МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

КОНСПЕКТ ЛЕКЦІЙ

з дисципліни «Крос-платформне програмування»
(для студентів усіх форм навчання напряму підготовки
121 «Інженерія програмного забезпечення»)

З А Т В Е Р Д Ж Е Н О
на засіданні кафедри
інформаційних технологій
та програмування

Протокол № 7 від 14.03.25

Київ 2025

УДК 004.43

Конспект лекцій з дисципліни «Крос-платформне програмування» (для студентів усіх форм навчання напряму підготовки 121 «Інженерія програмного забезпечення») (Електронне видання) / Уклад.: Дьомін М. К. - Київ: Вид–во СНУ ім. В. Даля, 2025. – 83 с.

Укладено на основі ОП підготовки бакалаврів напряму 121 «Інженерія програмного забезпечення та робочої навчальної програми з дисципліни «Крос-платформне програмування».

Укладач: к.т.н., доц. М.К.Дьомін

Резензент: д.т.н., доц. О.І. Захожай

Зміст

Лекція 1. Вступ до крос-платформної розробки

Крос-платформна розробка — це процес створення програмного забезпечення, яке може працювати на кількох операційних системах, таких як Windows, macOS і Linux або iOS чи Android, використовуючи спільну кодову базу. Це дозволяє розробникам писати код один раз і розгортати його на різних платформах. Крос-платформна розробка забезпечує уніфікований підхід до створення програмного забезпечення для різноманітної аудиторії.

Раніше розробникам доводилося писати абсолютно окремі кодові бази для кожної платформи. Зараз спеціалізовані фреймворки та інструменти забезпечують загальну кодову базу та абстрагують специфічні для платформи компоненти, що дозволяє розробникам зосередитися на написанні надійних, багатофункціональних програм. Це дозволяє розробникам писати код один раз і розгортати його на різних платформах Результатом є більш ефективний процес розробки, який забезпечує узгодженість у всіх підтримуваних середовищах. Серед популярних фреймворків розробки слід зазначити Electron, Qt, wxWidgets, React Native, Flutter, Xamarin.

Отже найважливішою рисою крос-платформної розробки є використання єдиної кодової бази для декількох платформ. Замість того, щоб дублювати зусилля для кожної операційної системи, розробники пишуть один раз і розгортають всюди. Це не тільки економить час, але й спрощує технічне обслуговування, оскільки оновлення або виправлення помилок здійснюються в одному місці та відображаються повсюдно.

Слід зазначити, що у спільній кодовій базі може зустрічатися код, який виконується лише на одній з цільових платформ. Цього не можливо уникнути, тому що відмінності між цільовими платформами бувають доволі значущими. Але кількість такого коду зазвичай не перевищує 5%. Також трапляється, що помилки проявляються лише в одній з цільових операційних систем. Тобто не можна стверджувати, що підхід написати код один раз, я розгорнути всюди завжди працює на всі 100%, але він працює для переважної більшості завдань, що виникають під час циклу розробки програмного забезпечення. Але зважаючи на ці особливість, тестування програмного забезпечення потрібно виконувати на всіх цільових платформах окремо.

Значення крос-платформної розробка можна зрозуміти через такі ключові **переваги**:

Економічна ефективність. Розробка окремих програм для кожної платформи потребує значних ресурсів, у тому числі кількох команд розробників і тривалого часу. Крос-платформна розробка зменшує витрати, об’єднуючи ці зусилля в єдиний робочий процес.

Ширше охоплення аудиторії. Завдяки підтримці кількох платформ крос-платформні програми охоплюють ширшу аудиторію. Підприємства можуть обслуговувати користувачів незалежно від того, яким пристроєм чи операційною системою вони віддають перевагу, підвищуючи доступність і охоплення аудиторії.

Швидший час виходу на ринок. Час є критичним фактором у розробці програмного забезпечення, особливо на конкурентних ринках. Єдина кодова база дозволяє розробникам розгортати оновлення, нові функції та навіть початковий продукт набагато швидше, ніж підходи, пов'язані з платформою.

Узгодженість між платформами. Уніфікована кодова база забезпечує узгоджену взаємодію з користувачем. Незалежно від того, чи користуються вони смартфоном, настільним комп'ютером чи планшетом, користувачі взаємодіють із програмою однаково, що підвищує лояльність користувача до продукту.

Крос-платформну розробку можна бути організована за допомогою різних підходів, кожен зі своїми перевагами та недоліками:

Гібридні технології поєднують такі веб-технології, як HTML, CSS і JavaScript, із власними функціями за допомогою фреймворків, таких як Apache Cordova або Ionic. Хоча їм може не вистачати продуктивності нативних програм, вони є економічно ефективними та легкими в розробці.

Нативна кросплатформенність. Власні крос-платформні фреймворки, такі як Flutter або Xamarin, компілюються у рідний код для кожної платформи. Цей підхід забезпечує майже нативну продуктивність і зовнішній вигляд, зберігаючи переваги спільної кодової бази.

Рішення на основі середовища виконання. Програми, створені за допомогою середовищ виконання, таких як Electron або віртуальна машина Java (JVM), виконуються в контейнері, який усуває розрив між програмою та платформою.

При крос-платформній розробці слід враховувати **тип програмного забезпечення**, що розробляється, бо для різних типів ПЗ зазвичай використовуються різні підходи. До типів ПЗ у цьому розумінні будемо відносити:

Настільне програмне забезпечення (desktop software). наприклад, браузери, ПЗ для дизайну. Настільне ПЗ, що вимагає максимальної швидкодії, зазвичай розробляється за допомогою мов, бібліотек та фреймворків, що дозволяють компілювати код безпосередньо у формат цільової платформи. Наприклад, браузер Chrome написаний здебільшого на C++, його платформно-залежні частини можуть використовувати інші мови як то Java для Android чи Objective C для iOS. У разі, якщо настільне програмне забезпечення має також веб версію доцільним є використання іншого підходу, наприклад, написання такого програмного забезпечення на JavaScript з використанням фреймворку Electron. Можно стверджувати, що цей підхід зараз є домінуючим при створенні настільного програмного забезпечення для широкого кругу користувачів.

Мобільне програмне забезпечення, наприклад, месенджери, програми для продуктивності. Для створення мобільного програмного забезпечення використовуються такі інструменти як React-Native, Flutter чи Xamarin.

Серверне програмне забезпечення наприклад, веб-сервери, СКБД, балансувальники навантаження. Для створення серверного крос-платформного ПЗ часто використовуються такі мови, як C++ чи Java. У разі якщо створюється серверна частина (backend) для веб сайту чи мобільного застосунку може використовуватися весь набір мов та засобів інтернет-технологій (php, JS + node JS, python, тощо).

Ігри. Для створення крос-платформних ігор зараз найчастіше використовуються такі засоби як Unity чи Unreal Engine.

Крос-платформна розробка зробила революцію в інженерії програмного забезпечення, дозволивши розробникам створювати програми, які безперебійно працюють у кількох операційних системах. Однак цей підхід не позбавлений недоліків. Впровадження крос-платформних рішень часто виявляє значні перешкоди. Далі розглядаються **основні проблеми крос-платформної розробки** та те, як вони можуть вплинути на процес створення програмного забезпечення.

Обмеження продуктивності. Однією з найпоширеніших проблем у крос-платформній розробці є продуктивність. Програми, створені з використанням крос-платформних фреймворків, часто мають накладні витрати на продуктивність порівняно з рідними програмами. Це значною мірою відбувається тому, що багато фреймворків покладаються на рівні абстракції або середовища виконання, щоб подолати розрив між кодом і цільовою платформою.

Наприклад, такі фреймворки, як Electron, об’єднують повноцінний веб-браузер і середовище виконання JavaScript для відтворення настільних програм, що може призвести до сповільнення виконання та збільшення використання пам’яті. Хоча це може бути непомітним для легких програм, це стає критичною проблемою для високопродуктивних програм, таких як ігри чи засоби редагування мультимедіа. Розробники, яким потрібна нативна швидкість і оперативність, часто повинні докладати додаткових зусиль для оптимізації.

Доступ до унікальних функцій платформи. Такі операційні системи, як Windows, macOS і Linux, пропонують унікальні функції та API, які можуть не повністю підтримуватися крос-платформними фреймворками. Ці специфічні для платформи можливості, такі як вдосконалена обробка графіки або спеціалізовані методи введення, часто вимагають спеціального кодування для ефективної інтеграції.

Розглянемо сценарій, коли розробник хоче включити функцію сенсорної панелі macOS або DirectX Windows для високопродуктивної графіки. У крос-платформних фреймворках може бути відсутнім пряма підтримка цих функцій, що змушує розробників писати код для конкретної платформи. Це порушує головну мету підтримки єдиної кодової бази та ускладнює процес розробки.

Узгодженість інтерфейсу користувача (UI). Створення користувальницького інтерфейсу, який буде одночасно узгодженим між платформами та та виглядати органічно на кожній з них, є одним із

найскладніших аспектів крос-платформної розробки. Різні платформи мають різні вказівки щодо дизайну та очікування користувачів. Наприклад, користувачі macOS очікують панелі меню у верхній частині екрана, а користувачі Windows очікують вбудованих меню у вікно програми.

Програми для Android часто використовують стандарт Material Design, тоді як програми для iOS дотримуються рекомендацій Apple щодо інтерфейсу (Human Interface Guidelines).

Щоб збалансувати ці відмінності при збереженні єдиного дизайну на різних платформах, потрібне ретельне планування та часто потреба в реалізації спеціального інтерфейсу користувача для конкретних платформ.

Складність налагодження та тестування. Тестування є критичним етапом розробки програмного забезпечення, але воно стає особливо складним у крос-платформних проектах. У той час як окрема кодова база може правильно функціонувати на одній платформі, на інших можуть виникати незначні помилки через відмінності в операційних системах, апаратних конфігураціях або середовищах виконання. Наприклад, програма може ідеально працювати в Linux, але матиме проблеми з дозволом на файл у macOS. Відлагодження таких проблем, пов’язаних із певною платформою, може зайняти багато часу та потребує ретельного тестування на всіх підтримуваних платформах. Інструменти автоматизованого тестування допомагають спростити процес, але вони не можуть повністю замінити потребу в ручному тестуванні в реальних сценаріях.

Обмеження інструментів. Інструменти для крос-платформної розробки є потужними, але вони не завжди синхронізуються з останніми досягненнями в операційних системах або апаратному забезпеченні. Наприклад, для інтеграції нової функції Android API або iOS у такі фреймворки, як Flutter або React Native, можуть знадобитися місяці. Тому розробники, які покладаються на ці фреймворки, можуть зіткнутися із затримками у впровадженні найсучасніших функцій.

Це відставання змушує розробників або чекати оновлень, або писати специфічний для платформи код, підриваючи ефективність крос-платформного підходу.

Більший розмір програм. Програми, створені за допомогою крос-платформних фреймворків зазвичай мають більший розмір порівняно з рідними програмами. Це пояснюється тим, що такі фреймворки часто об’єднують додаткові бібліотеки, середовища виконання або механізми для забезпечення сумісності між платформами. Наприклад, програма Electron зазвичай включає Chromium і Node.js, що може значно збільшити розмір програми. Хоча це може не бути проблемою для настільних програм, це може бути проблемою для мобільних користувачів з обмеженим обсягом пам’яті або повільним підключенням до Інтернету. Зменшення розміру програми без втрати функціональності стає додатковим викликом для розробників.

Виклики безпеки. Рівні абстракції та спільні кодові бази, які роблять можливим кросплатформену розробку, також можуть створювати вразливості.

Використання застарілих бібліотек може наражати програми на потенційні експлойти. Крім того, використання інструментів і плагінів сторонніх виробників додає ще один рівень складності в управлінні безпекою.

Розробники повинні бути пильними, регулярно відстежуючи наявність виправлень безпеки, оновлюючи залежності та використовуючи надійні методи тестування. Нехтування безпекою може призвести до вразливості, яка вплине на всі платформи, на яких розгорнуто додаток, посилюючи вплив злому.

Основні **мови програмування**, що використовуються для крос-платформної розробки представлені у таблиці 1. Перелік не є вичерпним, бо зараз майже будь-яка мова не обмежена однією платформою. Але для практичного використання можливості мови компілювати код під якусь платформу або наявності середовища виконання під ці платформу недостатньо для вибору її для створення крос-платформного ПЗ. Важливим також є наявність гарно документованих та підтримуваних бібліотек, фреймворків та інструментів, що відповідають особливостям ПЗ, що розробляється.

Таблиця 1.

Мова	Опис	Приклади використання	Обмеження
JavaScript / Type Script	JavaScript є універсальним, він працює в браузерах і на серверах (через Node.js) і підтримує настільні програми через фреймворки. Популярні фреймворки: React Native для мобільних програм, React або Angular для фронендуу, Electron для настільних програм, Express для серверного ПЗ.	Веб-програми, настільні програми (наприклад, Slack, Visual Studio Code) і мобільні програми.	Продуктивність може відставати від рідних мов у ресурсомістких програмах.
C++	C++ - це високопродуктивна мова, яка дозволяє розробникам створювати крос-платформні програми з мінімальними витратами. Такі фреймворки, як Qt і wxWidgets, надають інструменти для створення програм на основі графічного інтерфейсу користувача, які працюють на кількох платформах також існують бібліотеки, які дозволяють створювати високопродуктивне серверне ПЗ.	Високопродуктивні настільні та серверні програми, ігри та вбудовані системи.	Вимагає більше зусиль для керування пам’яттю та специфічними для платформи особливостями порівняно з мовами вищого рівня.
Java	Філософія Java «напиши один раз, запусти будь-де» є наріжним каменем крос-платформної розробки. Програми, написані на Java, можна запускати на будь-якому пристрої з віртуальною машиною	Корпоративне програмне забезпечення, розробка спеціалізованих серверів.	Потрібна JVM, яка може спричинити додаткові витрати порівняно з повністю рідними програмами.

	Java (JVM), що робить його незалежним від платформи. Популярні фреймворки: Spring для серверних програм і Swing або JavaFX для настільних програм з графічним інтерфейсом користувача.		
Python	Простота Python і великі бібліотеки роблять його ідеальним для швидкої розробки крос-платформних програм. Такі фреймворки, як Kivy та BeeWare, дозволяють створювати настільне та мобільне ПЗ.	Наукові обчислення, штучний інтелект, засоби автоматизації, нескладні настільні або мобільні програми.	Повільніша швидкість виконання порівняно з мовами, що компілюються, такими як C++.
C#	Завдяки .NET Core (.NET 6+) C# пропонує потужні крос-платформні можливості для створення програм у Windows, macOS і Linux. Xamarin розширює функціональність для розробки мобільних додатків.	Корпоративні програми, ігри (через Unity) і мобільні програми.	Сильно пов’язаний з екосистемою Microsoft, деякі функції не працюють на операційних системах, відмінних від Windows.
Dart	Dart, мова, що лежить в основі Flutter, розроблена для створення швидких і крос-платформних програм. Пропонує підхід «напишіть один раз, запустіть будь-де» для веб-додатків, мобільних і настільних програм.	Мобільні програми, прототипування веб-застосунків.	Хоча екосистема Flutter зростає, вона не така зріла, як старіші фреймворки. Також є багато недоліків при застосуванні Flutter для створення веб-застосунків.
Kotlin	Kotlin (що раніше використовувався лише для Android розробки) зараз	В основному для розробки мобільних	Обмеження окремими екосистемами, хоча їхні

	розвинувся та включає крос-платформні можливості через фреймворки, такі як Kotlin Multiplatform.	додатків, але також все частіше використовується для настільних і серверних програм.	крос-платформні інструменти постійно вдосконалюються.
Swift	Swift також деякою мірою можна вважати крос-платформною мовою бо вона дозволяє створювати ПЗ для декількох операційних систем, таких як iOS, iPadOS, watchOS та MacOS, особливо при використанні SwiftUI. Також існують можливості компілювати програми на Swift під інші платформи та використовувати цю мову для створення серверного ПЗ	Розробка додатків для iOS, MacOS, watchOS.	Зазичай використовується виключно в екосистемі Apple. Для розробки серверного ПЗ використовується дуже рідко.

З врахуванням навчальної програми спеціальності та потреб ринку праці України найкращім змістом цього курсу буде створення крос-платформного ПЗ (у тому числі і з графічним інтерфейсом користувача) з використанням мови програмування Java та бібліотек AWT та Swing.

Мова Java довгий час була основної в розробці крос-платформних проектів. Її принцип “Write Once, Run Anywhere” зробив революцію в програмуванні, дозволивши розробникам створювати програмне забезпечення, яке могло б працювати в різних операційних системах. Однак, як і будь-яка технологія, Java має свої сильні та слабкі сторони. Розглянемо детальніше переваги та недоліки використання Java для створення крос-платформного ПЗ. Спочатку відзначимо її переваги.

Незалежність від платформи. Однією з визначальних особливостей Java є її здатність працювати на кількох платформах без змін. Це досягається за допомогою віртуальної машини Java (JVM), яка діє як посередник між кодом і основною операційною системою. JVM перетворює байт-код Java в інструкції для конкретної платформи, дозволяючи програмам працювати в різних середовищах, таких як Windows, macOS і Linux.

Надійна екосистема. Зріла екосистема Java надає безліч інструментів, бібліотек і фреймворків, які спрощують процес розробки. Такі фреймворки, як Spring, спрощують розробку бекенда, тоді як Swing або JavaFX полегшує створення настільних програм. Крім того, доступні бібліотеки для таких завдань, як керування базами даних, робота в мережі та безпека.

Висока продуктивність. Продуктивність Java забезпечує баланс між швидкістю мов низького рівня, як-от C++, і зручністю мов вищого рівня, як-от Python. Компілятор Just-In-Time (JIT) і оптимізація часу виконання дозволяють додаткам Java працювати ефективно, що робить його придатним для додатків, які вимагають від середньої до високої продуктивності.

Масштабованість. Архітектура та об’єктно-орієнтований дизайн Java роблять її за своєю суттю масштабованою. Такі функції, як багатопотоковість і надійне керування пам’яттю, підтримують розробку високопродуктивних програм, здатних працювати при великому навантаженні.

Сильна підтримка спільноти. Java має одну з найбільших і найактивніших спільнот розробників у світі. Ця обширна система підтримки надає доступ до великої кількості посібників, форумів і проектів з відкритим кодом. Розробники можуть знайти рішення майже для будь-якої проблеми, скорочуючи час на усунення помилок і заохочуючи інновації.

Тепер зазначимо основні недоліки Java.

Залежність від JVM. Хоча JVM забезпечує незалежність від платформи, вона також вводить рівень абстракції, який може вплинути на продуктивність. Власні програми, написані спеціально для однієї платформи, часто перевершують програми Java за швидкодією, оскільки

вони виконуються безпосередньо в операційній. Також необхідність встановлення JVM перед запуском додатків дещо ускладнює процес розгортання ПЗ.

Більший розмір програми. Програми Java часто вимагають додаткових компонентів, таких як середовище виконання JVM і допоміжні бібліотеки. Ці залежності збільшують загальний розмір програми, роблячи рішення на основі Java важчими порівняно з альтернативами, що створені під конкретні платформи.

Складний внутрішній механізм управління пам'яттю. Збірка сміття Java спрощує керування пам’яттю для розробника шляхом автоматичного відновлення невикористаної пам’яті. Однак цей процес може спричинити затримку та непередбачувані паузи, що може бути проблематичним для програм реального часу, наприклад, програм для біржової торгівлі, де постійна продуктивність є важливою.

Обмеження інтерфейсу користувача. Незважаючи на те, що Java надає такі фреймворки інтерфейсу користувача, як Swing і JavaFX, ці інструменти менш сучасні та універсальні порівняно з такими альтернативами, як Flutter, Electron або React Native. Створення візуально привабливих і чуйних інтерфейсів на Java часто вимагає більших зусиль і досвіду у порівнянні з більш сучасними інструментами.

Повільніше впровадження нових функцій. Інколи зосередження Java на стабільності та зворотній сумісності обходиться ціною інновацій. Інтеграція нових парадигм програмування або вдосконалень, пов’язаних із платформою, в екосистему Java може зайняти більше часу, ніж у більш гнучкі мови.

Лекція 2. Основи Java. Класи у Java

Послідовність вивчення матеріалу у цьому курсі є дещо нестандартною. Зазвичай у книгах, присвячених Java спочатку викладаються всі основні конструкції мови програмування, а лише потім переходять до програмування графічного інтерфейсу користувача (graphic user interface - GUI). Студенти вже повинні вміти створювати консольні програми та гарно володіти мовою програмування C++, що є частково схожою на Java. Отже ці факти дозволяють нам організувати курс дещо іншим чином, щоб уникнути ситуації коли студенти на лабораторних заняттях протягом першої половини семестру будуть знову створювати консольні застосунки, але вже на іншій мові програмування. Структура цих лекцій побудована таким чином, щоб студенти могли якнайшвидше перейти до створення програм з GUI. Якнайшвидший перехід до створення нового типу програм дозволятиме підвищити мотивацію вивчення курсу і таким чином підняти ефективність навчання.

Перейдемо до основ мови програмування Java. Ми не будемо розглядати деякі особливості мови Java, які не відрізняються від C++. Такі елементи мови можуть бути зрозумілі з прикладів без додаткових пояснень.

Отже почнемо з типів даних у Java. У Java існують два основні групи типів даних, які відрізняються за способом зберігання та роботи з ними: **value-based** типи (типи на основі значень або примітивні типи) та **reference-based** типи (типи на основі посилань).

До типів на основі посилань відносяться:

- Цілі числа: byte, short, int, long
- Числа з плаваючою точкою: float, double
- Символи: char
- Логічний тип: boolean

При роботі з такими типами ми працюємо безпосередньо зі значенням, і кожна змінна такого типу зберігає окреме значення у своїй власній комірці пам’яті.

Характеристики примітивних типів:

- Зберігають значення напряму, тобто кожна змінна зберігає своє власне значення.
- Використовують стек пам’яті для зберігання змінних.
- Копіювання змінної створює копію значення, а не посилання на те ж саме значення.
- Під час передачі в методи примітивні типи копіюються за значенням, тобто зміни всередині методу не впливають на оригінальні значення.

Приклад:

```
int a = 10;
int b = a; // b отримує копію значення a
b = 20;
System.out.println(a); // Виведе 10, оскільки a не змінилося
```

Reference-based типи або посилальні типи — це всі об’єктні типи у Java, такі як класи, масиви та інтерфейси. На відміну від примітивних типів, змінні таких типів зберігають не саме значення, а посилання на об’єкт у пам’яті (зазвичай у кучі).

Основні посилальні типи в Java:

- Класи (наприклад, String, Integer, Object, користувацькі класи)
- Масиви (наприклад, int[], String[])
- Інтерфейси

Характеристики reference-based типів:

- Змінні зберігають посилання на об’єкт у кучі, а не саме значення.
- Копіювання змінної не створює новий об’єкт, а копіює посилання на той самий об’єкт.
- При передачі посилальних типів у методи передається посилання на об’єкт. Зміни, зроблені з об’єктом всередині методу, впливають на оригінальний об’єкт.

Приклад

```
class Car {
    String model;
}

Car car1 = new Car();
car1.model = "Tesla";

Car car2 = car1; // car2 отримує посилання на той же об'єкт, що і car1
car2.model = "BMW";

// Виведе "BMW", оскільки car1 і car2 посилаються на той же об'єкт
System.out.println(car1.model);
```

Тепер перейдемо до вивчення особливостей створення класів у Java. Клас у Java — це шаблон для створення об'єктів. Він визначає:

- Поля (атрибути або властивості): відображають стан об'єктів.
- Методи: визначають поведінку об'єктів.
- Конструктори: Ініціалізують об'єкти під час їх створення.

Приклад об'явлення класу:

```
package com.example.vehicles


public class Car {
    String model;
    int year;

    Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    void start() {
        System.out.println(model + " is starting");
    }
}
```

Завдання: самостійно визначите, які саме елементи класу з прикладу є полями, методами чи конструкторами. Для цього використовуйте вже наявні знання мови C++.

Відзначимо, що класи у Java групуються у пакети. Кожний пакет повинний міститися в окремій папці у файловій системі. Пакети використовуються, щоб уникнути конфліктів імен і написати код, який краще підтримувати. Кожний клас обов'язково повинний входити в якійсь пакет. Ім'я пакету зазначається у першому рядку файлу з класом. У нашому прикладі ім'я пакету це com.example.vehicles. Повне ім'я класу Car є com.example.vehicles.Car. Якщо необхідно використати клас Car в іншому пакеті, то можна використовувати його повне ім'я, але зазвичай так не роблять. Замість цього використовують ключове слово import. Всю інструкції import містяться після назви пакету. Наприклад, після використання інструкції import com.example.vehicles.Car, у відповідному файлі можна буде використовувати коротка ім'я Car.

Синтаксис створення об'єкту класу вже був наведений у попередньому прикладі. Зверніть увагу, що у Java об'єкти класів можна створювати лише за допомогою оператору new. Викликати delete для звільнення пам'яті не потрібно (і не можливо, бо у Java нема такого оператору), бо Java автоматично визначає, коли об'єкт перестає використовуватися за допомогою алгоритмів прибирання сміття (garbage collection), і після цього звільняє пам'ять, що використовувалась об'єктом.

Також важливо відзначити, що Ім’я файлу Java має збігатися з назвою класу. Це стосується лише тих класів, що можуть бути використані в інших частинах програми. Такі класи перед об’явленням мають ключове слово public.

Члени класу можуть бути звичайними або статичними. Статичні члени класу в Java - це змінні та методи, які належать не конкретному об’єкту, а самому класу. Це означає, що вони спільні для всіх об’єктів цього класу і можуть використовуватися без створення екземпляра класу. Ключове слово static використовується для позначення таких членів.

Статичні поля існують на рівні класу і спільні для всіх об’єктів класу. Вони використовуються для збереження даних, які мають бути однаковими для всіх екземплярів класу. Ініціалізація статичних полів відбувається під час завантаження класу в пам’ять, і вони існують протягом усього часу існування програми.

Статичні методи можна викликати без створення екземпляра класу. Вони можуть працювати лише з іншими статичними полями або методами класу, оскільки для нестатичних полів чи методів потрібен конкретний екземпляр класу.

Приклад використання статичних членів класу:

```
class MathUtils {
    static double pi = 3.14;
    static int square(int x) {
        return x * x;
    }
}

// Використання
public class Main {
    public static void main(String[] args) {
        System.out.println(MathUtils.square(5)); // Виведе: 25
    }
}
```

Також у цьому прикладі ми бачимо першу повноцінну програму на Java, що може бути виконана. Кожна програма обов’язково повинна мати статичний метод main в одному із її public класів. Цей метод є аналогом функції main з C++.

Ще однією відмінністю від C++ є способи ініціалізації членів класу. У Java їх більше ніж у C++. До способів ініціалізації членів класу належать:

- ініціалізація при оголошенні;
- конструктори;
- ініціалізатор екземпляра;
- статичні блоки ініціалізації.

Розглянемо ці способи на прикладі:

```
class Car {
    String model;
    int year = 2024; // ініціалізація при оголошенні
    static int  defaultYear;

    // ініціалізатор екземпляра
    {
        model = "Default Model";
    }

    // статичний блок ініціалізації
    static {
        defaultYear = 2024;
    }

    // конструктор
    Car(String model, int year) {
        this.model = model;
        this.year = year;
    }
}
```

Також розглянемо модифікатори доступу у Java. Вони дещо схожі з C++, але мають свою відмінності. Модифікатори доступу повинні вказуватись перед кожним членом класу. Модифікатори доступу бувають наступні:

- **public (публічний).** Елемент (клас, метод або змінна) з модифікатором public доступний з будь-якого місця в програмі, тобто з будь-якого класу чи пакета.
- **private (приватний).** Елемент з модифікатором private доступний тільки в межах класу, в якому він оголошений. Клас не може бути оголошений як private. Це можливо тільки для внутрішніх (вкладених) класів.
- **protected (захищений).** Елемент з модифікатором protected доступний: в межах одного пакета, у підкласах, навіть якщо вони знаходяться в іншому пакеті.
- **default (без модифікатора).** Якщо модифікатор не вказано, то доступ до елемента вважається default, або “пакетним” (іноді називають package-private). Елемент з default доступний тільки в межах того ж пакета.

Наступною важливою темою є успадкування. **Успадкування** (англ. Inheritance) — це один із ключових принципів об’єктно-орієнтованого програмування (ООП), який дозволяє одному класу успадковувати властивості та поведінку іншого класу. Це означає, що новий клас (нащадок або підклас) може використовувати всі змінні та методи існуючого класу

(батьківського або суперкласу), а також додавати або змінювати свою поведінку.

Основні поняття успадкування в Java:

- Батьківський клас (суперклас) — це існуючий клас, який передає свої змінні та методи підкласам. Батьківський клас може бути лише один, а також він повинний дозволяти успадкування.
- Підклас (клас-нащадок) — це новий клас, який успадковує змінні та методи батьківського класу.
- Перевизначення методів (overriding): підклас може перевизначити метод батьківського класу, змінюючи його поведінку. Для цього використовується ключове слово override.

Наведемо приклад:

```
class Vehicle {
    Vehicle() {
        System.out.println("Vehicle created");
    }
    void honk() {
        System.out.println("Beep beep!");
    }
}

class Car extends Vehicle {
    Car() {
        super(); // Викликає конструктор батьківського класу
        System.out.println("Car created");
    }
    @Override
    void honk() {
        System.out.println("Car beep!");
    }
}
public class Main {
    public static void main(String[] args) {
        Vehicle myVehicle = new Car();
        myVehicle.honk(); // Виведе: Car beep!
    }
}
```

Звернемо увагу на виклик конструктору батьківського класу через ключове слово super: Ключове слово super використовується для виклику конструктора або методів батьківського класу з підкласу.

Клас може містити абстрактні методи, у такому разі і сам клас буде абстрактним. Абстрактні методи, це метод, який додається у базовий клас, але не може мати реалізацію у ньому. Виходить, що дочірні класи відповідають за реалізацію абстрактних методів. Абстрактні класи потрібні для того, щоб забезпечити базову поведінку або структуру для підкласів, але дозволити кожному підкласу реалізувати власну специфічну поведінку.

Особливості абстрактних класів:

- Неможливо створити об’єкт абстрактного класу. Абстрактний клас може бути лише успадкованим іншими класами.
- Абстрактний клас може містити як абстрактні методи (без реалізації), так і звичайні методи з реалізацією.
- Ключове слово abstract використовується для оголошення абстрактного класу і абстрактних методів.
- Підклас, що успадковує абстрактний клас, повинен реалізувати всі абстрактні методи цього класу, якщо сам не є абстрактним.
- Абстрактний клас може містити поля та конструктори.
- Абстрактний клас може використовувати успадкування.

Приклад:

```
abstract class Animal {
    // Абстрактний метод (без реалізації)
    abstract void makeSound();

    // Звичайний метод
    void sleep() {
        System.out.println("Sleeping...");
    }
}
class Dog extends Animal {
    // Реалізація абстрактного методу
    @Override
    void makeSound() {
        System.out.println("Bark");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        dog.makeSound(); // Виведе "Bark"
        dog.sleep(); // Виведе "Sleeping..."
    }
}
```

Окрім абстрактних класів Java містить інтерфейси. Інтерфейс — це колекція абстрактних методів, які клас повинен реалізувати. Інтерфейси використовуються для того, щоб забезпечити контракт для класів, які цей інтерфейс реалізовують. Всі методи інтерфейсу за замовчуванням є абстрактними і публічними, але починаючи з Java 8, інтерфейси можуть містити статичні методи та методи з реалізацією за замовченням.

Особливості інтерфейсів:

- Клас, що реалізує інтерфейс, повинен реалізувати всі його методи (якщо він не абстрактний).
- Інтерфейси можуть містити:
    - Абстрактні методи (без реалізації).
    - Методи з реалізацією за замовченням методи.

    - Статичні методи.
    - Приватні методи (для внутрішнього використання) — з’явилися в Java 9.
- Множинна реалізація: клас може реалізовувати декілька інтерфейсів (на відміну від класів, де можливе лише одне успадкування).
- Інтерфейси використовуються для поліморфізму та для визначення того, які методи повинні бути реалізовані класом.
-

Приклад використання інтерфейсів:

```
interface Animal {
    // Абстрактний метод
    void makeSound();

    // Дефолтний метод
    default void sleep() {
        System.out.println("Sleeping...");
    }
}

class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        dog.makeSound(); // Виведе "Bark"
        dog.sleep(); // Виведе "Sleeping..."
    }
}
```

Також у Java існують фінальні класи та методи. Фінальний клас не може бути успадкований, а фінальний метод не може бути перевизначений.

Наведемо приклад:

```
final class Engine {
    // Жоден клас не може розширювати цей клас Engine
}

class Car {
    // метод startEngine не може бути перевизначений
    final void startEngine() {
        System.out.println("Engine started");
    }
}
```

Відмінності між абстрактними класами та інтерфейсами наведені у таблиці:

Характеристика	Абстрактний клас	Інтерфейс
Можливість реалізації методів	Може мати як абстрактні, так і звичайні методи	Може мати абстрактні методи, статичні методи та методи за замовченням
Множинне успадкування	Не підтримує множинне успадкування	Клас може реалізовувати кілька інтерфейсів
Поля	Може мати поля та конструктори	Може містити лише статичні та константні поля
Ключове слово	abstract	interface
Ключове слово для реалізації / наслідування	extends	Implements
Використання	Використовується для ієрархії класів, спільної поведінки	Використовується для визначення контрактів

Лекція 3. Основи бібліотеки Swing

Бібліотека Swing – це бібліотека для створення графічного інтерфейсу користувача на мові Java. Swing була побудована на основі бібліотеки AWT, яка існувала раніше. Отже спочатку нам потрібно розглянути основи бібліотеки AWT (Abstract Window Toolkit).

Як бібліотека Swing так і бібліотека AWT входить у JDK (Java Development Kit) і це означає, що ці бібліотеки нам не потрібно встановлювати окремо.

AWT — це перша бібліотека GUI (графічний інтерфейс користувача), що була представлена у Java. Вона надає основу для створення інтерфейсів користувача (UI) для програм Java і включає набір класів для створення вікон, кнопок, текстових полів та інших стандартних компонентів GUI.

AWT залежить від платформи, тобто використовує компоненти GUI операційної системи (наприклад, Windows, macOS, Linux). UI елементи можуть виглядати по-різному в різних операційних системах і дещо по-різному поводитися залежно від основної системи.

AWT не має розширених компонентів, які часто потрібні сучасним програмам GUI. Він не надає складних елементів керування, таких як таблиці, дерева

AWT надає різноманітні компоненти GUI для створення інтерфейсів, зокрема:

- Вікна:
    - Frame: головне вікно програми на основі AWT.
    - Dialog: спливаюче вікно, у якому користувач може вводити дані.
- Контейнери:
    - Panel: контейнер, який може містити інші компоненти, які використовуються для організації інтерфейсу користувача.
    - Canvas: порожня область, де можна малювати власну графіку або зображення.
- Компоненти інтерфейсу користувача:
    - Button: кнопка, яку можна натиснути.
    - TextField: однорядкове поле введення тексту.
    - Label: відображає текст, який не можна редагувати.
    - Checkbox, CheckboxGroup: для створення параметрів користувач може вибрати або скасувати вибір.
    - Choice: спадне меню для вибору одного елемента зі списку.
    - List: список елементів, з яких користувач може вибрати один або декілька.
    - Scrollbar: графічний елемент керування, який дозволяє користувачам прокручувати вміст.

Програмування, кероване подіями, є ключовою частиною AWT. Такі компоненти, як кнопки, генерують події (наприклад, клацання мишею), які обробляються слухачами або класами обробки подій.

Система подій Java AWT базується на моделі подій делегування, що означає, що об'єкт (джерело події) делегує обробку подій іншому об'єкту (случачу). Слухач (прослуховувач) подій — це класи, які реалізують інтерфейси подій то визначають що саме буде відбуватися у відповідь на подію.

Swing було представлено пізніше, щоб усунути деякі обмеження AWT. Swing побудовано на основі AWT і забезпечує багатший набір компонентів інтерфейсу користувача.

На відміну від AWT, компоненти Swing повністю написані на Java, що забезпечує узгоджений вигляд і відчуття на різних платформах. Компоненти AWT все ще можна використовувати в сучасних програмах Java, але їх часто замінюють компонентами Swing для більш складних і багатофункціональних GUI.

Зупинимося на особливостях роботи програм з графічним інтерфейсом користувача. Swing використовує окремий потік під назвою Event Dispatch Thread (EDT) для обробки всіх завдань, пов'язаних із графічним інтерфейсом користувача, включаючи реагування на дії користувача, такі як натискання клавіш.

Після створення головного вікна програми EDT продовжує працювати у фоновому режимі навіть після завершення роботи методу main(). Це тому, що програма все ще очікує на події (наприклад, закриття вікна, натискання кнопки тощо), які відбудуться в GUI.

JVM (Віртуальна машина Java) не припинить роботу програми, доки запущені будь-які потоки, що не є демонами. EDT — це недемонний потік, тобто він підтримує роботу програми навіть після завершення методу main().

Примітка: Daemon thread у Java— це тип потоку, який працює у фоновому режимі та зазвичай використовується для завдань, які підтримують програму, але не є важливими для її основних функцій.

Коли відбувається подія (наприклад, користувач натискає кнопку), відбувається наступний процес:

- Генерація події: компонент GUI генерує подію (наприклад, ActionEvent для натискання кнопки).
- Постановка події в чергу: подія розміщується в черзі подій.
- Відправка подій: EDT постійно перевіряє чергу подій і, коли виявляє нову подію, бере її з черги (event loop).
- Обробка події: подія надсилається відповідному слухачу події, який зареєстровано в джерелі події.

EDT є однопотоковим, тобто одночасно може оброблятися лише одна подія. Це гарантує, що компоненти графічного інтерфейсу оновлюються послідовно й без стану гонки (race condition).

Тепер перейдемо до розгляду основних класів бібліотеки Swing, що дозволяють створювати програми з графічним інтерфейсом користувача. Почнемо з класів JFrame та JPanel.

JFrame — це контейнер верхнього рівня, який представляє вікно з основними елементами, такими як рядок заголовка, кнопки.

JPanel — це загальний легкий контейнер для організації компонентів інтерфейсу користувача.

JFrame використовує JPanel як панель вмісту для зберігання компонентів. JFrame Може містити декілька панелей JPanel.

Наведемо приклад використання класів JFrame та JPanel:

```
import javax.swing.*;
import java.awt.*;

public class FrameExample {
    public static void main(String[] args) {
        // Створення вікна
        JFrame frame = new JFrame("Green JPanel Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);  // Встановити розмір вікна
        // Створення JPanel
        JPanel panel = new JPanel();
        // Встановити фоновий колір JPanel на зелений
        panel.setBackground(Color.GREEN);
        // Додати панель до вікна
        frame.add(panel);
        // Зробити вікно видимим
        frame.setVisible(true);
    }
}
```

Наступним розглянемо один з найпростіших елементів графічного інтерфейсу користувача — мітку. JLabel — це компонент, який використовується для відображення текстового рядка або зображення. Мітки, як правило, є неінтерактивними компонентами, тобто користувачі не можуть натискати або вводити їх. Мітки часто використовуються для опису інших компонентів (наприклад, полів введення чи кнопок) або для відображення статичної інформації.

Основні методи класу JLabel :

- setText(String text): встановити текст для мітки.
- setIcon(Icon icon): встановити зображення для мітки.
- setHorizontalAlignment(int alignment): встановити вирівнювання тексту.
- setForeground(Color color): встановити колір тексту.

JButton — це компонент, який користувачі можуть натиснути, щоб виконати дію. Це ключовий інтерактивний елемент у Swing. Кнопки можуть

відображати як текст, так і піктограми, і вони зазвичай пов’язуються з слухачами подій (наприклад, ActionListener), щоб визначити дії, які відбуваються під час натискання кнопки.

Основні властивості та методи:

- setText(String text): встановити текст для кнопки.
- setIcon(Icon icon): встановити зображення для кнопки.
- addActionListener(ActionListener listener): зареєструвати слухача подій для відповіді на натискання кнопки.

Більш детально розглянемо, яким чином відбувається обробка подій у бібліотеці Swing. Спочатку ми повинні створити об’єкт слухача подій. Слухачі подій є невід'ємною частиною керованої подіями моделі програмування, яка використовується для обробки взаємодії в графічних інтерфейсах користувача.

Компоненти, які можуть генерувати події, відомі як джерела подій (кнопки, вікна, текстові поля). Коли користувач взаємодіє з цими компонентами, генерується подія.

Кожна подія в Java представлена екземпляром об’єкта події, який містить деталі про подію.

Слухачі подій — це об’єкти, які «слухають» певний тип події та визначають, як реагувати, коли ця подія відбувається. Слухачі повинні реалізувати спеціальні інтерфейси із пакетів java.awt.event або javax.swing.event.

Щоб обробити подію, слухач події реєструється в джерелі події. Зазвичай це робиться за допомогою таких методів, як addActionListener(), addMouseListener() тощо.

Розглянемо, як це працює на прикладі:

```
import javax.swing.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

class MyActionListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        // Код, що виконується, коли натиснуто кнопку
        System.out.println("Button clicked!");
    }
}
public class EventListenerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Event Listener Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JButton button = new JButton("Click Me");
        // Додати слухача подій для кнопки
        button.addActionListener(new MyActionListener());
        frame.add(button);
        frame.setVisible(true);
    }
}
```

Відзначимо, що клас MyActionListener потрібний лише для створення одного об’єкту. У таких випадках можна використовувати так звані анонімні внутрішні класи. Перепишемо наш приклад з використанням анонімних класів.

```
import javax.swing.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

class MyActionListener implements ActionListener {}
public class EventListenerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Event Listener Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JButton button = new JButton("Click Me");
        // Додати слухача подій для кнопки
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // Код, що виконується, коли натиснуто кнопку
                System.out.println("Button clicked!");
            }
        });
        frame.add(button);
        frame.setVisible(true);
    }
}
```

Рядок, у якому додається слухач подій, виглядає ніби то у ньому створюється об’єкт інтерфейсу ActionListener. Але ми пам’ятаємо, що створювати об’єкти інтерфейсів не можна. Насправді тут створюється анонімний внутрішній клас, який реалізує інтерфейс ActionListener. Він анонімний, бо його ім’я насправді не важливо. Все що нам потрібно — це створити один єдиній об’єкт цього класу і саме це і відбувається. Цей клас є внутрішнім, бо він створюється всередині іншого класу, а саме EventListenerExample. Внутрішні класи мають доступ до полів та методів зовнішніх класів.

Окрім слухачів подій існують також адаптери подій. У Java адаптери (наприклад, ActionAdapter, MouseAdapter, KeyAdapter тощо) використовуються для спрощення обробки подій для інтерфейсів слухачів, які мають кілька методів. Ці класи адаптерів забезпечують порожні реалізації для всіх методів відповідного інтерфейсу слухача. Ви можете перевизначати лише ті методи, які вас цікавлять, без необхідності реалізації всіх методів в інтерфейсі.

Наведемо приклад обробки події натискання кнопки миші без використання адаптерів:

```
import javax.swing.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class WithAdapterExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Mouse Adapter Example");
        JButton button = new JButton("Click Me");
        // Використання MouseListener
        button.addMouseListener(new MouseListener() {
            public void mouseClicked(MouseEvent e) {
                System.out.println("Mouse Clicked");
            }
            public void mousePressed(MouseEvent e) {}
            public void mouseReleased(MouseEvent e) {}
            public void mouseEntered(MouseEvent e) {}
            public void mouseExited(MouseEvent e) {}
        });

        frame.add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Ми бачимо, що розробник хотів додати логіку лише до одного методу, що визначений у інтерфейсі MouseListener, але довелося додати пусті реалізації також для всіх інших методів інтерфейсу.

Отже використання адаптерів може значно спростити код:

```
import javax.swing.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class WithAdapterExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Mouse Adapter Example");
        JButton button = new JButton("Click Me");
        // Використання MouseAdapter
        button.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                System.out.println("Mouse Clicked");
            }
        });

        frame.add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Тепер розглянемо, яким чином можна задати правила розміщення елементів графічного інтерфейсу. Менеджери макетів автоматично впорядковують компоненти на основі попередньо визначених правил, спрощуючи дизайн GUI. Розробнику не потрібно вручну вказувати точні координати x і y для кожного компонента.

Менеджери макетів обробляють зміну розміру та впорядкування компонентів, коли розмір вікна змінюється.

Різні платформи можуть мати різні розміри шрифтів, роздільну здатність екрана або системи керування вікнами. Менеджери макетів забезпечують узгодженість інтерфейсу користувача на різних платформах, відповідно налаштовуючи компоненти.

Основні менеджери макетів: FlowLayout, BorderLayout, GridLayout, BoxLayout, CardLayout.

FlowLayout:

- Компоненти розташовані горизонтально, один за одним (як слова в реченні), і переходять на наступний рядок, коли закінчується місце.
- Добре підходить для простих макетів з невеликою кількістю компонентів.

BorderLayout:

- Розділяє контейнер на п’ять областей: Північ, Південь, Схід, Захід і Центр.
- Компоненти, додані до цих регіонів, змінюють розмір так, щоб відповідати розміру регіону, центр займає простір, що залишився.

GridLayout:

- Розміщує компоненти в сітці з рядками та стовпцями.
- Розміри всіх компонентів сітки змінюються, щоб рівномірно відповідати клітинкам.

BoxLayout:

- Розташовує компоненти вертикально або горизонтально, зберігаючи бажаний розмір.
- Корисно використовувати для укладання компонентів в один рядок або стовпець.

CardLayout:

- Дозволяє складати кілька панелей (наприклад, карток) одна на одну, де одночасно видно лише одну панель.
- Корисно використовувати для реалізації інтерфейсів, подібних до майстра (wizard), де користувач переміщується між різними панелями.

Наведемо декілька прикладів використання менеджерів компоновки. Почнемо з FlowLayout:

```
import javax.swing.*;
import java.awt.*;

public class LayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Layout Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 200);
```

```
        // Використання FlowLayout
        JPanel panel = new JPanel(new FlowLayout());

        panel.add(new JButton("Button 1"));
        panel.add(new JButton("Button 2"));
        panel.add(new JButton("Button 3"));

        frame.add(panel);
        frame.setVisible(true);
    }
}
```

Даля наведемо приклад використання BorderLayout.

```
import javax.swing.*;
import java.awt.*;

public class BorderLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("BorderLayout Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300); // Set the size of the frame
        // Використовуємо BorderLayout
        frame.setLayout(new BorderLayout());
        JButton northButton = new JButton("North");
        JButton southButton = new JButton("South");
        JButton eastButton = new JButton("East");
        JButton westButton = new JButton("West");
        JButton centerButton = new JButton("Center");

        frame.add(northButton, BorderLayout.NORTH);
        frame.add(southButton, BorderLayout.SOUTH);
        frame.add(eastButton, BorderLayout.EAST);
        frame.add(westButton, BorderLayout.WEST);
        frame.add(centerButton, BorderLayout.CENTER);

        frame.setVisible(true);
    }
}
```

Розглянемо ще декілька цікавих компонентів. Елемент вибору варіантів — це компонент, який дозволяє користувачам зробити один вибір із групи параметрів. Перемикачі представлені класом JRadioButton, і вони використовуються, коли потрібно обрати один параметр із попередньо визначеного набору.

Перемикачі працюють у групах, тобто коли вибрано один перемикач у групі, вибір з усіх інших у групі автоматично знімається. Групування відбувається за допомогою класу ButtonGroup.

Основні методи класу JRadioButton:

- isSelected(): повертає true, якщо вибрано перемикач, і false в іншому випадку.
- setSelected(boolean): вибирає перемикач програмно.
- addActionListener(ActionListener listener): зареєструвати слухач подій для відповіді на натискання кнопки.

Приклад використання елементів вибору:

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class RadioButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JRadioButton Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 200);
        // Створити елементи вибору
        JRadioButton option1 = new JRadioButton("Option 1");
        JRadioButton option2 = new JRadioButton("Option 2");
        JRadioButton option3 = new JRadioButton("Option 3");

        // Створити ButtonGroup для групування елементів вибору
        ButtonGroup group = new ButtonGroup();
        group.add(option1);
        group.add(option2);
        group.add(option3);

        // Створити мітку, що відображатиме обрану опцію
        JLabel label = new JLabel("Select an option:");

        // Додати слухачі подій для елементів
        option1.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                label.setText("You selected Option 1");
            }
        });

        option2.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                label.setText("You selected Option 2");
            }
        });

        option3.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                label.setText("You selected Option 3");
            }
        });

        JPanel panel = new JPanel();
        panel.add(option1);
        panel.add(option2);
        panel.add(option3);
        panel.add(label);

        frame.add(panel);

        frame.setVisible(true);
    }
}
```

Текстове поле (JTextField) — це компонент, який може відображати один рядок тексту, що можна редагувати. Є можливість встановити початковий текст, отримати введені користувачем дані та реагувати на дії користувача (наприклад, натискання Enter).

Основні методи класу JTextField:

- setText(String text): встановити початковий текст у текстовому полі.
- getText(): повертає поточний текст із текстового поля.
- setHorizontalAlignment(int alignment): встановити горизонтальне вирівнювання тексту.
- addActionListener(ActionListener listener): реагує на натискання клавіші Enter, коли користувач вводить текст у текстовому полі.
- Document getDocument(): повертає модель, що асоційована з текстовим полем

Приклад використання текстового поля:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
 public class JTextFieldButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JTextField and JButton Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 200);
        frame.setLayout(new FlowLayout());

        // довжина - 20 символів
        JTextField textField = new JTextField(20);
        JButton button = new JButton("Click Me");
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                // Отримати текст з поля вводу та надрукувати його
                String inputText = textField.getText();
                System.out.println("Text entered: " + inputText);
            }
        });
        frame.add(textField);
        frame.add(button);
        frame.setVisible(true);
    }
}
```

Більш складний приклад використання поля вводу:

```
import javax.swing.*;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;
import java.awt.*;

public class JTextFieldTextChangeExample {
```

```
    public static void main(String[] args) {
        JFrame frame = new JFrame("Text Change Event Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 200);
        frame.setLayout(new FlowLayout());
        JTextField textField = new JTextField(20);
        JLabel label = new JLabel("Start typing...");
        // Встановити DocumentListener для реагування на зміни тексту
        textField.getDocument().addDocumentListener(new DocumentListener() {
            @Override
            public void insertUpdate(DocumentEvent e) {
                // Цей код викликається при вставці тексту
                label.setText("Text: " + textField.getText());
            }
            @Override
            public void removeUpdate(DocumentEvent e) {
                // Цей код викликається при видаленні тексту
                label.setText("Text: " + textField.getText());
            }
             @Override
            public void changedUpdate(DocumentEvent e) {
            }
        });

        frame.add(textField);
        frame.add(label);

        frame.setVisible(true);
    }
}
```

Список, що розкриваються — це компонент, який надає користувачеві розкривний список елементів на вибір. Він широко використовується в додатках, щоб дозволити користувачам вибрати один елемент зі списку або за бажанням ввести спеціальне значення залежно від конфігурації.

JComboBox відображає список елементів, які можна розгорнути, натиснувши стрілку вниз. Користувачі можуть вибрати один пункт зі списку.

За замовчуванням JComboBox не можна редагувати (користувач може лише вибирати елементи зі списку). Однак можна зробити цей список редагованим, дозволяючи користувачам вводити власні дані на додаток до вибору з доступних параметрів.

JComboBox підтримується моделлю даних (ComboBoxModel), яка дозволяє маніпулювати елементами в списку.

Список, що розкриваються генерує події, коли користувач взаємодіє з ним (наприклад, вибирає елемент), що дозволяє розробникам реагувати на ввід користувача.

Також можна налаштувати спосіб відображення кожного елемента, встановивши спеціальний рендерер. Ця функція має назву — спеціальна візуалізація.

Основні операції:

- Конструювання:
  String[] items = { "Item 1", "Item 2", "Item 3" };
  JComboBox comboBox = new JComboBox(items);
- Додавання елементів:
  comboBox.addItem("Item 4");
- Додавання великої кількості елементів: потрібно створити DefaultComboBoxModel, заповнити її за допомогою addElement, а потім виконайте виклик методу setModel класу JComboBox.
- Видалення елементів:
  comboBox.removeItem("Item 2");
  comboBox.removeItemAt(1);
- Отримання індексу обраного елементу:
  int selectedIndex = comboBox.getSelectedIndex();

Лекція 4. Ієрархія класів-подій. Малювання. Обробка виключних ситуацій

І в Swing, і в AWT (Abstract Window Toolkit) ієрархія подій відіграє ключову роль у обробці взаємодії між користувачем і компонентами GUI.

Кореневий клас для всіх подій — це java.util.EventObject – це батьківський клас для усіх класів подій у Java. Він зберігає такі дані, як джерело події.

AWTEvent є підкласом EventObject. Він є базовий клас для всіх подій AWT.

Загальні події AWT:

- ActionEvent: генерується під час натискання кнопки, вибору пункту меню або активації пункту.
- AdjustmentEvent: ініціюється змінами в регульованих елементах, таких як смуги прокручування.
- ComponentEvent: виникає, коли компонент приховано, переміщено, змінено розмір або показано. Підкласи: ContainerEvent, FocusEvent, WindowEvent тощо.
- FocusEvent: запускається, коли компонент отримує або втрачає фокус клавіатури.
- InputEvent: батьківський клас для подій клавіатури та миші.
- KeyEvent: генерується під час натискання, відпускання або введення клавіші.
- MouseEvent: генерується, коли кнопку миші натискають, відпускають, клацають або переміщують над компонентом.
- WindowEvent: генерується, коли вікно відкривається, закривається, активується, деактивується або змінюється розмір.

Базові класи для подій Swing такі самі як і для подій AWT. Основні події з бібліотеки Swing:

- AncestorEvent: пов’язана зі змінами в предках компонента (наприклад, коли компонент додається до контейнера або видаляється з нього).
- CaretEvent: запускається, коли позиція каретки (текстового курсору) змінюється в текстовому компоненті.
- ChangeEvent: використовується в таких моделях, як BoundedRangeModel (для повзунків, індикаторів прогресу).
- DocumentEvent: ініціюється змінами у вмісті текстового документа.
- ListDataEvent: запускається, коли вміст списку змінюється.

- ListSelectionEvent: запускається, коли вибір у списку або таблиці змінюється.
- MenuDragMouseEvent, MenuKeyEvent, PopupMenuEvent: спеціальні події для взаємодії з меню.
- TableModelEvent: запускається, коли вміст таблиці змінюється.

Повна документація по подіям знаходиться за посиланням: https://docs.oracle.com/en/java/javase/17/docs/api/java.desktop/javax/swing/event/package-tree.html.

Події поділяються на семантичні та низкорівневі. Події низького рівня генеруються безпосередньо операційною системою або середовищем виконання Java, коли користувач взаємодіє з апаратними пристроями, такими як миша або клавіатура. Приклади: MouseEvent, KeyEvent, WindowEvent, FocusEvent.

Семантичні події — це події вищого рівня, які представляють значущі дії користувача або взаємодію з компонентами. Вони абстрагуються від конкретного апаратного введення та зосереджуються на тому, чого намагається досягти користувач, а не на тому, як відбулася взаємодія. Іншими словами, семантичні події відбуваються, коли система виявляє, що користувач виконав дію, яка має певне значення, наприклад вибір пункту. Приклади: ActionEvent, ItemEvent, AdjustmentEvent, TextEvent.

UML діаграма для основних подій наведена на рисунку 1:

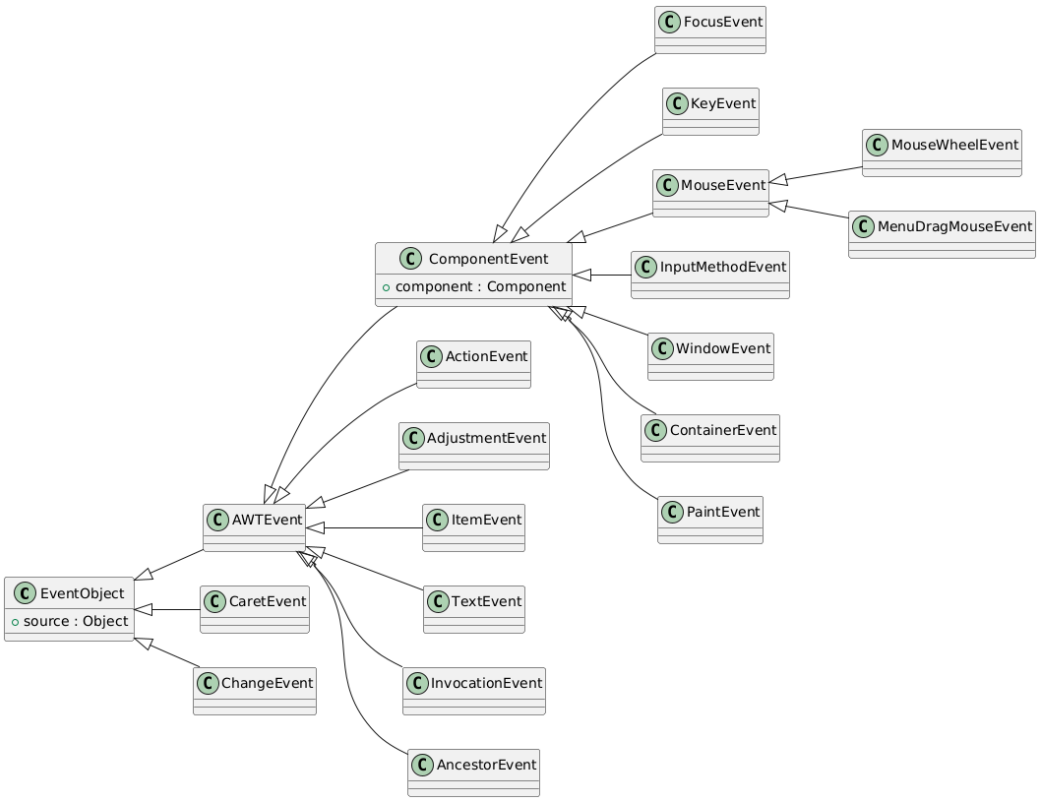

Рис 1. Діаграма класів основних подій

Далі представлений приклад обробки подій миші:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class MouseEventExample extends JFrame {
    private JLabel statusLabel;
    public MouseEventExample() {
        setTitle("MouseEvent Example");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new JPanel();
        panel.setBackground(Color.LIGHT_GRAY);
        add(panel, BorderLayout.CENTER);
        statusLabel = new JLabel("Mouse events will be displayed here");
        add(statusLabel, BorderLayout.SOUTH);
        panel.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                statusLabel.setText("Mouse Clicked at (" + e.getX() + ", " +
                                    e.getY() + ")");
            }
        });
        panel.addMouseMotionListener(new MouseMotionAdapter() {

            @Override
            public void mouseMoved(MouseEvent e) {
                statusLabel.setText("Mouse Moved to (" + e.getX() + ", " +
                                    e.getY() + ")");
            }
        });
    }

    public static void main(String[] args) {
        MouseEventExample example = new MouseEventExample();
        example.setVisible(true);
    });
}
```

У Java Swing і AWT малювання стосується відображення графіки на компоненті, наприклад фігур, тексту, зображень або інших графічних елементів. Це досягається за допомогою об’єкта Graphics, який надає методи малювання на поверхні компонента. Якщо необхідно хочете налаштувати зовнішній вигляд компонента або створити власні графічні елементи, то необхідно реалізовувати ці функції згідно правилами, які будуть наведені нижче.

Найважливішим для малювання є метод paintComponent. Цей метод є частиною класу JComponent і відповідає за малювання компонента. Коли потрібно виконати нестандартне малювання в компоненті Swing, потрібно замістити метод paintComponent і використовувати об’єкт Graphics, переданій, як параметр. Цей метод викликається автоматично щоразу, коли компонент потрібно перемалювати (наприклад, під час зміни розміру вікна або явного виклику repaint()).

Компоненти Swing за замовчуванням подвійно буферизуються, тобто вони малюються за межами екрана перед відображенням. Це допомагає уникнути мерехтіння під час зміни зображення.

Графічний контекст — це абстрактний клас, наданий пакетом AWT (java.awt.Graphics). Він діє як контекст для всіх операцій малювання та надає різні методи малювання фігур, тексту, зображень та інших компонентів. У Swing об'єкт Graphics створюється бібліотекою та передається відповідним методам.

Під час виконання спеціального малювання в Swing або AWT метод paint(Graphics g) або paintComponent(Graphics g) надає об'єкт Graphics як аргумент. Однак цей об'єкт насправді є екземпляром Graphics2D (підклас Graphics), але сигнатура методу використовує більш загальний тип Graphics для зворотної сумісності.

Клас Graphics був частиною оригінального API AWT, який має обмежену функціональність. У міру розвитку Java клас Graphics2D був представлений для надання більш просунутих і гнучких функцій двовимірної графіки, зберігаючи при цьому сумісність зі старішим API.

Клас Graphics2D пропонує додаткові методи та функції, недоступні в Graphics. Наприклад:

- Трансформації: обертання, масштабування, перенесення або зсув фігур.
- Можливість визначення стилів ліній (наприклад, пунктирні лінії).
- Можливість контролювати якість візуалізації (наприклад, згладжування).
- Альфа-компонування: застосування ефектів прозорості та змішування.
- Розширені форми: використовуйте такі об'єкти фігур, як Rectangle2D, Ellipse2D і Path2D.

Ключові методи класу Graphics2D:

- drawLine(int x1, int y1, int x2, int y2).
- drawRect(int x, int y, int width, int height).
- fillRect(int x, int y, int width, int height).
- drawOval(int x, int y, int width, int height).
- fillOval(int x, int y, int width, int height).
- drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight).
- fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight).
- drawPolygon(int[] xPoints, int[] yPoints, int nPoints).
- fillPolygon(int[] xPoints, int[] yPoints, int nPoints)
- drawString(String str, int x, int y)
- setColor(Color c)

Далі розглянемо найкращі практики, що мають застосовуватись при малюванні.

- Завжди викликайте super.paintComponent(g) на початку методу paintComponent, щоб переконатися, що фон компонента належним чином очищено.
- Використовуйте Graphics2D: якщо можливо, транслюйте об’єкт Graphics до Graphics2D для розширених функцій, таких як згладжування та трансформації.
- Уникайте інтенсивних обчислень: не виконуйте трудомісткі операції (наприклад, складні обчислення чи пошук даних) усередині paintComponent. Він має містити лише код малювання.
- Правильно запускайте перемальовування: використовуйте repaint(), щоб запускати повторне малювання, коли змінюється стан вашого компонента (наприклад, коли потрібно перемалювати фігуру).

Наведемо приклад реалізації малювання:

```
import javax.swing.*;
import java.awt.*;

public class Graphics2DExample extends JPanel {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        // Перетворюємо Graphics на Graphics2D
        Graphics2D g2d = (Graphics2D) g;

        // Увімкнення згладжування (антиаліасінг)
        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
           RenderingHints.VALUE_ANTIALIAS_ON);

        // Малюємо заповнений прямокутник із закругленими кутами
        g2d.setColor(Color.BLUE);
        g2d.fillRoundRect(50, 50, 200, 100, 20, 20);

        // Малюємо повернутий текст
        g2d.setColor(Color.RED);
        g2d.rotate(Math.toRadians(45), 150, 150);
        g2d.drawString("Hello, Graphics2D!", 100, 150);
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame("Graphics2D Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 400);
        frame.add(new Graphics2DExample());
        frame.setVisible(true);
    }
}
```

Наступною важливою темою є обробка виключних ситуацій. У Java виключною ситуацією є подія, яка порушує нормальний хід виконання програми. Виключення виникають під час виконання та можуть виникати

через різні причини, такі як неочікувані вхідні дані, збої, проблеми з мережею тощо. Java надає механізм для обробки таких помилок, використовуючи методи обробки виключних ситуацій.

Для кожного типу виключної ситуації у Java існує відповідний клас.

Усі класи для виключних ситуацій походять від класу java.lang.Throwable.

Синтаксис:

```
try {
  // Код, який може викликати виключення
} catch (ExceptionType1 e) {
  // Обробка виключної ситуації типу ExceptionType1
} catch (ExceptionType2 e) {
  // Обробка виключної ситуації типу ExceptionType2
} finally {
  // Код, який виконуватиметься незалежно від того, чи станеться виключна
  // ситуація
}
```

Також є можливість вручну створювати виняткові ситуації за допомогою ключового слова throw.

Наведемо декілька прикладів. Перший приклад стосується обробки виключних ситуацій.

```
public class ExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;  // Тут виникає ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic error: " + e.getMessage());
        } finally {
            System.out.println("This block always executes.");
        }
    }
}
```

Наступний приклад стосується створення виключних ситуацій.

```
public class ThrowExample {
    public static void main(String[] args) {
        int age = 15;
        try {
          checkAge(age);
        } catch (IllegalArgumentException e) {
          System.out.println(e.getMessage());
        }
    }

    public static void checkAge(int age) {
        if (age < 18) {
          // Тут створюється виключна ситуація типу IllegalArgumentException
          throw new IllegalArgumentException("Age must be 18 or older.");
        } else {
          System.out.println("Valid age.");
        }
    }
}
```

Java дозволяє створювати власні класи для нових типів виключних ситуацій, розширюючи клас Exception.

```
class MyCustomException extends Exception {
    public MyCustomException(String message) {
        super(message);
    }
}

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            throw new MyCustomException("This is a custom exception!");
        } catch (MyCustomException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Розглянемо ієрархію класів, що представляють виключні ситуації. Базовий клас для всіх виключень — це клас java.lang.Throwable. Він має два основних підкласи: Error та Exception.

Error вказує на серйозні проблеми, які, як правило, поза контролем програми, наприклад помилки на системному рівні або апаратні збої. Помилки зазвичай невиправні (наприклад, OutOfMemoryError, StackOverflowError).

Exception: представляє виключні ситуації, які програма може перехопити та обробити. Це суперклас для всіх виключень, які виникають під час нормального виконання програми. Такі виключення поділяються на два типи: оброблювані та необроблювані виключення.

- Оброблювані винятки: це винятки, які потрібно обробляти або за допомогою блоку try-catch, або шляхом оголошення їх у сигнатурі методу за допомогою ключового слова throws. Вони перевіряються під час компіляції (наприклад, IOException, SQLException). Суперклас: Exception.
- Необроблювані виключення: також відомі як винятки під час виконання, вони виникають під час виконання, і їх не потрібно оголошувати чи перехоплювати. Зазвичай вони є результатом логічних помилок або неправильного використання програми (наприклад, винятки NullPointerException, ArrayIndexOutOfBoundsException). Суперклас: RuntimeException, що є похідним від Exception

Діаграма класів виключних ситуацій наведена на рисунку 2:

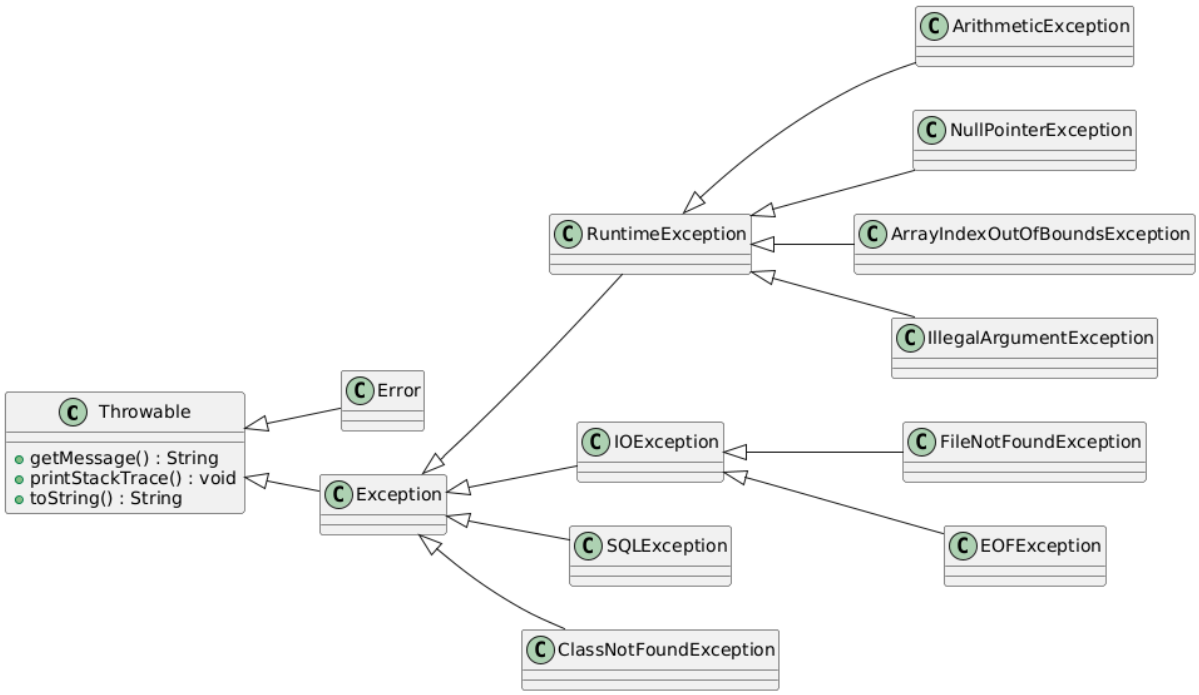

Рис 2. Діаграма класів виключних ситуацій

Найкращі практики роботи з виключними ситуаціями:

- Перехоплюйте конкретні виключення: уникайте перехоплення загальних виключень, таких як Exception. Завжди відловлюйте конкретні виключення, щоб правильно обробляти кожен випадок.
- Не ігноруйте виключення: уникайте перехоплення виключень без будь-якої логіки обробки. Принаймні залогуйте виключення, щоб допомогти з відлагодженням.
- Використовуйте блок finally для очищення: якщо у вас є такі ресурси, як файли або підключення до бази даних, які потрібно закрити, скористайтеся блоком finally, щоб переконатися, що вони закриті незалежно від того, чи станеться виключна ситуація.
- Уникайте надмірного використання оброблюваних виключень: використовуйте оброблювані виключення лише тоді, коли клієнт, може розумно впоратися з виключенням. В іншому випадку використовуйте необроблювані виключення для логічних помилок.
- Помірно використовуйте спеціальні виключення: створюйте спеціальні виключення лише тоді, коли стандартні виключення Java не відповідають вашому варіанту використання.
- Викидайте рано, виловлюйте пізно: створюйте винятки, щойно ви виявите помилку, але виловлюйте їх лише там, де ви можете їх осмислено обробити.

Лекція 5. Особливості масивів у Java. Основні колекції

Лекція присвячена тому, як у Java можна зберігати набори однотипних елементів. Як і у C++ в нас є дві можливості. Це або використовувати масиви або використовувати класи, які реалізують більш складні структури даних (так звані колекції).

Масиви у Java багато в чому схожі на масиви в C++ але мають ряд суттєвих відмінностей. Почнемо з розгляду особливостей використання масивів у Java.

Масив — це об'єкт-контейнер, який містить фіксовану кількість значень одного типу. Елементи в масиві впорядковані та доступні через їхній індекс, починаючи з 0 (індексування від нуля). Масиви в Java можуть містити або примітиви (наприклад, int, char, boolean тощо), або об'єкти (наприклад, String, спеціальні класи).

Масиви зберігаються в безперервних областях пам'яті, що робить їх швидкими для доступу до елементів за індексом Це також означає, що зміна розміру або вставлення елементів може бути дорогим.

Робота з масивами виглядає наступним чином:

1. Об'явлення:

```
int[] numbers;
String[] names;
```

2. Виділення пам'яті:

```
numbers = new int[5];
```

3. Обхід масиву:

```
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
for (int num : numbers) {
    System.out.println(num);
}
```

Масиви можна ініціалізувати при об'явленні:

```
// одномірний масив
int[] array = {1, 2, 3, 4, 5};

// багатомірний масив
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

```
// ступінчастий масив
int[][] jaggedArray = {
    {1, 2},
    {3, 4, 5},
    {6}
};
```

Відмітимо переваги та недоліки масивів.

Переваги:

- Швидкий доступ до елементів по індексу;
- Максимально ефективне використання пам’яті;
- Швидкий послідовний доступ (через оптимізацію використання кешу);

Недоліки:

- Фіксований розмір;
- Відсутність методів для розповсюджених операцій (але є пакет java.util.Arrays);
- Складності у роботі з узагальненими типами (generic types).

```
List<String>[] listArray = new List<String>[10]; // помилка компіляції
```

Оскільки масиви зберігають інформацію про свій тип виконання, а генерики ні, вони поводяться принципово несумісними способами. Масиви зберігають інформацію про свій тип під час виконання, тоді як узагальнені типи втрачають інформацію про тип через механізм «стирання». Цей конфлікт призводить до проблем при спробі створити масиви узагальнених типами.

Підсумуємо інформацію щодо масивів. Масиви Java є фундаментальною структурою даних із швидким доступом і низькими накладними витратами пам’яті, що робить їх придатними для сценаріїв, де продуктивність є критичною, а розмір даних відомий заздалегідь. Однак вони мають фіксований розмір, їм бракує гнучкості динамічних структур даних і вони не забезпечують вбудованих методів для більш складних операцій.

У розробці програмного забезпечення керування даними має надважливе значення. Java надає обширну бібліотеку під назвою Java Collections Framework, яка дозволяє працювати з групами об’єктів ефективно та з використанням єдиного підходу до колекцій різних типів. Розуміння колекцій надзвичайно важливе, оскільки вони спрощують те, як ми зберігаємо, керуємо та маніпулюємо даними. Незалежно від того, чи маєте ви справу з кількома елементами чи мільйонами, правильна колекція може мати вирішальне значення для ефективності роботи програми.

Java Collections Framework (JCF) – це бібліотека, яка є частиною JDK, що містить інтерфейси та класи, що їх реалізують, та надають можливість розробнику користуватися великою кількістю основних структур даних. JCF реалізує уніфіковану архітектуру для зберігання та керування групами об'єктів. JCF має вбудовану підтримку типових операцій, таких як пошук, сортування, вставка, видалення.

JCF заснований на 5 основних інтерфейсах: Collection, List, Set, Queue, Map. Інтерфейс Collection є батьківським для інтерфейсів List, Set, Queue.

Головні інтерфейси JCF:

- **Collection** – описує основні операції: додавання, видалення, отримання розміру.
- **List** – описує упорядковані колекції (також відомі як послідовності), може містити дублікати.
  Основні класи: ArrayList, LinkedList.
- **Set** - описує неупорядковані колекції, не може містити дублікати.
  Основні класи: HashSet, TreeSet.
- **Queue** – описує колекції типу FIFO (First-In-First-Out)
  Основні класи: LinkedList, PriorityQueue.
- **Map** - описує колекції, що зберігають пари типу ключ-значення.
  Основні класи: HashMap, TreeMap, LinkedHashMap.

Основні операції інтерфейсу Collection:

- **boolean add(E e);**
- **boolean addAll(Collection<? extends E> c)**
- **void clear()**
- **boolean contains(Object o)**
- **boolean containsAll(Collection<?> c)**
- **boolean equals(Object o)**
- **int size()**
- **boolean isEmpty()**
- **boolean removeAll(Collection<?> c)**
- **Iterator<E> iterator()**

Основні операції інтерфейсу List:

- **E get(int index)**
- **E set(int index, E element)**
- **void add(int index, E element)**
- **boolean addAll(int index, Collection<? extends E> c)**
- **E remove(int index)**
- **int indexOf(Object o)**
- **ListIterator<E> listIterator()**
- **ListIterator<E> listIterator(int index)**
- **List<E> subList(int fromIndex, int toIndex)**

Відмінності listIterator від iterator: iterator() є простішим і дозволяє обхід лише вперед, тоді як listIterator() є більш універсальним і підтримує як прямий, так і зворотний обхід, а також модифікації та доступ до індексів.

Основні операції інтерфейсу Queue:

- **T element()**
- **boolean add(T e)**
- **boolean offer(T e)** — вставляє елемент, якщо це можливо, інакше повертає false. Це відрізняється від методу add, який може не додати елемент лише шляхом викидання необроблюваного виключення.
- **peek()** — повертає, але не видаляє елемент з голови черги, або повертає null, якщо черга порожня.
- **poll()** — повертає та видаляє елемент з голови черги, або повертає null, якщо черга порожня.
- **remove()** — повертає та видаляє елемент з голови черги, відрізняється від **poll** лише тим, що він створює виключення, якщо ця черга порожня

Інтерфейси List та Queue та основні класи, що їх реалізують представлені на наступній діаграмі:

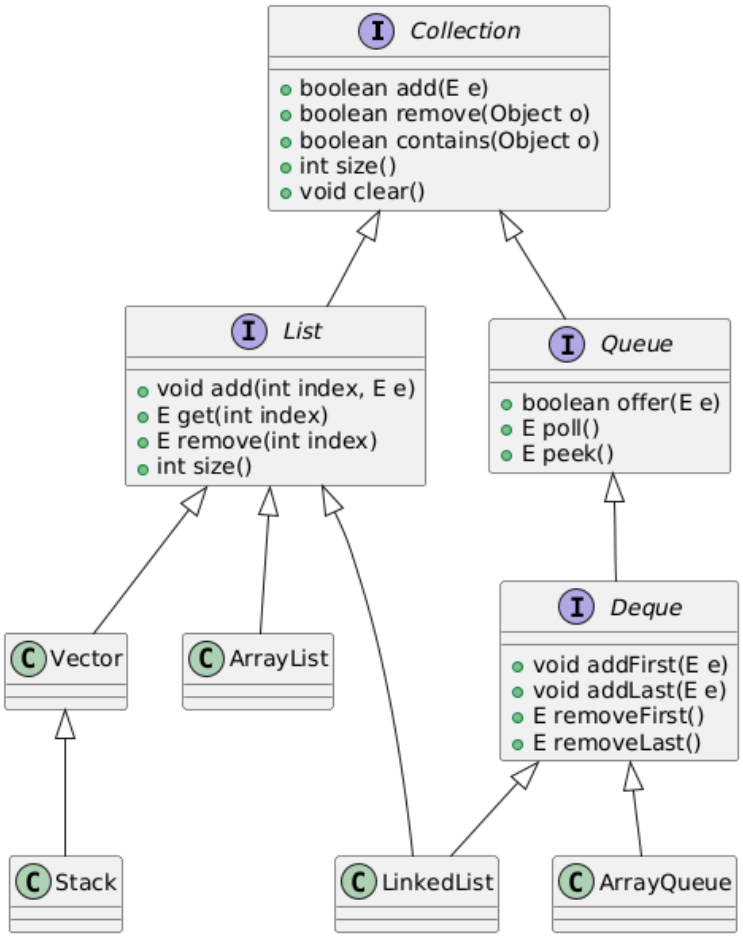

Рис 3. Діаграма класів, що реалізують інтерфейси List та Queue

Інтерфейс Set та основні класи, що його реалізують представлені на діаграмі:

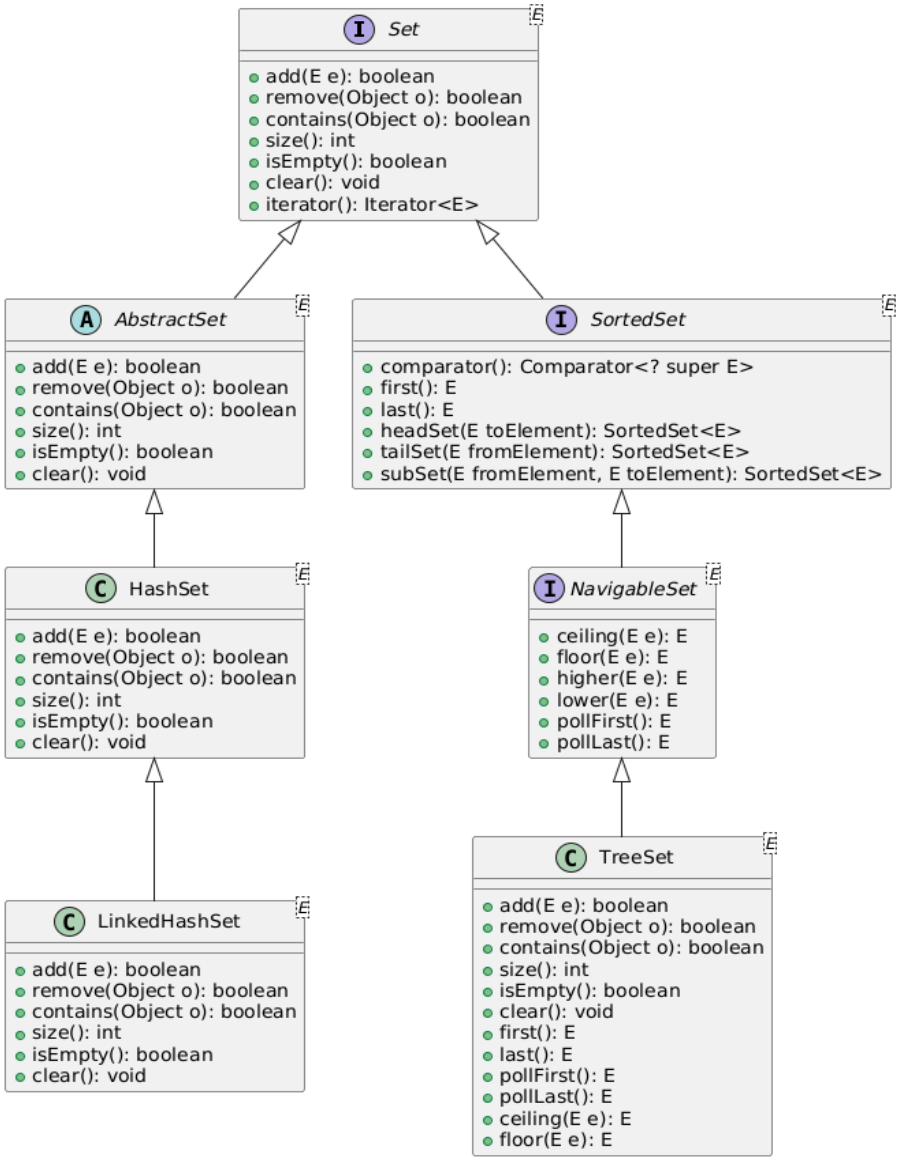

Рис 4. Діаграма класів, що реалізують інтерфейс Set

Основні операції інтерфейсу Map:

- **V put(K key, V value)**
- **V get(Object key)**
- **V remove(Object key)**
- **boolean containsKey(Object key)**
- **boolean containsValue(Object value)**
- **int size()**
- **boolean isEmpty()**
- **void clear()**
- **Set<K> keySet()**
- **Collection<V> values()**

Класи, об’єкти яких використовуються у якості ключів, повинні правильно реалізовувати методи hashCode() і equals(), а іноді і інтерфейс Comparable:

- hashCode(): цей метод має повертати однакове ціле значення для однакових об’єктів. Це значення використовується колекціями на основі хешу, як-от HashMap, для визначення сегмента, де зберігається пара ключ-значення.
- equals(Object o): цей метод має гарантувати, що він повертає true для об’єктів, які вважаються рівними. Якщо два ключі рівні відповідно до цього методу, вони повинні створити однаковий хеш-код.

Якщо клас ключа використовується в TreeMap, він повинен реалізовувати інтерфейс Comparable або мати Comparator, що надається під час створення об’єкту TreeMap.

Інтерфейс Map та основні класи, що його реалізують представлені на діаграмі:

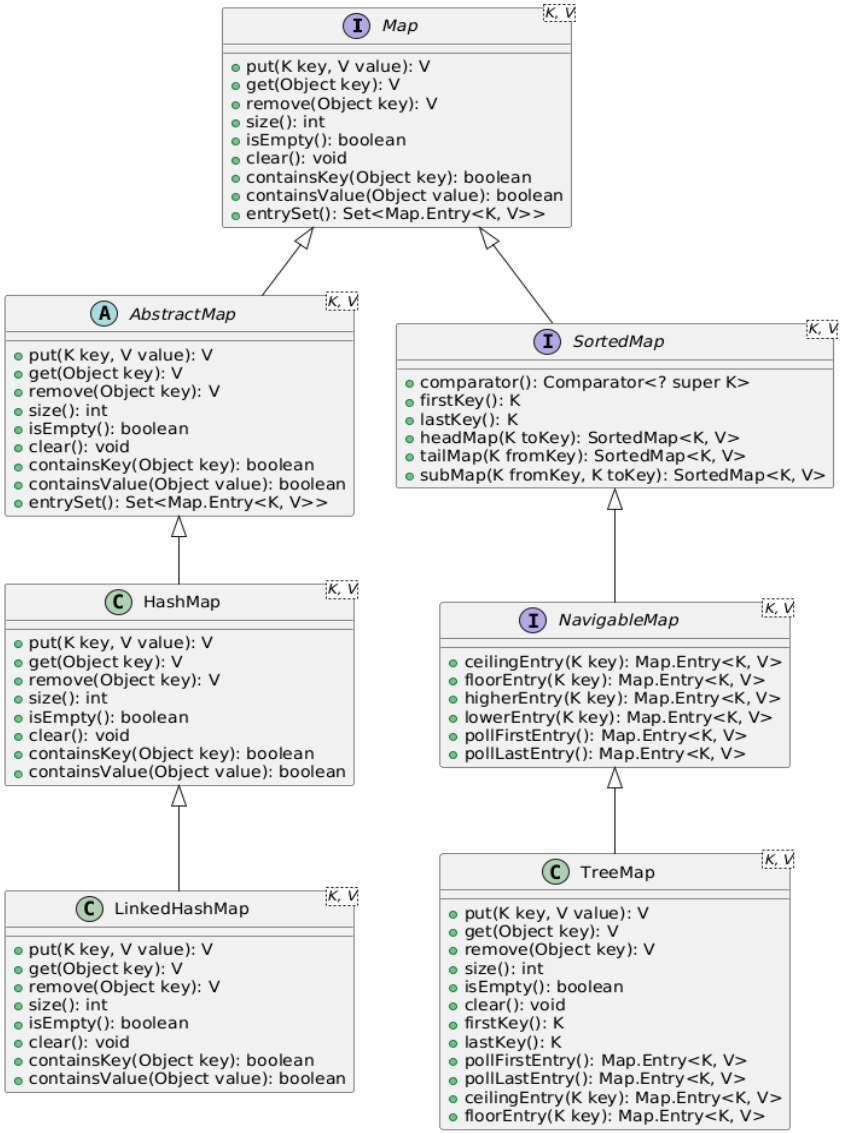

Рис 5. Діаграма класів, що реалізують інтерфейс Map

Приклади використання колекцій:

```
// Робота зі списком:
List<String> list = new ArrayList<>();
list.add("Alice");
list.add("Bob");
list.add("Alice");
System.out.println(list);
// Робота з множиною:
Set<String> set = new HashSet<>();
set.add("Alice");
set.add("Bob");
set.add("Alice");
System.out.println(set);
// Робота з чергою:
Queue<String> queue = new LinkedList<>();
queue.add("First");
queue.add("Second");
queue.add("Third");
System.out.println(queue.poll()); // Виводить "First"
```

Ітератор — це патерн проектування, який забезпечує стандартизований спосіб обходу колекції об’єктів без розкриття базової структури колекції. Властивості ітератора:

- Абстракція.
- Уніфікований інтерфейс.
- Збереження стану.

Основні методи:

- boolean hasNext() - повертає true, якщо в колекції є ще елементи, які не були оброблені. Це дозволяє перевірити, чи наступний виклик next() буде успішним.
- E next() - повертає наступний елемент ітерації. Цей метод також переміщує ітератор на наступну позицію.
- void remove() – видаляє елемент з колекції

Приклад використання ітераторів:

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class IteratorExample {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        fruits.add("Date");

        Iterator<String> iterator = fruits.iterator();
        while (iterator.hasNext()) {
            String fruit = iterator.next();
```

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

            if (fruit.equals("Banana")) {
                iterator.remove();
            }
        }
        System.out.println("After removal: " + fruits);
    }
}
```

Далі розглянемо особливості зберігання примітивних типів у колекціях. У Java не можливо безпосередньо зберігати примітивні типи даних у колекціях. Замість цього Java використовує класи-обгортки, щоб дозволити примітивним типам зберігатися в колекціях.

Класи-обгортки: Integer, Character, Byte, Short, Long, Float, Double, Boolean.

Java надає функцію під назвою автоупакування та авторозпаковування для полегшення перетворення між примітивними типами та їхніми відповідними класами-обгортками.

- Упакування (boxing): автоматичне перетворення примітивного типу у відповідний клас-обгортку під час додавання його до колекції. Наприклад, коли ви додаєте int до List<Integer>, Java автоматично перетворює його на Integer.
- Розпакування (unboxing): автоматичне перетворення класу-огортки назад до відповідного примітивного типу під час отримання його з колекції. Наприклад, коли ви отримуєте ціле число зі списку<Integer>, Java автоматично перетворює його назад на int.

Наведемо приклад роботи з примітивними типами:

```
import java.util.ArrayList;
import java.util.List;

public class PrimitiveInCollections {
    public static void main(String[] args) {
        List<Integer> integerList = new ArrayList<>();
        // autoboxing
        integerList.add(10);
        integerList.add(20);
        integerList.add(30);
        // unboxing
        for (Integer number : integerList) {
            System.out.println(number); // 10, 20, 30
        }

        List<Double> doubleList = new ArrayList<>();
        doubleList.add(3.14);
        doubleList.add(2.718);
        for (Double number : doubleList) {
            System.out.println(number); // 3.14, 2.718
        }
    }
}
```

Лекція 6. Комплексні компоненти для побудови графічного інтерфейсу користувача

Розглянемо такі компоненти бібліотеки Swing як слайдер, інкрементний регулятор меню, списки та таблиці.

Слайдер представлений класом JSlider. Це компонент для вибору значення з безперервного діапазону, пересуваючи ручку (регулятор). Зазвичай він використовується для вибору таких значень, як гучність, яскравість або будь-якого іншого значення у визначеному діапазоні.

Створення слайдеру:

```
// Вертикальний слайдер з діапазоном 0..100 та початковим значенням 50
JSlider slider = new JSlider(JSlider.VERTICAL, 0, 100, 50);
```

Налаштування слайдеру

```
slider.setMajorTickSpacing(15);// Великі позначки через кожні 15
одиниць
slider.setMinorTickSpacing(3);// Маленькі позначки через кожні 3
одиниці
slider.setPaintTicks(true);    // Відображення позначок
slider.setPaintLabels(true);   // Відображення числових міток
slider.setSnapToTicks(true);   // Прив’язати регулятор до позначок
```

Приклад використання слайдеру:

```
import javax.swing.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import java.awt.*;
public class SliderExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JSlider Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 200);
        JSlider slider = new JSlider(0, 100, 50);
        slider.addChangeListener(new ChangeListener() {
            @Override
            public void stateChanged(ChangeEvent e) {
                JSlider source = (JSlider) e.getSource();
                int value = source.getValue();
                System.out.println("Slider value: " + value);
            }
        });
        frame.add(slider);
        frame.setVisible(true);
    }
}
```

Інкрементний регулятор представлений класом JSpinner. Це компонент, який дозволяє користувачеві вибирати значення з послідовності або діапазону значень, наприклад чисел, дат або списків. Його часто використовують для вводу або безпосередньо значення, або для надання

можливості збільшувати/зменшувати значення за допомогою кнопок зі стрілками.

Використання JSpinner для вводу значення з діапазону:

```
// JSpinner із початковим значенням 10, мінімальним 0, максимальним 100
// і кроком 5
JSpinner spinner = new JSpinner(new SpinnerNumberModel(10, 0, 100, 5));
```

Використання JSpinner для вводу дати:

```
JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
dateSpinner.setEditor(new JSpinner.DateEditor(dateSpinner, "dd/MM/yyyy"));
```

Використання Jspinner для вибору значення:

```
String[] colors = {"Red", "Green", "Blue", "Yellow"};
JSpinner listSpinner = new JSpinner(new SpinnerListModel(colors));
```

Приклад використання інкрементного регулятору:

```
import javax.swing.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import java.awt.*;

public class SpinnerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JSpinner Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 200);
        JSpinner numberSpinner = new JSpinner(
            new SpinnerNumberModel(10, 0, 100, 1)
        );
        String[] colors = {"Red", "Green", "Blue", "Yellow"};
        JSpinner listSpinner = new JSpinner(
            new SpinnerListModel(colors)
        );

        listSpinner.addChangeListener(e -> {
            System.out.println("List Spinner Value: " +
            listSpinner.getValue());
        });
        frame.setLayout(new FlowLayout());
        frame.add(numberSpinner);
        frame.add(listSpinner);
        frame.setVisible(true);
    }
}
```

Звернемо увагу на використання лямбда виразу замість анонімного внутрішнього класу для програмування реакції на подію зміни значення інкрементного регулятору. Тут e — це об’єкт, що містить інформації про подію. Лямбда вирази були представлені у Java 8.

Для створення меню Swing пропонує наступні класи:

- JMenuBar: контейнер для всіх меню (наприклад, панель у верхній частині вікна).
- JMenu: представляє безпосередньо меню (наприклад, «Файл», «Редагувати»).
- JMenuItem: представляє окремі пункти меню, які можна натиснути, всередині JMenu.
- Меню можуть містити підменю (JMenu всередині іншого JMenu).

Меню також може містити прапорці або перемикачі. Для їх створення використовуються класи:

- JCheckfioxMenuItem – клас, що представляє елемент меню з прапорцем опцій.
- JRadioButtonMenuItem - клас, що представляє елемент меню з перемикачем.

Додати логіку до елементу меню можна за допомогою методу addActionListener. Альтернативний варіант – використання класу AbstractAction. Один і той самий об’єкт типу Action може бути пов’язаний з різними UI елементами. Action має метод setEnabled().

Приклад:

```
public class OpenAction extends AbstractAction {
    public OpenAction() {
        super("Open");
        putValue(SHORT_DESCRIPTION, "Opens a file");
        putValue(MNEMONIC_KEY, (int) 'O');
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Open action triggered");
    }
}
```

Далі наведемо приклад створення меню:

```
import javax.swing.*;
public class ActionMenuExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Action with JMenuItem Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);
        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu("File");
        Action openAction = new OpenAction();
        JMenuItem openItem = new JMenuItem(openAction);
        fileMenu.add(openItem);
        menuBar.add(fileMenu);
        frame.setJMenuBar(menuBar);
        frame.setVisible(true);
    }
}
```

Зазначимо, що при виконанні цієї програми під управлінням операційної системи Mac OS меню буде додано до вікна так саме, як і при виконанні на Windows. Це є абсолютно нетиповим для програм на Mac OS, бо втни використовують меню, що розташоване згори екрану. Для того, щоб виправити цю поведінку, нам необхідно додати спеціальну команду:

```
System.setProperty("apple.laf.useScreenMenuBar", "true");
```

Але виконувати цю команду має сенс лише на Mac OS. Тому спочатку потрібно перевірити операційну систему, під управлінням якої виконується програма. Оновлений приклад виглядає наступним чином (наведений повний приклад, який можна скомпілювати та виконати):

```
import java.awt.event.ActionEvent;
import javax.swing.*;

class OpenAction extends AbstractAction {
    public OpenAction() {
        super("Open");
        putValue(SHORT_DESCRIPTION, "Opens a file");
        putValue(MNEMONIC_KEY, (int) 'O');
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Open action triggered");
    }
}

public class App {
    public static void main(String[] args) {
       if (System.getProperty("os.name").toLowerCase().contains("mac")) {
            System.setProperty("apple.laf.useScreenMenuBar", "true");
        }
        JFrame frame = new JFrame("Action with JMenuItem Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);
        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu("File");
        Action openAction = new OpenAction();
        JMenuItem openItem = new JMenuItem(openAction);
        fileMenu.add(openItem);
        menuBar.add(fileMenu);
        frame.setJMenuBar(menuBar);
        frame.setVisible(true);
    }
}
```

Такий прийом доводиться часто використовувати при створенні крос-платформних програм. Бачимо, що більша частина коду працює на всіх цільових платформах, але є окремі інструкції, які виконуються лише на відповідних платформах. Цей випадок є найпростішим, бо System.setProperty може компілюватись на всіх платформах. Іноді буває, що деякі класи або методи існують лише на одній з платформ. Але програма повинна компілюватися на всіх платформах. Тоді для виклику таких методів

використовують більш складні прийоми. Наприклад механізм інтроспекції, що дозволяє викликати метод, маючи його назву у строковій змінній чи константі. Механізм інтроспекції у Java реалізований у пакеті reflection.

Наведемо більш складний приклад. Для зміни іконки застосунку на платформах Windows та Linux достатньо викликати метод setIconImage класу JFrame. Але це не працює на Mac OS. На Mac OS ми повинні використовувати клас Application з пакету com.apple.eawt. Але цього класу не існує на інших платформах, тому для використання цього класу доводиться використовувати механізм відображення (reflection).

Повний приклад представлений нижче:

```
package com.example.Lab1Test;
import javax.swing.*;
import java.awt.*;

public class App {

  public static void main(String[] args) {
    JFrame frame = new JFrame("Example");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(450, 140);

    ImageIcon icon = new ImageIcon("appIcon.png");
    frame.setIconImage(icon.getImage());

    // Для Mac OS можна встановити іконку використовуючи клас
    // com.apple.eawt.Application
    if (System.getProperty("os.name").toLowerCase().contains("mac")) {
      try {
          // Отримати класс Application та його метод
          // getApplication за допомогою reflection
          Class<?> applicationClass =
            Class.forName("com.apple.eawt.Application");
          Object application =
           applicationClass.getDeclaredMethod("getApplication")
            .invoke(null);

          // Викликати метод setDockIconImage для зміни іконки
          applicationClass.getDeclaredMethod("setDockIconImage",
            Image.class).invoke(application, icon.getImage());
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
    frame.setVisible(true);
  }
}
```

Списки у Swing представлені класом JList. JList використовується для відображення списку елементів для користувача, дозволяючи йому вибрати один або більше елементів.

Основні особливості списків:

- Одиночний або множинний вибір: можна налаштувати JList, щоб дозволити одиночний вибір (по одному елементу за раз) або

множинний (наприклад, утримуючи клавішу Ctrl, щоб вибрати кілька елементів).

- Спеціальна візуалізація: можна налаштувати спосіб відображення кожного елемента в списку, надавши спеціальний ListCellRenderer.
- Можливість прокручування: помістивши JList у JScrollPane, можна створювати списки, які можна прокручувати, для великих наборів даних.

Приклад використання списку:

```
   import javax.swing.*;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;
import java.awt.*;

public class JListExample {
    public static void main(String[] args) {
        String[] data = {"Item 1", "Item 2", "Item 3",
                         "Item 4", "Item 5"};
        JList<String> list = new JList<>(data);
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        // Обробка подій вибору елементу
        list.addListSelectionListener(new ListSelectionListener() {
            @Override
            public void valueChanged(ListSelectionEvent e) {
                if (!e.getValueIsAdjusting()) {
                    System.out.println("Selected: " +
                      list.getSelectedValue());
                }
            }
        });
        JScrollPane scrollPane = new JScrollPane(list);
        frame.add(scrollPane, BorderLayout.CENTER);
        frame.setVisible(true);
    }
}
```

Додаткові приклади використання списків наведені нижче.

Спеціальна візуалізація:

```
class MyCellRenderer extends DefaultListCellRenderer {
    @Override
    public Component getListCellRendererComponent(JList<?> list, Object
value, int index, boolean isSelected, boolean cellHasFocus) {
        JLabel label = (JLabel) super.getListCellRendererComponent(list,
value, index, isSelected, cellHasFocus);
        label.setIcon(new ImageIcon("path_to_icon.png"));
        label.setText("Item " + value);
        return label;
    }
}
list.setCellRenderer(new MyCellRenderer());
```

Додавання та видалення елементів:

```
DefaultListModel<String> model = new DefaultListModel<>();
model.addElement("New Item 1");
model.addElement("New Item 2");
model.removeElementAt(0);
```

Списки з множинним вибором:

```
list.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
List<String> selectedValues = list.getSelectedValuesList();
System.out.println("Selected items: " + selectedValues);
```

Клас JTable використовується для відображення та редагування табличних даних. Він дозволяє упорядковувати дані в рядки та стовпці, а також пропонує такі функції, як сортування, редагування та спеціальна візуалізація елементів.

Основні компоненти таблиці:

- Модель таблиці (JTableModel): TableModel визначає структуру та дані таблиці.
- Модель стовпців (JTableColumnModel): визначає поведінку та вигляд стовпців.
- Рендерер клітинок: керує відображенням окремих клітинок.
- Редактор клітинок: контролює спосіб редагування клітинок.

Приклад використання таблиці:

```
import javax.swing.*;

public class JTableExample {
    public static void main(String[] args) {
        String[][] data = {
                {"John", "Doe", "35"},
                {"Jane", "Doe", "30"},
                {"Jack", "Smith", "40"}
        };
        String[] columnNames = {"First Name", "Last Name", "Age"};
        JTable table = new JTable(data, columnNames);

        JScrollPane scrollPane = new JScrollPane(table);

        JFrame frame = new JFrame("JTable Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(scrollPane);
        frame.setSize(400, 200);
        frame.setVisible(true);
    }
}
```

Далі наведені приклади коду, що дозволяють використовувати деякі спеціальні функції таблиць.

Таблиці, що дозволяють редагування:

```
DefaultTableModel model = new DefaultTableModel(data, columnNames) {
    @Override
    public boolean isCellEditable(int row, int column) {
        return true;
    }
};
JTable table = new JTable(model);
```

Спеціальна візуалізація:

```
import javax.swing.table.DefaultTableCellRenderer;
import java.awt.Component;
class MyCellRenderer extends DefaultTableCellRenderer {
    @Override
    public Component getTableCellRendererComponent(JTable table, Object
value, boolean isSelected, boolean hasFocus, int row, int column) {
        Component c = super.getTableCellRendererComponent(table, value,
                                    isSelected, hasFocus, row, column);
        if (row %  2 == 0) {
            c.setBackground(Color.YELLOW);
        } else {
            c.setBackground(Color.WHITE);
        }
        return c;
    }
}
table.getColumnModel().getColumn(2).setCellRenderer(
  new MyCellRenderer()
);
```

Реакція на події:

```
table.getSelectionModel().addListSelectionListener(event -> {
    int selectedRow = table.getSelectedRow();
    if (selectedRow != -1) {
        System.out.println("Selected row: " + selectedRow);
    }
});
```

Сортування:

```
TableRowSorter<DefaultTableModel> sorter =
  new TableRowSorter<TableModel >(model);
table.setRowSorter(sorter);
```

Фільтрування:

```
RowFilter<DefaultTableModel, Object> filter =
RowFilter.regexFilter("Doe", 1);
sorter.setRowFilter(filter);
```

Спеціальне редагування:

```
TableColumn column = table.getColumnModel().getColumn(1);  // Column to
be edited with JComboBox
JComboBox<String> comboBox = new JComboBox<>(new String[]{"Doe", "Smith",
"Johnson"});
column.setCellEditor(new DefaultCellEditor(comboBox));
```

Лекція 7. Потоки вводу/виводу в Java

Почнемо з розгляду основних концепцій системи вводу/виводу.

Ввід — це дані, що надходять у програму. Це може бути файл, клавіатура, мережеве підключення чи інше джерело.

Вивід — це дані, що надсилаються з програми. Це може бути запис у файл, друк на консолі або надсилання даних через мережу.

Потоки є основною абстракцією, яку Java використовує для обробки операцій вводу/виводу. Java надає уніфікований API для вводу та виводу, незалежно від того, надходять дані з файлів, клавіатури, мережі чи інших джерел. Потоки є односпрямованими, тобто дані надходять лише в одному напрямку: або в програму (вхідний потік), або з програми (вихідний потік).

Потоки Java поділяються на два основні типи:

- Потоки байтів: робота з двійковими даними (таких як зображення, аудіо та інші нетекстові дані).
- Потоки символів: обробляйте текстові дані, піклуючись про кодування та декодування символів.

Потоки байтів представлені абстрактними класами InputStream та OutputStream. Основні класи: FileInputStream, FileOutputStream, BufferedInputStream, BufferedOutputStream.

Потоки символів представлені абстрактними класами Reader та Writer. Основні класи: FileReader, FileWriter, BufferedReader, BufferedWriter.

Потоки байтів використовуються для читання та запису двійкових даних. Це може включати будь-який тип даних, який природно не читається людиною, наприклад зображення, аудіофайли, відеофайли або будь-які спеціальні двійкові формати.

Кожен байт розглядається як одиниця даних, що не потребує додаткових перетворень, і кодування або декодування не застосовується.

Переваги байтових потоків:

- Гнучкість: вони можуть обробляти будь-які типи даних.
- Ефективність для двійкових даних.

InputStream — це абстрактний клас для читання байтів. Він надає методи для читання по одному байту з джерела (наприклад, файлу, мережі або масиву).

Основні методи класу InputStream :

- read(): читає один байт і повертає його як int. Повертає -1, коли досягнуто кінця потоку.
- read(byte[] b): читає байти в масив і повертає кількість прочитаних байтів.
- close(): закриває потік, щоб звільнити системні ресурси.

OutputStream — це абстрактний клас для запису байтів. Він надає методи для запису по одному байту до пункту призначення (наприклад, файлу, мережі чи масиву).

Основні методи класу OutputStream :

- write(int b): Записує один байт.
- write(byte[] b): записує масив байтів.
- close(): закриває потік, забезпечуючи запис усіх даних і звільнення системних ресурсів.

Перейдемо до класів, що реалізують байтові потоки, та які можна використовувати безпосередньо.

FileInputStream та FileOutputStream використовується для читання та запису файлів у двійковому форматі.

ByteArrayInputStream та ByteArrayOutputStream використовується для читання та запису даних у пам’яті з/до байтового масиву.

Класи BufferedInputStream та BufferedOutputStream:

- додають буферизацію до вхідних і вихідних потоків, підвищуючи продуктивність за рахунок зменшення кількості разів, коли дані зчитуються або записуються в базове джерело даних.
- класи зберігають дані в буфері, що дозволяє читати або записувати великі фрагменти даних одночасно.

Приклад використання класів FileInputStream та FileOutputStream:

```
FileOutputStream outputStream = new FileOutputStream("output.bin");
outputStream.write(65);  // 'A' у кодуванні ASCII
outputStream.close();

FileInputStream inputStream = new FileInputStream("output.bin");
int byteData = inputStream.read();
while (byteData != -1) {
  // Зчитування та обробка даних побайтово
  System.out.println(byteData);
  byteData = inputStream.read();
}
inputStream.close();
```

Використання BufferedOutputStream:

```
import java.io.*;

public class BufferedOutputExample {
    public static void main(String[] args) {
        try {
            FileOutputStream fileOutput =
              new FileOutputStream("output.txt");
            BufferedOutputStream bufferedOutput =
              new BufferedOutputStream(fileOutput);

            String data = "Buffered output example.";
```

```
            bufferedOutput.write(data.getBytes());
            bufferedOutput.flush();
            bufferedOutput.close();
            fileOutput.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Використання BufferedInputStream:

```
import java.io.*;

public class BufferedInputExample {
    public static void main(String[] args) {
        try {
            FileInputStream fileInput = new FileInputStream("input.txt");
            BufferedInputStream bufferedInput =
              new BufferedInputStream(fileInput);
            int data;
            while ((data = bufferedInput.read()) != -1) {
                System.out.print((char) data);
            }
            bufferedInput.close();
            fileInput.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Використання DataOutputStream:

```
import java.io.*;

public class BufferedInputExample {
    public static void main(String[] args) {
        try {
            FileOutputStream outputStream =
              new FileOutputStream("output.bin");
            DataOutputStream dataStream =
              new DataOutputStream(outputStream);
            dataStream.writeUTF("This is a string");
            dataStream.flush();
            dataStream.close();
            outputStream.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Далі розглянемо використання символьних потоків. Символьні потоки призначені для роботи з текстовою інформацією. Вони розроблені для роботи з символами Unicode, які можуть представляти практично будь-які символи будь-якої мови. Потоки символів працюють з текстовими даними,

перетворюючи байти на символи на основі вказаного кодування символів (UTF-8, UTF-16).

Unicode — це універсальний стандарт кодування символів, метою якого є забезпечення узгодженого способу представлення та обробки тексту майже всіма письмовими мовами, символами та шрифтами, що використовуються в усьому світі.

Unicode було створено, щоб усунути обмеження традиційних схем кодування символів, які часто підтримували лише обмежений набір символів (наприклад, ASCII, який підтримує лише 128 символів). Зі зростанням глобальної комунікації виникла потреба в стандарті, який міг би вмістити широкий спектр символів з різних мов і шрифтів.

Коли починали проектувати Unicode вважалося, що при використанні 2 байтів для кодування кожного символу можна буде закодувати всі символи всіх існуючих мов. У такому разі це було б універсальне і дуже просте кодування. Але згодом виявилося, що двох байтів недостатньо, тому виникли більш складні правила кодування.

У Unicode кожному символу присвоюється унікальний номер, відомий як кодова точка. Кодова точка зазвичай представляється у форматі U+XXXX, де «XXXX» число у шістнадцятковій системі счислення. Наприклад, буква «A» представлена як U+0041, а символ «漢» (китайський ієрогліф) представлений як U+6F22.

Юнікод може бути представлений у кількох формах кодування, найпоширенішими з яких є:

- UTF-8: кодування змінної довжини, яке використовує від 1 до 4 байтів для кожного символу. Воно зворотно сумісне з ASCII і широко використовується в Інтернеті.
- UTF-16: використовує 2 байти для більшості поширених символів і 4 байти для інших (зазвичай рідкісних символів). Його часто використовують у середовищах, де більшість тексту складається мовами, які використовують символи базової багатомовної площини.
- UTF-32: використовує фіксовану довжину 4 байти для всіх символів. Хоча воно спрощує деякі завдання програмування, воно є менш ефективним у порівнянні з UTF-8 або UTF-16.

Для роботи з текстом у кодуванні Unicode призначені класи Reader та Writer.

Reader — абстрактний клас для зчитування символів. Його основними методами є наступні:

- read(): читає один символ і повертає його як int. Повертає -1, коли досягнуто кінця потоку.
- read(char[] cbuf): читає символи в масив символів.
- close(): закриває потік, щоб звільнити системні ресурси.

Writer — абстрактний клас для запису символів. Основні методи цього класу:

- write(int c): записує один символ.
- write(char[] cbuf): записує масив символів.
- close(): закриває потік, забезпечуючи запис усіх даних і звільнення системних ресурсів.

Перейдемо до розгляду реалізацій символьних потоків. FileReader і FileWriter використовується для читання та запису файлів у текстовому форматі.

BufferedReader and BufferedWriter додають буферизацію до потоків читання та запису, покращуючи продуктивність за рахунок одночасного читання або запису більших фрагментів символів. BufferedReader також надає метод readLine(), який читає весь рядок тексту за раз.

Використання символьних потоків для запису даних:

```
FileWriter fileWriter = new FileWriter("output.txt");
fileWriter.write("This is an example of character stream.");
fileWriter.close();
```

Використання символьних потоків для зчитування даних:

```
BufferedReader bufferedReader = new BufferedReader(new
FileReader("input.txt"));
String line;
while ((line = bufferedReader.readLine()) != null) {
  System.out.println(line);
}
bufferedReader.close();
```

Класи InputStreamReader та OutputStreamWriter надають можливість встановлення кодування. Наведемо приклад зчитування файлу у кодуванні UFT-8:

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.io.IOException;
public class FileReadExample {
    public static void main(String[] args) {
        String filePath = "example.txt";
        String encoding = "UTF-8";
        try (FileInputStream fis = new FileInputStream(filePath);
             InputStreamReader isr =
               new InputStreamReader(fis, encoding);
             BufferedReader br = new BufferedReader(isr)) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

Зазначимо, що у цьому прикладі можна побачити, що при роботі з потоками можуть виникати виключні ситуації, які пов’язані з вводом/виводом. Розглянемо основні класи, що представляють виключні ситуації пов’язані з потоками.

IOException є суперкласом для всіх винятків, пов’язаних із вводом/виводом. Це сигналізує про те, що операція вводу/виводу сталася невдалою або була перервана. Ця виключна ситуація може статися під час читання, запису або закриття потоків.

FileNotFoundException — це підклас класу IOException, це виключення виникає, коли спроба відкрити файл, позначений вказаним шляхом, не вдається. Таке часто трапляється, якщо файл не існує або програма не має відповідних дозволів.

EOFException — це підклас класу IOException, цей виняток виникає, коли досягається неочікуваний кінець файлу під час спроби читання з потоку.

UnsupportedEncodingException — це також підклас класу IOException. Цей виняток виникає, коли певне кодування символів не підтримується.

По можливості рекомендується працювати з потоками, які були вже розглянуті у лекції. Але інколи буває необхідним використовувати специфічні можливості файлової системи або файлового вводу. У такому разі розробник буде використовувати безпосередньої файлові об’єкти відповідних класів.

Ключові характеристики роботи з файлами у Java:

- Обробка файлів: файловий ввід/вивід зосереджений на прямій взаємодії з файловою системою, дозволяючи створювати, видаляти та маніпулювати файлами.
- API вищого рівня: Java надає такі класи, як FileReader, FileWriter та File (з пакету java.io.file) та Files (з пакету java.nio.file) для операцій з файлами, що полегшує керування файлами.
- Дозволяє виконувати операції з файлами, такі як копіювання, переміщення та видалення.

У разі необхідності використання довільного доступу при роботі з файлами можна використовувати клас RandomAccessFile. Далі представлений приклад використання цього класу. У цьому прикладі ми працюємо з файлом у форматі png і визначаємо його розмір (висоту та ширину). Для зчитування розміру на потрібно зчитати лише деякі байти з заголовку файлу. Зчитування всіх інших байтів з файлу для даного завдання не є необхідним.

Приклад:

```
import java.io.IOException;
import java.io.RandomAccessFile;
public class PngImageSizeReader {
  public static void main(String[] args) {
    String filePath = "path/to/your/image.png";
    try {
        RandomAccessFile randomAccessFile =
          new RandomAccessFile(filePath, "r");
        randomAccessFile.seek(16);
        int width = readInt(randomAccessFile);
        int height = readInt(randomAccessFile);
        System.out.println("Image Width: " + width);
        System.out.println("Image Height: " + height);
        randomAccessFile.close();
    } catch (IOException e) {
      System.err.println("Error reading the PNG file: " +
        e.getMessage());
    }
  }
  private static int readInt(RandomAccessFile randomAccessFile)
    throws IOException {
        return (randomAccessFile.readUnsignedByte() << 24) |
                (randomAccessFile.readUnsignedByte() << 16) |
                (randomAccessFile.readUnsignedByte() << 8) |
                randomAccessFile.readUnsignedByte();
    }
}
```

Лекція 8. Серіалізація у Java. Створення діалогових вікон

Серіалізація — це процес перетворення об'єкта в потік байтів. Це дозволяє зберегти об’єкт у файл, надіслати його по мережі або перенести між різними віртуальними машинами Java (JVM). Після серіалізації об’єкт можна зберегти або передати, а потім реконструювати за допомогою десеріалізації, яка повертає процес назад, перетворюючи потік байтів назад на об’єкт.

У Java об’єкт серіалізується, якщо він реалізує інтерфейс Serializable, який є інтерфейсом маркером (це означає, що він не має методів для реалізації). Без реалізації цього інтерфейсу об’єкт не можна серіалізувати.

Серіалізований об’єкт може бути десеріалізований у точну копію оригіналу. Ця техніка іноді використовується для створення глибоких копій об’єктів.

Серіалізація виконується за допомогою класу ObjectOutputStream, який записує об’єкт у вихідний потік, зазвичай у файл або надсилається через мережу.

Десеріалізація (зворотний процес) використовує клас ObjectInputStream для читання потоку байтів і реконструкції об'єкта.

Коли Java серіалізує об’єкт, вона автоматично серіалізує всі поля (як примітивні, так і посилальні типи), які є частиною об’єкта, якщо вони не позначені як тимчасові.

- Примітивні типи (наприклад, int, float, boolean) безпосередньо записуються в потік байтів.
- об’єктні типи серіалізуються рекурсивно, якщо вони реалізують Serializable. Якщо будь-яке поле об’єкта не можна серіалізувати, серіалізація не вдасться, викинувши NotSerializableException.

Приклад серіалізації:

```
import java.io.Serializable;
class Person implements Serializable {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class SerializationExample {
    public static void main(String[] args) {
        String filePath = "path/to/your/image.png";
        try {
            Person obj = new Person("John", 30);
            FileOutputStream fileOut =
              new FileOutputStream("object.ser");
```

```
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(obj);
            out.close();
            fileOut.close();
        } catch (IOException e) {
            System.err.println("Error while writing to file ");
        }
    }
```

Приклад десеріалізації:

```
import java.io.Serializable;
class Person implements Serializable {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class SerializationExample {
    public static void main(String[] args) {
        String filePath = "path/to/your/image.png";
        try {
            FileInputStream fileIn = new FileInputStream("object.ser");
            ObjectInputStream in = new ObjectInputStream(fileIn);
            Person obj = (Person) in.readObject();
            in.close();
            fileIn.close();

        } catch (IOException e) {
            System.err.println("Error while reading file ");
        }
        catch (ClassNotFoundException y) {
            System.err.println("Class was not found");
        }
    }
}
```

Під час десеріалізації віртуальна машина Java (JVM) повинна знайти клас об’єкта, який десеріалізується. Якщо визначення класу не знайдено, виникає виключення ClassNotFoundException. Це може трапитися, якщо:

- клас було видалено.
- клас недоступний у поточному середовищі, наприклад, під час десеріалізації об’єктів, надісланих з іншої JVM.

Під час десеріалізації Java також перевіряє serialVersionUID, щоб переконатися, що серіалізовані дані відповідають поточній версії класу. Якщо є невідповідність, створюється виняткова ситуація InvalidClassException**.**

SerialVersionUID: це унікальний ідентифікатор для версії класу. Якщо клас змінюється (наприклад, додаються нові поля), serialVersionUID слід оновити, щоб відобразити зміни. Якщо ні, десеріалізація може завершитися помилкою.

Зазначимо, що статичні поля не серіалізуються бо серіалізація — це операція на рівні екземпляра. Статичні поля повторно ініціалізуються з визначення класу, коли клас завантажується.

Клас також може містити звичайні поля, які не будуть серіалізуватися. Це так звані тимчасові поля. Ключове слово **transient** використовується для позначення полів, які не слід серіалізувати. Коли поле оголошено як тимчасове, воно пропускається під час процесу серіалізації, тобто його значення не включається в потік байтів.

Тимчасові поля потрібні щоб уникати при серіалізації зберігання конфіденційних даних (наприклад, паролі, криптографічні ключі), які не можна зберігати чи передавати або виключити непостійні поля, значення яких є тимчасовим або похідним від інших полів.

Приклад використання тимчасових полів:

```
import java.io.*;
import java.time.LocalDate;
import java.time.Period;

public class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private LocalDate birthDate;
    private transient int age;
    public Person(String name, LocalDate birthDate) {
        this.name = name;
        this.birthDate = birthDate;
        age = calculateAge();
    }
    private int calculateAge() {
        return Period.between(birthDate, LocalDate.now()).getYears();
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        if (age == 0) {
            age = calculateAge();
        }
        return age;
    }
}
```

Також є можливість використовувати спеціальну серіалізацію щоб мати детальний контроль над тим, як серіалізується та десеріалізується об’єкт. Це особливо корисно в сценаріях, коли потрібно змінити типову поведінку серіалізації, яку забезпечує вбудований механізм серіалізації Java.

Спеціальна серіалізація може допомогти досягти різних цілей, наприклад, оптимізувати продуктивність, обробити конфіденційні дані або підтримувати зворотну сумісність.

Щоб реалізувати спеціальну серіалізацію, можна перевизначити два певних методи у своєму класі. Ці методи викликаються під час процесів серіалізації та десеріалізації, що дозволяє точно визначити, як стан об'єкта записується та зчитується з вихідного потоку.

Це методи:

- private void writeObject(ObjectOutputStream out) throws IOException
- private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException

Приклад реалізації спеціальної серіалізації:

```
import java.io.*;
import java.util.Base64;
public class User implements Serializable {
    private static final long serialVersionUID = 1L;
    private String username;
    private transient String password; // Original password (not
serialized)
    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }
    private String encryptPassword(String password) {
        return Base64.getEncoder().encodeToString(password.getBytes());
    }
    private String decryptPassword(String encryptedPassword) {
        return new String(Base64.getDecoder().decode(encryptedPassword));
    }
    public String getUsername() {
        return username;
    }
    public String getPassword() {
        // Повертає дійсний пароль після десеріалізації
        return password;
    }
    private void writeObject(ObjectOutputStream out) throws IOException {
        out.defaultWriteObject();

        String encryptedPassword = encryptPassword(password);
        // Записати закодований пароль
        out.writeObject(encryptedPassword);
    }
    private void readObject(ObjectInputStream in) throws IOException,
ClassNotFoundException {
        in.defaultReadObject();

        String encryptedPassword = (String) in.readObject();
        // Розшифрувати пароль
        this.password = decryptPassword(encryptedPassword);
    }
}
```

Окрім інтерфейсу Serializable у Java існує також інтерфейс Externalizable. Інтерфейс Externalizable розширює Serializable, але на відміну від Serializable, він вимагає, щоб програміст вручну визначив логіку серіалізації.

Реалізуючи Externalizable, розробник має повний контроль над тим, як об'єкт серіалізується та десеріалізується. Є можливість вибрати, які поля серіалізувати, як обробляти конфіденційні дані тощо.

При використанні Externalizable потрібно перевизначити два методи:

- void writeExternal(ObjectOutput out) throws IOException
- void readExternal(ObjectInput in) throws IOException, ClassNotFoundException

Приклад використання інтерфейсу Externalizable:

```
import java.io.*;
public class Employee implements Externalizable {
    private String name;
    private int age;
    public Employee() {}

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(name);
        out.writeInt(age);
    }
    @Override
    public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
        this.name = (String) in.readObject();
        this.age = in.readInt();
    }
}
```

Тепер повернемося до розмови про програмування графічного інтерфейсу користувача. Наступної важливою темою є створення діалогових вікон. Типова програма з GUI дуже рідко має лише одне основне вікно. Буває, що програма створює декілька вікон, з якими користувач може працювати одночасно. Але навіть якщо застосунок має всього одне головне вікно, він все одно час від часу відображає додаткові вікна, щоб повідомити користувача про якусь подію або щоб надати користувачеві зручний спосіб вводу даних.

Діалогові вікна це:

- спеціальні спливаючі вікна в графічному інтерфейсі користувача, які використовується для збору вхідних даних або надання інформації користувачам.

- тимчасові вікна, які з’являються поверх головного вікна програми та потребують взаємодії, перш ніж користувачі зможуть повернутися до головного інтерфейсу.

Призначення діалогових вікон:

- Збір вхідних даних: наприклад, коли користувачеві потрібно ввести такі дані, як ім’я користувача, пароль або будь-яку іншу інформацію (наприклад, у формі входу).
- Відображення інформації: діалогові вікна можуть відображати повідомлення для користувача, такі як попередження, сповіщення або інструкції (наприклад, повідомлення про помилки або сповіщення про успіх).
- Підтвердження дій: перед виконанням важливих дій (наприклад, видалення файлу або закриття програми) діалогові вікна можуть запитувати у користувачів підтвердження, гарантуючи, що вони усвідомлюють наслідки своїх дій (наприклад, «Ви впевнені, що хочете вийти?»)).

Діалогові вікна бувають модальними та немодальними.

Модальні діалоги:

- Модальні діалогові вікна блокують взаємодію користувача з іншими вікнами програми, доки діалогове вікно не буде закрито. Це означає, що користувач не може повернутися до головного вікна програми, доки не завершить роботу з модальним діалогом.
- Підходять для критичних сповіщень або підтверджень, які вимагають від користувача вжити заходів (наприклад, повідомлення про помилки, підтвердження видалення).
- Приклад: використання JOptionPane.showMessageDialog() створює модальне діалогове вікно, яке потребує дії користувача, перш ніж користувач зможе продовжити.

Немодальні діалоги:

- Немодальні діалогові вікна дозволяють користувачам взаємодіяти з іншими вікнами програми, коли діалог відкритий. Користувачі можуть перемикатися між діалоговим вікном та іншими компонентами без будь-яких обмежень.
- Підходять для діалогових вікон, які надають додаткові параметри чи інструменти, не перериваючи робочий процес користувача (наприклад, діалогові вікна налаштувань, вікна довідки).
- Приклад: спеціальне діалогове вікно, створене за допомогою JDialog, яке дозволяє здійснювати кілька взаємодій, залишаючись доступним разом із основною програмою.

Іноді буває необхідно відобразити дуже прості діалогові вікна для сповіщення користувача або для вводу даних. Зробити це з використанням

бібліотеки Swing надзвичайно просто. Розглянемо створення простих діалогів-сповіщень.

Діалогове вікно повідомлення — це тип діалогового вікна, яке відображає повідомлення для користувача та може надавати інформацію, попередження або повідомлення про помилки. Воно служить для інформування користувачів про стан програми, попередження про важливу інформацію або підтвердження успішного виконання дії.

JOptionPane — це службовий клас у Swing, який спрощує створення діалогових вікон у програмах Java. Він надає статичні методи для створення різних типів діалогів, включаючи діалоги повідомлень, діалоги введення та діалоги підтвердження.

Приклад створення простого діалогу-сповіщення:

```
import javax.swing.JOptionPane;
public class SimpleDialogExample {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null,
                                      "Hello, this is a message dialog!",
                                      "Message",
                                      JOptionPane.INFORMATION_MESSAGE);
    }
}
```

Діалогове вікно вводу — це тип діалогового вікна, яке пропонує користувачеві ввести дані. Зазвичай він представляє текстове поле для введення, що дозволяє користувачам надсилати інформацію назад до програми. Воно використовуються для збору введених користувачами даних, таких як імена, паролі або будь-яких інших даних, необхідних для роботи програми.

JOptionPane надає статичний метод для створення діалогових вікон введення, які є простими та ефективними для збору даних користувача.

Приклад створення простого діалогу для збору даних:

```
import javax.swing.JOptionPane;

public class InputDialogExample {
    public static void main(String[] args) {
        String name = JOptionPane.showInputDialog("Enter your name:");
        JOptionPane.showMessageDialog(null, "Hello, " + name + "!");
    }
}
```

Окрім простих діалогових вікон можна створювати будь які діалогові вікна, які необхідні для грамотної організації інтерфейсу користувача. Вони можуть дуже сильно відрізнятися одне від одного. Назвемо такі діалогові спеціальними.

Спеціальні діалогові вікна — це діалогові вікна, які створюються за допомогою класу JDialog. На відміну від стандартних діалогових вікон, наданих JOptionPane, користувальницькі діалогові вікна можуть містити будь-які необхідні компоненти, забезпечуючи більшу гнучкість у взаємодії з користувачем.

Спеціальні діалогові вікна використовуються для складніших взаємодій, які вимагають власних макетів, додаткових кнопок або спеціальних компонентів (наприклад, текстові поля, прапорці тощо), адаптованих до потреб конкретної програми.

Основні методи класу JDialog:

- setTitle() встановлює назву діалогового вікна.
- setSize() визначає розміри діалогу.
- setModal(true) робить діалогове вікно модальним, тобто блокуватиме введення в інші вікна, поки не буде закрито.

Приклад створення спеціального діалогового вікна:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class CustomDialog extends JDialog {
    private JTextField textField;
    private JButton okButton;
    private JButton cancelButton;
    private String userInput;
    public CustomDialog(JFrame parent) {
        super(parent, "Custom Dialog", true);
        setLayout(new GridLayout(2, 1));
        setSize(300, 150);
        setLocationRelativeTo(parent);
        JPanel inputPanel = new JPanel();
        inputPanel.setLayout(new FlowLayout());
        inputPanel.add(new JLabel("Enter your name:"));
        textField = new JTextField(15);
        inputPanel.add(textField);
        add(inputPanel);
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(1, 2));
        okButton = new JButton("OK");
        cancelButton = new JButton("Cancel");
        buttonPanel.add(okButton);
        buttonPanel.add(cancelButton);
        add(buttonPanel);
        okButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                userInput = textField.getText();
                setVisible(false);
            }
        });
        cancelButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
```

```
                userInput = null;
                setVisible(false);
            }
        });
    }
    public String getUserInput() {
        return userInput;
    }
}
```

Приклад використання спеціального діалогового вікна:

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class MainApp {
    public static void main(String[] args) {
        JFrame mainFrame = new JFrame("Main Window");
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainFrame.setSize(400, 200);
        mainFrame.setLayout(new FlowLayout());
        JButton openDialogButton = new JButton("Open Custom Dialog");
        mainFrame.add(openDialogButton);
        openDialogButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                CustomDialog dialog = new CustomDialog(mainFrame);

                // Відобразити спеціальний діалог
                dialog.setVisible(true);
                String input = dialog.getUserInput();
                if (input != null) {
                    JOptionPane.showMessageDialog(mainFrame,
                            "Hello, " + input + "!");
                } else {
                    JOptionPane.showMessageDialog(mainFrame,
                            "Dialog was cancelled.");
                }
            }
        });
        mainFrame.setVisible(true);
    }
}
```

Лекція 9. Узагальнені типи. Багатопоточність

Узагальнені типи (generics) в Java дозволяють створювати класи, інтерфейси та методи за допомогою параметрів типу. Замість жорсткого кодування конкретних типів використовується заповнювач (наприклад, <T>), який може представляти будь-який тип. Це означає, що можна написати код, який працює з різними типами даних, не переписуючи його для кожного з них.

Головною причиною використання узагальнених типів є безпека типів. Коли використовується узагальнена колекція з параметром типу, наприклад, List<String>, компілятор Java забезпечує додавання до списку лише об'єктів String. Це допомагає уникнути помилок під час отримання об'єктів із колекції, оскільки відомий точний тип об'єктів, які вона містить.

Продемонструємо корисність використання узагальнених типів на прикладі використання колекцій без параметрів типів.

```
List list = new ArrayList();
list.add("Java");
list.add(10);  // Нема помилки компіляції
String s = (String) list.get(1);// ClassCastException під час виконання
```

З використанням узагальнених типів приклад буде виглядати наступним чином:

```
List<String> list = new ArrayList<>();
list.add("Java");
list.add(10);  // Помилка на етапі компіляції
```

Бачимо, що узагальнені типи дозволяють виявити помилки ще на етапі компіляції. Це є великою перевагою, бо чим раніше буде виявлена помилка тим краще.

Ми вже стикалися з узагальненими типами при вивченні колекцій. Використання готових узагальнених типів є дуже простим. Це можна побачити з минулого прикладу. Але при написанні власних узагальнених типів потрібно розуміти багато додаткових правил.

По перше відмітимо, що узагальнені типи у Java схожі на шаблонні типи C++. При проектуванні узагальнених типів намагалися створити механізм схожий на шаблонні типи C++, але простіший для розуміння і реалізації. Але обмежитися простими правилами виходить лише для найпростіших випадків. Для досягнення гнучкості, що може бути порівняна з гнучкістю шаблонних типів, набір конструкцій, що стосуються узагальнених типів, довелося розширювати, що привело до значного ускладнення реалізації. Зараз Java має гнучкий і розвинений механізм узагальнених типів, який важко назвати простішим за механізм шаблонів, він просто інший.

Ключовою відмінністю узагальнених типів є те, що вони використовують «стирання типів» (type erasure). Під час компіляції компілятор забезпечує

безпеку типів, гарантуючи, що загальний код працює лише з указаними типами. Однак під час виконання вся інформація про тип стирається, а загальні типи замінюються їхніми базовими типами (часто це клас Object).

Пам'ятаємо, що C++ використовує інший механізм для шаблонів. Шаблони в C++ реалізуються за допомогою техніки, що називається інстанціюванням під час компіляції. Замість того, щоб видаляти інформацію про тип, компілятор генерує окремий код для кожного типу, який використовується з шаблоном. Java натомість має єдину реалізацію для всіх шаблонних типів. Після стирання генерики не існують під час виконання. Ось чому не можна визначити загальний параметр типу класу під час виконання за допомогою відображення (reflection).

Синтаксис узагальнених типів для класу або інтерфейсу:

```
class Box<T> {
    private T value;

    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}
```

Синтаксис узагальнених типів для методу:

```
public <T> void methodName(T param) {
    // Тіло методу, що використовує тип T
}
```

Обмеження в узагальнених типах дозволяє вказати, що загальний тип має бути підтипом (або супертипом) певного класу чи інтерфейсу. Це гарантує безпечне використання певних методів або властивостей у в загальному коді.

Синтаксис обмежені:

- <T extends ClassName>: параметр типу T має бути або класом ClassName, або підкласом ClassName. Це називається верхньою межею.
- Також можна застосувати кілька обмежень (за допомогою інтерфейсів), наприклад <T extends ClassA & InterfaceB>.
- <T super X>: параметр типу T має бути X або суперкласом X.

Більш складний приклад:

```
public <T extends Comparable<T>> void sortList(List<T> list) {
    Collections.sort(list);
}
```

Уявимо, що потрібно створити узагальнений тип, що буде процівати з будь якими числами. Зробити це можна наступним чином:

```
public class Calculator<T extends Number> {
    private T num1;
    private T num2;

    public Calculator(T num1, T num2) {
        this.num1 = num1;
        this.num2 = num2;
    }

    public double sum() {
        return num1.doubleValue() + num2.doubleValue();
    }
}
```

Маска позначається символом ? і використовуються для визначення невідомого типу. Символи підстановки роблять ваш загальний код більш гнучким і придатним для повторного використання, дозволяючи працювати з різними типами більш загальним способом.

Маски бувають наступних видів:

- Необмежена: ?
- Маска з верхньою межею: ? extends Тип
- Маска з нижньою межею: ? super тип

Наведемо приклади використання масок в узагальнених типах:

```
public void printList(List<?> list) {
    for (Object obj : list) {
        System.out.println(obj);
    }
}
public void printNumbers(List<? extends Number> list) {
    for (Number number : list) {
        System.out.println(number.doubleValue());
    }
}
public void addNumbers(List<? super Integer> list) {
    list.add(10);
    list.add(20);
}
public static <T> void sort(List<T> list, Comparator<? super T>
comparator) {
}
```

Для кращого розуміння масок розглянемо схожі приклади з використанням масок та без.

Приклад із використанням Wildcards (? extends Number):

```
public void processNumbers(List<? extends Number> list1, List<? extends
Number> list2) {
    // Немає гарантії, що list1 та list2 мають однаковий тип елементів
    Number first1 = list1.get(0);  // Працює, тому що `? extends Number`
гарантує, що це Number
    Number first2 = list2.get(0);  // Те саме

    // list2.add(first1); // ПОМИЛКА: Неможливо додати, тому що точний тип
невідомий
}
```

У цьому прикладі два списки можуть містити елементи різних типів (наприклад, List<Integer> і List<Double>). Немає способу забезпечити, щоб list1 і list2 мали однаковий тип.

Приклад із використанням параметра типу (<T extends Number>)

```
public <T extends Number> void processNumbers(List<T> list1, List<T> list2)
{
    // Обидва списки повинні містити елементи одного й того самого типу
    T first1 = list1.get(0);  // `T` є однаковим для обох списків
    T first2 = list2.get(0);  // Той самий `T`, що й у list1

    list2.add(first1); // Працює, тому що `T` гарантовано відповідає типу
}
```

Тут і list1 і list2 повинні мати однаковий тип елементів (наприклад, List<Integer> і List<Integer>). Параметр типу T забезпечує узгодженість типів.

Тепер перейдемо до наступної важливої теми — багатозадачності та багатопоточності.

Багатозадачність означає здатність комп'ютерної системи виконувати кілька завдань або процесів одночасно. Це забезпечує ефективне використання центрального процесору (ЦП), гарантуючи, що він обробляє кілька операцій без простою.

У багатозадачності операційна система розподіляє процесорний час для кожного процесу чи завдання таким чином, щоб вони виконувалися одночасно. Це досягається за допомогою перемикання контексту, коли ЦП швидко перемикається між завданнями, зберігаючи стан поточного завдання та завантажуючи стан наступного. Також зазначимо, що багато завдань можуть дійсно виконуватись одночасно, бо сучасні процесори зазвичай мають багато ядер.

Багатозадачність буває кооперативною та витісняючою. Кооперативна багатозадачність працює наступним чином: кожен процес або потік добровільно передає контроль над ЦП, щоб дозволити виконувати інші завдання. Система покладається на кожне завдання, щоб вказати, коли воно

завершено або готове відмовитися від контролю, дозволяючи іншим завданням продовжити.

Витісняюча багатозадачність побудована на тому, що операційна система контролює розподіл часу ЦП на завдання. Вона примушує виконуване завдання зупинятися через певний проміжок часу, переключаючись на інше завдання, щоб забезпечити чесне виконання та запобігти монополізації центрального процесора одним завданням.

Багатозадачність на основі процесів означає одночасну роботу кількох програм або процесів. Кожен процес працює незалежно і має власний простір пам'яті та системні ресурси.

Багатозадачність на основі потоків означає одночасну роботу кількох потоків в одному процесі. Ці потоки спільно використовують ту саму пам'ять і ресурси, але можуть виконуватися незалежно.

Багатопотоковість означає здатність системи виконувати декілька потоків одночасно. Потік — це найменша одиниця процесу, яка може виконуватися незалежно. Багатопотоковість дозволяє програмі виконувати кілька операцій одночасно, підвищуючи ефективність і продуктивність.

У Java багатопотоковість дозволяє запускати кілька частин програми (або завдань) одночасно, що дозволяє програмі виконувати більше ніж одну дію одночасно. Наприклад, у той час як один потік обробляє ввід від користувача, інший потік може обробляти операції вводу/виводу файлів або бази даних.

Основні переваги багатопоточності:

- Швидкодія: багатопочність дозволяє виконувати кілька завдань одночасно, використовуючи сучасні багатоядерні процесори для максимального використання ЦП. Замість того, щоб один потік виконував всю роботу, кілька потоків можуть виконувати різні завдання одночасно, прискорюючи загальну програму.
- Паралелізм: багатопочність забезпечує паралелізм, тобто кілька потоків можуть виконуватися «паралельно» з точки зору програміста, навіть на одноядерному ЦП, де операційна може розділяючи їх у часі. Для завдань, пов'язаних із вводом/виводом, багатопочність дозволяє програмі обробляти інші завдання в очікуванні завершення вводу/виводу, запобігаючи простою.
- Ефективне використання ресурсів: потоки спільно використовують ту саму пам'ять і системні ресурси в рамках процесу, що зменшує накладні витрати на перемикання контексту порівняно з кількома процесами. Завдяки цій моделі спільних ресурсів багатопочність легша, ніж створення окремих процесів для кожного завдання.
- Маштабування: багатопочність дозволяє програмам ефективно масштабуватися, особливо при роботі у системах з декількома ядрами або процесорами. З більшою кількістю ядер ЦП збільшується

можливість паралельного виконання потоків, що робить багатопочні програми здатними обробляти більші робочі навантаження та покращує загальну продуктивність.

Отже при багатопоточному програмуванні основною одиницею, яка забезпечуватиме всі зазначені переваги, буде потік. Потік має свій життєвий цикл, який ми і розглянемо далі.

У своєму життєвому циклі потік проходить через такі стани:

- Новий (New): потік створений, але ще не запущений.
- Готовий до виконання (Runnable): потік готовий до виконання, але ще не виконується.
- Виконується (Running): потік виконується.
- Заблокований (Blocked or Waiting): потік не виконується, тому що очікує на готовність необхідного йому ресурсу.
- Завершений (Terminated): потік виконав свою роботу і більше не виконується та не може перейти до стану «Готовий до виконання».

Типовий життєвий цикл потоку: **New → Runnable → Running → Terminated**. Потік також може багато разів переходити між станами **Running → Waiting → Runnable**.

Для створення потоку у Java виконайте наступні кроки :

1. Створіть клас, який реалізує інтерфейс Runnable.
2. Перевизначте метод run(), щоб визначити поведінку потоку.
3. Створіть екземпляр Thread і передайте свій об’єкт Runnable конструктору Thread.
4. Викличте метод start(), щоб почати виконання.

Приклад створення потоку:

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread is running");
    }
}
public class Test {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        thread.start(); // Запустити потік
    }
}
```

Основні методи класу Thread:

- **start()** – переводить потік зі стану New до стану Runnable
- **sleep(long millis)** – призупиняє виконання потоку на визначений час
- **join()** – очікування завершення іншого потоку

- **interrupt()** – використовується щоб сигналізувати потоку, що він повинен припинити свою роботу. Це не зупиняє потік примусово, а замість цього встановлює прапорець перерваного потоку. Потоки можуть періодично перевіряти цей прапорець за допомогою isInterrupted() або обробляти InterruptedException.
- **isAlive()** – повертає true, якщо потік все ще активний

Важливою темою при багатопоточному програмуванні є синхронізація потоків. У багатопоточному середовищі, де кілька потоків можуть одночасно отримувати доступ до спільних ресурсів (наприклад, змінних, об'єктів або структур даних), необхідно керувати доступом до цих ресурсів, щоб запобігти конфліктам або неправильним результатам.

Коли кілька потоків отримують доступ до спільних даних, можуть виникнути такі проблеми, як умови змагання або неузгоджені стани. Синхронізація гарантує, що лише один потік може одночасно отримати доступ до критичного ресурсу, запобігаючи конфліктам.

Метод може бути визначений як синхронізований, що гарантує, що лише один потік може виконувати його одночасно для даного об'єкта. Замість синхронізації всього методу можна синхронізувати лише окремий блок коду в межах методу.

Найпростіший спосіб синхронізації у. Java це використання ключового слова synchronized для позначення синхронізованих блоків або методів.

Приклад використання синхронізованих методів:

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }
    public synchronized int getCount() {
        return count;
    }
}
```

Приклад використання синхронізованих блоків:

```
class Counter {
    private int count = 0;

    public void increment() {
        synchronized (this) {  // Synchronized block
            count++;
        }
    }

    public int getCount() {
        synchronized (this) {  // Synchronized block
            return count;
        }
    }
}
```

Навчальне видання

КОНСПЕКТ ЛЕКЦІЙ
з дисципліни «Крос-платформне програмування»
(для студентів усіх форм навчання напряму підготовки
121 «Інженерія програмного забезпечення»)

Укладач:
Дьомін Максим Костянтинович

Редактор *М.К.Дьомін*
Техн. редактор *М.К.Дьомін*
Оригінал - макет *М.К.Дьомін*

Підписано до друку _____
Формат $60\times84\,{}^{1}\!/_{16}$. Папір типограф. Гарнитура *Times*.
Друк офсетний. Ум. друк. арк.___. Обл.-вид.арк._____.
Тираж ___ прим. Вид. №_____. Замовл. №_____. Ціна договірна.

**Видавництво Східноукраїнського національного університету
імені Володимира Даля**

Свідоцтво про реєстрацію:
Адреса університета: вул. Іоанна Павла II, 17
м. Київ, 01042, Україна
e-mail: vidavnictvoSNU.ua@gmail.com.