МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Кафедра «Інженерія програмного забезпечення»

Автор: **Тихонов Євгеній Сергійович,** старший викладач, аспірант

НАВЧАЛЬНО-МЕТОДИЧНІ МАТЕРІАЛИ
до вивчення дисципліни
«КОНСТРУЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. JAVA»

Київ-2018

Даний методичний збірник піддійте будь-якій людині, котра прямує в напрямку розвитку IT сфери. В виданні висвітлюються базові поняття основ конструювання програмного забезпечення. Також людина використовуючи цій збірник зможе навчитися язику програмування високого рівня – Java. Будуть розглядатись приклади щодо якісного стилю програмування.

ЗМІСТ

ВСТУП

Давно не секрет, що сфера інформаційних технологій є одним із головних драйверів економіки нашої країни. Вона займає третій рядок за обсягом експортної виручки і поступається за цим параметром лише таким «класичним» індустріям, як агро і металургія. Тому для студента напрямку «**Інженерія програмного забезпечення**» повинні входити поняття про методи ефективного та оптимального в певному сенсі кодування алгоритмів в першу чергу на мовах високого рівня.

В 2018 році програміст повинен генерувати не просто будь-який код, який працює, а і обов'язково володіти якісним стилем програмування, при чому якісно, а також методами документування, застосовувати методи мінімізації коду, проводити ефективний пошук помилок, зокрема не явних, на етапі відладки та вміти якісно тестувати власний програмний продукт.

Розгляданню саме цих питань і присвячені навчально-методичні матеріали, в яких представлений матеріал, що направлений на набуття насамперед практичних навичок ефективного програмування.

Ми спочатку розглянемо:

- Що таке КПЗ.
- Базові поняття в Java, достатні для подальшого самостійного розвитку розробника.
- Рефакторинг
- Навички тестування
- UML діаграми.

1. ОСНОВИ КОНСТРУЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

1.1. Місце КПЗ в життєвому циклі програмної системи.

Розробка програмного забезпечення (ПЗ) – це складний процес, в який входить багато складових. В загальному випадку це:

- визначення проблеми;
- вироблення вимог;
- створення плану конструювання;
- розробка архітектури ПЗ, або високорівневе проектування;
- детальне проектування;
- кодування і відлагодження;
- блочне тестування;
- інтеграційне тестування;
- інтеграція;
- тестування системи;
- корегувальне супроводження.

Термін конструювання програмного забезпечення (software construction) описує детальне створення робочої програмної системи за допомогою комбінації кодування, верифікації (перевірки), модульного тестування (unit testing), інтеграційного тестування та відлагодження.

На рис.1 показано місце конструювання як частину кроків серед процесів, що проходять при побудові ПЗ.

Рис.1. Конструювання серед процесів побудови ПЗ

Процеси конструювання зображені всередині сірого еліпсу. Головними компонентами конструювання є кодування та відлагодження, однак воно включає і детальне проектування, блочне тестування, інтеграційне тестування та інші процеси.

Іноді конструювання називають "кодуванням" або "програмуванням". Кодування в даному випадку видається не найкращим терміном, так як воно має на увазі механічну трансляцію розробленого плану в команди мови програмування, тоді як конструювання є зовсім не механічним процесом і часто пов’язане з творчістю та аналізом. Сенс слів "конструювання" та "програмування" досить близький.

Дана область знань пов'язана з іншими областями. Найбільш сильний зв'язок існує з проектуванням (Software Design) і тестуванням (Software Testing). Причиною цього є те, що сам по собі процес конструювання програмного забезпечення зачіпає важливі аспекти діяльності з проектування й тестування. Крім того, конструювання відштовхується від результатів проектування, а тестування (у будь-якій своїй формі) передбачає роботу з результатами конструювання. Досить складно визначити межі між проектуванням, конструюванням і тестуванням, тому що всі вони пов'язані в єдиний комплекс процесів життєвого циклу і, в залежності від обраної моделі життєвого циклу і застосовуваних методів (методології), таке розділення може мати різний вигляд.

Хоча ряд операцій з проектування детального дизайну може відбуватися до стадії конструювання, великий обсяг такого роду проектних робіт відбувається паралельно з конструюванням або як його частина. Це є сутність зв'язку з областю знань "Проектування програмного забезпечення".

У свою чергу, протягом всієї діяльності з конструювання, інженери використовують модульне і інтеграційне тестування. Таким чином пов'язана дана галузь знань з "Тестуванням програмного забезпечення".

У процесі конструювання звичайно створюється більша частина активів програмного проекту – конфігураційних елементів (configuration items). Тому в реальних проектах просто неможливо розглядати діяльність по конструюванню у відриві від галузі знань "конфігураційного управління" (Software Configuration Management).

Так як конструювання неможливе без використання відповідного інструментарію і, ймовірно, дана діяльність є найбільш інструментально-насиченою, важливу роль у конструюванні грає область знань "Інструменти і методи програмної інженерії" (Software Engineering Tools and Methods).

Безумовно, питання забезпечення якості значимі для всіх галузей знань і етапів життєвого циклу. У той же час, код є основним результуючим елементом програмного проекту. Таким чином, явно напрошується і присутній зв'язок обговорюваних питань з областю знань "Якість програмного забезпечення" (Software Quality).

З пов'язаних дисциплін програмної інженерії (Related Disciplines of Software Engineering) найбільш тісний і природний зв'язок даної галузі знань існує з комп'ютерними науками (computer science). Саме в них, звичайно, розглядаються питання побудови та використання алгоритмів і практик кодування. Нарешті,

конструювання стосується і управління проектами (project management), причому, в тій мірі, наскільки діяльність з управління конструюванням важлива для досягнення результатів конструювання.

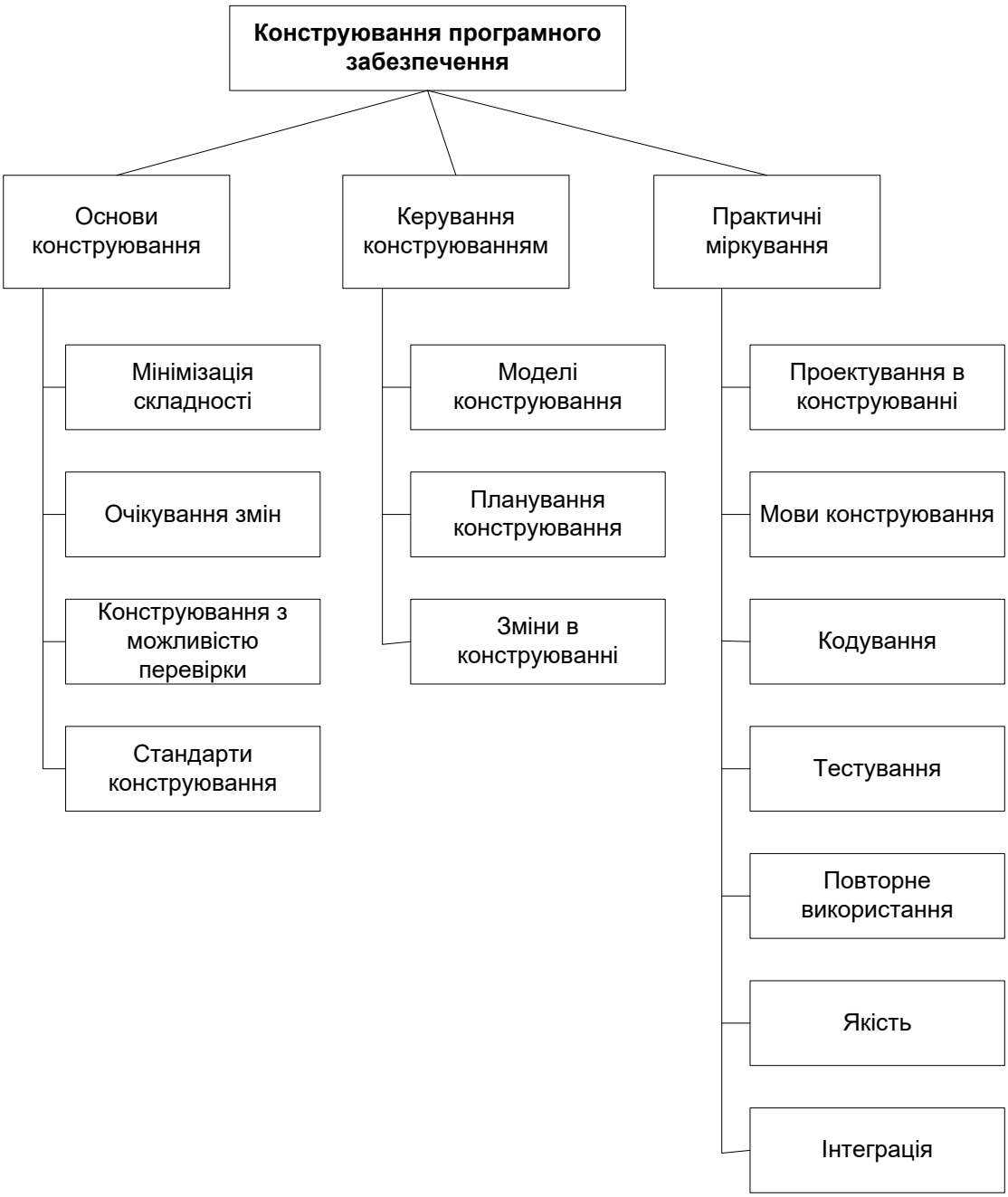

Рис. 2. Область знань "Конструювання програмного забезпечення"

Далі наведено деякі конкретні задачі, що виникають в процесі розробки ПЗ, пов’язані з конструюванням:

- перевірка виконання умов, необхідних для успішного конструювання;
- визначення способів подальшого тестування коду;
- проектування та написання класів та методів;
- створення та присвоєння імен змінних та іменованих констант;
- вибір управляючих структур та організації блоків команд;

- блочне тестування, інтеграційне тестування і відлагодження власного коду;
- взаємний огляд коду та низькорівневих програмних структур членами групи;
- "шліфування" коду шляхом його ретельного форматування та коментування;
- інтеграція програмних компонентів, створених окремо;
- оптимізація коду, яка направлена на підвищення його швидкодії і зниження міри використання ресурсів.

З іншого боку, з видів діяльності, що проходять в процесі розробки ПЗ, до конструювання не відносяться: керування процесом розробки, виробки вимог, розробка високорівневої архітектури програми, проектування інтерфейсу користувача, тестування системи і її супроводження – для кожного з цих пунктів є своя наука.

1.2. Фундаментальні складові конструювання програмного забезпечення.

Фундаментальні основи конструювання програмного забезпечення включають:

- Мінімізація складності
- Очікування змін
- Конструювання з можливістю перевірки
- Стандарти у конструюванні

Перші три концепції застосовуються не тільки до конструювання, але й проектування, і лежать в основі сучасних методологій управління життєвим циклом програмних систем.

1.3. Мінімізація складності (Minimizing Complexity)

Основною причиною того, чому люди використовують комп'ютери в бізнес-цілях, є обмежені можливості людей в обробці і зберіганні складних структур і великих обсягів інформації, зокрема, протягом тривалого періоду часу. Це міркування є однією з основних рушійних сил у конструюванні програмного забезпечення: мінімізація складності. Потреба у зменшенні складності впливає на всі аспекти конструювання і особливо критична для процесів верифікації (перевірки) і тестування результатів конструювання, тобто самих програмних систем.

Зменшення складності у конструюванні програмного забезпечення досягається за допомогою звертання особливої уваги на створення простого коду, який легко читається, іноді на шкоду прагненню зробити його ідеальним (наприклад, з точки зору гнучкості або слідування тим чи іншим уявленням про красу, витонченість коду, вправність тих чи інших прийомів тощо). Це не означає, що повинно обмежуватися застосування тих чи інших розвинених мовних можливостей використовуваних засобів програмування. Мається на увазі "лише" надання більшої значимості читаності коду, простоті тестування,

прийнятного рівня продуктивності та задоволенню заданих критеріїв, замість постійного вдосконалення коду, не оглядаючись на терміни, функціональність і інші характеристики та обмеження проекту.

Мінімізація складності досягається, зокрема, за рахунок слідування стандартам, використанням низки специфічних технік кодування і підтримкою практик, спрямованих на забезпечення якості в конструюванні.

1.4. Очікування змін (Anticipating Changes)

Більшість програмних систем змінюються з плином часу. Причин цьому – безліч. Очікування змін є однією з рушійних сил конструювання програмного забезпечення. Програмне забезпечення не є ізольованим від зовнішнього оточення (як системного, так і з точки зору галузі діяльності, для автоматизації задач і проблем якого воно застосовується). Більш того, програмні системи є частиною середовища, що змінюється, і повинні змінюватися разом з нею, а, іноді, і бути джерелом змін самого середовища.

Очікування змін підтримується рядом технік кодування.

1.5. Конструювання з можливістю перевірки (Constructing for Verification)

"Конструювання для перевірки" (а саме такий сенс закладений в оригінальній назві даної підтеми) припускає, що побудова програмних систем повинна вестися таким чином, щоб сама програмна система допомагала вести пошук причин збоїв, будучи прозорою для застосування різних методів перевірки (і, до речі, внесення необхідних змін), як на стадії незалежного тестування (наприклад, інженерами-тестувальниками), так і в процесі операційної діяльності – експлуатації, коли особливо важлива можливість швидкого виявлення та виправлення помилок що виникають.

Серед технік, спрямованих на досягнення такого результату конструювання:

- Огляд, оцінка коду (code review)
- Модульне тестування (unit-testing)
- Структурування коду для і спільно з застосуванням автоматизованих засобів тестування (automated testing)
- Обмежене застосування складних або важких для розуміння мовних структур

2. СТАНДАРТИ У КОНСТРУЮВАННІ

Стандарти, які безпосередньо застосовуються при конструюванні, включають:

- Комунікаційні методи (наприклад, стандарти форматів документів і оформлення вмісту)
- Мови програмування і відповідні стилі кодування (наприклад, Java Language Specification, що є частиною стандартної документації JDK – Java Development Kit і Java Style Guide, що пропонує загальний стиль кодування для мови програмування Java)
- Платформи (наприклад, стандарти програмних інтерфейсів для викликів функцій операційного середовища, такі як прикладні програмні інтерфейси платформи Windows – Win32 API, Application Programming Interface або .NET Framework SDK, Software Development Kit)
- Інструменти (не в термінах середовищ розробки, але можливих засобів конструювання – наприклад, UML як один зі стандартів для визначення нотацій для діаграм, що представляють структура коду і його елементів або деяких аспектів поведінки коду)

Використання зовнішніх стандартів. Конструювання залежить від зовнішніх стандартів, пов'язаних з мовами програмування, використовуваним інструментальним забезпеченням, технічними інтерфейсами і взаємним впливом конструювання програмного забезпечення та інших галузей знань програмної інженерії (в тому числі, пов'язаних дисциплін, наприклад, управління проектами). Стандарти створюються різними органами, наприклад, консорціумом OMG – Object Management Group (зокрема. Стандарти CORBA, UML, MDA, ...), міжнародними організаціями з стандартизації такими, як ISO/IEC, IEEE, TMF, ..., виробниками платформ, операційних середовищ і т.д. (Наприклад, Microsoft, Sun Microsystems, CISCO, NOKIA, ...), виробниками інструментів, систем управління базами даних і т.п. (Borland, IBM, Microsoft, Sun, Oracle, ...). Розуміння цього факту дозволяє визначити достатній і повний набір стандартів, які застосовуються у проектній команді або організації в цілому.

Кожна програмна система протягом свого існування проходить з певною послідовністю фази або стадії від задуму до його втілення в програми, експлуатацію та вилучення. Така послідовність фаз називається життєвим циклом розробки (Software life cycle processes). На кожній фазі відбувається певна сукупність процесів, кожен з яких породжує певний продукт, використовуючи певні ресурси.

Усі продукти всіх процесів програмної інженерії являють собою певні описи – тексти вимог до розробки, погодження домовленостей, документацію, тексти програм, інструкції з експлуатації тощо.

Головними ресурсами програмної інженерії, які визначають ефективність її розробок, є час і вартість цих розробок.

Різновиди діяльності, котрі становлять процеси життєвого циклу програмної системи, зафіксовано в міжнародному стандарті ISO/IEC 12207 : 1995—0801 : Informational Technology - Software life cycle processes.

Згідно з наведеним стандартом, усі процеси поділено на три групи:

- головні процеси;
- допоміжні процеси;
- організаційні процеси.

До **головних** процесів віднесено такі:

- процес придбання, який ініціює життєвий цикл системи та визначає організацію-покупця автоматизованої системи, програмної системи або сервісу;
- процес розроблення, який означає дії організації — розробника програмного продукту;
- процес постачання, який означає дії під час передавання розробленого продукту покупцеві;
- процес експлуатації, який означає дії з обслуговування системи під час її використання — консультації користувачів, вивчення їхніх побажань тощо;
- процес супроводження, який означає дії з керування модифікаціями, підтримки актуального стану та функціональної придатності, інсталяцію та вилучення версій програмних систем у користувача.

У свою чергу, до процесу розроблення входять такі процеси:

- інженерія вимог до системи;
- проектування;
- кодування й тестування.

До допоміжних процесів віднесено ті, що так чи інакше забезпечують якість продукту. Терміном якість продукту позначено сукупність властивостей, які зумовлюють можливість задоволення потреб замовника, котрий сформулював їх у формі вимог на розробку.

До організаційних процесів віднесено менеджмент розробки, створення структури організації, навчання персоналу, визначення відповідальності кожного з учасників процесів життєвого циклу розробки.

Стандарт ISO/IEC 12207:1995 - 0801: Informational Technology - Software life cycle processes є головним чинником визначення змісту діяльності у сфері програмної інженерії, і всі знання, яких потребують професіонали з програмної інженерії, формулюються стосовно процесів, визначених цим стандартом.

Зупинимося докладніше на процесах розробки програмного забезпечення, які в сукупності мають забезпечити шлях від усвідомлення потреб замовника до передавання йому готового продукту. На цьому шляху виділяють низку характерних робіт:

- Визначення вимог. Збір та аналіз вимог замовника виконавцем і подання їх у нотації, яка є зрозумілою як для замовника, так і для виконавця.
- Проектування. Перетворення вимог до розробки в послідовність проектних рішень щодо способів реалізації вимог: формування загальної архітектури програмної системи та принципів її прив'язки до конкретного середовища функціонування; визначення детального складу модулів кожної з архітектурних компонент.
- Реалізація. Перетворення проектних рішень на програмну систему, яка реалізує такі рішення.
- Тестування. Перевірка кожного з модулів та способів їхньої інтеграції; тестування програмного продукту в цілому (так звана верифікація); тестування відповідності функцій працюючої програмної системи вимогам (requirements), поставленим до неї замовником (так звана валідація).
- Експлуатація та супроводження готової програмної системи.

3. ЗНАЙОМСТВО З МОВОЮ ПРОГРАМУВАННЯ JAVA

3.1. Що таке Java?

Java – Це острів Ява в Малайському архіпелазі, територія Індонезії. Деякий сорт кофе, який полюбляють пити творці Java. А якщо серйозно, то відповісти на це питання досить важко, тому що границі Java, і без того розмиті, увесь час розширюються.

Java (вимовляється Джава; у нас інколи ви можете почути Ява, але це не зовсім правильно, тому краще Джава) – об'єктно-орієнтована мова програмування, випущена компанією **Sun Microsystems** у 1995 році, як основний компонент платформи Java.

Синтаксис мови багато в чому походить від C та C++. Java програми виконуються у середовищі **віртуальної машини** Java. Java програми компілюються у **байткод**, який при виконанні інтерпретується віртуальною машиною для конкретної платформи.

В 2009 році Sun Microsystems придбала компанія Oracle, яка продовжує розвивати Джаву.

Java дозволяє створювати самодостатні програми для різних операційних системах, наприклад Windows, Linux тощо.

Крім того, в даний час Java широко застосовується для програмування різних пристроїв, наприклад, мобільних телефонів, на ній також пишуться комп'ютерні ігри для них, створюють також програми для інтернету, програми для серверів – Сервлети та JSP (Java Server Pages).

Мова створювалася на основі наступних 5 принципів проектування:

1. **Проста,** об’єктно-орієнтована і звична. Java містить в собі невелике ядро узгоджених основних понять, які швидко засвоюються. Спочатку мова моде-лювалася на основі популярної в той час мови C++, тому програмісти могли без зусиль перейти на Java. Крім цього, мова наслідувала парадигмі об’єктної орієнтованості: системи складаються з інкапсульованих об’єктів, які взаємодіють шляхом передачі один одному повідомлень.
2. **Надійна і безпечна.** У мову включені перевірки на етапі компіляції та виконання, що забезпечує швидке виявлення помилок. Крім цього до мови включені функції безпечного доступу до мережі і файлів, щоб робота розподілених додатків не могла бути порушена в результаті вторгнення або пошкодження.
3. **Нейтральна до архітектури і кросплатформенна.** Однією з основних переваг Java є її переносимість. Додатки без зусиль переносяться з однієї платфо-рми на іншу з мінімальними змінами або взагалі без змін. Девіз "Написано один раз – працює скрізь!", що супроводжував випуск Java 1.0 в 1995р., відноситься до міжплатформенних переваг мови.
4. **Високопродуктивна.** Додатки виконуються швидко і ефективно завдяки різним функціям нижнього рівня, таким як можливість роботи інтерпретатора Java незалежно від середовища виконання і використання програми автоматичного чищення пам’яті для звільнення невживаної пам’яті.

5. **Така, що інтерпретується, забезпечує створення потоків, і динамічна.** При використанні Java вихідний код, написаний розробником, компілюється в проміжну форму, що інтерпретується, відому під назвою байт-код. Поняття "набір команд байт-коду" відноситься до машинної мови, яка використовується віртуа-льною машиною Java (JVM — Java Virtual Machine). При наявності відповідного інтерпретатора цю мову можна транслювати в машинний код платформи, на якій він виконується. Можливість створення декількох потоків підтримується, голов-ним чином, засобами класу Thread, що забезпечує одночасну роботу кількох за-вдань. Мова і система етапу виконання є динамічними в тому сенсі, що додатки під час виконання можуть пристосовуватися до змін робочого середовища.

3.2. Найменування різних версій Java

На поточний 2018 рік вийшов уже 10-й випуск Java. Назви випусків подавалися розробниками Java дещо по різному, що створило в літературі деяку плутанину.

Спочатку писали JDK 1.0, JDK 1.1 (JDK – Java Development Kit). Після появи версії 1.2 у випусках java почали в назві писали Java2SE, наприклад, J2SE 1.4 (SE – це Standart Editon, крім того додатково ідуть випуски EE – Enterprise Edition, ME – Mobile Edition).

П'яту ж версію почали писати J2SE 5.0, а далі викинули 2 з назви і почали писати Java SE 6. Різкі зміни в найменуванні передусім пояснювалися ґрунтовними переробками у Java. За історію Java найбільш серйозні зміни були внесені у версію 1.2 відповідно це пояснює появу 2-йки в назвах випусків. Наступний перегляд був у версії 1.5 так звана Java 5.

3.3. Мінімум для програмування на Java

Мінімум необхідного для програмування на Java – це **JDK** (Java Developer Kit – комплект розробника Java) та звичайний текстовий редактор. Щоправда бажано, щоб редактор хоча б здійснював підсвітку синтаксису програми, здійснював компіляцію, запуск та зневадження програми одним натисненням кнопки. Найкращим же варіантом є спеціалізовані середовища розробки (IDE — інтегроване середовище розробки) як то Eclipse, NetBeans або IDEA. При розробці великих проектів інтегровані середовища розробки значно полегшують роботу програмістові і скорочують час на розробку.

3.4. Поняття JDK, JRE, JVM

Що таке JDK?

JDK це набір бібліотек класів, утиліт та документації для програмування на Java. Він складається з кількох компонентів. Ось основні із них:

- компілятор javac. Транслює текст програми на Java в байт-коди віртуальної машини.

- інтерпретатор java. З його допомогою запускаються програми відкомпільовані в байт-код. Містить в собі JVM (Віртуальну машину Java).
- утиліта appletviewer. З її допомогою можна запускати аплети - графічні програми, які виконуються в інтернет браузері. Фактично вона являє собою браузер, який може запускати тільки аплети.
- утиліта javadoc – призначена для створення документації.

Що таке JRE?

Набір програм і пакетів класів JRE містить все необхідне для виконання байт-кодів, в тому числі інтерпретатор java (в попередніх версіях полегшений інтерпретатор jre) і бібліотеку класів. Це частина JDK, яка не містить компілятори, налаштовувачі і інші засоби розробки. Якраз JRE або його аналог інших фірм міститься в браузерах, що вміють виконувати програми на Java, в операційних системах і системах управління базами даних.

Після установки Java ви одержите каталог з назвою, наприклад, jdk1.8, а в ньому підкаталоги:

- bin, який містить виконувані файли;
- demo, який містить приклади програм;
- docs, котрий містить документацію, якщо ви її установили;
- include, котрий містить заголовкові файли "рідних" методів;
- jre, котрий містить набір JRE;
- old-include, для сумісності зі старими версіями;
- lib, котрий містить бібліотеки класів і файли властивостей;
- src, з вихідними текстами програм JDK. В нових версіях замість каталогу знаходиться упакований файл src.jar.

Набір JDK містить вихідні тексти більшості своїх програм, написані на Java. Це дуже зручно. Ви завжди можете з точністю дізнатись, як працює той чи інший метод обробки інформації із JDK, проглянувши вихідний код даного метода. Це дуже корисно і для вивчення Java на "живих" працюючих прикладах.

Що таке JVM?

Java Virtual Machine (JVM) – віртуальна машина Java, частина середовища виконання Java, що виконує інтерпретацію Java байт-коду. Java Virtual Machine специфікується набором команд байт-коду, набором регістрів, стеком, складальником сміття і простором зберігання методів.

Java байт-код (Java bytecode) – машинно-незалежний код, який генерує Java-компілятор. Байт-код виконується Java-інтерпретатором. Віртуальна машина Java є повністю стекової: не потрібно складна адресація комірок пам'яті і велика кількість регістрів. Тому команди JVM короткі, більшість з них має довжину 1 байт, від чого команди JVM називають байт-кодами (bytecodes), хоча є команди довжиною 2 і 3 байти (середня довжина команди складає 1,8 байти).

Програма, написана на мові Java, перекладається компілятором в байт-код. Байт-код записується в одному або декількох файлах, може зберігатися у зовнішній пам'яті або передаватися по мережі. Це особливо зручно завдяки невеликому розміру файлів з байт-кодом.

Отриманий в результаті компіляції байт-код можна виконувати на будь-якому комп'ютері, що має систему, що реалізовує JVM (не залежно від типу якого-небудь конкретного процесора і архітектури ПК). Так реалізується принцип Java: «Write once, run anywhere» – «Написано один раз, виконується де завгодно».

Для написання вашої першої програми необхідно встановити спочатку Java на офіційному сайті, а потім JDK [http://www.oracle.com/technetwork/java/javase/downloads/jdk10-downloads-4416644.html]. Для того, щоб було зручно використовувати JDK потрібно задати змінні середовища і оголосити змінну classpath.

Для перевірки правильності налаштувань і встановлення наберіть наступну програму:

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

Збережіть з розширенням під ім'ям HelloWorld.java. Зверніть увагу, що назва файлу повинна співпадати з назвою класу у ньому.

Відкрийте командний рядок, зайдіть в потрібний вам каталог і наберіть:

- C:\>javac HelloWorld.java

В результаті, якщо не видало помилок, в каталозі з вашою програмою, повинен утворитися файл HelloWorld.class

Для запуску програми наберіть:

- C:\>java HelloWorld

Розширення файлу при запуску вказувати не потрібно. В результаті на екрані з'явиться рядок тексту:

- C:\>java HelloWorld
  Привіт світе!

3.5. Коментарі

Для того, щоб в тексті програми було легше розібратися іншому програмісту або, навіть, згадати, що до чого через значний проміжок часу, під час створення програми корисно в текст вставляти коментарі. Вони не беруться до уваги компілятором, тож на об'єм скомпільованої програми не впливають і їх можна вставляти будь-де.

В Java існує три типи коментарів:

1. **однорядкові** з застосуванням двох похилих рисок (//). Все що йде в рядку після них вважається коментарем.
   // це коментар
   System.out.println("Вивчаємо коментарі"); // виводимо на екран
2. **багаторядкові** — текст можна вказувати в кілька рядків, також зручно, якщо потрібно усунути кілька рядків тексту під час зневадження програми. Для відкриття коментаря використовується /*, а для закриття */. Все що

між ними вважається коментарем. Вкладення багаторядкового коментаря в інший багаторядковий коментар не дозволяється.

```
/*
        Це багаторядковий коментар
        System.out.println("Це не буде виведено на екрані");
        System.out.println("І це також не буде виведено на екрані");
*/
System.out.println("А це буде виведено");
```

3. **для автоматичної генерації документації** – розпочинаються символами **/**** і закінчуються ***/**. Використовуються для додавання додаткової інформації про класи та методи, їх призначення, параметри. На основі даної інформації можна згенерувати документацію по розроблених вами класах та їх методах. Дана документація буде подібною до офіційної документації по класах Java.

3.6. Типи даних

Java – строго типізована мова. Тобто, на відміну від, наприклад, JS в Java потрібно вказувати тип змінної при її оголошенні. У Java існують такі типи даних:

1. *примітивні* типи даних,
2. *посилальні* типи даних.

В Java є вісім основних (примітивних) типів даних. П'ять із них – цілочисельні (включаючи символьний тип char), два – дійсні (float, double) і один логічний (булевий) тип даних.

- **byte** (цілі числа, 1 байт);
- **short** (цілі числа, 2 байти);
- **int** (цілі числа, 4 байти);
- **long** (цілі числа, 8 байтів);
- **float** (дійсні числа, 4 байти);
- **double** (дійсні числа, 8 байтів);
- **boolean** (значення істина / неправда, 1 байт).

Ці 8 типів служать основою для всіх інших типів даних. Примітивні типи мають явний діапазон допустимих значень:

- **byte** – діапазон допустимих значень від -128 до 127;
- **short** – діапазон допустимих значень від -32768 до 327678;
- **int** – діапазон допустимих значень від -2147483648 до 2147483647.

Тип int використовується частіше при роботі з цілочисельними даними, ніж byte і short, навіть якщо їх діапазону вистачає. Це відбувається тому, що при вказуванні значень типу byte і short у виразах, їх тип все одно автоматично під-вищується до int при обчисленні.

- **long** – діапазон допустимих значень від -9223372036854775808 до +9223372036854775807. Тип зручний для роботи з великими цілими числами.

- **float** – діапазон допустимих значень від ~ 1,4*10-45 до ~ 3,4*1038. Зручний для використання, коли не потрібно особливої точності в дробовій частині числа.
- **double** – діапазон допустимих значень від ~ 4,9*10-324 до ~ 1,8*10308. Математичні функції такі як sin(), cos(), sqrt() повертають значення double.
- **char** – символьний тип даних являє собою один 16-бітний Unicode символ. Він має мінімальне значення '\u0000' (або 0), і максимальне значення '\uffff' (або 65535 включно). Символи char можна задавати також за допомогою відпові-дних чисел.
- **boolean** – призначений для зберігання логічних значень. Змінні цього типу можуть приймати тільки одне з двох можливих значень – true або false.

У **посилальні типи** входять всі класи, інтерфейси, масиви.

Також існують класи-оболонки: Byte, Short, Integer, Long, Float, Double, Character, Boolean, String.

На відміну від примітивних типів вони пишуться з великої літери. Ці типи відповідають примітивним типам, однак є посилальними. Їх класи містять методи для перетворення типів, а також інші методи і константи, корисні при роботі з примітивними типами даних.

В якості типу також виступає будь-який створений нами клас при створенні екземпляру класу.

3.7. Змінні

У мові Java всі змінні повинні бути оголошені перед тим, як вони будуть використовуватися.

Оголошення змінних в Java:

```
int x = 1;
int y = 2;
```

При оголошенні змінної в наступній послідовності зазначаються:

- тип даних (в даному прикладі int – змінна містить ціле число),
- ім'я змінної (в даному прикладі імена – x і y),
- початкове значення змінної або, іншими словами, ініціалізація змінної.

У даному прикладі змінним x і y привласнені значення 1 і 2. Однак це не є обов'язковою умовою при оголошенні змінної.

Якщо потрібно оголосити кілька змінних одного типу, то це можна зробити одним рядком, вказавши імена змінних через кому.

Правила іменування змінних в Java:

- Ім'я змінної має починатися з літери (маленької) і складатися з букв (Unicode), цифр і символу підкреслення «_». Технічно можливо почати ім'я змін-ної з «$» або «_», проте це заборонено угодою з оформлення коду в Java (Java Code Conventions). Крім того, символ долара "$", за угодою, ніколи не використовується взагалі. Відповідно до угоди ім'я змінної повинно починатися саме з маленької літери (з великої літери починаються імена класів). Пробіли при іменуванні змінних не допускаються.

- Ім’я змінної не повинно бути ключовим або зарезервованим словом мови Java.
- Ім’я змінної чутливе до регістру: newVariable і newvariable — різні імена.
- При виборі імені змінних, слід використовувати повні слова замість загадкових абревіатур. Це зробить ваш код більш зручним для читання і розуміння. У багатьох випадках це також зробить ваш код самодокументованим.
- Якщо вибране вами ім’я змінної складається тільки з одного слова – запишіть його маленькими буквами. Якщо воно складається з більш ніж одного слова, то відокремлюйте кожне наступне слово в імені змінної великою літерою.
  Наприклад: superCounter, myCat.
- Якщо змінна зберігає постійне значення, то кожне слово слід писати великими літерами і відокремлювати за допомогою символу підкреслення.
  Приклад: static final int NUMBER_OF_HOURS_IN_A_DAY = 24.

3.8. Оператори

Оператор (англ. operator) – це спеціальний символ, який повідомляє транслятору про те, що ви хочете виконати операцію з деякими операндами (наприклад, +,-,%, <<). Зазвичай, мови програмування визначають набір опера́торів, подібних до операторів в математиці. В певному сенсі, оператори є спеціальними функціями. Окрім арифметичних дій, оператори в мовах програмування можуть виконувати операції на логічних значеннях, операції з рядками. Оператори порівняння можуть використовуватись для перевірки рівності двох значень. На відміну від функцій, оператори є базовими діями мови програмування, їх назва коротша, та, як правило, складається із спеціальних символів.

Оператори виконують відповідні операції над операндами (наприклад, додавання). Деякі оператори вимагають одного операнда, їх називають унарними. Одні знаки операції ставляться перед операндами і називаються префіксними, інші – після, тоді вони називаються постфіксними. Більшість же операторів ставляться між двома операндами, такі оператори називаються інфіксними бінарними операторами.

Java надає багате операторами середовище. Більшість операторів можна поділити на 4 групи:

- арифметичні,
- розрядні(бітові),
- відношення,
- логічні(булеві).

Інколи окремо виділяють ще оператори присвоєння, представники якої є у всіх інших групах. Також Джава визначає деякі додаткові оператори, які застосовуються в певних специфічних ситуаціях, наприклад instanceof. Існує тернарний оператор, що працює з трьома операндами (?:).

Слід зауважити щодо термінології. У літературі, що можна знайти у нашій країні, операторами інколи називають окремі види інструкцій (англ statement, досл. твердження, вираз), як то цикли та умовні інструкції, а то й просто функції. Саме слово operator відповідно заміняють словом операція. Часто різнобій в перекладах зустрічається в різних частинах однієї книги, що пов'язано з низькою якістю перекладів ряду книг, різнобоєм в англо-українських та англо-російських словниках, різноманітністю термінології в різних мовах програмування та неоднозначністю перекладу з англійської. В даній книзі дотримуватиметься термінологія, яка також зустрічається в нашій перекладній літературі, однозначніша і наближена до англомовного оригіналу термінології мови Java, тобто: operator – оператор, statement – інструкція.

3.8.1. Арифметичні оператори

Арифметичні оператори використовуються в математичних виразах так само як і в алгебрі

- + Додавання. +=, Додаваня з присвоєнням
- -, Віднімання (а також унарний мінус). -=,Віднімання з присвоєнням
- *,Множення. *=,Множення з присвоєнням
- /, Ділення. /=, Ділення з присвоєнням
- %, Залишок ділення по модулю. %=, Залишок ділення по модулю з присвоєнням
- ++, Інкремент (збільшення на 1). --, Декремент (зменшення на 1)

3.9. Блоки

Перед тим як знайомитись з керувальними структурами, спочатку необхідно ознайомитися з блоками. Блок або складена інструкція – це будь-яка кількість простих інструкцій, які оточені парою фігурних дужок. Блок визначає область видимості ваших змінних. Блоки можуть бути вкладені в середину інших блоків. Ви вже зустрічалися з блоками при створенні найпростіших програм у методі main(). Наступний приклад демонструє вкладення блоку у блок методу main:

```
public static void main(String[] args) {
  int n;
  . . .
  {
    int k;
    . . .
  } // змінна k визначена лише до цього місця
}
```

Проте не можна визначати однакові змінні в двох вкладених блоках (на відміну від C++, де це можливо).

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

```
. . .
  {
  int k;
  int n; // помилка! – не можна перевизначити n у внутрішньому блоці
  . . .
  }
}
```

3.10. Умовні конструкції

Умовна інструкція в Java має форму:

if (умова) інструкція;

Умова повинна бути оточена дужками і, якщо, умова вірна (true) буде виконана інструкція за умовою, інакше вона не буде виконана, а буде виконана наступна інструкція після умовної інструкції.

Приклад:

```
int a = 5;
if (a < 100) System.out.println("Число менше ста");
```

Зазвичай, необхідно виконати не одну інструкцію, в такому разі інструкції розміщають у блоці:

```
if (умова){
  інструкція 1;
  …..
  інструкція n;
}
```

В такому разі при істинності умови, виконуються усі інструкції у блоці, якщо умова невірна, то виконується наступна інструкція після закриваючої дужки блоку. Якщо ж необхідно здійснити певну дію в разі не виконання умови, то в такому разі застосовують умовну інструкцію наступного виду:

```
if (умова) інструкція1 else інструкція2
if (yourSales >= target) {
  performance = "Satisfactory";
  bonus = 100 + 0.01 * (yourSales - target);
} else {
  performance = "Unsatisfactory";
  bonus = 0;
}
```

Інструкції if можуть іти одна за одною без використання else:

```
if (x <= 0) if (x == 0) sign = 0; else sign = -1;
```

Для того, щоб програма була більш читабельна бажано застосовувати фігурні дужки:

```
if (x <= 0) { if (x == 0) sign = 0; else sign = -1; }
```

Вони нічого не змінюють, але вираз стає більш зрозумілим. Інструкція чи блок інструкцій виконується лише в разі виконання усіх умов.

Щоправда дану інструкцію можна також переписати ускладнивши умову використавши булевий оператор і (&&):

```
if (x <= 0 && x==0) sign = 0; else sign=-1;
```

Можна також використовувати повторюваність інструкцій if….else.

```
if (yourSales >= 2 * target) {
  performance = "Excellent";
  bonus = 1000;
} else if (yourSales >= 1.5 * target) {
   performance = "Fine";
   bonus = 500;
} else if (yourSales >= target) {
   performance = "Satisfactory";
   bonus = 100;
} else {
   System.out.println("You're fired");
}
```

Це дає можливість перевірити ряд умов, якщо попередні умови не виконуються.

3.10.1. Множинний вибір

Інструкція if/else може бути доволі громіздкою, якщо необхідно здійснити множинний вибір з багатьох альтернатив. Тож як і в C/C++ в java існує інструкція switch, яка здійснити вибір з багатьох варіантів. Щоправда вона дещо незграбна і деякі програмісти вважають за краще уникати її використання.

Наприклад, якщо Ви організовуєте певне меню і пропонуєте користувачу вибрати, номер конкретного пункту, то можна використати наступний код:

```
Scanner in = new Scanner(System.in);
System.out.print("Select an option (1, 2, 3, 4) ");
int choice = in.nextInt();
switch (choice) {
   case 1:
      . . .
      break;
   case 2:
      . . .
      break;
   case 3:
      . . .
      break;
   case 4:
      . . .
      break;
```

```
default:
   // bad input
   . . .
   break;
```

Якщо пропустити інструкцію break, то всі інші інструкції будуть також виконані. Тобто якщо справдиться умова першого варіанту, то будуть здійснені ще й дії вказані для виконання у всіх інших варіантах.

До версії Java 7, яка вийшла у 2011 році, case мітка мала бути лише цілим числом або нумерованою константою. Починаючи із Java 7 можна перевіряти таким чином на рівність також рядки:

```
String input = . . .;
switch (input)
{
   case "A":
      . . .
      break;
   . . .
}
```

3.11. Цикли

Цикли – це послідовність інструкцій, які можуть повторно виконуватись певну кількість раз в залежності від заданої в програмі умови. Розрізняють цикли з передумовою, з післяумовою та з лічильником.

3.11.1. Цикл while

Цикл while (перекладається як «доки») – це цикл з передумовою, тіло якого (тобто інструкція або блок інструкцій) виконується, якщо умова істинна. Якщо умова з самого початку хибна, то цикл не виконається жодного разу.

Загальний вигляд:

while (умова) інструкція;

Якщо немає фігурних дужок, то перша інструкція, яка йде після оголошення циклу, вважається тілом циклу. Всі інші інструкції знаходяться поза циклом. Якщо в циклі повинні виконуватись кілька інструкцій, то необхідно використати фігурні дужки. Вони також можуть використовуватись при одній інструкції заради кращого візуального розуміння коду.

```
while (умова){
  інструкція 1;
  ...
  інструкція N
}
```

В наступному прикладі демонструється мінігра із вгадуванням числа від 0 до 10, яка створена з використанням циклу while:

```
import java.util.*;

public class Game {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in); // створюємо сканер для введення даних з консолі
      Random generator = new Random();       //створюємо генератор випадкових чисел
      System.out.println("Спробуйте відгадати число від 0 до 10");
      int gn;
      String more = "Y";

      while (more.equals("Y") || more.equals("y")) {  // поки змінна more рівна “Y” або “y”

         gn = generator.nextInt(10); // генерація випадкового числа від 0 до 10;
         System.out.print("Введіть число від 0 до 10: ");
         int number = in.nextInt();  // зчитуємо число з клавіатури

         if (gn == number)
            System.out.print("Вгадали!!!  Спробуєте ще раз? (Y/N)");
         else
            System.out.print("Не вгадали. Спробуєте ще раз? (Y/N)");

         more = in.next();    // отримати відповідь
      }
   }
}
```

Результат виконання:

```
Спробуйте відгадати число від 0 до 10
Введіть число від 0 до 10: 6
Не вгадали. Спробуєте ще раз? (Y/N)y
Введіть число від 0 до 10: 7
Вгадали!!!  Спробуєте ще раз? (Y/N) n
```

Програма генерує випадкове число при кожному повторі циклу і пропонує вгадати його. Після вводу користувачем числа - виводить відповідне повідомлення вгадано чи ні. Після цього пропонується здійснити нову спробу. Якщо користувач вводить з клавіатури “Y" або “y”, то гра продовжується, якщо введе щось інше, то завершується.

Для генерації випадкових чисел використано клас Random, що містить методи для генерації випадкових чисел. Зокрема, у нашій програмі використано метод nextInt(), який дозволяє генерувати випадкові числа.

Для зчитування з клавіатури використано клас Scanner, який був доданий в java 5.0, для зручного вводу з клавіатури. Метод nextInt() – читає ціле число, next() – читає цілий рядок з клавіатури.

Для того, щоб переконатися, що користувач хоче продовжити гру використано метод equals() з класу String - Strint1.equals (String2), що перевіряє чи один рядок (String1) тексту рівний іншому (String2).

У вищенаведеній програмі пояснення потребують три рядка. Для початку Вам необхідно лише в загальному зрозуміти їхнє призначення.

```
import java.util.*;
Random generator = new Random();
gn=generator.nextInt(10);
```

Нам необхідний генератор цілих чисел, який реалізовує класс Random у пакеті java.util. Для цього ми імпортуємо відповідний клас (в C/C++ аналогом є підключення бібліотеки). Далі в програмі створюємо змінну generator, яка вказуватиме на екземпляр класу random і дозволятиме звернення до методів даного класу. new Random() - створює відповідний об'єкт. Для генерації випадкових чисел ми використовуємо метод nextInt() класу Random.

3.11.2. Цикл do/while

Якщо необхідно, щоб умова виконувалася хоча б один раз можна скористатися циклом з післяумовою do/while:

do інструкція while (умова);

Зокрема, в програмі з вгадуванням чисел, більш логічніше було б застосувати саме даний цикл, оскільки необхідне хоча б одне виконання тіла циклу.

```
import java.util.*;

public class Tmp {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in); // створюємо Сканер для введення даних з консолі
      Random generator = new Random();       // створюємо генератор випадкових чисел
      System.out.println("Спробуйте відгадати число від 0 до 10");
      int gn;
      String more;

      do {
         gn = generator.nextInt(10); //генерація випадкового числа від 0 до 10;
         System.out.print("Введіть число від 0 до 10: ");
         int number = in.nextInt();

         if (gn == number)
```

```
            System.out.print("Вгадали!!!  Спробуєте ще раз? (Y/N)");
          else
            System.out.print("Не вгадали. Спробуєте ще раз? (Y/N)");

          more = in.next();
        } while (more.equals("Y") || more.equals("y"));
    }
}
```

3.11.3. Цикл з лічильником for

Цикл for – доволі часто вживаний цикл. Він застосовується при необхідності виконати інструкції певну кількість раз з одночасним збільшенням або зменшенням певної змінної. Часто використовується для здійснення перебору певних масивів даних, зокрема, також для сортування масивів.

Приклад використання:

```
for (int i = 1; i <= 10; i++) {
        System.out.println(i);
}
```

Наведений вище приклад виведе на консолі в стовпчик числа від 1 до 10. Як бачимо в умові циклу перший слот відводиться для ініціалізації змінної, причому оголосити змінну можна і в іншому місці. Другий слот – для умови, яка перевіряється перед виконанням ітерації, третій слот – вказує як модифікувати змінну-лічильник. Тобто в наведеному прикладі при кожному виконанні ітерації, лічильник "i" буде збільшуватися на одиницю поки не стане рівним десяти.

Найчастіше даний цикл використовується для перебору елементів масиву.

Масив – це впорядкований набір даних одного типу. Найпростіший масив можна оголосити та ініціалізувати таким чином: int a[]={1, 5, 6, 1, 3};. Для того, щоб звернутися до певного елементу масиву використовуються квадратні дужки з відповідним індексом елементу.

Наприклад a[3] – звернення до четвертого елементу масиву (номери елементів відраховуються з нуля). В наступному прикладі створюється масив і послідовно виводяться його елементи:

```
public class MyArray {
   public static void main(String[] args) {
      int a[] = {1, 5, 6, 1, 3}; // створюємо масив і заповнюємо його числами
      int size = a.length;
      System.out.println("Елементи масиву:");
      for (int j = 0; j < size; j++) {
         System.out.println("a[" + j + "]=" + a[j]);
      }
   }
}
```

Результат виконання:

```
Елементи масиву:
```

a[0]=1
a[1]=5
a[2]=6
a[3]=1
a[4]=3

3.11.4. Цикл «for each»

Починаючи з java SE 5.0 в мові з'явився новий цикл, призначення якого є перебір елементів масиву або подібних до масиву типів даних (колекції).

Загальний вигляд циклу наступний:

```
for (type var : arr) {
   //тіло циклу
}
```

Наприклад, вивести елементи масиву, можна таким чином:

```
for (int element : a)
   System.out.println(element);
```

Використання даного циклу, дозволяє уникнути проблем пов'язаних з помилками при заданні умови в класичному циклі for. В інших мовах програмування цикл такого виду так і називається foreach, проте, щоб уникнути необхідності значних змін в пакетах, в java пішли простішим шляхом і перевантажили цикл for.

3.12. Масиви

Масив – це впорядкований набір однотипних елементів, на які посилаються по спільному імені. Це доволі зручний засіб групування інформації. Масиви можна створювати з елементів будь-якого типу. До конкретного елементу в масиві звертаються по індексу (номеру). Вони можуть бути як одновимірні так і багатовимірні.

3.12.1. Одновимірні масиви

Одновимірні масиви – це список однотипних елементів. Загальний формат оголошення такого масиву:

тип-елементів назва-масиву[];

Наприклад:

int month_days[]; // масив цілих чисел

Існує також інша форма оголошення масиву:

int[] month_days;

Проте для того, щоб масив почав існувати необхідно виділити під нього пам'ять, за допомогою операції new.

назва-масиву = new тип-елементів [розмір];

де розмір – планована кількість елементів у масиві.

```
month_days = new int[12];
```
або зразу ж:
```
int month_days[] = new int[12];
```

Таким чином відбувається виділення пам'яті під масив і ініціалізації елементів масиву нулями. В подальшому можна напряму звертатися до елементів масиву вказуючи індекс у квадратних дужках. Нумерація елементів в масиві в java відбувається з нуля. Тобто в наведеному прикладі звернення до першого(нульового) елемента – month_days[0], а до останнього – month_days[11]. Java не дозволить програмі звернутися поза межі масиву, щоправда помилка буде вказана лише на етапі виконання програми через викидання винятку(виключення).

```
month_days[5] = 30;
System.out.println(month_days[5]);
```

Масиви також можна ініціалізувати зразу ж при їхньому оголошенні, не використовуючи операції new, аналогічно як це відбувається при роботі з простими типами даних.

Наступний приклад зразу ж при оголошенні ініціалізує масив month_days[] кількістю днів в місяцях.

```
public class DaysOfMonth {
   public static void main(String[] args) {
   int month_days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; //оголошуємо та ініціалізуємо масив
   System.out.println("Травень має " + month_days[4] + " день"); // вивід на консоль
   }
}
```

Результат виконання на екрані:

```
Травень має 31 день
```

Наступний приклад демонструє знаходження **максимального числа** в одновимірному масиві.

```
public class ArrayMax {
   public static void main(String[] args) {
      double array[] = {1.1, 2.2, 1.1, 3.2, 1.2, 2.1};
      double max = array[0];
      for (int i = 0; i < 6; i++) {
         if (max < array[i])
            max = array[i];
      }
      System.out.println("Максимальне число в масиві: " + max);
   }
}
```

Як бачимо спочатку змінній max присвоюється значення нульового елементу масиву, після чого в циклі іде послідовне порівняння з кожним

наступним числом до останнього. Якщо при порівнянні чергове значення в масиві більше за максимальне в змінній max, то змінній max присвоюється дане значення. Як Ви уже зрозуміли, по закінченню циклу у змінній max міститиметься максимальне значення, яке і буде виведене на консоль:

Максимальне число в масиві: 3.2

В Java масиви є спеціальними об'єктами (про об'єкти детальніше в наступних розділах присв'ячних об'єтно-орієнтованому програмуванню), що забезпечує деяку додаткову функціональність масивам. Зокрема, можна дізнатися довжину масиву таким чином array.length. Для вищенаведеного прикладу можна замінити рядок з циклом таким чином:

for (int i =0; i < array.length; i++){ }

3.12.2. Багатовимірні масиви

Багатовимірні масиви по суті – це масив масивів. Робота з багатовимірними масивами подібна до роботи з одновимірними. Відмінність лише в тому, що використовуються додаткові квадратні дужки. Переважно використовуються двовимірні масиви, які служать для роботи з табличними даними та трьохвимірні масиви. Двовимірний масив та трьохвимірний, можна оголосити наступним чином:

int twoD[][] = new int [4][5]; //створення масиву 4x5
int threeD[][][] = new int[5][5][5]; //створення масиву 5x5x5

Для двовимірного лівий індекс означає номер рядка, а правий номер стовпця. Це можна уявити наступним чином:

[0,0][0,1][0,2][0,3][0,4]
[1,0][1,1][1,2][1,3][1,4]
[2,0][2,1][2,2][2,3][2,4]
[3,0][3,1][3,2][3,3][3,4]

Трьохвимірний масив можна уявити у вигляді куба. Крім номера рядка і номера стовпця, додається ще індекс елемента вглибину.

Наступна програма створює масив 5 на 4, заповнює його випадковими числами і виводить на екран.

```
import java.util.Random;                    // імпортуємо клас Random

public class RandomArray {
   public static void main(String[] args) {
      int m = 5, n = 4;                     //оголошуємо і ініцілізуємо змінні з розмірами масиву
      int Array[][] = new int[m][n];    //оголошуємо і ініціалізуємо масив
      Random generator = new Random();      // створюємо генератор випадкович чисел
      int gn;                  //змінна в яку буде записуватися згенероване генератором число
```

```
        /* заповнюємо масив випадковими числами */
        for (int i = 0; i < m; i++)            //проходимось по стовпцях
           for (int j = 0; j < n; j++) {       //проходимось по рядках
              gn = generator.nextInt(100);  //генерація випадкового числа від 0 до 100;
              Array[i][j] = gn;            //записуємо згенероване випадкове число
           }

        /* Виводимо результат */
        for (int i = 0; i < m; i++) {
           for (int j = 0; j < n; j++)       // зверніть увагу на відсутність фігурної дужки
              System.out.print(Array[i][j] + "    ");//даний рядок відноситься до масиву
по j
           System.out.println();          //виводимо символи переводу каретки і нового
рядка
           //після кожного проходження стовпцевих елементів рядка
        }
     }
}
```

В результаті на екрані одержимо:

```
94   47   65   0
99   20   60   69
80   33   63   73
35   50   48   81
39   19   4   85
```

В наведеному прикладі, в кожному рядку, однакова кількість елементів(стовбців). В Java можна створити двовимірні масиви з різною кількістю елементів в рядках.

```
int twoD[][] = new int[5][]; //створюємо двовимірний масив з 5-ма рядками
twoD[0] = new int[5]; // виділяємо пам'ять для 5-ти елементів нульового рядка
twoD[1] = new int[4]; // перший рядок матиме 4-ри елементи
twoD[2] = new int[3]; // другий - 3
twoD[3] = new int[2]; // третій - 2
twoD[4] = new int[1]; // четвертий – 1
```

Використання таких нерівних (нерегулярних) масивів не рекомендується, оскільки з ними важче працювати і можна припуститися ряд помилок, але в деяких ситуаціях можуть бути доволі корисними.

Як і з одновимірними масивами. Ми можемо зразу ж ініціалізувати масив необхідними значеннями при його оголошенні.

Приклад - Array2.java

```
public class Array2 {
```

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

      int[][] Array= {
         {5, 6, 1, 3},
         {3, 4, 2, 1},
         {1, 2, 2, 2}
       };
       for (int i = 0; i < 3; i++){
         for (int j = 0; j < 4; j++)
             System.out.print (Array[i][j]+"    ");
         System.out.println();
      }
   }
}
```

Результат виконання:

```
5   6   1   3
3   4   2   1
1   2   2   2
```

3.13. Методи

Методи в Java – це закінчена послідовність дій (інструкцій), спрямованих на вирішення окремого завдання. По суті це добре знайомі функції (вони ж процедури, підпрограми) більш ранніх, не ООП мов. Лишень ці функції є членами класів і задля розрізнення із звичайними функціями, згідно термінології об'єктно-орієнтованого програмування функції-члени називаються методами.

В загальному метод описується(визначається) наступним чином:

тип-повернення ідентифікатор-методу (параметри) {
тіло методу

return повертаєме-значення // якщо void, то інструкція return не обов'язкова
}

Тобто необхідно вказати тип значення, що метод повертатиме, наприклад double, float і т.д. або ж void – якщо нічого не повертає. ідентифікатор-методу – це його назва, за якого до нього звертатимуться. Методи рекомендується іменувати з маленької букви, наприклад, sum, addSum і т.д. В дужках вказується параметри методу через кому. При вказанні параметру необхідно вказати його тип та назву.

Більш повне визначення метода в класі наступне:

```
class MyClass {
 ...
   public ReturnType methodName( ParamOneType param1, ParamTwoType param2 ) throws ExceptionName
    {
```

```
    ReturnType retType;
    ...
    return retType;
  }
 ...
}
```

Якщо метод не повертає нічого, то в якості типу-повернення використовується ключове слово "void"

```
private void methodName( String param1, String param2 ) {
 ...
 return;
}
```

Ключове слово "return" в кінці метода можна пропустити. Крім того "return" може застосовуватись будь-де в методі, якщо необхідно закінчити виконання метода, наприклад, після виконання певної умови.

В наступному прикладі використовується метод sum() для знаходження суми трьох чисел та його виклик.

```
public class ProgramSum{
   public static void main(String[] args) {
      int a=2, b=3, k=4;
      int sum=sum(a, b, k);
      System.out.println("Сума трьох чисел дорівнює "+sum);
   }
   public static int sum (int a, int b, int c){
      return a+b+c;
   }
}
```

Як бачимо спочатку оголошуються і ініціалізуються змінні a, b, k. В подальшому дані змінні передаються у метод sum(), де вони сумуються і повертається їхня сума. Результат присвоюється змінній sum. Яка і виводиться на екран. В результаті на екрані з'явиться наступний напис:

Сума трьох чисел дорівнює 9

Зверніть увагу, що третій параметр у тілі метода носить інше ім’я – «c», а не «k». Це демонструє те, що назви параметрів у методі не пов’язані з назвами, які використані при виклику метода. При його виклику відбувається ініціалізація нових змінних зі списку параметрів методу відповідними значеннями змінних, що використані при його виклику (аргументів).

З виходом Java 5 з'явилася можливість викликати методи із змінною кількістю аргументів одного типу. Для цього метод оголошують наступним чином:

public double sum(double...nums)

Наступний приклад демонструє як можна підрахувати суму масиву використовуючи метод оголошений таким чином:

```
public class TestArgs {
```

```
    static double   k[]={5.1, 5.2, 5.3, 5.4, 5.5};

    static public double sum(double...nums){
            double sum=0;
            System.out.println("Кількість аргументів="+nums.length);
        for ( double n : nums )
        sum+=n;

        return sum;
        }

        static public void main(String[] args) {
            System.out.println("Сума="+sum(k ));
            System.out.println("Сума="+sum(33.4, 2.0));

        }
}
```

Результат:

Кількість аргументів=5
Сума=26.5
Кількість аргументів=2
Сума=35.4

В разі потреби таким чином можна створити і метод для прийому різної кількості масивів, наприклад, sum(double[]...nums). Якщо необхідно, щоб метод приймав і інші аргументи, то відповідні параметри йдуть спочатку, а згодом конструкуція для змінної кількості аргументів, наприклад sum(int k, double...nums).

3.13.1. Перевантаження методів

В мові Java в межах одного класу можна визначити два або й більше методів під одним іменем, що мають параметри, які відрізняються або кількістю, або типом. Такі методи називаються перевантаженими, а сам процес як перевантаження (англ. overloading) методів. Це один із способів реалізації поліморфізму.

Наступний приклад демонструє перевантаження методу sum:

```
public class ProgramSum{
   public static void main(String[] args) {
      int a=2, b=3, k=4;
      double a1=2.10, b1=4.20, k1=5.30;
      int sum=sum(a, b, k);
      double sum2=sum(a1, b1, k1);
      System.out.println("Сума трьох цілих чисел дорівнює: "+sum);
      System.out.println("Сума трьох дробових чисел дорівнює: "+sum2);
   }
```

```
    public static int sum (int a, int b, int c){
        return a+b+c;
    }
    public static double sum (double a, double b, double c){
        return a+b+c;
    }
}
```

В результаті на екрані отримаємо:

Сума трьох цілих чисел дорівнює: 9
Сума трьох дробових чисел дорівнює: 11.600000000000001

Як бачимо тіло методів практично не відрізняються. Відрізняються лише типом параметрів, що приймаються і типом параметрів, що повертаються, хоча останнє може бути однаковим, або й взагалі метод може нічого не повертати. Цікаво, що якщо б не було першого методу з цілочисловими вхідними параметрами, то можливий виклик другого методу без приведення змінних до типу.

```
public class ProgramSum{
    public static void main(String[] args) {
        // TODO code application logic here
        int a=2, b=3, k=4;
        double a1=2.10, b1=4.20, k1=5.30;
        double sum=sum(a, b, k);
        double sum2=sum(a1, b1, k1);
        System.out.println("Сума трьох цілих чисел дорівнює: "+sum);
        System.out.println("Сума трьох дробових чисел дорівнює: "+sum2)
    }
    public static double sum (double a, double b, double c){
        return a+b+c;
    }
}
```

Результат:

Сума трьох цілих чисел дорівнює: 9.0
Сума трьох дробових чисел дорівнює: 11.600000000000001

Зверніть увагу, що результат у першій стрічці дробовий. Одержаний результат отримали через те, що Java здійснила автоматичне перетворення типів int у double. Проте зворотнє перетворення не здійснюється автоматично, якщо було б навпаки і існував лише перший метод, то при спробі виклику з double параметрами ми б отримали помилку компіляції. В таких випадках потрібно здійснювати явне приведення типів.

В наступному прикладі відбувається виклик трьох перевантажених методів. Додано метод sum() з одним цілочисловим і одним дробовим параметрами.

```
public class ProgramSum{
    public static void main(String[] args) {
        int a=2, b=3, k=4;
```

```
        double a1=2.10, b1=4.20, k1=5.30;
        System.out.println("Сума трьох цілих чисел дорівнює: "+sum(a, b, k));
        System.out.println("Сума трьох дробових чисел дорівнює: "+sum(a1, b1, k1));
        System.out.println("Сума одного цілого числа та одного дробового дорівнює:
"+sum(a, k1));
    }
    public static int sum (int a, int b, int c){
        System.out.print("1-й метод.");
        return a+b+c;
    }
    public static double sum (double a, double b, double c){
        System.out.print("2-й метод.");
        return a+b+c;
    }
    public static double sum (int a, double b){
        System.out.print("3-й метод.");
        return a+b;
    }
}
```

Результат виконання.

1-й метод.Сума трьох цілих чисел дорівнює: 9
2-й метод.Сума трьох дробових чисел дорівнює: 11.600000000000001
3-й метод.Сума одного цілого числа та одного дробового дорівнює: 7.3

Можна перевантажити також методи із змінною кількістю аргументів:

static public double sum(double...nums){...}
static public double sum(int...nums){...}

Проте потрібно зважати, що sum(double...nums) та sum(double k, double...nums) буде сприйнято компілятором як методи з однаковою сигнатурою і видасть помилку, що в коді є неоднозначність. Це пояснюється тим, що такі оголошення компілятор при розборі перетворює у методи, що приймають масив і в даному випадку компілятор створить аналоги методів з одними і тими ж вхідними параметрами. Тож два однакові методи не можуть існувати в коді.

3.14. Конструктори

Замість методу safeValue(), який в нас заповнює змінні об’єкту значеннями ми можемо створити метод, який буде мати назву таку ж як і клас, тобто Safe(pWidth, pHeight, pDepth). Це дасть нам можливість ще скоротити програму, оскільки при створенні об’єкту ми зможемо зразу ж задавати розміри сейфу.

Safe mySafe1 = new Safe(10.0, 15.0, 20.0)

Такі методи носять назву “конструктор класу”. Коли ми писали просто new Safe(); то віртуальна машина використовувала конструктор по замовчуванню без параметрів, який практично нічого корисного для нас не робив. Тепер же ми можемо використовувати новий створений нами конструктор.

Таким чином, новий варіант програми:

```
class Safe {
   double width;
   double height;
   double depth;

   // конструктор
   Safe(double pWidth, double pHeight, double pDepth) {
      width = pWidth;
      height = pHeight;
      depth = pDepth;
   }

   // обчислюємо об'єм сейфу
   double getVolume() {
      return width * height * depth;
   }
}

public class CoinsVolume {
   public static void main(String[] args) {
      double width1 = 10, height1 = 20, depth1 = 40;
      Safe safe1 = new Safe(width1, height1, depth1); // створюємо 1-й сейф
      Safe safe2 = new Safe(10.0, 15.0, 20.0); // створюємо 2-й сейф
      Safe safe3 = new Safe(10.3, 15.4, 20.5); // створюємо 3-й сейф
      Safe safe4 = new Safe(20.0, 30.0, 20.0); // створюємо 4-й сейф

      printSafeVolume(safe1, 1); // виводимо об'єм 1-го сейфу
      printSafeVolume(safe2, 2); // виводимо об'єм 2-го сейфу
      printSafeVolume(safe3, 3); // виводимо об'єм 3-го сейфу
      printSafeVolume(safe4, 4); // виводимо об'єм 4-го сейфу
   }

   // вивід об'єму сейфу
   // safe - сейф
   // number - номер сейфу
   static void printSafeVolume(Safe safe, int number) {
      // викликаємо метод getVolume(), що обчислює об’єм сейфу і результат виводимо на екран
      System.out.println("Об'єм " + number + "-го сейфу = " + safe.getVolume());
   }
}
```

Результат виконання:

Об'єм 1-го сейфу = 8000.0

Об'єм 2-го сейфу = 3000.0

Об'єм 3-го сейфу = 3251.71
Об'єм 4-го сейфу = 12000.0

В наведеному прикладі, щоб обчислити об’єм нового сейфу, нам потрібно додати лише два рядочки тексту (дві інструкції). Щоправда навіть це можна автоматизувати за допомогою використання іншого типу даних – масивів, які будуть розглядатися пізніше. Можна також додати клас Coin, в якому був би метод для обчислення сукупного об’єму різноманітних монет. Ви можете спробувати зробити таку програму зараз. Це буде корисним для засвоєння викладеного матеріалу.

3.15. Алгоритми

Зараз познайомимось на базовому рівні с алгоритмами, які набирають популярності в Біг Даті та машинному навчанню.

3.15.1. Випадковий ліс

Random forest (англ. випадковий ліс) – алгоритм машинного навчання, запропонований Лео Брейманом і Адель Катлер, що полягає у використанні комітету (ансамблю) вирішальних дерев. Алгоритм поєднує в собі дві основні ідеї: метод беггінга Брейман і метод випадкових підпросторів, запропонований Tin Kam Ho. Алгоритм застосовується для задач класифікації, регресії і кластеризації.

Алгоритм навчання класифікатора.

Нехай навчальна вибірка складається з N прикладів, розмірність простору ознак дорівнює M, і заданий параметр m (в задачах класифікації зазвичай m approx sqrtM).

Усі дерева комітету будуються незалежно один від одного за такою процедурою:

1. Згенеруємо випадкову підвибірку з повторенням розміром n з навчальної вибірки. (Таким чином, деякі приклади потраплять в неї кілька разів, а приблизно N / 3 прикладів не ввійдуть у неї взагалі)
2. Побудуємо дерево рішень, яке класифікує приклади даної підвибірки, причому в ході створення чергового вузла дерева будемо вибирати ознаку, на основі якої проводиться розбиття, не з усіх M ознак, а лише з m випадково вибраних. Вибір найкращого з цих m ознак може здійснюватися різними способами. В оригінальному коді Брейман використовується критерій Джині[en], що застосовується також в алгоритмі побудови вирішальних дерев CART. У деяких реалізаціях алгоритму замість нього використовується критерій приросту інформації.
3. Дерево будується до повного вичерпання підвибірки і не піддається процедурі відсікання (на відміну від дерев рішень, побудованих за таким алгоритмам, як CART або C4.5).

Класифікація об'єктів проводиться шляхом голосування: кожне дерево комітету відносить об'єкт, який класифікується до одного з класів, і перемагає клас, за який проголосувало найбільше число дерев.

Оптимальне число дерев підбирається таким чином, щоб мінімізувати помилку класифікатора на тестовій вибірці. У разі її відсутності, мінімізується оцінка помилки out-of-bag: частка прикладів навчальної вибірки, неправильно класифікованих комітетом, якщо не враховувати голоси дерев на прикладах, що входять в їх власну навчальну підвибірку.

Переваги

1. Здатність ефективно обробляти дані з великим числом ознак і класів.
2. Нечутливість до масштабування (і взагалі до будь-яких монотонних перетворень) значень ознак.
3. Однаково добре обробляються як безперервні, так і дискретні ознаки. Існують методи побудови дерев за даними з пропущеними значеннями ознак.
4. Існують методи оцінювання значущості окремих ознак в моделі.
5. Внутрішня оцінка здатності моделі до узагальнення (тест out-of-bag).
6. Здатність працювати паралельно в багато потоків.
7. Масштабованість.

Недоліки

1. Алгоритм схильний до перенавчанню на деяких завданнях, особливо з великою кількістю шумів.[7]
2. Великий розмір отримуваних моделей. Потрібно O(NK) пам'яті для зберігання моделі, де K — число дерев.

3.5.12 Мурашиний алгоритм

Мурашиний алгоритм (алгоритм оптимізації мурашиної колонії, англ. ant colony optimization, ACO) – один з ефективних поліноміальних алгоритмів для знаходження наближених розв'язків задачі комівояжера, а також аналогічних завдань пошуку маршрутів на графах. Підхід запропонований бельгійським дослідником Марко Доріго. Суть підходу полягає в аналізі та використанні моделі поведінки мурах, що шукають дороги від колонії до їжі.

У основі алгоритму лежить поведінка мурашиної колонії – маркування вдалих доріг великою кількістю феромону. Робота починається з розміщення мурашок у вершинах графа (містах), потім починається рух мурашок – напрям визначається імовірнісним методом, на підставі формули:

$$P_i = \frac{{l_i}^q \cdot {f_i}^p}{\sum_{k=0}^{N} {l_k}^q \cdot {f_k}^p},$$

де,

P_i – ймовірність переходу шляхом i,

l_i – довжина i–ого переходу,

f_i – кількість феромонів на i-ому переході,

q – величина, яка визначає «жадібність» алгоритму,

p – величина, яка визначає «стадність» алгоритму $q + p = 1$

Результат не є точним і навіть може бути одним з гірших, проте, в силу імовірності рішення, повторення алгоритму може видавати (досить) точний результат.

Області застосування

Алгоритм оптимізації мурашиної колонії може бути успішно застосований для вирішення складних комплексних завдань оптимізації. Мета вирішення складних комплексних завдань оптимізації - пошук і визначення найбільш відповідного рішення для оптимізації (знаходження мінімуму або максимуму) цільової функції (ціни, точності, часу, відстані тощо) з дискретної множини можливих рішень.

Типовими прикладами вирішення такого завдання є задача календарного планування, завдання маршрутизації транспорту, різних мереж (GPRS, телефонні, комп'ютерні тощо), розподіл ресурсів та робіт. Ці задачі виникають у бізнесі, інженерії, виробництві та багатьох інших областях. Дослідження показали, що метод мурашиних колоній може давати результати, навіть кращі, ніж при використанні генетичних алгоритмів і нейронних мереж

3.15.3. Timsort

Timsort – гібридний алгоритм сортування, що поєднує сортування вставками і сортування злиттям, опублікований в 2002 році Тімом Петерсом. В даний час Timsort є стандартним алгоритмом сортування в Python, OpenJDK 7 і реалізований в Android JDK 1.5. Основна ідея алгоритму в тому, що в реальному світі сортовані масиви даних часто містять в собі впорядковані подмассіви. На таких даних Timsort істотно швидше багатьох алгоритмів сортування

Основна ідея алгоритму

- За спеціальним алгоритмом вхідний масив поділяється на підмасиви.
- Кожен підмасив сортується сортуванням вставками.
- Відсортовані підмасиви збираються в єдиний масив за допомогою модифікованої сортування злиттям.

Принципові особливості алгоритму в деталях, а саме в алгоритмі поділу і модифікації сортування злиттям.

3.15.4. Сортування включеням

Сортування включенням (Insertion sort) – простий алгоритм сортування на основі порівнянь. На великих масивах є значно менш ефективним за такі алгоритми, як швидке сортування, пірамідальне сортування та сортування злиттям. Однак, має цілу низку переваг:

- простота у реалізації
- ефективний (зазвичай) на маленьких масивах
- ефективний при сортуванні масивів, дані в яких вже непогано відсортовані: продуктивність рівна O(n + d), де d — кількість інверсій
- на практиці ефективніший за більшість інших квадратичних алгоритмів (O(n2)), як то сортування вибором та сортування бульбашкою: його швидкодія рівна n2/4, і в найкращому випадку є лінійною
- є стабільним алгоритмом

Наприклад, більшість людей при сортуванні колоди грального карт, використовують метод, схожий на алгоритм сортування включенням.

На кожному кроці алгоритму ми вибираємо один з елементів вхідних даних і вставляємо його на потрібну позицію у вже відсортованому списку до тих пір, доки набір вхідних даних не буде вичерпано. Метод вибору чергового елементу з початкового масиву довільний; може використовуватися практично будь-який алгоритм вибору. Зазвичай (і з метою отримання стійкого алгоритму сортування), елементи вставляються за порядком їх появи у вхідному масиві.

4. ООР В JAVA

Об'єктно-орієнтоване програмування (ООП) – значно змінило підхід до програмування увівши у вжиток цілий ряд понять і зокрема такі поняття як клас та об'єкти (екземпляри класу).

Парадигма(ідеологія) об’єктно-орієнтованого програмування (ООП) в даний час стала домінувати в програмному світі. Вона прийшла на зміну структурній техніці програмування, що була розроблена в 1970. Java є повністю об’єктно-орієнтованою мовою, тому потрібно засвоїти принципи ООП якомога краще.

В структурному програмуванні передбачалася розробка окремих алгоритмів та процедур для вирішення конкретної задачі. Такий підхід виправдує себе для невеликих задач, проте для великих проектів ООП більш виправдане. В літературі можна знайти приклад, що для реалізації простого веб-браузера необхідно близько 2000 процедур при структурному програмування. При використанні ж ООП можна створити 100 класів з приблизно 20-ма процедурами(далі методами) в кожному з них. Таким чином набагато простіше шукати помилку серед 20-ти методів одного класу ніж шукати її серед 2000 методів.

Клас – це певний шаблон, який слугує для створення об’єктів. Ви можете розробити власний клас, а можете отримати його від інших розробників. В інтернеті зараз є чимало бібліотек класів різноманітного призначення як безкоштовних так і платних. Наприклад, бібліотеки для промальовування різноманітних діаграм у своїх програмах і т.п.

ООП в java базується на ряді понять (або ж концепцій):

- **Клас** – певна абстрактна сутність, наприклад, "Пес", "Кіт", "Автомобіль", "Ціна товару", що на програмному рівні представлена змінними (полями даних) та методами, що оперують над цими полями даних.
- **Об'єкт** – конкретний екземпляр класу. Наприклад, зелена "Тойота" вашого сусіда є екземпляром класу "Автомобіль". По суті, це клас, поля якого ініціалізовані і він завантажений у пам'ять комп'ютера. На основі одного класу, можна створити безліч об'єктів. При цьому в об’єктах виділяють:
    - **Поведінку об’єкту** – що можна з робити з даним об’єктом, або які методи можна застосовувати до нього
    - **Стан об’єкту** – те як об’єкт змінюється, коли Ви застосовуєте його методи
    - **Ідентичність об’єкту** – відмінність об’єкту від інших об’єктів. Об’єкти можуть мати однаковий стан, проте все рівно вони ідентифікуються як різні об’єкти.
- **Успадкування (або ж "спадкоємство")** – утворення нових класів на основі інших.
- **Інтерфейс** – посилальний тип даних. Інтерфейси схожі на класи, проте їхні поля даних є константами, а методи не реалізовані. Об'єкти на основі інтерфейсів не створюються, проте класи можуть реалізовувати певний інтерфейс. І через об'єктну змінну інтерфейсного типу можна викликати реалізації даних методів.

- **Пакети** – каталоги, у яких розміщуються класи. Таким чином ми можемо використовувати однойменні класи, оскільки їхнє розрізнення іде не тільки за іменами, але й за розміщенням їх у каталогах (пакетах).

Насправді доволі часто різні автори по різному виділяють головні концепції ООП. Причиною є те, що в ООП справді доволі багато понять. І з плином часу та розвитком програмування їх лише більшає. Різноманітні поняття доволі тісно взаємопов'язані між собою. В літературі часто виділяють наступні три концепції і навіть вказують, що вони основні для ООП ("три кити"):

- **інкапсуляція (incapsulation)** – концепція побудови класів через закриття(капсулювання) їхньої реалізації.
- **успадкування (inheritance)** – створення одних класів на основі інших
- **поліморфізм (polymorphism)** – можливість використання батьківських класів замість класів нащадків. По суті є частиною реалізованої в мові концепції успадкування.

Деякі теоретики додають до цих трьох ще "**абстрагування**". Власне коли програміст створює клас, він створює певну абстракцію, модель чогось із реального світу. Ряд теоретичних книг побудовані на тому, як потрібно створювати класи, їхні ієрархії, зв'язки між ними.

Проте переважно вони настільки теоретичні, що практичні програмісти часто питання абстрагування вирішують виходячи із конкретної задачі, яку потрібно вирішити. Щоправда є і винятки. В даний час виділяють так звані Патерни проектування, ряд шаблонів чи то зразків того, як найбільш ефективно вирішити деякі задачі в ООП. Проте для того, щоб їх можна було освоїти, необхідне ґрунтовне вивчення об'єктно-орієнтованого програмування і зокрема наведених вище понять.

Поняття з обох вищенаведених списків є важливими для ООП і потрібно розуміти, що є що і як працює. Саме всім цим поняттям та їхній реалізації в мові програмування Java і присвячений даний розділ.

4.1. Інкапсуляція

Одною із переваг ООП є те, що користувачі класів можуть використовувати їх, майже не знаючи як вони реалізовані, використовуючи лише кілька методів для роботи з конкретним об’єктом. Таке закриття реалізації класу слугує також і певним захистом від неправильної роботи з даними. Даний принцип реалізації називається «інкапсуляцією». Якщо візьмемо реальний світ, то наприклад ви не знаєте як влаштований телевізор, проте ви можете його увімкнути та за допомогою кнопок переключати канали.

Сама ж схемо-технічна реалізація прикрита корпусом телевізора і переважно невідома глядачам ТВ. Аналогічно і класи розробляються за схожим принципом. Створюються певні методи, через які відбувається доступ до полів класу(змінних) та його методів. Всі інші класи, методи, поля, які слугують лише для обслуговування внутрішнього функціонування класу намагаються захистити, щоб до них не було доступу без крайньої на те потреби. Це в свою

чергу зменшує кількість помилок в програмі через невмілі дії користувача цього класу.

Інкапсуляція – один з основних принципів об'єктно-орієнтованого програмування. Йдеться про те, що об’єкт вміщує не тільки дані, але і правила їх обробки, оформлені в вигляді виконуваних фрагментів(методів). Доступ до стану об'єкта напряму заборонено, і ззовні з ним можна взаємодіяти виключно через заданий інтерфейс, що дозволяє контролювати правильність звернення до полів та їхню ініціалізацію. Також інкапсулюються методи, які виконують допоміжну роль і не бажано, щоб користувач-програміст мав доступ до них. Оскільки користувачі-програмісти працюють лише через відкриті елементи класів, то розробники класу можуть як-завгодно змінювати всі закриті елементи і навіть перейменовувати та видаляти їх, не турбуючись, що десь хтось їх використовує у своїх програмах.

Наприклад, вам потрібно зробити певну програму по роботі із списком автомобілів. Для цього потрібний відповідний клас Car:

```
public class Car {
    public static int count;
    public int id;

    public String _maker;
    public double _price;
    public String _year;
    public String _color;

    //конструктор без параметрів

    public Car() {
        count++;
        id = count;

    }

    //конструктор з параметрами, який ініціалізує всі поля класу

    public Car(String maker, String color, double price, String year) {
        _maker = maker;
        _price = price;
        _year = year;
        _color = color;

        count++;
        id = count;

    }
    //заміщення (перевизначення) методу toString() класу Object
```

```
        //замість дескриптора об'єкту, він виводитиме інформацію по автомобілю

        @Override
        public String toString() {
           return "Авто " + id + " " + _maker + " " + _color + " " + _price + " " + _year + " ";
        }

        //тестовий метод main
        public static void main(String[] args) {
          //створюємо об'єкт car1 конструктором без параметрів
           Car car1 = new Car();
           car1._maker = "Audi";
           car1._price = 10000;
           car1._year = "2000";
           car1._color = "red";

          //створюємо об'єкт car2 конструктором з параметрами
           Car car2 = new Car("BMW", "black", 12000, "2001");

           //вивід інформації про автомобілі
           //при цьому застосовуватиметься заміщений в цьому класі метод toString
           System.out.println(car1);
           System.out.println(car2);

        }
    }
```

Результат виконання:

Авто 1: Audi red 10000.0 2000

Авто 2: BMW black 12000.0 2001

У вищенаведеному прикладі для того, щоб рахувати кількість об'єктів ми створюємо статичну змінну count, яка буде спільною для всіх об'єктів car. В методі main ми створюємо екземпляри класу Car двома способами. Спочатку вручну ініціалізуємо кожне поле car1, а поля car2 ініціалізуємо через конструктор. Крім того, що другий спосіб є більш простий, перший спосіб може слугувати ще й джерелом ряду помилок. Наприклад, ми можемо забути ініціалізувати певне поле.

Крім того ініціалізуючи через конструктор ми можемо здійснити попередню перевірку на правильність введення даних. Наприклад на правильність введення назви виробника і т.п. Також, маємо два службових поля: count та id, які ініціалізуються в конструкторах, проте ми можемо задати значення і напряму, що може зашкодити логіці функціонування об'єкту класу. Ми запросто можемо вказати, що є 100 автомобілів, хоча насправді їх 66. І якщо

будемо десь використовувати цикл з перебору автомобілів з врахування змінної count, це викличе помилку.

Тому в ООП і придумано інкапсуляцію – приховування внутрішньої реалізації класу. Взагалі радиться, оголошувати поля та методи з самого початку закритими і лише в разі необхідності надавати до них більший доступ.

Модифікуємо дещо нашу програму:

```
public class Car {
   private static int count=0;
   private int id;

   private String _maker;
   private double _price;
   private String _year;
   private String _color;

   //конструктор з параметрами, який ініціалізує всі поля класу
   public Car(String maker, String color, double price, String year) {
      _maker = maker;
      _price = price;
      _year = year;
      _color = color;

      count++;
      id = count;
   }

   //заміщення (перевизначення) методу toString() класу Object
   //замість дескриптора об'єкта, він виводитеме інформацію по автомобілю

   @Override
   public String toString() {
      return id+". "+ _maker + " " + _color + " " + _price + " " + _year + " ";

   }

   //метод для отримання значення поля id
   public int getId() {
      return id;
   }
   //метод для отримання кількості автомобілів
   public static int getCount() {
      return count;
   }
```

```
    //тестовий метод main

    public static void main(String[] args) {
       Car car[]=new Car[5];

       car[0] = new Car("Audi","red",10000,"2000" );
       car[1] = new Car("BMW", "black", 12000, "2001");
       car[2] = new Car("Daewoo", "white", 8000, "2001");
       car[3] = new Car("Reno", "black", 12000, "2001");

       for (int i = 0; i < Car.getCount(); i++) {
          System.out.println(car[i]);
       }
    }
}
```

Результат виконання:

1. Audi red 10000.0 2000
2. BMW black 12000.0 2001
3. Daewoo white 8000.0 2001
4. Reno black 12000.0 2001

Як бачимо усі поля у нас тепер приватні. Робота з закритими полями можлива лише через відповідні методи. Для доступу до полів count та id створено методи getCount та getId. В разі необхідності можна створити методи для доступу до інших полів.

Також можна створити певні методи модифікації окремих полів (setId і т.п.) та передбачити в них попередню перевірку значень, що вводяться. Тож ми усунули можливість появи помилок при програмуванні пов'язаних з неправильним використанням полів та методів класу. Іншим програмістам буде значно легше використовувати клас Car, їм не потрібно вникати в особливості реалізації даного класу.

Необхідно зазначити, що методи, які читають значення полів прийнято називати з використанням префіксу get з наступним вказанням назви змінної, а для тих які модифікують значення використовують префікс set. В англомовній термінології можна зустріти терміни getter та setter методи або метод accesor та метод mutator.

Зверніть увагу як здійснюється звернення до змінної count. Насправді до методу можна звернутися з використанням об'єкту car[i].getCount(), але оскільки даний метод та змінна є статичними, тобто вона та метод спільно використовуються усіма об'єктами, то більш логічним є зверненням до неї через назву клас Car.getCount(). Даний метод можна викликати до створення будь-яких об'єктів. В такому разі ми просто отримаємо 0. Якщо б змінна не була приватною, то доступ до неї можна було б робити без застосування методу, безпосередньо: Car.count.

4.2. Успадкування

Успадкування або ж спадкоємство(англ. inheritance) – це ще один важливий механізм об'єктно-орієнтованого програмування, який дозволяє створювати нові класи на основі вже існуючих. Клас на базі якого створюється підклас називають надкласом, суперкласом або ж батьківським класом. Новий клас, що розширює (extends) батьківський клас називають підкласом або дочірнім класом. На відміну від C++ де клас може мати кілька батьківських класів, мова програмування Java підтримує одинарне успадкування, тобто може бути лише один безпосередній надклас.

У надкласу звичайно може бути свій надклас, проте також лише один безпосередній і т.д. Множинне успадкування доволі складне у застосуванні і вимагає обережного підходу, тому творці Java вирішили відмовитися від нього.

Допустимо у вас є клас SimpleRoom, який містить поля width (ширина) та length (довжина) кімнати та методи виведення інформації про кімнату.

```
public class SimpleRoom {

   protected double width=0.0;
   protected double length=0.0;

   public SimpleRoom(double width, double length) {
      this.width=width;
      this.length=length;
      System.out.println("SimpleRoom створено");
   }

   public void info (){
      System.out.println("Кімната: ширина = "+width+", довжина = "+length);
      System.out.println("Площа кімнати: "+width*length);
   }

   public static void main(String[] args) {
      SimpleRoom s=new SimpleRoom(5, 5);
      s.info();
   }
}
```

Результат виконання:

SimpleRoom створено
Кімната: ширина = 5.0, довжина = 5.0
Площа кімнати: 25.0

Допустимо вам необхідний клас, який би містив ще й інформацію про висоту кімнати і видав, ще й об'єм кімнати. Можна створити повністю новий клас, можна модифікувати вже існуючий клас, а можна створити клас

розширивши базовий клас SimpleRoom. Якщо клас SimpleRoom вже використовується в інших програмах, то змінювати його прийдеться обережно.

Для даного прикладу прийдеться створити ще один конструктор, а не модифікувати існуючий. В складніших випадках може знадобитися набагато більше дій. Крім того може бути, що клас SimpleRoom вже стандартизований, задокументований, його використовує чимало інших розробників і змінювати його просто так ви не маєте права.

Тож виходом є створення нового класу під ваші потреби. Завдяки ж можливості успадкування, нам не потрібно повністю переписувати клас. На основі класу SimpleRoom можна створити новий клас SimpleRoom2. Для цього достатньо вказати ключове слово extends (що означає "розширює") і вказати назву батьківського класу. Новий, дочірній клас отримує доступ до публічних і захищених полів та методів батьківського класу.

```
public class SimpleRoom2 extends SimpleRoom {
   protected double height;
   public SimpleRoom2(double w, double l, double h) {
      super(w, l);
      height=h;
      System.out.println("SimpleRoom2 створено");

   }

   public void info2(){
      System.out.println("Кімната: ширина = "+super.width+", довжина = "+super.length+", висота= "+this.height);
      System.out.println("Площа кімнати: "+width*length); // якщо немає конфлікту з іменами, то можна і пропустити super
      System.out.println("Об'єм кімнати: "+width*length*height);
   }

   public static void main(String[] args) {
      SimpleRoom2 s2 = new SimpleRoom2(5, 5, 3);
      System.out.println("Метод info SimpleRoom");
      s2.info();
      System.out.println();
      System.out.println("Метод info2 SimpleRoom2");
      s2.info2();
   }
}
```

Розберемо вищенаведений приклад. При створенні класу ми зазначили, який клас ми розширюємо. В класі введене нове поле height. Далі ми створили конструктор, в якому іде звернення до батьківського конструктора. Якщо б не було конструкторів з параметрами, то неявні виклики конструкторів викликалися б у такій же послідовності. Спочатку викликається конструктор дочірнього класу, з нього викликається батьківський конструктор, створюється

батьківський об'єкт, далі іде завершення конструктора дочірнього класу і створюється дочірній об'єкт. Для виклику конструктора суперкласу ми скористалися методом super з відповідними аргументами для батьківського конструктора:

super(w, l);

Також зверніть увагу як іде звернення до полів батькіського класу. Якщо поля не приватні, то вони доступні з дочірнього класу. Тож до них можна звертатися безпосередньо по імені, або ж скористатися для доступу ключовим словом super (взамін об'єктної змінної). Якщо б у нашому дочірньому класі існували б однойменні поля з батьківським класом (наприклад, і там і там width), то поля батьківського класу були б доступні лише з допомогою super.

Результат виконання SimpleRoom2:

SimpleRoom створено
SimpleRoom2 створено
Метод info SimpleRoom
Кімната: ширина = 5.0, довжина = 5.0
Площа кімнати: 25.0

Метод info2 SimpleRoom2
Кімната: ширина = 5.0, довжина = 5.0, висота= 3.0
Площа кімнати: 25.0
Об'єм кімнати: 75.0

Зверніть увагу, що ми без проблем через об'єктну змінну класу SimpleRoom2 викликаємо метод info класу SimpleRoom:

s2.info();

Тож крім полів дочірньому класу також доступні неприватні методи батьківського класу.

Насправді, у дочірньому класі (SimpleRoom2) можна було б створити однойменний клас info і він би замістив відповідний метод батьківського класу.

Даний механізм так і називається "**заміщення методів**" (англ. method **overriding**)

Інколи може виникнути необхідність **заборонити** можливість здійснення успадкування. Тобто можливість створення нових класів, на базі певного класу. Тоді *клас* можна оголосити як **final**. Також можна заборонити здійснення заміщення методу у класах потомках, використавши при оголошенні методу той же модифікатор **final**. Як уже мабуть ви знаєте, цей же модифікатор final також застосовується для оголошення констант – змінних, які ініціалізуються лише раз. В Java фінальним оголошено клас String.

4.3. Поліморфізм

Поліморфізм – важливий механізм в програмуванні, що дозволяє використовувати спільний інтерфейс для обробки даних різних спеціалізованих типів. Прикладом може слугувати перевантаження методів. З появою ООП концепція поліморфізму розширилась. В контексті об'єктно-орієнтованого

програмування, найпоширенішим різновидом поліморфізму є здатність екземплярів підкласу грати роль об'єктів батьківського класу, завдяки чому екземпляри підкласу можна використовувати там, де використовуються екземпляри батьківського класу. І як уже зрозуміло з визначення, поліморфізм тісно пов'язаний з успадкуванням.

Так, уявімо, у нас є клас Солдат та на його основі створено класи Генерал та Сержант. Логічно, що кожен Генерал є солдатом і кожен Сержант є солдатом, проте не кожен солдат є Генералом чи Сержантом. Тож Генерал може виконувати функції звичайного солдата, а солдат функції Генерала не зможе. Все вищесказане в ООП реалізовується через об'єктні змінні, які є поліморфними. Тому в коді ми можемо писати наступні інструкції:

Soldier s = new Soldier("Солдат"); // звичайне створення об'єкту Soldier

Soldier s2 = new General("Генерал"); // об'єктна змінна типу Soldier посилається на об'єкт типу General

Проте якщо б ми зробили б навпаки, спробували із солдата зробити генерала, то наступний рядок викликав би помилку на етапі компіляції:

General g = new Soldier("Солдат"); // !!! Помилка приведення типу (солдат не генерал)

Можна спробувати здійснити приведення до типу General, компілятор пропустить, проте під час виконання програми знову ж виникне помилка приведення типу (виняток виду: java.lang.ClassCastException: package.Soldier cannot be cast to osvjava.ua.General):

General g = (General)new Soldier("Солдат"); // ПОМИЛКА ВИКОНАННЯ!!!

Проте коли ми звертаємося до Генерала як до Солдата, то і функціональні можливості Генерала звужуються. Ми можемо викликати методи класу Солдат, проте з методами класу Генерал будуть проблеми.

Наприклад, в класі Soldier є метод getHealth(), а в класі General є метод getSlogan():

Soldier sg= new General("Генерал"); //змінна sg посилається на об'єкт типу General

sg.getHealth(); //методи класу Soldier доступні

// sg.getSlogan(); //методи класу General недоступні

Проте, якщо б метод getSlogan був би у класі Soldier, то викликана була б версія методу getSlogan класу General, оскільки при поліморфізмі заміщення методів все ж відбувається (крім статичних методів).

Якщо нам все ж необхідно звернутися до специфічних методів класу General, які притаманні лише йому, то необхідно створити відповідну об'єктну змінну типу General та здійснити явне приведення типу:

General general=(General)sg; // наш Генерал тепер повноцінний

general.getSlogan(); // метод класу General доступний

Якщо виникає необхідність визначити, до якого класу належить відповідний об'єкт, то можна використати метод getClass(), який може викликати будь-який об'єкт оскільки він дістається йому від прабатька усіх класів Object:

System.out.println(sg.getClass());
// результат: class osvjava.ua.General

Власне об’єктні змінні класу Object доволі часто застосовуються з метою збереження посилань на інші класи. Це дозволяє одночасно працювати з різнотипними об’єктами. Наприклад, можна, тримати різнотипні об’єкти в одному масиві типу Object. Також використання поліморфних об’єктних змінних дозволяє створювати своєрідні універсальні класи та методи (узагальнене програмування).

Таким чином непотрібне створення великої кількості перевантаження методів з різним типом параметрів. Власне практично всі внутрішні бібліотеки Java спочатку будувалися таким чином. Щоправда, використання поліморфних змінних може слугувати джерелом багатьох помилок, тому в Java починаючи з JSE 5.0 у мову введено так звані **Узагальнення** (**Generics**), які дозволяють більш краще будувати такі універсальні засоби

4.4. Абстрактні класи

В деяких випадках немає необхідності реалізовувати усі методи класу. Наприклад, якщо відомо, що ці методи в дочірніх класах і так обов'язково будуть заміщатися, то немає сенсу їх реалізовувати в батьківському класі. Також такі класи корисні при груповій роботі над класами, коли один програміст визначає список методів і реалізує основні з них, а реалізацію інших методів покладено на інших програмістів, які можуть створити кілька версій реалізації абстрактного класу. Такі класи, в яких немає реалізації усіх методів називаються абстрактними класами.

При описі абстрактного класу використовується модифікатор abstract. Нереалізовані методи також позначаються як абстрактні методи з використанням модифікатора abstract.

Потомок класу повинен реалізувати усі методи класу, або ж також бути оголошеним абстрактним.

Екземпляр абстрактного класу не можливо створити. Тобто ми не можемо створити об'єкт з використанням ключового слова new.

Для того, щоб позначити клас абстрактним, модифікатор **abstract** вписується в рядку-заголовку класу. Аналогічно даний модифікатор пишеться при оголошенні абстрактного методу.

Розглянемо наступну задачу.

Фірма виробник автомобілів, хоче мати програму, яка б розраховувала оптимальну мінімальну вартість автомобіля у конкретній країні. Виготовляються автомобілі у себе в країні і перевозяться у інші країни для продажу. При формуванні ціни, необхідно враховувати собівартість автомобіля, вартість транспортування та легалізація автомобіля в певній країні (мито, ПДВ, та інші збори). Тож звідси уже видно, що необхідно три методи. Розрахунок собівартості може зробити програміст в країні виробника. Інший програміст може розширити базовий клас та реалізувати метод, що обчислює вартість транспортування в кожній країні. І вже в країнах, куди автомобіль ввозиться, місцеві програмісти,

що володіють знаннями про всі податки, збори та витрати, можуть зробити метод, що розраховує вартість "легалізації" автомобіля в певній країні.

На основі вищесказаного визначаємо наш базовий абстрактний клас із одним реалізованим методом і двома пустими (без тіла):

```
package firstAbs;
/**
 * Абстрактний клас
 * Список методів до реалізації
 */
public abstract class CarCost {
    /**
     * обчислення собівартості
     * @return - Собівартість автомобіля на заводі
     */
     public double countPrimeCost() {
        //обчислення собівартості
        return 50000.0;
     }

    /**
     * Обчислення вартості перевезення
     * @param Country - країна
     * @return - вартість перевезення
     */
    public abstract double countTransportationCosts(String Country);

    /**
     * Обчислення вартості автомобіля в салонах продажів
     * @return - остаточна ціна автомобіля у певній країні
     */
    public abstract double countLocaleCost();
}
```

В реальному прикладі, програміст змушений би був отримувати вихідні дані із бази даних і опрацьовувати їх. Ми не будемо ускладнювати і вдаватися у деталі. Наш метод countPrimeCost повертає 50000.0. Приймемо умовно, що це і є результат обрахунку собівартості. Перед заголовком методів описано призначення методу у вигляді javadoc коментаря. Якщо ви використовуєте інтегроване середовище розробки, то при зверненні до методу через оператор "." (крапка), буде спливати підказка із наведеними відомостями про метод та його параметри.

4.5. Інтерфейси

Інтерфейси (interfaces) в мові програмування Java – це посилальний тип даних (reference data type), що подібний до класу, проте може містити лише константи, сигнатуру методу та вкладені типи. По суті інтерфейси подібні до абстрактних класів, проте якщо останні можуть містити крім сигнатури методів, ще й їхню реалізацію, то інтерфейси можуть містити лише опис даних методів.

Інтерфейс визначається наступним чином:

```
public interface MyInterface <T> {
   public String MYCONST="It is constant"; // константа
   public int method(T o); // нереалізований метод
}
```

<T> - це частина механізму **Generics**(**Узагальнень**), що дозволяє ввести додаткову перевірку типу при роботі, в даному випадку вказується, що інтерфейс буде працювати з будь-яким типом. При створенні об'єктної змінної можна уточнити тип (наприклад, MyInterface<String> myintvar). Generics були введено у випуск Java 5. Таким чином усуваються деякі можливі помилки, пов'язані із типами даних і зокрема з приведенням типів(детальніші у відповідному розділі). Інтерфейси можна створювати і старим способом, без використання узагальнень:

```
public interface MyInterface {
   public String MYCONST="It is constant" // константа
   public int method1(Object o); // нереалізований метод1
   public int method2(String str); // нереалізований метод2
}
```

Поля у інтерфейсі є константами, по суті в звичайному класі вони б оголошувались би як

```
public static final тип НАЗВА_КОНСТАНТИ;
```

В інтерфейсі дані модифікатори можна не вказувати. Згідно офіційних рекомендацій Oracle не рекомендується писати зайві модифікатори до членів інтерфейсу і згідно тих же рекомендацій константи в Java пишуться з великої літери.

Методи інтерфейсу мають єдиний тип доступу - public, тому в інтерфейсі можна описувати методи без модифікатора доступу. Проте при їх реалізації модифікатор public необхідно вказати.

Деякі автори книг не рекомендують використовувати інтерфейси для збереження констант.

Якщо клас реалізовує інтерфейс, то при його оголошенні в кінці вживається ключове слово implements з назвою інтерфейсу. Причому один клас може реалізувати безліч інтерфейсів. Наприклад:

```
public class Car extends SimpleCar implements Moveable, Comparable{
   .....
}
```

Переваги інтерфейсів

Перевагою інтерфейсів над абстрактними класами є те, що можна реалізувати кілька інтерфейсів в одному класі. Розглядаючи абстрактні класи,

було сказано, що вони зручні при груповій роботі, коли частину методів робить один програміст, а іншу частину інший. Крім того можна ієрархію класів де різні реалізації одного абстрактного класу можуть виконувати схожі, проте в дечому відмінні задачі.

В інтерфейсах дана ідея ще більше розширюється. Так, якщо один програміст розробляє клас(назвемо умовно FirstClass), який використовує інший клас(UseableClass), що повинен розробити інший програміст, то першому не потрібно чекати, коли другий програміст зробить необхідний йому UseableClass.

Вони можуть узгодити, що клас повинен мати певні методи, описати інтерфейс для даного класу (наприклад, interface Useable) і далі працювати кожен над своїм класом паралельно. Один пише клас UseableClass реалізовуючи(implements) логіку методів інтерфейсу Useable. Інший пише FirstClass так нібито клас UseableClass вже реалізовано.

Єдине, що їм потрібне – це узгоджений між ними інтерфейс, і, зокрема, сигнатура майбутніх методів. Як і у випадку з абстрактними класами в пригоді стає поліморфізм: змінна інтерфейсного типу може певним чином замінити ще не реалізований клас.

Звичайно, що вищесказане можна реалізувати і через абстрактний клас, проте, абстрактні класи працюють в ієрархії і ми не можемо реалізувати більше одного абстрактного класу в своєму класі. Інтерфейси ж знімають дане обмеження. Можна також створити абстрактні класи, які частково реалізовуватимуть певний інтерфейс. Також можна створити інтерфейси на основі інших, розширивши останні.

В Java існує ряд широковживаних інтерфейсів. Щоб об’єкти можна було порівняти між собою, необхідно реалізувати інтерфейс Comparable і його метод compareTo(). По суті призначенням інтерфейсів є змусити програміста, що розробляє певний клас, реалізувати методи, щоб об’єкт правильно функціонував.

В **Java SE 8** додана можливість додавати в інтерфейси статичні методи, та методи з реалізацією по замовчуванню з використанням ключового слова default перед ними.

5. ВИНЯТКИ В JAVA

Виняток (Exception) в java – це об'єкт, який описує виняткову (тобто, помилкову) ситуацію, що відбулась в певному місці коду. Коли така ситуація виникає створюється об’єкт, який передається («вкидається») в метод, в якому виникла помилка. Далі в методі даний виняток може оброблятися, або бути переданий ще кудись для обробки.

Розглянемо для прикладу наступну програму:

```
public class DivZero {
   public static void main(String args[]){
      int my=0;
      int medium=44/my;
      System.out.println("medium="+medium);
   }
}
```

Як бачимо в програмі присутнє ділення на нуль. При компіляції ми не отримаємо помилок. Проте, після запуску програми, отримаємо наступне:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at DivZero.main(DivZero.java:11)
Java Result: 1
```

Це так звана траса стеку викликів. Перший рядок означає тип-винятку. Як бачимо тут маємо ArithmeticException з діленням на нуль. java.lang – це пакет класів, який завжди доступний в програмі і який містить найбільш використовувані класи. Тобто його не потрібно імпортувати. Другий рядок вказує де саме відбулась виняткова ситуація: 11-й рядок у файлі DivZero.java в методі main() класу DivZero.

При цьому як бачимо програма завершила своє виконання аварійно. Для уникнення цього існує відповідний механізм обробки винятків, який дозволяє перехопити виняток, одержати інформацію про нього, обробити його здійснивши певні дії для нормального закінчення або продовження виконання програми.

Усі типи винятків є підкласами класу **Throwable**, який входить в базовий пакет класів **Java – java.lang**. Тобто він є вершиною ієрархії класів винятків. Його два підкласи Error та Exception утворюють дві основні гілки винятків.

Клас **Error** з його підкласами **–** це помилки виконавчого середовища java. І які зазвичай не виникають при нормальній роботі середовища java. Такі винятки зазвичай не можуть бути оброблені в програмі.

Гілка класу **Exception –** це винятки, які програма повинна вловлювати(catch). Від даного класу та його підкласів можна утворювати власні підкласи. Важливим його підкласом є клас RuntimeException. Винятки даного типу включають такі винятки як ділення на нуль та помилкова індексація масивів.

Актуальну ієрархію класів винятків можна подивитися і уточнити в офіційній документації до JDK.

5.1. Конструкція try

Для обробки виняткових ситуацій використовується п’ять ключових слів: try, catch, throw, throws та finally. Інструкції програми, в яких може виникнути помилка, контролюються за допомогою конструкції try.

Загальна форма наступна:

```
try{
//блок коду для контролю над помилками
} catch (тип-винятку1 об’єкт-винятку) {
  //дії при виникненні типу винятку1 (обробник винятку)
} catch (тип-винятку2 об’єкт-винятку) {
  //дії при виникненні типу винятку2 (обробник винятку)
}
 //….
finally{
        //дії при виході з конструкції try.
 }
```

Після інструкції try ми розміщуємо «небезпечний» код, у блоці catch відбувається обробка винятку, причому може бути кілька інструкцій catch. Завершувати конструкцію може інструкція finally, в ній розміщується код, який буде виконаний після обробки винятку в інструкції catch.

Таким чином можемо переписати програму з діленням на нуль:

```
public class DivZero {
   public static void main(String args[]){
      int my=0;
      try{
      int medium=44/my;
      System.out.println("medium="+medium);
      }catch(ArithmeticException e){
         System.out.println("Ділення на нуль!");
      }
      System.out.println("Продовження виконання...");
   }
}
```

Результат виконання:

```
Ділення на нуль!
Продовження виконання...
```

Як бачимо, для перехоплення винятку, код, через який виникала помилка, знаходиться у середині конструкції try. Також, зверніть увагу, що після виникнення виняткової ситуації наступний рядок

System.out.println("medium="+medium);

не було виведено, оскільки виняток був переданий для обробки в catch. Після інструкції програми в якій відбулась виняткова ситуація, всі наступні

рядки до інструкції catch пропускаються і не будуть виконуватись. І, як бачимо з результату виконання, наша програма продовжила виконання по закінченню блоку try, а не завершила своє виконання аварійно.

5.2. try з ресурсами

При використанні всередині try певних ресурсів, наприклад, файлів, при виникненні винятку необхідно було передбачити закриття відкритих ресурсів. Для цієї мети раніше приходилося використовувати блок finally. У Java 7 з'явилася конструкція try з ресурсами (англ. try-with-resources)[2]. Тепер просто можна створити ресурс у дужках зразу ж після ключового слова try і java сама потурбується про закриття ресурсу. Приклад:

```
static String readFirstLineFromFile(String path) throws IOException {
   try (BufferedReader br =
                new BufferedReader(new FileReader(path))) {
      return br.readLine();
   }
}
```

5.3. throw

В наведених вище прикладах ми здійснювали лише обробку винятків викинутих виконавчим середовищем java. Проте існує можливість викидання власних винятків. Для цього існує інструкція Throw. Загальна форма її наступна:

throw ThrowableInstance;

Тут **ThrowableInstance** – це тип винятку, який повинен бути або типом Throwable, або мати тип його підкласів. Щоб програма викинула ваш виняток необхідно скористатися оператором new. Для того, щоб одержати(перехопити) тип винятку можна скористатися інструкцією catch, як це ми робили вище.

Після інструкції throw відбувається аналогічне породження винятку як у вище наведених прикладах. Тобто усі інструкції пропускаються до найближчого блоку інструкції catch, де необхідно здійснити обробку винятку. Якщо catch не буде знайдено, то обробник винятків, що використовується по замовчуванню, призупиняє виконання програми і друкує відбиток стеку (stack trace).

Приклад:

```
public class ThrowNullException {
   public static void main(String args[]){
      try{
         throw new NullPointerException("Пробний виняток");
      }catch(Exception e){
         System.out.println("Наш витяток: "+e);
      }
   }
}
```

Результат:

Наш витяток: java.lang.NullPointerException: Пробний виняток

Програма демонструє як створювати один із стандартних винятків. Більшість вбудованих run-time винятків Java мають щонайменше два конструктори. Один за замовчуванням без параметрів і один із параметром String, який дозволяє задати додатковий опис. Опис можна вивести на консоль за допомогою методів print(), println(). Також його можна отримати використавши метод getMessage() класу Throwable.

Можна створити власний тип винятку як підклас уже існуючого типу.

5.4. throws

Якщо метод породжує виняток і не обробляє його, то він повинен вказати про це, щоб обробка винятку була здійснена у місці виклику даного методу. Це здійснюється за допомогою застереження throws в оголошенні методу. Після нього вказується підряд через кому усі винятки, які можуть бути викинуті методом, окрім винятків класів Error та RuntimeException і їхніх підкласів.

Нагадаємо, що клас Error – це необроблювані винятки, RuntimeException це винятки, які виникають в результаті помилки програміста (вихід за межі масиву, нульове посилання, невірне перетворення типів). Інші винятки – це помилки доступу, які доволі часто вимагають відповідної обробки.

Загальна форма оголошення методу наступна:

тип ім'я_методу(список_параметрів) throws список_винятків
{
// тіло методу
}

Наступна програма демонструє використання throws у методі де виникає виняток *IllegalAccessException*.

```
public class ThrowsException {
   public static void exceptionMethod () throws IllegalAccessException{
      System.out.println("Всередині exceptionMethod().");
      throw new IllegalAccessException("Помилка доступу");

   }
    public static void main(String args[]){
      try{
        exceptionMethod();
        System.out.println("Кінець програми"); //даний рядок не буде виведений
      }catch(IllegalAccessException e){
        System.out.println("Наш витяток: "+e);
      }
   }
}
```

Результат:

Всередині ExceptionMethod().
Наш витяток: java.lang.IllegalAccessException: Помилка доступу

Як бачимо тепер обробка винятку відбувається у методі main(). Без інструкції try-catch програма призупинятиметься з друком відбитку стеку. Слід зауважити, що у методах інструкція throw поводить себе подібно до інструкції return. Тобто виконання методу припиняється і відбувається повернення в місце виклику методу.

6. УДОСКОНАЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

6.1. Рефакторинг

Існує широко поширена помилкова думка відносно того, що в процесі добре організованого процесу розробки програмної системи ретельно розробляються методичні вимоги і визначається незмінний список задач програми, проект системи відповідає заданим вимогам і, в результаті кодування може здійснюватись лінійно, з початку до кінця, коли більшість коду пишеться один раз, тестується і про нього можна забути. Відповідно до цієї точки зору, суттєва зміна коду відбувається лише під час супроводження ПЗ, тобто вже після випуску початкової версії програми. Насправді ж код суттєво еволюціонує вже впродовж етапу початкової розробки. Багато змін, потреба в яких видимою під час початкової розробки є, як мінімум, такими ж кардинальними, як зміни під час підтримки.

Навіть в проектах з добре побудованим процесом управління вимоги до системи змінюються в процесі її розробки. Дані зміни позначаються на розроблюваному коді, іноді дуже суттєво. Інший факт – сучасні методи розробки збільшують потенціал для зміни коду в процесі конструювання. В старих життєвих циклах фокус був на уникненні змін коду. Сучасні підходи відходять від повної передбачуваності в розробці коду – вони більш кодо-центровані, і в процесі життєвого циклу проекту код еволюціонує більше, ніж колись.

Обов’язково потрібно як писати початковий код, так і вносити зміни, пам’ятаючи про майбутні зміни в коді. "Основне правило еволюції програмного забезпечення" говорить, що впродовж еволюції внутрішня якість програми повинна покращуватись.

6.1.1. Еволюція програми

В еволюції програми, як і в еволюції живих організмів, деякі зміни (мутації) є корисними, в той час як багато інших можуть бути шкідливими. Ключовим у питанні еволюції програмного забезпечення є те, чи покращується якість коду, чи погіршується. Програміст повинен розглядати процес внесення змін в програму як можливість покращити якість її дизайну та коду. Якщо ж помічається деградація якості коду, то це однозначний сигнал про те, що еволюція програми йде неправильно.

Другим фактором, який впливає на еволюцію ПЗ є те, чи зміни вносяться під час розробки, чи під час підтримки. Ці зміни розрізняються за декількома параметрами. Зміни під час розробки вносяться в більшості початковим розробником до того, як код є повністю забутим. Система ще не знаходиться в роботі, тому немає такого часового тиску, як під час підтримки. З тієї ж причини, зміни під час розробки дешевші, тому що система знаходиться в більш гнучкому динамічному стані. Дані обставини диктують стиль внесення змін, який відрізняється від того, що використовується під час підтримки.

6.1.2. Поняття рефакторингу

Ключовим методом виконання "основного правила еволюції програмного забезпечення" є рефакторинг. За визначенням Мартіна Фаулера це "зміна, внесена у внутрішню структуру програмного забезпечення для більшої його зрозумілості і здешевлення внесення змін без зміни зовнішньої поведінки програми". Слово “refactoring” в сучасному програмуванні походить від поняття “factoring” в структурному програмуванні, яке означало як можна більшу декомпозицію програми на складові частини.

6.1.3. Ознаки того, що потрібен рефакторинг

Перерахуємо основні ознаки ("попереджувальні знаки" – Мартін Фаулер називав їх "smells"), що вказують на потребу в рефакторингу:

- Є дублювання коду
- Метод занадто довгий
- Цикл занадто довгий або занадто велика вкладеність циклів
- Клас має погану внутрішню зв'язність
- Інтерфейс класу не забезпечує достатній рівень абстракції
- Список параметрів занадто довгий
- Зміни, що вносяться в клас, мають сегментний характер
- Зміни вимагають паралельну модифікацію багатьох класів
- Різні ієрархії класів повинні модифікуватися паралельно
- Пов’язані між собою дані, що використовуються разом, не організовані в один клас
- Метод використовує більше членів іншого класу, ніж свого
- Базовий тип перевизначено
- Клас не має достатньої функціональної навантаженості
- В ланцюгу викликів методів є ланцюг параметрів, що передаються
- Проміжний об’єкт не робить нічого власного
- Клас занадто зв’язаний з іншим класом
- Метод має погане ім’я
- Поля є public
- Клас-нащадок використовує тільки невелику кількість з методів класу-предка
- Коментарі використовуються для пояснення складного коду
- Використовуються глобальні змінні
- Метод потребує код ініціалізації перед викликом або пост-код після виклику
- Програма вміщує код, який здається таким, що "може знадобитись колись"

Розглянемо деякі з них.

Дублювання коду

Воно майже завжди означає, що при початковому проектуванні коду не було правильно проведено декомпозицію. Дублювання коду заставляє програміста

вносити паралельні зміни. Це суперечить широко відомому принципу DRY - Don’t Repeat Yourself.

Метод занадто довгий

Об’єктно-орієнтованому програмуванні методи величиною більші за екран рідко бувають потрібні і зазвичай означають спробу впхнути принципи структурного програмування в об’єктно-орієнтовану програму.

Одним зі шляхів покращення системи є збільшення її модульності – збільшення кількості добре визначених, добре іменованих методів, які добре роблять одну визначену річ. Якщо метод стає зрозумілішим при перенесенні частини його в окремий метод, потрібно створювати цей метод.

Інтерфейс класу не забезпечує достатній рівень абстракції

Навіть класи, які на початку мали раціонально побудований інтерфейс, можуть втратити свою цілісність з плином часу. Інтерфейс класу має тенденцію до розпливання в процесі розробки, особливо коли в клас вносяться термінові зміни. Врешті інтерфейс класу починає негативно впливати на зрозумілість програми.

Зміни, що вносяться в клас, мають сегментний характер

Іноді клас вирішує 2 або більше чітко визначених задач. Якщо так трапляється, програміст модифікує або одну частину класу, або іншу, але дуже рідко зміни зачіпають обидві одночасно. Це явна ознака того, що клас повинен бути розбитим на декілька в чіткій відповідності з задачами.

Пов’язані між собою дані, що використовуються разом, не організовані в один клас.

Якщо є набір змінних, які багаторазово використовуються разом, варто подумати над об’єднанням їх в клас.

В ланцюгу викликів методів є ланцюг параметрів, що передаються

Передача параметрів методу для того, щоб той передав їх як параметри в інший метод, може бути і добре, і ні. Потрібно подивитись, чи така передача узгоджується з абстракцією, вираженою в інтерфейсі кожного з методів. Якщо абстракція для кожного методу є відповідною, то і передача параметрів є відповідною, якщо ні, потрібно подумати над відповідністю інтерфейсів.

Метод має погане ім’я

Якщо виявлено, що метод має невідповідне ім’я, потрібно виправити його в місці опису та всіх місцях використання. Потрібно зробити це зразу при виявленні, тому що з часом це буде зробити ще важче.

Поля є public

public полів потрібно уникати, тому що вони роблять нечіткою лінію між інтерфейсом та реалізацією, порушуючи принцип інкапсуляції і обмежуючи гнучкість в майбутньому. Тому потрібно приховувати доступ до полів за public методами.

Коментарі використовуються для пояснення складного коду

Коментарі є важливими, але вони не повинні слугувати для пояснення поганого коду. Поганий код не потрібно пояснювати – його потрібно переписувати.

Метод потребує код ініціалізації перед викликом або пост-код після виклику

Код наступного виду є "попереджувальним знаком":

```
WithdrawalTransaction withdrawal;
withdrawal.SetCustomerId( customerId );
withdrawal.SetBalance( balance );
withdrawal.SetWithdrawalAmount( withdrawalAmount );
withdrawal.SetWithdrawalDate( withdrawalDate );
ProcessWithdrawal( withdrawal );
customerId = withdrawal.GetCustomerId();
balance = withdrawal.GetBalance();
withdrawalAmount = withdrawal.GetWithdrawalAmount();
withdrawalDate = withdrawal.GetWithdrawalDate();
```

Подібним знаком є випадок, коли створюється спеціальний конструктор для класу WithdrawalTransaction, який має параметрами значення для підмножини його властивостей:

```
withdrawal = new WithdrawalTransaction( customerId, balance,
withdrawalAmount, withdrawalDate );
ProcessWithdrawal( withdrawal );
delete withdrawal;
```

Кожен раз, коли зустрічається таке, потрібно задуматись, чи інтерфейс методу правильно представляє відповідну абстракцію. В даному випадку, напевне, метод *ProcessWithdrawal()* потрібно додати до класу WithdrawalTransaction, або список параметрів ProcessWithdrawal повинен бути змінений, щоб код мав вигляд:

```
ProcessWithdrawal( balance, withdrawalAmount, withdrawalDate );
```

Потрібно відмітити, що якщо подивитись навпаки, то можна побачити схожу проблему. Якщо виявляється, що існує об’єкт WithdrawalTransaction, а потрібно передавати значення декількох його властивостей в інший метод:

```
ProcessWithdrawal( withdrawal.GetCustomerId(), withdrawal.GetBalance(),
    withdrawal.GetWithdrawalAmount(), withdrawal.GetWithdrawalDate() );
```

то потрібно розглянути можливість зміни інтерфейсу методу ProcessWithdrawal, щоб передавати об’єкт, а не його окремі властивості. Будь-який з даних підходів може бути правильним чи ні – це залежить від того, чи абстракція інтерфейсу ProcessWithdrawal() є тим, що очікує отримати чотири окремі порції даних, чи цілий WithdrawalTransaction об’єкт.

6.1.4. Рівні рефакторингу

Термін рефакторинг іноді використовується в широкому значенні як будь-яке внесення змін в програму – з метою виправлення помилок, додавання функціональності, зміни дизайну тощо. Таке використання даного терміну не є точним і його потрібно намагатись уникати. Рефакторинг наголошує не просто на внесенні змін як таких, а на їх цілеспрямованості на досягнення покращення коду, що тягне за собою стійке покращення якості програми і запобігає широко відомій руйнівній спіралі програмної ентропії.

Приклади можливих кроків в процесі рефакторингу

Рефакторинги рівня даних

- Замінити магічне число іменованою константою.
- Замінити ім’я змінної на більш зрозуміле та інформативне.
- Відмовитись від проміжної змінної, записавши вираз в рядку використання його значення.
- Замінити вираз викликом підпрограми.
- Ввести проміжну змінну.
- Замінити змінну, яка використовується багаторазово для різних цілей, групою змінних, кожна з яких використовується з однією метою.
- Використовувати локальні змінні для локальних цілей, а не параметри.
- Перетворити змінну базового типу в об’єкт.
- Перетворити набір значень в клас.
- Перетворити набір значень в клас з підкласами.
- Перетворити масив на об’єкт.
- Інкапсулювати колекцію.
- Замінити запис (структуру) на клас.

Рефакторинги рівня операторів

- Провести декомпозицію логічного виразу.
- Перенести складний логічний вираз в добре іменовану булеву функцію .
- Об’єднати оператори, що знаходяться в різних частинах умовного оператора.
- Використовувати break чи return замість змінних, що служать для виходу з циклу.
- Виконувати return як тільки обчислено результат функції замість присвоєння значення, що повертається, спеціальній змінній.
- Заміняти умови поліморфізмом (особливо case-оператори).
- Створювати і використовувати null об’єкти замість перевірки на значення null.

Рефакторинги рівня методів

- Виокремити метод.
- Перенести код методу у місце виклику.
- Перетворити довгий метод у клас.
- Замінити простий алгоритм більш ефективним складнішим алгоритмом.
- Додати параметр.
- Вилучити параметр.
- Відділити операції читання від операцій запису.
- Об’єднати схожі методи, використавши параметри.
- Розділити методи, чия поведінка залежить від переданого параметра.
- Передавати як параметр цілий об’єкт, а не окремі поля.
- Передавати як параметр окремі поля, а не цілий об’єкт.
- Інкапсулювати приведення типу вниз.

Рефакторинги рівня реалізації класу

- Уникати багаторазового створення об’єктів шляхом використання вказівників на об’єкт, що розділяється декількома частинами програми (для великих, громіздких об’єктів).
- Уникати використання багатьох вказівників шляхом створення багатьох об’єктів (для малих об’єктів).
- Змінювати положення методів або даних в ієрархії.
- Виокремити спеціалізований клас в підклас.
- Скомбінувати схожий код в клас вищого рівня ієрархії.

Рефакторинги рівня інтерфейсу класу

- Перенести метод в інший клас.
- Розбити клас на два.
- Прибрати клас.
- Вилучити проміжний метод.
- Вилучити set()-методи для полів, які не можуть змінюватись.
- Приховувати методи, які не використовуються поза межами класу.
- Об’єднати клас-нащадок та клас-предок, якщо їх реалізація дуже подібна.

Рефакторинги рівня системи

- Робити копії даних, якими програміст не керує.
- Замінити односторонній зв’язок класів двостороннім.
- Замінити двосторонній зв’язок класів одностороннім.
- Створити фабрику об’єктів замість простого конструктора.
- Замінити коди помилок на виключні ситуації або навпаки.

6.1.5. Безпечний рефакторинг

Рефакторинг – ефективний і потужний інструмент програмування, але як і всі потужні інструменти при неправильному використанні він може нанести шкоду. Тому при рефакторингу потрібно користуватись наступними прийомами:

- Збереження початкового коду.
- Обмеження об’єму окремих видів рефакторингу.
- Виконання окремих видів рефакторингу по одному за раз.
- Складання списку дій, які програміст збирається виконати.
- Складання і підтримка списку видів рефакторингу, які потрібно виконати пізніше.
- Часте створення контрольних точок.
- Використання попереджень компілятора.
- Виконання регресивного тестування.
- Створення додаткових тестів.
- Виконання оглядів змін.
- Зміна підходу в залежності від ризикованості рефакторингу.

6.1.6. Стратегії рефакторингу

Число видів рефакторингу, вигідних для конкретної програми, майже нескінчене. Рефакторинг підлягає тому ж закону зниження вигоди, що й інші процеси програмування і до нього теж можна застосувати правило 80/20. Тому доцільно витрачати час на 20% видів рефакторингу, які забезпечать 80% вигоди.

При визначенні найважливіших видів рефакторингу варто:

- Виконувати рефакторинг при створенні нових методів.
- Виконувати рефакторинг при створенні нових класів.
- Виконувати рефакторинг при виправленні дефектів.
- Виконувати рефакторинг модулів, в яких велика ймовірність виникнення помилок.
- Виконувати рефакторинг складних модулів.
- При супроводженні програми покращувати фрагменти, які доводиться виправляти.
- Визначити інтерфейс між акуратним і поганим кодом та перенести поганий код на інший бік цього інтерфейсу.

6.2. Якість конструювання

6.2.1. Тестування коду розробником

Існує багато видів тестування програмного забезпечення. Деякі з них зазвичай здійснюються розробником, а деякі спеціальним персоналом – тестувальниками.

Unit-тестування – це запуск підпрограми, класу чи малої програми, створеної одним програмістом або групою програмістів, окремо від системи вищого рівня.

Компонентне тестування – це запуск класу, пакету, малої програми чи іншої програмної одиниці, яка включає роботу групи програмістів або програмістських команд, окремо від системи вищого рівня.

Інтеграційне тестування – це комбінований запуск двох або більше класів, пакетів, компонентів, підсистем, які створені групою програмістів або програмістських команд. Даний тип тестування зазвичай починається тоді, коли завершено створення двох класів і продовжується до закінчення роботи над системою.

Регресійне тестування – це повторення раніше виконаних тестів з метою знаходження дефектів в ПЗ, яке раніше пройшло той же набір тестів.

Системне тестування – це запуск ПЗ в його кінцевій конфігурації, включаючи інтеграцію з іншими програмними та апаратними системами. Тут іде перевірка на безпеку, швидкодію, втрату ресурсів, проблеми таймінгу та інші проблеми, які не можуть бути виявлені на нижчих рівнях інтеграції

Зазвичай, розробник здійснює unit-тестування, компонентне тестування та інтеграційне тестування, яке може включати регресійне та системне тестування. Численні інші види тестування здійснюються спеціальним персоналом (наприклад, тестувальниками) і до нього рідко залучаються розробники (це включає бета-тестування, тестування на прийняття замовником (customer-acceptance tests), тестування продуктивності, конфігураційне тестування, тестування платформи, стресове тестування, тестування зручності використання (usability tests) тощо).

Тестування можна розділити на 2 категорії – тестування "чорного ящика" та тестування "білого ящика". Тестування "білого ящика" має на увазі, що той, хто тестує, знає про внутрішню структуру об’єкту тестування – це якраз підходить для тестування розробником. Не слід плутати поняття "тестування" та "відлагодження" – тестування є засобом виявлення помилок, відлагодження – засіб виявлення і виправлення причин помилок, які вже виявленні.

Рекомендується виділяти на тестування від 8 до 25% часу, що йде на розробку системи.

Коли створювати тести?

Попереднє написання тестів, до написання коду, який вони тестують, дозволяє звести до мінімуму інтервал часу між моментами внесення дефекту та його виявлення/виправлення. Є й інші мотиви для попереднього написання тестів:

- створення тестів до написання коду вимагає тих же зусиль – просто змінюється порядок виконання цих двох етапів;
- попереднє написання тестів заставляє хоч трохи задуматись про вимоги і проект до написання коду, що сприяє покращенню коду;
- попереднє написання тестів дозволяє знайти проблеми в вимогах до написання коду, тому що складно створити тест для поганої вимоги.

Програмування з попередніми тестами є дуже ефективною методикою розробки програм, хоча воно теж має загальні обмеження і недоліки тестування розробником.

6.2.2. TDD (Test-Driven Development)

TDD є однією з основоположних частин XP (eXtreme Programming). Ідеї, покладені в основу XP, направлені на те, щоб зробити процес розробки ПЗ простішим і мати короткі, повторювані цикли розробки з постійним зворотнім зв'язком щодо стану програмної системи. TDD дає можливість розробляти складні програмні системи шляхом виконання простих ітерацій, в яких тести задають напрямок розвитку дизайну та реалізації системи.

Найскладнішим у використанні TDD є зміна уявлень програміста про способи побудови коду програми. Замість створення проекту, який описує, які програмні одиниці потрібно створити, створюються тести, кожен з яких описує, як повинна функціонувати маленька програмна одиниця системи. Ці тести визначають дизайн коду, який буде реалізовувати дані програмні одиниці системи.

TDD базується на двох методологіях – unit-тестуванні та рефакторингу. Загальна схема розробки при використанні TDD така:

1. Написати тест, який визначає, як, на думку автора, маленька програмна одиниця повинна поводитись.
2. Створити програмну одиницю якомога простішими засобами так, щоб вона пройшла тест.
3. Здійснити рефакторинг коду, зробивши його більш якісним і перевіряючи за допомогою тестів його правильність після внесення кожної зміни.

Оскільки TDD є ітеративним процесом, то потрібно повторювати дані кроки, поки не буде досягнуто бажаної якості коду.

При використанні TDD системні вимоги (наприклад, сформовані у вигляді use cases) декомпонуються в набір дій, виконання яких приведе до виконання вимог. Для кожної з даних дій спочатку пишеться unit-тест, який повинен перевірити правильність виконання даної дії. Разом зі створенням тесту визначаються чіткі критерії, за якими можна визначити, коли написано достатньо коду для виконання заданої дії. Однією з переваг попереднього написання тестів є те, що це допомагає чіткіше визначити бажану поведінку системи та дати відповіді на деякі основні питання її проектування.

Приклад. Однією з вимог до програми є визначити процент влучності баскетболіста за гру та за сезон. З вимоги можна визначити ряд дій, які повинна робити програмна одиниця:

- ідентифікувати гравця;
- ввести дані, необхідні для визначення проценту влучності в грі;
- ідентифікувати гру;
- обрахувати середній процент влучності за гру;
- обрахувати процент влучності за сезон.

Зробивши спрощене припущення, що гравці ідентифікуватимуться за ім'ям, а ігри за датою, можна написати код:

```
Player wally = PlayerList.getPlayer("Wally Wiffer"); // Get player
wally.addGameA("8/9/2003",15,30);
wally.addGameA("8/16/2003",25,75);
wally.addGameA("8/23/2003",4,16);
float gameA = wally.getGameA("8/9/2003"); // Get average for game on 8/9
float seasonA = wally.getA(); // Get average season
assertEquals(0.5, gameA); // Check game average calculation
assertEquals(0,36, seasonA); // Check season average calculation
```

Написавши даний тест, маємо, що потрібно створити класи PlayerList та Player. В PlayerList повинен бути метод getPlayer(String), а в Player – addGameA(String, int, int), getGameA(String), getA().

Коли тест написано, він повинен бути запущений. Спочатку він, звичайно, повинен виконатись з помилками (оскільки код самої програми ще не створено) – це покаже, що сам тест запускається і функціонує. Потім пишеться програма до того часу, поки в тестах не зникнуть помилки. Потім проводиться рефакторинг; після кожного виду рефакторингу знову запускаються тести для перевірки, чи не зруйновано правильність коду.

Потім іде перехід до наступної дії програми з повторенням тих же кроків. Даний інкрементальний підхід заставляє програміста здійснювати постійний рефакторинг коду і приводить до того, що зменшується небезпека "занадто-проектування" (overdesign) системи і створення роздутого коду, оскільки код додається невеликими, прорефакториними порціями.

Коли розробляється більше і більше коду, число тестів швидко збільшується і створюється ціла система тестів, яку можна виконати в будь-який момент. Дана система тестів є важливою в TDD, тому що її можна використовувати для постійного моніторингу програмної системи і миттєвого виявлення проблем. Оскільки тести виконуються регулярно впродовж процесу розробки, вони допомагають оперативно виявляти і виправляти помилки. Це робить процес розробки ПЗ більш безпечним в плані помилок.

6.2.3. Переваги, які надає TDD

1. Спрощена, інкрементна розробка. Програміст концентрується на розробці невеликих блоків коду, таким чином просуваючись в розробці. Це краще, ніж заздалегідь виконати велику роботу з проектування і потім виявити в

проекті масу неузгодженостей. Також однією з основних переваг TDD є те, що вже на початку розробки одержується робоча система, нехай і з обмеженою функціональністю.

2. Можливість постійного регресійного тестування. В програмуванні часто зустрічається ефект доміно – коли невелика зміна в якійсь частині програми може потягти за собою непередбачувані зміни у функціонуванні всієї системи. Частиною TDD є проведення регресійного тестування після внесення кожної зміни в код, що дає можливість виявляти помилки зразу після їх внесення і таким чином захищає програміста від необхідності високовартісного виправлення помилок в майбутньому.
3. Покращена комунікація і централізація знань. Тести є хорошим методом документування коду – більш точним, ніж словесна або графічна форма і дає можливість розробнику описати ідеї дизайну своєї програми в формі, яка буде зрозумілою іншим.
4. Покращене розуміння вимог до програми, і, відповідно, покращений дизайн програми. Написання тестів до коду дає можливість поглянути на неї спочатку з точки зору користувача і уяснити вимоги до програми краще.
5. Покращена інкапсуляція і модульність. В процесі розробки програміст може внести в код небажані зв’язки між класами. Один з принципів TDD стверджує, що unit-тести повинно бути легко виконати. Це означає, що потрібно мінімізувати вимоги необхідні для запуску тестів. Фокусування на простоті тестів веде до покращення модульності класів, зменшуючи залежності.
6. Зменшення складності за рахунок не внесення надлишкового коду. Розробники часто вносять в код надлишкові методи в сподіванні, що ті потім знадобляться, намагаючись зробити програму більш гнучкою, більш придатною до модифікації в майбутньому. Це веде до збільшення складності програми. Наявність набору тестів дає програмісту більшу впевненість в позитивному результаті внесення змін (навіть досить суттєвих) в код і це приводить до відсутності необхідності в надлишковому коді.

6.2.4. Фреймворк JUnit

JUnit 4.x є тестовим фреймворком для Java, який, на відміну від попередніх версій, використовує анотації для позначення тестових методів.

Таблиця 1. Анотації JUnit

Анотація	Опис
@Test public void method()	Анотація @Test позначає, що даний метод є тестовим методом
@Before public void method()	Виконує method() перед кожним тестом. Цей метод може підготовлювати середовище тестування, наприклад, вводити тестові дані, ініціалізувати клас тощо)
@After public void method()	Тестовий метод, який запускається з тестом
@BeforeClass public void method()	Виконує метод перед початком всіх тестів. Це використовується для виконання дій, що вимагають великих витрат, наприклад, під'єднання до бази даних.
@AfterClass public void method()	Виконує метод після закінчення всіх тестів. Це може бути використано для виконання завершальних дій, наприклад для від'єднання від бази даних.
@Ignore	Тестовий метод буде ігноруватись; є корисним тоді, коли тестований код змінився, а тест ще не адаптували до нього чи встановлення середовища виконання тесту займає занадто багато часу.
@Test(expected=IllegalArgumentException.class)	Тестує, якщо метод генерує вказану виключну ситуацію.
@Test(timeout=100)	Виконується з помилкою, якщо метод виконується довше за 100 мілісекунд

Таблиця 2. Методи JUnit

Метод	Опис
fail(String)	Дає можливість методу виконатись не успішно; може бути корисним для визначення того, що певна частина коду не досягнута.
assertsEquals([String message], expected, actual)	Перевіряє, чи значення є однаковими.. Примітка: для масивів перевіряється рівність вказівників, а не значень комірок масивів
assertsEquals([String message], expected, actual, delta)	Використовується для типів float та double; delta вказує значення припустимої розбіжності між значеннями
assertNull([message], object)	Перевіряє, чи об'єкт null

Метод	Опис
assertNotNull([message], object)	Перевіряє, чи об’єкт не null
assertSame([String], expected, actual)	Перевіряє, чи обидва вказівники посилаються на один і той же об’єкт
assertNotSame([String], expected, actual)	Перевіряє, чи обидва вказівники не посилаються на один і той же об’єкт.
assertTrue([message], boolean condition)	Перевіряє, чи булевий вираз дорівнює is true.

Приклад створення JUnit тесту (в середовищі Eclipse).
Припустимо, в нашій програмі існує клас

```
package de.vogella.junit.first;
public class MyClass {
    public int multiply(int x, int y) {
            return x / y;
    }
}
```

Якщо натиснути правою кнопкою мишки на класі і вибрати New ->JUnit Test case, то з’явиться вікно рис.3.

Тести прийнято розміщувати в окремій папці поза межами src, наприклад, в папці test. Потрібно вказати Source folder, вибрати New JUnit 4 test та натиснути Next. Потім потрібно вибрати методи для тестування (рис.4).

Якщо це робиться в перший раз, то потрібно вказати підключення бібліотеки(рис. 5).

Створимо тест:

```
package de.vogella.junit.first;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class MyClassTest {
    @Test
    public void testMultiply() {
            MyClass tester = new MyClass();
            assertEquals("Result", 50, tester.multiply(10, 5));
    }
}
```

Натиснувши правою кнопкою мишки на класі тесту і вибравши Run-As-> Junit Test, одержуємо результат тестування у вікні JUnit (рис.6). Результат вказує на помилку. Виправивши в методі multiply рядок return x/y; на рядок return x*y;, одержимо правильний результат.

7. КОРЕКТНИЙ ТА НЕКОРОКТНИЙ ПІДХІД ПРОГРАМУВАННЯ

(JAVA CODE CONVENTION)

7.1. Використання іменованих констант.

Уявімо, в методі потрібно розташувати текстові поля рівно одне під одним, а справа від кожного текстового поля кнопку. Недоцільно для задання координат вказувати безпосередні числа, тут доцільно використати іменовані константи. Це дасть змогу швидко і без зусиль змінити розташування елементів, а також робить програму більш зрозумілою. В даному випадку не потрібно боятись зайвих рядків коду, тому що зрозумілість програми і простота її модифікації мають величезне значення.

Некоректний підхід	*Коректний підхід*

public void locateVisualElements (){
for (int i=0;i<4;i++){
 	
textFields[i].setBounds(40,60+i*60,120,50);
 	
buttons[i].setBounds(190,60+i*60,110,50);
}
}
	final int TEXT_FIELDS_X=40;
	final int ELEMENTS_Y=60;
	final int BUTTONS_X=40;
	final int ELEMENT_SHIFT=60;
	final int BUTTONS_WIDTH=110;
	final int TEXT_FIELDS_WIDTH=120;
	final int ELEMENT_HEIGHT=50;
	public void locateVisualElements (){
	for (int i=0;i<4;i++){
	
	textFields[i].setBounds(**TEXT_FIELDS_X,**
	ELEMENTS_Y +i*****
	ELEMENT_SHIFT,
	TEXT_FIELDS_WIDTH,
	ELEMENT_HEIGHT);
	buttons[i].setBounds(**BUTTONS_X,**
	ELEMENTS_Y +i*****
	ELEMENT_SHIFT,
	BUTTONS_WIDTH,
	ELEMENT_HEIGHT);
	}
	}

7.2. Змінні

Імена змінних повинні виражати їхню сутність. Вважається, що оптимальною є довжина ім’я 10-16 символів. Крім того, існують різні конвенції іменування; деякі з правил іменування є обов’язковим для вживання в певних мовах. Наприклад, в мові Java імена змінних починаються з маленької букви, а кожне смислове слово всередині імені починається з великої.

Некоректний підхід	*Коректний підхід*

k=a+b;
c=met(k);
k+=c;
	totalPrice=wholesalePrice+profit;
	vat=calculateVat(totalPrice);
	totalPrice+=vat;

7.3. Методи

Параметри методів

Даний приклад ілюструє один з аспектів використання параметрів методів. Параметри методів, як правило, нечисленні. Дуже рідко можна зустріти метод, в якому більше 4 параметрів. Якщо в методі є багато параметрів одного й того ж типу, це неприпустимо – потрібно об'єднати їх в рамках однієї змінної, наприклад, масиву або колекції.

Некоректний підхід	*Коректний підхід*

public void inputData (JTextField tf1, JTextField tf2, JTextField tf3, JTextField tf4){
name=tf1.getText();
height= Integer.parseInt(tf2.getText());
weight= Integer.parseInt(tf3.getText());
sort= tf4.getText();
}
	public void inputData (**JTextField tf[]**){
	name=tf[0].getText();
	height= Integer.parseInt(tf[1].getText());
	weight= Integer.parseInt(tf[2].getText());
	sort= tf[3].getText();
	}

Довжина методів

Загальноприйнято, що довжина методу не повинна перевищувати одного екрану комп'ютера. Якщо написаний вами метод виявився занадто довгим, то його потрібно розбити на декілька методів, кожен з яких повинен мати ім'я, яке виражає сутність того, що він робить. Розбивати метод на декілька зручно за допомогою пункту Extract method засобів рефакторингу візуальних середовищ розробки програм.

Некоректний підхід

```
class StringProcessing{
...
private static String halfStringToUpperCase(
                                        String
initialString) {

 }
 public static void main(String[] args){
   JFrame frame=new JFrame("Laba 1 ");
   frame.setDefaultCloseOperation(

JFrame.EXIT_ON_CLOSE);
   JButton btn=new JButton("Start");
```

Коректний підхід

```
class StringProcessing{
...
 private static String halfStringToUpperCase(
                                        String
initialString) {
   String firstHalf=new String();
   String secondHalf=new String();
   for(int i=0; i<initialString.length()/2;i++)
     firstHalf=firstHalf+initialString.charAt(i);
   for(int i=initialString.length()/2;
                                  i<initialString.length();
i++)
```

```
        Label infoLabel=new Label("Lababoratorna1",
                                  Label.CENTER);
        final JTextField textIn=new JTextField();
        final JTextField textOut=new JTextField();
        frame.getContentPane().add(infoLabel);
        frame.getContentPane().add(textIn,

BorderLayout.NORTH);
        frame.getContentPane().add(textOut,

BorderLayout.SOUTH);

frame.getContentPane().add(btn,BorderLayout.EAST);
        frame.setSize(450,100);
        frame.setLocation(300,250);
        frame.setVisible(true);
        frame.setResizable(false);
        btn.addActionListener(new ActionListener(){
         public void actionPerformed(ActionEvent e){
          String initialString = textIn.getText();
          String firstHalf=new String();
          String secondHalf=new String();
          for(int i=0; i<initialString.length()/2;i++)
         firstHalf=firstHalf+initialString.charAt(i);
          for(int                    i=initialString.length()/2;
i<initialString.length();

i++)
            secondHalf=secondHalf+initialString.charAt(i);
         firstHalf=firstHalf.toUpperCase();
         textOut.setText(halfStringToUpperCase(

firstHalf+secondHalf));
          }
        });
       ...
      }
     }
```

```
secondHalf=secondHalf+initialString.charAt(i);
      firstHalf=firstHalf.toUpperCase();
      return  firstHalf+secondHalf;
     }
     public static void main(String[] args){
      JFrame frame=new JFrame("Laba 1 ");
      frame.setDefaultCloseOperation(

JFrame.EXIT_ON_CLOSE);
      JButton btn=new JButton("Start");
      Label infoLabel=new Label("Lababoratorna1",

Label.CENTER);
      JTextField textIn=new JTextField();
      JTextField textOut=new JTextField();
      frame.getContentPane().add(infoLabel);
      frame.getContentPane().add(textIn,

BorderLayout.NORTH);
      frame.getContentPane().add(textOut,

BorderLayout.SOUTH);

frame.getContentPane().add(btn,BorderLayout.EAST);
      frame.setSize(450,100);
      frame.setLocation(300,250);
      frame.setVisible(true);
      frame.setResizable(false);
      btn.addActionListener(new ActionListener(){
       public void actionPerformed(ActionEvent e){
        String symbols = textIn.getText();

textOut.setText(halfStringToUpperCase(symbols));
        }
       });
       ...
      }
     }
```

Принцип DRY (Don't Repeat Yourself)

Принцип DRY наголошує на тому, що в програмі не повинно бути повторюваних блоків коду, і, відповідно, не повинно бути випадків, коли в декілька частин програми зміни завжди повинні вноситись паралельно.

Некоректний підхід

```
public void makeTransfer(){
 try{
  userIsLoggedIn();
  userHasEnoughRights();
  userAccountIsActive();
  makeTransfer ();
 }
 catch(Exception e){
  System.out.println("Операція                не
дозволена
```

Коректний підхід

```
public void checkUser() throws userException{
  userIsLoggedIn();
  userHasEnoughRights();
  userAccountIsActive();
}
public void makeTransfer(){
 try{
  checkUser()
  makeTransfer ();
 }
```

```
для даного користувача");
 }
}
public void makeDeposit(){
 try{
  userIsLoggedIn();
  userHasEnoughRights();
  userAccountIsActive();
  makeDepositOperation();
 }
 catch(Exception e){
  System.out.println("Операція не
дозволена
для даного користувача");
 }
}
```

```
 catch(Exception e){
  System.out.println("Операція не дозволена
для
даного користувача");
 }
}
public void makeDeposit(){
 try{
  checkUser()
  makeDepositOperation();
 }
 catch(Exception e){
  System.out.println("Операція не дозволена
для
даного користувача");
 }
}
```

Чітка визначеність та одиничність мети методу

Потрібно намагатись уникати написання в одному методі блоків коду, що мають різне по суті призначення. Зокрема, комбінування в одному методі введення даних через засоби інтерфейсу користувача, обробку даних та виведення.

Некоректний підхід

```
class StringProcessing{
 public static void countLetters(char letter) {
  String initialString = textIn.getText();
  int counter=0;
  for(int i=0; i<initialString.length();i++)
   if (initialString.charAt(i)==letter)
    counter++;
  textOut.setText(""+counter);
 }
 public static void main(String[] args){
  JFrame frame=new JFrame("Laba 1 ");
  frame.setDefaultCloseOperation(

JFrame.EXIT_ON_CLOSE);
  JButton btn=new JButton("Start");
  Label infoLabel=new Label("Lababoratorna1",

Label.CENTER);
  JTextField textIn=new JTextField();
  JTextField textOut=new JTextField();
  frame.getContentPane().add(infoLabel);
  frame.getContentPane().add(textIn,

BorderLayout.NORTH);
  frame.getContentPane().add(textOut,

BorderLayout.SOUTH);

frame.getContentPane().add(btn,BorderLayout.EAST);
```

Коректний підхід

```
class StringProcessing{
 public static int countLetters(String initialString,
                                char
letter) {
  int counter=0;
  for(int i=0; i<initialString.length();i++)
   if (initialString.charAt(i)==letter)
    counter++;
  return  counter;
 }
 public static void main(String[] args){
  JFrame frame=new JFrame("Laba 1 ");
  frame.setDefaultCloseOperation(

JFrame.EXIT_ON_CLOSE);
  JButton btn=new JButton("Start");
  Label infoLabel=new Label("Lababoratorna1",

Label.CENTER);
  JTextField textIn=new JTextField();
  JTextField textOut=new JTextField();
  frame.getContentPane().add(infoLabel);
  frame.getContentPane().add(textIn,

BorderLayout.NORTH);
  frame.getContentPane().add(textOut,

BorderLayout.SOUTH);
  frame.getContentPane().add(btn,BorderLayout.EAST);
```

```
    frame.setSize(450,100);
    frame.setLocation(300,250);
    frame.setVisible(true);
    frame.setResizable(false);
    btn.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
        countLetters('a');
      }
    });
  }
}
```

```
    frame.setSize(450,100);
    frame.setLocation(300,250);
    frame.setVisible(true);
    frame.setResizable(false);
    btn.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
        String initialString = textIn.getText();
        textOut.setText(""+countLetters(initialString,'a'));
      }
    });
  }
}
```

7.4. Інкапсуляція

Поля прийнято описувати як private, а для доступу до них створювати спеціальні get та set методи.

Некоректний підхід	*Коректний підхід*

public class Man {
public String name;
public int age;
...
}
public class Run{
public static void main(String[] args){
Man man=new Man();
man.name="John";
...
 	
System.out.println("Name="+man.name);
}
}
	public class Man {
	private String name;
	private int age;
	public String getName(){
	return name;
	}
	public void setName(String name){
	this.name=name;
	}
	public int getAge(){
	return age;
	}
	public void setAge(int age){
	this.age=age;
	}
	...
	}
	public class Run{
	public static void main(String[] args){
	Man man=new Man();
	man.setName("John");
	...
	
	System.out.println("Name="+man.**getName**());
	}
	}

Поля прийнято описувати як private, особливо, якщо для них існують спеціальні public get та set методи. Якщо клас описаний як public, то його конструктор теж варто зробити public, якщо немає іншого спеціального методу для створення об’єктів даного класу, наприклад, фабрики об’єктів.

Некоректний підхід	*Коректний підхід*

public class Dog {
int age;
String breed;
Dog {
...
}
public int getAge(){
return age;
}
public int getBreed(){
return breed;
}
public void setAge(int age){
this.age=age;
}
public void setBreed(int breed){
this.breed=breed;
}
public void barks(){
...
}
}
	public class Dog {
	private int age;
	private String breed;
	public Dog {
	...
	}
	public int getAge(){
	return age;
	}
	public int getBreed(){
	return breed;
	}
	public void setAge(int age){
	this.age=age;
	}
	public void setBreed(int breed){
	this.breed=breed;
	}
	public void barks(){
	...
	}
	}

8. UML

8.1. Знайомство з UML

Універсальна мова моделювання (Unified Modelling Language або UML) – є спільноцільовою мовою візуального моделювання, яку розроблено для специфікації, візуалізації, проектування та документування компонентів програмного забезпечення, бізнес-процесів та інших систем.

Мова UML одночасно є простим і потужним засобом моделювання, який може бути ефективно використаний для побудови концептуальних, логічних та графічних моделей складних систем самого різного цільового призначення. Ця мова увібрала в себе найкращі якості методів програмної інженерії, які з успіхом використовувалися впродовж останніх років при моделюванні великих і складних систем.

Якщо подивитися специфікацію UML, то можна помітити деяку її надмірність. Сама специфікація займає близько 900 сторінок формату А4. На щастя, для читання UML-діаграм потрібно знати тільки умовні позначки, що застосовуються в UML.

Мова UML заснована на деякому числі базових понять, які можуть бути вивчені й застосовані більшістю програмістів і розробників, знайомих з методами об'єктно-орієнтованого аналізу і проектування. При цьому базові поняття можуть комбінуватися і розширюватися таким чином, що фахівці об'єктного моделювання отримують можливість самостійно розробляти моделі великих та складних систем у самих різних областях додатків.

Конструктивне використання мови UML ґрунтується на розумінні загальних принципів моделювання складних систем та особливостей процесу об'єктно-орієнтованого аналізу і проектування зокрема. Вибір засобів для побудови моделей складних систем зумовлює ті завдання, які можуть бути вирішені з використанням даних моделей.

При цьому одним з основних принципів побудови моделей складних систем є принцип абстрагування, який наказує включати в модель тільки ті аспекти проектованої системи, які мають безпосереднє відношення до виконання системою своїх функцій або свого цільового призначення. При цьому всі другорядні деталі опускаються, щоб надмірно не ускладнювати процес аналізу та дослідження отриманої моделі.

Іншим принципом побудови моделей складних систем є принцип багатомодельності. Цей принцип є твердженням про те, що жодна єдина модель не може з достатнім ступенем адекватності описувати різні аспекти складної системи. Стосовно до методології ООАП це означає, що достатньо повно модель складної системи допускає деяке число взаємопов'язаних уявлень (views), кожне з яких адекватно відображає певний аспект поведінки або структури системи.

При цьому найбільш загальними уявленнями складної системи прийнято вважати статичне і динамічне представлення, які у свою чергу можуть підрозділятися на інші більш приватні уявлення. Феномен складної системи якраз і полягає в тому, що ніяке її єдине подання не є достатнім для адекватного вираження всіх особливостей модельованої системи.

Словник UML:

Модель представляється у вигляді сутностей і відносин між ними, які показуються на діаграмах.

Сутності – це абстракції, які є основними елементами моделей. Є чотири типи сутностей**:**

- структурні (клас, інтерфейс, компонент, варіант використання, кооперація, вузол),
- поведінкові (взаємодія, стан),
- групуючі (пакети)
- анотаційні (коментарі).

Кожен вид сутностей має своє графічне представлення.

Відносини показують різні зв'язки між сутностями. У UML визначено такі типи відносин:

- *Залежність* показує такий зв'язок між двома сутностями, коли зміна однієї з них – незалежної, може вплинути на семантику іншої - залежної. Залежність зображується пунктирною стрілкою, спрямованої від залежної сутностіі до незалежної.
- *Асоціація* – це структурне ставлення, яке показує, що об'єкти однієї сутності пов'язані з об'єктами іншої. Графічно асоціація показується у вигляді лінії, що з'єднує зв'язувані сутності. Асоціації служать для здійснення навігації між об'єктами. Наприклад, асоціація між класами <Замовлення> та <Товар> може бути використана для знаходження усіх товарів, зазначених у конкретному замовленні, з одного боку, або для знаходження усіх замовлень в яких є даний товар, з іншого. Зрозуміло, що у відповідних програмах повинен бути реалізований механізм, що забезпечує таку навігацію. Якщо потрібна навігація тільки в одному напрямку, вона показується стрілкою на кінці асоціації. Окремим випадком асоціації є агрегування - відношення виду <ціле> – <частина>. Графічно воно виділяється за допомогою ромбика на кінці близько сутності-цілого.
- *Узагальнення* – це відношення між сутністю-батьком і сутністю-нащадком. По суті, це ставлення відображає властивість наслідування для класів та об'єктів. Узагальнення показується у вигляді лінії, що закінчується трикутником спрямованим до батьківської сутності. Нащадок успадковує структуру (атрибути) і поведінку (методи) батька, але в той же час він може мати нові елементи структури і нові методи. UML допускає множинне спадкування, коли сутність пов'язана більш ніж з однієї батьківської сутністю.
- *Реалізація* – відношення між сутністю, яка визначає специфікацію поведінки (інтерфейс) із сутністю, яка визначає реалізацію цієї поведінки (клас, компонент). Це відношення зазвичай використовується при моделюванні компонент.

У UML використовуються наступні види діаграм:

1. **Structure Diagrams (Структурні діаграми)**:

- Class diagram (Класів)
- Component diagram (Компонентів)
- Composite structure diagram (Композитної структури)
    - Collaboration (Кооперації)
- Deployment diagram (Розгортання)
- Object diagram (Об'єктів)
- Package diagram (Пакетів)
- Profile diagram (Профілів)

2. **Behavior Diagrams (Діаграми поведінки)**:

- Activity diagram (Діяльності)
- State Machine diagram (Станів)
- Use case diagram (Варіантів використання)
- Interaction Diagrams (Діаграми взаємодії):
    - Communication diagram (Комунікації) / Collaboration (Кооперації)
    - Interaction overview diagram (Огляду взаємодії)
    - Sequence diagram (Послідовності)
    - Timing diagram (Синхронізації)

8.2. Діаграми класів

На діаграмах класів буде показано різноманітні класи, які утворюють систему і їх взаємозв'язки. Діаграми класів називають «статичними діаграмами», оскільки на них показано класи разом з методами і атрибутами, а також статичний взаємозв'язок між ними: те, яким класам «відомо» про існування яких класів, і те, які класи «є частиною» інших класів, – але не показано методи, які при цьому викликаються.

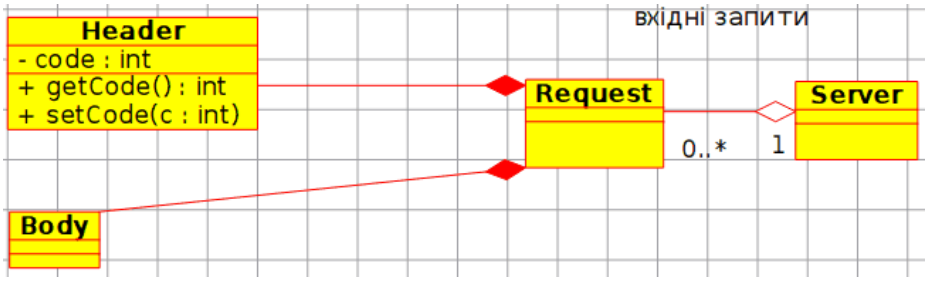

Рис. 3. Діаграми класів

8.2.1. Клас

Клас визначає атрибути і методи набору об'єктів. Всі об'єкти цього класу (екземпляри цього класу) мають спільну поведінку і однаковий набір атрибутів (кожен з об'єктів має свій власний набір значень). Іноді замість назви «клас» використовують назву «тип», але, слід зауважити, що ці назви описують різні речі: тип є загальнішим визначенням.

У UML класи позначаються прямокутниками з назвою класу, у цих прямокутниках у вигляді двох «відсіків» може бути показано атрибути і операції класу.

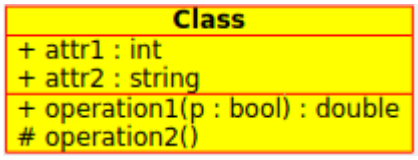

Рис. 4. Представлення класу в UML

8.2.2. Атрибути

У UML атрибути показуються щонайменше назвою, також може бути показано їх тип, початкове значення і інші властивості. Крім того, атрибути може бути показано з областю видимості атрибута:

- + відповідає публічним (public) атрибутам
- # відповідає захищеним (protected) атрибутам
- - відповідає приватним (private) атрибутам

8.2.3. Операції

Операції (методи) також показуються принаймні назвою, крім того, може бути показано їх параметри і типи значень, які буде повернуто. Операції, як і атрибути, може бути показано з областю видимості:

- + відповідає публічним (public) операціям
- # відповідає захищеним (protected) операціям
- - відповідає приватним (private) операціям

8.2.4. Шаблони

Серед класів можуть бути шаблони, значення, які використовуються для невизначеного класу або типу. Тип шаблону визначається під час ініціалізації класу (тобто, під час створення об’єкта). Шаблони існують у сучасних мовах програмування C++, Java, але в Джаву вони прийшли з версії 1.5 і отримали назву Generic.

8.3. Асоціації класів

8.3.1. Узагальнення

Наслідування є однією з фундаментальних основ об’єктно-орієнтованого програмування, у якому клас «отримує» всі атрибути і операції класу, нащадком якого він є, і може перевизначати або змінювати деякі з них, а також додавати власні атрибути і операції.

У UML пов’язування Узагальнення між двома класами розташовує їх у вузлах ієрархії, яка відповідає концепції успадкування класу-нащадка від базового класу. У UML узагальнення буде показано у вигляді лінії, яка поєднує два класи, зі стрілкою, яку спрямовано від базового класу.

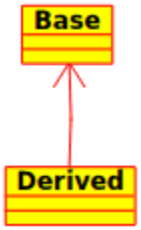

Рис. 5. Узагальнення

8.3.2. Асоціації

Асоціація означає взаємозв’язок між класами, вона є базовим семантичним елементом і структурою для багатьох типів «з’єднань» між об’єктами.

Асоціації є тим механізмом, який надає об’єктам змогу обмінюватися даними між собою. Асоціація описує з’єднання між різними класами (з’єднання між дійсними об’єктами називається об’єктним з’єднанням, або зв’язком).

Асоціації можуть виконувати роль, яка визначає призначення асоціації і може бути одно- чи двосторонньою (другий варіант означає, що у межах зв’язку кожен з об’єктів може надсилати повідомлення іншому, перший же — варіанту, коли лише один з об’єктів знає про існування іншого). Крім того, кожен з кінців асоціації має значення численності, яке визначає кількість об’єктів на відповідному кінці асоціації, які можуть мати зв’язок з одним з об’єктів на іншому кінці асоціації.

У UML асоціації позначаються лініями, що з’єднують класи, які беруть участь у зв’язку, крім того, може бути показано роль і численність кожного з учасників зв’язку. Численність буде показано у вигляді діапазону [мін..макс] невід’ємних чисел, зірочка (*) на боці максимального значення позначає нескінченність.

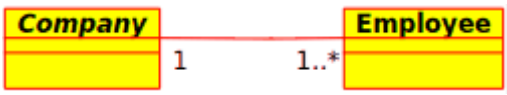

Рис. 6. Асоціація

8.3.3. Агрегація

Агрегації є особливим типом асоціацій, за якого два класи, які беруть участь у зв'язку не є рівнозначними, вони мають зв'язок типу «ціле-частина». За допомогою агрегації можна описати, яким чином клас, який грає роль цілого, складається з інших класів, які грають роль частин. У агрегаціях клас, який грає роль цілого, завжди має численність рівну одиниці.

У UML агрегації буде показано асоціаціями, у яких з боку цілої частини буде намальовано ромб.

Рис. 7. Агрегація

8.3.4. Композиція

Композиції – це асоціації, які відповідають дуже сильній агрегації. Це означає, що у композиціях ми також маємо справу з співвідношеннями ціле-частина, але тут зв'язок є настільки сильним, що частини не можуть існувати без цілого. Вони існують лише у межах цілого, після знищення цілого буде знищено і його частини.

У UML композиції буде показано як асоціації з зафарбованим ромбом з боку цілого

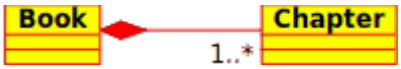

Рис. 8. Композиція

8.4. Діаграми послідовностей

На діаграмах послідовностей буде показано обмін повідомленнями (тобто виклик методів) між декількома об'єктами у окремій обмеженій часом ситуації. Об'єкти є екземплярами класів. Основний наголос на діаграмах послідовностей робиться на порядок і моментах часу, у які повідомлення надсилаються об'єктам.

На діаграмах послідовностей об'єкти буде показано вертикальними штриховими лініями з назвою об'єкта над ними. Вісь часу також має вертикальний напрямок, її спрямовано вниз, повідомлення, які надсилаються від одного об'єкта до іншого, буде позначено стрілками з назвами операції і параметрів.

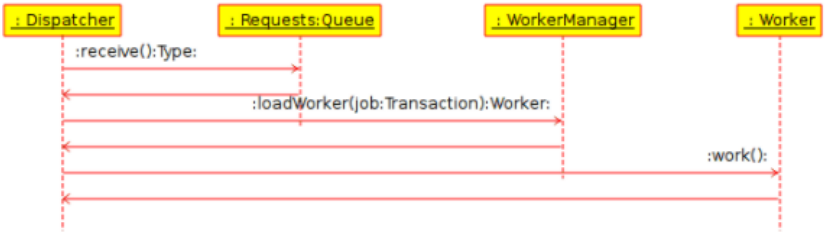

Рис. 9. Діаграми послідовностей

Повідомлення можуть бути або синхронними, звичайного типу повідомленнями, за виклику яких керування передається викликаному об'єкту

до завершення виконання методу, або асинхронними, за виклику яких керування передається назад напряму об’єкту, який здійснював виклик. За використання синхронного повідомлення збоку від викликаного об’єкта буде показано вертикальний блок, який показуватиме перебіг виконання програми.

8.5. Діаграма випадків використання

Діаграми випадків використання описують взаємозв’язки і залежності між групою випадків використання і акторами, що беруть участь у процесі.

Важливо зауважити, що діаграми випадків використання не призначено для показу компонування, вони не можуть описати внутрішню структуру системи. Діаграми випадків використання призначено для полегшення обміну інформацією між майбутніми користувачами системи і замовником, вони особливо корисні для визначення переліку можливостей, які повинна мати система.

За діаграмами випадків використання можна сказати, що система має робити, але не те, як вона досягає потрібних результатів, для останнього ці діаграми просто не придатні.

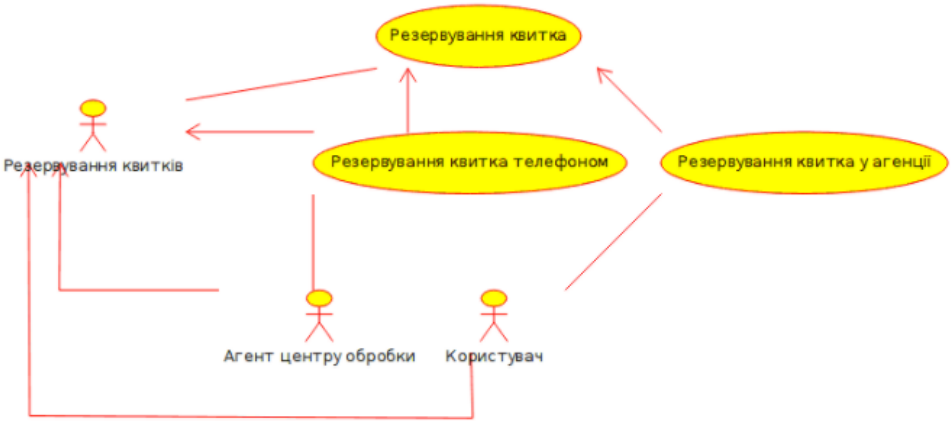

Рис. 10. Діаграми послідовностей

8.5.1. Випадок використання

Випадок використання визначає, з точки зору акторів (користувачів), групу дій у системі, які призводять до конкретного видимого результату.

Випадки використання є описом типових елементів взаємодії користувачів системи з самою системою. Вони відповідають зовнішньому інтерфейсу системи і визначають форму вимог до того, що має робити система (зауважте, лише «що», а не «як»).

Під час роботи з випадками використання важливо пам’ятати декілька простих правил:

Кожен випадок використання має бути пов’язано принаймні з одним актором.

У кожного з випадків використання має бути ініціатор (тобто актор)

Кожен з випадків використання має призводити до відповідного результату (результату з «комерційним значенням»)

Випадки використання можуть мати зв’язки з іншими випадками використання. Ось три найпоширеніших зв’язки між випадками використання:

- «включення», яке вказує на те, що випадок використання відбувається всередині іншого випадку використання
- «розширення», яке означає, що у певних випадках або у певній точці (яку називають точкою розширення) випадок використання буде розширено іншим випадком використання.
- «узагальнення», за якого випадок використання успадковує характеристики випадку використання «вищого рангу», при цьому можливе перевизначення деяких з характеристик у спосіб, подібний до успадкування між класами.

8.5.2. Актор

Актор – це зовнішній чинник (поза межами системи), який взаємодіє з системою шляхом участі (і часто ініціювання) у випадку використання. Акторами, на практиці, можуть бути звичайні люди (наприклад, користувачі системи), інші комп’ютерні системи або зовнішні події.

Акторам відповідають не реальні люди або системи, а лише їх ролі. Це означає, що коли особа у різний спосіб взаємодіє з системою (виконуючи різні ролі), їй відповідають декілька акторів. Наприклад, особа, яка виконує підтримку користувачів телефоном і приймає замовлення від користувачів до системи, може бути показано актором «Персонал служби підтримки» і актором «Відповідальний за продажі».

8.5.3. Опис випадків використання

Описи випадків використання – це текстові примітки до випадків використання. Зазвичай, вони мають форму нотаток або документа, який певним чином пов’язано з випадком використання, і який пояснює процеси або дії, які відбуваються під час випадку використання.

8.6. Діаграми співпраці

На діаграмах співпраці показується взаємодія між об’єктами, які беруть участь у певній події. Ця інформація більшою чи меншою мірою подібна до інформації, показаної на діаграмі послідовностей, але там наголос робиться на часовій характеристиці взаємодії, а на діаграмах співпраці основний наголос робиться на взаємодії між об’єктами та її топології на передньому плані.

На діаграмах співпраці повідомлення надіслані від одного з об’єктів до іншого позначаються стрілочками, поряд з якими показано назву повідомлення, параметри і послідовність повідомлення. Діаграми співпраці найкраще пасують для показу специфічного перебігу виконання або ситуацій у програмі. Такі діаграми є найкращим засобом для швидкого показу і пояснення окремого процесу у програмній логіці.

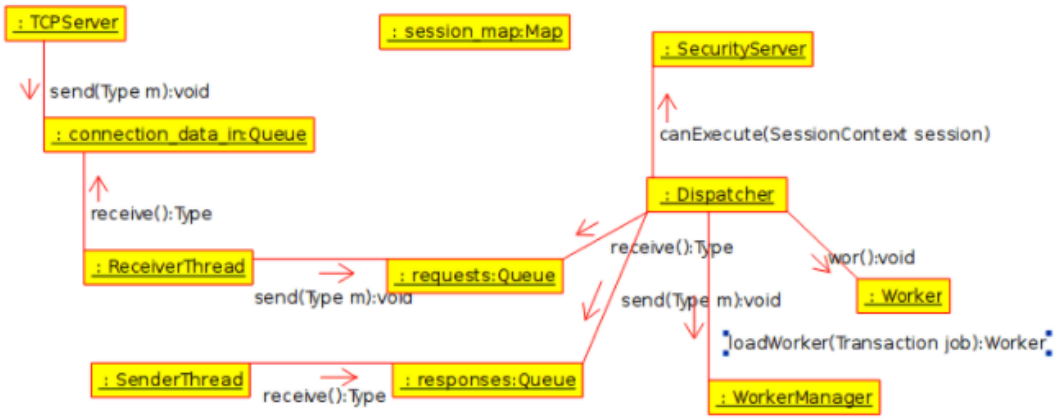

Рис. 11. Діаграми співпраці

8.7. Діаграма станів

На діаграмах станів зображають різні стани об’єкта під час його існування і стимули, які призводять до переходу об’єкта з одного стану у інший.

На діаграмах стану об’єкти розглядаються як машини станів або скінченні автомати, які можуть перебувати у одному зі станів скінченного набору станів, і які можуть змінювати цей стан через вплив одного зі стимулів зі скінченного набору стимулів. Наприклад, об’єкт типу Сервер мережі може перебувати у одному з таких станів протягом існування:

- Готовність
- Очікування
- Робота
- Зупинка

а подіями, які можуть спричинити зміну стану об’єкта можуть бути:

- Створення об’єкта
- Об’єкт отримує повідомлення «очікувати»
- Клієнт надсилає запит на з’єднання мережею
- Клієнт перериває запит
- Запит виконано і перервано
- Об’єкт отримує повідомлення «зупинка»

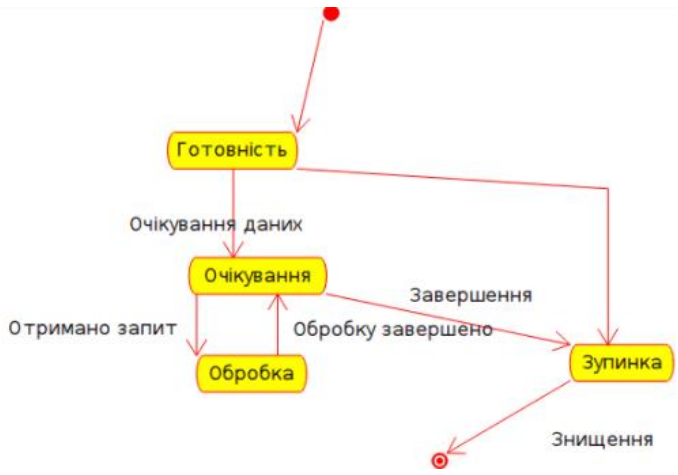

Рис. 12. Діаграми станів

8.7.1. Стан

Будівельними цеглинками діаграм станів є стани. Стан належить лише одному класу і відповідає переліку значень атрибутів, які може приймати клас. У UML стан описує внутрішній стан об'єкта одного з окремих класів

Зауважте, що не кожну зміну одного з атрибутів об'єкта має бути показано станом, станам відповідають лише ті зміни, які значно впливають на виконання об'єктом завдань.

Існує два особливих типи станів: початок і кінець. Їх особливість полягає у тому, що не існує жодної події, яка може спричинити повернення об'єкта до його початкового стану, так само, не існує жодної події, яка б могла повернути об'єкт зі стану кінця, тільки-но він його досягне.

8.8. Діаграма діяльності

На діаграмі діяльності буде показано послідовність актів дій системи на основі Діяльностей. Діаграми діяльності є особливою формою діаграм стану, на яких містяться лише (або головним чином) діяльності.

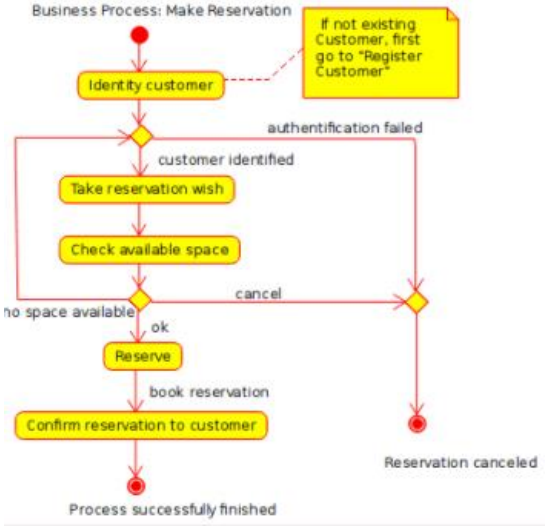

Рис. 13. Діаграми діяльності

Діаграми діяльності подібні до процедурних діаграм потоку, але відрізняються від них тим, що діяльності точно прив'язано до об'єктів.

Діаграми діяльності завжди пов'язано з класом, операцією або випадком використання.

На діаграмах діяльності може бути показано як послідовні, так і паралельні діяльності. Паралельне виконання буде показано за допомогою піктограм Розділити/Чекати, для діяльностей, які виконуються паралельно, неважливим є порядок їх обробки (їх може бути виконано одночасно або одну за одною).

8.8.1. Діяльність

Діяльність є окремим кроком у процесі. Одній діяльності відповідає окремий стан у системі з внутрішньою діяльністю і, принаймні, одна вихідна

транзакція. Крім того, діяльності можуть мати декілька вихідних транзакцій, якщо умови цих транзакцій є різними.

Діяльності можуть формувати ієрархічні структури, це означає, що діяльність може бути складено з декількох «менших» діяльностей, у цьому випадку вхідні і вихідні транзакції мають відповідати вхідним і вихідним транзакціям докладної діаграми.

8.8.2. Допоміжні елементи

У UML є декілька елементів, які не мають реального семантичного змісту для моделі, але допомагають прояснити частини діаграми. Цими елементами є:

- Рядки тексту
- Текстові нотатки і якорі
- Блоки

Рядки тексту можуть знадобитися, якщо до діаграми слід додати коротку текстову інформацію. Вони є довільно розташованим тестом і не мають значення для самої моделі.

Нотатками можна скористатися для додавання докладніших відомостей щодо об’єкта або певної ситуації. У них є велика перевага у тому, що нотатки можна пов’язати з елементами UML, щоб було видно, що нотатка «стосується» певного об’єкта або ситуації.

Блоки є довільно розташованими прямокутниками, які можна використовувати для групування об’єктів діаграми, яке зробить діаграму зрозумілішою. Вони не мають логічного навантаження у межах моделі.

8.9. Діаграми взаємозв’язків сутностей

На діаграмах взаємозв’язку сутностей (діаграмах ВС (ER)) показують концептуальний дизайн програм для роботи з базами даних. На них показують різноманітні сутності (концепти) у інформаційній системі і існуючі взаємозв’язки і обмеження між ними. Розширення діаграм взаємозв’язку сутностей називають «Розширеними діаграмами взаємозв’язку сутностей» або «Покращеними діаграмами взаємозв’язку сутностей» (EER), їх використовують для інтеграції методик компонування орієнтованих на об’єкти у діаграми ВС.

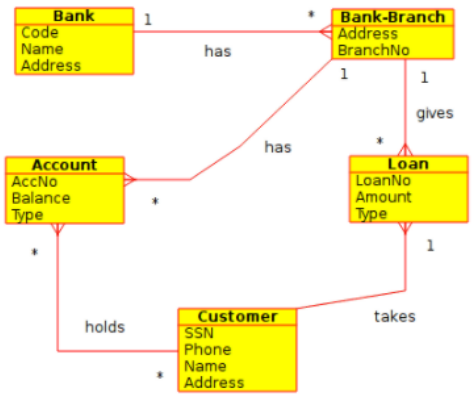

Рис. 14. Діаграми взаємозв’язків сутностей

8.10. Діаграми компонентів

На діаграмах компонентів буде показано компоненти програмного забезпечення (або технології компонентів, такі як KParts, компоненти CORBA або Java Beans, або просто розділи системи, які чітко відрізняються один від одного), а також елементи, з яких вони складаються, такі як файли з початковими кодами, програмні бібліотеки або таблиці реляційних баз даних.

Компоненти можуть мати інтерфейси (тобто абстрактні класи з операціями), які надають змогу створювати асоціації між компонентами.

8.11. Діаграми впровадження

На діаграмах впровадження буде показано екземпляри компонентів та їх асоціації. На них буде показано вузли, які є фізичними ресурсами, типово, окремими комп’ютерами. Крім того, на них показують інтерфейси і об’єкти (екземпляри класів).

ЛІТЕРАТУРА

1. Інформатика: Комп’ютерна техніка. Комп’ютерні технології. Посіб. / За ред. О.І.Пушкаря – К: Видавничий центр “Академія”, 2001. – 696с. (Альма-матер)
2. Макконнелл С. Совершенный код. Мастер-класс / Пер. с англ.- М.: Издательско-торговый дом "Русская Редакция"; СПб.: Питер, 2005.- 896 стр.: ил.
3. Основы Программной Инженерии (по SWEBOK). 3. Конструирование програмного обеспечения. http://swebok.sorlik.ru/3_software_construction.html.
4. Bohm, Corrado; and Giuseppe Jacopini (May 1966). "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". Communications of the ACM 9 (5): 366–371.
5. Програмування на Java, Gorban
6. Dijkstra, E. W. (Aug 1972). "The Humble Programmer". Communications of the ACM 15 (10): 859–866.
   http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html. (EWD 340) PDF, 1972 ACM Turing Award lecture
7. Dijkstra, E.W., "Structured Programming," Software Engineering Techniques, Buxton, J.N., and Randell, B., eds. Brussels, Belgium, NATO Science Committee, 1969.
8. Структури даних і алгоритми Java. Роберт Лафоре
9. B. Meyer, Object-Oriented Software Construction, second ed., Prentice Hall, 1997, Chap. 6, 10, 11.
10. Діаграми UML. https://www.omg.org/
11. Guide to the Software Engineering Body of Knowledge (SWEBOK). CHAPTER 4.SOFTWARECONSTRUCTION.http://www.computer.org/portal/web/swebok/html/ch4K. Beck, Test-Driven Development: By Example, Addison-Wesley, 2002.
12. McCabe : Complexity Measure, IEEE Transacions on Software Engineering, Volume 2, No 4, pp 308-320, December 1976
13. M. Fowler and al., Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2002.
14. Breiman, Leo (2001). Random Forests. Machine Learning[en] 45 (1). с. 5–32.
15. Опис алгоритму на сайті Лео Брейман.
16. Некорректная работа функции сортировки в Android, Java и Python.
17. Russell Gold, Thomas Hammell, Tom Snyder. Test Driven Development: A J2EE Example.- Apress, 2005.- 296 pages.