2.3.8. Форматоване виведення

Наведемо приклад використання функції printf для форматування виведення.

У результаті виконання фрагменту коду

```
double r = 10./3.;
System.out.println("Result is "+r);
```

буде виведено

```
Result is 3.3333333333333335
```

Зрозуміло, що таке виведення є важким для сприйняття людиною. Форматування, зокрема встановлення кількості десяткових знаків, які виводяться, дозволить подати результат у більш приємному вигляді.

Код може мати вигляд

```
double r = 10./3.;
System.out.printf("Result is %4.2f\n",r);
```

Він виведе

```
Result is 3,33
```

Тут на виведення результату відводиться 4 позиції; на дробову частину – 2 позиції.

2.3.9. Використання класу Scanner

У Java 5 з'явився клас **java.util.Scanner**, який надає зручні засоби для введення даних. Спочатку створюється екземпляр цього класу:

```
Scanner sc = new Scanner(System.in);
```

після чого можна використовувати методи цього класу для введення даних різних типів. Наведемо кілька прикладів.

Спробуємо ввести ім'я користувача як рядок за допомогою методу **nextLine()** та вивести привітання цьому користувачеві:

```
package demoscanner;

import java.util.Scanner;

public class DemoScanner {

 public static void main(String[] args) {

      Scanner sc = new Scanner(System.in);
      System.out.println("Enter your name");
      String st = sc.nextLine();
      System.out.println("Hello, "+st);

 }

}
```

Лістинг 2.3.9.1. Клас Scanner: введення рядка

Введемо з клавіатури ціле число за допомогою функції **nextInt()** та виведемо на екран квадрат цього числа:

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

      Scanner sc = new Scanner(System.in);
      System.out.println("Enter the number");
      int n = sc.nextInt();
      System.out.println("The square of "+n+" is "+n*n);

 }
```

Лістинг 2.1.9.2. Клас Scanner: введення цілого числа

Як бачимо, вводити числа у порівнянні з тим, що було до появи класу Scanner, стало набагато зручніше.

Спробуємо ввести рядок, який складається з кількох слів, та розділити його на окремі слова. Скористаємося тим, що метод **next()** зчитує наступне слово (хоча слід зазначити, що з метою розщеплення рядків частише використовуються інші засоби). Спочатку розглянемо більш простий випадок, коли ми заздалегідь знаємо кількість слів у рядку (наприклад, 3):

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

      Scanner sc = new Scanner(System.in);
      System.out.println("Enter the string");
      System.out.println(sc.next());
      System.out.println(sc.next());
      System.out.println(sc.next());

 }
```

Лістинг 2.3.9.3. Використання Scanner для розщеплення рядка; кількість слів відома заздалегідь

Якщо ввести речення `Life is wonderful`, програма виведе

```
Life
is
wonderful
```

Якщо ж кількість слів надто велика або взагалі невідома, слід використовувати ***цикли***, наприклад:

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

      Scanner sc = new Scanner(System.in);
      System.out.println("Enter the string");
      while (sc.hasNext()) {
           System.out.println(sc.next());
      }
```

Лістинг 2.3.9.4. Використання Scanner для розщеплення рядка; кількість слів невідома

Вводимо речення `To be or not to be`; програма виводить

```
To
be
or
not
to
be
```

2.3.10. Деякі особливості та сюрпризи класу Scanner

Застосування класу Scanner пов’язано з рядом особливостей та тонкощів. Проілюструємо деякі з них.

Розглянемо програму

```
package scannersurprise;

import java.util.Scanner;

public class ScannerSurprise {

                public static void main(String[] args) {
Scanner scan = new Scanner(System.in);

        System.out.println("Enter your number");
        int n = scan.nextInt();
        System.out.println("Enter first string");
        String first = scan.nextLine();
        System.out.println("Enter second string");
        String second = scan.nextLine();
        System.out.printf("Number was %d; strings were:%s and
%s\n", n, first, second);

                }

}
```

Лістинг 2.3.10.1. Неправильне введення цілих чисел та рядків

Ця програма працює неправильно: перескакує через введення першого рядка.
Діалог з користувачем може мати вигляд:

```
Enter your number
5
Enter first string
Enter second string
qwerty
Number was 5; strings were: and qwerty
```

Можна спробувати після введення числа додати операцію

```
scan.skip("\n");
```

(пропуск певного тексту, у даному випадку – символа нового рядка), Але і ця операція теж не є гарною. В ній жорстко прописаний роздільник рядків, і тому робота програми стає залежною від середовища.

Краще отримати роздільник з системних властивостей:

```
String ls = System.getProperty("line.separator");
        scan.skip(ls);
```

Але навіть у цьому випадку програма працює по-різному в різних середовищах, зокрема в NetBeans та Eclipse.

Для цієї задачі можна спробувати простіше рішення: після зчитування числа просто додатково зчитати наступний (пустий) рядок:

```
    Scanner scan = new Scanner(System.in);

            System.out.println("Enter your number");
            int n = scan.nextInt();
    scan.nextLine();

            System.out.println("Enter first string");
            String first = scan.nextLine();
            System.out.println("Enter second string");
            String second = scan.nextLine();
            System.out.printf("Number was %d; strings were:%s and
%s\n", n, first, second);
```

Лістинг 2.3.10.1. Введення цілих чисел та рядків: виправлений варіант

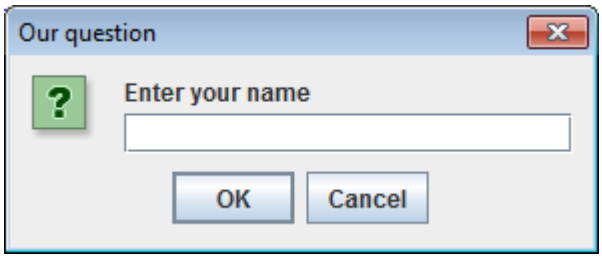
2.3.11. Використання діалогових вікон для введення введення

Інколи для введення застосовуються елементи графічного інтерфейсу. Для цього використовуються стандартні діалогові вікна, що надаються класом **JOptionPane** з пакету **javax.swing**. Така програма може мати вигляд:

```
package inputdialog;
import javax.swing.JOptionPane;

public class InputDialog {

 public static void main(String[] args) {
      String inp = JOptionPane.showInputDialog(
                null,
                "Enter your name",
                "Our question",
                JOptionPane.QUESTION_MESSAGE);
      System.out.println("Hello, "+inp+"!");

 }

}
```

Лістинг 2.3.11.1. Введення за допомогою діалогового вікна

Після запуску програми з’являється наступне діалогове вікно, за допомогою якого і здійснюється введення:



2.3.12. Клас Console

У Java 6 з’явився клас **Console.** Незважаючи на багатообіцяючу назву, цей клас не є таким вже потужним: він має методи лише для введення з клавіатури рядків та паролів. Вважається, що основним призначенням цього класу було забезпечення можливості для введення паролів у зашифрованому вигляді.

Наприклад, програма, яка виводить на екран введені ім’я користувача та пароль, може мати вигляд:

```
import java.io.Console;

public class UseConsole {

public static void main (String[] args) {
final String LOGIN_PROMPT="Enter login:";
final String PASSWORD_PROMPT="Enter password:";
Console console = System.console();
String login = console.readLine(LOGIN_PROMPT);
char[] password = console.readPassword(PASSWORD_PROMPT);
System.out.println("Login is "+login);
System.out.println("Password is "+new String(password));

}

}
```

Лістинг 2.3.12.1. Клас Console: введення імен та паролів

Читачеві пропонується самостійно модифікувати цю програму так, щоб вона перевіряла правильність введення.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Охарактеризуйте відомі Вам інструментальні засоби для створення Java-програм.
2. Опишіть призначення основних утиліт JDK (java, javac, jar, javap, javadoc).
3. Як засобами JDK відкомпілювати програму та запустити її на виконання?
4. Охарактеризуйте призначеня змінних оточення PATH та CLASSPATH.
5. Як можна забезпечити виконання програми,якщо її класу немає в поточному каталозі та він не записаний в CLASSPATH?
6. Охарактеризуйте основні кроки для створення Java-застосування в середовищі Eclipse.
7. Яким чином можна переглянути байт-код класу?
8. Опишіть основне правило щодо назв публічних класів.
9. Чи обов'язково назва класу повинна співпадати з назвою файлу, в якому міститься його початковий код?
10. З виклику якого методу починається виконання звичайного консольного застосування?
11. Чи може в класі бути кілька методів main?
12. Чи можна написати клас, в якому немає методу main?
13. Чи можна написати консольне застосування - клас, в якому немає методу main, але застосування нормально компілюється і виконується?
14. Перелічіть відомі Вам варіанти написання методу main.
15. Що таке аргументи командного рядка?
16. Яким чином можна задати аргументи командного рядка в середовищі Eclipse?
17. Охарактеризуйте поняття пакету.
18. Що таке повне ім'я класу?
19. Назвіть повне ім'я стандартного класу String (рядок).
20. Який пакет імпортується автоматично?
21. Перелічіть стандартні потоки введення-виведення в Java
22. Яким чином можна вивести на екран певну інформацію?
23. Охарактеризуйте відомі Вам способи введення рядків та даних інших типів.
24. Як перетворити рядок на число?
25. Яким чином можна відформатувати виведення?
26. Охарактеризуйте основні можливості класу Scanner.
27. Яким чином для введення інформації можна використовувати діалогові вікна?
28. Охарактеризуйте клас Console. Якої можливості класу Console немає в класі java.io.BufferedReader?

ЧАСТИНА 3. ОГЛЯД ОСНОВНИХ МОЖЛИВОСТЕЙ МОВИ

Вступ

Java є об'єктно-орієнтованою мовою. Безпосередньо підтримується можливість опису класів, які відповідають певним поняттям; створення конкретних екземплярів класів; взаємодії між ними тощо. Але об'єктно-орієнтована парадигма суттєво спирається на методики процедурного, або імперативного програмування. Органічно використовуються умовні операції, цикли тощо. Методи класів – це по суті функції, включені до складу класів.

До цього слід додати наступне міркування. Об'єктна парадигма – це необхідний засіб зменшення складності програмного проекту за рахунок його декомпозиції на відносно незалежні складові – класи. пакети і т.п. Якщо ж проект не є складним, програма може залишатися по суті процедурною. Той факт, що Java вимагає оформлення такої програми як методу деякого класу, ніяк не змінює цього висновку.

Аналогічно, процедурна за своєю сутті програма може звертатися на рівні "чорної скриньки" до функцій, які надаються існуючими класами. Наприклад, для виведення поточної дати можна написати код

```
package showdate;

import java.util.Date;

public class ShowDate {

     public static void main(String[] args) {

        System.out.println("Current time is "+new Date());
    }
}
```

Лістинг 3.В.1. Приклад використання класів – виведення поточної дати

Тут використовуються класи, але для розуміння цього коду не обов'язково мати глибокі знання в галузі об'єктно-орієнтованого програмування. Все, що потрібно знати – це те, що потрібно створити екземпляр класу **Date** з пакету **java.util** (при цьому імпортувати сам пакет) та вивести його на екран.

Аналогічно, для того, щоб вивести на екран довжину деякого рядка, наприклад:

```
String s = "qwerty";
int len = s.length();
System.out.println("Length of the string "+s+" is "+len);
```

достатньо знати, що рядок є екземпляром класу **String**, і для отримання довжини рядка треба на цьому екземплярі викликати метод ***length()*** цього класу.

Мова Java має досить стандартний набір процедурних рис (умовні операції, цикли різних типів, перемикачі і т.п.), які в основному співпадають з процедурними характеристиками мови C++. Що стосується об'єктних рис, відмінності Java від C++ є набагато більш суттєвими – і за синтаксисом, і за семантикою.

У цій частині ми на досить інтуїтивному рівні зробимо початковий огляд основних процедурних та об'єктних рис Java для розуміння програмних конструкцій, які будуть використовуватися або згадуватися далі. Ми зробили певну спробу і в напрямку того, щоб

розділ міг читатися людьми, для яких Java – взагалі перша мова програмування (хоча для початкового ознайомлення з фундаментальними концепціями, як-от: загальні вимоги до програм; алгоритми, що лежать в основі програм; програми і дані тощо цього, звичайно, недостатньо).

Розділ 3.1. Огляд процедурних можливостей мови

3.1.1. Коментарі

У програмах дуже рекомендується використовувати коментарі, які пояснюють призначення всієї програми або окремих її фрагментів та роблять програму більш читабельною та зрозумілою.

Наведемо програму з двома коментарями, один з яких розміщується в одному рядку, а інший – в кількох рядках

```
package usecomments;

public class UseComments {

 public static void main(String[] args) {
 /*
       Ця програма виводить інформацію щодо коментарів.
       Даний коментар розміщується в кількох рядках
 */

 System.out.println("Use comments");//Однорядковий коментар

 }

}
```

Лістинг 3.1.1.1. Використання коментарів

3.1.2. Змінні та вирази

Використання змінних у Java в своїй основі мало чим відрізняється від таких мов, як C/C++ або Паскаль. Інтуїтивно змінні – це величини, з якими працює програма. Наприклад, ми хочемо написати програму, яка рахує площу прямокутника *S* за формулою *S=ab*, де *a* і *b* - довжини сторін. Подібний код пишеться майже автоматично. Потрібно описати три змінні: довжини сторін *a* і *b* та площу *s*. Усі три змінні є дійсними і відповідно мають тип double.

Саме обчислення площі могло б мати такий вигляд (тут ми абстрагуємося від того, що конкретні значення краще не задавати прямо в програмі, а вводити як зовнішні дані):

```
public class DemoVariables {

 public static void main(String[] args) {
      double s1 = 4.0;
      double s2 = 2.5;
      double square = s1*s2;
      System.out.println("The square of rectangle with sides
"+s1+" and "+s2+" is "+square);


 }

}
```

Лістинг 3.1.2.1. Використання змінних – обчислення площі прямокутника за його сторонами

Одиниці вимірювання довжини тут не мають суттєвого значення; нехай це будуть сантиметри. Тоді у даному випадку довжина сторони *a* дорівнює 4 см; довжина сторони b – 2.5 см, а результат s обчислюється за відповідною формулою. Запис ***s1*s2*** – це простий приклад ***виразу***.

З технічної ж точки зору змінна – це певна ділянка пам'яті з визначеною адресою, яка на програмному рівні характеризується своїм ім'ям та куди записується те чи інше значення.

3.1.3. Найважливіші типи даних

Java є мовою зі строгою типізацією. Кожна змінна, яка використовується в програмі, має бути оголошена. При цьому повинен бути вказаний її ***тип***, який визначає, яким чином слід інтерпретувати цю змінну та які операції над нею можна здійснювати.

Наведемо приклад того, як інтерпретація змінної залежить від того, який вона має тип.

```
public class DemoVariables {

 public static void main(String[] args) {

 char c = 'A';
 System.out.println(c); //Виводиться символ 'A'
 System.out.println((int) c); //Виводиться 65 - код символа

 }

}
```

Лістинг 3.1.3.1. Виведення символу та його коду

Тут змінна **c** має символьний тип **char** і тому інтерпретується як символ. Але результат ***приведення*** **(int) c** вже має тип **int**, і тому інструкція

```
System.out.println((int) c);
```

виводить код символа.

Всі типи в Java поділяються на ***примітивні*** (інша назва – ***прості***) та об'єктні. До примітивних типів належить чотири цілочисельні типи (вони відрізняються розміром; найбільш вживаним серед них є тип **int**); два дійсних (найбільш вживаний – **double**); символьний тип – **char**. Ми уже бачили приклади використання цих типів.

Є ще один простий тип – **boolean**; він призначений для операцій з логічними величинами. Змінні цього типу можуть набувати одного з двох значень – **false** (хиба) та **true** (істина).

Проілюструємо цей тип на наступному прикладі.

```
public static void main(String[] args) {
      boolean b = (2+2==5);
      System.out.println(b);

 }
```

Лістинг 3.1.3.2. Приклад булевих змінних

Тут змінна b дорівнює результатові перевірки того, чи дорівнює вираз 2+2 значенню 5. Як можна очікувати, програма виводить **false.**

Програміст має можливість описувати власні типи – перш за все ***класи***. Існує велика кількість готових класів. Зараз відмітимо два з них:

- клас **Object** – вершина ієрархії для всіх об'єктних типів;
- клас **String** для задання рядків.

3.1.4. Константи

У програмах часто буває потрібно оголошувати ***константи***. Для цього використовується ключове слово final, наприклад:

```
final double PI = 3.14159;
```

3.1.5. Надання значення

Операція = є операцією ***надання значення*** (або ***присвоювання***). Вона виконується наступним чином: обчислюється значення виразу, який записаний справа від знака =, після чого це значення надається змінній, яка стоїть зліва. Так, після виконання дій

```
int a = 2;
int b = 5;
double s = a*b;
```

змінні *a, b, s* набудуть значень 2, 5, 10.

Не треба плутати операції = та == - це різні операції. Нехай у нас є дві змінні типу int (цілочисельний тип):

```
int m=2;
int n=3.
```

Тоді операція **n=m** – це надання значення; після неї змінна n набуде значення 2; змінна **m** не зміниться. А вираз **n==m** – це операція перевірки на рівність, результатом якого буде логічне значення **false** (хибність); самі значення змінних не зміняться.

3.1.6. Основні операції Java

Як і в інших мовах програмування, у виразах Java можна застосовувати різноманітні операції (або оператори). Повний аналіз операторів Java та їх властивостей заслуговує на окрему розмову. На інтуїтивному ж рівні більшість операторів не потребують особливих коментарів. Такими є, зокрема, арифметичні оператори: + (додавання), - (віднімання), * (множення), / (ділення – звичайне для дійсних чисел і цілочисельне для цілих), % (ділення за модулем); логічні: ! (заперечення), && (кон'юнкція), || (диз'юнкція); відношення: == (дорівнює), != (не дорівнює); інші порівняння (>, <, <=, >=); ряд інших операторів.

У розділах 3.1.8 та 3.1.9 ми зупинимося на деяких менш тривіальних операціях, які будуть часто використовуватися в цьому розділі.

3.1.7. Оператор чи інструкція?

Слід звернути увагу на невелику термінологічну тонкість.

Для мови Паскаль були звичними терміни "умовний оператор", "оператор циклу тощо". У мовах C/C+ була прийнята дещо інша термінологія. **Оператор** (operator) у цих мовах – це паскалівська **операція**, як-от +, -, / тощо. Саме подібні оператори перш за все фігурують в арифметичних, логічних та інших виразах. Паскалівські ж оператори, тобто програмні ж конструкції if, for та ін. – це **інструкції** (statements) мов C/C++.

У цьому плані Java тяжіє до термінології C/C++, і ми будемо орієнтуватися саме на неї. З іншого боку, формулювання “оператор циклу”, ”умовний оператор” і т.п. є загальновживаними, і ми теж будемо їх використовувати.

3.1.8. Складені присвоювання

В ході розв’язку багатьох задач виникає необхідність збільшити або зменшити значення певної змінної на деяку кількість одиниць, наприклад:

```
c=c+5;
```

Така потреба виникає настільки часто, що для подібних операцій існує і є загальновживаною спеціальна форма запису:

```
c+=5;
```

Подібні форми складених присвоювань існують і для багатьох інших операторів.

3.1.9. Інкременти та декременти

Для збільшення або зменшення змінної на 1 існують оператори ***інкременту*** та ***декременту***, а саме:

`c++` - ***постінкремент***; значення змінної c використовується, а потім збільшується на 1;

`c--` - ***постдекремент***; значення змінної c використовується, а потім зменшується на 1;

`++c` - ***преінкремент***; значення змінної c збільшується на 1, а потім використовується;

`--c` - ***предекремент***; значення змінної c зменшується на 1, а потім використовується.

Розглянемо приклади, які ілюструють різницю між постінкрементом та преінкрементом; ситуація з декрементами є аналогічною.

В результаті виконання фрагменту коду

```
int c = 5;
System.out.println(c++);
```

буде виведено 5, оскільки значення змінної c збільшується уже після виведення. В результаті ж виконання фрагменту коду

```
int c=5;
System.out.println(++c);
```

буде виведено 6.

3.1.10. Умовні оператори

У програмі часто буває потрібно реалізовувати розгалуження, тобто вибирати один з кількох можливих варіантів дій в залежності від поточної ситуацій. Умовний оператор є основним засобом реалізації подібних розгалужень.

Одна з можливих форм умовного оператора виглядає так:

```
if (умовний вираз) операція_1; else операція_2;
```

Це означає наступне: якщо *умовний вираз* є істинним, повинна бути виконана *операція_1*; в іншому випадку – *операція_2*.

Як приклад розглянемо код, який знаходить найбільше з двох чисел. Він може мати вигляд (як і раніше, ми для простоти задаємо значення змінних безпосередньо в програмі):

```
public static void main(String[] args) {
      int a = 5;
      int b = 6;
      int maximum;
      if (a > b)
           maximum = a;
      else
           maximum = b;
      System.out.println("Maximum is " + maximum);

 }
```

Лістинг 3.1.10.1. Знаходження максимуму двох чисел за допомогою умовного оператора

Слід зазначити, що при всій простоті цього коду дана задача може бути розв'язана ще більш просто. Часто умовний оператор може бути замінений ***тернарною операцією*** (див. п.3.1.11). Що ж стосується власне максимумів або мінімумів двох чисел, то для їх знаходження можна скористатися функцією **max** з класу **Math**, наприклад:

```
int a = 5;
        int b = 6;
        int maximum = Math.max(a, b);

        System.out.println("Maximum is "+maximum);
```

Лістинг 3.1.10.2. Знаходження максимуму двох чисел за допомогою готових функцій

3.1.10. Тернарна операція

Тернарна операція часто використовується для спрощення обчислення змінних, які залежать від певної умови. У найпростішому випадку ця операція має вигляд **`умова?значення_1:значення_2.`** Якщо **`умова`** є істинною, значенням цього виразу є **`значення_1`**, якщо хибною – **`значення_2`**.

Перепишемо код для знаходження максимуму двох чисел (лістинг 3.1.10.1) з використанням тернарної операції:

```
public static void main(String[] args) {
      int a = 5;
      int b = 6;
      int maximum = (a>b)?a:b;

      System.out.println("Maximum is " + maximum);

 }
```

Лістинг 3.1.11.1. Знаходження максимуму двох чисел з використанням тернарної операції

3.1.12. Цикли

Цикл – це повторення деякої операції фіксовану кількість разів або поки виконується певна умова. Кожне окреме повторення операції називається ***ітерацією*** циклу.

З самого початку Java мала три традиційні цикли C/C++: цикл з передумовою, цикл з постумовою та цикл for. Починаючи з Java 5, з'явився ще один цикл: модифікований for, або, як його часто називають за аналогією з іншими мовами програмування, цикл foreach.

На протязі цієї книги зустрічається значна кількість різноманітних циклів. Тому тут буде наведений чисто ілюстративний приклад розв'язання однієї й тієї самої задачі на основі всіх типів циклів (крім модифікованого for, який буде проілюстровано на трохи іншому прикладі).

Задача **"Сума елементів масиву"** полягає в знаходженні суми елементів масиву цілих чисел. Масиви будуть розглядатися дещо пізніше, але зараз можна сказати, що *масив* – це деяка послідовність однотипних елементів.

Сам масив будемо задавати в програмі таким чином:

int[] m = {*перелік елементів*};

Наприклад:

```
int[] m = {5, 2, 10, 3, 40};
```

Доступ до елементів здійснюється за його індексом (номером); при цьому не забуваємо, що нумерація елементів (так само, як і в C++) починається з 0. Так, щоб вивести на екран перший елемент масиву, ми можемо написати

```
System.out.println(m[0]);
```

Нам знадобиться довжина (кількість елементів) масиву; її можна отримати за допомогою виразу `m.length`. Позначимо довжину масиву через *n*. Для простоти будемо вважати, що масив гарантовано не пустий, і в ньому є хоча б кілька елементів.

Алгоритм знаходження суми стандартний: введемо змінну s, в якій буде накопичуватися значення суми, і на кожній ітерації циклу до цієї змінної буде додаватися черговий елемент масиву.

Цикл з передумовою (цикл while) має вигляд

```
while (умова) операція;
```

Це означає наступне: тіло циклу (*операція)* має виконуватися, поки *умова* є істинною. Якщо *умова* є хибною з самого початку, цикл не буде виконаний жодного разу.

Як застосувати цей цикл до нашої задачі? Введемо допоміжну змінну *k*, яку будемо збільшувати на кожній ітерації циклу; на кожній ітерації до змінної s буде додаватися елемент `m[k]`. Цикл буде виконуватися поки *k* буде меншим за *n*.

Отже, розв'язок нашої задачі за допомогою циклу з передумовою може мати вигляд:

```
public class SumOfElements {

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

```
        int[] m = {5, 2, 10, 3, 40};
        int n = m.length;
        int s=0;
          int k=0;
          while (k<n) {
              s+=m[k];
              k++;
          }

        System.out.println("Sum of elements is "+s);

     }

    }
```

Лістинг 3.1.12.1. Знаходження суми елементів масиву за допомогою циклу з передумовою

Що буде, якщо в тілі циклу не написати операцію *k*++? Це типова помилка, яку часто роблять через неуважність. У цьому випадку *k* змінюватися не буде, і тому умова *k*<*n* завжди залишатиметься істинною. Виникне ***нескінченний цикл***, який ніколи не завершиться (якщо програму не зупинити примусово). Кажуть, що програма ***зациклюється***.

Цикл з постумовою (або цикл do-while) має вигляд

```
do операція while умова;
```

На відміну від циклу з передумовою, в циклі з постумовою спочатку виконується тіло циклу, а потім перевіряється умова виходу з циклу, так що цикл з постумовою завжди виконується принаймні один раз.

Розв'язок нашої задачі на основі циклу з постумовою може мати вигляд

```
public class SumOfElements {

 public static void main(String[] args) {
        int[] m = {5, 2, 10, 3, 40};
        int n = m.length;
        int s=0;
          int k=0;
          do {
              s+=m[k];
              k++;
          }
          while (k<n);

        System.out.println("Sum of elements is "+s);

     }

    }
```

Лістинг 3.1.12.2. Знаходження суми елементів масиву за допомогою циклу з постумовою

Цикл for. Часто кажуть, що цикл for краще використовувати в ситуації, коли кількість повторень циклу є фіксованою. Можливо, ця думка певною мірою навіяна аналогіями з циклом for в Алголі або Паскалі.

Для задачі "Сума елементів масиву" цикл for дійсно підходить ідеально. Код на основі цього циклу, швидше за все, матиме вигляд

```
public class SumOfElements {

 public static void main(String[] args) {
      int[] m = {5, 2, 10, 3, 40};
      int n = m.length;
      int s=0;
        for (int i=0; i<n; i++) {
           s+=m[i];
        }
      System.out.println("Sum of elements is "+s);

 }

}
```

Лістинг 3.1.12.3. Знаходження суми елементів масиву за допомогою циклу for.

Змінну k в цьому контексті можна охарактеризувати як ***лічильник циклу***.

Зверніть увагу: на відміну від попередніх циклів, **змінна k в тілі циклу не змінюється**. Вона вважається керуючою змінною, і змінювати її рекомендується тільки в заголовку циклу, але не в його тілі (можливо, крім окремих виняткових випадків).

Модифікований for. Ми казали, що цикл **for** ідеально підходить для роботи з масивами. Але ***модифікований for*** (часто за аналогією з іншими мовами програмування його ще називають циклом **foreach**) для перебору масивів є ще більш зручним – саме для цього він і створювався.

Цей цикл з'явився в Java 5. Він призначений для зручного перегляду наборів даних: масивів, колекцій тощо.

Код для розв'язання задачі "Сума елементів масиву", швидше за все, буде мати вигляд, подібний до такого:

```
public class SumOfElements {

 public static void main(String[] args) {
      int[] m = {5, 2, 10, 3, 40};
      int s=0;
        for (int i:m) {
           s+=i;
        }
      System.out.println("Sum of elements is "+s);

 }

}
```

Лістинг 3.1.12.4. Знаходження суми елементів масивку за допомогою модифікованого for

Зверніть увагу на наступне. Для розв'язку нашої задачі за допомогою модифікованого for знати довжину масиву не потрібно. Лічильник циклу не використовується; змінна *i* в наведеному циклі – це не індекс чергового елементу масиву, а сам цей елемент. Спроба записати модифікований for за аналогією зі звичайним циклом for:

```
public class SumOfElements {

 public static void main(String[] args) {
      int[] m = {5, 2, 10, 3, 40};
      int s=0;
        for (int i:m) {
           s+=m[i];
        }
      System.out.println("Sum of elements is "+s);

 }

}
```

Лістинг 3.1.12.5. Модифікований for – характерна помилка

є типовою для початківців помилкою, яка, швидше за все, призведе до аварійного завершення програми.

Для виходу з циклу типовим є використання інструкції ***break,*** а для переходу до наступної ітерації – інструкції ***continue***.

3.1.13. Цикли: спочатку думаємо, потім кодуємо

Ми казали про те, що цикли слід використовувати для обробки наборів та послідовностей даних, зокрема для знаходження сум, добутків і т.п. У більшості випадків це повністю відповідає дійсності – але не завжди.

Розглянемо задачу "Добуток дробів". Потрібно написати функцію, яка обчислює добуток

$$f(n) = \prod_{i=1}^{n} \frac{i}{i+1}$$

число $n>0$ має бути параметром функції. Для простоти ми ігноруємо обробку помилкових даних (наприклад, ситуацію, коли $n<1$).

"Лобове" рішення, яке часто пишуть внаслідок бездумного механічного кодування

```
static double prodDrob(int n) {
      double prod=1.;
      for (int i=1; i<=n; i++) {
           prod*=i/(i+1);
 }
      return prod;
 }
```

Лістинг 3.1.13.1. Задача "Добуток дробів": неправильне рішення

звичайно ж, не працює. Для буль-якого аргумента *n>1* ця функція повертає 0. Справа тут у тому, що змінна `i` має тип `int`. Тому вираз `i/(i+1)` обчислюється за правилами цілочисельного ділення і для будь-якого значення `i` дорівнює `0`.

Спробуємо виправити цю помилку на основі використання `double`:

```
static double prodDrob(int n) {
      double prod=1.;
      for (int i=1; i<=n; i++) {
           prod*=i/(i+1.);
 }
      return prod;
 }
```

Лістинг 3.1.13.2. Задача “Добуток дробів”: рішення формально правильне, але неефективне

Тепер функція видає вірні результати (принаймні для невеликих *n*), але цей варіант все одно незадовільний. Питання тут в ефективності: **для обчислення *f(n)* цикл взагалі не потрібний**. Чисельники та знаменники взаємно скорочуються, і

f(n) = (1/2)(2/3)*…*(n/(n+1)) = 1/(n+1).*

Тому код функції повинен мати вигляд

```
static double prodDrob(int n) {
      return 1./(n+1);
 }
```

Лістинг 3.1.13.3. Задача “Добуток дробів”: виправлений варіант

Мораль проста: навіть якщо задача здається очевидною, не варто зразу програмувати перший варіант, який спадає на думку. Варто замислитися над суттю задачі і подумати, чи немає кращих варіантів. Особливо це стосується коду, який часто повторюється – тобто циклів.

3.1.14. Оператор варіанту

Часто буває так, в залежності від значення певної змінної треба виконувати різні дії. В принципі, подібна логіка може бути реалізована на основі умовних операцій, але це досить незручно і призводить до громіздкого коду. ***Оператор варіанту*** **switch** (який часто називають ***перемикачем***) є стандартним рішенням цієї проблеми.

Розглянемо невеличкий приклад.

Інтерактивне керування. Написати програму, яка виводить на екран поточне значення деякої цілої змінної та дозволяє її змінити наступним чином. Користувач має ввести з клавіатури один з символів:

i - збільшити значення змінної на 1;

d - зменшити значення на 1;

u – відновити значення за замовченням (нехай це значення дорівнюватиме 10);

q – завершити виконання програми.

Коректність введення не гарантується. Якщо користувач ввів неправильні дані, йому повинно бути виведено відповідне повідомлення.

Цю задачу можна розв'язувати різними способами; оберемо один з найпростіших. Реалізуємо потенційно нескінченний цикл, на кожній ітерації якого будемо вводити чергову команду. Якщо довжина цього рядка дорівнює 1 – беремо її перший символ та реалізуємо потрібні дії в залежності від значення цього символу.

Програма може мати вигляд:

```
package interactive;

import java.util.Scanner;


public class Interactive {

    public static void main(String[] args) {
        final int defaultValue = 10;
        final String message = "Please enter your command";
        final String errorMessage = "Please enter correct
data";
        int currentValue = defaultValue;
        boolean exit=false;
        Scanner scan = new Scanner(System.in);
        while (!exit) {
            System.out.println("Current value is
"+currentValue);
            System.out.println(message);
            String command = scan.nextLine();
            if (command.length()!=1)
System.out.println(errorMessage);
            else
            {char cmd = command.charAt(0);
                switch (cmd) {
                    case 'i': currentValue++; break;
                    case 'd': currentValue--; break;
                    case 'u': currentValue=defaultValue;
break;
                    case 'q': exit=true; break;
                    default:
System.out.println(errorMessage); break;
                }
            }
        }
    }
```

Лістинг 3.1.14.1. Інтерактивне керування – ілюстрація оператора варіанту

Тут буде змінюватися змінна **currentValue**. Ми ввели також три константи: **defaultValue** для значення за замовченням, а також **message** та **errorMessage** для повідомлень.

Потенційно нескінченний цикл організовано досить звичним чином. Ми ввели керуючу логічну змінну

```
boolean exit=false;
```

цикл має вигляд

```
while (!exit) {…}
```

Відповідно до логіки програми, за певних умов у тілі циклу змінна exit набуває значення true, і це означає вихід з циклу та завершення програми.

Перший символ рядка отримуємо за допомогою виклику

```
char cmd = command.charAt(0);
```

Центральне місце займає оператор switch:

```
switch (cmd) {
    case 'i': currentValue++; break;
    case 'd': currentValue--; break;
    case 'u': currentValue=defaultValue; break;
    case 'q': exit=true; break;
    default: System.out.println(errorMessage); break;
               }
```

Ми бачимо, що в операторі switch міститься кілька розділів case – гілок, які відповідають тому чи іншому значенні керуючої змінної cmd та задають дії, які мають бути виконані в цьому випадку. Є також розділ default, який відповідає будь-якому іншому значенню cmd.

Зверніть увагу на інструкції break в кінці кожного розділу. Якщо їх опустити, то відбудеться так званий “провал” - після виконання розділу виконання перейде до наступного.

3.1.15. Функції

Поняття “функція” в цілому співпадає з загальноприйнятим. У вигляді функцій зазвичай оформлюються часто повторювані ділянки коду, до яких можна звертатися з інших місць програми.

Ми уже бачили приклади використання функцій з різних класів. Напишемо ілюстративний приклад власної функції. Напишемо програму, яка вводить з клавіатури дійсне число та виводить на екран його квадрат. Обчислення квадрату оформимо як функцію:

```
static double kvadr(double x) {
     return x*x;
    }
```

Лістинг 3.1.15.1. Функція, яка обчислює квадрат дійсного числа

Наша функція буде називатися **kvadr.** Вона приймає один параметр дійсного типу та має повертати дійсне значення. У даному випадку функція має бути описана як static, оскільки вона викликається з функції main, яка теж є статичною.

Оператор return визначає результат виконання функції, який буде отриманий кодом, що її викликає.

Вся програма може мати вигляд

```
package demofunctions;
```

```
import java.util.Scanner;

public class DemoFunctions {

 public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter your number");
      double v = sc.nextDouble();
      System.out.println("The square of "+v+" is "+kvadr(v);

 }

 static double kvadr(double x) {
      return x*x;
     }


}
```

Лістинг 3.1.15.2. Ілюстрація використання функцій

3.1.16. Рядки

Інтуїтивно рядок – це деяка послідовність символів. Рядок Java – це екземпляр класу **String**.

Сюрпризом для програмістів на C може стати те, що рядок Java не можна розглядати як масив символів. Наприклад, як вивести на екран перший символ рядка? Варіант

```
String s = "qwerty";
System.out.println(s[0]);
```

Лістинг 3.1.16.1. Виведення першого символа рядка: характерна помилка

є незаконним і призводить до помилки компіляції.

Натомість слід використовувати метод **charAt**:

```
String s = "qwerty";
System.out.println(s.charAt(0));
```

Лістинг 3.1.16.2. Виведення першого символа рядка: виправлений варіант

Рядки Java мають ряд характерних особливостей. У цьому початковому огляді необхідно звернути увагу на наступні:

- для конкатенації (зчеплення) рядків існує метод concat(), але натомість можна використовувати операцію +;
- для порівняння рядків на рівність замість операції == слід використовувати метод **equals**.

Наведемо невеликий приклад. Що виведе програма

```
String first = "qwerty";
String s1 = "qwer";
String s2 = "ty";
String second = s1+s2;
System.out.println(second);
System.out.println(first==second);
```

Лістинг 3.1.16.3. Порівняння рядків за допомогою ==

Виводиться

```
qwerty
false
```

Незважаючи на те, що рядки **first** та **second** фактично мають однаковий вміст (`qwerty`), вираз **first==second** є хибним.

Ситуація зміниться, якщо замінити **==** на метод **`equals`**.

```
String first = "qwerty";
String s1 = "qwer";
String s2 = "ty";
String second = s1+s2;
System.out.println(second);
System.out.println(first.equals(second));
```

Лістинг 3.1.16.4. Порівняння рядків за допомогою методу equals

Тепер виводиться

```
qwerty
true
```

Клас String містить значну кількість функцій; читачеві пропонується ознайомитися з ними самостійно.

Розділ 3.2. Класи та об’єкти

3.2.1. Найпростіший приклад

Як для будь-якої іншої об’єктно-орієнтованої мови, поняття класу та об’єкту {екземпляру класу) відіграють для Java ключову роль. Якщо процедурні риси Java мало чим відрізняються від аналогічних рис C++, то об’єктна модель Java має ряд суттєвих відмінностей від C++ - і за синтаксисом, і за семантикою.

JDK містить величезну кількість готових бібліотечних класів, які виконують найрізноманітніші функції. Крім цього, можна використовувати бібліотеки сторонніх розробників, а також писати власні класи.

Почнемо з невеликого прикладу створення класу та його використання.

Інколи можна зустріти твердження про те, що клас – це дещо, що поєднує дані та функції для роботи з ними (методи класу) . Подібні уявлення є дуже базовими та примітивними, але в цьому параграфі ми тимчасово приймемо цю точку зору. При цьому ми повністю абстрагуємося від усіх питань, пов’язаних з проектуванням та аналізом предметної області, інкапсуляцією даних та ін. – це просто деякий синтаксично коректний клас.

Створимо клас **PrimitiveClass**:

```
package democlass;

public class PrimitiveClass {

 int a;
 int b;

    PrimitiveClass(int aa, int bb) {
        a=aa;
        b=bb;
    }

    void inform() {
        System.out.println("a="+a+"; b="+b);
    }


}
```

Лістинг 3.2.1.1. Простий клас

У цьому класі описані дві змінні (a та b; обидві типу int) – це ***поля*** класу; дані, з якими можуть працювати функції, що описані в класі (в термінології об’єктно-орієнтованого програмування – ***методи*** класу). В класі міститься один метод inform, який просто виводить на екран значення полів.

Крім того, в класі міститься ***конструктор***, який має вигляд:

```
PrimitiveClass(int aa, int bb) {
        a=aa;
        b=bb;
    }
```

Конструктор – це спеціальний метод, ім'я якого співпадає з іменем класу. В конструкторі містяться дії, які мають виконуватися при створенні нового екземпляру класу. В даному випадку конструктор – це функція з двома аргументами, яка встановлює значення полів.

Методу main() у цьому класі немає.

Для використання цього класу створимо інший клас **DemoClass** (за аналогією з процедурним програмуванням назвемо його основним класом, або ж основною програмою). Цей клас уже бути мати точку входу до програми – метод main().

```
package democlass;

public class DemoClass {

 public static void main(String[] args) {
      PrimitiveClass ekz = new PrimitiveClass(10,20);
      ekz.inform();
      System.out.println(ekz.a+" "+ekz.b);
 }

}
```

Лістинг 3.2.1. 2. Демонстраційна програма, яка використовує клас

Програма виводить:

```
a=10; b=20
10 20
```

Що відбувається в цій програмі? За допомогою операції **new** створюється ***екземпляр класу*** з конкретними значеннями полів:

```
PrimitiveClass ekz = new PrimitiveClass(10,20);
```

Після цього на екземплярі `ekz` можна викликати методи:

```
ekz.inform();
```

Власне кажучи, поки що ми робимо з класом, створеним нами, точно те саме, що раніше робили з бібліотечними класами.

Далі програма ще раз виводить поля екземпляру – але тепер вона звертається до них напряму:

```
System.out.println(ekz.a+" "+ekz.b);
```

Ще раз підкреслимо – цей приклад ніяк не може бути взірцем для наслідування. Навпаки – це антивзірець. Це чисто ілюстративний приклад, в якому для простоти порушено багато ключових вимог до коду, і тому в реальних програмах так писати не треба.

3.2.2. Селектори та модифікатори

Перш за все – код, наведений у попередньому параграфі, грубо порушує вимоги ***інкапсуляції*** (відділення інтерфейсу від реалізації). Вважається, що поля класу – це здебільшого деталі реалізації. Тому, якщо з полями класу дійсно потрібно працювати, треба робити це через відповідні методи – і таким чином контролювати доступ до них.

Таким чином, поля класу слід оголошувати закритими (**private)**, а операції з ними здійснювати через методи. Такі методи, швидше за все, будуть загальнодоступними (**public**). З дотриманням цих вимог код класу міг би виглядати так:

```
package democlass;

public class AdvancedClass {

 private int a;
    private int b;

    public AdvancedClass(int aa, int bb) {
        a=aa;
        b=bb;
    }

 public int getA() {
      return a;
 }

 public void setA(int a) {
      this.a = a;
 }

 public int getB() {
      return b;
 }

 public void setB(int b) {
      this.b = b;
 }



}
```

Лістинг 3.2.2.1. Клас з селекторами і модифікаторами

Розглянемо одне з полів (нехай це буде поле a):

```
private int a;
```

ситуація з іншими полями аналогічна. Воно має модифікатор доступу **private** (закрите), і тому напряму звертатися до нього з-за меж класу не можна (ми це побачимо далі).

Для роботи з ним є два методи з модифікатором **public**, які можуть викликатися з зовнішнього коду:

```
public int getA() {
      return a;
 }

 public void setA(int a) {
      this.a = a;
 }
```

Метод **getA()** для отримання значення поля прийнято називати ***селектором***, а метод **setA()** для встановлення нового значення – ***модифікатором***.

Зверніть увагу на назви цих методів – вони безпосередньо пов'язані з назвою поля. Так, для поля **a** селектор прийнято називати **getA(),** а модифікатор – **setA().** З огляду на це селектори часто називають ***геттерами***, а модифікатори – ***сеттерами***.

Переконаємося в тому, що робота з полями здійснюється так, як потрібно. Тепер спроба напряму отримати значення полів не проходить:

```
public static void main(String[] args) {
      AdvancedClass ekz = new AdvancedClass(10,20);
      System.out.println(ekz.a+" "+ekz.b);

   }
```

Лістинг 3.2.2.2. Поля закриті – помилка компіляції

Поля **a** і **b** оголошені як **private**, і тому компілятор видає повідомлення про помилку. Але значення полів можна отримати за допомогою селекторів:

```
public static void main(String[] args) {
      AdvancedClass ekz = new AdvancedClass(10,20);
      System.out.println (ekz.getA()+" "+ekz.getB());

   }
```

Лістинг 3.2.2.3. Для отримання значень полів використовуємо селектори

За допомогою ж модифікатору можна встановити нове значення поля, наприклад:

```
ekz.setA(50);
```

Тут значення поля a буде встановлене в 50.

3.2.3. Подальший розвиток

Клас **AdvancedClass** з лістингу 3.2.2.1 має ще не зовсім загальноприйнятий вигляд.

По-перше, конструктори у більшості типових випадків варто робити публічними.

По-друге, рекомендується явно писати конструктор без параметрів.

По-третє, конструктор з параметрами частіше пишуть іншим чином. У нашому випадку він буде мати вигляд

```
public AdvancedClass(int a, int b) {
        this.a = a;
        this.b = b;
    }
```

Лістинг 3.2.3.1. Типовий вигляд конструктора з параметрами

Слово **this** означає “цей екземпляр”, точніше “посилання на цей екземпляр”. Тут воно є необхідним для того, щоб відрізнити поля класу від параметрів конструктора.

По-третє, для того, щоб передбачити можливість для зручного виведення екземплярів класу на екран, зазвичай використовується перевизначення методу toString(). Детальніше про це буде йтися далі.

3.2.4. Спадкування

Спадкування, або ***наслідування***, є однією з найбільш фундаментальних рис об’єктно-орієнтованого програмування.

У першому наближенні, підкласи можуть спадкувати поля і методи від батьківських класів (суперкласів). Розглянемо невеликий ілюстративний приклад.

Нехай дано клас Base, в якому описано метод metod():

```
class Base {

    public void metod() {
        System.out.println("Ku-ku");
    }
}
```

Створимо похідний клас Sub, який спадкує від Base (синтаксично для відношення спадкування використовується ключове слово **extends**):

```
class Sub extends Base {

}
```

Як ми бачимо, в цьому класі немає ніяких методів. Але за рахунок спадкування методи, описані в батьківському класі Base, доступні і для цього підкласу. Перевіримо це.

Створимо екземпляр класу Sub:

```
Sub sub = new Sub();
```

Викличемо метод metod():

```
sub.metod();
```

Цей код буде нормально відкомпільований і виконаний та виведе на екран рядок `Ku-ku`.

Наведемо всю програму цілком:

```
package demoinherit;

public class Demoinherit {

    public static void main(String[] args) {
        Sub sub = new Sub();
        sub.metod();
    }
}
```

```
class Base {

    public void metod() {
        System.out.println("Ku-ku");
    }
}

class Sub extends Base {

}
```

Лістинг 3.2.4.1. Спадкування методів підкласами

Методи суперкласу можуть ***перевизначатися*** підкласами. Змінимо клас Sub наступним чином:

```
class Sub extends Base {

   @Override
   public void metod() {
      System.out.println("Ku-ku from subclass");;
   }

}
```

Лістинг 3.2.4.2. Перевизначення методів у підкласі

Запис **@Override** – це так звана ***анотація***, яка сигналізує компілятору про те, що даний метод - це перевизначений метод надкласу.

Тепер метод **metod()** перевизначений у підкласі, і код

```
Sub sub = new Sub();
        sub.metod();
```

призводить до виведення `Ku-ku from subclass`.

Звичайно, спадкування – це дуже велика і серйозна тема, і в цьому короткому огляді дано лише дуже початкове та інтуїтивне уявлення про нього.

3.2.5. Приклад: спадкування методу main

Ми уже бачили, що якщо клас розглядається як застосування, що виконується під управлінням операційної системи, в ньому обов'язково маж бути метод `main().` Але ж цей метод може бути визначений у надкласі…

Спробуємо.

Створимо клас `Base` з методом main():

```
public class Base {

public static void main(String[] args) {
System.out.println("Main is not necessary for subclasses");
}
```

```
}
```

Лістинг 3.2.5.1. Суперклас з методом main

Створимо пустий підклас Sub:

```
public class Sub extends Base {

}
```

Лістинг 3.2.5.2. Клас, який спадкує від суперкласу з методом main()

Відкомпілюємо обидва класи звичайним чином та запустимо Sub на виконання:

```
java Sub
```

Програма буде виконана та виведе

```
Main is not necessary for subclasses
```

У самому класі Sub немає методу main(), але викликається метод main() суперкласу.

Зауважимо: цей приклад є чисто ілюстративний, і ми не рекомендуємо так робити. Крім того, метод main() є статичним, і його спадкування має особливості, які заслуговують на окремий розгляд.

3.2.6. Клас Object

Клас **Object** (точніше, **java.lang.Object**, клас **Object** з пакету **java.lang**) має для мови Java особливе значення. Цей клас знаходиться на вершині ієрархії класів, і всі інші класи є похідними від нього – прямо чи опосередковано.

У класі Object визначено ряд методів; багато з них перевизначаються в підкласах для надання їм потрібної функціональності. Розглянемо кілька прикладів.

Метод equals(). Призначений для перевірки рівності об'єктів. У самому класі Object цей метод визначений так, що він порівнює лише посилання на об'єкти. Для того, щоб порівнювати вміст самих об'єктів, цей метод повинен бути перевизначений у відповідному класі. Приклад використання цього методу наведено в п. 3.2.7.

Метод toString(). Повертає рядкове подання об'єкта; широко застосовується при виведенні об'єктів. Детальніше про цей метод буде йтися в п.3.2.8.

Клас Object містить ряд інших важливих методів, з якими ми будемо знайомитися по мірі викладення.

3.2.7. Елементарний демонстраційний клас

Проілюструємо методи **equals()** та **toString()** на дуже простому прикладі. З цією метою створимо дуже простий клас з одним полем **value**, необхідними конструкторами, а також "стандартними" селекторами та модифікаторами:

```
package elementarydemo;

public class ElementaryClass {

 private int value;
```

```
 public ElementaryClass() {

 }

 public ElementaryClass(int value) {
      this.value = value;
 }

 public int getValue() {
      return value;
 }

 public void setValue(int value) {
      this.value=value;
 }

}
```

Лістинг 3.2.7.1. Елементарний клас для демонстрації методів equals() та toString()

Змістовно такий клас може існувати для зберігання інформації про певну величину та маніпулювання нею. Тоді значення цієї величини є основною характеристикою конкретного екземпляра класу та фактично визначає вміст цього екземпляру.

3.2.8. Метод equals()

Дослідимо питання рівності об’єктів. Як ми зазначали, зі змістовної точки зору ми можемо вважати об’єкти рівними між собою, якщо їх змістовні компоненти, тобто в нашому випадку поля **value** дорівнюють одне одному. Але перевіримо це в програмному коді.

Для цього створимо два екземпляри класу **ElementaryClass** з однаковими значеннями поля **value.** Ми хотіли б вважати їх рівними між собою. Будемо перевіряти їх на рівність; спочатку скористаємося оператором ==.

```
package elementarydemo;

public class DemoEquals {

 public static void main(String[] args) {
      ElementaryClass o1 = new ElementaryClass(10);
      ElementaryClass o2 = new ElementaryClass(10);
      System.out.println(o1==o2);

 }

}
```

Лістинг 3.2.8. 1. Перевірка рівності об’єктів: використання оператора ==

Програма виводить на екран `false`. Дійсно, оператор == не аналізує значення полів об’єктів; вона перевіряє лише ідентичність самих об’єктів. Це питання вимагає більш грунтовного аналізу (див. п.), але зараз можна сказати таке. Екземпляри `o1` та `o2` – це різні об’єкти, і тому вираз `o1==o2` дорівнює `false`.

Тепер перевіримо рівність об’єктів за допомогою методу equals():

```
package elementarydemo;

public class DemoEquals {

 public static void main(String[] args) {
      ElementaryClass o1 = new ElementaryClass(10);
      ElementaryClass o2 = new ElementaryClass(10);
      System.out.println(o1.equals(o2));

 }

}
```

Лістинг 3.2.8.2. Перевірка рівності об’єктів: використання equals

Програма знову виводить `false`. Спробуємо розібратися, чому.

Метод `equals()` – це метод класу `Object` для перевірки об’єктів на рівність. Але яким має бути цей метод? Клас `Object` нічого не знає про можливі підкласи та про поля цих підкласів. Тому для методу `equals()` класу `Object` не залишається нічого кращого, ніж робити те саме, що робить операція порівняння `==`, тобто перевіряти ідентичність об’єктів.

Для того, щоб екземпляри класу можна було порівнювати за змістом, необхідно задати критерій такого порівняння. Для цього в класі має бути перевизначений метод `equals()`.

Спробуємо:

```
package elementarydemo;

public class ElementaryClass {

 private int value;

 public ElementaryClass() {

 }

 public ElementaryClass(int value) {
      this.value = value;
 }

 public int getValue() {
      return value;
 }

 public void setValue(int value) {
      this.value=value;
 }

 @Override
 public boolean equals (Object obj) {
      ElementaryClass other = (ElementaryClass) obj;
      return this.getValue()==other.getValue();
```

```
 }


}
```

Лістинг 3.2.8.4. Перевизначення методу equals

Таке перевизначення методу `equals()` не є повністю загальноприйнятим, але для наших цілей цього поки що буде достатньо.

Тепер в результаті виконання коду

```
ElementaryClass o1 = new ElementaryClass(10);
      ElementaryClass o2 = new ElementaryClass(10);
      System.out.println(o1.equals(o2));
```

на екран виводиться `true`.

Розглянемо перевизначений метод `equals()` більш детально:

```
@Override
 public boolean equals (Object obj) {
      ElementaryClass other = (ElementaryClass) obj;
      return this.getValue()==other.getValue();
 }
```

Метод повертає результат обчислення виразу **this.getValue()==other.getValue(),** що цілком логічно. Але параметр цього методу має тип Object, і тому для звернення до полів підкласу ми повинні були здійснити приведення типів:

Звернемо ще раз увагу на операцію

```
return this.getValue()==other.getValue();
```

Тут метод класу для доступу до полів використовує селектори. Будь-який метод має повний доступ до всіх полів цього класу, в тому числі до тих, які мають модифікатор private. Тому можна було б написати і так:

```
return this.value==other.value;
```

Але ряд авторів надають перевагу першому записові, оскільки селектори можуть контролювати доступ до полів не лише з-за меж класу, але й з методів самого класу.

3.2.9. Метод toString()

Повернемося до класу **ElementaryClass** з лістингу 3.2.7.1; для того, щоб цей код був “під рукою”, повторимо його ще раз.

```
package elementarydemo;

public class ElementaryClass {

 private int value;
```

```
 public ElementaryClass() {

 }

 public ElementaryClass(int value) {
      this.value = value;
 }

 public int getValue() {
      return value;
 }

 public void setValue(int value) {
      this.value=value;
 }

}
```

Спробуємо написати програму, яка створює екземпляр цього класу та виводить його на екран:

```
package elementarydemo;

public class DemoToString {

 public static void main(String[] args) {
      ElementaryClass ekz = new ElementaryClass(10);
      System.out.println(ekz);

 }

}
```

Лістинг 3.2.9.1. Спроба вивести на екран екземпляр класу

Але виведення буде досить незугарним. Воно може бути, наприклад, таким:

```
elementarydemo.ElementaryClass@1db9742
```

Що відбувається? При виклику

```
System.out.println(ekz);
```

на об’єкті `ekz` неявно викликається метод `toString()`. Тому для того, щоб зробити виведення більш осмисленим, в класі `ElementaryClass` потрібно задати, яким чином треба виводити його екземпляри, а для цього вказаний метод**.** Клас з перевизначеним методом `toString()` може виглядати, наприклад, таким чином:

```
package elementarydemo;

public class ElementaryClass {
```

```
    private int value;

    public ElementaryClass() {

    }

    public ElementaryClass(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value=value;
    }

    @Override
    public String toString() {
        return "value = " + value;
    }


}
```

Лістинг 3.2.9.2. Перевизначення методу toString()

Метод `toString()` формує рядкове подання об’єкту, і ми можемо зробити сформувати це подання так, як вважаємо за потрібно.

Тепер в результаті виконання коду

```
ElementaryClass ekz = new ElementaryClass(10);
System.out.println(ekz);
```

буде виведено

```
value = 10
```

3.2.10. “Канонічний” склад класу

Як ми бачили, навіть найпростіший клас має бути написаний грамотно. Відповідно до загальноприйнятих вимог, типовий клас має містити:

- поля, які відповідно до вимог інкапсуляції потрібно описувати як `private` (крім окремих випадків);
- конструктори: один конструктор без параметрів та конструктори, які дозволяють задати значення своїх параметрів;
- селектори та модифікатори відповідно до потреби; можна уявити собі ситуації, коли селектори та/або модифікатори не варто робити публічними, або ж вони взагалі не потрібні (наприклад, якщо значення поля повинно обчислюватися автоматично і надалі не змінюватися).

Типовим є визначення в класі методів toString() та equals(). Якщо визначається equals(), має визначатися і метод hashCode(). Ця функція повертає ціле число, яке можна

використовувати для ефективного пошуку об’єктів у пам’яті. Останнє правило пов’язано з тим, що для рівних об’єктів хеш-коди повинні також співпадати.

Звичайно в класі можуть бути і інші методи, які можуть використовуватися іншими класами для виконання тих чи інших сервісних функцій.

3.2.11. Використання IDE для автоматичного конструювання класів

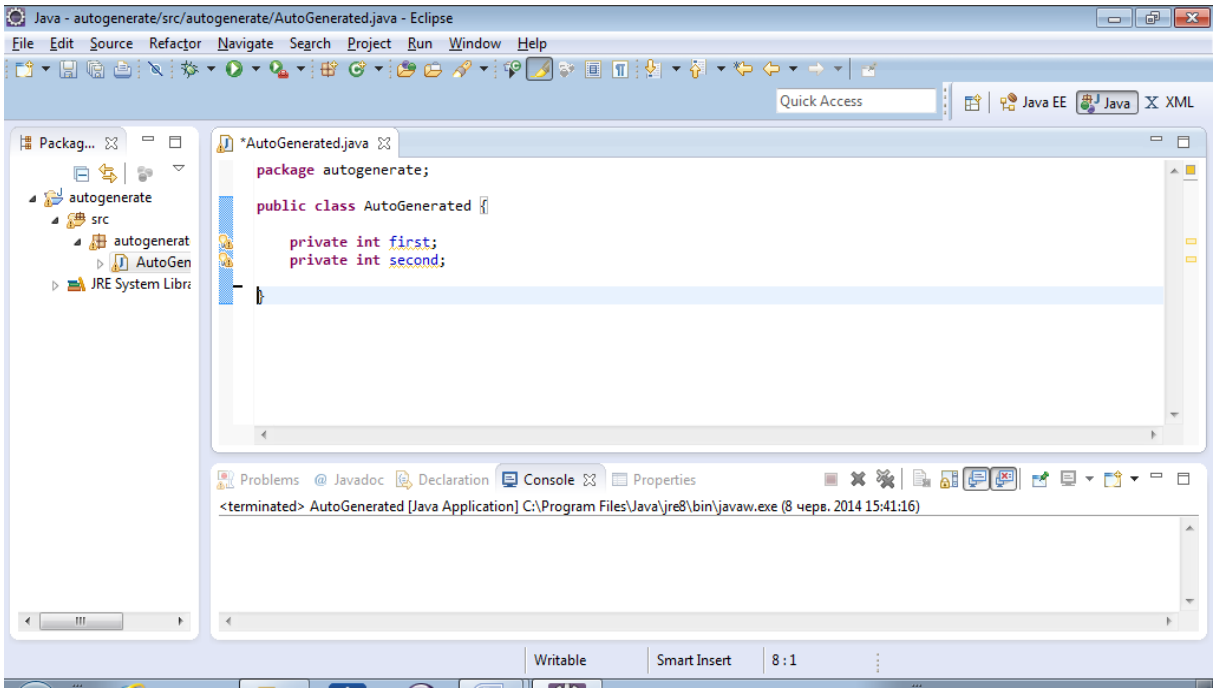
Ми бачимо, що програмування класів потребує рутинних операцій, пов’язаних з написанням значної кількості методів - конструкторів, селекторів/модифікаторів та ін. На щастя, значну частину цього процесу можна автоматизувати при використанні таких IDE, як Eclipse або NetBeans. Покажемо, як можна пришвидшити створення класу в середовищі Eclipse; можливості NetBeans є аналогічними.

Створимо звичайним чином проект в Eclipse. У цьому проекті створимо проект, а в ньому клас з двома цілочисельними полями `first` та `second`:

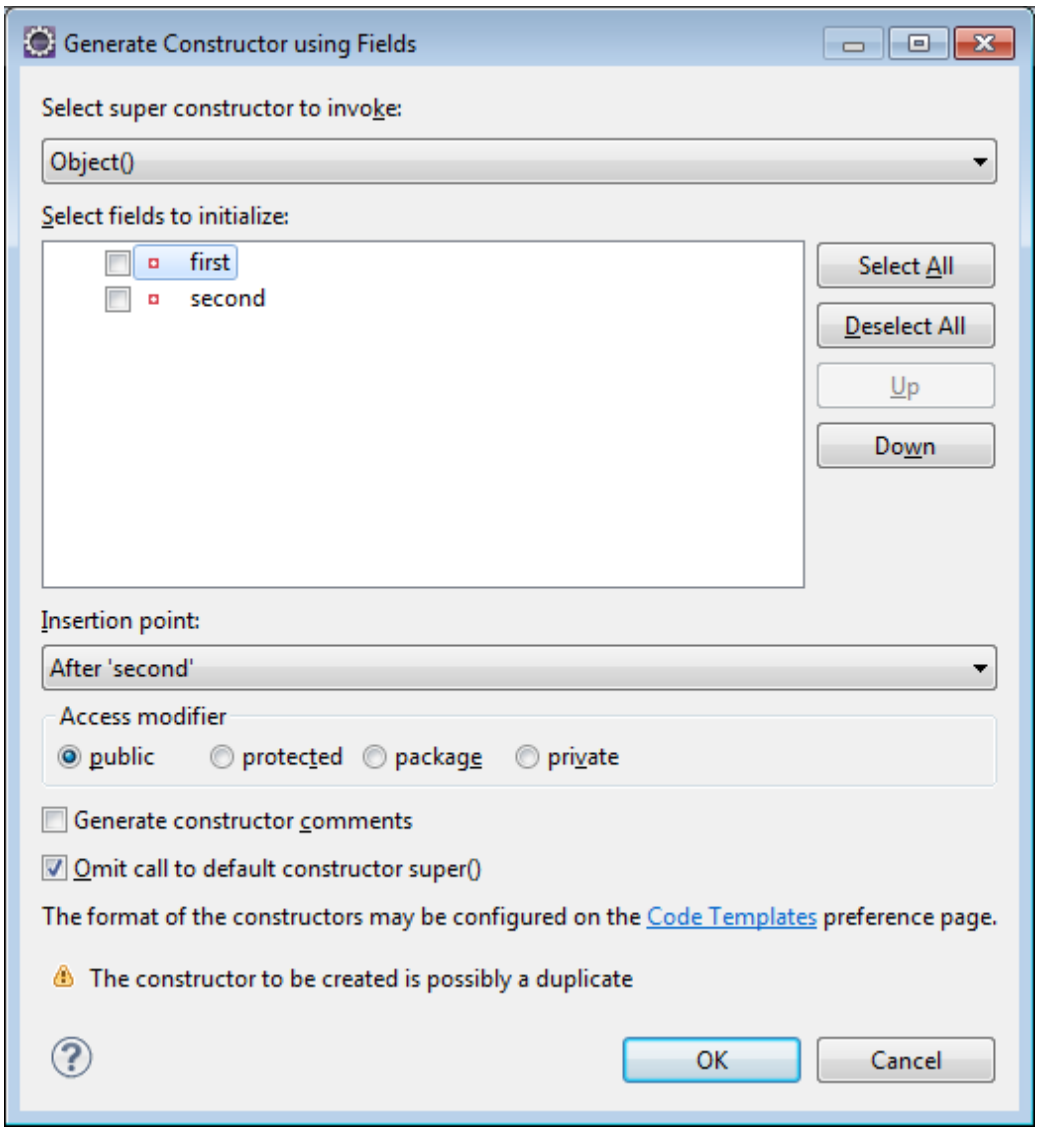
```
package autogenerate;

public class AutoGenerated {

 private int first;
 private int second;

}
```

Лістинг 3.2.11.1. Автоматична генерація елементів класу: перший крок

Після створення такого класу середовище повинно мати вигляд, подібний до такого:



Меню Source містить ряд функцій, які дозволяють створювати необхідні компоненти коду. Наприклад, створимо конструктор без параметрів. Вибираємо функцію `Generate Constructor Using Fields…` Ми маємо побачити віконце, подібно до такого:



Не будемо відмічати ніяких полів; встановимо опцію `Omit call to default constructor super()` (опустити виклик конструктора суперкласу за замовченням).

Буде згенеровано код

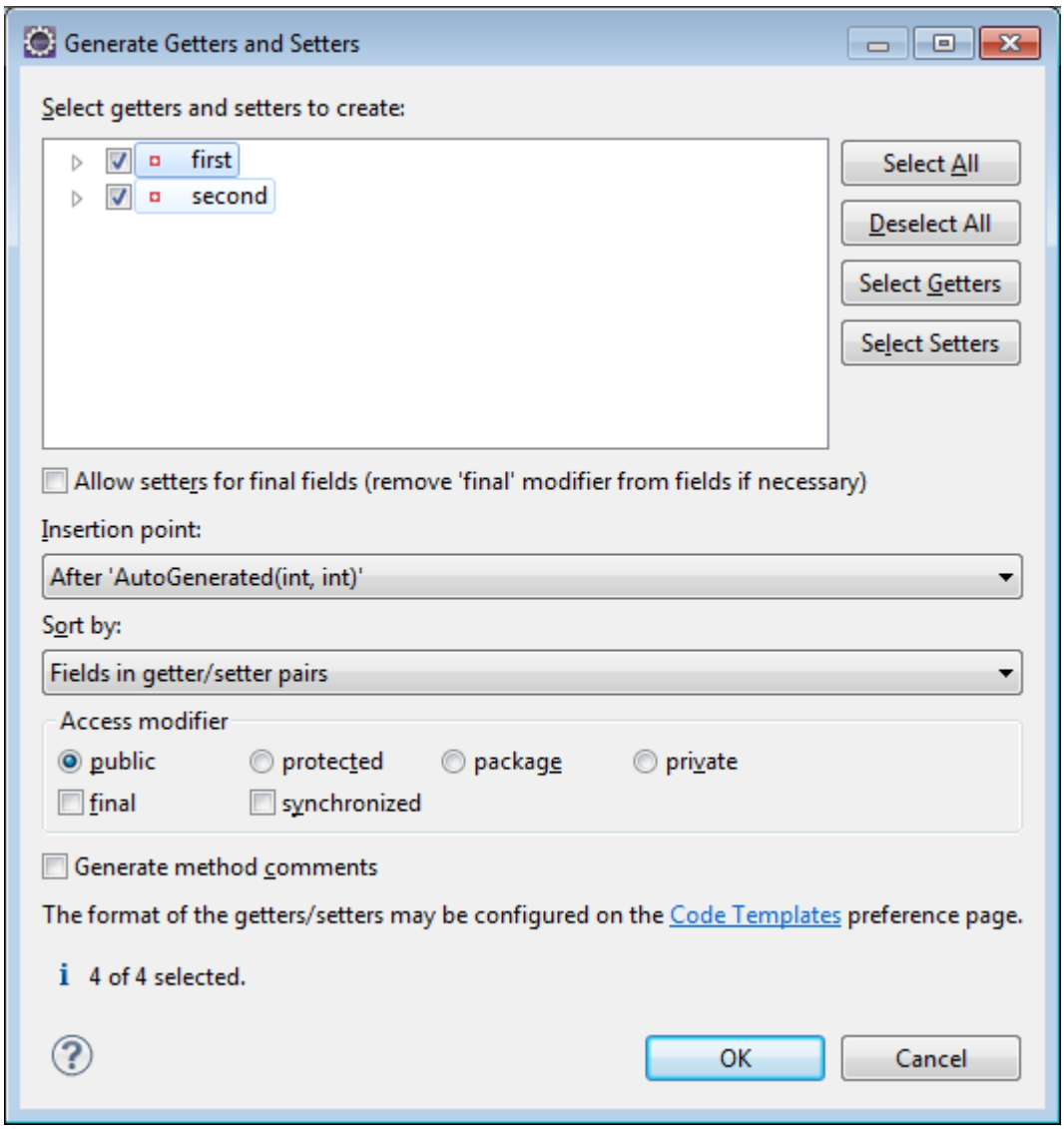
```
public AutoGenerated() {
 }
```

Аналогічно, якщо відмітити обидва поля:

буде створений “стандартний” конструктор, який встановлює значення відмічених полів:

```
public AutoGenerated(int first, int second) {
      this.first = first;
      this.second = second;
 }
```

Для створення селекторів та модифікаторів обираємо пункт Generate Getters and Setters… У вікні



відмічаємо, які саме селектори та модифікаторів потрібно згенерувати.

Тут ми вибрали всі селектори та модифікатори. Не забуваємо про те, що модифікатор доступу до цих методів має бути public.

Подібним чином створюємо методи `toString(), equals(), hashCode()`.

Зверніть увагу на код згенерованого методу `equals()`:

```
@Override
 public boolean equals(Object obj) {
      if (this == obj)
           return true;
      if (obj == null)
           return false;
      if (getClass() != obj.getClass())
           return false;
      AutoGenerated other = (AutoGenerated) obj;
      if (first != other.first)
           return false;
      if (second != other.second)
           return false;
      return true;
 }
```

Лістинг 3.2.11.2. Згенерований код методу equals()

Тут, крім значень полів, перевіряється, чи є об’єкти, які порівнюються, екземплярами одного й того самого класу. Для отримання інформації про клас об’єкту використовується функція `getClass()`.

Наведемо остаточний код класу:

package autogenerate;

```
public class AutoGenerated {

 private int first;
 private int second;

 public AutoGenerated() {
 }

 public AutoGenerated(int first, int second) {
      this.first = first;
      this.second = second;
 }

 public int getFirst() {
      return first;
 }

 public void setFirst(int first) {
      this.first = first;
 }

 public int getSecond() {
      return second;
 }

 public void setSecond(int second) {
      this.second = second;
 }

 @Override
 public String toString() {
      return "AutoGenerated [first=" + first + ", second=" +
second + "]";
 }

 @Override
 public int hashCode() {
      final int prime = 31;
      int result = 1;
      result = prime * result + first;
      result = prime * result + second;
      return result;
 }
```

```
    @Override
    public boolean equals(Object obj) {
         if (this == obj)
              return true;
         if (obj == null)
              return false;
         if (getClass() != obj.getClass())
              return false;
         AutoGenerated other = (AutoGenerated) obj;
         if (first != other.first)
              return false;
         if (second != other.second)
              return false;
         return true;
    }



}
```

Лістинг 3.2.11.3. Остаточний код класу з автоматично згенерованими основними методами

3.2.12. Інтерфейсні типи

У Java на рівні мови підтримується поняття ***інтерфейсу***. Інтерфейси роблять програму більш гнучкою і широко використовуються в практичному програмуванні.

Інтерфейс можна уявити собі як контракт класу, тобто як перелік методів, які повинні бути в класі. Клас може **реалізувати** інтерфейс за допомогою ключового слова **implements**, в цьому випадку він повинен реалізувати всі методи, описані в інтерфейсі.

Розглянемо невеликий приклад. Створимо інтерфейс Interf, в якому описано метод `foo()`:

```
package demointerfaces;

public interface Interf {

 public void foo();

}
```

Лістинг 3.2.12.1. Інтерфейс з одним методом

Зверніть увагу: метод `foo()` не має тіла.

У класі `MyClass`, який реалізує цей інтерфейс, **повинна** міститися конкретна реалізація методу `inform()`, описаного в інтерфейсі, тому наступний код не буде відкомпільований:

```
package demointerfaces;

public class MyClass implements Interf {

}
```

Лістинг 3.2.12.2. Неправильна імплементація інтерфейсу: необхідний метод не реалізований

Правильна реалізація може бути, наприклад, такою:

```
package demointerfaces;

public class MyClass implements Interf {

 @Override
 public void foo() {
      System.out.println("Ku-ku!");

 }

}
```

Лістинг 3.2.12.3. Правильна імплементація інтерфейсу

3.2.13. Перелічення.

Перелічення, або ***перелічувані типи*** призначені для опису властивостей, які можуть набувати лише значень з деякого переліку. Наприклад, колір може бути зеленим, червоним, блакитним тощо. Аналогічно, можна перелічити можливі марки автомобілів (Mercedes, Honda, Ferrari, Slavuta і т.п.), дні тижня, назви місяців тощо.

Як працювати з такими даними в програмі? Клас `String`, який дозволяє задати довільний рядок, є надто універсальним. Можна було б кодувати можливі значення властивості за допомогою цілих чисел, але тоді доведеться витрачати багато зусиль на відстеження цієї відповідності та на взаємні перетворення.

У Java починаючи з версії 1.5 існує пряма підтримка перелічуваних типів, які дозволяють безпосередньо оперувати з константами цих типів. Для опису перелічень використовується ключове слово **enum.**

Розглянемо приклад. Введемо перелічуваний тип `Car`, який описує марки автомобілів:

```
enum Car {

        Slavuta, Ferrari, Mersedes, Honda
    }
```

Лістинг 3.2.13.1. Приклад перелічуваного типу

Проілюструємо використання цього типу. Спочатку за допомогою виклику

```
Car [] cr = Car.values();
```

отримаємо масив усіх значень типу `Car`. Далі проходимо по цьому масиву та для кожного значення формуємо рядок з відповідним повідомленням, яке потім буде виводитися на екран:

```
for (Car c:cr) {
switch (c) {
    case Slavuta: claim="Крута тачка"; break;
```

```
    case Ferrari: claim="Теж нічого"; break;
    case Mersedes: claim = "Навіщо Вам та колимага?";break;
    case Honda: claim = "А в Китаї теж роблять машини";
}
System.out.println (c.ordinal()+"."+c+" - "+claim);
    }
```

Вся програма має вигляд:

```
package demoenum;
enum Car {

        Slavuta, Ferrari, Mersedes, Honda
    }

public class Demoenum {
    public static void main(String[] args) {
        Car [] cr = Car.values();
String claim = "";
for (Car c:cr) {
switch (c) {
    case Slavuta: claim="Крута тачка"; break;
    case Ferrari: claim="Теж нічого"; break;
    case Mersedes: claim = "Навіщо Вам та колимага?";break;
    case Honda: claim = "А в Китаї теж роблять машини";
}
System.out.println (c.ordinal()+"."+c+" - "+claim);
    }
    }

    }
```

Лістинг 3.2.13.2. Створення та використання перелічуваних типів

В результаті отримуємо таке виведення:

```
0.Slavuta - Крута тачка
1.Ferrari - Теж нічого
2.Mersedes - Навіщо Вам та колимага?
3.Honda - А в Китаї теж роблять машини
```

3.2.14. Виключення

Виключення – це нестандартні, часто помилкові ситуації, які можуть виникати під час виконання програм. Це може бути, наприклад, ділення на нуль, помилка введення-виведення тощо. Виключення розглядається як помилка часу виконання; збій, який у випадку, якщо його не обробити, призведе до аварійної зупинки програми (або окремого потоку).

Кожне виключення є екземпляром деякого класу; на вершині ієрархії класів виключень знаходиться клас **Throwable.** Найчастіше (хоча й не обов’язково) виключення– це екземпляр класу **Exception** або одного з його підкласів.

Виключення поділяються на ***контрольовані*** та ***неконтрольовані.*** Йдеться про контроль з боку компілятора. Вважається, що ситуації, в яких можуть виникнути контрольовані виключення, легше передбачити та забезпечити їх належну обробку. Ідея концепції контрольованих виключень полягає в тому, щоб доручити компілятору слідкувати за тим, що така обробка дійсно здійснюється.

Якщо в ході виконання методу може виникнути контрольоване виключення, слід діяти одним з двох способів: або передбачити його обробку в самому методі за допомогою конструкції **try-catch-finally**, або повідомити компілятор про те, що метод буде викидати виключення назовні за допомогою ключового слова **throws.**

Повернемося до відомого нам прикладу з введенням рядків. Код

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class DemoExceptions {

 public static void main(String[] args) {
      BufferedReader    br    =    new    BufferedReader(new
InputStreamReader(System.in));
          System.out.println("Please enter the string");
          String st = br.readLine();
          System.out.println("You entered: "+st);

 }

}
```

Лістинг 3.2.14.1. Помилка компіляції: відсутність обробки контрольованих виключень

не компілюється через вищезазначену причину. Вважається, що під час введення-виведення можуть виникати помилки, і тому метод `readLine()` класу `BufferedReader` може генерувати контрольоване виключення `IOException`.

Що можна зробити?

Спосіб 1. Ми можемо вирішити, що нас задовольняє стандартна реакція системи (програма зупиняється і виводиться стандартне повідомлення інтерпретатора). Тоді можна просто змінити заголовок методу `main` з використанням ключового слова **throws**:

```
package demoexceptions;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class DemoExceptions {

 public static void main(String[] args) throws IOException {
      BufferedReader    br    =    new    BufferedReader(new
InputStreamReader(System.in));
          System.out.println("Please enter the string");
          String st = br.readLine();
          System.out.println("You entered: "+st);
```

```
 }

}
```

Лістинг 3.2.14.2. Використання ключового слова throws

Спосіб 2. Ми можемо передбачити обробку виняткової ситуації в самій програмі. Включимо код, який може генерувати виключення до блоку **try** і перехопимо виключення за допомогою блоку **catch**:

```
package demoexceptions;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class DemoExceptions {

 public static void main(String[] args) {
      BufferedReader    br    =    new    BufferedReader(new
InputStreamReader(System.in));
        System.out.println("Please enter the string");
        try {
        String st = br.readLine();
        System.out.println("You entered: "+st);
        }
        catch (IOException e) {
            System.out.println("Error in reading");
        }
      }


 }
```

Лістинг 3.2.14.3. Обробка виключень в блоці try-catch

Більш загально, перехоплювати і обробляти можна кілька типів виключень, тоді кожному типу виняткових ситуацій буде відповідати окремий блок **catch**.

У схемі обробки виключень важливу роль грає ще один елемент: блок **finally**. Загальну схему обробки виключень на основі конструкції try-catch-finally можна охарактеризувати наступним чином:

```
try {…}
catch (виключення e) {…}
…
finally {…}
```

Блок finally виконується завжди (крім випадків, коли зупиняється вся програма або даний потік), незалежно від того, чи трапилося виключення, чи ні, і що саме відбувалося в блоках try або catch. Тому до блоку **finally** рекомендується включати дії, які мають бути виконані у будь-якому випадку. Що стосується операцій введення-виведення, до таких дій відноситься закриття потоку. Потоки та інші ресурси, які використовувалися блоком try, мають закриватися та звільнятися в блоці finally.

Подивимося на більш розгорнутий приклад. З клавіатури вводиться дійсне число та обчислюється експонента від нього. Тут можуть виникати різні типи помилок: помилки читання, помилки закриття потоку, помилки перетворення у випадку, якщо не вдається перетворити введений рядок на дійсне число, арифметичні помилки. Тому одному блоку try відповідає кілька блоків catch. Зверніть увагу також на блок try, включений до блоку finally.

```
    import java.io.BufferedReader;
    import java.io.IOException;
    import java.io.InputStreamReader;


    public class Readwrite {

        public static void main(String[] args) {
            System.out.println("Enter the real value");
    BufferedReader     br     =     new     BufferedReader(new
InputStreamReader(System.in));
    try {
    String s = br.readLine();

    double d = Double.parseDouble(s);
    double dexp = Math.exp(d);
    System.out.println("Result is "+dexp);
    }
    catch (IOException ioe) {System.out.println("Input error");}
    catch (NumberFormatException ioe) {
     System.out.println ("Please enter the proper value");
    }
    catch (ArithmeticException are) {
     System.out.println("Error in calculations ");
    }
    catch (Exception e) {
     System.out.println ("Something strange and unbelievable");
    }
    finally {try {if (br!=null) br.close();
    else System.out.println("Inproper closing of the stream");}
    catch (Exception e)
     {System.out.println("The God is against you");}
    }
          }
    }
```

Лістинг 3.2.14.4. Введення числа: багато можливих виключень

Розділ 3.3. Примітивні та об’єктні типи

3.3.1. Різниця між примітивними та об’єктними типами

Всі типи в Java поділяються на дві категорії: ***примітивні (прості)*** та ***об'єктні (указникові).***

В Java існує 8 примітивних типів:

- цілочисельні типи (`byte, short, int, long`);
- дійсні типи (`float, double`);
- символьний тип (`char`);
- логічний, або буле вий, тип (`boolean`).

Змінні примітивних типів поводяться подібно до їх аналогів у C++ або в Паскалі. Операція

```
int a;
```

призводить до того, що на стеку виконання програми під змінну виділяється пам’ять потрібного розміру. У такому варіанті змінна `a` залишається неініціалізованою (якщо це локальна змінна, а не поле класу). Але якщо написати

```
int a=10;
```

до відповідної ділянки пам’яті запишеться число `10`.

В результаті операції

```
int b = a;
```

на стеку буде створена ще одна змінна `b`, до якої буде скопійоване значення змінної `a`.

До об’єктних типів належать класи (в тому числі рядки), інтерфейси, а також масиви.

По-перше, змінні об’єктних типів створюються не на стеку, а в динамічній купі. По-друге, всі операції з об’єктними змінними – це операції з указниками.

3.3.2. Чи є в Java указники

Офіційно указників у Java немає. Дійсно, в Java відсутні розіменування, немає арифметики указників. Але змінні об’єктних типів ведуть себе саме як указники.

Створимо клас `MyClass` з цілочисельним полем `a` та повним джентльменським набором: конструктори, сеттери та геттери і метод `toString()`:

```
package javapointer;

public class MyClass {


 public MyClass() {
```

```
}

public MyClass(int a) {
     this.a = a;
}


public int getA() {
     return a;
}

public void setA(int a) {
     this.a = a;
}


@Override
public String toString() {
     return ""+a;
}


private int a;

}
```

Що можна сказати про наступну операцію:

```
MyClass ekz;
```

Це не є створення екземпляру класу – створюється лише указник відповідного типу. Новий екземпляр можна створити з використанням операції new:

```
MyClass ekz = new MyClass(10);
```

Тепер буде створено екземпляр класу `MyClass`, і на нього буде встановлений указник `ekz`. Взагалі, доступ до екземплярів класів здійснюється тільки через указники на них.

Якщо проводити аналогії з C++, то код Java

```
MyClass ekz = new MyClass();
```

є по суті еквівалентним наступному коду в C++:

```
MyClass* ekz = new MyClass();
```

До чого призведе операція:

```
MyClass other = ekz;
```

Новий екземпляр не створюється, ніякого копіювання полів не відбувається. Просто буде створений ще один указник `other`, який вказує на той самий об'єкт (тобто здійснюється копіювання указників; указнику `other` надається значення указника `ekz`).

Відповідно до цього, операції з використанням одного указника рівнозначні операціям з використанням іншого, і тому код

```
MyClass ekz = new MyClass(10);
      MyClass other = ekz;
      other.setA(20);
      System.out.println(ekz);
```

виводить на екран 20, а не 10, як могло б здатися на перший погляд.

3.3.3. Низькорівнева реалізація списку

Для ілюстрації виконаємо ще одну вправу – ручна реалізація списку. Напишемо застосування, яке створює зв'язаний список цілих чисел від 1 до 10, а потім переглядає цей список. При цьому колекції не використовуємо; потрібно обмежитися лише низькорівневими операціями з указниками.

Якщо писати подібну програму на C/C++, основна ідея полягає в наступному. Потрібно створити структуру або клас з двома полями: саме число і указник на наступну компоненту. Сама програма може мати вигляд (ми спеціально обрали стиль, максимально наближений до стилю Java):

```
#include <iostream>
#include <conio.h>

using namespace std;

struct Node {
                    int value;
                    Node* pointer;
};

void main()
{

                    Node* beg=NULL;
                    Node* p=NULL;

                    for (int i=1; i<=10; i++) {
                     p = new Node();
                     (*p).value=i;
```

```
                    (*p).pointer=beg;
                    beg=p;
                    }

                    p=beg;
                    for (int i=1; i<=10; i++) {
                    cout<<(*p).value<<" ";
                    p=(p*).pointer;

                    }
                    _getch();

}
```

Лістинг 3.3.3.1. Низькорівнева реалізація списку на C++ з використанням указників

Програма виведе:

```
10 9 8 7 6 5 4 3 2 1
```

(числа йдуть у зворотному порядку відповідно до стекового принципу).

Якщо написати подібну програму на Java, то основна її частина по суті буде відрізнятися лише відсутністю розіменувань (для простоти ми не дотримуємося вимог інкапсуляції):

```
package javapointer;

public class Main {


 public static void main(String[] args) {
      Node beg=null;
      Node p=null;

      for (int i=1; i<=10; i++) {
           p = new Node();
           p.value=i;
           p.pointer=beg;
           beg=p;
      }

      p=beg;
      for (int i=1; i<=10; i++) {
           System.out.print(p.value+" ");
           p=p.pointer;
```

```
      }


 }

}

class Node {
 int value;
 Node pointer;
}
```

Лістинг 3.3.3.2. Низькорівнева реалізація списку на Java

3.3.4. Коли `x instanceof Object` дорівнює `false`

Як ми уже зазначали, на вершині ієрархії типів знаходиться клас `Object`.

В якому випадку результатом операції `x instanceof Object` є `false`?

Часто кажуть: у випадку, якщо `x` є змінною примітивного типу. Насправді це не так: в цьому випадку цей вираз спричиняє помилку компіляції.

Оскільки будь-який клас є нащадком `Object`, для указника `x`, що вказує на екземпляр будь-якого класу, значенням виразу `x instanceof Object` є `true`. І єдина можливість, при якій цей вираз дорівнює `false` – це коли `x` дорівнює `null`.

Розділ 3.4. Набори даних: масиви та колекції

3.4.1. Поняття про набори даних

Часто потрібно працювати з деякими наборами даних: послідовностями, масивами та ін. Як і в інших мовах, у Java підтримуються ***масиви*** - класична структура даних для зберігання послідовностей однотипних елементів. Крім того, існує велика кількість ***колекцій*** – високорівневих структур для зберігання різних наборів даних. Розглянемо кілька базових прикладів.

3.4.2. Основні риси масивів

Ми уже зустрічалися з масивами, зокрема при розгляді циклів.

Для створення масиву потрібно вказати тип елементів та задати розмір, наприклад:

```
int[] mas = new int[10];
```

Тут створюється масив цілих чисел з 10 елементів.

Основна операція для роботи з масивами – це індексація, тобто доступ до елемента за його номером. При цьому нумерація елементів масиву починається з 0. Наприклад:

```
System.out.println(mas[0]);  //Виведення на екран першого елемента масиву
mas[1]=10; //Значення другого елемента встановлюється в 10.
```

Для отримання довжини цього масиву слід використовувати вираз **`mas.length`** (зверніть увагу на синтаксис!).

Для перебору елементів масиву зручніше за все використовувати модифікований for.

Масив може бути заданий і літерально, тобто прямим переліченням його елементів, наприклад:

```
int[] mas = {10,20,40,50,30};
```

Наведемо програму, яка ілюструє всі перелічені риси. Вона створює масив та виводить на екран його довжину, а також всі елементи масиву:

```
public class DemoArrays {

 public static void main(String[] args) {
      int[] mas = {10,50,40,20,30};
      System.out.println("The length of array is "+mas.length);
      for (int m:mas) {
      System.out.print(m+" ");
      }
 }

}
```

Лістинг 3.4.2.1. Літеральне задання та перебір масиву

Програма виведе

```
The length of array is 5
10 50 40 20 30
```

3.4.3. Знайомство з класом Arrays: сортування масиву

Багато зручних засобів для роботи з масивами зосереджено в класі java.util.Arrays. Наприклад, спробуємо відсортувати масив з попереднього лістингу:

```
package demoarrasys;

import java.util.Arrays;


public class DemoArrays {

 public static void main(String[] args) {
      int[] mas = {10,50,40,20,30};
      Arrays.sort(mas);
      System.out.println("The    length    of    array    is
"+mas.length);
      for (int m:mas) {
      System.out.print(m+" ");
      }
 }

}
```

Лістинг 3.4.3. 1. Приклад сортування масиву

Тепер програма виводить відсортований масив:

```
The length of array is 5
10 20 30 40 50
```

Насправді засобів сортування досить багато. Зокрема, можна задати критерій сортування; починаючи з Java 8 для цього можна використовувати ***лямбда-вирази***. Наприклад, відсортуємо масив рядків за зростанням їх довжин:

```
package demoarrasys;

import java.util.Arrays;


public class DemoArrays {

 public static void main(String[] args) {
      String[] mas = {"zzz","a","long","this string is almost
a blob","very long"};
      Arrays.sort(mas,(a,b)->(a.length()-b.length()));
      for (String s:mas) {
      System.out.println(s);
      }
 }

}
```

Лістинг 3.4.3.2. Сортування масиву рядків з використанням ламбда-виразів

Програма виводить

```
a
zzz
long
very long
this string is almost a blob
```

Тут критерій сортування – це вираз **`(a,b)->(a.length()-b.length()).`** Це і є ламбда-вираз; він описує функцію, яка приймає два рядки та повертає різницю їх довжин. Відкладемо подальший аналіз цього коду і поки що будемо розглядати його як деяку демонстрацію окремих можливостей.

3.4.4. Динамічне заповнення масиву

Звичайно, масиви можна заповнювати і динамічно, по ходу виконання програми. Спробуємо заповнити масив випадковими цілими числами в діапазоні від 0 до 9. Для отримання випадкових значень використаємо клас **`Random`** з пакету **`java.util`**.

Звична техніка використання класу Random полягає в наступному. Спочатку створюється власне генератор випадкових чисел як екземпляр цього класу:

```
Random rand = new Random();
```

Далі можна викликати методи для отримання чергового випадкового числа. Наприклад, виклик

```
int v = rand.nextInt(10);
```

призведе до того, що змінна v отримає випадкове ціле значення в діапазоні від 0 до 9 включно.

Звертаємо увагу на те, що пам’ять під масив виділяється уже під час виконання програми. Це означає, що для задання розміру масиву можно використовувати змінні. Скористаємося цим фактом для того, щоб і сам масив мав випадковий розмір, але не перевищував 20 (для простоти ми ігноруємо той факт, що отримана довжина масиву може дорівнювати 0):

```
package demoarrasys;

import java.util.Random;


public class DemoArrays {

 public static void main(String[] args) {
      Random rand = new Random();
      int len = rand.nextInt(21); //Довжина масиву
      System.out.println("Length of array is "+len);
      int[] mas = new int[len]; //Створення масиву
      for (int i=0; i<len; i++) {
           mas[i]=rand.nextInt(10); // Черговий елемент
```

```
        }
        //Виведення масиву
        for (int m:mas) {
             System.out.print(m+" ");
        }
    }

}
```

Лістинг 3.4.4.1. Масив формується випадковим чином

Звісно, результати виконання програми можуть змінюватися від одного запуску до іншого. Виведення може бути, наприклад, таким:

```
Length of array is 11
6 4 3 7 4 1 3 6 5 0 5
```

3.4.5. Масив з екземплярів класу

Екземпляри деякого класу теж можуть бути елементами масиву. Розглянемо простий приклад.

Створимо простий клас `ElemClass` з одним полем цілого типу, необхідними конструкторами, селекторами, модифікаторами і методом `toString()`:

```
package demoarrasys;

public class ElemClass {

 private int n;

 public ElemClass() {
 }

 public ElemClass(int n) {
      this.n = n;
 }

 public int getN() {
      return n;
 }

 public void setN(int n) {
      this.n = n;
 }

 @Override
 public String toString() {
      return " "+n;
 }



}
```

Лістинг 3.4.5.1. Екземпляри цього класу будуть елементами масиву

Масив з 5 елементів цього класу можна створити наступним чином (при цьому для задання довжини введемо константу LEN):

```
final int LEN = 10; //Довжина масиву
ElemClass[] mas = new ElemClass[LEN]; //Створення масиву
```

Заповнюємо його:

```
for (int i=0; i<LEN; i++) {
          mas[i] = new ElemClass(i);
     }
```

Переглядати масив будемо звичайним чином. Вся програма має вигляд:

```
package demoarrasys;


public class DemoArrays {

 public static void main(String[] args) {
 final int LEN = 10; //Довжина масиву
 ElemClass[] mas = new ElemClass[LEN]; //Створення масиву

     for (int i=0; i<LEN; i++) {
          mas[i] = new ElemClass(i);
     }

     //Виведення масиву
     for (ElemClass m:mas) {
          System.out.print(m+" ");
     }
 }

}
```

Лістинг 3.4.5.2. Програма, яка працює з масивом екземплярів

Вона виводить:

```
0  1  2  3  4  5  6  7  8  9
```

3.4.6. Проблеми з розміром масиву

Масиви як засіб зберігання послідовностей – це структура досить ефективна, але недостатньо гнучка. Хоч і дозволяється створювати масиви під час виконання програми, але при створенні масиву ми все-таки повинні оголосити його розмір. Після цього нам уже не вдасться збільшити розмір цього масиву і додати до нього ще один елемент.

Можна реалізувати один з двох шляхів вирішення цієї проблеми:

- спочатку оголошується масив невеликого розміру; в разі потреби створюється новий масив більшого розміру, до якого переписуються існуючі дані;

- масив створюється з деяким запасом, тобто його розмір заздалегідь оголошується досить великим; вводиться спеціальна змінна, яка відстежує фактичну кількість елементів масиву.

Розглянемо наступну задачу: потрібно заповнити масив екземплярами класу **ElemClass** з лістингу 3.5.4.1. Поля цього класу мають бути цілими додатніми числами. Ці числа вводяться з клавіатури, але їх кількість заздалегідь невідома. Ознакою кінця має бути введення 0.

Реалізуємо другий підхід: заздалегідь зарезервуємо значний розмір масиву; позначимо його константою MAXLEN. Для зберігання поточного розміру введемо змінну count; при введенні нового елемента ця змінна буде збільшуватися на 1.

Перший варіант програми, яка заповнює масив, а потім виводить його вміст на екран, може виглядати наступним чином (для простоти ми вважаємо введення гарантовано коректним та ігноруємо можливі виключення)

```
package demoarrasys;

import java.util.Scanner;


public class DemoArrays {

 public static void main(String[] args) {
      final int MAXLEN = 10; //Довжина масиву
      ElemClass[] mas = new ElemClass[MAXLEN]; //Створення
масиву
      int count=0; //Поточна довжина

      Scanner sc = new Scanner(System.in);

      while (true) {
           System.out.println("Введіть наступне число");
           int elem=sc.nextInt();
           if (elem==0) break;
           mas[count++] = new ElemClass(elem);

      }

      //Виведення масиву
      for (ElemClass m:mas) {
           System.out.print(m+" ");
      }
 }

}
```

Лістинг 3.4.6.1. Заповнення та виведення масиву: неправильне використання модифікованого for

Для введення даних реалізували потенційно нескінченний цикл; якщо чергове введене число дорівнює 0, ми виходимо з цього циклу за допомогою інструкції `break`.

Але виведення програми є не зовсім правильним. Так, якщо ми введемо числа `15, 35, 50` і 0 (сигнал кінця введення), програма виведе

```
15  35  50 null null null null null null null
```

тобто виводяться зайві значення null. Це й не дивно – адже ми виводимо весь зарезервований масив, а не ту його частину, яка була фактично заповнена.

Можливо, допоможе використання для виведення звичайного циклу? Спробуємо:

```
package demoarrasys;

import java.util.Scanner;


public class DemoArrays {

 public static void main(String[] args) {
      final int MAXLEN = 10; //Довжина масиву
      ElemClass[] mas = new ElemClass[MAXLEN]; //Створення масиву
      int count=0; //Поточна довжина

      Scanner sc = new Scanner(System.in);

      while (true) {
           System.out.println("Введіть наступне число");
           int elem=sc.nextInt();
           if (elem==0) break;
           mas[count++] = new ElemClass(elem);

      }

      //Виведення масиву
      for (int i=0; i<mas.length; i++) {
           System.out.print(mas[i]+" ");
      }
 }

}
```

Лістинг 3.4.6.2. Заповнення та виведення масиву: неправильний розмір масиву

Результат той самий. Дійсно, цикл

```
for (int i=0; i<mas.length; i++) {
           System.out.print(mas[i]+" ");
      }
```

є невірним; вираз **`mas.length`** повертає оголошений розмір масиву, а не фактичну кількість введених елементів.

Фактичний розмір заповненої частини – це змінна **`count`**. Тому програму слід переписати наступним чином:

```
package demoarrasys;

import java.util.Scanner;


public class DemoArrays {

 public static void main(String[] args) {
      final int MAXLEN = 10; //Довжина масиву
      ElemClass[] mas = new ElemClass[MAXLEN]; //Створення
масиву
      int count=0; //Поточна довжина

      Scanner sc = new Scanner(System.in);

      while (true) {
           System.out.println("Введіть наступне число");
           int elem=sc.nextInt();
           if (elem==0) break;
           mas[count++] = new ElemClass(elem);

      }

      //Виведення масиву
      for (int i=0; i<count; i++) {
           System.out.print(mas[i]+" ");
      }
 }

}
```

Лістинг 3.4.6.3. Заповнення та виведення масиву: виправлений варіант

3.4.7. Перехід до колекцій

Розглянемо інший та більш рекомендований варіант розв'язання нашої задачі (заповнення масиву, розмір якого заздалегідь невідомий)– з використанням ***колекцій***.

Існує кілька типів колекцій. Для наших цілей підійдуть послідовності, найбільш вживаним з яких є **ArrayList** з пакету **java.util**.

`ArrayList` прийнято характеризувати як ***динамічний масив***, який автоматично збільшується в разі необхідності. При використанні цього класу нам більше не потрібно турбуватися про розмір колекції при додаванні нових елементів.

Дуже рекомендується зробити колекцію ***типізованою***, тобто явно вказати тип її елементів.

Оголошення колекції може мати вигляд:

```
ArrayList<ElemClass> dataSet = new ArrayList<ElemClass>();
```

Тут тип самої колекції – `ArrayLis`t; тип елементів (`ElemClass`) ми вказали в кутових дужках. Розмір колекції вказувати не обов'язково. Інтуїтивно ArrayList – це динамічний масив, до якого можна додати стільки елементів, скільки потрібно.

Починаючи з Java 7, можна використовувати спрощений синтаксис опису колекцій:

```
ArrayList<ElemClass> dataSet = new ArrayList<>();
```

Нові елементи до колекції можна додавати за допомогою методу add.
Перепишемо нашу програму з використанням колекцій:

```
package demoarrasys;

import java.util.ArrayList;
import java.util.Scanner;


public class DemoArrays {

 public static void main(String[] args) {
      ArrayList<ElemClass> dataSet = new ArrayList<>();

      Scanner sc = new Scanner(System.in);

      //Заповнення колекції
      while (true) {
           System.out.println("Введіть наступне число");
           int elem=sc.nextInt();
           if (elem==0) break;
           dataSet.add(new ElemClass(elem));
      }

      //Виведення колекції
      for (ElemClass el: dataSet) {
           System.out.print(el);
      }
 }

}
```

Лістинг 3.4.7.1. Заповнення та виведення колекції

Як ми бачимо, все значно спростилося. Зникла необхідність у змінних для зберігання розміру, а для виведення елементів колекції знову можна використовувати модифікований for.

3.4.8. Знайомство з багатовимірними масивами

На відміну від багатьох інших мов програмування, у Java немає справжніх багатовимірних масивів. Насправді багатовимірний масив Java – це “масив масивів”, тобто масив, елементами якого є масиви меншої розмірності. З іншого боку, ці масиви можуть мати різну довжину.

Наведемо невеличкий приклад, в якому здійснюється виведення двовимірного масиву по рядкам. Сам масив задається літерально. Крім того, виводяться довжини окремих підмасивів.

```
    public static void main(String[] args) {
          int[][] polyArray = { { 11, 12, 13, 14, 15 }, { 21, 22,
23 },
                      { 31, 32, 33, 34 } };

          for (int[] ir : polyArray) {
```

```
            System.out.print(ir.length + " - ");
            for (int m : ir) {
                 System.out.print(m + " ");
            }
            System.out.println();
       }
 }
```

Лістинг 3.4.8.1. Виведення двовимірного масиву

Програма виводить:

```
5 - 11 12 13 14 15
3 - 21 22 23
4 - 31 32 33 34
```

Зверніть особливу увагу на цикл, який здійснює виведення

```
for (int[] ir : polyArray) {
            System.out.print(ir.length + " - ");
            for (int m : ir) {
                 System.out.print(m + " ");
            }
            System.out.println();
       }
```

Цей цикл є подвійним, що цілком відповідає інтуїтивним очікуванням. Але, якщо змінна `m` у внутрішньому циклі – це звичайна цілочисельна змінна, яка пробігає по всім елементам внутрішнього масива, то змінна `ir` у зовнішньому циклі – це масив, і тому має тип `int[]`.

КОНТРОЛЬНІ ЗАПИТАННЯ ДО ЧАСТИНИ 3

1. Навіщо в програмі використовувати коментарі?
2. Як записуються однорядкові та багаторядкові коментарі в Java?
3. Що таке змінна?
4. Як у програмі задати константу?
5. Охарактеризуйте поняття типу даних.
6. Які Ви знаєте типи даних у Java?
7. Який тип даних найчастіше використовується для опису цілих чисел?
8. Який тип має вираз `a==b`?
9. Як виконується оператор присвоювання (надання значення)?
10. Чим відрізняються операції x==y та x=y?
11. Що означає запис `c/=5`?
12. Що таке інкремент і декремент?
13. Чим відрізняється преінкремент від постінкременту?
14. Що таке умовний оператор?
15. Як знайти найбільше з двох чисел за допомогою умовного оператора?
16. Як знайти найбільше з двох чисел за допомогою тернарної операції?
17. Перелічіть відомі Вам типи циклів.
18. Намалюйте блок-схему циклу for.
19. Напишіть цикл while, еквівалентний циклові `for(int i=0; i<100; i++0 {doSomething(i);}`
20. Яким чином можна знайти суму елементів масиву за допомогою звичайного і модифікованого for?
21. Для чого використовуються інструкції break та continue?
22. Для чого використовується та як записується оператор варіанту?
23. Що таке функція?
24. Яким чином функція повертає результат свого виконання?
25. Чи може тип аргументів функції відрізнятися від типу її результату?
26. Що означає слово void у записі `void foo() {}`?
27. Екземпляром якого класу є рядок?
28. Яка існує операція для конкатенації рядків?
29. Яким чином можна отримати символ, який стоїть на заданій позиції в рядку?
30. Як порівняти два рядки на рівність?
31. Охарактеризуйте поняття класу та екземпляру класу.
32. Як у Java описуються класи?
33. Як створити екземпляр класу?
34. Що таке конструктор?
35. Чи може в класі бути кілька конструкторів?
36. Яке існує правило щодо полів класу з точки зору інкапсуляції?
37. Що таке селектори і модифікатори?
38. Що означають описи `private` та `public`, якщо вони застосовуються до полів і методів класу?
39. Що означає ключове слово `this`?
40. Як у Java записується спадкування?
41. Який клас знаходиться на вершині ієрархії класів?
42. Які Ви знаєте методи класу Object?
43. Охарактеризуйте призначення методу equals().
44. Охарактеризуйте призначення методу toString().
45. Яким чином можна забезпечити, щоб для екземпляру ekz деякого класу можна було написати `System.out.println(ekz)`?

46. Яким чином можна забезпечити, щоб для екземпляру ekz деякого класу при виклику `System.out.println(ekz)` виводилися значення полів цього екземпляру?
47. Чи можна створити клас з власними методами equals() та toString()?
48. Для чого призначена анотація @Override?
49. Як можна пришвидшити написання конструкторів, селекторів і модифікаторів при створенні класу в Eclipse?
50. Що таке інтерфейс?
51. Яким чином клас може реалізувати деякий інтерфейс?
52. Що таке перелічуваний тип?
53. Як описати перелічуваний тип?
54. Чи може керуюча змінна в операторі варіанту switch бути перелічуваного типу?
55. Що таке виключення?
56. Охарактеризуйте поняття контрольованого та неконтрольованого виключення.
57. До якого класу виключень відноситься IOException: контрольованих чи неконтрольованих?
58. Що означає запис throws Exception в описі методу?
59. Опишіть конструкцію try-catch-finally.
60. Чи можлива ситуація, коли з секцією try пов’язано кілька секцій catch?
61. Що таке масив?
62. Як створити масив?
63. Чи може масив бути заданий літерально?
64. Яким чином можна отримати доступ до елемента масиву з заданим номером?
65. Для чого призначений клас Arrays?
66. За допомогою якого методу класу Arrays можна вивести на екран всі елементи масиву?
67. Як відсортувати масив?
68. Чи можна при заданні розміру масиву вказувати змінні?
69. З чим пов’язана основна незручність при використанні масивів?
70. Опишіть колекцію ArrayList.
71. Як до колекції класу ArrayList додати новий елемент?

ЧАСТИНА 4. АНАЛІЗ ЕФЕКТИВНОСТІ, ВІДЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ

Розділ 4.1. Відлагодження програм

4.1.1. Загальне уявлення про відлагодження

Програмування – це не тільки написання коду. Значна частка роботи програміста припадає на ***відлагодження*** програми, тобто на усунення помилок у ній.

Написати хоч трохи нетривіальну програму так, щоб вона з першого разу успішно компілювалася та правильно працювала, практично ніколи не вдається. Програмні помилки можуть бути пов'язані з найрізноманітнішими факторами: недостатня уважність або ж недбалість програміста, недостатнє розуміння особливостей мови, помилки в самому алгоритмі тощо. Існує навіть жартівливий вислів: "у будь-якій програмі, що працює, є принаймні одна помилка".

На щастя, значна кількість помилок виявляється ще на етапі компіляції. Компілятор – наш друг, але його можливості не безмежні. Якщо помилка залишається невиявленою, вона призводить до аварійного завершення програми або ж до неправильних результатів.

Класична методика відлагодження, тобто виявлення і усунення програмних помилок – відстеження проміжних результатів та значень змінних. Ця методика дозволяє проаналізувати, як змінюються значення змінних під час виконання програми і на основі цього робити припущення і висновки, чи правильно вона працює, а якщо ні – що саме йде не так.

4.1.2. Найпростіші прийоми відлагодження

Розглянемо простий приклад – цикл, який повинен підрахувати суму перших п'яти натуральних чисел. Нехай перша версія цього циклу має вигляд

```
package demodebug;

public class DemoDebug {


     public static void main(String[] args) {

          int s=0;

          for (int i=0; i==5; i++) {

               s=+i;
          }

          System.out.println("Sum is "+s);

     }

}
```

Лістинг 4.1.2.1. Початковий неправильний варіант підрахунку суми натуральних чисел

Результат повинен дорівнювати 15, але програма виводить 0. Що саме не так? Важливо робити якісь розумні припущення. Перш за все варто пересвідчитися, чи

виконується цикл взагалі. При цьому одночасно можна подивитися, як змінюється сума. Для цього на кожній ітерації будемо просто виводити значення змінної `s`:

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

      int s=0;

      for (int i=0; i==5; i++) {
           s=+i;
           System.out.println("s="+s);
      }
      System.out.println("Sum is "+s);
 }
```

Лістинг 4.1.2.2. Перша спроба виправлення – виведення проміжних результатів

В результаті отримуємо

```
Sum is 0
```

В циклі нічого не було виведено - отже, він взагалі не працював. Треба з'ясувати, чому. Тут спадає на думку подивитися на заголовок циклу. Дійсно, замість правильної умови продовження циклу `i<5` написано `i==5`, а цей вираз на початку роботи програми є хибним. Тому цикл і не виконується жодного разу.

Виправляємо цю помилку:

```
package demodebug;

public class DemoDebug {

 public static void main(String[] args) {
      int s = 0;

      for (int i = 0; i < 5; i++) {

           s=+i;
           System.out.println("s="+s);
      }

      System.out.println("Sum is " + s);

 }

}
```

Лістинг 4.1.2.3. Виправлення заголовку циклу

Програма виводить:

```
s=0
s=1
s=2
s=3
```

```
s=4
Sum is 4
```

Тепер ми бачимо, що ітерації циклу виконуються, але результат все одно неправильний.

Дивимося на проміжні результати – вони показують, що саме відбувається з сумою `s`. Бачимо, що вона змінюється не так, як потрібно. З'ясовуємо, чому так – для цього дивимося на тіло циклу. Так і є: замість правильного

```
s+=i;
```

у нас написано

```
s=+i;
```

тобто на кожній ітерації черговий доданок не додається до суми, а повністю замінює її.

Виправляємо:

```
package demodebug;

public class DemoDebug {

 public static void main(String[] args) {
      int s = 0;

      for (int i = 0; i < 5; i++) {

           s+=i;
           System.out.println("s="+s);
      }

      System.out.println("Sum is " + s);

 }

}
```

Лістинг 4.1.2.4. Не та сума

Цей варіант програми виводить

```
s=0
s=1
s=3
s=6
s=10
Sum is 10
```

В принципі, сума підраховується правильно – але не та сума, яка потрібно. Ця помилка пов'язана уже не з кодуванням, а з недостатнім розумінням самої задачі. Перше натуральне число – це 1, а не 0, і тому треба рахувати суму не від 0 до 4, а від 1 до 5.

Вносимо відповідні зміни:

```
package demodebug;

public class DemoDebug {

 public static void main(String[] args) {
      int s = 0;

      for (int i = 1; i <= 5; i++) {

           s+=i;
           System.out.println("s="+s);
      }

      System.out.println("Sum is " + s);

 }

}
```

Лістинг 4.1.2.5. Підрахунок суми натуральних чисел: виправлений варіант

Тепер результат правильний. Програма виводить:

```
s=1
s=3
s=6
s=10
s=15
Sum is 15
```

Звичайно, в цьому прикладі ми мали справу з елементарними помилками – але ж помилки можуть бути і більш замаскованими…

Залишається ще одна проблема. Після завершення відлагодження виведення проміжних результатів стає непотрібним, і відповідні інструкції потрібно вилучити з програмі. Ця робота є марудною і мало кому подобається. Крім того, таке вилучення – це теж модифікація коду, під час якої легко помилитися і щось зіпсувати.

На щастя, існують засоби для полегшення відлагодження. Познайомимося з деякими з них.

4.1.3. Використання протоколювання для відлагодження

Протоколювання – це фактично основа для моніторингу ходу виконання програми; виведення потрібної інформації на друк, у файл тощо. Типові системи моніторингу дозволяють керувати цим виведенням, об'єднувати логічно пов'язані протокольні повідомлення в групи, задавати різні формати виведення для різних груп і виконувати ряд інших корисних функцій.

Ряд можливостей для протоколювання надаються класом **`Logger`** з пакету **`java.util.logging`**.

Повернемося до лістингу 4.1.2.3; наведемо його ще раз:

```
package demodebug;

public class DemoDebug {
```

```
 public static void main(String[] args) {
      int s = 0;

      for (int i = 0; i < 5; i++) {

           s=+i;
           System.out.println("s="+s);
      }

      System.out.println("Sum is " + s);

 }

}
```

Тепер будемо здійснювати виведення проміжних результатів не за допомогою “класичного” **`System.out.prinln(),`** а за допомогою класу **`Logger`**:

```
package demodebug;

import java.util.logging.Logger;

public class DemoDebug {

 public static void main(String[] args) {
      Logger logger = Logger.getLogger
(Logger.GLOBAL_LOGGER_NAME);

      int s = 0;

      for (int i = 1; i <= 5; i++) {

           s+=i;
           logger.info("s="+s);
      }

      System.out.println("Sum is " + s);

 }

}
```

Лістинг 4.1.3.1. Виведення проміжних результатів за допомогою класу Logger

Тут ми використали стандартний ***реєстратор*** протокольних повідомлень (**`Logger.GLOBAL_LOGGER_NAME`**). Реєстратор фактично пов’язується з ідентифікатором, який визначає деяку логічну групу повідомлень. Дуже легко створити власний реєстратор, така практика є більш рекомендованою.

Як і раніше, виводяться проміжні результати; виведення може мати вигляд

```
черв. 13, 2014 10:19:17 AM demodebug.DemoDebug main
```

```
INFO: s=1
черв. 13, 2014 10:19:17 AM demodebug.DemoDebug main
INFO: s=3
черв. 13, 2014 10:19:17 AM demodebug.DemoDebug main
INFO: s=6
черв. 13, 2014 10:19:17 AM demodebug.DemoDebug main
INFO: s=10
черв. 13, 2014 10:19:17 AM demodebug.DemoDebug main
INFO: s=15
Sum is 15
```

При бажанні формат виведення може бути змінений. Але подивимося, як легко можна відключити протокольні повідомлення, коли вони стануть непотрібними. Це робиться за допомогою однієї інструкції, а саме:

```
logger.setLevel(Level.OFF);
```

Програма матиме вигляд:

```
package demodebug;

import java.util.logging.Level;
import java.util.logging.Logger;

public class DemoDebug {

 public static void main(String[] args) {
      Logger                    logger                    =
Logger.getLogger(Logger.GLOBAL_LOGGER_NAME);
      logger.setLevel(Level.OFF);

      int s = 0;

      for (int i = 1; i <= 5; i++) {

           s+=i;
           logger.info("s="+s);
      }

      System.out.println("Sum is " + s);

 }

}
```

Лістинг 4.1.3.2. Відключення протокольних повідомлень

Тепер протокольні повідомлення відімкнені, і виводиться лише остаточний результат.

4.1.4. Інструментальні засоби відлагодження

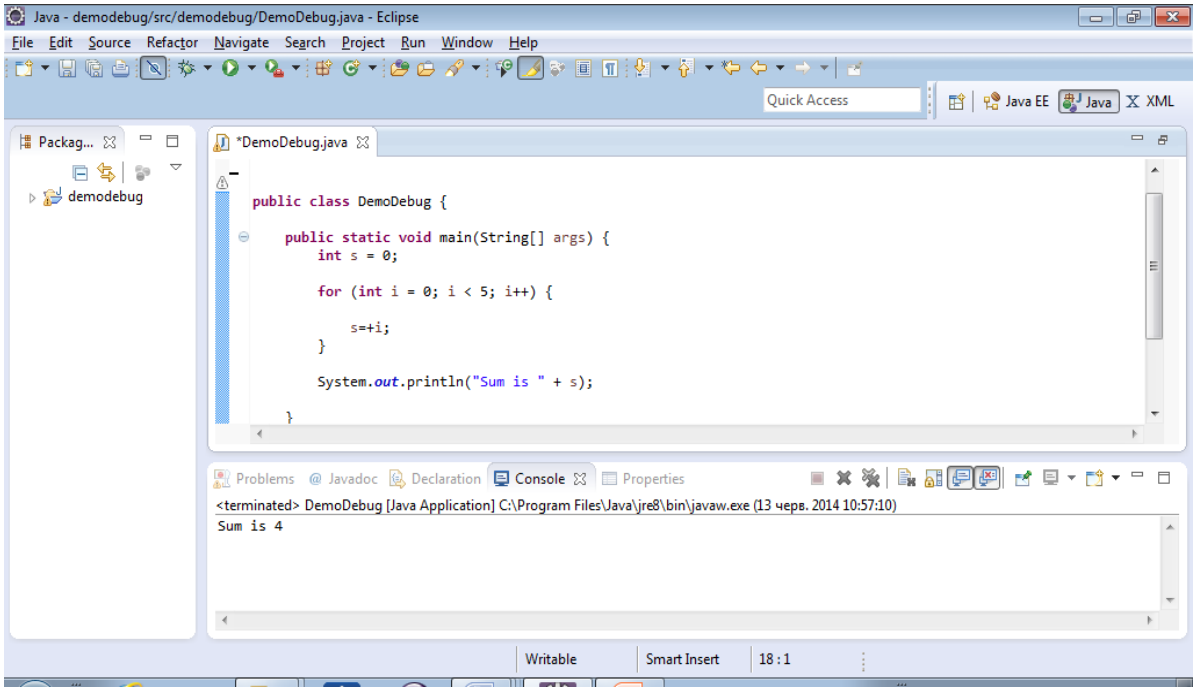
Інтегровані середовища розробки, зокрема `Eclipse` або `NetBeans`, надають у розпорядження дуже зручні засоби відлагодження. Подивимося, як можна здійснювати відлагодження в Eclipse.

Звичайним чином підготуємо в Eclipse наступний код:

```
package demodebug;


public class DemoDebug {

 public static void main(String[] args) {
      int s = 0;

      for (int i = 0; i < 5; i++) {

           s=+i;
      }

      System.out.println("Sum is " + s);

 }


 }
```

Лістинг 4.1.4.1. Програма для демонстрації можливостей Eclipse для відлагодження

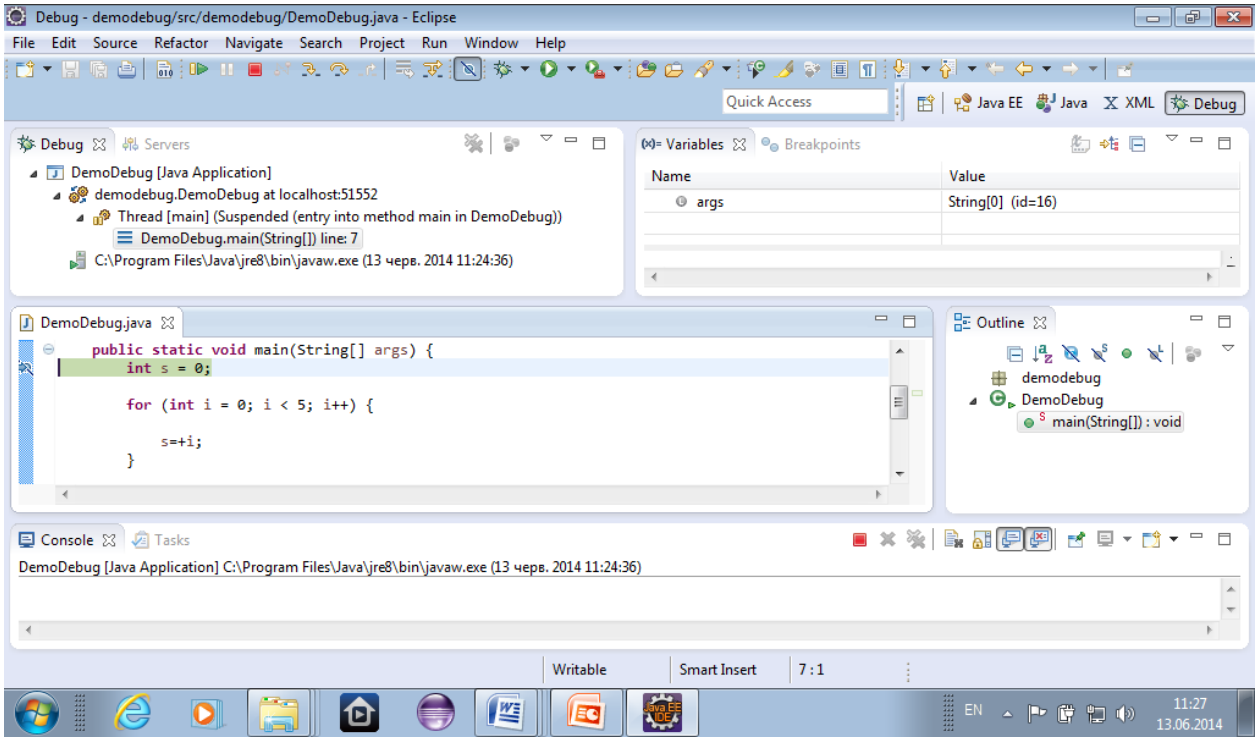
Зверніть увагу: виведення проміжних результатів тут немає.
Середовище має мати вигляд, подібний до такого:



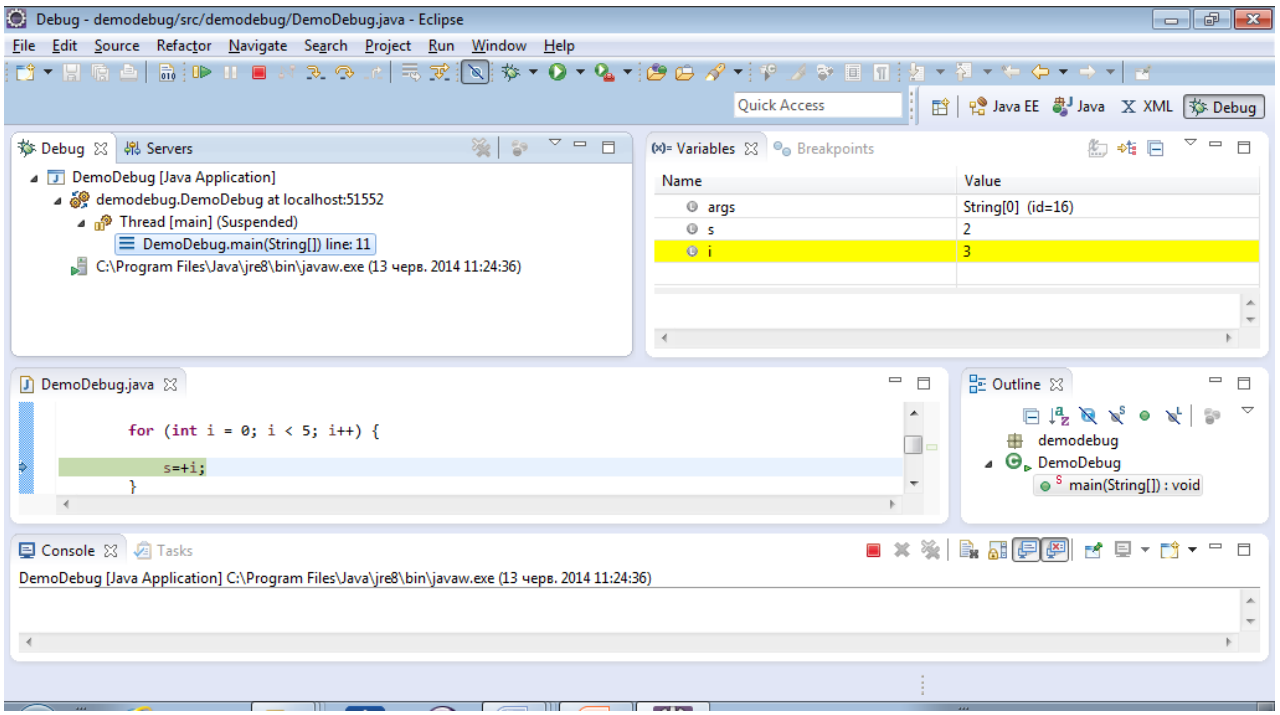
Почнемо відлагодження. Для цього має бути встановлена перспектива `Debug`. Потрібно також встановити точку зупинки на початку програми (це можна зробити за допомогою опції `Toggle Breakpoint` контекстного меню).

Запускаємо програму в режимі відлагодження (пункт меню `Run=>Debug` або відповідна піктограма). Програма зупиниться на встановленій нами точці зупинки. В окремому вікні можна побачити існуючі на даний момент змінні (на початку програми це масив аргументів командного рядка).

Середовище буде мати вигляд, подібний до такого:



Далі продовжуємо виконання в покроковому режимі за допомогою пунктів меню `Run=>Step Into/Step Over` або відповідних піктограм. Після кожного кроку стан змінних може змінюватися, і можна відстежувати ці зміни. Наприклад, на певній ітерації циклу середовище може мати вигляд



Ми бачимо, що в даний момент змінна `s` має значення 2, а змінна `i` – значення 3. Аналогічним чином можна відстежувати поля екземплярів класів, колекції і т.п.

Розділ 4.2. Тестування програм

4.2.1. Роль тестування в програмному проекті

Тестування є невід'ємною частиною процесу розробки програмного забезпечення. Метою тестування є перевірка правильності роботи програми, і без цього етапу експлуатація розроблених програмних засобів була б або неможливою, або надто ризикованою.

Розрізняють різні типи тестування. Зокрема, м***одульне тестування*** спрямовано на перевірку правильності роботи окремих класів та функцій. Метою ж ***інтеграційнего тестування*** є перевірка роботи всього програмного комплексу в цілому.

Типова схема модульного тестування полягає в перевірці роботи модулю на основі заздалегідь підібраних ***тестів*** – наборів даних, для яких відомі правильні результати. Розбіжність між очікуваними та фактично отриманими результатами сигналізує про наявність помилки. Співпадіння результатів не доводить правильності програми, але підвищує міру впевненості у відсутності помилок.

Чим повнішим і більш представницьким є набір тестів, тим більше підстав довіряти програмі. Тестові приклади мають покривати, по-перше, можливий діапазон вхідних даних, а за можливості – варіанти роботи програми відповідно до її логіки.

В ряді методик розробки програмного забезпечення тестування взагалі відіграє ключову роль. Наприклад, ***TDD*** (***Test-Driven Development***, розробка на основі тестів) передбачає спочатку написання тестів, а вже потім реального коду.

Очевидним підходом до модульного тестування є просте виведення на екран фактичних результатів роботи програми поряд з правильними. Але перегляд виведення з метою співставлення результатів не є особливо зручним. Крім того, в коді з'являється багато операцій виведення, які потім стануть непотрібними. Гарною ідеєю було б, по-перше, відділити основний робочий код від запуску тестів, а по-друге – якимось чином автоматизувати сам процес перевірки. Існує ряд утиліт, призначених для автоматизації

процесу модульного тестування. Широкого розповсюдження набула утиліта **JUnit**, яку можна використовувати автономно або в складі інтегрованих середовищ,

4.2.2. Знайомство з JUnit

Проілюструємо роботу цієї утиліти на дуже простому прикладі, а саме – на прикладі створення функції, яка знаходить добуток двох цілих чисел. На цьому іграшковому прикладі ми промоделюємо ситуацію, нібито ми заздалегідь не знаємо, як це робиться. Візьмемо такі тестові дані:

```
Вхід: 0 0; результат - 0.
Вхід: 1 1: результат - 1.
Вхід: 2 2; результат - 4.
Вхід: 3 4; результат - 12.
```

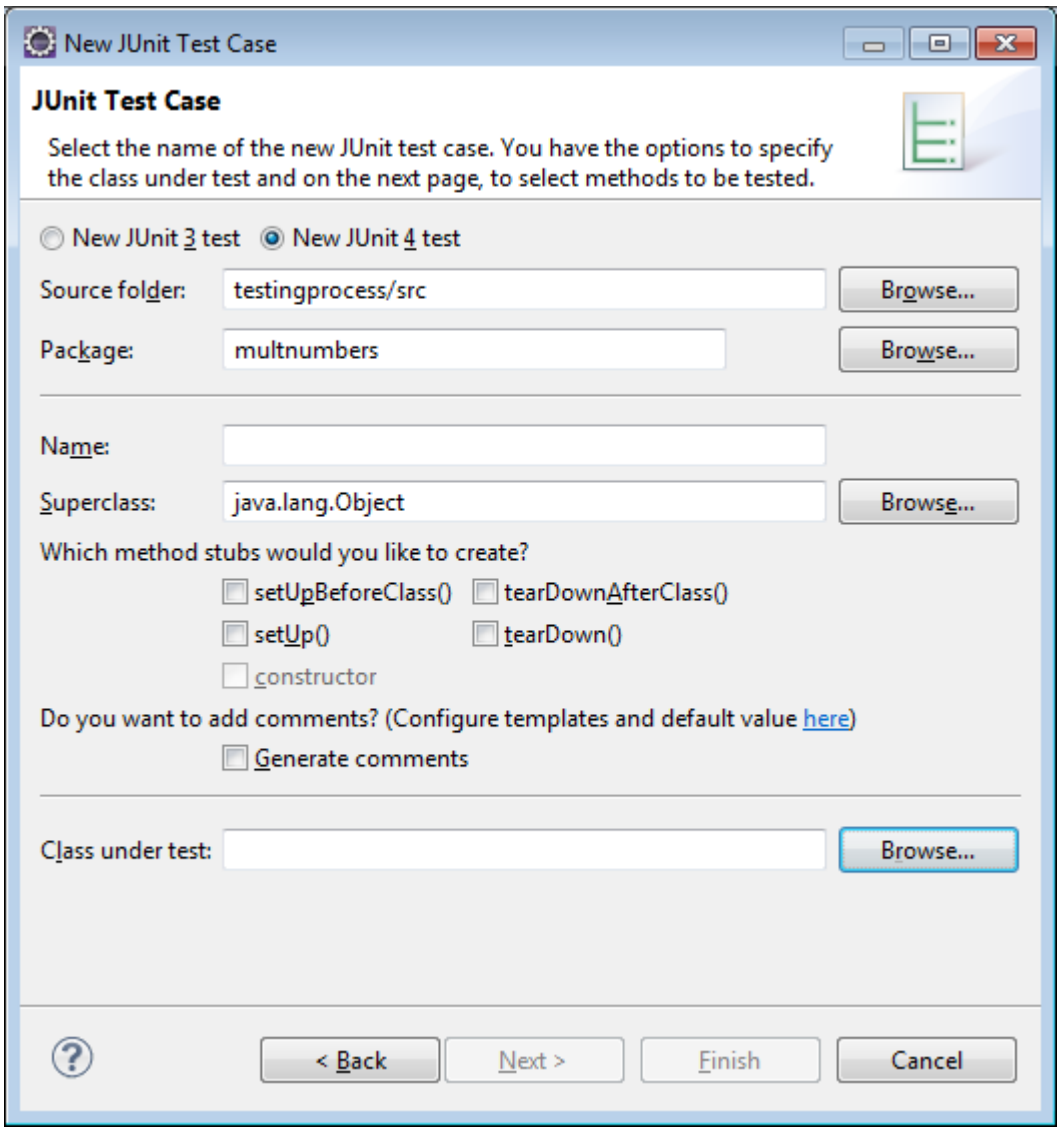
Створимо в середовищі `Eclipse` новий проект. Створимо в цьому проекті клас, в якому напишемо заглушку для функції:

```
package multnumbers;

public class MultNumbers {

 public byte multNumbers(byte a, byte b) {
      return 0;
 }

}
```

Лістинг 4.2.2.1. Функція обчислення добутку: початковий варіант

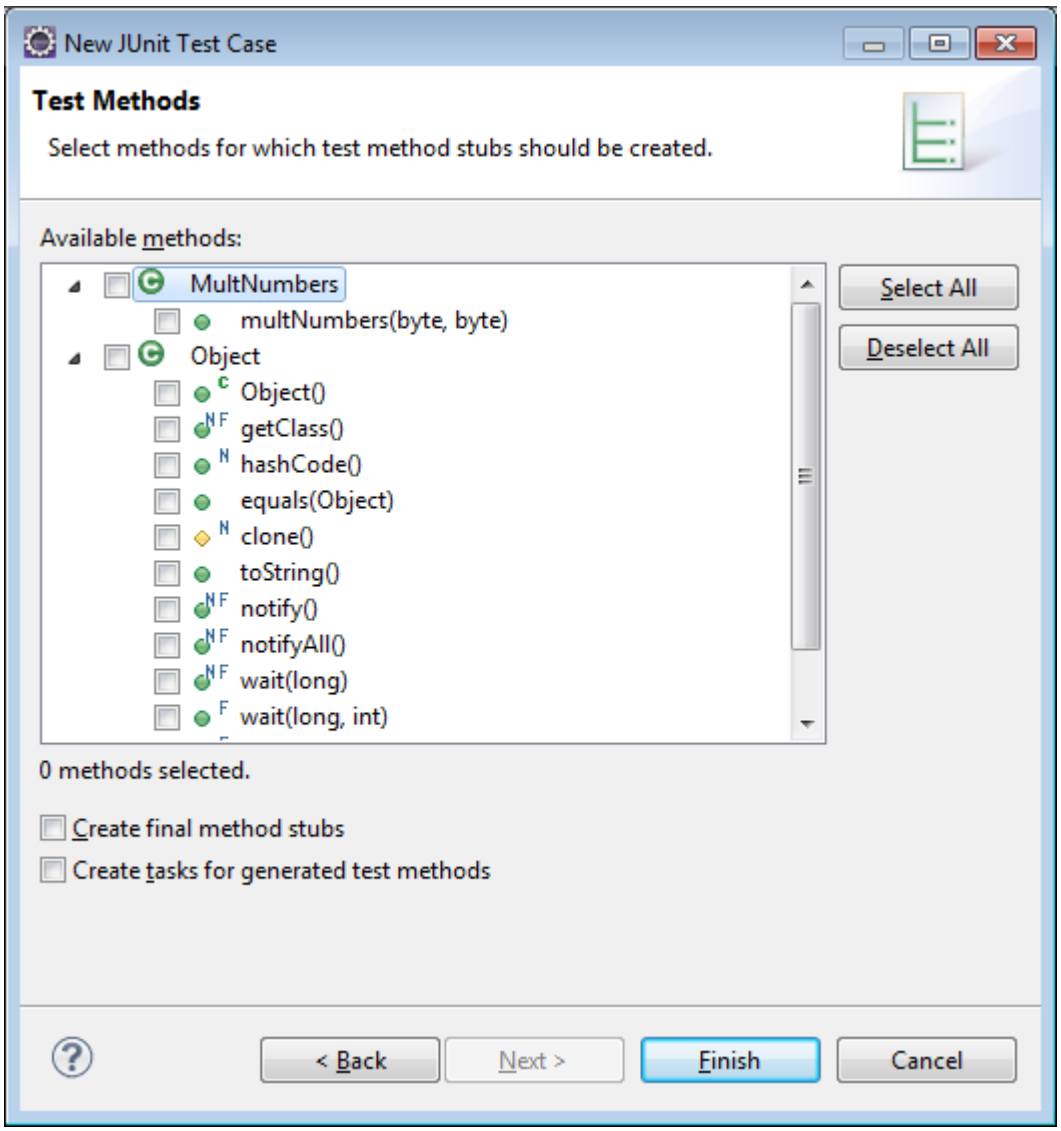
Спочатку ця функція завжди повертає 0.

Тепер підготуємо засоби для тестування. Для простоти розмістимо їх в цьому ж пакеті. Клікнемо правою кнопкою миші по назві пакету та виберемо `New->Other->JUnit-> JUnit Test Case.` Повинно з’явитися наступне діалогове вікно:



У полі Name задамо назву тестового класу (назвемо його `NumberTest`), а у полі `Class under test` – назву класу, який ми тестуємо (це `MultNumbers`).

Далі буде запропоновано вибрати методи, які ми будемо тестувати, і для яких потрібно створити тестові заглушки:



Відмітимо метод `multNumbers(byte, byte)`.

Далі нам може бути запропоновано підключити бібліотеку JUnit (якщо це не було зроблено раніше). Погодимося з цією пропозицією.

В результаті повинен бути створений новий тестовий клас:

```
package multnumbers;

import static org.junit.Assert.*;

import org.junit.Test;

public class NumberTest {

 @Test
 public void testMultNumbers() {
      fail("Not yet implemented");
 }

}
```

Лістинг 4.2.2.2. Клас для модульного тестування, згенерований JUnit

Ми бачимо заготовку тестового методу `testMultNumbers()`, який потрібно наповнити конкретним змістом. Вона супроводжується анотацією `@Test`.

У цьому методі ми можемо здійснити перевірку зразу всіх тестових прикладів, але тоді важко буде відділити, для яких з них результати були правильні, а для яких – ні. Тому зробимо для кожного тестового прикладу окремі методи.

Основний інструмент контролю, який ми будемо застосовувати – це метод `Assert.assertTrue(condition).` Якщо умова `condition` є хибною, він викидає виключення.

Як підготовчі дії, треба створити екземпляр класу `MultNumbers`.

Один із тестів може мати вигляд:

```
    @Test
     public void testMultNumbers00() {
          Assert.assertTrue(mn.multNumbers((byte)    0,    (byte)
0)==0);
     }
```

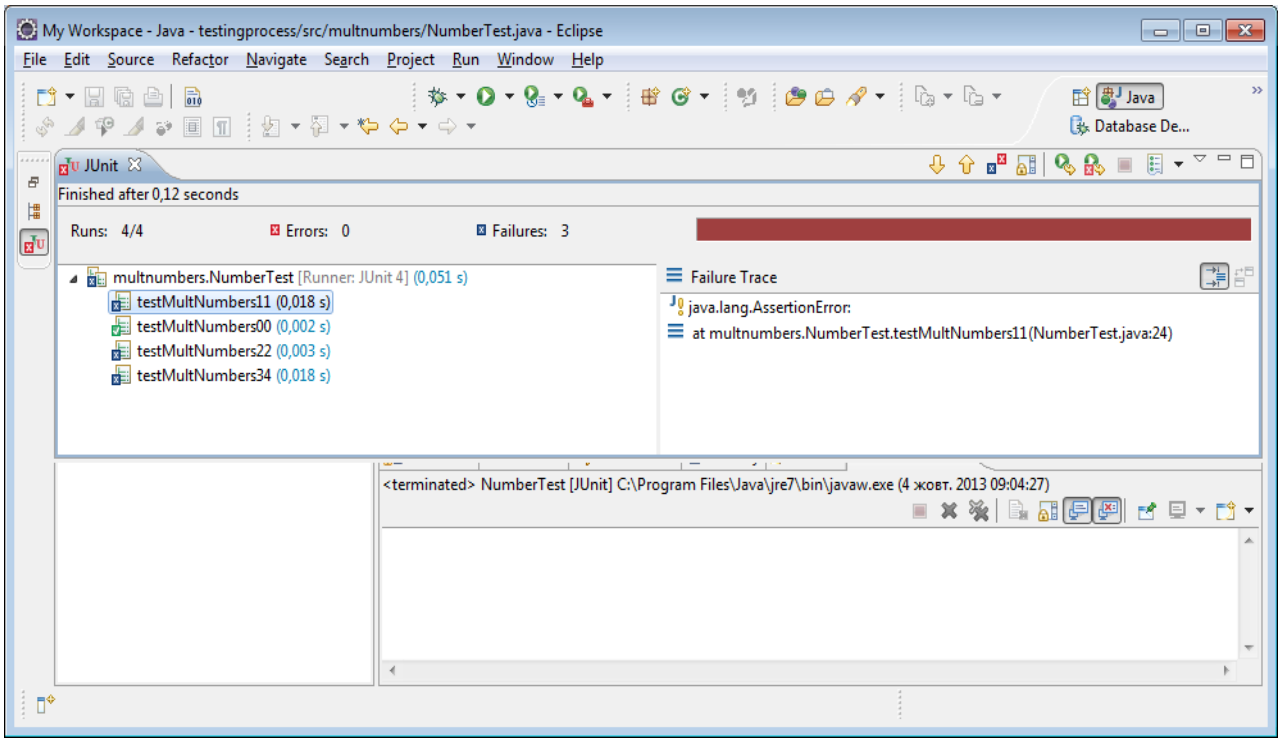
Напишемо інші тестові методи. Весь тестовий клас стане таким:

```
    package multnumbers;

    import static org.junit.Assert.*;

    import org.junit.Assert;
    import org.junit.Test;

    public class NumberTest {

     //Вхід: 0 0; результат - 0.
     //Вхід: 1 1: результат - 1.
     //Вхід: 2 2; результат - 4.
     //Вхід: 3 4; результат - 12.


     MultNumbers mn = new MultNumbers();

     @Test
     public void testMultNumbers00() {
          Assert.assertTrue(mn.multNumbers((byte)    0,    (byte)
0)==0);
     }

     @Test
     public void testMultNumbers11() {
          Assert.assertTrue(mn.multNumbers((byte)    1,    (byte)
1)==1);
     }

     @Test
     public void testMultNumbers22() {
          Assert.assertTrue(mn.multNumbers((byte)    2,    (byte)
2)==4);
```

```
    }

    @Test
    public void testMultNumbers34() {
        Assert.assertTrue(mn.multNumbers((byte)    3,    (byte)
4)==12);
    }



}
```

Лістинг 4.2.2.3. Скоригований тестовий клас, в якому різні тестові приклади розділені

Тепер можна запустити тест так само, як ми запускали звичайне застосування (зелений трикутник або пункт меню `Run -> Run`). Повинно з'явитися наступне віконце:



Червона смужка свідчить про те, що тест провалився. Зліва ми можемо бачити, на яких саме тестових прикладах були помилки. У даному випадку правильні результати були отримані тільки для вхідних даних `(0, 0)` – нагадаємо, що замість справжніх обчислень у нас поки що є заглушка, яка завжди повертає 0.

Будемо змінювати код функції. Спробуємо наступний варіант:

```
public byte multNumbers(byte a, byte b) {
      return (byte) (a+b);
 }
```

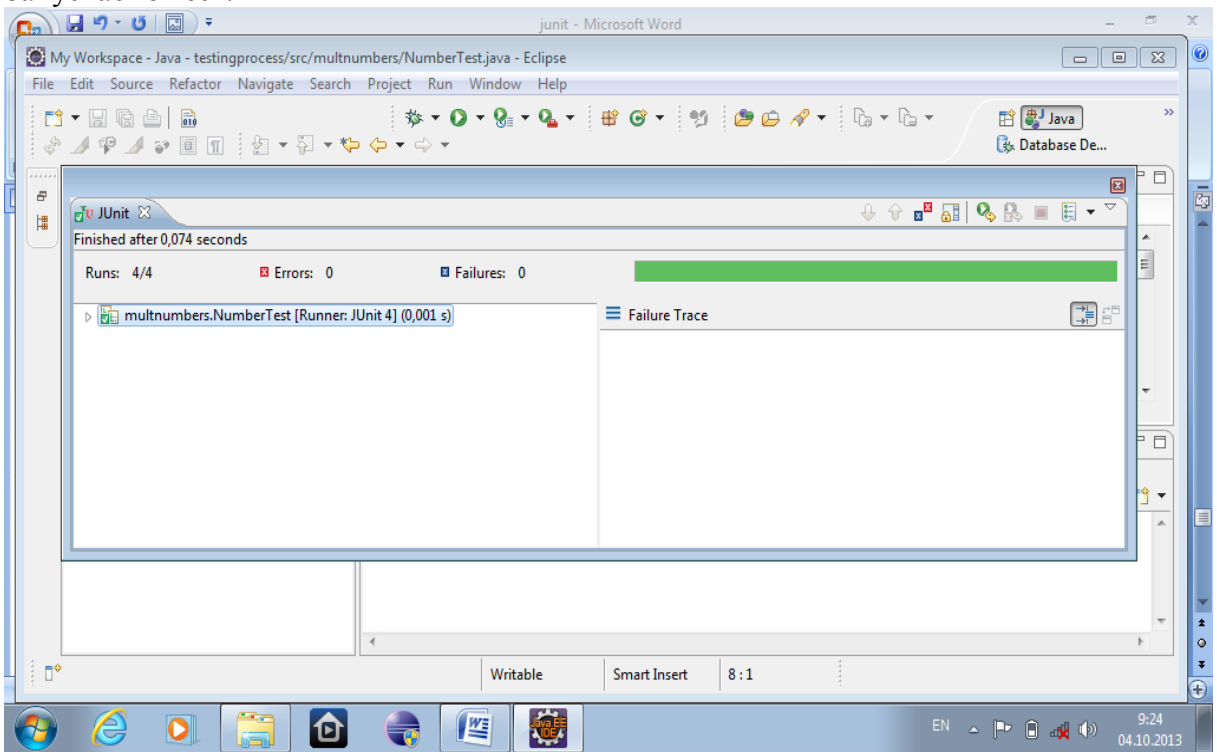
Лістинг 4.2.2.4. Тест для цієї функції провалюється: помилкова операція

Ще раз запустимо тест. Тепер успішними є два тестових приклади. Спробуємо ще один варіант обчислень:

```
public byte multNumbers(byte a, byte b) {
      return (byte) (a*b);
 }
```

Лістинг 4.2.2.5. Для цієї функції тест успішний

Запускаємо тест.



Ура! Зелена смужка свідчить про успішність тесту: помилок на тестових прикладах не було. Але чи доводить це правильність програми?

Як загальне правило – ні, не обов'язково. У даному випадку сам набір тестових прикладів був недостатньо представницьким: всі вони були розраховані лише на маленькі числа. Додамо ще один приклад: спробуємо помножити 10 на 30:

```
@Test
 public void testMultNumbers1030() {
      Assert.assertTrue(mn.multNumbers((byte)    10,    (byte)
30)==300);
 }
```

Лістинг 4.2.2.6. Ще один тестовий приклад; на ньому тест провалюється

На цьому прикладі тест провалюється.

Спробуємо замість типу `byte` працювати з типом `int`. Змінюємо код функції:

```
public int multNumbers(int a, int b) {
      return a*b;
 }
```

Лістинг 4.2.2.7. Скоригована функція множення; тест успішний

Разом з цим доведеться прибрати приведення типів у тестових прикладах. Отримуємо успішне проходження тесту. Ура…

Залишається запитання: а що буде, якщо значення операндів або результат множення вийдуть за рамки типу `int` (чи навіть `long`)? Але це уже дещо інше питання…

Розділ 4.3. Якість та ефективність програм

4.3.1. Поняття про якість програмного забезпечення

При розробці програмного забезпечення, як і при створенні будь-якої іншої продукції, велике значення має питання якості. Серед основних вимог до програм, які зумовлюють якість програмного продукту як з точки зору користувачів, так і з точки зору розробників:

- ***правильніст***ь. Програма повинна видавати правильні результати. Якщо програма не працює, значна частина інших факторів вже не має суттєвого значення;
- ***ефективність***.- міра використання системних ресурсів. Програма принаймні має видавати результати за прийнятний час і не бути надто ресурсоємною;
- ***надійність*** – здатність до виконання необхідних функцій у визначених умовах, стійкість до збоїв;
- ***зручність у використанні.*** Програма повинна бути зручною в засвоєнні і використанні, а її інтерфейс – інтуїтивно зрозумілий;
- ***адаптованість*** – здатність до використання програми без змін в умовах, на які вона не була розрахована безпосередньо;
- ***зручність супроводу*** – можливість легкого внесення змін для розширення можливостей програми, виправлення дефектів тощо;
- ***гнучкість*** – легкість зміни коду програми для використання в умовах, на які вона не була безпосередньо розрахована;
- ***можливість повторного використання*** – легкість використання окремих частин програми в інших програмах і середовищах;
- ***читабельність*** – легкість читання програмного коду.

Якість об’єктно-орієнтованого програмного проекту в першу чергу визначається структурою класів та зв’язків між ними. Продуманість або, навпаки, непродуманість цієї структури справляє вирішальний вплив на зручність супроводу, зручність у використанні, читабельність тощо. Але все це ніяк не применшує ролі ефективності, якій і присвячений цей розділ.

4.3.2. Основні показники ефективності

Ефективність програми визначається на основі наступних критеріїв:

- **час виконання програми**, швидкодія;
- **кількість пам’яті**, необхідна для виконання програми;
- використання інших системних ресурсів.

Ефективність програми в першу чергу залежить від ефективності алгоритму, але його реалізація теж відіграє значну роль.

Для кількісного вимірювання ефективності існує багато засобів моніторингу та профілювання – зокрема, ***профілювальники***, які включені до складу IDE або можуть бути підключені до них як плагіни. Але існують прості і загальновживані засоби вимірювання ефективності окремих фрагментів програми прямо в її коді.

4.3.3. Дослідження часу виконання фрагментів коду

Найпростіший спосіб визначити час виконання того чи іншого фрагменту програми – скористатися функцією **System.nanotime().** Ця функція повертає кількість наносекунд, яка пройшла з деякого випадкового моменту часу.

Ідея хронометрування для дослідження часу виконання того чи іншого фрагменту коду полягає в наступному. Перед його початком за допомогою виклику

```
long t1=System.nanoTime();
```

фіксується одна часова мітка, а одразу ж після завершення цього фрагменту – друга. Тоді різниця між цими фрагментами і буде часом виконання коду, що досліджується.

Часто зручніше виводити не кількість наносекунд, а кількість більш великих відрізків часу – мікросекунд або мілісекунд.

Описаний підхід можна проілюструвати наступним кодом:

```
long t1=System.nanoTime();
   //Код, який досліджується
long t2 = System.nanoTime();
long elapsedTime = t2-t1;
System.out.println("Elapsed time was "+elapsedTime+" ns");
```

Лістинг 4.3.3.1. Хронометрування за допомогою System.nanoTime()

Для прикладу визначимо час, потрібний для обчислення суми деякої кількості членів гармонійного ряду:

```
import java.util.Scanner;

public class DemoGarmonic {

 public static void main(String[] args) {
      System.out.println("Enter the number of members:");
      Scanner sc = new Scanner(System.in);
      final int n = sc.nextInt();
      double s=0;
      long t1=System.nanoTime();

      for (int i=1; i<=n; i++) {
           s+=1./i;
      }

        long t2 = System.nanoTime();
        long elapsedTime = (t2-t1)/1000;
        System.out.printf("Sum of %d members of harmonic
series is%7.3f\n",n,s);
        System.out.println("Elapsed time was "+elapsedTime+"
mks");


 }

}
```

Лістинг 4.3.3.2. Вимірювання часу обчислення членів гармонійного ряду

Якщо введена кількість членів дорівнює 100, програма може вивести такі результати:

```
Sum of 100 members of harmonic series is  5,187
Elapsed time was 43 mks
```

З аналогічною метою часто використовується `System.currentTimeMillis()`. На відміну від `System.nanoTime()`, ця функція повертає кількість мілісекунд, які минули від півночі 1 січня 1970 р.

Останнім часом набув популярності клас `StopWatch`, для використання якого потрібно підключити бібліотеку `Spring.Framework` (пакет **`org.springframework.util`**). Відповідний код може мати вигляд:

```
StopWatch sw = new StopWatch();
        sw.start();
        //Estimated code
        sw.stop();
        System.out.println("Elapsed      time:      "      +
sw.getLastTaskTimeMillis() + ” ms”);
```

Для початку відліку часу використовується функція `start()`, для завершення – `stop()`. Функція `getLastTaskTimeMillis())` повертає час, що минув. Функції `shortSummary()` та `prettyPrint()` повертають більш повний опис задачі, яка виконувалася.

Проілюструємо використання `StopWatch` для задачі про суму гармонійного ряду. Програма може бути такою:

```
package demostopwatch;

import java.util.Scanner;
import org.springframework.util.StopWatch;

public class Demostopwatch {

    public static void main(String[] args) {
        System.out.println("Enter the number of members:");
        Scanner sc = new Scanner(System.in);
        final int n = sc.nextInt();
        double s = 0;

        StopWatch sw = new StopWatch();
        sw.start("Harmonic Series");
        for (int i = 1; i <= n; i++) {
            s += 1. / i;
        }

        sw.stop();
        System.out.printf("Sum  of  %d  members  of  harmonic
series is %7.3f\n", n, s);
        System.out.println("Elapsed      time      "      +
sw.getLastTaskTimeMillis());
        System.out.println(sw.shortSummary());
        System.out.println("*******");
        System.out.println(sw.prettyPrint());
    }
}
```

Лістинг 4.3.3.3. Використання StopWatch

При n=10000 виведення може мати вигляд:

```
Sum of 10000 members of harmonic series is   9,788
Elapsed time 3
StopWatch '': running time (millis) = 3
*******
StopWatch '': running time (millis) = 3
----------------------------------------
ms     %     Task name
----------------------------------------
00003  100%  Harmonic Series
```

4.3.4. Приклад: порівняння двох способів отримання рядка з примітивного типу

Як отримати рядок зі змінної примітивного типу? Наприклад, якщо у нас є змінна

```
int a=2;
```

може виникнути потреба отримати рядок з відповідним значенням, тобто 2.

Найбільш відомими є два способи вирішення цієї задачі:

- операція **+: ""+a**;
- виклик функції **String.valueOf(a);** це статична функція класу String.

Порівняємо ці два способи за швидкодією з використанням описаної вище методики. Для цього будемо створювати рядок на кожній ітерації достатньо довгого циклу.

1-й спосіб. Відповідна програма має вигляд

```
public static void main(String[] args) {
      final int N = 100000;

      int a = 2;
      long t1 = System.nanoTime();

      for (int i=0; i<N; i++) {
           String q=""+a;
      }

      long t2 = System.nanoTime();
      System.out.println("Elapsed   time:   "+(t2-t1)/1000+"
mks");

 }
```

Лістинг 4.3.4.1. Дослідження часу отримання рядка зі змінної примітивного типу на основі операції +

Різні запуски давали різний час виконання циклу; він коливався в діапазоні 70 000 – 100 000 мкс. Але, якщо замість змінної ми брали літеральне примітивне значення:

```
public static void main(String[] args) {
      final int N = 100000;
```

```
        long t1 = System.nanoTime();

        for (int i=0; i<N; i++) {
             String q=""+2;
        }
        long t2 = System.nanoTime();
        System.out.println("Elapsed   time:   "+(t2-t1)/1000+"
mks");

     }
```

Лістинг 4.3.4.2. Дослідження часу отримання рядка з літералу примітивного типу на основі.операції +

час виконання був значно меншим і коливався в діапазоні 3000-6000 мкс.

Спосіб 2. Тепер будемо отримати рядок зі змінної за допомогою `String.valueOf()`.

```
    public static void main(String[] args) {
        final int N = 100000;

        int a=2;

        long t1 = System.nanoTime();

        for (int i=0; i<N; i++) {
             String q=String.valueOf(a);
        }
        long t2 = System.nanoTime();
        System.out.println("Elapsed   time:   "+(t2-t1)/1000+"
mks");

     }
```

Лістинг 4.3.4.3. Дослідження часу отримання рядка зі змінної примітивного типу на основі String.valueOf()

Час виконання циклу складав 30000 – 40000 мкс, тобто менше, ніж аналогічного циклу при використанні операції +. Але тепер заміна змінної на літерал практично не змінила ситуації.

4.3.5. Приклад: порівняння інтерпретації та синхронної компіляції

Ми уже казали, що JIT-компіляція (або синхронна компіляція), яка здійснюється віртуальною машиною, дозволяє досягти суттєвого скорочення часу виконання програм порівняно з традиційною інтерпретацією. Спробуємо застосувати техніку хронометрування програм, описану в попередньому розділі, для експериментального дослідження цього питання.

Схема експерименту полягає в наступному.

1. Напишемо більш-менш складний цикл, бажано достатньо складний. Забезпечимо хронометраж програми (або самого циклу).

2. Компілюємо програму.

3. Виконуємо програму звичайним чином (це буде комбінований режим з переходом до синхронної компіляції).

4. Виконуємо програму в режимі чистої інтерпретації:

```
java –Xint клас
```

5. Порівнюємо результати.

Експеримент 1.

Порівняно простий цикл (але з заплутаними виразами):

```
public class explorecycle {

    public static void main(String[] args) {

        int REPEATS = 100000;
int q;
        long t1;
        long t2;
double elapsed;

        int s=0;



        t1=System.nanoTime();

        for (int i = 0; i < REPEATS; i++) {
q=1;
q=q+i;
q=q-i;
s=(i+1)*(q+i-i)*(q+q*i-q*i);
        }

t2=System.nanoTime();

elapsed = ((double)t2-t1)/1000.;

              System.out.println("Elapsed time is "+elapsed+"
microsecondss");
              System.out.println("Result is "+s);

}
}
```

Лістинг 4.3.5.1. Дослідження ефективності та синхронної компіляції: деякий надуманий цикл

Було проведено три експерименти (час дано в мікросекундах)

№	JIT-компіляція	Інтерпретація
1	11281	37476

2	11475	42976
3	11228	37581

Як бачимо, досягається пришвидшення приблизно в 3,5 разів.

Експеримент 2

Для того ж коду зменшимо кількість ітерацій циклу (візьмемо REPEATS=1000).

№	JIT-компіляція	Інтерпретація
1	391,7	421,2
2	421,8	379,9
3	378,7	376,3

Ми бачимо, що інтерпретація може працювати навіть швидше, ніж синхронна компіляція. Це можна пояснити: спочатку віртуальна машина все одно працює в режимі інтерпретації. Для переходу в режим JIT-компіляції їй потрібно набрати статистику, а це пов'язано з певними накладними витратами.

4.3.6. Поняття про Java Code Conventions

Документ **Java Code Conventions**, який можна завантажити з сайту Oracle, містить ряд "канонічних" вимог до Java-коду. Всі програмісти, які пишуть програми на Java, мають знати ці вимоги та дотримуватися їх.

Багато рекомендацій Java Code Conventions носять загальний характер. Як приклад наведемо сакраментальні заклики щодо необхідності використання коментарів або щодо того, що назви змінних, функцій тощо мають бути осмисленими і відображати їх призначення. Нехтування цим принципом різко знижує читабельність програми. Дійсно, наступна функція для обчислення пройденого шляху за відомими швидкістю і часом руху

```
duuble calculateDistance(double speed, double time) {

return speed*time;

}
```

виглядає набагато краще, ніж функція

```
double a(double b, double c){

return b*c;

}
```

Крім того, **Java Code Conventions** містить ряд вимог, специфічних для Java. Перелічимо деякі з них.

Імена функцій мають починатися з маленької літери – calculate(), а не Calculate(). Якщо ж ім'я є складним і складається з кількох слів, перше слово має починатися з маленької літери, а всі наступні – з великої (так званий ***camel-стиль***): наприклад, **calculateAnnualIncome().** Ті ж правила стосуються імен змінних.

Для імен функцій краще використовувати дієслова. Так, ім'я **makeReport()** є кращою назвою для функції, ніж **myReport().**

Назви класів слід починати з великої літери (наприклад, **Entity**). Якщо ж назва класу складається з кількох слів, з великої літери має починатися кожне слово (наприклад, **MyWonderfulEntity**).

Імена констант слід писати великими літерами. Наприклад:

```
final int NAME = “Bob”;
```

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Охарактеризуйте поняття відлагодження програм.
2. Які Ви знаєте способи відлагодження програм.
3. Які Ви знаєте недоліки звичайне виведення проміжних результатів з метою відлагодження?
4. Як можна відімкнути проміжне виведення при протоколюванні з використанням **`java.util.logging`**?
5. Яким чином можна відстежувати хід виконання програми та зміну змінних в середовищі Eclipse?
6. Охарактетизуйте поняття тестування програм.
7. Що таке модульне тестування?
8. Охарактеризуйте поняття Test Driven Development.
9. Опишіть відомі Вам підходи до здійснення тестування.
10. Що таке тестові приклади?
11. Чи завжди успішне проходження всіх тестів доводить правильність програми?
12. Опишіть використання JUnit для тестування.
13. Які Ви знаєте вимоги до програмного забезпечення?
14. Перелічіть основні показники, які визначають ефективність програм.
15. Опишіть, як можна використовувати функцію ***System.nanoTime*** для вимірювання часу виконання програм.
16. Перелічіть основні вимоги Java Code Conventions щодо назв класів, методів, змінних та констант.

ВПРАВИ НА ПРОГРАМУВАННЯ З JAVA

Найпростіші консольні застосування

Hello, World! Написати консольне застосування, яке виводить на екран деякий текст (наприклад, `Hello, world!`). Вправу виконати в двох варіантах:
- текст, що виводиться, має бути літералом в операції виведення;
- текст, що виводиться, має бути полем основного (і єдиного) класу.

Привітання. Написати консольне застосування, яке вводить з клавіатури ім'я користувача та виводить на екран привітання цьому користувачеві. Введення виконати у двох варіантах:
- з використанням класу `BufferedReader`;
- з використанням класу `Scanner`;
- з використанням класу `JOptionPane`

Робота з числами. Написати консольне застосування, яке вводить з клавіатури дійсне число і виводить на екран двійковий логарифм цього числа. Введення виконати у двох варіантах:
- з використанням класу `BufferedReader`;
- з використанням класу `Scanner`.

Виведення теж виконати у двох варіантах:
- у `System.out.println` використати рядкову конкатенацію;
- використати форматоване виведення (функцію `System.out.printf`).

На прикладі обчислення логарифмів проілюструвати статичний імпорт.

Аргументовані команди. Написати консольне застосування, яке підраховує суму дійсних чисел, заданих як аргументи командного рядка. Програма має контролювати коректність введення і у випадку, якщо аргументів командного рядка немає або вони мають неправильний формат, видавати відповідне повідомлення. Програму запускати як у командному рядку, так і засобами IDE.

Паролі з консолі. Написати консольне застосування, яке з використанням класу `Console` вводить логін та пароль користувача і, якщо вони правильні, виводить You are welcome, а в іншому випадку - `Access denied`.

Сума натуральних чисел. Написати функцію, яка знаходить суму перших n натуральних чисел; число n є параметром функції.

Добуток дробів. Написати та протестувати функцію, яка обчислює добуток

$$\prod_{i=1}^{n} \frac{i}{i+1}$$

число $n>0$ має бути параметром функції.

Інтерактивне керування. Написати програму, яка виводить на екран поточне значення деякої цілої змінної та дозволяє її змінити наступним чином. Користувач має ввести з клавіатури один з символів:

i - збільшити значення змінної на 1;

d - зменшити значення на 1;

u – відновити значення за замовченням (нехай це значення дорівнюватиме 10);
q – завершити виконання програми.

Коректність введення не гарантується. Якщо користувач ввів неправильні дані, йому повинно бути виведено відповідне повідомлення.

Рекордні рядки. Написати консольне застосування, яке вводить з клавіатури деяку кількість рядків та виводить на екран найдовший і найкоротший з них, а також довжини цих рядків.

Основні типи даних та операції з ними

Двійкове подання. Написате функцію/функції, які дозволяють отримати внутрішнє двійкове подання для змінних всіх примітивних типів, крім `boolean`.

Сума цифр. Написати консольне застосування, яке вводить з клавіатури ціле невід'ємне число, кількість розрядів якого гарантовано не перевищує 200, та виводить на екран суму його цифр.

Довгі цілі. Написати консольне застосування, яке вводить з клавіатури два великих цілих числа і знаходить їх суму. Використати клас `BigInteger`.

Обчислення факторіалів. Вивести на екран таблицю значення факторіалів `n!` від 1 до максимального цілого числа, для якого факторіал обчислюється точно. Для контролю точності використати клас `BigInteger`. Порівняти результати обчислень для випадків, коли типом результату є `int, long` та `double`. Пояснити, чому використання типу double не дає суттєвого виграшу порівняно з типом `long`.

.

Скільки літ, скільки зим. Написати програму, яка вводить з клавіатури два моменти часу та підраховує, скільки між ними пройшло днів.

Основні алгоритмічні конструкції

Числа Фібоначчі. Написати функцію, яка приймає ціле число та повертає число Фібоначчі з цим номером. Послідовність чисел Фібоначчі обчислюється наступним чином: перші два члени дорівнюють 1, а кожне наступне дорівнює сумі двох попередніх. Вправу виконати в двох варіантах:
- нерекурсивний;
- рекурсивний.

Дослідити час виконання програми. Подбати про максимально можливе зниження часових затрат (особливо для рекурсивного варіанту).

Наш синус. Написати функцію, яка обчислює синус дійсного аргументу на основі розкладу в ряд Тейлора. Стандартні функції використовувати лише для тестування роботи функції.

Половинне ділення. Написати функцію, яка знаходить на відрізку [a,b] корінь рівняння f(x)=0 методом половинного ділення. Вважається, що функція f(x) гарантовано є або монотонно зростаючою, або монотонно спадною, та має єдиний корінь на вказаному відрізку.. Кінці відрізку та точність розв;язку мають бути параметрами функції.

Решето Ератосфена. Розглядається задача: на основі решета Ератосфена знайти всі прості числа, що не перевищують заданого числа, та виводить їх на екран; при цьому використовуються масиви. Дано такий варіант програми:

```
import java.util.Arrays;

public class Eratosfen {

     public static void main(String[] args) {
          final int n = 100;
          boolean[] flags = new boolean[n + 1];
          Arrays.fill(flags, false);

          int curr = 2;

          while (curr < Math.sqrt(n)) {

               for (int k = 2; k <= (n / curr); k++) {
                    flags[curr*k] = true;

               }

               curr++;
          }



          for (int i = 2; i <= n; i++) {
               if (!flags[i])
                    System.out.print(i + " ");
          }
          System.out.println("\n");

     }

}
```

Дослідити час виконання основного циклу. Модифікувати програму так, щоб максимально зменшити часові затрати на його виконання

Швидке піднесення. Написати функцію, яка підносить натуральне число до цілого ступеня *n*, здійснюючи при цьому суттєво менше ніж *n* множень. Використати метод інваріантів.

Геометрична прогресія. Написати програму, яка вводить з клавіатури `n` цілих чисел, перевіряє, чи утворюють вони геометричну прогресію, і якщо так – виводить на екран коефіцієнт прогресії. Число `n` вводиться з клавіатури.

Золота середина. Написати функцію, яка приймає як аргументи три цілі числа та повертає середнє з них. Наприклад, для аргументів `25, 2, 30` вона має повернути `25`.

Масиви та рядки

Послідовність цілих. Написати програму, яка формує послідовність цілих додатних чисел, які вводяться з клавіатури. Кількість чисел заздалегідь невідома; ознакою кінця послідовністю є введення 0. Якщо вводиться від'ємне число або щось, що не є цілим числом, програма має попросити ввести правильні дані..
Вправу виконати в двох варіантах:
- з використанням масивів;
- з використанням ArrayList.

Обернений масив. Написати функцію, яка приймає масив цілих чисел як аргумент та повертає масив, у якому елементи вхідного масиву розташовані в зворотному порядку.

Багатовимірний масив. Написати функцію, яка приймає змінну кількість рядків та повертає багатовимірний масив, елементами якого в свою чергу є масиви типу `int[],` що складаються з цифр кожного рядка і відсортовані за спаданням. Наприклад, при вхідних даних "`q2r14gf`", "`5w9`" функція повинна повернути масив `[[4,2,1],[9,5]].` Проілюструвати роботу функції в консольному застосуванні.

Масиви – геть! Переробити вправу "Багатовимірний масив" так, щоб функція повертала не багатовимірний масив, а колекцію відповідного типу.

Агреговані стовпчики. Ми бажаємо інтерпретувати двовимірний масив як таблицю, вирівняну по лівому краю. Наприклад, масивові

```
int[][] arr = {
{11, 12, 13},
{21, 22};
{31},{32},{33},{34}
};
```

відповідає таблиця

11	12	13	
21	22		
31	32	33	34

Напишіть функцію, яка приймає як аргумент двоовимірний масив та повертає колекцію ArrayList, кожний елемент якого дорівнює кількості заповнених клітинок у відповідному стовпчику таблиці. Наприклад, у нашому випадку функція має повернути колекцію `{3, 3, 2, 1}`

Системи рівнянь. Написати функцію, яка розв'язує систему лінійних алгебраїчних рівнянь; функція повинна приймати як аргументи двовимірний масив коефіцієнтів типу double[][] та одновимірний масив вільних членів типу double[] і повертати розв'язок у вигляді одновимірного масиву дійсних чисел.

Паліндром. Написати програму, яка вводить з клавіатури рядок і перевіряє, чи є він паліндромом, тобто чи читається він однаково зліва направо та справа наліво.

Сортування рядків. Написати функцію, яка приймає як параметр рядок та повертає рядок, відсортований за символами в зворотному порядку. Наприклад, при аргументі

qwerty функція має повернути *ywtqre*. Перевірити роботу написаної функції в основній програмі.

Пакети

Робота з пакетами. Створити пакет `mycompany.training.<прізвище>`, до якого включити кілька довільних класів. Упакувати пакет до jar-архіву. Написати консольне застосування, яке ілюструє використання як неупакованого пакету, так і упакованого jar-файлу.

Основні прийоми ООП

Канонічний клас. Написати клас `MyClass` з полем `a` типу `int`. З кожним екземпляром класу має пов'язуватися його порядковий номер. З цією метою в класі потрібно передбачити поле для номера, а в конструкторі - встановлення номера.
Забезпечити наявність необхідних конструкторів, а також можливість виведення екземплярів класу на екран. При цьому дотримуватися принципів інкапсуляції.
Створити колекцію `ArrayList`, яка має складатися з кількох екземплярів цього класу, та вивести її вміст на екран. Кожний екземпляр має виводитися у вигляді: `номер – a`.

Перелік лістингів програм

3.1.16.3. Порівняння рядків за допомогою ==
3.1.16.4. Порівняння рядків за допомогою методу equals
3.2.1.1. Простий клас
3.2.1. 2. Демонстраційна програма, яка використовує клас
3.2.2.1. Клас з селекторами і модифікаторами
3.2.2.2. Поля закриті – помилка компіляції
3.2.2.3. Для отримання значень полів використовуємо селектори
3.2.3.1. Типовий вигляд конструктора з параметрами
3.2.4.1. Спадкування методів підкласами
3.2.4.2. Перевизначення методів у підкласі
3.2.5.1. Суперклас з методом main
3.2.5.2. Клас, який спадкує від суперкласу з методом main()
3.2.7.1. Елементарний клас для демонстрації методів equals() та toString()
3.2.8.1. Перевірка рівності об'єктів: використання оператора ==
3.2.8.2. Перевірка рівності об'єктів: використання функції equals()
3.2.8.3. Перевизначення методу equals
3.2.9.1. Спроба вивести на екран екземпляр класу
3.2.9.2. Перевизначення методу toString()
3.2.11.1. Автоматична генерація елементів класу: перший крок
3.2.11.2. Згенерований код методу equals()
3.2.11.3. Остаточний код класу з автоматично згенерованими основними методами
3.2.12.1. Інтерфейс з одним методом
3.2.12.2. Неправильна імплементація інтерфейсу: необхідний метод не реалізований
3.2.12.3. Правильна імплементація інтерфейсу
3.2.13.1. Приклад перелічуваного типу
3.2.13.2. Створення та використання перелічуваних типів
3.2.14.1. Помилка компіляції: відсутність обробки контрольованих виключень
3.2.14.2. Використання ключового слова throws
3.2.14.3. Обробка виключень в блоці try-catch
3.2.14.4. Введення числа: багато можливих виключень
3.3.3.1. Низькорівнева реалізація списку на C++ з використанням указників
3.3.3.2. Низькорівнева реалізація списка на Java
3.4.2.1. Літеральне задання та перебір масиву
3.4.3. 1. Приклад сортування масиву
3.4.3.2. Сортування масиву рядків з використання ламбда-функцій
3.4.4.1. Масив формується випадковим чином
3.4.5.1. Екземпляри цього класу будуть елементами масиву
3.4.5.2. Програма, яка працює з масивом екземплярів
3.4.6.1. Заповнення та виведення масиву: неправильне використання модифікованого for
3.4.6.2.Заповнення та виведення масиву: неправильний розмір масиву
3.4.6.3. Заповнення та виведення масиву: виправлений варіант
3.4.7.1. Заповнення та виведення колекції
3.4.8.1. Виведення двовимірного масиву
4.1.2.1. Початковий неправильний варіант підрахунку суми натуральних чисел
4.1.2.2. Перша спроба виправлення – виведення проміжних результатів
4.1.2.3. Виправлення заголовку циклу
4.1.2.4. Не та сума
4.1.2.5. Підрахунок суми натуральних чисел: виправлений варіант
4.1.3.1. Виведення проміжних результатів за допомогою класу Logger
4.1.3.2. Відключення протокольних повідомлень
4.1.4.1. Програма для демонстрації можливостей Eclipse для відлагодження

4.2.2.1. Функція обчислення добутку: початковий варіант

4.2.2.2. Клас для модульного тестування, згенерований JUnit

4.4.2.3. Скоригований тестовий клас, в якому різні тестові приклади розділені

4.2.2.4. Тест для цієї функції провалюється: помилкова операція

4.2.2.5. Для цієї функції тест успішний

4.2.2.6. Ще один тестовий приклад; на ньому тест провалюється

4.2.2.7. Скоригована функція множення; тест успішний

4.3.3.1. Хронометрування за допомогою System.nanoTime()

4.3.3.2. Вимірювання часу обчислення членів гармонійного ряду

4.3.3.3. Використання StopWatch

4.3.4.1. Дослідження часу отримання рядка зі змінної примітивного типу на основі операції +

4.3.4.2. Дослідження часу отримання рядка з літералу примітивного типу на основі.операції +

4.3.4.3. Дослідження часу отримання рядка зі змінної примітивного типу на основі String.valueOf()

4.3.5.1. Дослідження ефективності та синхронної компіляції: деякий надуманий цикл