Вінницький національний технічний університет
(повне найменування вищого навчального закладу)

Факультет інформаційних технологій та комп'ютерної інженерії
(повне найменування інституту, назва факультету (відділення))

Кафедра програмного забезпечення
(повна назва кафедри (предметної, циклової комісії))

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Методи та програмні засоби підтримки Java коду на платформі Apple iOS»

Виконав: студент 2 курсу, групи 2ПІ-22м спеціальності 121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Мисловський І. В.
(прізвище та ініціали)

Керівник: к.т.н., доцент кафедри ПЗ
Черноволик Г. О.
(прізвище та ініціали)
«18» грудня 2023 р.

Опонент: к.т.н., доцент кафедри ОТ
Тарновський М. Г.
(прізвище та ініціали)
«18» грудня 2023 р.

Допущено до захисту
Зав. кафедри ПЗ Романюк О. Н.
«18» грудня 2023р.

Вінниця ВНТУ – 2023 рік

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти другий (магістерський)
Галузь знань 12 – Інформаційні технології
Спеціальність 121 – Інженерія програмного забезпечення
Освітньо-професійна програма – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
Романюк О.Н.
«19» вересня 2023 р.

ЗАВДАННЯ
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Мисловському Ігорю Валерійовичу

1. Тема роботи – «Методи та програмні засоби підтримки Java коду на платформі Apple iOS»

Керівник роботи:

Черноволик Галина Олександрівна, к. т. н., доцент кафедри ПЗ,

затверджені наказом вищого навчального закладу від «18» вересня 2023 р. № 247.

2. Термін подання студентом роботи :

5 грудня 2023 р.

3. Вихідні дані: цільова платформа – Apple iOS, цільова версія JVM – 16, функціонал для реалізації – JEP 395 (Java Records); операційна система – MacOSx; середовище розробки – Intellij Idea ; мова програмування – Java.

4. Зміст текстової частини: вступ; аналіз існуючих засобів для компіляції Java на платформі Apple iOS; розробка методів і алгоритмів для підтримки JEP 395; тестування отриманих результатів; економічна частина; висновки; додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): мета і задачі роботи; наукова новизна та практична цінність роботи; блок-схема алгоритму синтезу методів; класи і методи додатку; основні результати роботи.

6. Консультанти розділів бакалаврської дипломної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	завдання прийняв
1–4	Черноволик Г. О.., доц. каф. ПЗ, к.т.н	19.09.2023	05.10.2023
5	Кавецький В. В., доц. каф. ЕПВМ, к.т.н	17.11.2023	25.11.2023

7. Дата видачі завдання 19 вересня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Обґрунтування вибору методу розробки та постановка задач	20.09.23 – 02.10.23	вик
2	Розробка архітектури та алгоритмів програмного продукту	03.10.23 – 23.10.23	вик
3	Аналіз і вибір мови програмування та середовища розробки	16.10.23 – 23.10.23	вик
4	Розробка програмного продукту	24.10.23 – 13.11.23	вик
5	Тестування програми	14.11.23 – 21.11.23	вик
6	Розробка економічної частини	17.11.23 –25.11.23	вик
7	Оформлення матеріалів до захисту МКР	22.11.23 – 01.12.23	вик

Студент _____ (підпис) **Мисловський І. В.** _____ (прізвище та ініціали)

Керівник магістерської кваліфікаційної роботи _____ (підпис) **Черноволик Г.О.** _____ (прізвище та ініціали)

АНОТАЦІЯ

УДК 004.932

Мисловський І. В. Методи та програмні засоби підтримки Java коду на платформі Apple iOS. Магістерська кваліфікаційна робота зі спеціальності 121 – Інженерія програмного забезпечення, освітня програма – Інженерія програмного забезпечення. Вінниця: ВНТУ, 2023. – 89 с.

На укр. мові. Бібліогр.: 34 назва; рис.: 20; табл.: 7.

Метою роботи є розробка алгоритмів і методів для забезпечення виконання сучасного Java коду на платформі Apple iOS.

В роботі проаналізовано сучасний стан наявних рішень і засобів підтримки Java коду на платформі Apple iOS. Розглянутий та реалізований метод підтримки Java Records (JEP 395) в рішенні RoboVM. Показано, що ця підтримка дозволяє використовувати існуючу Java екосистему на цільовій платформі, остільки досягнутий рівень підтримки Java є достатнім та сучасним.

Програмне забезпечення розроблено в інтегральному середовищі Intellij Idea використовуючи мову програмування Java. Досягнуті результати дозволяють використовувати Java код версії JVM16 на платформі Apple iOS.

Ключові слова: Java, JEP-395, Apple iOS, RoboVM, Java, програма.

ABSTRACT

Myslovsky I. V. Methods and software tools for supporting Java code on the Apple iOS platform. Master's thesis on the specialty 121 - Software engineering, educational program - Software engineering. Vinnytsia: VNTU, 2023. - 89 p.
In Ukrainian language. Bibliographer: 34 titles; fig.: 20; tabl.: 7.

The goal of the work is to develop algorithms and methods to ensure the execution of modern Java code on the Apple iOS platform.

The paper analyzes the current state of available solutions and means of supporting Java code on the Apple iOS platform. Considered and implemented Java Records support method (JEP 395) in the RoboVM solution. It is shown that this support allows the use of the existing Java ecosystem on the target platform, as long as the achieved level of Java support is sufficient and up-to-date.

The software is developed in the integrated environment of Intellij Idea using the Java programming language. The achieved results allow the use of Java code version JVM16 on the Apple iOS platform.

Keywords: Java, JEP-395, Apple iOS, RoboVM, Java, program.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. Сьогодні мобільні пристрої проникли в усі сфери повсякденного життя. І вже давно мобільний телефон є не тільки засобом зв'язку, але й персональним помічником в усіх сферах. Придатність до використання у цих сферах забезпечується не самим мобільним пристроєм, а мільйоном різноманітних додатків, створених сотнями тисяч розробників. У кожного розробника мобільних є ціль зробити швидкий, сучасний мобільний додаток, затративши мінімальний обсяг часу. Ринок мобільних платформ розділений між двома основними гравцями: Google Android та Apple iOS. Кожен розробник прагне отримати максимальну вигоду зі свого продукту тож бажає охопити всі основні платформи (в основному дві вищенаведені).

Тож зазвичай для отримання однакового функціоналу необхідно розробити дві копії додатки використовуючи наступні мови програмування:

- Java/Kotlin для Google Android;
- Swift для Apple iOS.

Це накладає подвійні вимоги як для бюджету, кількості персоналу за затрат, оскільки потребує розробки двох копій продукту.

Оптимізаціє даної проблеми – є використання спільної кодової бази. І рішенням є використання Java коду на платформі Apple iOS. На сьогодні немає рішень, які є сучасними та повністю задовільнять вимоги сьогодення. А тому, попри всі переваги використання існуючих методів нативної розробки, слід розробити методи, що наддадуть можливість використовувати Java код на платформі Apple iOS та дозволить використовувати єдину кодову базу між платформами Apple iOS та Google Android.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконувалася згідно плану виконання наукових досліджень на кафедрі програмного забезпечення

Мета та завдання дослідження. Метою роботи є збільшення ефективності використання єдиної кодової бази на наявних мобільних

платформах за рахунок розробки методів па програмних засобів виконання сучасного Java коду на платформі Apple iOS.

Основними задачами дослідження є:

- провести аналіз існуючих методів і засобів виконання Java коду на платформі Apple iOS;
- виявити необхідний та достатній обсяг функціоналу, що необхідно реалізувати;
- запропонувати - новий метод реалізації необхідного функціоналу;
- реалізувати програмні компоненти на основі запропонованого методу;
- провести експериментальні дослідження розроблених засобів.

Об’єкт дослідження – виконання сучасного Java коду на платформі Apple iOS.

Предмет дослідження – методи та програмні засоби віртуальної машини Java, необхідні для виконання Java коду на платформі Apple iOS для отримання користувачем можливості використовувати єдиної кодової бази на наявних мобільних платформах.

Методи дослідження. У процесі досліджень використовувались:

- методи розробки Java додатків для реалізації проекту;
- методи аналізу сучасного Java байт-коду перевірки коректності структури згенерований Java Records;
- методи синтезу байт-коду для внесення змін і заміни фабрик статичними реалізаціями;
- методи об’єктно-орієнтованого програмування та шаблони програмування для розробки архітектури класів та сервісів програмного забезпечення.

Наукова новизна отриманих результатів.

1. Подальшого розвитку дістав метод реалізації Java Records (JEP 395) для MobiVM, який відрізняється реалізацією записів через Java класи та дозволяє синтезувати необхідні методи.

2. Подальшого розвитку набув метод використання динамічного зв'язування коду DynamicInvoke, який відрізняється обробкою bootstrap методу і дозволяє розширити підтримку динамічного зв’язування транслятором MobiVM.

Практична цінність отриманих результатів. Практична цінність одержаних результатів полягає в тому, що на основі отриманих в магістерській кваліфікаційній роботі методів запропоновано рішення для виконання сучасного Java коду на платформі Apple iOS, що дозволяє розробникам реалізувати переваги єдиної кодової бази, зменшити час на розробку та подальшу підтримку продукту.

Особистий внесок здобувача. Усі розроблені методи, наукові результати та результати тестування, викладені у магістерській кваліфікаційній роботі, отримані автором особисто.

Апробація роботи. Результати роботи доповідалися на Всеукраїнській науково-практичній інтернет-конференції молодих вчених та студентів «Сучасні інформаційні системи та технології» (2023 р., м. Херсон).

Публікації. Основні результати досліджень опубліковано в науковій праці [2] у збірниках матеріалів конференцій.

РОЗДІЛ 1
АНАЛІЗ ЗАВДАННЯ ТА ОБҐРУНТУВАННЯ ЗАДАЧ

1.1 Аналіз доступних мов програмування для платформи Apple iOS

Платформа Apple iOS є популярною і займає вагоме місце в світі. Тож не дивно, що для неї існує досить великий набір програмного забезпечення, яке дозволяє створювати додатки для запуску на платформі Apple iOS.

Розглянемо наявні мови програмування та рішення, що дозволяють їх використовувати.

Swift - це високорівнева мова програмування, розроблена компанією Apple, яка стала стандартною мовою для розробки додатків для платформ iOS, macOS, watchOS та tvOS.

Swift визначається чистим та легким синтаксисом, що полегшує розуміння коду розробниками. Це забезпечує більшу продуктивність та прискорює процес розробки. Крім того, Swift орієнтований на безпеку програмування, має вбудовані механізми захисту від помилок, такі як опціональні типи даних та автоматичне усунення аркушів пам'яті.

Swift відзначається високою швидкістю в порівнянні з іншими мовами програмування. Він використовує передові техніки оптимізації, щоб забезпечити ефективну роботу додатків. Важливо зазначити, що Swift надає прямий доступ до низькорівневих оптимізацій, що дозволяє розробникам максимально використовувати можливості платформи.

Swift підтримує розробку різноманітних додатків, починаючи від простих мобільних додатків і закінчуючи складними системами. Ця мова підтримує об'єктно-орієнтоване та функціональне програмування, що надає розробникам гнучкість у виборі підходів до розробки.

Оскільки Swift є офіційною мовою від Apple, вона гармонійно інтегрується зі специфічними технологіями та сервісами Apple. Розробники можуть легко використовувати функції, такі як Core Data для роботи з базами даних, Core Animation для анімації та інші API, що допомагають оптимально використовувати можливості платформи.

Swift має активну та зростаючу спільноту розробників, що призводить до швидкого розвитку мови та надання нововведень. Регулярні оновлення та документація від Apple роблять Swift привабливою мовою для розробників.

Objective-C, як мова програмування для розробки додатків на платформі iOS, має свої особливості та технічні характеристики, але також стикається з певними обмеженнями, що можуть робити її менш привабливою порівняно з більш сучасними мовами, зокрема Swift.

Objective-C використовує динамічну типізацію, що може забезпечувати гнучкість при роботі з об'єктами під час виконання програми. Це було важливою особливістю для довгого часу, але зараз може вважатися менш сучасним підходом через важкочитаемий синтаксис та збільшення кількості коду, який потрібно написати.

Однією з обмежень Objective-C є відсутність певних сучасних можливостей, які надає Swift. Swift дозволяє використовувати лямбда-функції, покращені ітератори та інші синтаксичні зручності, які полегшують розробку та підтримку коду.

Окрім того, Objective-C використовує "повільний" динамічний об'єктний підхід, що може впливати на продуктивність порівняно з "швидким" статичним підходом Swift. Swift визначається вищою швидкодією та оптимізацією для використання у сучасних мобільних пристроях.

Хоча Objective-C була ключовою мовою для розробки додатків для iOS, поступове впровадження Swift свідчить про тенденцію до переходу до більш сучасних технологій. Тим не менше, деякі проекти продовжують використовувати Objective-C, особливо ті, що мають існуючий код на цій мові.

Xamarin, фреймворк для крос-платформеної розробки на основі мови програмування C#, надає розробникам можливість створювати мобільні додатки, включаючи ті для iOS. Розглянемо технічні аспекти Xamarin та обговоримо чому, не зважаючи на свої переваги, деякі розробники можуть вважати його застарілим.

Xamarin вирізняється своєю крос-платформеністю, дозволяючи розробникам використовувати загальний код для реалізації додатків на різних мобільних платформах, включаючи iOS. Використання мови програмування C# і інтеграція з .NET підтримують екосистему Microsoft, що може бути зручним для команд, які вже працюють з цими технологіями.

Один із методів оптимізації в Xamarin - це використання JIT компіляції, яка перетворює код C# в машинний код під час виконання програми. Це може покращити продуктивність, але при цьому існує вартість оптимізації, так як це може вплинути на час запуску додатка.

Основним принципом Xamarin є можливість використовувати спільний код для розробки додатків на різних платформах. Це дозволяє зменшити витрати часу та зусиль при розробці для iOS та інших систем.

Не зважаючи на свої переваги, Xamarin може вважатися застарілим у порівнянні з більш новими фреймворками, такими як SwiftUI (для нативної розробки додатків на платформі iOS) або Flutter (фреймворк для крос-платформеної розробки). Ці фреймворки надають нові підходи та зручності, зокрема візуальні інструменти для розробки і модерні елементи інтерфейсу користувача.

MobiVM представляє собою конструктивну альтернативу для розробників, які вибирають використання мови програмування Java для створення мобільних додатків для платформи iOS.

Можливість крос-платформеного розроблення в MobiVM відкриває можливості для розробників використовувати загальний код для створення додатків для різних мобільних платформ, включаючи iOS. Це дозволяє командам використовувати існуючий Java-код або надавати переносимість коду між платформами.

MobiVM дозволяє розробникам використовувати багатий набір Java-бібліотек та інструментів, що робить його привабливим для тих, хто шукає різноманітні інструменти для вирішення конкретних завдань.

Для розробників, які мають значний досвід роботи з Java, MobiVM стає зручним інструментом, оскільки він дозволяє їм використовувати свої навички та досвід в мобільному розробленні для платформи iOS.

MobiVM може слугувати як чудовий компроміс між крос-платформеністю та нативністю. Розробники можуть зберігати переваги крос-платформеного підходу, а водночас отримувати доступ до нативних функцій та бібліотек для платформи iOS.

Зазначимо, що вибір MobiVM може бути розглянутий як позитивна опція, особливо у випадках, коли розробники мають великий обсяг існуючого Java-коду або коли команда вже володіє досвідом роботи з Java.

1.2 Аналіз стану систем виконання Java коду

Java - це високорівнева, об'єктно-орієнтована та платформо-незалежна мова програмування.

Процес перетворення початкового коду Java на виконавчий код включає кілька кроків, від написання вихідного коду до виконання його на цільовому обладнанні [1]:

- Написання коду програми на мові Java (source code);
- Компіляція в байт-код;
- Виконання.

Файли вихідного коду мають розширення .Java і містять текстовий код, який визначає класи, методи та інші елементи програми.

Вихідного код компілювати в байт-код. Компіляція - це процес перекладу високорівневого коду Java в більш низькорівневий формат. Цей крок виконується Компілятором Java (зазвичай javac) і призводить до створення файлів з розширенням .class. Байт-код - це не машинний код, а незалежне від платформи представлення програми[2].

Байт-код - це набір інструкцій, призначених для виконання віртуальною машиною Java (JVM). Це бінарний формат, але він не зв'язаний з конкретним обладнанням або операційною системою. Ця незалежність - ключова особливість

Java. Файли байт-коду можуть розповсюджуватися та виконуватися на будь-якій платформі, на якій є сумісна JVM.

JVM інтерпретує або компілює байт-код в машинний код, специфічний для цільової платформи.

Виконання на цільовому обладнанні виконується платформо-залежними JVM, яка виділенням пам'яті, збором сміття та іншими низькорівневими задачами, забезпечуючи рівень абстракції від обладнання. Програма виконується як окремий процес та взаємодіє з операційною системою хосту за потреби.

1.3 Порівняльний аналіз наявних JVM на платформі Apple iOS та вибір системи для подальшого вдосконалення

Java[3] – це потужна та універсальна мова програмування, яка знайшла широке застосування в різних галузях розробки програмного забезпечення.

Однією з ключових переваг Java є її переносимість. Програми, написані на Java, можна виконувати на будь-якому пристрої чи операційній системі, яка підтримує віртуальну машину Java (JVM). Це робить Java ідеальним вибором для кросплатформенної розробки.

Java використовує об'єктно-орієнтований підхід до програмування, що сприяє модульності, повторному використанню коду та полегшує розвиток та супровід програм.

Java є популярною мовою для розробки веб-додатків та веб-сервісів, зокрема завдяки технологіям, таким як JavaServer Faces (JSF)[4], Spring[5] та іншим.

Java підтримує виконання багатьох завдань одночасно, що робить її ідеальним рішенням для розробки систем, які вимагають обробки багатьох паралельних операцій.

Java є однією з основних мов програмування для розробки додатків для мобільної платформи Android[6].

Java наявна на таких операційних системах як Windows, macOS, Linux, Solaris та інших unix подібних операційні системи. Широко застосуємо на серверах для створення корпоративних додатків та веб-сервісів.

Використовується для розробки додатків для Android, що робить його ключовим компонентом для мобільної розробки. Використовується в вбудованих системах, таких як роутери, телевізори, медичне обладнання та інші.

Спеціалізовані версії Java використовуються для великих даних(Big Data), машинного навчання та інших сучасних областей.

На сьогоднішній день розвиток та підтримку мови Java веде корпорація Oracle. Вони активно випускають нові версії мови, вносять зміни та покращення, а також забезпечують підтримку користувачам.

Java залишається важливим рішенням у світі програмування, забезпечуючи велику гнучкість та ефективність у багатьох галузях розробки програмного забезпечення. На 2023 рік остання версія мови програмування Java та JVM – 21[7].

Нажаль, на сьогоднішній день підтримка Java Virtual Machine (JVM) для платформи Apple iOS залишається обмеженою через відсутність офіційної підтримки зі сторони Apple.

Apple активно просуває використання мов програмування Swift[8] та Objective-C[9] для розробки додатків для своїх пристроїв. Вони надають інструменти та середовище розробки, такі як Xcode, які оптимізовані для цих мов. Офіційна підтримка JVM відсутня, що обмежує можливості використання Java на iOS.

Розробка для iOS також пов'язана зі специфічними вимогами та обмеженнями, які накладає Apple. Наприклад, додатки повинні бути підписані та пройти обов'язковий процес апробації для розміщення в App Store.

Оскільки розробники iOS часто використовують інструментарій, наданий Apple, вони можуть отримати доступ до оптимізацій та інтеграцій, які недоступні для JVM. Це може включати глибоку інтеграцію з функціоналом пристроїв, оптимізації продуктивності, а також доступ до новітніх функцій iOS.

У деяких випадках розробники можуть використовувати інші віртуальні машини, такі як Xamarin для мови C# або Flutter для Dart, які дозволяють більш широко використовувати один код для розробки додатків для різних платформ.

Тобто платформа Apple iOS офіційно не підтримується.

Та можливість використання Java програм на платформі Apple iOS можлива використовуючи наступні сторонні реалізації JVM:

- Gluon;
- Multi-OS Engine;
- MobiVM;

Розглянемо кожну платформу більш детально:

Gluon [10]: крос-платформна фреймворк, який дозволяє використовувати JavaFX[11], Java-бібліотеку для створення користувацьких інтерфейсів, для побудови iOS, Android і настільних додатків. Він також підтримує GraalVM, код компілятора, який може оптимізувати Java-код для iOS.

- До переваг можно віднести:
- Активна підтримка виробника;
- 11-та версія Java;
- Наявність комерційної підтримки.

До недоліків:

- Орієнтованість на JavaFx (що обмежує можливість використання в сферах де вона цей функціонал не потрібен);
- 11-та версія Java (порівняно з сучасною 21ю);
- обмеженість безкоштовних ліцензій;
- відсутність API для роботи з функціями системи Apple iOS.

Multi-OS Engine(MOE)[12]: Це інструмент, який був розроблений Intel і пізніше придбаний Migeran. Це відкрите програмне забезпечення для розробки мобільних додатків для платформи iOS за допомогою мови програмування Java. В основі цього інструменту лежить ідея забезпечити можливість використання Java-коду для створення додатків, які можуть працювати на пристроях з операційною системою iOS.

Однією з ключових особливостей MOE є його інтеграція з популярною інтегрованою середою розробки - IntelliJ IDEA. Розробники можуть використовувати звичайні інструменти та середовища, що вони вже знають та використовують для розробки на мові Java.

MOE використовує технологію виконання коду на основі LLVM[13], що дозволяє перетворювати Java-код у виконуваний код для платформи iOS. Цей підхід забезпечує високий рівень продуктивності та оптимізації, що важливо для мобільних додатків.

MOE також надає доступ до нативних бібліотек та функціоналу iOS, що дозволяє використовувати повний спектр можливостей пристроїв Apple.

На даний момент підтримка продукту зупинена.

До переваг можно віднести:

- opensource and freeware;
- наявність API для роботи з функціями системи Apple iOS.

До недоліків:

- підтримка зупинена – нові версії не випускаються;
- 8-та версія Java (порівняно з сучасною 21ю);
- застарілий API для роботи з функціями системи Apple iOS (рівень iOS13).
- відсутність налагоджувача.

MobiVM[14] – це інструмент для розробки мобільних додатків, який дозволяє використовувати мову програмування Java для створення додатків для платформи iOS. Технічно MobiVM використовує технологію перетворення Java-коду у виконуваний код, який може бути використаний на iOS-пристроях.

Основна ідея MobiVM полягає в тому, щоб дозволити розробникам використовувати існуючий Java-код для створення додатків, які можуть працювати на пристроях Apple. Це досягається завдяки перетворенню Java-байткоду у виконуваний код, оптимізований для роботи на iOS-пристроях.

MobiVM інтегрується з інтегрованими середовищами розробки, такими як IntelliJ IDEA, що полегшує процес розробки та дозволяє використовувати звичайні інструменти для роботи з Java.

Одна з ключових технічних особливостей MobiVM - це його можливість використовувати LLVM (Low Level Virtual Machine)[13] для генерації оптимізованого виконуваного коду. Це дозволяє досягти високої продуктивності та ефективності виконання додатків на iOS-пристроях.

MobiVM також забезпечує взаємодію з API та функціоналом iOS, дозволяючи розробникам використовувати всі можливості платформи.

Загалом, технічно MobiVM робить можливим використання Java для розробки додатків для iOS, забезпечуючи перетворення коду та інтеграцію з інтегрованими середовищами розробки, що робить його зручним інструментом для розробників, які вже працюють з Java та бажають розширити свої додатки на iOS.

До переваг можна віднести:

- opensource and freeware[14];
- наявність API для роботи з функціями системи Apple iOS (ios 16);
- робочий налагоджувач, інтеграція з AndroidStudio/Intellij Idea;
- активна підтримка та періодичний випуск нових версій.
- орієнтованість на розробку нативних iOS додатків, широке коло користувачів з проекту libGDX[15].

До недоліків:

- 11-та версія Java (порівняно з сучасною 21ю);
- відсутність комерційної підтримки:

Крім наведених вище існують інші проекти, які дозволяють виконувати Java код на платформі Apple iOS.

- Codename One[16];
- j2objc[17];
- XMLVM[18].

Та всі вони мають критичні недоліки, як неповна відповідність JVM, низька розповсюдженість, відсутність підтримки чи припинення розробки. Порівняємо всі системи між собою, результати зведемо в таблицю 1.1.

Таблиця 1.1 – Порівняльна характеристика наявних для iOS JVM

Критерії	Gluon	MOE	MobiVM
Активний статус проекту	+	-	+

Версія Java > 11	+	-	+
OpenSource	-	+	+
Комерційна підтримка	+	-	-
Api для роботи з iOS	-	+/-	+
Орієнтованість саме на iOS	-	+	+
Наявність налагоджувача	-	-	-
Загальна оцінка	42%	35%	71%

В цілому, тільки системи з відкритим кодом мають потенціал для покращення. В результаті для розгляду залишаються MOE та MobiVM. Останній виглядає краще за всіма параметрами: активному статусу проекта, орієнтованість на iOS та наявність засобів розробки, таких як налагоджувач. Ці критерії і підтверджує загальна оцінка в 71% від критеріїв.

Отже, MobiVM обрано для покращення та реалізації відсутнього функціоналу мови програмування Java, який був добавлений нових версіях Java, а саме JEP 395: Records [19].

1.4 Аналіз методів розв'язання задачі

Старіння підтримуваної версії Java призводить до того, що наявний JVM не можливо використовувати з сучасною екосистемою Java. Адже використання Java має один із основних плюсів це досить розвинута та розповсюджена бібліотека доступних компонентів та бібліотек. Тільки Maven Central розміщає понад 260 тисяч бібліотек які завантажуються більше ніж 70 мільйонів раз на тиждень[20].

Неможливість бути сумісним з екосистемою – означає смерть продукту. Підтримка повного функціоналу Java досить затратний процес для однієї людини та можливо вирішити проблеми, які блокують використання вже зараз.

Випуски версій Java відбуваються періодично, та не всі вони приносять нові зміни, які є критичні. Для аналізу методів, які дозволять виконувати сучасний Java код на платформі Apple iOS розглянемо значущі нововведення, які стали зміною в самій мові програмування і не можуть бути реалізовані сторонніми бібліотеками чи власним кодом.

Виділимо наступні “віхи”, які має сенс розглянути для впровадження:

- Java 14: JEP358 Helpful NullPointerExceptions[21]
- Java 16: JEP395 Records [19]
- Java 17: JEP409 Sealed Classes [22]
- Java 21: JEP444 Virtual Threads [23]

Розглянемо кожну з цих новинок детально.

1.4.1 Java 14: JEP358 Helpful NullPointerExceptions.

JEP 358, також відомий як "Helpful NullPointerExceptions," є пропозицією для покращення способу, яким виводяться винятки NullPointerException у Java. Основна мета - зробити ці винятки більш інформативними та допомагати розробникам в легкому виявленні місця, де сталася помилка.

За допомогою JEP 358, стандартний NullPointerException тепер містить більше інформації про те, яке саме значення було null. Тепер виняток містить ім'я змінної чи об'єкта, яке призвело до винятку, що дозволяє розробникам швидше визначити джерело помилки.

Якщо виклик складається з ланцюжка методів, які викликають один одного, новий підхід дозволяє більш ефективно відслідковувати, який саме метод призвів до NullPointerException. Такий аналіз ланцюжків викликів робить виняток більш деталізованим та зрозумілим.

Додатково, якщо деякий параметр чи результат методу має спеціальну анотацію, яка вказує на його можливість мати значення null (@Nullable, @NotNull, тощо), ця інформація теж включається до винятку. Це може бути корисно при використанні анотацій для покращення безпеки коду.

Покращення дозволяє включати точні номери строк коду у стек викликів, що полегшує розуміння, де саме в коді виникла помилка.

Виняток тепер може містити ім'я змінної у вигляді спеціального маркера {name}, що дозволяє легко розпізнати ім'я, коли воно доступне, та ігнорувати його, якщо воно відсутнє.

Оголошення про дружелюбніші винятки NullPointerException спрощує роботу розробників, зокрема тих, хто намагається розібратися в коді, в якому виникає помилка.

Взагалі, JEP 358 додає значно більше інформації та деталей до винятків NullPointerException, щоб полегшити життя розробникам та зробити їх код більш безпечним і зрозумілим.

Хоча, даний функціонал і є дуже приємний, в той же час, він не є критичним, оскільки не запроваджує критичні зміни і є повністю сумісним зі старим кодом. Цей функціонал додано в Java 14.

1.4.2 Java 16: JEP395 Records. JEP 395

Java Enhancement Proposal 395, також відомий як "Records," є нововведенням у мові програмування Java, введеним починаючи з версії Java 16. Ця функція спрощує створення і використання класів для представлення даних шляхом надання вбудованої підтримки для створення так званих "record classes" (класів-записів). Розглянемо докладніше, як це працює:

Записи вводять новий синтаксис для декларативного визначення класів даних. Клас-запис може бути визначений одним рядком коду, забезпечуючи коротку та зрозумілу реалізацію.

Всі основні методи, такі як методи equals(), hashCode(), і toString(), створюються автоматично компілятором. Це зменшує кількість шаблонного коду, який розробник повинен писати.

Клас-запис є немутабельним, що означає, що його екземпляри не можуть бути змінені після створення. Це сприяє безпеці та простоті використання в многопотокових середовищах.

Внутрішні змінні-поля (fields) класу-запису автоматично визначаються як final і приватні, і до них можна отримати доступ через автоматично створені методи-акцесори.

Розробник може додавати власні методи до класу-запису, які доповнюють автоматично створені методи. Це забезпечує гнучкість та розширює функціональність класів-записів.

Записи дозволяють визначати локальні конструктори для зручного створення екземплярів класу зазначеними значеннями.

Класи-записи можуть реалізовувати інтерфейси та розширювати інші класи. Це робить їх універсальними для використання в різноманітних контекстах.

1.4.3 Java 17: JEP409 Sealed Classes.

JEP 409, також відомий як "Sealed Classes," є нововведенням у мові програмування Java, яке було впроваджено починаючи з версії Java 17. Цей функціонал дозволяє вводити обмеженя на спадкування класів, що поліпшує контроль над ієрархією класів та дозволяє точніше визначати, які класи можуть бути спадкоємцями.

Sealed classes дозволяють вказувати обмеження на те, які класи можуть бути спадкоємцями даного класу. Це досягається шляхом визначення обмеженого списку класів чи інтерфейсів, які можуть бути підкласами.

Sealed classes можуть бути абстрактними або конкретними. Абстрактний sealed class може мати своїх спадкоємців, які можуть бути конкретними класами.

За допомогою sealed classes розробник може визначити поведінку для всіх можливих спадкоємців. Це спрощує процес розробки та дозволяє більш точно контролювати структуру програми.

Для тих класів, які не обмежуються sealed, вводиться новий модифікатор "non-sealed," що дозволяє класам мати необмежений список спадкоємців.

Компілятор перевіряє, чи всі можливі варіанти спадкоємців були визначені для sealed classes. Це дозволяє виявити та уникнути потенційних порушень структури класів.

У sealed classes зберігається інформація про можливих спадкоємців, яка може бути використана для подальшого аналізу, наприклад, під час рефакторингу чи інструментів рефлексії.

Sealed classes можуть використовуватися у спільноті з іншими функціями мови, такими як шаблони та анотації, для розширення функціональності та зручності використання.

Введення sealed classes дозволяє використовувати більш точні та безпечні ієрархії класів, зменшуючи кількість непередбачених варіантів спадкування та роблячи код більш зрозумілим та підтримуваним.

1.4.4 Java 21: JEP444 Virtual Threads.

JEP 444, також відомий як "Virtual Threads," є нововведенням у мові програмування Java, введеним починаючи з версії Java 21. Ця функція вводить віртуальні потоки, які дозволяють ефективно використовувати так звані легкі потоки для роботи з великою кількістю задач.

Віртуальні потоки є новою абстракцією, яка реалізується на рівні віртуальної машини Java (JVM) та використовує легкі потоки (Lightweight Threads) для виконання завдань. Це дозволяє створювати значно більше потоків, ніж традиційні справжніх потоки, що полегшує роботу з великою кількістю одночасних задач.

Віртуальні потоки ефективно використовують ресурси, оскільки їх велику кількість можна обслуговувати меншою кількістю справжніх потоків. Це полегшує роботу з великою кількістю одночасних завдань та зменшує витрати пам'яті та процесорних ресурсів.

Для роботи з віртуальними потоками вводиться новий API для створення, керування та взаємодії з легкими потоками. Це дозволяє розробникам ефективно використовувати цю нову концепцію потоків.

Віртуальні потоки використовують кооперативне планування, що означає, що керування передається від одного потоку до іншого згідно з визначеними точками виклику. Це дозволяє уникнути накладання навантаження планування на операційну систему та зменшує накладні витрати.

Вводиться підтримка віртуальних потоків у CompletableFuture, що полегшує асинхронне програмування та обробку паралельних операцій.

Віртуальні потоки ідеально підходять для ситуацій, коли велика кількість завдань вимагає одночасного виконання. Це може бути корисно у веб-серверах, мережевих додатках та інших сценаріях високої одночасності.

Віртуальні потоки можуть співіснувати з традиційними справжніми потоками, що дозволяє поступово впроваджувати їх у вже існуючі додатки.
Введення віртуальних потоків дозволяє більш ефективно використовувати потоки в Java, забезпечуючи високий рівень одночасності та ефективне використання ресурсів системи.

1.4.5 Вибір методу для реалізації.

Всі перераховані нововведення одночасно корисні в той же час одночасно можуть стати критичними в майбутньому у разі відсутності реалізації.

Одночасно, немає сенсу фокусуватися на нововведені, яке було додане в більш пізній версії Java і не реалізовувати попереднє.

Такий критерій автоматично відсіює JEP409 і JEP444 додані в Java 17 та 21. Залишаючи вибір між JEP358 Helpful NullPointerException та JEP395 Records.

Helpful NullPointerException – є дуже, дуже корисним нововведенням, яке сильно полегшує життя розробникам, але в той же час не є критичним. На відміну від JEP395 Java Records.

Останній функціонал розширю мову Java новими визначеннями, що робить код, реалізований у Java 16 з цим фунціоналом несумісним з попередніми версіями Java.

Записи вводять в Java новий спосіб створення та використання класів даних, зменшуючи кількість коду та забезпечуючи високу зрозумілість коду. Вони спрощують роботу розробників, особливо при роботі з об'єктами, представляючи дані в програмі.

Однією з новинок, які були фіналізовані в Java 16 та широко використовуються це Java Records (JEP 395). Records – це записи, які є класами, що діють як прозорі носії для незмінних даних. Записи дозволяють виразно моделювати об'єкти, спрощують написання коду.

Оскільки Java Records є частино мови програмування Java починаючи з версії 16, відсутність підтримки цього фунціоналу автоматично робить реалізацію віртуальної машини Java несумісною з кодом, реалізованим з використанням Java records.

І з плином часу кількість несумісного програмного забезпечення буде тільки збільшуватись, що приведе до западу продукту.

Підтримка базових елементів мови Java – є найкращим методом для доступності і можливості використання сучасного Java коду на платформі Apple iOS. JEP 395 є прикладом саме такого коду.

Отже для продовження можливості подальшого використання MobiVM на платформі iOS – потрібно реалізувати підтримку JEP 395.

1.5 Висновки

У першому розділі було розглянуто стан наявних віртуальних Java машин які дозволяють виконувати Java код на платформі Apple iOS. Розглянуті недоліки та переваги наявних рішень. Аналіз показав відсутність наявності підтримки сучасної версія мови програмування на цій платформі в усіх наявних версіях. Що відкриває широкі можливості для впровадження.

Як базу для подальшого розвитку і провадження сучасного функціоналу мови програмування Java на платформі Apple iOS було визначено як основну - JEP 395: Records та для платформи для реалізації обрано MobiVM.

РОЗДІЛ 2
РОЗРОБКА АЛГОРИТМІВ ТА МЕТОДІВ РОБОТИ JEP JEP 395: Records

2.1 Загальний принцип роботи MobiVM

Шлях програми від початкового коду на Java до бінарного файлу, що можна виконати на платформі Apple iOS виглядає як показано на рисунку 2.1.

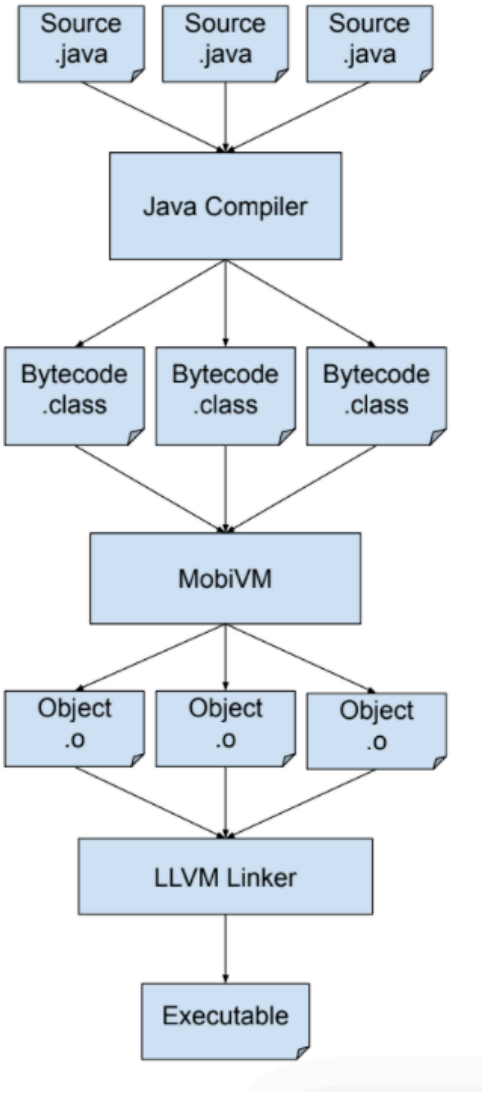

Рисунок 2.1 – Життєвий шлях програми на Java від початкового коду до бінарного виконавчого файлу

Зі схеми видно, що рішення MobiVM не є самостійним, а лише ланкою процесу, який складається з наступних етапів, вхідних та вихідних даних:

- Вхідні данні - початковий код програми. Один або більше тестових файлів з кодом програми на мові програмування Java;
- Java compiler – виконує компіляцію початкового коду програми в байт-код віртуальної машини Java. Результати – для кожного .java файлу один або більше .class файлів.
- Проміжні дані – байт-код. Для класичної віртуальної Java машини цього достатньо, оскільки JVM може виконувати байт-код. В випадку Apple iOS, віртуальні машини відсутні і виконання їх не можливо, відповідно ці данні мають бути оброблені далі.
- MobiVM – виконує трансляцію байт-коду в об’єктний код цільової платформи (в нашому випадку Apple iOS).
- Проміжні дані - об’єктний код – це бінарний формат програми в машинних кодах цільової платформи, який готовий для упаковки в виконавчий файл компонувальником.
- Компонувальник – LLVM Linker[24] який входить до пакету Xcode[25], середовища розробки для продуктів Apple. Компонувальник об'єднує всі файли об’єктного коду разом для створення єдиного виконуваного файлу. Цей процес включає в себе об'єднання власних машинних кодів, даних та різних сегментів, що використовуються у програмі.

Отже, Оскільки MobiVM працює вже з згенерованими .class файлами, немає потреби реалізовувати JEP 395 у самому компілятору Java, та необхідно розпізнати ці структури та відповідно згенерувати машинний код.

В той же час, компілятор Java виконує багато наступних операцій з автоматичного генерування коду, який відповідає основним характеристикам записів:

- Генерація Конструктора. Компілятор автоматично генерує конструктор для класу Record, який приймає аргументи для всіх полів класу. Цей конструктор ініціалізує всі поля об'єкта.
- Генерація Методів equals() і hashCode():Метод equals() порівнює об'єкти за значенням їхніх полів, а метод hashCode() генерує хеш-код на основі

значень цих полів. Компілятор автоматично генерує код цих методів, що дозволяє коректно використовувати об'єкти Record в колекціях.

- Генерація Методу toString(): Метод toString() створює рядкове представлення об'єкта. Компілятор генерує код для цього методу так, щоб виводити значення всіх полів об'єкта, що робить відладку зручною.
- Автоматичне Додавання Аннотації @Override: Для всіх автоматично згенерованих методів, таких як equals(), hashCode(), і toString(), компілятор автоматично додає аннотацію @Override для підказки компілятору, що ці методи перевизначають методи з класу java.lang.Object.

Тобто, під час компіляції класу Java Record компілятор вирішує багато рутинних завдань, що дозволяє розробникам ефективно використовувати записи для представлення даних в Java, спрощуючи при цьому багато стандартних операцій, які раніше потребували б великої кількості коду.

2.2 Огляд MobiVM як транслятора

Оскільки Java компілятори зі складу відповідних Java Development Kit[] підтримують Java Records з версії JDK16 і ці структури формуються у проміжних файлах байт-коду, MobiVM це та ланка, яка має розпізнати, підтримати ці структури і сформувати відповідний бінарний код, який можна виконати на платформі Apple iOS.

Байт-код (Java Bytecode) - це проміжний код, який генерується під час компіляції програм на Java та використовується для виконання на віртуальній машині Java (JVM).

Байт-код є незалежним від конкретної архітектури процесора чи операційної системи. Він створюється під час компіляції і може виконуватися на будь-якому пристрої, який має встановлену віртуальну машину Java.

Байт-код складається з імперативних інструкцій, які виконують конкретні операції, такі як арифметичні обчислення, керування потоком виконання, робота з пам'яттю тощо.

Віртуальна машина Java використовує стек для виконання операцій. Байт-код взаємодіє з вершинами стеку, що дозволяє ефективно виконувати різні операції.

Байт-код Java підтримує об'єктно-орієнтовані концепції, такі як створення об'єктів, виклик методів, робота з успадкуванням, інтерфейсами та іншими аспектами ООП.

Інструкції байт-коду дозволяють реалізувати структури керування потоком виконання, такі як умовні переходи, цикли та виклики методів.

Віртуальна машина Java відповідає за автоматичне управління пам'яттю, включаючи видалення непотрібних об'єктів збирачем сміття.

Байт-код Java є ключовим елементом концепції "Write Once, Run Anywhere" (пиши один раз, виконуй де завгодно), що робить його основним етапом для переносимості та виконання Java-програм.

Розглянемо більш-детально конвеєр операції в MobiVM, з допомогою яких, виконуються операції над вхідним файлом байт-коду. Схема операцій показана на рис. 2.2

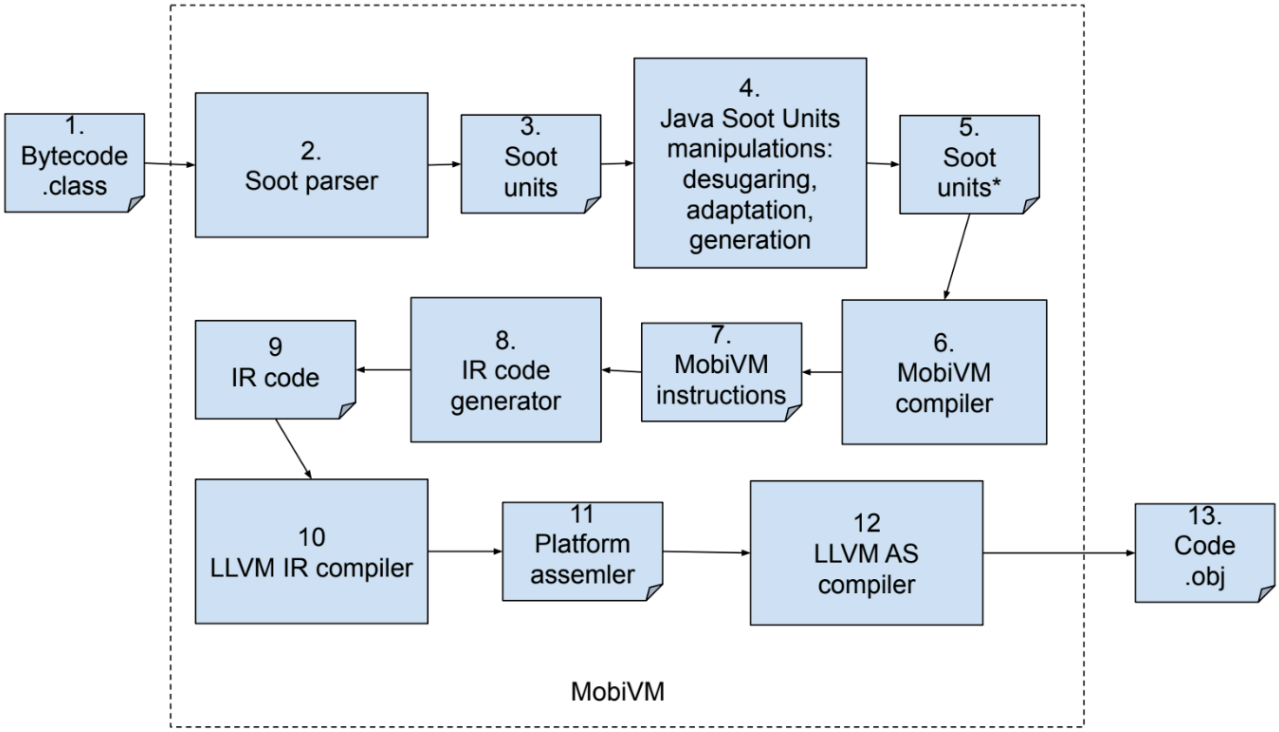

Рисунок 2.2 – Конвеєр операцій транслятора MobiVM

Схема описує шлях, який проходить байт-код, необхідний для отримання платформо-залежного машинного коду, який можна виконувати на платформі Apple iOS. Він складається с наступного:

1. Вхідні дані – Java байт-код. У вигляді .class файлу, як отримали від Java компілятору.
2. Soot parser[26]. Це спеціалізоване програмне забезпечення у вигляді бібліотеки, яку використовує MobiVM для зчитування, розбору .class файлу і представлення його у вигляді об’єктів.
3. Проміжні дані – Soot Units. Це результат роботи Soot бібліотеки. Представляють собою послідовність байт-код інструкцій, прочитаних і розібраних з .class файлу, які далі можуть бути проаналізовані і оброблені.
4. Етап маніпуляції з байт-код інструкціями. Це важливий етап, на якому Java байт-код інструкції можуть бути мутовані, видаленні, або навіть цілі методи чи класи синтезовані. Цей етап відбувається для необхідності реалізації функціоналу що не підтримується, через наявний. Такий процес називається «знецукрення» (code desugaring).
5. Проміжні дані – Soot Units*. Отримані в результаті маніпуляцій. Це все ті ж інструкції в представлені бібліотеки Soot, але вони вже не містять інструкцій, що не підтримуються і є повністю сумісними з MobiVM.
6. MobiVM компілятор – етап трансляції інструкцій байт-коду в внутрішнє представлення MobiVM. На цьому етапі також виконується проміжна оптимізація.
7. Проміжні дані – інструкції в представлені MobiVM. Це послідовності валідних операцій, з яких можна сформувати код в потрібній мові програмування (транслювати).
8. Генератор IR[27] коду – трансляція внутрішнього коду в початковий код LLVM Intermediate Representation. IR це платформо-незалежний, асемблеро-подібний проміжний код, який можна скомпілювати в асемблерний код цільової платформи.
9. Проміжні дані - IR код.

10. LLVM IR компілятор – виконує компіляцію IR коду в платформо-залежний початковий асеблерний код. В даному випадку ми отримуємо представлення на мові асеблеру для платформи Apple iOS.
11. Проміжні дані – асемблерний код.
12. LLVM AS компілятор[13] – компілює початковий код на мові асеблеру для платформи Apple iOS в платформо-залежний машинний код у вигляді об’єктного файлу.
13. Вихідні дані – об’єктний файл машинного коду для платформи Apple iOS що відповідає початковому байт-коду.

Окремо можна спинитись на описі проміжного коду LLVM (LLVM Intermediate Representation або LLVM IR), який є ключовим етапом у компіляційному процесі і дозволяє відділити етап генерації машинного коду від етапу власне компіляції для конкретної архітектури процесора. Його основними перевагами є:

- Універсальність та Нейтральність: LLVM IR є абстрактним та нейтральним проміжним представленням, яке не залежить від конкретної архітектури процесора чи мови програмування. Це дозволяє використовувати його для різноманітних мов програмування та цільових архітектур.
- Легкість Оптимізацій: LLVM IR є високорівневим, але в той же час низькорівневим представленням, що дозволяє проводити оптимізації на великому наборі різноманітних програм. Це полегшує створення потужних оптимізаторів та трансляторів.
- Структурованість: LLVM IR має структурований формат, що спрощує читання та розуміння коду. Він представляє програму у вигляді інструкцій, які використовують імперативний стиль, а також використовують орієнтований граф зв'язків.
- Модульна Система: Програми у LLVM IR представляють собою модулі, які містять функції, змінні та інші елементи. Це дозволяє використовувати проміжний код для великих та складних програм, розбитих на модулі.

- Безпека типів даних: LLVM IR використовує статичну типізацію, що дозволяє забезпечити безпеку типів та оптимізації на етапі компіляції. Це сприяє виявленню помилок ще до виконання програми.
- Операції та Інструкції: Інструкції LLVM IR виконують широкий спектр операцій, від арифметичних операцій до роботи з пам'яттю та управління потоком виконання.
- Можливості Оптимізацій та Інструменти: У LLVM IR вбудована розширена система оптимізацій, що дозволяє покращувати ефективність програм на етапі компіляції. Також існують інструменти для аналізу та відлагодження коду.

В цілому, LLVM IR є потужним та гнучким інструментом у світі компіляції, що дозволяє вирішувати різні завдання, починаючи від генерації машинного коду до використання високорівневих мов програмування.

Отже, проаналізувавши приведену вище схему, можна зробити висновок, щодо місця, в якому необхідно додати зміни для підтримки JEP 395. Це етап номер 4 – етап маніпуляції з Soot інструкція. Саме на цьому етапі будуть виявлені класи, що є записами і для них будуть синтезовані необхідні методи, що вже підтримуються MobiVM і можуть бути скомпільовані в машинний код, який можна виконати на платформі Apple iOS.

2.3 Технічне порівняння Java записів та класичних класів

Java Records були введені для спрощення розробки простих класів для зберігання даних в мові програмування Java.

Основна мета полягала в тому, щоб автоматизувати генерацію загальних методів, таких як equals(), hashCode(), toString(), а також конструкторів та методів-аксесорів для полів класу. Це значно зменшило обсяг рутинної роботи, яку розробники повинні були виконувати для створення простих об'єктів-записів.

Розглянемо приклад класу Account, призначений для зберігання двох полів даних:

Name – ім’я користувача

Id – його ідентифікатор

Використовуючи JEP 395 такий клас для зберігання даних буде виглядати як показано на рис.2.3.

```
3    public record Account(String name, int id) {
4
5    }
```

Рисунок 2.3 – клас Account використовуючи Java Records

У відповідності до опису JEP395 за нас компілятор синтезує наступне:

- Фінальні поля name та id;
- Ініціалізуючий конструтор;
- Аксесори до полів name та id;
- Метод equals();
- Метод hashCode();
- Метод toString().

Для порівняння реалізує цей самий клас Account використовуючи класичний підхід, у результаті він буде виглядати як показано на рис.2.4.

Цей приклад демонструє наскільки використання Java Records зменшує кількість коду, необхідну для реалізації.

Відповідно зі зменшенням об’єму коду також зменшується вірогідність помилки.

```java
public class Account {
    6 usages
    private final String name;
    5 usages
    private final int id;

    48 usages
    public Account(String name, int id) {
        this.name = name;
        this.id = id;
    }

    3 usages
    @Override
    public String toString() {
        return "Account{" +
                "name='" + name + '\'' +
                ", id=" + id +
                '}';
    }

    no usages
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Account account = (Account) o;
        if (id != account.id) return false;
        return Objects.equals(name, account.name);
    }

    1 usage
    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + id;
        return result;
    }
}
```

Рисунок 2.4. Класична реалізація класу Account

2.4 Розробка методу та алгоритмів реалізації JEP 395

Java компілятор відповідає за логічну та структурну перевірку на етапі компіляції тож отриманий .class файл має однозначно вірну структуру якій можна сліпо довіряти.

У відповідності до JEP 395 необхідно підтримати наступні зміни з точки зору .class файлу та кодогенерації:

додано базовий клас java.lang.Record;

авто генеровані методи equals(), hashCode(), toString()

Оскільки Record з точки зору JVM являє собою Java клас з особливостями описаними вище, то технічно має транслюватись наявною інфраструктурою, але це не можливо з наступних причин:

- відсутність базового класу java.lang.Record в Runtime Library;
- методи equals(), hashCode(), toString() геренуються через Lambda Factory, підтримка останніх досить лімітована.

2.5 Розробка методу реалізації базового класу та методу реалізації відсутніх методів

Відповідно до JEP 395, java.lang.Record є базовим абстрактним класом та частиною Java Runtime. Він не має реалізації і є декларативним, і має бути доданий до класів MobiVM і містити наступне:

заявлений як abstract class java.lang.Record;

мати заявлений метод public abstract boolean equals(Object obj);

мати заявлений метод public abstract int hashCode();

мати заявлений метод public abstract String toString();

Найбільшу складність представляють собою методи equals(), hashCode(), toString() оскільки вони не реалізуються на етапі компіляцію та представлені у вигляді викликів DynamicInvoke та мають бути реалізовані динамічно з допомого фабрики лямбд в конкретній реалізації віртуальної машини.

DynamicInvoke в Java є потужним механізмом, який дозволяє динамічно викликати методи під час виконання програми. Ця можливість забезпечується інструкцією invokedynamic, яка була введена в Java 7, як частина JSR 292 і використовується головним чином для підтримки динамічних мов програмування та складних сценаріїв в байткоді Java. [28].

У основі DynamicInvoke лежить інструкція invokedynamic. На відміну від інших інструкцій виклику в байт-коді Java, invokedynamic є більш гнучким та адаптивним. Вона не має фіксованого зв’язування під час компіляції; натомість

вона спирається на метод запуску (bootstrap method), щоб надавати інформацію про зв’язування під час виконання.

Метод запуску є важливою складовою invokedynamic. Це метод, який викликається один раз під час першого виклику інструкції invokedynamic. Його завдання - створювати CallSite, який інкапсулює інформацію про зв’язування для динамічного виклику.

У багатьох випадках invokedynamic використовується разом із LambdaMetafactory з пакету java.lang.invoke. LambdaMetafactory відповідає за генерацію екземплярів функціональних інтерфейсів динамічно. Це особливо корисно для сценаріїв, де вирази лямбда або посилання на метод повинні створюватися під час виконання.

DynamicInvoke дозволяє розробникам створювати та зв’язувати виклики методів динамічно. Це відхід від традиційного статичного зв’язування, яке відбувається під час компіляції. З invokedynamic зв’язування визначається під час виконання, що забезпечує більшу гнучкість та адаптивність.

Нажаль, фабрика лямбд не підтримується в MobiVM тож метод реалізації цих методів буде статичний, на етапі трансляції Byte коду з .class файлу у .llvm код.

Для цього необхідно слідувати наступній послідовності логічних дій:

- виявити що маємо справу з Record – проаналізувавши базовий клас, має відповідати java.lang.Record.
- для всіх методів: equals(), hashCode(), toString() необхідно визначити список полів структури;
- для методу equals() – виконати порівняння всіх полів структури з полями кандидату:
  - Згенерований метод equals() перевіряє рівність, порівнюючи кожне поле запису.
  - Генерується логічний вираз рівності, який перевіряє, чи є кожне поле поточного запису рівним відповідному полю іншого об'єкта.
- для методу hashCode() – сформувати композитний код, що включає hashCode() кожного з поля стуктури:

- Згенерований метод hashCode() використовує хеш-коди кожного поля для обчислення хеш-коду всього запису.
- Використовується комбінація хеш-кодів полів для забезпечення унікальності.

– для методу toString() – сформувати строкове представлення, що має складатися з: імені структури та переліку пар назви та значення полів:

- Згенерований метод toString() включає ім'я класу, а також назви та значення полів.
- Використовується форматування, яке дозволяє зручно представити дані об'єкта у вигляді рядка.

Об’єднавши наведену вище інформацію, отримуємо блок-схему алгоритму обробки Java записів, наведену на рис. 2.5.

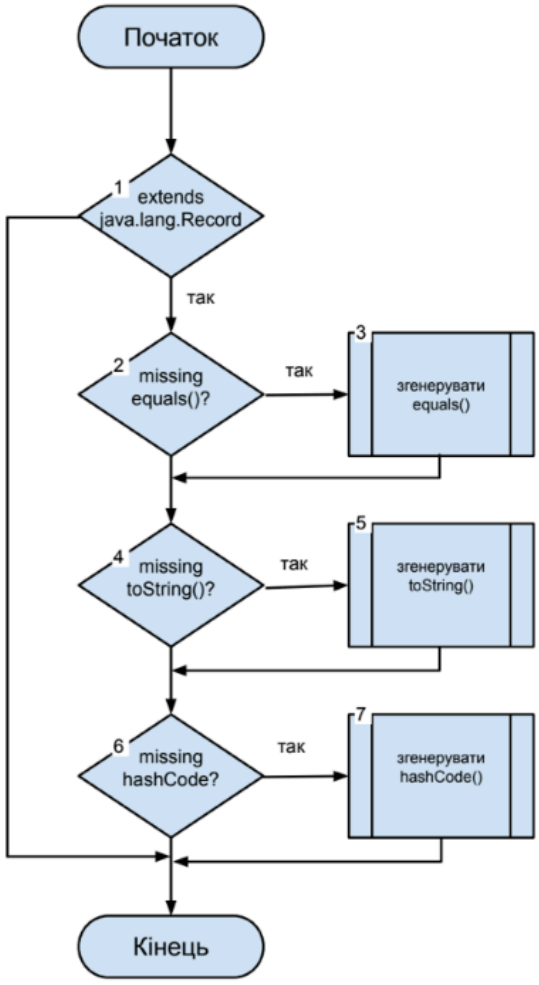

Рисунок 2.5 – блок схема алгоритму виявлення класу Java записів

2.6 Висновки

У другому розділі було розглянуто технічні аспекти реалізації записів Java як того вимагає JEP 395: Records.

Проведено порівняння реалізації дата класів використовую класичні класи та класи записи, відзначивши очевидні переваги останніх.

Проаналізовано принципи роботи InvokeDynamic у контексті реалізації даного проекту. У зв’язку з існуючими обмеженнями MobiVM було визначено неможливим використовувати підхід описаний в JEP395, то ж алгоритм реалізації буде побудовано на статичному синтезуванні методів equals(), toString(), hashCode() під час трансляції Java байт-коду у машинний код платформи Apple iOS.

Розроблені алгоритми реалізації відсутнього функціоналу для підтримки JEP 395: Records в інструменті MobiVM, для компіляції Java коду під платформу Apple iOS.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Варіантний аналіз і обґрунтування вибору засобів для реалізації програмного засобу

MobiVM реалізований мовою програмування Java, тож для розширення його функціональних можливо використовувати тільки її.

Java – об’єктно-орієнтована мова програмування, одна з найпоширеніших мов програмування в світі. Основною особливістю є те, що її програмний код спочатку транслюється в спеціальний байт-код, незалежний від платформи, а згодом байт-код виконується віртуальною машиною JVM (Java Virtual Machine). Це дозволяє код написаний на Java запускати майже на всіх відомих системах: Windows, Mac, Android тощо.

Крім мови програмування важливу роль відіграє вибір інтегрального середовища для розробки.

Для розробки Java існує багато IDE, які мають різні переваги та недоліки. Найпопулярніші IDE для Java є:

- IntelliJ IDEA: Це потужне та гнучке IDE, яке підтримує Java, Kotlin та інші мови JVM. Воно має багато функцій, які полегшують написання та аналіз коду, таких як підказки, автодоповнення, рефакторинг, налагодження, профілювання, інтеграція з Git та іншими системами контролю версій, плагіни та хмарні сервіси. IntelliJ IDEA є лідером серед IDE для Java та Kotlin[29].

Рисунок 3.1 – стартовий екран Intellij Idea

• Eclipse: Це відкрите та безкоштовне IDE, яке також популярне серед розробників Java. Воно базується на плагінах, які дозволяють розширювати функціональність IDE. Воно також має вбудований компілятор Java, інструменти для моделювання, графіки, тестування та звітності[30].

Рисунок 3.2 – стартовий екран Eclipse

• NetBeans: Це ще одне відкрите та безкоштовне IDE для розробки Java. Воно має простий та інтуїтивний інтерфейс користувача, швидку настановлення проекту та гарну підтримку стандартних бібліотек Java. Воно також позволяє розробляти веб-додатки з використанням Jakarta EE (раніше Java EE), Spring Boot, Hibernate та інших фреймворків[31].

Рисунок 3.3 - стартовий екран Apache Netbeans

Для розробки MobiVM рекомендовано використовувати IntelliJ IDEA, тож порівняймо IntelliJ IDEA з іншими інтегрованими середовищами розробки та доведемо переваги IntelliJ IDEA:

1.IntelliJ IDEA vs. Eclipse:

- Аналіз коду: IntelliJ IDEA надає більше інструментів для аналізу та вдосконалення коду та виявлення помилок.
- Підтримка мов: IntelliJ IDEA добре підтримує мови, такі як Kotlin та Scala.
- Автодоповнення коду: IntelliJ IDEA має вище якість та інтелектуальну систему автодоповнення.
- Підтримка плагінів: IntelliJ IDEA відомий своєю багатою екосистемою плагінів для підтримки різних мов та інструментів розробки.

2. IntelliJ IDEA vs. NetBeans:

- Висока продуктивність: IntelliJ IDEA відомий своєю швидкістю та реагує відмінно при роботі над великими проектами.
- Підтримка найсвіжіших версій Java: IntelliJ IDEA надає актуальну підтримку для нових версій Java, включаючи JEP 395 (Records).
- Багатофункціональність та плагіни: IntelliJ IDEA пропонує розширений набір функціональності та можливість легко встановлювати плагіни для розширення функціональних можливостей.
- Аналіз та рефакторинг коду: Інтелектуальні інструменти аналізу та рефакторингу коду в IntelliJ IDEA допомагають підтримувати високу якість коду та робити його оптимізацію.

Загалом, IntelliJ IDEA виділяється завдяки своїй високій продуктивності, широкому спектру інтелектуальних інструментів, які полегшують розробку, і актуальної підтримці нових технологій та версій Java. Ці переваги роблять IntelliJ IDEA однією з найкращих IDE для розробки на Java та реалізації JEP 395 (Records) для MobiVM.

3.2 Розробка базового класу та bootstrap методу для JEP 395

У відповідності до вимог JEP 395, java.lang.Record є декларативним базовим класом і його реалізація відповідає декларації, та виглядає як показано на рис.3.1:

```
/** This is the common base class of all Java language record classes. ...*/
public abstract class Record {
    Constructor for record classes to call.
    protected Record() {}

    /** Indicates whether some other object is "equal to" this one.  In addition ...*/
    @Override
    public abstract boolean equals(Object obj);

    /** Returns a hash code value for the record. ...*/
    @Override
    public abstract int hashCode();

    /** Returns a string representation of the record. ...*/
    @Override
    public abstract String toString();
}
```

Рис. 3.1 – декларативний клас java.lang.Record

Методи equals(), hashCode(), toString() у випадку Java Records не геренуються під час компіляції, а очікуються бути синтезованими під час виконання програми, оскільки викликаються через інструкцію DynamicInvoke. Ця інструкція звертається до bootstrap методу.

У контексті dynamicInvoke і віртуальної машини Java (JVM) bootstrap-метод відіграє важливу роль в реалізації invokedynamic, яка є байткод-інструкцією, введеною в Java 7 для підтримки динамічних мов та покращення продуктивності в деяких сценаріях використання.

Bootstrap-метод — це метод, який надає користувач або середовище виконання мови, і відповідає за створення об'єкта CallSite під час процесу зв’язування інструкції invokedynamic. Bootstrap-метод вказується при створенні інструкції invokedynamic і викликається в JVM під час виконання для отримання початкового об'єкта CallSite.

Інструкція invokedynamic є більш гнучким та динамічним способом виклику методу порівняно з традиційними інструкціями, такими як invokevirtual чи invokestatic. Замість прямого вказівки цільового методу, invokedynamic покладається на механізм зв’язування, який дозволяє динамічним мовам чи середовищам виконання визначити поведінку виклику динамічно.

Під час процесу зв’язування bootstrap-метод викликається для отримання об'єкта CallSite, який, фактично є змінною-контейнером для об'єкта методу (method handle), який представляє цільовий метод.

Bootstrap-метод відповідає за визначення об'єкта методу, який буде пов'язаний з CallSite. Це може залежати від різних факторів, таких як середовище виконання, семантика динамічної мови чи інші розглядувані умови виконання.

Після створення CallSite і пов'язання його з об'єктом методу, подальші виклики інструкції invokedynamic будуть використовувати метод з CallSite для динамічного виклику методу.

Інструкція invokedynamic надає гнучкість, оскільки метод, який слід викликати, визначається під час виконання, що дозволяє динамічно адаптуватися до різних умов виконання.

Динамічний характер invokedynamic дозволяє краще оптимізувати виклик методу, відкладаючи визначення методу до часу виконання. Це особливо корисно для динамічних мов чи сценаріїв, де метод, який слід викликати, не відомий на етапі компіляції.

Розглянемо, на прикладі наступного класу (рис. 3.2), які invokedynamic виклики генерує Java компілятор для 395:

```
3    public record Account(String name, int id) {
4
5    }
```

Рисунок 3.2 – клас Account для аналізу invokedynamic

У відповідності до JEP 395 використовуються наступні виклики invokedynamic:

- Для методу **toString**():

  InvokeDynamic з параметрами:

- name: toString
- type: (LAccount;)Ljava/lang/String;
- bootstrap: java/lang/runtime/ObjectMethods.bootstrap, з параметрами:
    - class: Account
    - fields: String = “name;id”
    - getters: [
        - REF_getField Account.name:Ljava/lang/String;
        - REF_getField Account.id:I

    ]

- Для методу **hashCode**():

  InvokeDynamic з параметрами:

  - name: hashCode
  - type: (LAccount;)I
  - bootstrap: java/lang/runtime/ObjectMethods.bootstrap, з параметрами:
      - class: Account
      - fields: String = “name;id”
      - getters: [
          - REF_getField Account.name:Ljava/lang/String;
          - REF_getField Account.id:I

      ]

- Для методу **equals()**:

  InvokeDynamic з параметрами:

  - name: equals
  - type: (LAccount;Ljava/lang/Object;)Z
  - bootstrap: java/lang/runtime/ObjectMethods.bootstrap, з параметрами:
      - class: Account
      - fields: String = “name;id”
      - getters: [
          - REF_getField Account.name:Ljava/lang/String;
          - REF_getField Account.id:I

]

В усіх випадках використовується єдиний bootstrap метод java/lang/runtime/ObjectMethods.bootstrap який і необхідно реалізувати.

В залежності від параметру name метод повертає різні CallSite для реалізації виклику, відповідно реалізація матиме вигляд, як на рис.3.3:

```
public class ObjectMethods {
    no usages
    public static Object bootstrap(MethodHandles.Lookup lookup, String methodName, TypeDescriptor type,
                                   Class<?> recordClass,
                                   String names,
                                   MethodHandle... getters) throws Throwable {
        MethodType methodType = (MethodType) type;
        List<MethodHandle> getterList = List.of(getters);
        MethodHandle handle = switch (methodName) {
            case "equals"   -> {
                if (methodType != null && !methodType.equals(MethodType.methodType(boolean.class, recordClass, Object.class)))
                    throw new IllegalArgumentException("Bad method type: " + methodType);
                yield makeEquals(recordClass, getterList);
            }
            case "hashCode" -> {
                if (methodType != null && !methodType.equals(MethodType.methodType(int.class, recordClass)))
                    throw new IllegalArgumentException("Bad method type: " + methodType);
                yield makeHashCode(recordClass, getterList);
            }
            case "toString" -> {
                if (methodType != null && !methodType.equals(MethodType.methodType(String.class, recordClass)))
                    throw new IllegalArgumentException("Bad method type: " + methodType);
                List<String> nameList = "".equals(names) ? List.of() : List.of(names.split( regex: ";"));
                if (nameList.size() != getterList.size())
                    throw new IllegalArgumentException("Name list and accessor list do not match");
                yield makeToString(lookup, recordClass, getters, nameList);
            }
            default -> throw new IllegalArgumentException(methodName);
        };
        return methodType != null ? new ConstantCallSite(handle) : handle;
    }
```

Рисунок 3.3 – імплементація ObjectMethods.bootstrap

3.3 Розробка загального коду для equals()

Метод equals() у Java є важливим для порівняння об'єктів на рівність. Він є частиною концепції порівняння об'єктів за значенням, а не за посиланням. Він дозволяє розробникам визначити, яким чином об'єкти порівнюються за їхніми значеннями, а не за адресами пам'яті. Це особливо важливо для класів, які представляють дані (в нашому випадку – Java Records). Метод equals() використовується для визначення, чи є два об'єкти рівними за значенням. Це важливо для визначення унікальності об'єктів в масивах чи колекціях.

Коли розробники створюють власні класи, вони можуть визначити власний спосіб порівняння об'єктів за значенням, реалізовуючи метод equals(). Це дозволяє визначити, коли два об'єкти є ідентичними за певними критеріями. equals() зазвичай використовується разом із методом hashCode(). Коли два об'єкти рівні за значенням, їх хеш-коди також повинні бути однаковими. Це допомагає коректно використовувати об'єкти в колекціях, які базуються на хеш-коді.

У багатьох випадках правильна імплементація методу equals() є важливою для коректної роботи класу в різних сценаріях, таких як порівняння об'єктів у логіці додатка чи в обчисленнях.

Реалізований BootStrap для методу equals(), повертає CallSite, що реалізовано в методі makeEquals. Робота цього методу полягає в формуванні послідовності викликів до обробників методів які:

- Порівняють вказівники об’єкті і якщо вони однакові, поверне true;
- Обробник що перевірить аргумент на відповідність до заданого класу запису і якщо класи різні, поверне false оскільки не має більше сенсу виконувати перевірку;
- Пройде по списку всіх полів запису і порівняє відповідні поля двох об’єктів;

Реалізація функціоналу в цілому відповідає такій, яку реалізують для звичайних класів, які мають коректно обробляти виклик equals

3.4 Розробка методу hashCode()

Метод hashCode() в Java має важливе значення для хеш-заснованих структур даних і алгоритмів, таких як хеш-таблиці та хеш-мапи. Його основне значення полягає в тому, що він надає числове представлення об'єкта, яке використовується цими структурами даних для ефективного зберігання та вилучення ключ-значення. Наприклад структури даних, таких як HashMap,

HashSet та інші в пакеті java.util, використовують хеш-код об'єктів для їх розподілу між відсічками або слотами. Цей розподіл дозволяє швидше вилучення та вставку - метод hashCode() використовується для обчислення індексу чи хеш-коду, який визначає місце, де повинен бути розміщений об'єкт.

Метод hashCode(), разом із методом equals(), використовується для порівняння об'єктів за значенням. Коли об'єкти порівнюються за рівністю, їх хеш-коди повинні бути однаковими. Це дозволяє ефективно використовувати об'єкти в колекціях, таких як HashSet чи HashMap.

Також, розробники можуть імплементувати власний метод hashCode() для своїх класів, щоб визначити специфічний спосіб обчислення хеш-коду. Це особливо важливо для класів, які використовуються в хеш-заснованих колекціях.

Гарний хеш-код повинен забезпечувати рівномірний розподіл об'єктів в структурі даних. Це допомагає уникнути конфліктів хеш-кодів, які можуть призвести до поганої продуктивності.

Реалізований BootStrap для методу hashCode(), повертає CallSite, що реалізовано в методі makeHashCode. Робота цього методу полягає в формуванні послідовності викликів до обробників методів які:

Імплементація цього фунціоналу виглядає як:

- крокуючи по всім полям зі списку;
- отримуємо значення поля для об’єкту;
- оновлюємо hashCode за формулою:
  hashCode = hashCode * 31 + field[i].hashCode
- по закінченню циклу повертаємо розраховане значення.

Реалізація функціоналу в цілому відповідає такій, яку реалізують для звичайних класів, які мають коректно обробляти виклик equals() та hashCode(). Реалізація приведене на рис.3.4.

```
54    private static int hashCombiner(int x, int y) {
55        return x*31 + y;
56    }
57
      1 usage
58    private static MethodHandle hasher(Class<?> clazz) {
59        return (clazz.isPrimitive()
60                ? primitiveHashers.get(clazz)
61                : OBJECTS_HASHCODE.asType(MethodType.methodType(int.class, clazz)));
62    }
63
      1 usage
64    private static MethodHandle makeHashCode(Class<?> receiverClass,
65                                             List<MethodHandle> getters) {
66        MethodHandle accumulator = MethodHandles.dropArguments(ZERO, pos: 0, receiverClass); // (R)I
67
68        for (MethodHandle getter : getters) {
69            MethodHandle hasher = hasher(getter.type().returnType()); // (T)I
70            MethodHandle hashThisField = MethodHandles.filterArguments(hasher, pos: 0, getter);    // (R)I
71            MethodHandle combineHashes = MethodHandles.filterArguments(HASH_COMBINER, pos: 0, accumulator, hashThisField); // (RR)I
72            accumulator = MethodHandles.permuteArguments(combineHashes, accumulator.type(), ...reorder: 0, 0); // adapt (R)I to (RR)I
73        }
74
75        return accumulator;
76    }
```

Рисунок 3.4 – реалізація makeHashCode

Повна реалізація цього функціоналу приведена у додатку В, метод makeHashCode().

3.5 Розробка методу toString()

Метод toString() в Java відіграє важливу роль у контексті записів (Java Records), надаючи зручний та інформативний спосіб представлення об'єктів. Він автоматично генерується для записів та надає зручне та стандартизоване представлення їхніх полів. Це спрощує відлагоджування і дозволяє розробникам швидко переглядати значення об'єкта. Автоматично згенерований метод toString() дозволяє уникнути написання однотипного коду для виводу значень полів.

Завдяки методу toString(), значення об’єктів можна легко виводити на консоль або в журналах логування.

Хоча метод toString() автоматично генерується, розробники можуть впливати на його вигляд, перевизначаючи його у власному коді. Це надає можливість налаштування формату виводу відповідно до потреб додатку.

Враховуючи ці аспекти, метод toString() в записах спрощує вивід та представлення об'єктів, забезпечуючи розробникам потужний інструмент для відладки і аналізу програм.

Реалізований BootStrap для методу toString(), повертає CallSite, що реалізовано в методі makeToString. Робота цього методу полягає в формуванні послідовності викликів до обробників, робота яких полягає формуванні строкового представлення об’єкта з значення полів записів, їх назви та сам об’єкт. Список полів передається у вигляді строки, де назви полів розділені крапкою з комою. Тож, для отримання списку необхідно спочатку розділи цю строку по символу розмежування(крапка з комою).

Далі необхідно сформувати строкове представлення структури у вигляді:

{RecordName}=[{fieldA}=a, {fieldB}=b, …]

Для цього імплементуєм функціонал наступними кроками:

- для накопичення даних в стрічці будем використовувати StringBuilder;
- задаєм його початкове значення з назви структури та відкриваючої квадратної скобки ({RecordName}=[)
- крокуючи по всім полям зі списку;
- отримуємо значення поля для об’єкту;
- формуємо строкову пару name=value та додаємо до результуючої строки;
- по закінченню закриваємо квадратну скобку (])
- повертаємо строкове представлення об’єкта

Реалізація функціоналу в цілому відповідає такій, яку реалізують для звичайних класів, які мають коректно обробляти виклик equals() та hashCode().

Повна реалізація цього функціоналу приведена у додатку В, метод makeToString().

Роботу методу toString() можна спостерігати в вікні виводу логів, як показано на рис.3.5.

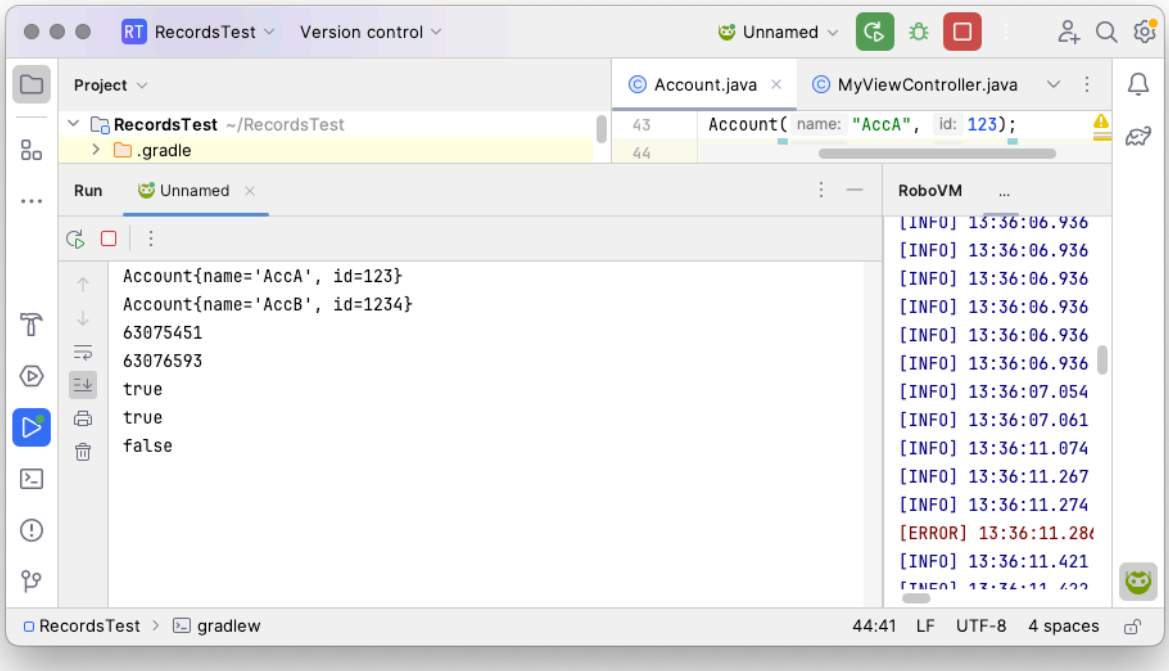

Рисунок 3.5 – робота методу toString()

3.5 Розробка доповнень до Java.lang.Class

У відповідності до JEP 395 java.lang.Class має бути розширений наступними методами, які стануть частиною Reflection API:

- boolean isRecord() — Повертає true, якщо заданий клас був визначений як запис (схоже до isEnum для енамів);
- RecordComponent[] getRecordComponents() — Повертає масив об'єктів java.lang.reflect.RecordComponent. Елементи цього масиву відповідають компонентам запису в тому ж порядку, в якому вони вказані у визначенні запису. З кожного елемента масиву можна отримати додаткову інформацію, таку як його ім'я, анотації та метод доступу.

Реалізація цих змін показана на рис.3.6.

```
9   public class Class {
        1 usage
10      public boolean isRecord() {
11          // this superclass and final modifier check is not strictly necessary
12          // they are intrinsified and serve as a fast-path check
13          return getSuperclass() == java.lang.Record.class &&
14                  (this.getModifiers() & Modifier.FINAL) != 0 ;
15      }
16
        no usages
17      public RecordComponent[] getRecordComponents() {
18          if (!isRecord()) {
19              return null;
20          }
21          return getRecordComponents0();
22      }
        1 usage
23      private native RecordComponent[] getRecordComponents0();
```

Рисунок 3.6 – зміни до java.lang.Class

3.6 Висновки

У третьому розділі було обґрунтовано вибір мови програмування та технологій для розробки програмного продукту. Реалізовано базовий клас для записів. Розроблено метод Bootstrap, який необхідний для заміни динамічного виклику методів toString(), hashCode(), equals() статичними реалізаціями на етапі компіляції.

Реалізація bootstrap методу показує на своєму прикладі базовий підхід у реалізації динамічного зв’язування коду при використовувані інструкції DynamicInvoke. Адже саме цей метод має створити послідовність CallSite що буде виконуватись за заданого виклику. Також це місце де може відбуватись кешування вже створених послідовностей і подальше повернення раніше створених, замість створення нових.

Кожний з приведений вище методів: toString(), hashCode(), equals() був розглянутий окремо і реалізований у відповідних генераторах обробників методів: makeToString , makeHashCode, makeEquals.

Крім явних нововведень змін зазнало також Reflection API. Клас java.lang.Class був розширений методами isRecord та getRecordComponents.

Перший метод дозволяє визначити чи клас є записом, а другий отримати інформацію по всім полям запису, у порядку, як вони були вказані.

РОЗДІЛ 4
ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Аналіз методів тестування програмного забезпечення

Тестування програмного забезпечення — процес перевірки відповідності заявлених до продукту вимог і реально реалізованої функціональності, здійснюваний шляхом спостереження за його роботою в штучно створених ситуаціях і на обмеженому наборі тестів, обраних певним чином [32].

Методи тестування включають процес пошуку помилок або інших дефектів і тестування програмних компонентів для їх оцінки. Може оцінюватись: – відповідність вимогам дизайнерів та розробників; – правильна відповідь на всі можливі вхідні дані; – виконання функцій у розумні терміни; – практичність; – сумісність із програмним забезпеченням та операційними системами; – відповідність завданням замовника.

Сьогодні тестування ПЗ є одним із найдорожчих етапів життєвого циклу ПЗ [34], на нього припадає від 50% до 65% загальних витрат. Існує кілька основних методів тестування, які є частиною режиму тестування програмного забезпечення. Ці тести, як правило, вважаються самостійними у пошуку помилок у всій системі.

Тестування методом «чорної скриньки» здійснюється без будь-яких знань внутрішньої роботи системи. Тестер симулюватиме дії користувача програмного середовища, надаючи різні входи і тестуючи згенеровані виходи. Цей тест також відомий як Black-box, closed-box тестування або функціональне тестування. Основні особливості тестування методом «чорної скриньки»:

– Тестування, як функціональне, так і нефункціональне, що не передбачає знання внутрішнього пристрою компонента або системи. 62 – Тест-дизайн, заснований на методі «чорної скриньки» – процедура написання або вибору тест-кейсів на основі аналізу функціональної або нефункціональної специфікації компонента або системи без знання її внутрішнього пристрою. Мета цього методу це пошук помилок у таких категоріях: – неправильно реалізовані або відсутні функції; – помилки інтерфейсу; – помилки у структурах даних чи

організації доступу до зовнішніх баз даних; – помилки поведінки або недостатня продуктивність системи. Переваги: – тестування проводиться з позиції кінцевого користувача і може допомогти виявити неточності та протиріччя у специфікації; – тестувальнику не потрібно знати мови програмування та заглиблюватися особливо реалізації програми; – тестування може проводитись фахівцями, незалежними від відділу розробки, що допомагає уникнути упередженого ставлення; – можна починати писати тест-кейси, як тільки готова специфікація.

Недоліки: – тестується лише дуже обмежена кількість шляхів виконання програми; – без чіткої специфікації досить складно скласти ефективні тест-кейси; – деякі тести можуть бути надмірними, якщо вони вже були проведені розробником на рівні модульного тестування.

Таким чином, ми не маємо уявлення про структуру та внутрішній устрій системи. Потрібно зосереджуватися на тому, що програма робить, а не на тому, як вона це робить.

Протилежністю техніки «чорної скриньки» є тестування методом «білої скриньки». Тестування методом «білої скриньки» враховує внутрішнє функціонування і логіку роботи коду. Для виконання цього тесту, тестувальник 63 повинен мати досвід програміста, щоб дізнатися точну частину коду, в якій виникають ті чи інші помилки.

Основні особливості тестування методом «білої скриньки»: – тестування, засноване на аналізі внутрішньої структури компонента або системи; – тест-дизайн, що базується на методі «білої скриньки» – процедура написання або вибору тест-кейсів на основі аналізу внутрішнього пристрою системи або компонента. Переваги: – тестування може проводитися на ранніх етапах: немає необхідності чекати створення інтерфейсу користувача; – можна провести ретельніше тестування, з покриттям великої кількості шляхів виконання програми. Недоліки: – для виконання тестування білої скриньки потрібна велика кількість спеціальних знань; – при використанні автоматизації тестування на цьому рівні, підтримка тестових скриптів може бути досить накладною, якщо програма часто змінюється.

4.1 Системні вимоги

Для роботи продукту MobiVM з реалізаціє підтримки Java Records(JEP 395) необхідно задовільнити наступні системні вимоги.

Вимоги до робочої станції, на якій буде запускатись MobiVM компілятор:

– персональний комп’ютер типу Apple Mac, підтримуються архітектури як x86_64 так і Arm64 m1/2/3;

– Об’єм оперативної пам’яті 8gb;

– 4GB вільного дискового простору;

– MacOSx13 та новіша;

– XCode 14 та новіше;

– Intellij Idea 2022.1 та новіше

Також існують вимоги до цільового апаратного забезпечення, на якому буде виконуватись скомпільована програма, це:

– iOS версії 8 та новіше;

– підтримуються як thumb32 так і arm64 архітектури;

– 512 MB оперативної пам’яті для запуску та роботи;

– 256 MB дискового простору для встановлення додатку;

4.2 Тестування підтримки JEP 395: Records

Для тестування напишемо простий юніт-тест(unit-test), що перевірить можливість компіляції, та вірність роботи реалізованих методів.

Модульне тестування, або юніт-тестування - процес в програмуванні, що дозволяє перевірити на коректність окремі модулі вихідного коду програми.

Ідея полягає в тому, щоб писати тести для кожної найбільш нетривіальноїфункції або методу. Це дозволяє досить швидко перевірити, чи не призвело чергову зміну коду до регресії, тобто до появи помилок у вже відтестованих місцях програми, а також полегшує виявлення та усунення таких помилок.

Коли говорять про якусь користь, більшість людей вважає що є якась метрика яка показує що без unit тестів було ось так, а з ними стало по іншому. Але таку метрику не так просто знайти чи внести на проект. З одного боку ми прагнемо підняти якість коду, а з іншого – unit тести покривають дуже малу і специфічну, ізольовану частину коду. Якщо говорити про ООП, то тести будуть перевіряти правильність роботи методів і класів, їхній зв'язок. Одними unit тестами ми не гарантуємо якість коду, але вони зменшать кількість дефектів спровокованих банальними помилками ізольованими в одному модулі. Це можна рахувати однією з метрик для вимірювання користі unit тестів.

Іншою метрикою можна назвати можливість легко змінювати систему.

Якщо код повністю непокритий unit тестами – то недоцільно вносити будь-які зміни в систему, і тому набагато простіше скопіювати кусок коду і помістити поряд таку ж реалізацію. Якщо ж у вас є тести, то ви можете більш-менш безболісно змінити код і перевірити чи система продовжує працювати коректно.

Якщо є велика система, як перевірити що вона працює правильно і так як задумано? Для цього її потрібно протестувати. Запускають весь цикл регресивного тестування залучивши до цього QA. Після внесення змін не можна бути впевненим, що система все так же якісно працює, тому треба знову все тестувати. Чим більше регресивного і ручного тестування – тим дорожчою являється розробка, а програміст пізніше отримує інформацію про зміни в системі.

Unit тести – це самий простий і близький до програміста спосіб сказати що система все ще якісна після того, як були внесені зміни [32]. Розробка системи без unit тестів буде швидша лише до певного етапу складності даної системи. Після досягнення цієї межі – кожне внесення зміни буде коштувати дорожче по часу, просто тому що невідомо на скільки якісно працює система, відсутній інструмент який може це сказати. І тому потрібно буде збільшувати кількість регресивно і ручного тестування. Побачити важливість тестів можна на тенденції створення нових мов і фреймворків, які зразу розробляються з вбудованими інструментами для unit тестів. Якщо спочатку сісти й написати всю систему, а потім уже до неї писати тести – то так, час розробки збільшиться. При розробці

системи вже її тестували: виводили в логи, проводили пошук багів. На цю роботу уже було потрачено багато часу. Якщо ж робити все по TDD, то спочатку ви пишуть тест, визначають мету, а потім уже починають це реалізовувати. Виходить наступне:

1. фінальна реалізація буде набагато менша. У вас не включиться “режим архітектора” і не будете робити такі речі як реалізація на всі випадки життя. TDD каже, щоб ви робили тільки те що потрібно в цій задачі і нічого більше.

2. економиться час видумуванням способів як провалідувати код, не треба запускати всю системи доки не буде повністю завершена задача.

3. велику частину коду можна згенерувати з тесту. Сучасні IDE дуже розвинені. Не потрібно про це забувати. Наприклад в Java або .NET можна згенерувати частину коду по певному сценарію. Це економить багато часу.

Unit тести зменшують кількість дефектів від 40% до 90%, хоч при цьому і збільшується початковий час розробки. Потрібно навчитись писати тести, писати код так щоб його можна було тестувати. На початковому етапі часу буде тратитись на 15-30% більше, але і через пів року можна буде вносити зміни так же легко. Це довготермінові інвестиції. Вони прості й зрозумілі.

Характеристика Unit тестів. Вони повинні забезпечувати Quality gate – захищати від допущених помилок, якщо ви пішли й змінили щось в чому не дуже розібралися. Для цього вони повинні бути ізольованими. Найменування тестів. Вони повинні казати що "впали" та чому. Тести потрібно ділити на маленькі ізольовані частини. Unit тест який не можна відрізнити хто з команди написав. Повинна бути консистентність. Уникати ситуацій коли назва тесту написана не інформативно і не описує його роботу. Для цього потрібно залучити Code review і статичний аналіз коду.

Тести потрібно перевіряти так само як і основний код. Unit тести повинні бути без side ефектів. Вони повинні бути незалежні і запускатися окремо.

Читабельність. Коли є вхідні параметри, виконусь ось це і результат такий. Це створює автодокументацію коду і допомагає робити code reveiw більше ефективним. Review юніт тестів разом з кодом допомагає зрозуміти які use кейси

були покриті, чи все реалізовано. Чисті тести. Все з основ чистого коду. Повинен читатися легко, навіть якщо тестує складний код.

Тести повинні бути швидкими. Вони не повинні підключатися до бази даних, щось отримувати з мережі чи читати з файлової системи. Навіть на дуже великому проекті всі тести повинні проходити не довше декількох хвилин. Тести повинні перевіряти ЩО робить код а не ЯК він це робить. Щоб була можливість код змінити або порефакторити.

Unit тести повинні бути саме Unit тестами а не інтеграційними, наприклад.

Тулзи дозволять писати інтеграційні тести, які ми називаємо Unit тестами, а потім жаліємось чому вони такі нестабільні.

Намагатися зберігати пірамуду тестування. Unit тестів повинно бути багато, а тестів вищого рівня менше. Часто на проектах пишуть багато інтеграційних тестів і мало unit – це антипатерн, перевернута піраміда.

Дана метрика лише допомагає і ніколи не повинна визначати. Це не повинна бути метрика менеджера чи користувача. Дана метрика повинна допомагали лише розробнику зрозуміти працює він добре чи погано. Вона ніколи не визначає чи тести написані добре, навіть якщо покриття 100%. Code coverage – це інструмент розробника для розробки, а не менеджера для оцінки чого небудь.

Закінчивши задачу, розробник повинен протестуват code coverage і той повинен показати йому сценарії які він не врахував.

Цифра покриття, по суті, нічого не значить. Значення має звіт, який ми читаємо і розуміємо. Він показує куски коду які не покриті. Можна побачити що якийсь кусок коду критичний і потребую тесту, інший ж не критичний і не потребує. Покриття повинне показувати код який ще не покрили. Відсоток покриття може становити і 90%, але є 10% критичного коду який не покритий. В цьому випадку використання code coverage неправильне.

Юніт тестування не являється чимось особливим, чимось стороннім, додатковим. Під час розробки ми постійно тестуємо код, тому що так побудований спосіб мислення і процес розробки. При цьому ми постійно придумуємо test cases, аналізуємо як код виконує одну чи іншу задачу і постійно

їх виконуємо порівнюючи результат з тим що ми повинні отримати. Практично, різниця між автоматичним юніт тестом і тим тестуванням яке ми робимо під час розробки в своїй голові полягає лише в тому що ми його кудись записуємо – unit test як код.

Немає ніякого іншого додаткового процесу написання unit test, він завжди являється невід'ємною частиною розробки. [32]. Під "юніт" розуміються невеликі блоки коду, наприклад, окремі методи класу.

Тобто. можна протестувати спосіб працездатність в автоматичному режимі. Коли програма досить велика, що містить багато класів, методів і тим більше якщо планується подальше його розширення, варто зайнятися тестуванням і в цьому допоможе Junit5. Він є окремою бібліотекою класів, які потрібно підключити при створенні свого тесту.

Для вирішення нашого завдання було вирішено написати декілька unit-test для перевірки коректності роботи критично важливих етапів алгоритму.

Юніт-тестування — це метод тестування програмного коду на рівні окремих компонентів або модулів з метою перевірки правильності їхньої роботи та відповідності специфікаціям. Основна ідея юніт-тестів полягає в тому, щоб тестувати найменші незалежні частини програми (юніти) для підтвердження їх очікуваної поведінки.

Головною метою юніт-тестування є валідація коректності роботи окремих частин програми. Це дозволяє виявляти та виправляти помилки на ранніх стадіях розробки.

Юніт-тести зазвичай автоматизовані, тобто вони можуть бути виконані автоматично без необхідності вручного втручання розробника. Це полегшує і швидше виявлення помилок.

Кожен юніт-тест перевіряє конкретний аспект поведінки частини програми. Він містить конкретні очікування результатів та дій.

Юніт-тести повинні бути незалежними один від одного. Це означає, що результат одного тесту не повинен впливати на виконання іншого.

Оскільки юніт-тести перевіряють окремі частини коду, вони зазвичай виконуються дуже швидко. Це важливо для того, щоб розробники могли швидко перевіряти свою роботу.

Юніт-тести допомагають виявляти помилки на ранніх етапах розробки, що полегшує їх виправлення перед тим, як вони стануть більш серйозними проблемами.

Юніт-тести дозволяють розробникам вносити зміни в код (рефакторинг) з впевненістю, що ці зміни не порушать існуючу функціональність.

Юніт-тести легко інтегруються з процесами Continuous Integration (CI) та Continuous Deployment (CD), дозволяючи автоматично виконувати тести при кожній зміні коду.

Юніт-тестування відіграє ключову роль у розробці програмного забезпечення, сприяючи створенню надійного та функціонального коду.

Код, що виконує юніт-тестування правильності реалізації записів Java у відповідност до JEP 395 показано на рис.4.1:

```
1  public record Account(String name, int id) {
2      public static void main(String[] args) {
3          var a = new Account( name: "AccA",  id: 123);
4          var b = new Account( name: "AccB",  id: 1234);
5          // testing to String
6          System.out.println(a);
7          System.out.println(b);
8          // testing hashCode()
9          System.out.println(a.hashCode());
10         System.out.println(b.hashCode());
11         // testing equals()
12         System.out.println(a.equals(a));
13         System.out.println(b.equals(b));
14         System.out.println(a.equals(b));
15
16     }
17 }
```

Рис. 4.1 Код для тестування реалізації JEP 395

Код успішно скомпільований та виконаний, у результаті отримано наступний вивід, рис. 4.2:

```
Account[name=AccA, id=123]
Account[name=AccB, id=1234]
63075451
63076593
true
true
false
```

Рис. 4.2 Вивід коду для тестування JEP 395

З виводу видно наступне:

1. toString() формує очікувані строки для об’єктів A та B, строкове представлення містять назву класу записи, його поля з відповідним значенням кожного. Кожна стрічка відповідає конкретному екземпляру запису:
    a. Account[name=AccA, id=123]
    b. Account[name=AccB, id=1234]

2. hashCode() формує числове значення, при чому, як і очікувано, вони різні для об’єктів A та B:
    a. 63075451
    b. 63076593

3. equals() вірно порівнює об’єкти A та B. Повертає позитивний результат при однакових об’єктах та негативний при різних:
    a. true (a = a)
    b. true (b = b)
    c. false (a != b)

4.3 Тестування на симуляторі Apple iOS

Для підтвердження валідності виконання коду на платформі Apple iOS виконаємо тест використовуючи симулятор пристрою Apple iPhone 12 mini, що працює під операційною системою Apple iOS17, має 64 бітний Arm процесор.

Для цього використаєм інтеграцію MobiVM з Intellij Idea – яка дозволяє використовувати це інтегроване середовище для розробки коду, компіляції Java коду та трансляції Java байт-коду в код платформи Apple iOS використовуючи MobiVM.

Як виглядає інтегроване середовище, з відкритим Java проектом та активною інтеграцією MobiVM показано на рис.4.1.

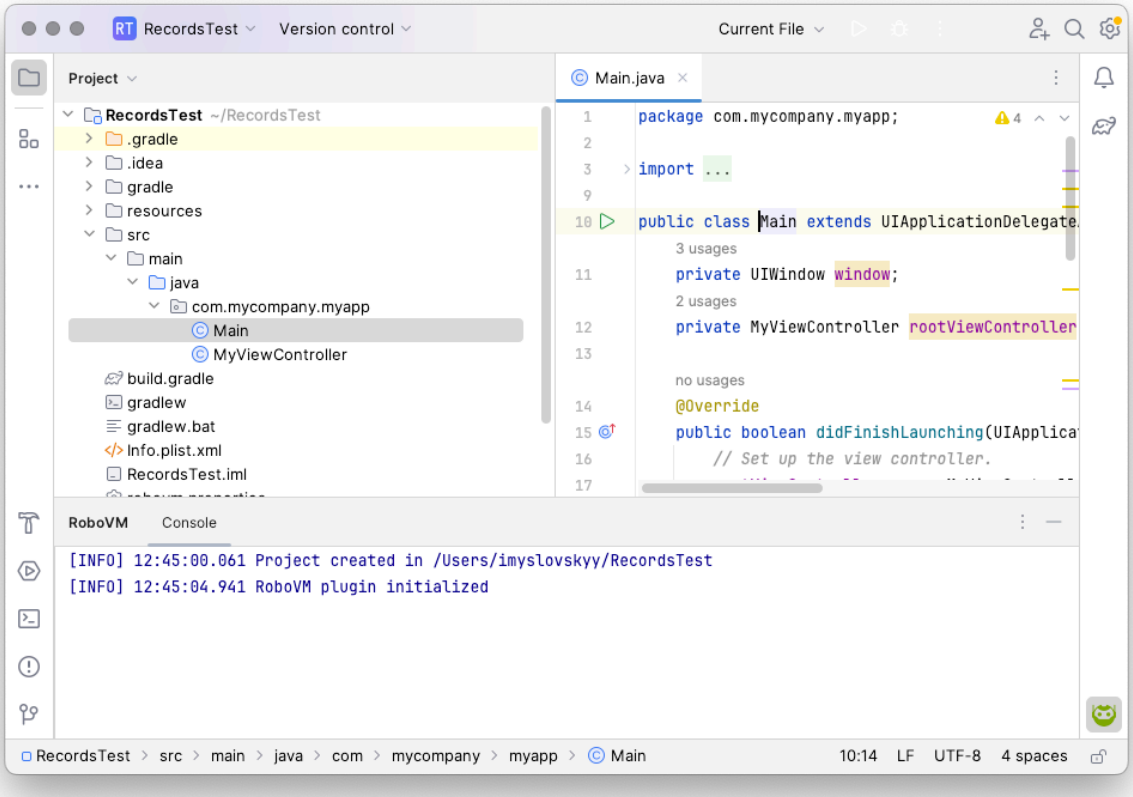

Рисунок 4.1 – Intellij Idea з MobiVM інтеграцією

Першим кроком буде створення проекту. Для цього використаємо шаблон «RoboVM iOS App with story board» як показано на рис.4.2.

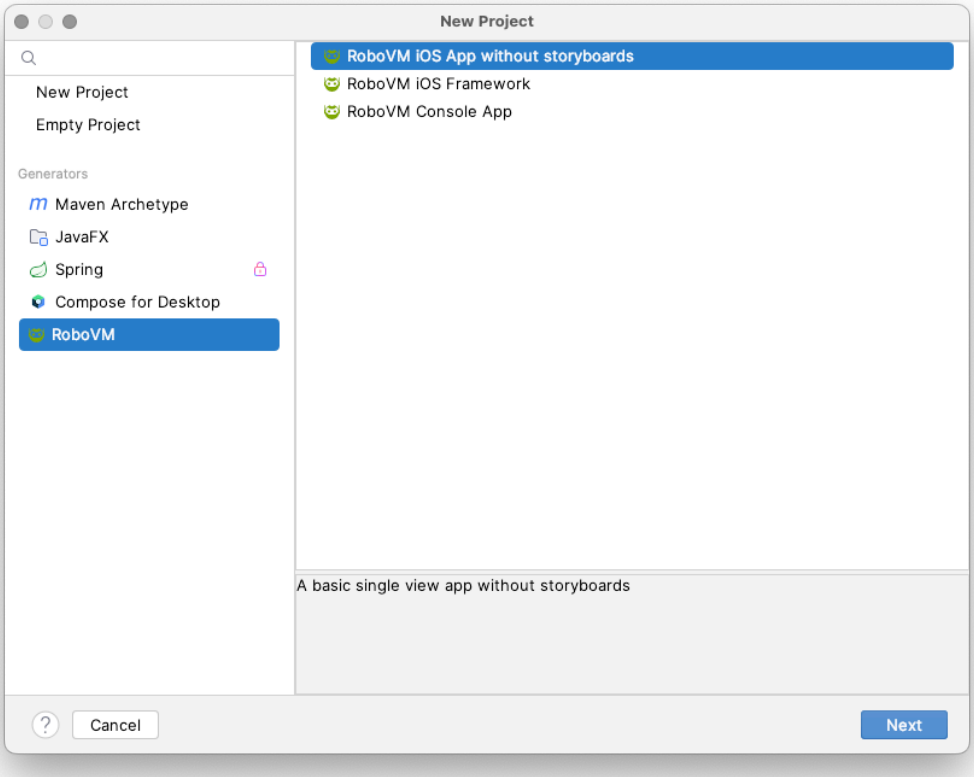

Рисунок 4.2 – діалог створення нового MobiVM проекту

Наступний крок – це вибір системи збірки проекту та налаштування параметрів, таких як ім’я програми, ідентифікатори і т.п. В якості системи збірки будемо використовувати Gradle[34]. Цей крок показано на рис.4.3.

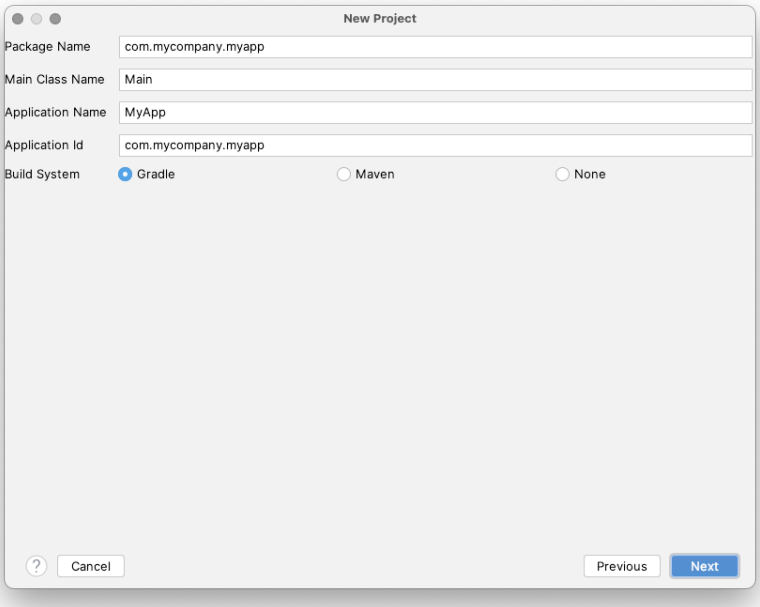

Рисунок 4.3 – налаштування новоствореного проекту

В новостворений проект необхідно додати record клас для тесту як показано на рис.4.4. Код цього класу було скопійовано з попереднього тесту. Та на відміну від останнього, в цей раз тест буде виконуватись на цільовій системі Apple iOS в середовищі симулятора пристрою Apple iPhone Mini 12.

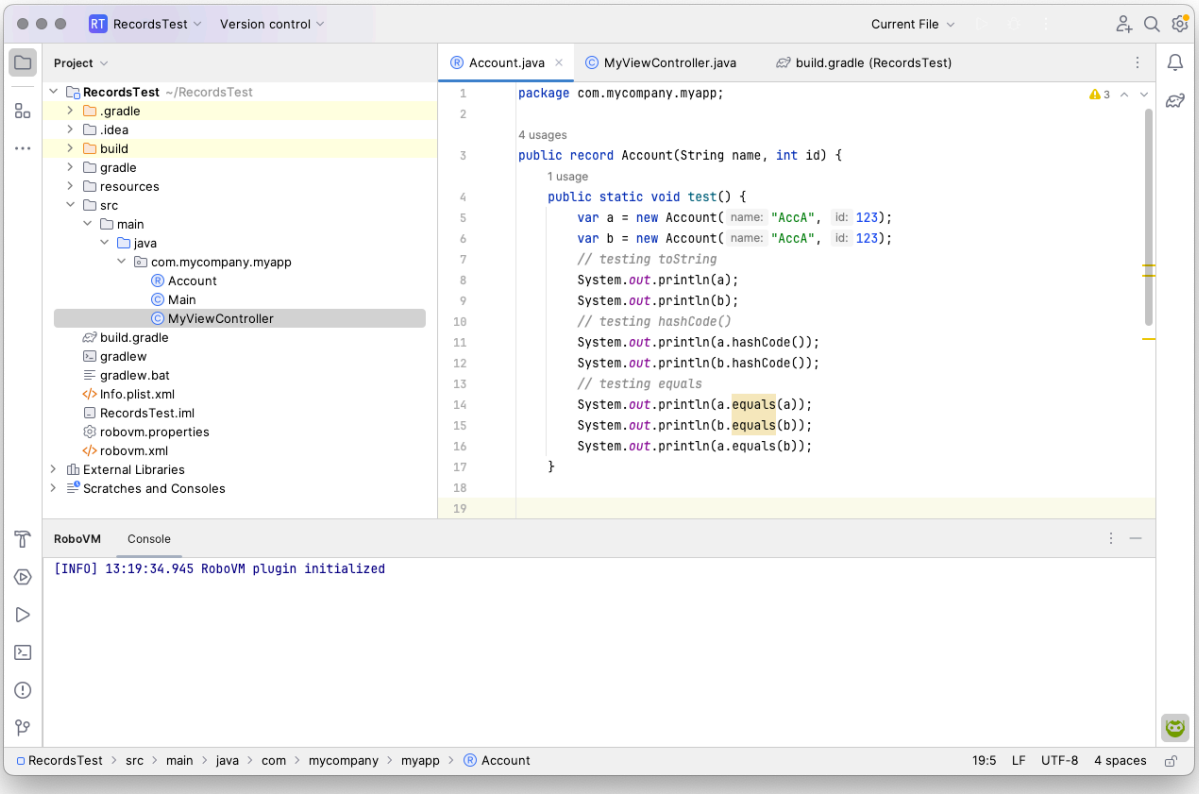

Рисунок 4.4 – Intellij Idea з доданим тестовим класом

Для компіляції проекту під цільову систему необхідно створити конфігурацію для запуску, використавши менеджер конфігурацій:

- меню Run;
- підменю Edit Configuration;
- кнопка Add New Configuration.

Та вказавши наступні параметри:

- тип конфігурації: RoboVM iOS;
- вибрати модуль RecordsTest;
- вибрати Simulator в панелі вибору цілі для запуску;

- та вибрати симулятор iPhone Mini 12;

Вибрані параметри повинні відображатись як у діалогу конфігурації показано на Рис.4.5. Також діалог не повинен містити попереджень про помилки. У разі наявності останніх їх потрібно виправити.

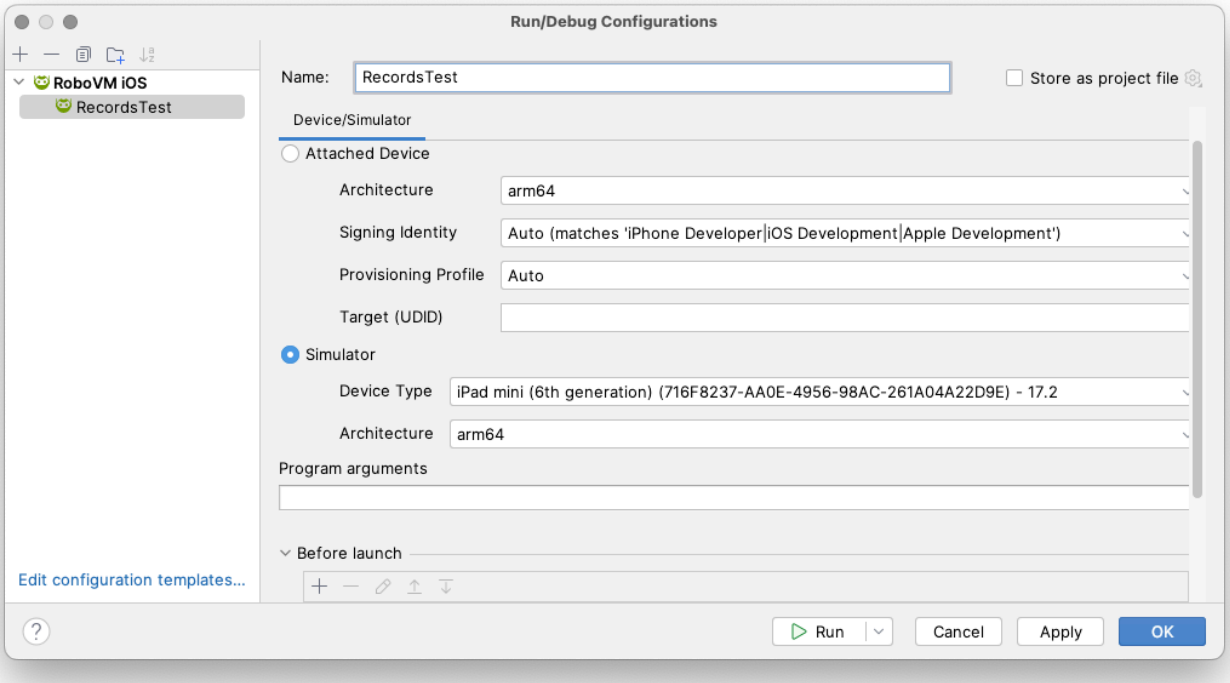

Рисунок 4.5 – діалог налаштування конфігурації для запуску

Виконавши запуск програми, через створену конфігурацію, ініціюємо компіляцію Java програми та подальшу трансляцію Java байт-коду в нативний код для запуску на платформі Apple iOS, процес компіляції можна спостерігати в панелі RoboVM, як показано на рис.4.6.

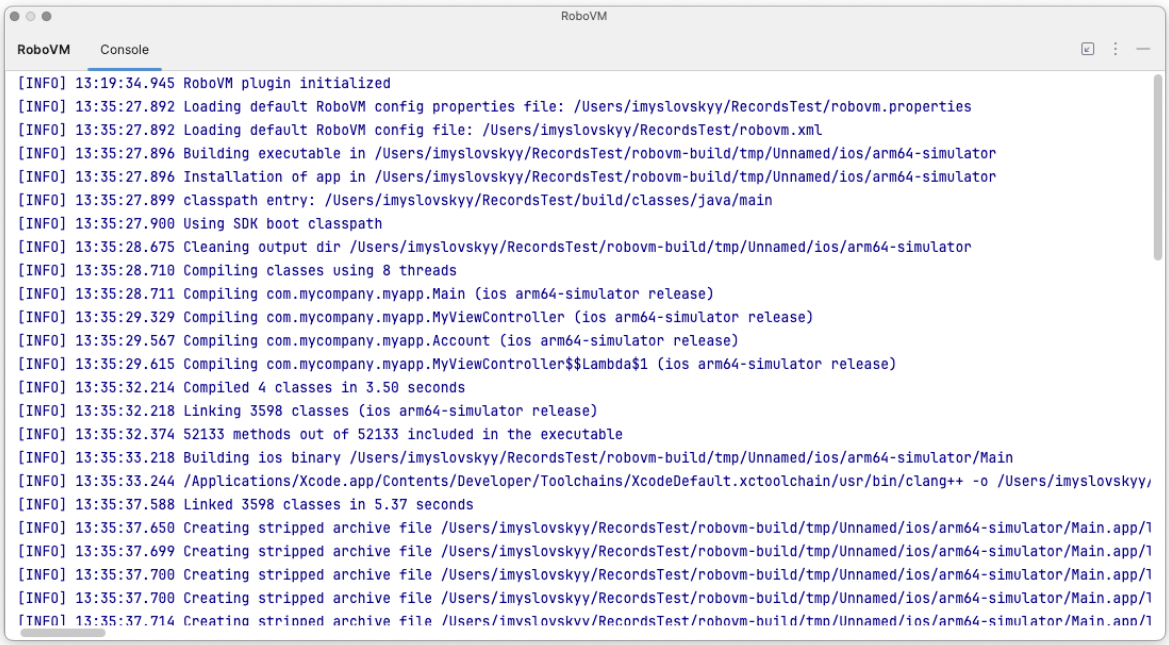

Рисунок 4.6 – панель процесу компіляції

Запуск відбувається на симуляторі, що показую правильно сформований бінарний код під платформу Apple iOS, рис.4.7.

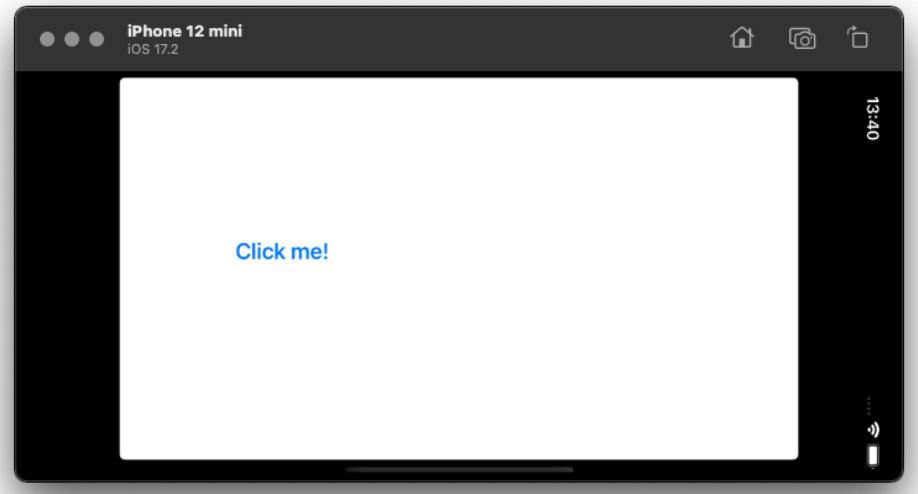

Рисунок 4.7– тестова програма запущена у симуляторі iPhone12 mini

Натиснувши кнопку «Click me!» ініціюємо виконання тестового коду для перевірки роботи доданої підтримки Java Records у відповідності до JEP 395. Результати спостерігаємо у вікні Run/Console як показано на рис.4.8.

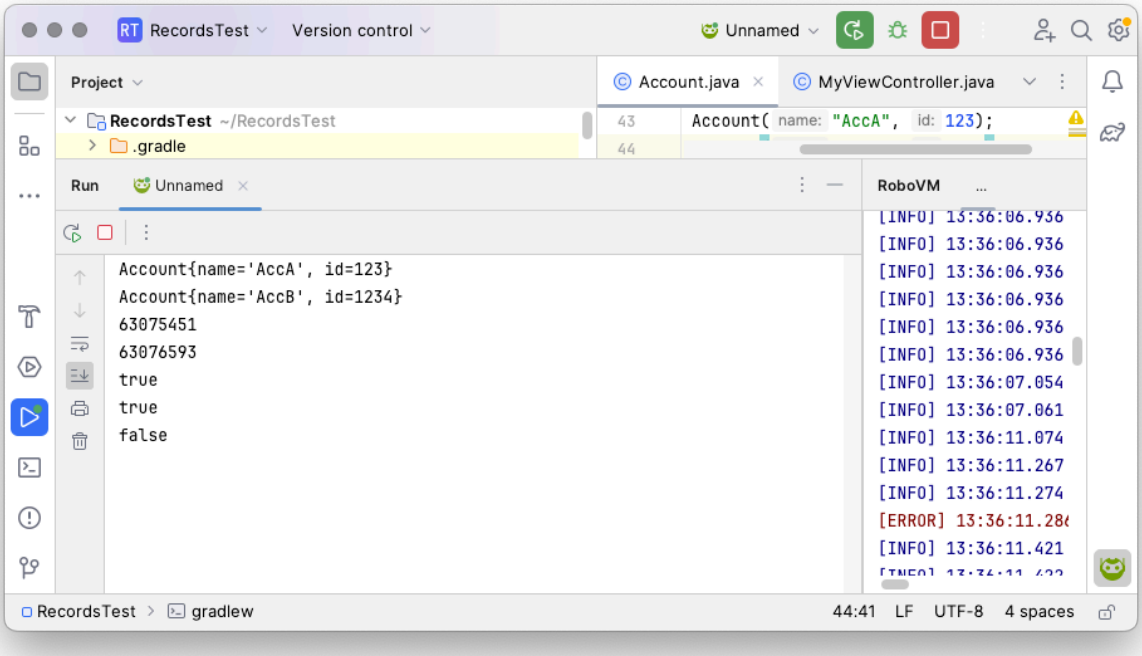

Рисунок 4.8 – результати виконання тесту

Як видно, отримані результати підтверджують коректність виконання програми та відповідність реалізації до JEP-395.

4.4 iOS Симулятор як достовірний тест

Тест виконувався на iOS симуляторі, що працює в операційній системі MacOSx на апаратному забезпечені.

Та результати отримані на симуляторі еквівалентні запуску на реальному пристрої, оскільки, тест, що запускався у симуляторі iOS, поводиться аналогічно до роботи на реальному пристрої. Симулятор дозволяє розробникам відтворити середовище iOS на комп'ютері для тестування та відлагодження додатків.

Симулятор використовує ту ж версію операційної системи iOS, яка існує для реального пристрою. Це забезпечує консистентність у роботі додатку між симулятором і реальним пристроєм.

Симулятор дозволяє встановлювати різні мовні налаштування. Це важливо для перевірки та підтримки різних мов у вашому додатку.

Симулятор використовує ту ж архітектуру процесора, що і реальні пристрої (наприклад, ARM для сучасних iPhone і iPad або Apple Silicon для нових моделей Mac).

Симулятор передає функціонал сенсорів, таких як тачскрін, акселерометр, гіроскоп та інші, щоб додаток міг реагувати на взаємодію користувача, аналогічно реальному пристрою.

Віртуальний Ідентифікатор Програмного Забезпечення (UDID). Кожен симулятор має унікальний віртуальний ідентифікатор програмного забезпечення (UDID), що дозволяє точно ідентифікувати його як окремий пристрій.

При тестуванні у симуляторі важливо враховувати, що певні аспекти, такі як робота з мережею чи камерою, можуть відрізнятися від реального пристрою.

4.5 Аналіз проміжних даних

Також цікавим є аналіз проміжних даних, які демонструють еволюцію Java байт-коду на шляху до машинного коду цільової платформи, оскільки ілюструють повний шлях перетворень інструкцій в різні форми. Розглянемо, як змінювався конструктор:

- початковий код на Java:

```
public record Account(String name, int id) {
}
```

- байт-коду:

```
public Account(java.lang.String, int);
    descriptor: (Ljava/lang/String;I)V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=2, locals=3, args_size=3
         0: aload_0
         1: invokespecial #1                  // Method
java/lang/Record."<init>":()V
         4: aload_0
         5: aload_1
         6: putfield      #7                  // Field
name:Ljava/lang/String;
         9: aload_0
        10: iload_2
        11: putfield      #13                 // Field id:I
        14: return
```

```
      LineNumberTable:
        line 1: 0
    MethodParameters:
      Name                           Flags
      name
      id
```

- проміжний LLVM IR:

```
define weak void
@"[J]com.mycompany.myapp.Account.<init>(Ljava/lang/String;I)V"(%Env* %p0,
%Object* %p1, %Object* %p2, i32 %p3) nounwind noinline optsize {
label0:
    %r0 = alloca %Object*
    %r1 = alloca %Object*
    %i0 = alloca i32
    call void @"checkso"()
    store %Object* %p1, %Object** %r0, !dbg !{i32 37, i32 0, !7, null}
    store %Object* %p2, %Object** %r1, !dbg !{i32 37, i32 0, !7, null}
    store i32 %p3, i32* %i0, !dbg !{i32 37, i32 0, !7, null}
    %t0 = load %Object** %r0, !dbg !{i32 37, i32 0, !7, null}
    call void
@"[j]java.lang.Object.<init>()V[Invokespecial(com/mycompany/myapp/Account,
com/mycompany/myapp/Account)]"(%Env* %p0, %Object* %t0), !dbg !{i32 37,
i32 0, !7, null}
    %t1 = load %Object** %r1, !dbg !{i32 38, i32 0, !7, null}
    %t2 = load %Object** %r0, !dbg !{i32 38, i32 0, !7, null}
    call void
@"[j]com.mycompany.myapp.Account.name(Ljava/lang/String;)[set]"(%Env* %p0,
%Object* %t2, %Object* %t1), !dbg !{i32 38, i32 0, !7, null}
    %t3 = load i32* %i0, !dbg !{i32 39, i32 0, !7, null}
    %t4 = load %Object** %r0, !dbg !{i32 39, i32 0, !7, null}
    call void @"[j]com.mycompany.myapp.Account.id(I)[set]"(%Env* %p0,
%Object* %t4, i32 %t3), !dbg !{i32 39, i32 0, !7, null}
    ret void, !dbg !{i32 40, i32 0, !7, null}
}
```

- проміжний LLVM AS:

```
    .section    __TEXT,__textcoal_nt,coalesced,pure_instructions
    .globl "_[J]com.mycompany.myapp.Account.<init>(Ljava/lang/String;I)V"
    .weak_definition
"_[J]com.mycompany.myapp.Account.<init>(Ljava/lang/String;I)V"
    .align 2
"_[J]com.mycompany.myapp.Account.<init>(Ljava/lang/String;I)V": ;
@"[J]com.mycompany.myapp.Account.<init>(Ljava/lang/String;I)V"
Lfunc_begin4:
    .file  1 "Account.java"
    .loc   1 37 0                  ; Account.java:37:0
    .cfi_startproc
; BB#0:                                 ; %label0
    stp    x20, x19, [sp, #-32]!
```

```
    stp    x29, x30, [sp, #16]
    add    x29, sp, #16            ; =16
    sub    sp, sp, #16             ; =16
Ltmp10:
    .cfi_def_cfa w29, 16
Ltmp11:
    .cfi_offset w30, -8
Ltmp12:
    .cfi_offset w29, -16
Ltmp13:
    .cfi_offset w19, -24
Ltmp14:
    .cfi_offset w20, -32
    mov     x19, x2
    ; InlineAsm Start
    sub    x9, sp, #16, lsl #12    ; =65536
    ldr     x9, [x9]
    ; InlineAsm End
    .loc   1 37 0 prologue_end     ; Account.java:37:0
Ltmp15:
    str    x1, [sp, #8]
    str    w3, [sp, #4]
    bl "_[J]java.lang.Object.<init>()V"
    .loc   1 38 0                  ; Account.java:38:0
    ldr    x8, [sp, #8]
    str    x19, [x8, #16]
    dmb    ish
    .loc   1 39 0                  ; Account.java:39:0
    ldr    w8, [sp, #4]
    ldr    x9, [sp, #8]
    str    w8, [x9, #24]
    dmb    ish
    .loc   1 40 0                  ; Account.java:40:0
    sub    sp, x29, #16            ; =16
    ldp    x29, x30, [sp, #16]
    ldp    x20, x19, [sp], #32
    ret
Ltmp16:
Lfunc_end4:
"L_[J]com.mycompany.myapp.Account.<init>(Ljava/lang/String;I)V_end":
```

Лістинги цих проміжних файлів наведено в додатку Г.

4.6 Висновки

У четвертому розділі було проведено тестування підтримки JEP 395 в MobiVM. Було створено простий тест та виконано перевірку методів toString(), hashCode(), equals().

Додатково було створено проект в середовищі Intellij Idea та простестовано запуск програми на цільовий системі – Apple iOS, яке виконувалось на симуляторі пристрою Apple iPhone12 mini. Код компілювався для процесора з ядром Arm64.

Робота методів відповідає специфікації наведеній в JEP 395 як у простому тесті так і у випадку виконання на симуляторі Apple iOS.

РОЗДІЛ 5
ЕКОНОМІЧНА ЧАСТИНА

5.1 Оцінювання комерційного потенціалу розробки

Головною метою проведення комерційного аудиту є оцінка комерційного потенціалу впровадження методів та засобів розробленої системи адаптивного тестування знань. Для здійснення технологічного аудиту було залучено трьох незалежних експертів з Вінницького національного технічного університету: к.т.н., доцент Ракитянська Г. Б., к.т.н., доцент Майданюк В. П., к.т.н., доцент Рейда О. М. з кафедри програмного забезпечення..

Аудит науково-технічної розробки та її комерційного потенціалу проведено за допомогою таблиці 5.1, застосовуючи п’ятибальну шкалу оцінювання за 12-ма критеріями оцінки.

Таблиця 5.1 – Рекомендовані критерії оцінювання комерційного потенціалу розробки та їх оцінка у балах

Критерії оцінювання та бали (за 5-ти бальною шкалою)					
#	0	1	2	3	4
1	2	3	4	5	6
Технічна здійсненність концепції:					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено роботоздатність продукту в реальних умовах
Ринкові переваги (недоліки):					
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому ринку	Продукт не має аналогів на великому ринку
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів
4	Технічні та споживчі властивості продукту значно гірші, ніж в аналогів	Технічні та споживчі властивості продукту трохи гірші, ніж в аналогів	Технічні та споживчі властивості продукту на рівні аналогів	Технічні та споживчі властивості продукту трохи кращі, ніж в аналогів	Технічні та споживчі властивості продукту значно кращі, ніж в аналогів

Продовження таблиці 5.1.

1	2	3	4	5	6
5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційних витрат аналогів	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів
Ринкові перспективи					
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на ринку	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає
Практична здійсненність					
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання наявних фахівців	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовуються у військово-промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуються у виробництві
11	Термін реалізації ідеї більший за 10 років	Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10-ти років	Термін реалізації ідеї від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій від 3-х до 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій менше 3-х років

Продовження таблиці 5.1.

1	2	3	4	5	6
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки повідомлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь-які регламентні обмеження на виробництво та реалізацію продукту

В таблиці 5.2 наведено результати оцінювання науково-технічного рівня і комерційного потенціалу розробки.

Таблиця 5.2 – Результати оцінювання комерційного потенціалу розробки

Критерії	Прізвище, ініціали, посада експерта		
	1. Ракитянська Г. Б.	2. Майданюк В. П.	3. Рейда О. М.
	Бали, виставлені експертами:		
1	3	4	2
2	3	4	4
3	2	3	1
4	4	2	2
5	2	3	3
6	4	3	2
7	4	2	2
8	1	3	1
9	3	3	4
10	4	2	3
11	2	1	2
12	3	2	3
Сума балів	$СБ_1$=35	$СБ_2$=32	$СБ_3$=29
Середньоарифметична сума балів $\overline{СБ}$	$\overline{СБ} = \frac{\sum_1^3 СБ_i}{3} = \frac{35 + 32 + 29}{3} = 32$		

В таблиці 5.3 наведено шкалу оцінки комерційного потенціалу розробки.

Таблиця 5.3 – Рівні комерційного потенціалу розробки

Середньоарифметична сума балів СБ, розрахована на основі висновків експертів	Рівень комерційного потенціалу розробки
0-10	Низький
11-20	Нижче середнього
21-30	Середній
31-40	Вище середнього
41-48	Високий

За результатами розрахунків, наведених в таблиці 5.2 та шкалою оцінки наведеної в таблиці 5.3 можна зробити висновок щодо рівня комерційного потенціалу розробки. Середньоарифметична сума балів, виставлених експертами склала 32, що відповідає рівню «вище середнього». Така оцінка забезпечена завдяки великому стабільному ринку Java програмного забезпечення та одночасною повною відсутністю рушень для платформи Apple iOS, особливо в розрізі підтримки сучасних версії JVM.

5.2 Прогнозування витрат на виконання науково-дослідної роботи

Запланована форма реалізації діяльності – ФОП без найманих робітників, ПДВ, єдиний податок – група 3.

Витрати, пов’язані з проведенням науково-дослідної роботи можна розрахувати за наступними статтями:

- Витрати на оплату праці;
- Витрати на інструментальне та інфраструктурне програмне забезпечення;
- Амортизаційні відрахування;
- Енергія для науково-виробничих цілей.

5.2.1 Основна заробітна плата

Заробітна плата кожного із залучених осіб визначається за формулою (5.1)

$$З_О = \sum_{i=1}^{K} \frac{M_{ni} * t_i}{T_p} \text{ (грн)}$$

(5.1)

Де k – кількість посад працівників, залучених до процесу дослідження і розробки;

M_{ni} – місячний посадовий оклад конкретного розробника (інженера, дослідника, науковця тощо), грн.;

$Т_р$– число робочих днів в місяці, при 4х денному робочому тижні; приблизно $Т_р = 18$;

t - кількість робочих днів роботи працівника.

Для проектування і розробки системи підтримки Java коду на платформі Apple iOS було залучено одного уніфікованого робітника: Java Software Engineer. Посадові оклади, число днів роботи та витрати на компенсацію наведено в таблиці 5.4

Таблиця 5.4 - Компенсація спеціаліста в дослідницькій установі

Найменування посади	Місячний посадовий оклад, грн.	Оплата за робочий день, грн.	Число робочих днів на місяць	Витрати на заробітну плату грн.
Software Engineer	54000	3000	18	54000
Всього				54000

5.2.2 Розрахунок додаткової заробітної плати робітників

При зазначеній організації виробництва (ФОП, єдиний податок), додаткова заробітна плата не передбачена.

$$З_д = (З_о + З_р) * \frac{Н_{доп}}{100\%} \quad (5.2)$$

$$З_д = 0 \text{ грн}$$

5.2.3 Нарахування на заробітну плату

Для ФОП на третій групі оподаткування єдиний соціальний внесок складає 22% від мінімальної заробітної плати, яка на 2023 рік становить 6700 грн, тоді

Нарахування на заробітну плату $Н_{ЗП}$ робітників, які брали участь у виконанні роботи, розраховуються за формулою (5.3):

$$З_н = З_{мін} * \frac{22}{100} \text{ (грн)} \quad (5.3)$$

де $З_{мін}$ – мінімальна заробітна плата, грн.;

Основна ставка єдиного внеску на загальнообов'язкове державне соціальне страхування на 2023 рік – 22 %, тоді:

$$З_н = 6700 * 0.22 = 1474(\text{грн})$$

5.2.4 Витрати на матеріали та комплектуючі вироби

Дана стаття витрат включає витрати на матеріали, пристрої, засоби, які використовують при виготовленні одиниці продукції. Сучасна розробка програмного не потребує витратних матеріалів: обмін документами виконується в цифровому вигляді, звіти у державні органи реалізує підрядна організація. Тож витрати в цій категорії становлять 0 грн.

5.2.5 Витрати на програмне забезпечення

До даної статті входять витрати на програмне забезпечення, необхідне для проектування та системи підтримки Java коду на платформі Apple iOS. Балансову вартість програмного забезпечення розраховують за формулою 5.5:

$$В_{прг} = \sum_{i=1}^{k} Ц_{іпрг} \cdot С_{пргі} \cdot К_і, \quad (5.5)$$

де $Ц_{іпрг}$ – ціна придбання/використання одиниці програмного засобу цього виду, грн;

$С_{пргі}$ – кількість одиниць програмного забезпечення відповідного найменування, які придбані для проведення досліджень, шт.;

$К_і$ – коефіцієнт, що враховує інсталяцію, налагодження програмного засобу тощо ($К_і$ = 1,10…1,12).

k – кількість найменувань програмних засобів.

Отримані результати наведено в таблиці 5.6.

Таблиця 5.6 – Витрати на використання програмних

Найменування устаткування	Час використання, місяців	Ціна за місяць, грн	Вартість, грн
Підписка Intellij All Products Pack	1	1070	1070
Всього			1070

Отже витрати на програмне забезпечення складають 1070грн на місяць.

$$В_{прг} = 1070 \text{ грн/місяць}$$

5.2.6 Амортизація обладнання

До даної статті включаються амортизаційні відрахування по кожному виду обладнання, устаткування яке використовувалось для проектування та розробки системи адаптивного тестування знань.

$$А_{обл} = \frac{Ц_б}{Т_в} * \frac{t_{вик}}{12} \quad (5.6)$$

де $Ц_б$ – балансова вартість даного виду обладнання (приміщень), грн.;

$t_{вик}$ – час користування;

$Т_в$ – термін використання обладнання (приміщень), цілі місяці.

Для розробки використовувався персональний комп'ютер MacBookPro вартістю 98 999 грн. Амортизаційні відрахування наведено в таблиці 5.7.

Таблиця 5.7 – Амортизаційні відрахування по кожному виду обладнання

Найменування обладнання	Балансова вартість, грн	Строк корисного використання, років	Термін використання обладнання, місяців	Амортизаційні відрахування, на місяць, грн
Ноутбук MacBookPro	98 999	4	48	2062
Всього				2062

$$А_{обл} = 2062 \text{ грн/місяць}$$

5.2.7 Енергія для науково-виробничих цілей

До даної статті відносяться витрати на всі види палива й енергії, що безпосередньо використовуються з технологічною метою на проведення досліджень.

Розробка програмного забезпечення не потребує палива, електрична енергія входить в оплату місця в коворкінгу, тобто для системи підтримки Java коду на платформі Apple iOS енергетичні витрати становлять $B_e = 0$ грн

5.2.8 Витрати на роботи, які виконують сторонні підприємства, установи і організації

Дана стаття витрат включає витрати на проведення досліджень, що не можуть бути виконані штатними працівниками або наявним обладнанням організації, а виконуються на договірній основі іншими підприємствами, установами і організаціями незалежно від форм власності та позаштатними працівниками.

Єдиною статтею витрат в цій категорії є бухгалтерський супровід ФОП і він складає 800 грн на місяць:

$$B_{сп} = 800 \text{ грн}$$

5.2.9 Інші витрати

До статті «Інші витрати» належать витрати, які не знайшли відображення у попередніх статтях витрат і можуть бути віднесені безпосередньо на собівартість розробки за прямими ознаками.

Єдиною статтею з групи «інші витрати» є витрати на оренду робочого місця в коворкінгу і вони складають 4500 грн на місяць.

$$I_в = 4500 \text{ грн/м}$$

5.2.10 Загальновиробничі витрати

Дана стаття витрат охоплює витрати на управління організацією, оплату службових відряджень, витрати на утримання, ремонт та експлуатацію основних засобів, витрати на опалення, освітлення, водопостачання, охорону праці тощо.

Розробка програмного забезпечення не потребує загальновиробничих витрат, оскільки вище перелічені статті витрат покриваються оплатою місця в коворкінгу, тобто для системи підтримки Java коду на платформі Apple iOS загальновиробничі витрати становлять $В_{нзв} = 0$ грн

5.2.11 Загальні витрати

Сума всіх статей витрат дає в результаті витрати на проведення дослідження та розробку системи підтримки Java коду на платформі Apple iOS і розраховується за формулою:

$$В = З_о + З_р + З_{дод} + З_н + М + К_в + В_{спец} + В_{прг} + А_{обл} + В_е + В_{св} + В_{сп} + І_в + В_{нзв} \quad (5.9)$$

$$В = 54000 + 0 + 0 + 1474 + 0 + 0 + 0 + 1070 + 2062 + 0 + 800 + 4500 + 0 = 63906 \text{ грн}$$

Загальні витрати ЗВ на завершення роботи та оформлення її результатів розраховуються за формулою:

$$ЗВ = \frac{В}{\eta}, \quad (5.10)$$

де η – коефіцієнт, який характеризує стадію виконання даної НДР.

Оскільки, загальний строк розробки становить 2 місяці, то коефіцієнт $\beta = 0.5$.

Звідси:

$$ЗВ = \frac{63906}{0.5} = 127812 \text{ грн.}$$

5.3 Розрахунок економічної ефективності науково-технічної розробки

Економічна ефективність дозволяє спрогнозувати чистий прибуток, який може бути отриманий від впровадження розробленої системи.

При розрахунку економічної ефективності потрібно обов’язково враховувати зміну вартості грошей у часі, оскільки від вкладення інвестицій до отримання прибутку минає чимало часу.

Для розрахунку збільшення чистого прибутку підприємства $\Delta П_i$, для кожного із років, протягом яких очікується отримання позитивних результатів від впровадження розробки використовується формула формулою 5.11:

$$\Delta П_i = \sum_1^n \left(\Delta Ц_o \cdot N + Ц_o \cdot \Delta N\right)_i \cdot \lambda \cdot \rho \cdot \left(1 - \frac{\upsilon}{100}\right) \quad (5.11)$$

де $\Delta Ц_0$ – покращення основного оціночного показника від впровадження результатів розробки у даному році.

N – основний кількісний показник, який визначає діяльність підприємства у даному році до впровадження нової розробки;

ΔN – покращення основного кількісного показника діяльності підприємства від впровадження результатів розробки:

Цо – основний оціночний показник, який визначає діяльність підприємства у даному році після впровадження результатів розробки;

n – кількість років, протягом яких очікується отримання позитивних результатів від впровадження розробки:

λ – коефіцієнт, який враховує сплату податку на додану вартість. Оскільки вибрана модель оподаткування ФОП, група 3, не платник ПДВ, то *л* =1

ρ – коефіцієнт, який враховує рентабельність продукту. р = 0,25;

ϑ – ставка податку на прибуток. У 2023 році для ФОП 3ї групи – 5%.

Впровадження результатів розробки дозволяє ширше використовувати Java на платформі Apple iOS, відповідно збільшить кількість розробників на цій платформі. Концепція отримання прибутку будується на підтримці рішення, покращуючи існуючий функціонал та надаючи послуги технічної експертизи. Основні джерела: пожертви на підтримку (donate) через систему підтримки Patreon індивідуальні консультації/підтримка. Припустимо, середній внесок та рахунок за підтримку становитимуть 2000 грн, 18 денний робочий тиждень дозволить приділити увагу 10-ти зверненням на місяць на підтримку і ці показники не планують збільшуватись тож складуть 2000*10*12 = 240000 грн на рік. Оскільки до впровадження рішення цих надходжень не було врахуємо їх в розрахунках. Пожертви не вимагають приділення часу і планується отримання

30 пожертв (що складе 12 * 30 = 360 платежів на рік) на початку розповсюдження розробки з подальшим збільшенням на 20% щорічно. Крім того, розмір внесків, може поетапно збільшуватись на 10%.

$$\Delta П_1 = (240000 + 200 \cdot 360 + (2000 + 200) \cdot 72) \cdot 1 \cdot 0.25 \cdot \left(1 - \frac{5}{100}\right) = 111720 \text{ грн}$$

$$\Delta П_2 = (240000 + 200 \cdot 360 + (2000 + 400) \cdot 144) \cdot 1 \cdot 0{,}25 \cdot \left(1 - \frac{5}{100}\right) = 156180 \text{ грн}$$

$$\Delta П_3 = (240000 + 200 \cdot 360 + (2000 + 600) \cdot 216) \cdot 1 \cdot 0.25 \cdot \left(1 - \frac{5}{100}\right) = 207480 \text{ грн}$$

Далі за формулою 5.12 розраховують приведену вартість збільшення всіх чистих прибутків ПП, що їх може отримати потенційний інвестор від можливого впровадження та комерціалізації розробки:

$$ПП = \sum_{i=1}^{T} \frac{\Delta П_i}{(1+\tau)^t} \qquad (5.12)$$

де ΔПі – збільшення чистого прибутку у кожному з років, протягом яких виявляються результати впровадження науково-технічної розробки, грн;

Т – період часу, протягом якого очікується отримання позитивних результатів від впровадження та комерціалізації розробки, роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні, τ =0,05…0,15. В Україні рівень інфляції за підсумком 2023 року склав 10.6%, прогнозований рівень на 2024 рік – 8.5% ;

t – період часу (в роках) від моменту початку впровадження розробки до моменту отримання потенційним інвестором додаткових чистих прибутків у цьому році.

$$ПП = \frac{111720}{1 + 0.1} + \frac{156180}{(1 + 0.085)^2} + \frac{207480}{(1 + 0.085)^3} = 436734 \text{ грн}$$

За формулою 5.13 необхідно розрахувати величину початкових інвестицій PV, які потенційний інвестор має вкласти для впровадження і комерціалізації розробки.

$$PV = k_{\text{інв}} \cdot \text{ЗВ} \quad (5.13)$$

де $k_{\text{інв}}$ – коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки та її комерціалізацію. Оскільки реалізація силами одного розробника на схемі ФОП не включає витрати на підготовку приміщень, розробку технологій, навчання персоналу, маркетингові заходи, приймаємо $k_{\text{інв}} = 1$;

ЗВ – загальні витрати на проведення науково-технічної розробки та оформлення її результатів, грн.

$$PV = 1 \cdot 127812 = 127812 \text{ грн}$$

Тоді абсолютний економічний ефект $E_{\text{абс}}$ або чистий приведений дохід (NPV, Net Present Value) для потенційного інвестора від можливого впровадження та комерціалізації розробки становитиме:

$$E_{\text{абс}} = \text{ПП} - PV \quad (5.14)$$

де ПП – приведена вартість зростання всіх чистих прибутків від можливого впровадження та комерціалізації науково-технічної розробки, грн;

PV – теперішня вартість початкових інвестицій, грн.

$$E_{\text{абс}} = 436734 - 127812 = 308922 \text{ грн}$$

Оскільки величина економічного ефекту має велике додатне значення, це свідчить про потенційну зацікавленість інвесторів у впровадженні та комерціалізацію розробки.

5.4 Розрахунок ефективності вкладених інвестицій та періоду їх окупності

Для остаточного прийняття рішення про впровадження розробки та виведення її на ринок необхідно розрахувати внутрішню економічну дохідність $E_в$ або показник внутрішньої норми дохідності (IRR, Internal Rate of Return) вкладених інвестицій та порівняти її з так званою бар'єрною ставкою дисконтування, яка визначає ту мінімальну внутрішню економічну дохідність, нижче якої інвестиції в будь-яку розробку вкладати буде економічно недоцільно.

Розрахуємо відносну (щорічну) ефективність вкладених в наукову розробку інвестицій $E_в$. Для цього користуються формулою 5.15:

$$E_в = \sqrt[T_ж]{1 + \frac{E_{абс}}{PV}} - 1, \tag{5.15}$$

де $E_{абс}$ – абсолютний економічний ефект вкладених інвестицій, грн; PV – теперішня вартість початкових інвестицій, грн;

$T_ж$ – життєвий цикл науково-технічної розробки, тобто час від початку її розробки до закінчення отримування позитивних результатів від її впровадження, роки.

$$E_в = \sqrt[3]{1 + \frac{308922}{127812}} - 1 = \sqrt[3]{3.417} - 1 = 0.50 = 50\%$$

Визначимо мінімальну ставку дисконтування, яка у загальному вигляді визначається за формулою 5.16:

$$\tau = d + f, \tag{5.16}$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2023 році в Україні d= (0,14…0,2);

f−показник, що характеризує ризикованість вкладень; зазвичай, величина f = (0,05...0,1).

$$\tau_{min} = 0.15 + 0.06 = 0.21$$

Так як $E_в > \tau_{min}$ то інвестор може бути зацікавлений у фінансуванні даної наукової розробки.

Далі розраховується період окупності інвестицій, вкладених у реалізацію проекту за формулою 5.17.

$$T_{ок} = \frac{1}{E_в} \qquad (5.17)$$

де Ев – внутрішня економічна дохідність вкладених інвестицій.

$\mathrm{T_{ок}} = \frac{1}{0.5} = 2$ роки

Так як $\mathrm{T_{ок}} \leq 3$ років, то це свідчить про комерційну привабливість науково-технічної розробки і може спонукати потенційного інвестора профінансувати впровадження цієї розробки та виведення її на ринок.

5.5 Висновки

В економічній частині оцінено комерційний потенціал розробки системи підтримки Java коду на платформі Apple iOS, який склав значення вище середнього.

Запрогнозовано витрати на проектування та реалізацію системи за необхідними статтями витрат, які склали 63906 грн. Загальна величина витрат склала 127812 грн.

Обрахунок економічної ефективності та терміну окупності вкладених інвестицій показали, що розробка є економічно доцільною та привабливою для потенційних інвесторів. Термін окупності інвестицій складає 2 роки та прогнозований прибуток за 3 роки 436734 грн

ВИСНОВКИ

У даній магістерській кваліфікаційній роботі було проведено дослідження методів та засобів необхідних для виконання Java коду на платформі Apple iOS рівня JVM 16, з приділення уваги реалізації підтримки Java Records (JEP-395).

Отримані основні результати роботи:

1. Проаналізовано існуючий стан рішень та обрано найкраще існуюче рішення, для удосконалення розробленими методами.
2. Проаналізовано існуючий стан та сформовано перелік засобів необхідних для реалізації: перелік класів та засобів реалізації.
3. Проаналізовані середовища розробки, мови програмування та інший наявний інструментарій. Як інтегральне середовище розробки обрано Intellij Idea, оскільки це рішення є стандартом де-факту в розробці програмного забезпечення на мові програмування Java і також має безкоштовну версію з достатнім функціоналом (Community Edition).
4. Для розробки Java віртуальних машин використовується мова програмування Java. Рішення досягнуто використовуючи цю мову програмування.
5. Проведено тестування і виконання JVM16 коду на платформі Apple iOS. Результати показали коректність поведінки отриманого машинного коду у відповідності до JEP-395.
6. Розрахунки в економічному розділі показали економічну доцільність даної розробки, що підтверджує отриманий коефіцієнт доцільності виконання науково-дослідної роботи $K_P > 1$.

Всі цілі та поставлені задачі магістерської кваліфікаційної роботи було виконано.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Esen Sagynov: Understanding JVM Internals
2. Tim Lindholm, Frank Yellin. Java Virtual Machine Specification Second Edition
3. Oracle Java.
   URL: https://www.oracle.com/cis/java/technologies/downloads/
4. JavaServer Faces Technology.
   URL: https://www.oracle.com/java/technologies/javaserverfaces.html
5. Spring Boot. URL: https://spring.io/projects/spring-boot/
6. Android. URL: https://www.android.com
7. JDK-21. URL: https://openjdk.org/projects/jdk/21/
8. Swift is general purpose programming language.
   URL: https://www.swift.org
9. Objective C.
   URL:https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html
10. Gluon. URL**:** https://gluonhq.com/
11. JavaFX. URL: https://openjfx.io
12. Multi-OS Engine(MOE). URL: https://multi-os-engine.org/
13. The LLVM Compiler Infrastructure. URL: https://llvm.org
14. MobiVM. URL: https://github.com/MobiVM/robovm
15. libGDX cross platform game development framework. URL: https://libgdx.com
16. CodeName One. URL: https://www.codenameone.com
17. j2objc. URL: https://developers.google.com/j2objc
18. XMLVM. URL: http://www.xmlvm.org/overview/
19. JEP 395. URL: Records: https://openjdk.org/jeps/395
20. Maven Central statistics. URL: https://blog.sonatype.com/2010/12/now-available-central-download-statistics-for-oss-projects/

21. JEP358 Helpful NullPointerExceptions.
URL: https://openjdk.org/jeps/358
22. JEP409 Sealed Classes. URL: https//openjdk.org/jeps/409
23. JEP444 Virtual Threads. URL: https//openjdk.org/jeps/444
24. LLD - The LLVM Linker. URL: https://lld.llvm.org
25. XCode: . URL https://developer.apple.com/xcode/
26. Soot - A framework for analyzing and transforming Java and Android applications. URL: http://soot-oss.github.io/soot/
27. LLVM Language Reference Manual. URL: https://llvm.org/docs/LangRef.html
28. SR 292: Supporting Dynamically Typed Languages on the JavaTM Platform. URL: https://jcp.org/en/jsr/detail?id=292
29. Intellij Idea. URL: https://www.jetbrains.com/idea/
30. Eclipse. URL: https://www.eclipse.org/downloads/
31. Apache NetBeans: . URL https://netbeans.apache.org
32. Junit5. URL: https://junit.org/junit5/
33. How Much Does It Cost To Test An App In 2024. URL: https://www.appsierra.com/blog/cost-to-test-an-app#
34. Gradle. URL: https://gradle.org

87

ДОДАТОК А

(обов'язковий)

Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
Завідувач кафедрою ПЗ
д.т.н., проф.
_____ О. Н. Романюк
"19" вересня 2023 р.

Технічне завдання
на магістерську кваліфікаційну роботу «Методи та програмні засоби підтримки Java коду на платформі Apple iOS» за спеціальністю 121 – Інженерія програмного забезпечення

Керівник магістерської кваліфікаційної роботи:
_____ к.т.н., доцент Г. О. Черноволик
"19" вересня 2023 р.
Виконав:
_____ студент гр. 2ПІ-22м І. В. Мисловський
"19" вересня 2023 р.

Вінниця – 2023 року

1. Найменування та галузь застосування

Магістерська кваліфікаційна робота: «Методи та програмні засоби підтримки Java коду на платформі Apple iOS».

Галузь застосування – компіляція Java коду, розробка.

2. Підстава для розробки.

Підставою для виконання магістерської кваліфікаційної роботи (МКР) є індивідуальне завдання на БДР та наказ № 247 від «18» вересня 2023 р. ректора ВНТУ про закріплення тем МКР.

3. Мета та призначення розробки.

Метою магістерської кваліфікаційної роботи є підтримка сучасного Java коду на платформі Apple iOS.

Основними задачами дослідження є:

- обґрунтування доцільності розробки нового технічного рішення;
- розробка методів і засобів для підтримки Java Records (JEP-395);
- розрахунок економічних показників розробки.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись МКР:

1. Tim Lindholm. The Java Virtual Machine Specification. Java SE 18 edition.
2. Compilation and Execution of a Java Program: https://www.geeksforgeeks.org/compilation-execution-java-program/
3. Oracle Java: https://www.oracle.com/cis/java/technologies/downloads/
4. JEP 395: Records: https://openjdk.org/jeps/395.

5. Технічні вимоги

– середовище розробки – Intellij Idea;
– мова програмування – Java;
– відповідність JEP-395;

6. Конструктивні вимоги

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред'являється по закінченню робіт:

1. Пояснювальна записка до МКР;
2. Технічне завдання;
3. Лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

9. Стадії та етапи розробки:

№ з/п	Назва етапів дипломного проєкту	Строк виконання етапів роботи
1	Обґрунтування вибору методу розробки та постановка задач	20.09.23 – 02.10.23
2	Розробка архітектури та алгоритмів програмного продукту	03.10.23 – 23.10.23
3	Аналіз і вибір мови програмування та середовища розробки	16.10.23 – 23.10.23
4	Розробка програмного продукту	24.10.23 – 13.11.23
5	Тестування програми	14.11.23 – 21.11.23
6	Розробка економічної частини	17.11.23 –25.11.23
7	Оформлення матеріалів до захисту МКР	22.11.23 – 01.12.23

10. Порядок контролю та прийняття

Порядок контролю і приймання роботи регламентується відповідними документами ВНТУ і державними стандартами.

90

ДОДАТОК Б

(обов'язковий)

ПРОТОКОЛ ПЕРЕВІРКИ НАВЧАЛЬНОЇ (КВАЛІФІКАЦІЙНОЇ) РОБОТИ

Назва роботи:

Тип роботи: магістерська кваліфікаційна робота

Підрозділ : кафедра програмного забезпечення, ФІТКІ, 2ПІ – 22м

Науковий керівник:

Unicheck	
Оригінальність	**84**
Схожість	16

Аналіз звіту подібності

■ **Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.**

□ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.

□ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку _____ Черноволик Г. О.

Опис прийнятого рішення: **допустити до захисту**

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck

Автор роботи _____ Мисловський І.В.

Керівник роботи _____ Черноволик Г.О.

ДОДАТОК В

(довідниковий)

Лістинг програми

```
package java.lang;

/**
 * This is the common base class of all Java language record classes.
 */
public abstract class Record {
    /**
     * Constructor for record classes to call.
     */
    protected Record() {}

    /**
     * Indicates whether some other object is "equal to" this one.  In addition
     */
    @Override
    public abstract boolean equals(Object obj);

    /**
     * Returns a hash code value for the record.
     * @return  a hash code value for this record.
     */
    @Override
    public abstract int hashCode();

    /**
     * Returns a string representation of the record.
     */
    @Override
    public abstract String toString();
}


public class RecordsSupport {

    private static int hashCombiner(int x, int y) {
        return x*31 + y;
    }

    private static boolean eq(Object a, Object b) { return a == b; }
    private static boolean eq(byte a, byte b) { return a == b; }
    private static boolean eq(short a, short b) { return a == b; }
    private static boolean eq(char a, char b) { return a == b; }
    private static boolean eq(int a, int b) { return a == b; }
    private static boolean eq(long a, long b) { return a == b; }
    private static boolean eq(float a, float b) { return Float.compare(a, b) ==
0; }
    private static boolean eq(double a, double b) { return Double.compare(a, b)
== 0; }
    private static boolean eq(boolean a, boolean b) { return a == b; }

    /** Get the method handle for combining two values of a given type */
    private static MethodHandle equalator(Class<?> clazz) {
        return (clazz.isPrimitive()
                ? primitiveEquals.get(clazz)
                : OBJECTS_EQUALS.asType(MethodType.methodType(boolean.class,
clazz, clazz)));
    }

    /** Get the hasher for a value of a given type */
    private static MethodHandle hasher(Class<?> clazz) {
```

```
        return (clazz.isPrimitive()
                ? primitiveHashers.get(clazz)
                : OBJECTS_HASHCODE.asType(MethodType.methodType(int.class,
clazz)));
    }

    /** Get the stringifier for a value of a given type */
    private static MethodHandle stringifier(Class<?> clazz) {
        return (clazz.isPrimitive()
                ? primitiveToString.get(clazz)
                : OBJECTS_TOSTRING.asType(MethodType.methodType(String.class,
clazz)));
    }

    /**
     * Generates a method handle for the {@code equals} method for a given data
class
     */
    private static MethodHandle makeEquals(Class<?> receiverClass,
                                           List<MethodHandle> getters) {
        MethodType rr = MethodType.methodType(boolean.class, receiverClass,
receiverClass);
        MethodType ro = MethodType.methodType(boolean.class, receiverClass,
Object.class);
        MethodHandle instanceFalse = MethodHandles.dropArguments(FALSE, 0,
receiverClass, Object.class); // (RO)Z
        MethodHandle instanceTrue = MethodHandles.dropArguments(TRUE, 0,
receiverClass, Object.class); // (RO)Z
        MethodHandle isSameObject = OBJECT_EQ.asType(ro); // (RO)Z
        MethodHandle isInstance =
MethodHandles.dropArguments(CLASS_IS_INSTANCE.bindTo(receiverClass), 0,
receiverClass); // (RO)Z
        MethodHandle accumulator = MethodHandles.dropArguments(TRUE, 0,
receiverClass, receiverClass); // (RR)Z

        for (MethodHandle getter : getters) {
            MethodHandle equalator = equalator(getter.type().returnType()); //
(TT)Z
            MethodHandle thisFieldEqual =
MethodHandles.filterArguments(equalator, 0, getter, getter); // (RR)Z
            accumulator = MethodHandles.guardWithTest(thisFieldEqual,
accumulator, instanceFalse.asType(rr));
        }

        return MethodHandles.guardWithTest(isSameObject,
                                           instanceTrue,

MethodHandles.guardWithTest(isInstance, accumulator.asType(ro), instanceFalse));
    }

    /**
     * Generates a method handle for the {@code hashCode} method for a given
data class
     */
    private static MethodHandle makeHashCode(Class<?> receiverClass,
                                             List<MethodHandle> getters) {
        MethodHandle accumulator = MethodHandles.dropArguments(ZERO, 0,
receiverClass); // (R)I

        // @@@ Use loop combinator instead?
        for (MethodHandle getter : getters) {
            MethodHandle hasher = hasher(getter.type().returnType()); // (T)I
            MethodHandle hashThisField = MethodHandles.filterArguments(hasher,
0, getter);    // (R)I
            MethodHandle combineHashes =
MethodHandles.filterArguments(HASH_COMBINER, 0, accumulator, hashThisField); //
(RR)I
```

```
            accumulator = MethodHandles.permuteArguments(combineHashes,
accumulator.type(), 0, 0); // adapt (R)I to (RR)I
        }

        return accumulator;
    }

    /**
     * Generates a method handle for the {@code toString} method for a given
data class
     */
    private static MethodHandle makeToString(MethodHandles.Lookup lookup,
                                             Class<?> receiverClass,
                                             MethodHandle[] getters,
                                             List<String> names) {
        assert getters.length == names.size();
        if (getters.length == 0) {
            // special case
            MethodHandle emptyRecordCase = MethodHandles.constant(String.class,
receiverClass.getSimpleName() + "[]");
            emptyRecordCase = MethodHandles.dropArguments(emptyRecordCase, 0,
receiverClass); // (R)S
            return emptyRecordCase;
        }

        boolean firstTime = true;
        MethodHandle[] mhs;
        List<List<MethodHandle>> splits;
        MethodHandle[] toSplit = getters;
        int namesIndex = 0;
        do {
            /* StringConcatFactory::makeConcatWithConstants can only deal with
200 slots, longs and double occupy two
             * the rest 1 slot, we need to chop the current `getters` into
chunks, it could be that for records with
             * a lot of components that we need to do a couple of iterations.
The main difference between the first
             * iteration and the rest would be on the recipe
             */
            splits = split(toSplit);
            mhs = new MethodHandle[splits.size()];
            for (int splitIndex = 0; splitIndex < splits.size(); splitIndex++) {
                String recipe = "";
                if (firstTime && splitIndex == 0) {
                    recipe = receiverClass.getSimpleName() + "[";
                }
                for (int i = 0; i < splits.get(splitIndex).size(); i++) {
                    recipe += firstTime ? names.get(namesIndex) + "=" + "\1" :
"\1";
                    if (firstTime && namesIndex != names.size() - 1) {
                        recipe += ", ";
                    }
                    namesIndex++;
                }
                if (firstTime && splitIndex == splits.size() - 1) {
                    recipe += "]";
                }
                Class<?>[] concatTypeArgs = new
Class<?>[splits.get(splitIndex).size()];
                // special case: no need to create another getters if there is
only one split
                MethodHandle[] currentSplitGetters = new
MethodHandle[splits.get(splitIndex).size()];
                for (int j = 0; j < splits.get(splitIndex).size(); j++) {
                    concatTypeArgs[j] =
splits.get(splitIndex).get(j).type().returnType();
                    currentSplitGetters[j] = splits.get(splitIndex).get(j);
```

```
                }
                MethodType concatMT = MethodType.methodType(String.class,
concatTypeArgs);
                try {
                    mhs[splitIndex] =
StringConcatFactory.makeConcatWithConstants(
                            lookup, "",
                            concatMT,
                            recipe,
                            new Object[0]
                    ).getTarget();
                    mhs[splitIndex] =
MethodHandles.filterArguments(mhs[splitIndex], 0, currentSplitGetters);
                    // this will spread the receiver class across all the
getters
                    mhs[splitIndex] = MethodHandles.permuteArguments(
                            mhs[splitIndex],
                            MethodType.methodType(String.class, receiverClass),
                            new int[splits.get(splitIndex).size()]
                    );
                } catch (Throwable t) {
                    throw new RuntimeException(t);
                }
            }
            toSplit = mhs;
            firstTime = false;
        } while (splits.size() > 1);
        return mhs[0];
    }

    /**
     * Chops the getters into smaller chunks according to the maximum number of
slots
     */
    private static List<List<MethodHandle>> split(MethodHandle[] getters) {
        List<List<MethodHandle>> splits = new ArrayList<>();

        int slots = 0;

        // Need to peel, so that neither call has more than acceptable number
        // of slots for the arguments.
        List<MethodHandle> cArgs = new ArrayList<>();
        for (MethodHandle methodHandle : getters) {
            Class<?> returnType = methodHandle.type().returnType();
            int needSlots = (returnType == long.class || returnType ==
double.class) ? 2 : 1;
            if (slots + needSlots > MAX_STRING_CONCAT_SLOTS) {
                splits.add(cArgs);
                cArgs = new ArrayList<>();
                slots = 0;
            }
            cArgs.add(methodHandle);
            slots += needSlots;
        }

        // Flush the tail slice
        if (!cArgs.isEmpty()) {
            splits.add(cArgs);
        }

        return splits;
    }

}
```

ДОДАТОК Г

(довідниковий)

Лістинг проміжних артефактів

Початковий код на Java:

```
public record Account(String name, int id) {
}
```

Відповідний байт-код

```
Отримано командою декомпіляції:
javap -v -c -p -s -constants ./Account.class

Classfile /Users/imyslovskyy/dkimitsa/Account.class
  Last modified 20 Dec 2023; size 1197 bytes
  SHA-256 checksum
e8de9b79d61d0b405d0eecbf3fe756912477b952bd1f2c13d6fb3b9cd5c88920
  Compiled from "Account.java"
public final class Account extends java.lang.Record
  minor version: 0
  major version: 63
  flags: (0x0031) ACC_PUBLIC, ACC_FINAL, ACC_SUPER
  this_class: #8                          // Account
  super_class: #2                         // java/lang/Record
  interfaces: 0, fields: 2, methods: 6, attributes: 4
Constant pool:
   #1 = Methodref          #2.#3          // java/lang/Record."<init>":()V
   #2 = Class              #4             // java/lang/Record
   #3 = NameAndType        #5:#6          // "<init>":()V
   #4 = Utf8               java/lang/Record
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Fieldref           #8.#9          // Account.name:Ljava/lang/String;
   #8 = Class              #10            // Account
   #9 = NameAndType        #11:#12        // name:Ljava/lang/String;
  #10 = Utf8               Account
  #11 = Utf8               name
  #12 = Utf8               Ljava/lang/String;
  #13 = Fieldref           #8.#14         // Account.id:I
  #14 = NameAndType        #15:#16        // id:I
  #15 = Utf8               id
  #16 = Utf8               I
  #17 = InvokeDynamic      #0:#18         //
#0:toString:(LAccount;)Ljava/lang/String;
  #18 = NameAndType        #19:#20        //
toString:(LAccount;)Ljava/lang/String;
  #19 = Utf8               toString
  #20 = Utf8               (LAccount;)Ljava/lang/String;
  #21 = InvokeDynamic      #0:#22         // #0:hashCode:(LAccount;)I
  #22 = NameAndType        #23:#24        // hashCode:(LAccount;)I
  #23 = Utf8               hashCode
  #24 = Utf8               (LAccount;)I
  #25 = InvokeDynamic      #0:#26         //
#0:equals:(LAccount;Ljava/lang/Object;)Z
  #26 = NameAndType        #27:#28        //
equals:(LAccount;Ljava/lang/Object;)Z
  #27 = Utf8               equals
  #28 = Utf8               (LAccount;Ljava/lang/Object;)Z
  #29 = Utf8               (Ljava/lang/String;I)V
```

```
  #30 = Utf8               Code
  #31 = Utf8               LineNumberTable
  #32 = Utf8               MethodParameters
  #33 = Utf8               ()Ljava/lang/String;
  #34 = Utf8               ()I
  #35 = Utf8               (Ljava/lang/Object;)Z
  #36 = Utf8               SourceFile
  #37 = Utf8               Account.java
  #38 = Utf8               Record
  #39 = Utf8               BootstrapMethods
  #40 = MethodHandle       6:#41          // REF_invokeStatic
java/lang/runtime/ObjectMethods.bootstrap:(Ljava/lang/invoke/MethodHandles$Looku
p;Ljava/lang/String;Ljava/lang/invoke/TypeDescriptor;Ljava/lang/Class;Ljava/lang
/String;[Ljava/lang/invoke/MethodHandle;)Ljava/lang/Object;
  #41 = Methodref          #42.#43        //
java/lang/runtime/ObjectMethods.bootstrap:(Ljava/lang/invoke/MethodHandles$Looku
p;Ljava/lang/String;Ljava/lang/invoke/TypeDescriptor;Ljava/lang/Class;Ljava/lang
/String;[Ljava/lang/invoke/MethodHandle;)Ljava/lang/Object;
  #42 = Class              #44            // java/lang/runtime/ObjectMethods
  #43 = NameAndType        #45:#46        //
bootstrap:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/i
nvoke/TypeDescriptor;Ljava/lang/Class;Ljava/lang/String;[Ljava/lang/invoke/Metho
dHandle;)Ljava/lang/Object;
  #44 = Utf8               java/lang/runtime/ObjectMethods
  #45 = Utf8               bootstrap
  #46 = Utf8
(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/Type
Descriptor;Ljava/lang/Class;Ljava/lang/String;[Ljava/lang/invoke/MethodHandle;)L
java/lang/Object;
  #47 = String             #48            // name;id
  #48 = Utf8               name;id
  #49 = MethodHandle       1:#7           // REF_getField
Account.name:Ljava/lang/String;
  #50 = MethodHandle       1:#13          // REF_getField Account.id:I
  #51 = Utf8               InnerClasses
  #52 = Class              #53            //
java/lang/invoke/MethodHandles$Lookup
  #53 = Utf8               java/lang/invoke/MethodHandles$Lookup
  #54 = Class              #55            // java/lang/invoke/MethodHandles
  #55 = Utf8               java/lang/invoke/MethodHandles
  #56 = Utf8               Lookup
{
  private final java.lang.String name;
    descriptor: Ljava/lang/String;
    flags: (0x0012) ACC_PRIVATE, ACC_FINAL

  private final int id;
    descriptor: I
    flags: (0x0012) ACC_PRIVATE, ACC_FINAL

  public Account(java.lang.String, int);
    descriptor: (Ljava/lang/String;I)V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=2, locals=3, args_size=3
         0: aload_0
         1: invokespecial #1                  // Method
java/lang/Record."<init>":()V
         4: aload_0
         5: aload_1
         6: putfield      #7                  // Field name:Ljava/lang/String;
         9: aload_0
        10: iload_2
        11: putfield      #13                 // Field id:I
        14: return
      LineNumberTable:
        line 1: 0
```

```
    MethodParameters:
      Name                           Flags
      name
      id

  public final java.lang.String toString();
    descriptor: ()Ljava/lang/String;
    flags: (0x0011) ACC_PUBLIC, ACC_FINAL
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokedynamic #17,  0             // InvokeDynamic
#0:toString:(LAccount;)Ljava/lang/String;
         6: areturn
      LineNumberTable:
        line 1: 0

  public final int hashCode();
    descriptor: ()I
    flags: (0x0011) ACC_PUBLIC, ACC_FINAL
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokedynamic #21,  0             // InvokeDynamic
#0:hashCode:(LAccount;)I
         6: ireturn
      LineNumberTable:
        line 1: 0

  public final boolean equals(java.lang.Object);
    descriptor: (Ljava/lang/Object;)Z
    flags: (0x0011) ACC_PUBLIC, ACC_FINAL
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
         1: aload_1
         2: invokedynamic #25,  0             // InvokeDynamic
#0:equals:(LAccount;Ljava/lang/Object;)Z
         7: ireturn
      LineNumberTable:
        line 1: 0

  public java.lang.String name();
    descriptor: ()Ljava/lang/String;
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: getfield      #7                  // Field name:Ljava/lang/String;
         4: areturn
      LineNumberTable:
        line 1: 0

  public int id();
    descriptor: ()I
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: getfield      #13                 // Field id:I
         4: ireturn
      LineNumberTable:
        line 1: 0
}
SourceFile: "Account.java"
Record:
  java.lang.String name;
```

```
    descriptor: Ljava/lang/String;

  int id;
    descriptor: I

BootstrapMethods:
  0: #40 REF_invokeStatic
java/lang/runtime/ObjectMethods.bootstrap:(Ljava/lang/invoke/MethodHandles$Looku
p;Ljava/lang/String;Ljava/lang/invoke/TypeDescriptor;Ljava/lang/Class;Ljava/lang
/String;[Ljava/lang/invoke/MethodHandle;)Ljava/lang/Object;
    Method arguments:
      #8 Account
      #47 name;id
      #49 REF_getField Account.name:Ljava/lang/String;
      #50 REF_getField Account.id:I
InnerClasses:
  public static final #56= #52 of #54;    // Lookup=class
java/lang/invoke/MethodHandles$Lookup of class java/lang/invoke/MethodHandles
```

Відповідний LLVM IR-код

```
define private %Object* @"[j]com.mycompany.myapp.Account[allocator]"(%Env* %p0)
nounwind alwaysinline optsize {
label0:
    %t0 = call i8** @"[j]com.mycompany.myapp.Account[info]"()
    %t1 = call %Object* @"_bcAllocate"(%Env* %p0, i8** %t0)
    ret %Object* %t1
}

define external %Object*
@"[j]com.mycompany.myapp.Account[allocator][clinit]"(%Env* %p0) nounwind
noinline optsize {
label0:
    %t0 = call i8** @"[j]com.mycompany.myapp.Account[info]"()
    %t1 = bitcast i8** %t0 to {i8*, i32}*
    %t2 = getelementptr {i8*, i32}* %t1, i32 0, i32 1
    %t3 = load i32* %t2
    %t4 = and i32 %t3, 512
    %t5 = icmp eq i32 %t4, 512
    br i1 %t5, label %label1, label %label2
label1:
    %t6 = call %Object* @"[j]com.mycompany.myapp.Account[allocator]"(%Env* %p0)
    ret %Object* %t6
label2:
    call void @"_bcInitializeClass"(%Env* %p0, i8** %t0)
    br label %label1
}

define external %Object*
@"[j]com.mycompany.myapp.Account.name(Ljava/lang/String;)[get]"(%Env* %p0,
%Object* %p1) nounwind alwaysinline optsize {
label0:
    %t0 = bitcast %Object* %p1 to i8*
    %t1 = getelementptr i8* %t0, i32 ptrtoint (%Object** getelementptr
({%DataObject, <{<{}>, <{<{}>, %Object*}>, <{<{}>, i32}>}>}* null, i32 0, i32 1,
i32 1, i32 1) to i32)
    %t2 = bitcast i8* %t1 to %Object**
    %t3 = load %Object** %t2
    ret %Object* %t3
}

define private void
@"[j]com.mycompany.myapp.Account.name(Ljava/lang/String;)[set]"(%Env* %p0,
%Object* %p1, %Object* %p2) nounwind alwaysinline optsize {
label0:
    %t0 = bitcast %Object* %p1 to i8*
```

```
    %t1 = getelementptr i8* %t0, i32 ptrtoint (%Object** getelementptr
({%DataObject, <{<{}>, <{<{}>, %Object*}>, <{<{}>, i32}>}>}* null, i32 0, i32 1,
i32 1, i32 1) to i32)
    %t2 = bitcast i8* %t1 to %Object**
    store %Object* %p2, %Object** %t2
    fence seq_cst
    ret void
}

define external i32 @"[j]com.mycompany.myapp.Account.id(I)[get]"(%Env* %p0,
%Object* %p1) nounwind alwaysinline optsize {
label0:
    %t0 = bitcast %Object* %p1 to i8*
    %t1 = getelementptr i8* %t0, i32 ptrtoint (i32* getelementptr ({%DataObject,
<{<{}>, <{<{}>, %Object*}>, <{<{}>, i32}>}>}* null, i32 0, i32 1, i32 2, i32 1)
to i32)
    %t2 = bitcast i8* %t1 to i32*
    %t3 = load i32* %t2
    ret i32 %t3
}

define private void @"[j]com.mycompany.myapp.Account.id(I)[set]"(%Env* %p0,
%Object* %p1, i32 %p2) nounwind alwaysinline optsize {
label0:
    %t0 = bitcast %Object* %p1 to i8*
    %t1 = getelementptr i8* %t0, i32 ptrtoint (i32* getelementptr ({%DataObject,
<{<{}>, <{<{}>, %Object*}>, <{<{}>, i32}>}>}* null, i32 0, i32 1, i32 2, i32 1)
to i32)
    %t2 = bitcast i8* %t1 to i32*
    store i32 %p2, i32* %t2
    fence seq_cst
    ret void
}

define weak void
@"[J]com.mycompany.myapp.Account.<init>(Ljava/lang/String;I)V"(%Env* %p0,
%Object* %p1, %Object* %p2, i32 %p3) nounwind noinline optsize {
label0:
    %r0 = alloca %Object*
    %r1 = alloca %Object*
    %i0 = alloca i32
    call void @"checkso"()
    store %Object* %p1, %Object** %r0, !dbg !{i32 37, i32 0, !7, null}
    store %Object* %p2, %Object** %r1, !dbg !{i32 37, i32 0, !7, null}
    store i32 %p3, i32* %i0, !dbg !{i32 37, i32 0, !7, null}
    %t0 = load %Object** %r0, !dbg !{i32 37, i32 0, !7, null}
    call void
@"[j]java.lang.Object.<init>()V[Invokespecial(com/mycompany/myapp/Account,com/my
company/myapp/Account)]"(%Env* %p0, %Object* %t0), !dbg !{i32 37, i32 0, !7,
null}
    %t1 = load %Object** %r1, !dbg !{i32 38, i32 0, !7, null}
    %t2 = load %Object** %r0, !dbg !{i32 38, i32 0, !7, null}
    call void
@"[j]com.mycompany.myapp.Account.name(Ljava/lang/String;)[set]"(%Env* %p0,
%Object* %t2, %Object* %t1), !dbg !{i32 38, i32 0, !7, null}
    %t3 = load i32* %i0, !dbg !{i32 39, i32 0, !7, null}
    %t4 = load %Object** %r0, !dbg !{i32 39, i32 0, !7, null}
    call void @"[j]com.mycompany.myapp.Account.id(I)[set]"(%Env* %p0, %Object*
%t4, i32 %t3), !dbg !{i32 39, i32 0, !7, null}
    ret void, !dbg !{i32 40, i32 0, !7, null}
}


define weak %Object*
@"[J]com.mycompany.myapp.Account.toString()Ljava/lang/String;"(%Env* %p0,
%Object* %p1) nounwind noinline optsize {
label0:
```

```
    %r0 = alloca %Object*
    %$r1 = alloca %Object*
    %$i0 = alloca i32
    %$v3 = alloca %Object*
    %$v4 = alloca %Object*
    %$v5 = alloca %Object*
    %$v6 = alloca %Object*
    call void @"checkso"()
    store %Object* %p1, %Object** %r0, !dbg !{i32 60, i32 0, !11, null}
    %t0 = load %Object** %r0, !dbg !{i32 60, i32 0, !11, null}
    %t1 = call %Object*
@"[j]com.mycompany.myapp.Account.name(Ljava/lang/String;)[get]"(%Env* %p0,
%Object* %t0), !dbg !{i32 60, i32 0, !11, null}
    store %Object* %t1, %Object** %$r1, !dbg !{i32 60, i32 0, !11, null}
    %t2 = load %Object** %r0, !dbg !{i32 60, i32 0, !11, null}
    %t3 = call i32 @"[j]com.mycompany.myapp.Account.id(I)[get]"(%Env* %p0,
%Object* %t2), !dbg !{i32 60, i32 0, !11, null}
    store i32 %t3, i32* %$i0, !dbg !{i32 60, i32 0, !11, null}
    %t4 = call %Object*
@"[j]java.lang.StringBuilder[New(com/mycompany/myapp/Account)]"(%Env* %p0)
    store %Object* %t4, %Object** %$v3
    %t5 = load %Object** %$v3
    call void
@"[j]java.lang.StringBuilder.<init>()V[Invokespecial(com/mycompany/myapp/Account
,java/lang/StringBuilder)]"(%Env* %p0, %Object* %t5)
    %t6 = call %Object* @"[j]str_Account_7Bname_3D_27_00[ldcstring]"(%Env* %p0)
    store %Object* %t6, %Object** %$v4
    %t7 = load %Object** %$v3
    %t8 = load %Object** %$v4
    %t9 = call %Object*
@"[j]java.lang.StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
[Invokevirtual(com/mycompany/myapp/Account,java/lang/StringBuilder)]"(%Env* %p0,
%Object* %t7, %Object* %t8)
    %t10 = load %Object** %$v3
    %t11 = load %Object** %$r1
    %t12 = call %Object*
@"[j]java.lang.StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
[Invokevirtual(com/mycompany/myapp/Account,java/lang/StringBuilder)]"(%Env* %p0,
%Object* %t10, %Object* %t11)
    %t13 = call %Object* @"[j]str__27_2C_20id_3D_00[ldcstring]"(%Env* %p0)
    store %Object* %t13, %Object** %$v5
    %t14 = load %Object** %$v3
    %t15 = load %Object** %$v5
    %t16 = call %Object*
@"[j]java.lang.StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
[Invokevirtual(com/mycompany/myapp/Account,java/lang/StringBuilder)]"(%Env* %p0,
%Object* %t14, %Object* %t15)
    %t17 = load %Object** %$v3
    %t18 = load i32* %$i0
    %t19 = call %Object*
@"[j]java.lang.StringBuilder.append(I)Ljava/lang/StringBuilder;[Invokevirtual(co
m/mycompany/myapp/Account,java/lang/StringBuilder)]"(%Env* %p0, %Object* %t17,
i32 %t18)
    %t20 = call %Object* @"[j]str__7D_00[ldcstring]"(%Env* %p0)
    store %Object* %t20, %Object** %$v6
    %t21 = load %Object** %$v3
    %t22 = load %Object** %$v6
    %t23 = call %Object*
@"[j]java.lang.StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
[Invokevirtual(com/mycompany/myapp/Account,java/lang/StringBuilder)]"(%Env* %p0,
%Object* %t21, %Object* %t22)
    %t24 = load %Object** %$v3
    %t25 = call %Object*
@"[j]java.lang.StringBuilder.toString()Ljava/lang/String;[Invokevirtual(com/myco
mpany/myapp/Account,java/lang/StringBuilder)]"(%Env* %p0, %Object* %t24)
    store %Object* %t25, %Object** %$r1
    %t26 = load %Object** %$r1, !dbg !{i32 60, i32 0, !11, null}
```

```
    ret %Object* %t26, !dbg !{i32 60, i32 0, !11, null}
}

define weak i8
@"[J]com.mycompany.myapp.Account.equals(Ljava/lang/Object;)Z"(%Env* %p0,
%Object* %p1, %Object* %p2) nounwind noinline optsize {
label0:
    %r0 = alloca %Object*
    %r1 = alloca %Object*
    %$r3 = alloca %Object*
    %$r4 = alloca %Object*
    %r2 = alloca %Object*
    %$i0 = alloca i32
    %$i1 = alloca i32
    %$r5 = alloca %Object*
    %$r6 = alloca %Object*
    %$z0 = alloca i32
    call void @"checkso"()
    store %Object* %p1, %Object** %r0, !dbg !{i32 68, i32 0, !13, null}
    store %Object* %p2, %Object** %r1, !dbg !{i32 68, i32 0, !13, null}
    %t0 = load %Object** %r0, !dbg !{i32 68, i32 0, !13, null}
    %t1 = load %Object** %r1, !dbg !{i32 68, i32 0, !13, null}
    %t2 = icmp ne %Object* %t0, %t1, !dbg !{i32 68, i32 0, !13, null}
    br i1 %t2, label %label2, label %label1, !dbg !{i32 68, i32 0, !13, null}
label1:
    %t3 = trunc i32 1 to i8, !dbg !{i32 68, i32 0, !13, null}
    ret i8 %t3, !dbg !{i32 68, i32 0, !13, null}
label2:
    %t4 = load %Object** %r1, !dbg !{i32 69, i32 0, !13, null}
    %t5 = icmp eq %Object* %t4, null, !dbg !{i32 69, i32 0, !13, null}
    br i1 %t5, label %label4, label %label3, !dbg !{i32 69, i32 0, !13, null}
label3:
    %t6 = load %Object** %r0, !dbg !{i32 69, i32 0, !13, null}
    %t7 = call %Object* @"intrinsics.java_lang_Object_getClass"(%Env* %p0,
%Object* %t6), !dbg !{i32 69, i32 0, !13, null}
    store %Object* %t7, %Object** %$r3, !dbg !{i32 69, i32 0, !13, null}
    %t8 = load %Object** %r1, !dbg !{i32 69, i32 0, !13, null}
    %t9 = call %Object* @"intrinsics.java_lang_Object_getClass"(%Env* %p0,
%Object* %t8), !dbg !{i32 69, i32 0, !13, null}
    store %Object* %t9, %Object** %$r4, !dbg !{i32 69, i32 0, !13, null}
    %t10 = load %Object** %$r3, !dbg !{i32 69, i32 0, !13, null}
    %t11 = load %Object** %$r4, !dbg !{i32 69, i32 0, !13, null}
    %t12 = icmp eq %Object* %t10, %t11, !dbg !{i32 69, i32 0, !13, null}
    br i1 %t12, label %label5, label %label4, !dbg !{i32 69, i32 0, !13, null}
label4:
    %t13 = trunc i32 0 to i8, !dbg !{i32 69, i32 0, !13, null}
    ret i8 %t13, !dbg !{i32 69, i32 0, !13, null}
label5:
    %t14 = load %Object** %r1, !dbg !{i32 71, i32 0, !13, null}
    %t15 = call %Object*
@"[j]com.mycompany.myapp.Account[Checkcast(com/mycompany/myapp/Account)]"(%Env*
%p0, %Object* %t14), !dbg !{i32 71, i32 0, !13, null}
    store %Object* %t15, %Object** %r2, !dbg !{i32 71, i32 0, !13, null}
    %t16 = load %Object** %r0, !dbg !{i32 73, i32 0, !13, null}
    %t17 = call i32 @"[j]com.mycompany.myapp.Account.id(I)[get]"(%Env* %p0,
%Object* %t16), !dbg !{i32 73, i32 0, !13, null}
    store i32 %t17, i32* %$i0, !dbg !{i32 73, i32 0, !13, null}
    %t18 = load %Object** %r2, !dbg !{i32 73, i32 0, !13, null}
    %t19 = call i32 @"[j]com.mycompany.myapp.Account.id(I)[get]"(%Env* %p0,
%Object* %t18), !dbg !{i32 73, i32 0, !13, null}
    store i32 %t19, i32* %$i1, !dbg !{i32 73, i32 0, !13, null}
    %t20 = load i32* %$i0, !dbg !{i32 73, i32 0, !13, null}
    %t21 = load i32* %$i1, !dbg !{i32 73, i32 0, !13, null}
    %t22 = icmp eq i32 %t20, %t21, !dbg !{i32 73, i32 0, !13, null}
    br i1 %t22, label %label7, label %label6, !dbg !{i32 73, i32 0, !13, null}
label6:
    %t23 = trunc i32 0 to i8, !dbg !{i32 73, i32 0, !13, null}
```

```
    ret i8 %t23, !dbg !{i32 73, i32 0, !13, null}
label7:
    %t24 = load %Object** %r0, !dbg !{i32 74, i32 0, !13, null}
    %t25 = call %Object*
@"[j]com.mycompany.myapp.Account.name(Ljava/lang/String;)[get]"(%Env* %p0,
%Object* %t24), !dbg !{i32 74, i32 0, !13, null}
    store %Object* %t25, %Object** %$r5, !dbg !{i32 74, i32 0, !13, null}
    %t26 = load %Object** %r2, !dbg !{i32 74, i32 0, !13, null}
    %t27 = call %Object*
@"[j]com.mycompany.myapp.Account.name(Ljava/lang/String;)[get]"(%Env* %p0,
%Object* %t26), !dbg !{i32 74, i32 0, !13, null}
    store %Object* %t27, %Object** %$r6, !dbg !{i32 74, i32 0, !13, null}
    %t28 = load %Object** %$r5, !dbg !{i32 74, i32 0, !13, null}
    %t29 = load %Object** %$r6, !dbg !{i32 74, i32 0, !13, null}
    %t30 = call i8
@"[j]java.util.Objects.equals(Ljava/lang/Object;Ljava/lang/Object;)Z[Invokestati
c(com/mycompany/myapp/Account)]"(%Env* %p0, %Object* %t28, %Object* %t29), !dbg
!{i32 74, i32 0, !13, null}
    %t31 = sext i8 %t30 to i32, !dbg !{i32 74, i32 0, !13, null}
    store i32 %t31, i32* %$z0, !dbg !{i32 74, i32 0, !13, null}
    %t32 = load i32* %$z0, !dbg !{i32 74, i32 0, !13, null}
    %t33 = trunc i32 %t32 to i8, !dbg !{i32 74, i32 0, !13, null}
    ret i8 %t33, !dbg !{i32 74, i32 0, !13, null}
}


define weak i32 @"[J]com.mycompany.myapp.Account.hashCode()I"(%Env* %p0,
%Object* %p1) nounwind noinline optsize {
label0:
    %r0 = alloca %Object*
    %$r1 = alloca %Object*
    %$i1 = alloca i32
    %i0 = alloca i32
    %$i2 = alloca i32
    call void @"checkso"()
    store %Object* %p1, %Object** %r0, !dbg !{i32 79, i32 0, !15, null}
    %t0 = load %Object** %r0, !dbg !{i32 79, i32 0, !15, null}
    %t1 = call %Object*
@"[j]com.mycompany.myapp.Account.name(Ljava/lang/String;)[get]"(%Env* %p0,
%Object* %t0), !dbg !{i32 79, i32 0, !15, null}
    store %Object* %t1, %Object** %$r1, !dbg !{i32 79, i32 0, !15, null}
    %t2 = load %Object** %$r1, !dbg !{i32 79, i32 0, !15, null}
    %t3 = icmp eq %Object* %t2, null, !dbg !{i32 79, i32 0, !15, null}
    br i1 %t3, label %label2, label %label1, !dbg !{i32 79, i32 0, !15, null}
label1:
    %t4 = load %Object** %r0, !dbg !{i32 79, i32 0, !15, null}
    %t5 = call %Object*
@"[j]com.mycompany.myapp.Account.name(Ljava/lang/String;)[get]"(%Env* %p0,
%Object* %t4), !dbg !{i32 79, i32 0, !15, null}
    store %Object* %t5, %Object** %$r1, !dbg !{i32 79, i32 0, !15, null}
    %t6 = load %Object** %$r1, !dbg !{i32 79, i32 0, !15, null}
    %t7 = call i8 @"checknull"(%Env* %p0, %Object* %t6), !dbg !{i32 79, i32 0,
!15, null}
    %t8 = call i32
@"[j]java.lang.String.hashCode()I[Invokevirtual(com/mycompany/myapp/Account,java
/lang/String)]"(%Env* %p0, %Object* %t6), !dbg !{i32 79, i32 0, !15, null}
    store i32 %t8, i32* %$i1, !dbg !{i32 79, i32 0, !15, null}
    br label %label3, !dbg !{i32 79, i32 0, !15, null}
label2:
    store i32 0, i32* %$i1, !dbg !{i32 79, i32 0, !15, null}
    br label %label3
label3:
    %t9 = load i32* %$i1, !dbg !{i32 79, i32 0, !15, null}
    store i32 %t9, i32* %i0, !dbg !{i32 79, i32 0, !15, null}
    %t11 = load i32* %i0, !dbg !{i32 80, i32 0, !15, null}
    %t10 = mul i32 31, %t11, !dbg !{i32 80, i32 0, !15, null}
    store i32 %t10, i32* %$i1, !dbg !{i32 80, i32 0, !15, null}
```

```
    %t12 = load %Object** %r0, !dbg !{i32 80, i32 0, !15, null}
    %t13 = call i32 @"[j]com.mycompany.myapp.Account.id(I)[get]"(%Env* %p0,
%Object* %t12), !dbg !{i32 80, i32 0, !15, null}
    store i32 %t13, i32* %$i2, !dbg !{i32 80, i32 0, !15, null}
    %t15 = load i32* %$i1, !dbg !{i32 80, i32 0, !15, null}
    %t16 = load i32* %$i2, !dbg !{i32 80, i32 0, !15, null}
    %t14 = add i32 %t15, %t16, !dbg !{i32 80, i32 0, !15, null}
    store i32 %t14, i32* %i0, !dbg !{i32 80, i32 0, !15, null}
    %t17 = load i32* %i0, !dbg !{i32 81, i32 0, !15, null}
    ret i32 %t17, !dbg !{i32 81, i32 0, !15, null}
}
```

Відповідний LLVM AS-код

```
    .section    __TEXT,__text,regular,pure_instructions
    .build_version iossimulator, 14, 0
    .section    __DWARF,__debug_info,regular,debug
Lsection_info:
    .section    __DWARF,__debug_abbrev,regular,debug
Lsection_abbrev:
    .section    __DWARF,__debug_line,regular,debug
Lsection_line:
    .section    __DWARF,__debug_str,regular,debug
Linfo_string:
    .section    __DWARF,__debug_loc,regular,debug
Lsection_debug_loc:
    .section    __DWARF,__debug_ranges,regular,debug
Ldebug_range:
    .section    __TEXT,__text,regular,pure_instructions

    .globl "_[j]com.mycompany.myapp.Account.name(Ljava/lang/String;)[get]"
    .align 2
"_[j]com.mycompany.myapp.Account.name(Ljava/lang/String;)[get]": ;
@"[j]com.mycompany.myapp.Account.name(Ljava/lang/String;)[get]"
    .cfi_startproc
; BB#0:                                 ; %label0
    ldr    x0, [x1, #16]
    ret
    .cfi_endproc

    .globl "_[j]com.mycompany.myapp.Account.id(I)[get]"
    .align 2
"_[j]com.mycompany.myapp.Account.id(I)[get]": ;
@"[j]com.mycompany.myapp.Account.id(I)[get]"
    .cfi_startproc
; BB#0:                                 ; %label0
    ldr    w0, [x1, #24]
    ret
    .cfi_endproc

    .section    __TEXT,__textcoal_nt,coalesced,pure_instructions
    .globl "_[J]com.mycompany.myapp.Account.<init>(Ljava/lang/String;I)V"
    .weak_definition
"_[J]com.mycompany.myapp.Account.<init>(Ljava/lang/String;I)V"
    .align 2
"_[J]com.mycompany.myapp.Account.<init>(Ljava/lang/String;I)V": ;
@"[J]com.mycompany.myapp.Account.<init>(Ljava/lang/String;I)V"
Lfunc_begin4:
    .file  1 "Account.java"
    .loc   1 37 0                   ; Account.java:37:0
    .cfi_startproc
; BB#0:                                 ; %label0
    stp    x20, x19, [sp, #-32]!
    stp    x29, x30, [sp, #16]
```

```
    add    x29, sp, #16            ; =16
    sub    sp, sp, #16             ; =16
Ltmp10:
    .cfi_def_cfa w29, 16
Ltmp11:
    .cfi_offset w30, -8
Ltmp12:
    .cfi_offset w29, -16
Ltmp13:
    .cfi_offset w19, -24
Ltmp14:
    .cfi_offset w20, -32
    mov     x19, x2
    ; InlineAsm Start
    sub    x9, sp, #16, lsl #12    ; =65536
    ldr     x9, [x9]
    ; InlineAsm End
    .loc   1 37 0 prologue_end     ; Account.java:37:0
Ltmp15:
    str    x1, [sp, #8]
    str    w3, [sp, #4]
    bl "_[J]java.lang.Object.<init>()V"
    .loc   1 38 0                  ; Account.java:38:0
    ldr    x8, [sp, #8]
    str    x19, [x8, #16]
    dmb    ish
    .loc   1 39 0                  ; Account.java:39:0
    ldr    w8, [sp, #4]
    ldr    x9, [sp, #8]
    str    w8, [x9, #24]
    dmb    ish
    .loc   1 40 0                  ; Account.java:40:0
    sub    sp, x29, #16            ; =16
    ldp    x29, x30, [sp, #16]
    ldp    x20, x19, [sp], #32
    ret
Ltmp16:
Lfunc_end4:
"L_[J]com.mycompany.myapp.Account.<init>(Ljava/lang/String;I)V_end":

    .cfi_endproc

    .section    __TEXT,__textcoal_nt,coalesced,pure_instructions
    .globl "_[J]com.mycompany.myapp.Account.toString()Ljava/lang/String;"
    .weak_definition
"_[J]com.mycompany.myapp.Account.toString()Ljava/lang/String;"
    .align 2
"_[J]com.mycompany.myapp.Account.toString()Ljava/lang/String;": ;
@"[J]com.mycompany.myapp.Account.toString()Ljava/lang/String;"
Lfunc_begin9:
    .loc   1 60 0                  ; Account.java:60:0
    .cfi_startproc
; BB#0:                                 ; %label0
    stp    x22, x21, [sp, #-48]!
    stp    x20, x19, [sp, #16]
    stp    x29, x30, [sp, #32]
    add    x29, sp, #32            ; =32
Ltmp37:
    .cfi_def_cfa w29, 16
Ltmp38:
    .cfi_offset w30, -8
Ltmp39:
    .cfi_offset w29, -16
Ltmp40:
    .cfi_offset w19, -24
Ltmp41:
    .cfi_offset w20, -32
```

```
Ltmp42:
    .cfi_offset w21, -40
Ltmp43:
    .cfi_offset w22, -48
    ; InlineAsm Start
    sub    x9, sp, #16, lsl #12    ; =65536
    ldr     x9, [x9]
    ; InlineAsm End
    .loc   1 60 0 prologue_end     ; Account.java:60:0
Ltmp44:
    ldr    x19, [x1, #16]
    ldr    w20, [x1, #24]
    mov     x21, x0
                                        ; kill: X0<def> X21<kill>
    bl "_[j]java.lang.StringBuilder[allocator][clinit]"
    mov     x22, x0
    mov     x0, x21
    mov     x1, x22
    bl "_[J]java.lang.StringBuilder.<init>()V"
    mov     x0, x21
    bl "_[j]str_Account_7Bname_3D_27_00[ldcstring]"
    mov     x2, x0
    mov     x0, x21
    mov     x1, x22
    bl
"_[J]java.lang.StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
"
    mov     x0, x21
    mov     x1, x22
    mov     x2, x19
    bl
"_[J]java.lang.StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
"
    mov     x0, x21
    bl "_[j]str__27_2C_20id_3D_00[ldcstring]"
    mov     x2, x0
    mov     x0, x21
    mov     x1, x22
    bl
"_[J]java.lang.StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
"
    mov     x0, x21
    mov     x1, x22
    mov     w2, w20
    bl "_[J]java.lang.StringBuilder.append(I)Ljava/lang/StringBuilder;"
    mov     x0, x21
    bl "_[j]str__7D_00[ldcstring]"
    mov     x2, x0
    mov     x0, x21
    mov     x1, x22
    bl
"_[J]java.lang.StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
"
    mov     x0, x21
    mov     x1, x22
    bl "_[J]java.lang.StringBuilder.toString()Ljava/lang/String;"
    ldp    x29, x30, [sp, #32]
    ldp    x20, x19, [sp, #16]
    ldp    x22, x21, [sp], #48
    ret
Ltmp45:
Lfunc_end9:
"L_[J]com.mycompany.myapp.Account.toString()Ljava/lang/String;_end":

    .cfi_endproc
```

```
    .globl "_[J]com.mycompany.myapp.Account.equals(Ljava/lang/Object;)Z"
    .weak_definition
"_[J]com.mycompany.myapp.Account.equals(Ljava/lang/Object;)Z"
    .align 2
"_[J]com.mycompany.myapp.Account.equals(Ljava/lang/Object;)Z": ;
@"[J]com.mycompany.myapp.Account.equals(Ljava/lang/Object;)Z"
Lfunc_begin14:
    .loc   1 68 0                  ; Account.java:68:0
    .cfi_startproc
; BB#0:                                 ; %label0
    stp    x20, x19, [sp, #-32]!
    stp    x29, x30, [sp, #16]
    add    x29, sp, #16            ; =16
    sub    sp, sp, #32             ; =32
Ltmp56:
    .cfi_def_cfa w29, 16
Ltmp57:
    .cfi_offset w30, -8
Ltmp58:
    .cfi_offset w29, -16
Ltmp59:
    .cfi_offset w19, -24
Ltmp60:
    .cfi_offset w20, -32
    mov     x19, x0
    ; InlineAsm Start
    sub    x9, sp, #16, lsl #12    ; =65536
    ldr     x9, [x9]
    ; InlineAsm End
    .loc   1 68 0 prologue_end     ; Account.java:68:0
Ltmp61:
    stp    x2, x1, [sp, #16]
    cmp     x1, x2
    b.eq   LBB14_6
; BB#1:                                 ; %label2
    cbz    x2, LBB14_5
; BB#2:                                 ; %label3
    .loc   1 69 0                  ; Account.java:69:0
    ldr     x8, [x1]
    ldr    x1, [sp, #16]
    ldr     x9, [x1]
    cmp     x8, x9
    b.ne   LBB14_5
; BB#3:                                 ; %label5
    .loc   1 71 0                  ; Account.java:71:0
    mov     x0, x19
    bl "_[j]com.mycompany.myapp.Account[checkcast]"
    .loc   1 73 0                  ; Account.java:73:0
    ldr    x8, [sp, #24]
    .loc   1 71 0                  ; Account.java:71:0
    str    x0, [sp, #8]
    .loc   1 73 0                  ; Account.java:73:0
    ldr    w9, [x8, #24]
    ldr    w10, [x0, #24]
    cmp     w9, w10
    b.ne   LBB14_5
; BB#4:                                 ; %label7
    .loc   1 74 0                  ; Account.java:74:0
    ldr    x1, [x8, #16]
    ldr    x2, [x0, #16]
    mov     x0, x19
    bl
"_[j]java.util.Objects.equals(Ljava/lang/Object;Ljava/lang/Object;)Z[clinit]"
    b  LBB14_7
LBB14_5:
    mov     w0, wzr
    b  LBB14_7
```

```
LBB14_6:
    orr    w0, wzr, #0x1
LBB14_7:                                ; %label1
    .loc   1 68 0                   ; Account.java:68:0
    sub    sp, x29, #16             ; =16
    ldp    x29, x30, [sp, #16]
    ldp    x20, x19, [sp], #32
    ret
Ltmp62:
Lfunc_end14:
"L_[J]com.mycompany.myapp.Account.equals(Ljava/lang/Object;)Z_end":

    .cfi_endproc


    .globl "_[J]com.mycompany.myapp.Account.hashCode()I"
    .weak_definition   "_[J]com.mycompany.myapp.Account.hashCode()I"
    .align 2
"_[J]com.mycompany.myapp.Account.hashCode()I": ;
@"[J]com.mycompany.myapp.Account.hashCode()I"
Lfunc_begin16:
    .loc   1 79 0                   ; Account.java:79:0
    .cfi_startproc
; BB#0:                                 ; %label0
    stp    x29, x30, [sp, #-16]!
    mov     x29, sp
    sub    sp, sp, #16              ; =16
Ltmp64:
    .cfi_def_cfa w29, 16
Ltmp65:
    .cfi_offset w30, -8
Ltmp66:
    .cfi_offset w29, -16
    ; InlineAsm Start
    sub    x9, sp, #16, lsl #12     ; =65536
    ldr     x9, [x9]
    ; InlineAsm End
    .loc   1 79 0 prologue_end      ; Account.java:79:0
Ltmp67:
    str    x1, [sp, #8]
    ldr    x1, [x1, #16]
    cbz    x1, LBB16_2
; BB#1:                                 ; %label1
    ldrb    wzr, [x1]
    bl "_[J]java.lang.String.hashCode()I"
    lsl    w8, w0, #5
    sub     w8, w8, w0
    b  LBB16_3
LBB16_2:
    mov     w8, wzr
LBB16_3:                                ; %label3
    .loc   1 80 0                   ; Account.java:80:0
    ldr    x9, [sp, #8]
    ldr    w9, [x9, #24]
    add     w0, w9, w8
    .loc   1 81 0                   ; Account.java:81:0
    mov     sp, x29
    ldp    x29, x30, [sp], #16
    ret
Ltmp68:
Lfunc_end16:
"L_[J]com.mycompany.myapp.Account.hashCode()I_end":

    .cfi_endproc
```

ДОДАТОК Е

(обов’язковий)

ІЛЮСТРАТИВНА ЧАСТИНА

МЕТОДИ ТА ПРОГРАМНІ ЗАСОБИ ПІДТРИМКИ JAVA КОДУ НА ПЛАТФОРМІ APPLE IOS

Рисунок Е.1 – Слайд презентації №1

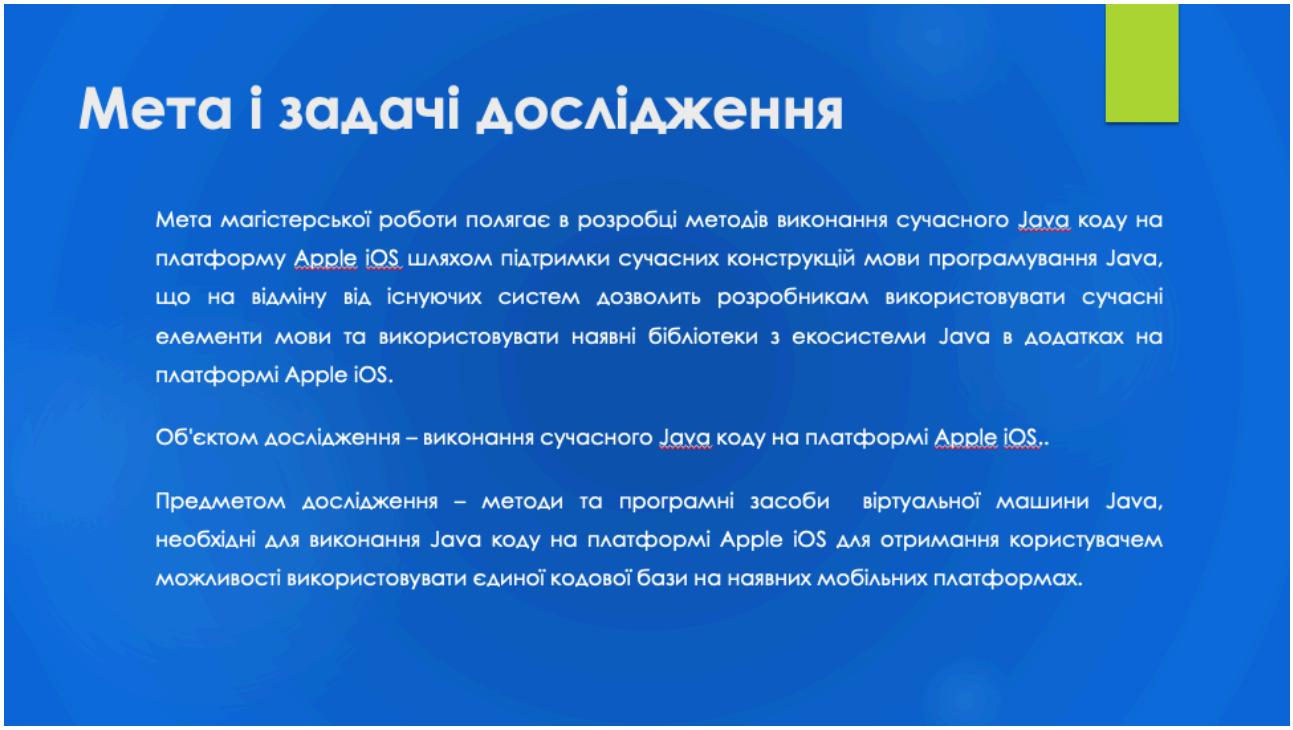

Рисунок Е.2 – Слайд презентації №2

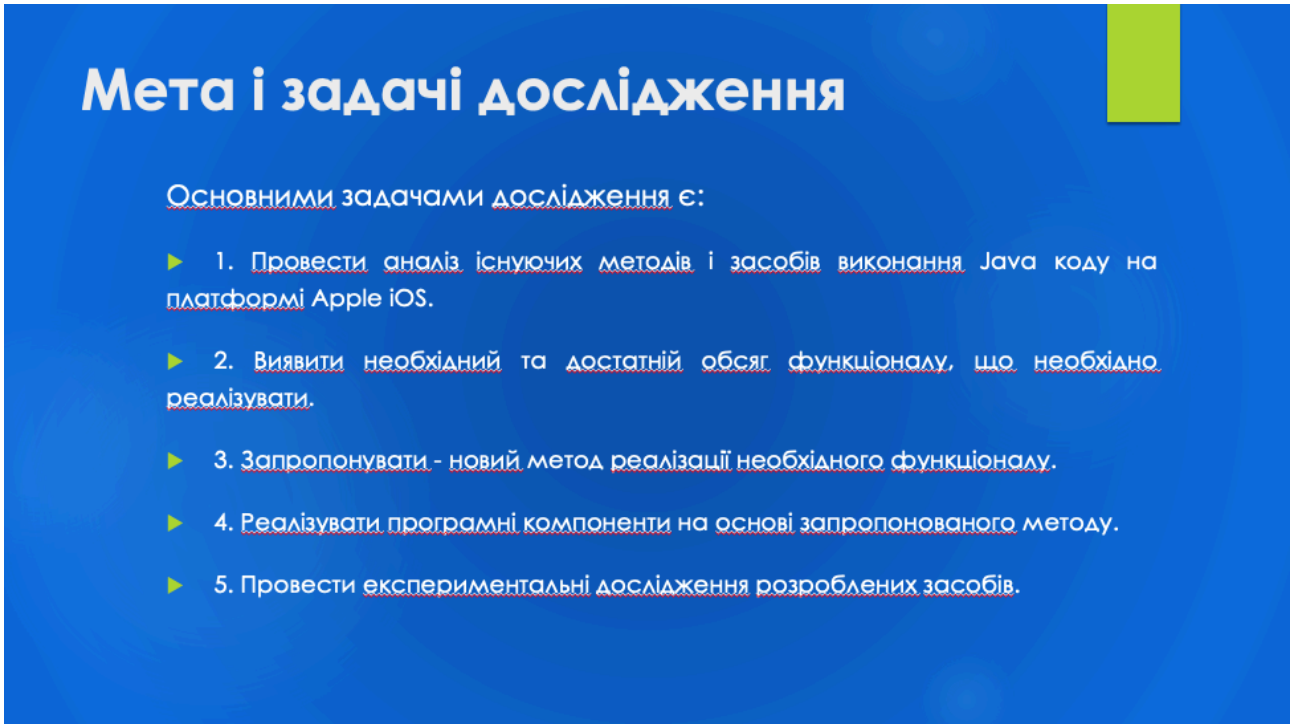

Рисунок Е.3 – Слайд презентації №3

Рисунок Е.4 – Слайд презентації №4

Рисунок Е.5 – Слайд презентації №5

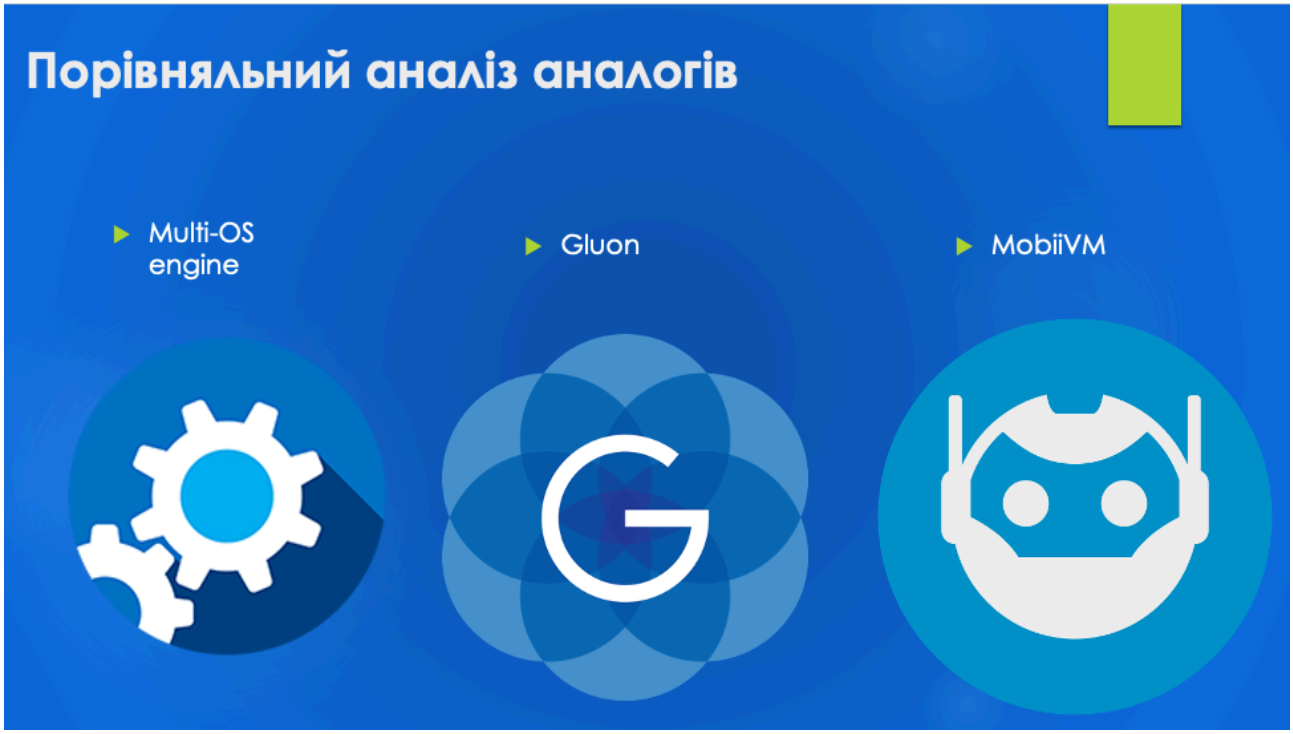

Рисунок Е.6 – Слайд презентації №6

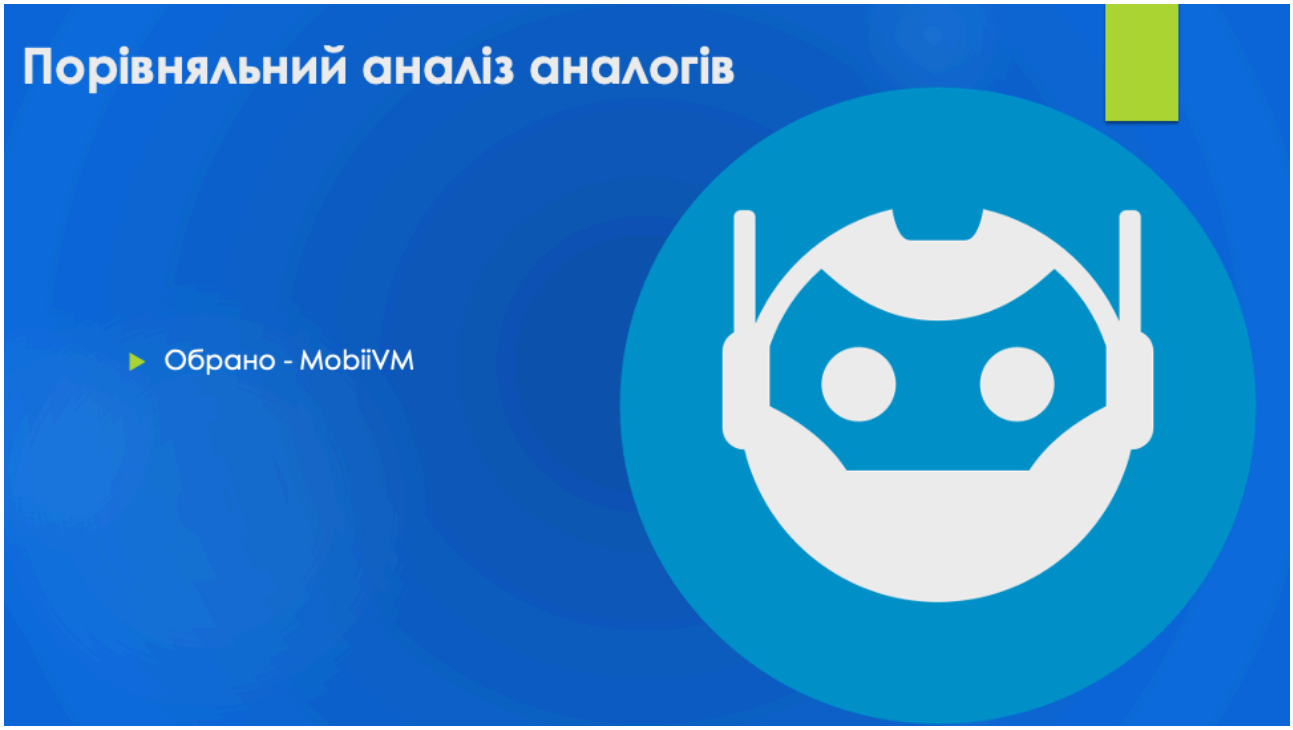

Рисунок Е.7 – Слайд презентації №7

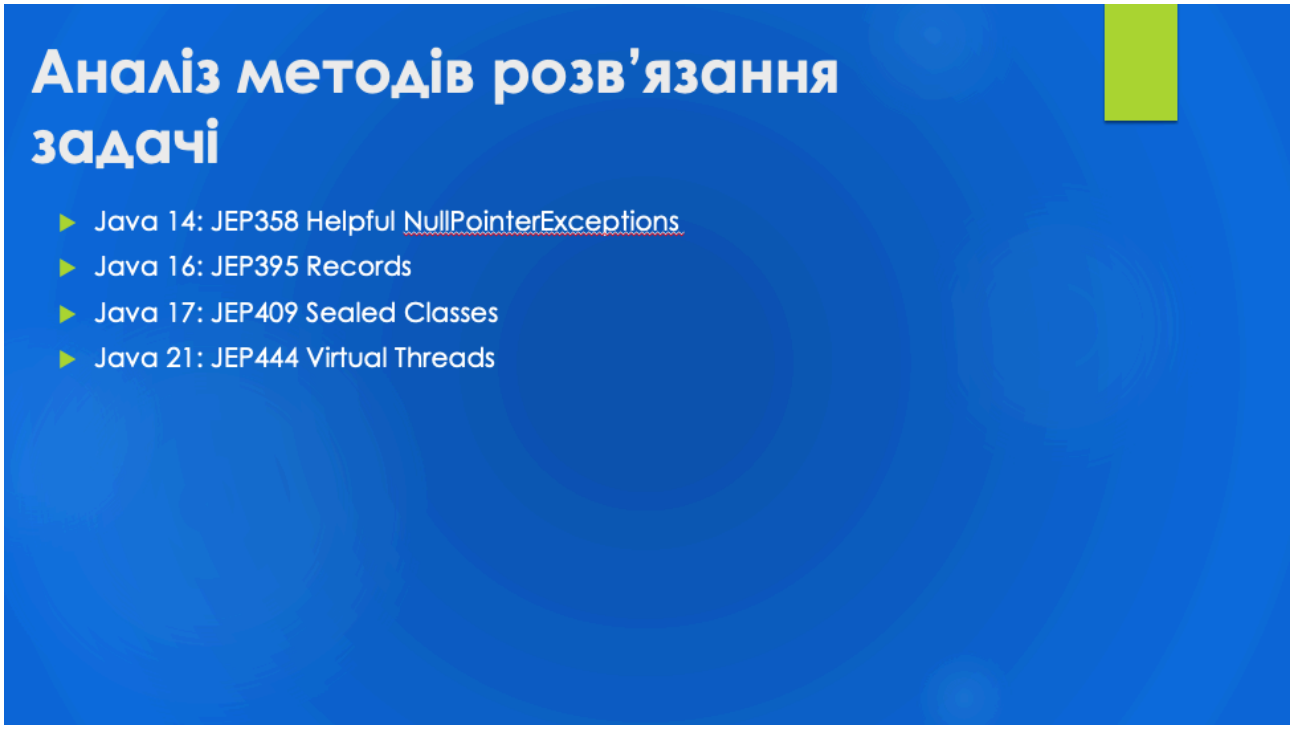

Рисунок Е.8 – Слайд презентації №8

Рисунок Е.9 – Слайд презентації №9

Рисунок Е.10 – Слайд презентації №10

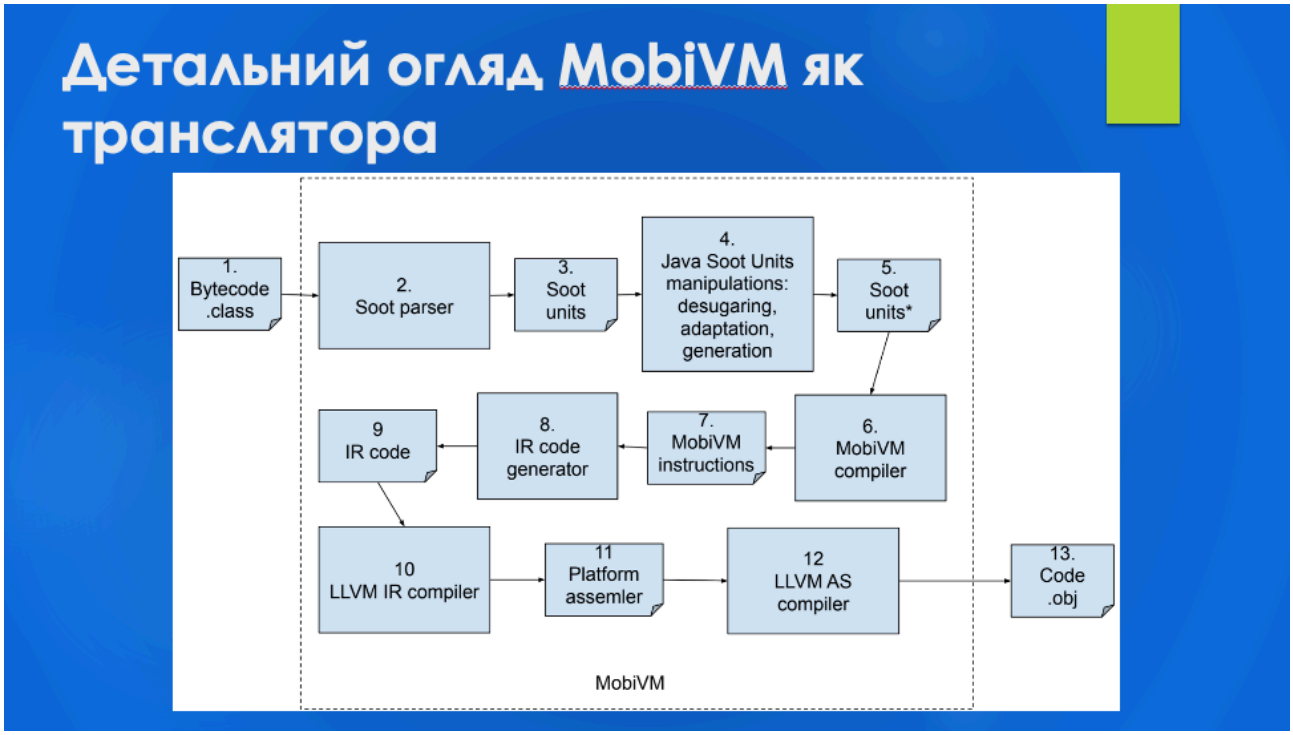

Рисунок Е.11 – Слайд презентації №11

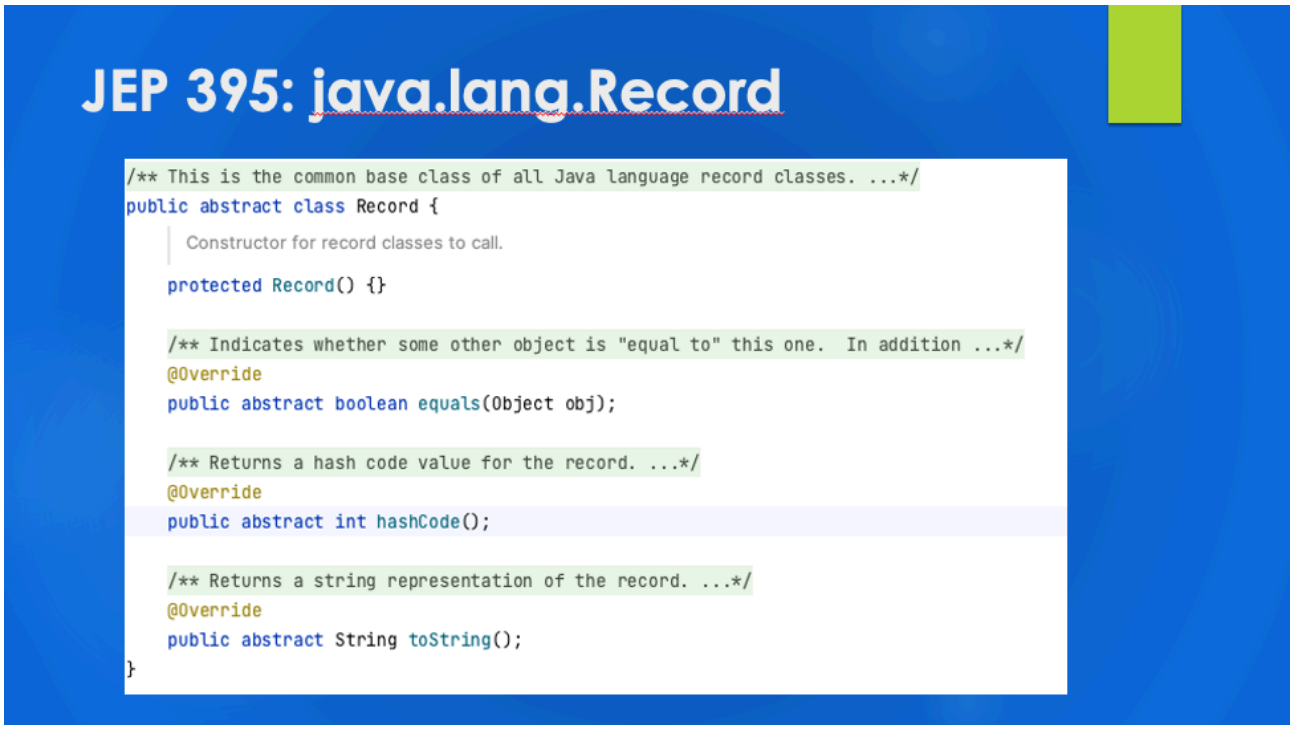

Рисунок Е.12 – Слайд презентації №12

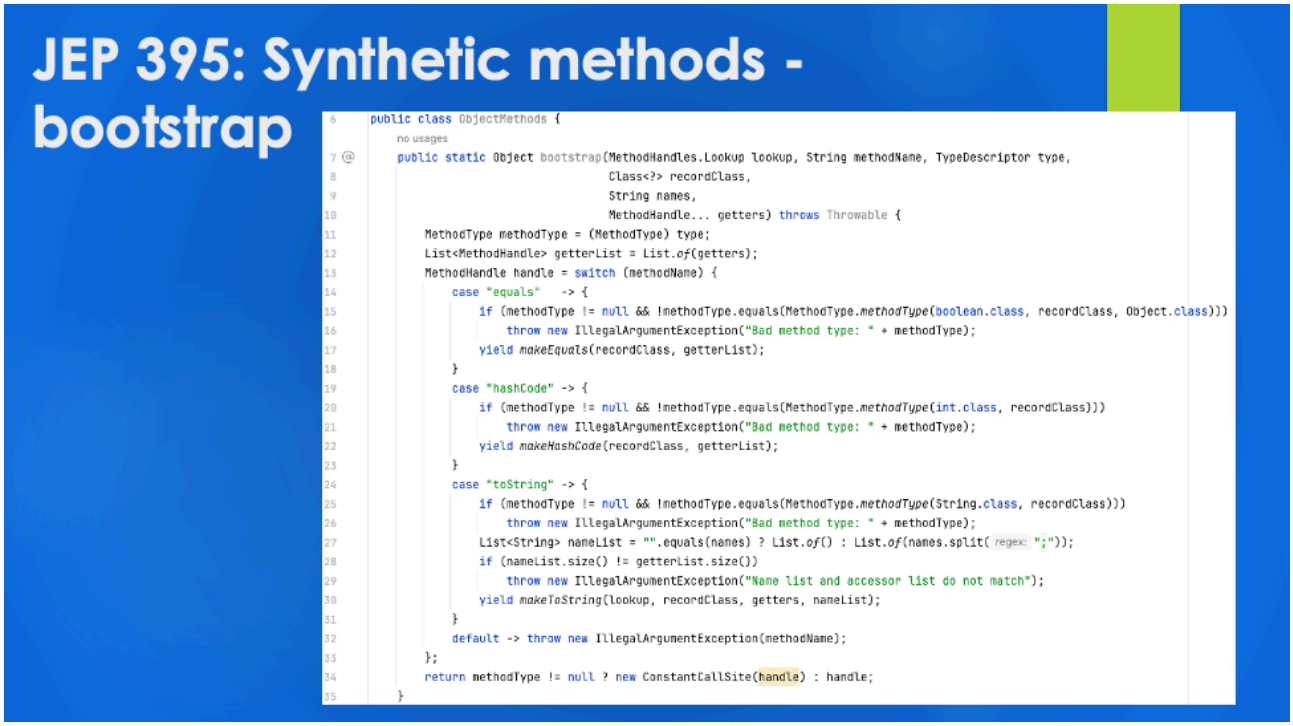

Рисунок Е.13 – Слайд презентації №13

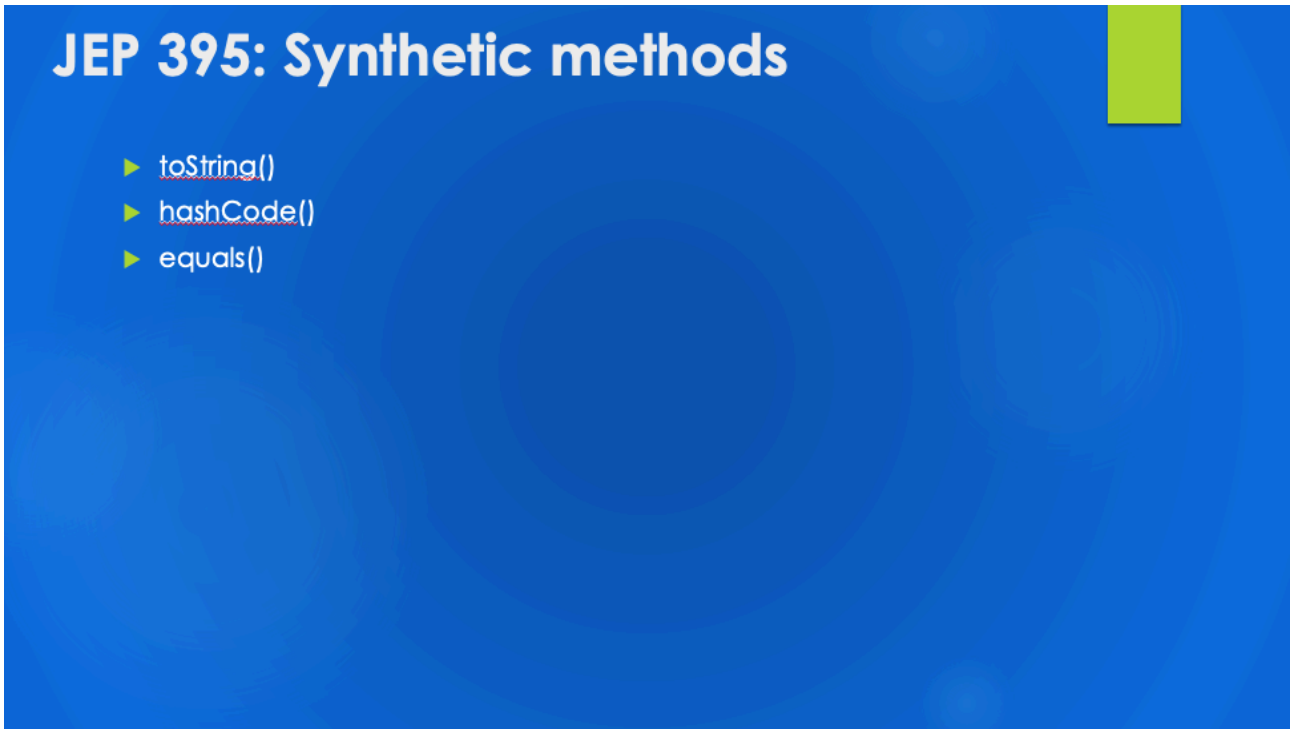

Рисунок Е.14 – Слайд презентації №14

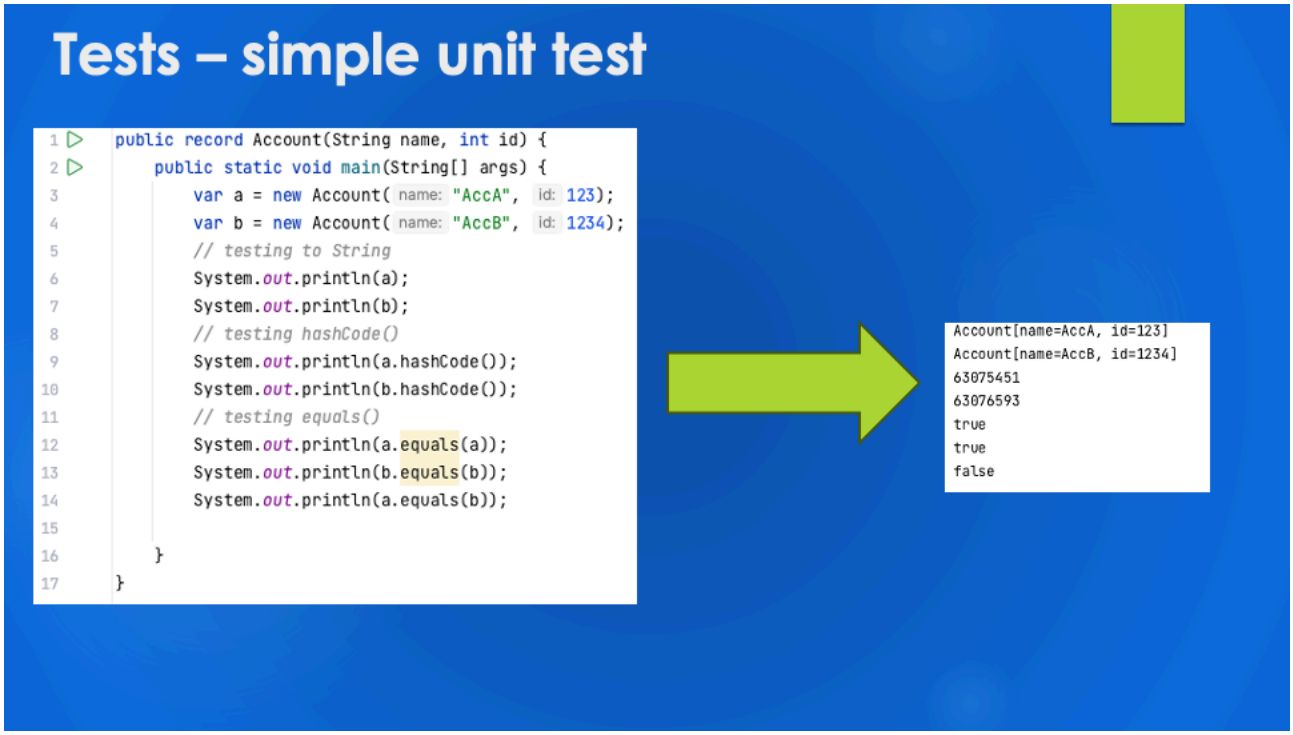

Рисунок Е.15 – Слайд презентації №15

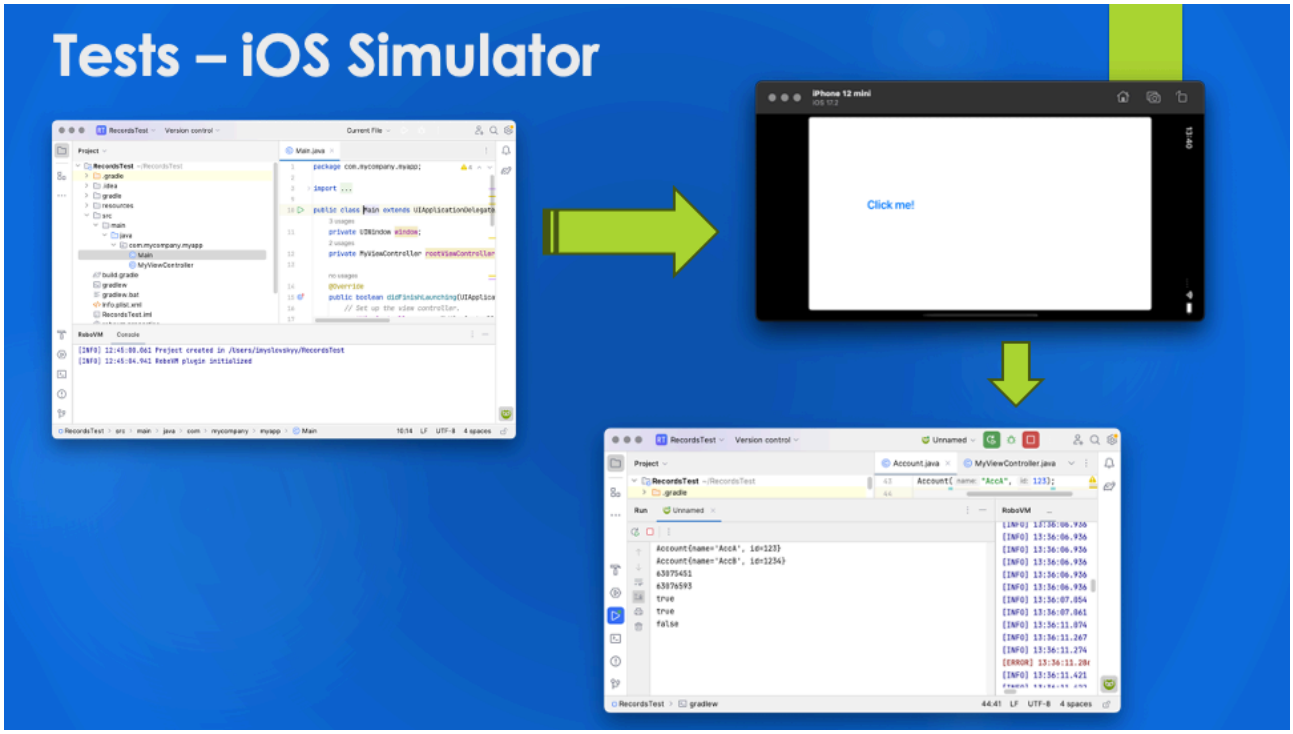

Рисунок Е.16 – Слайд презентації №16

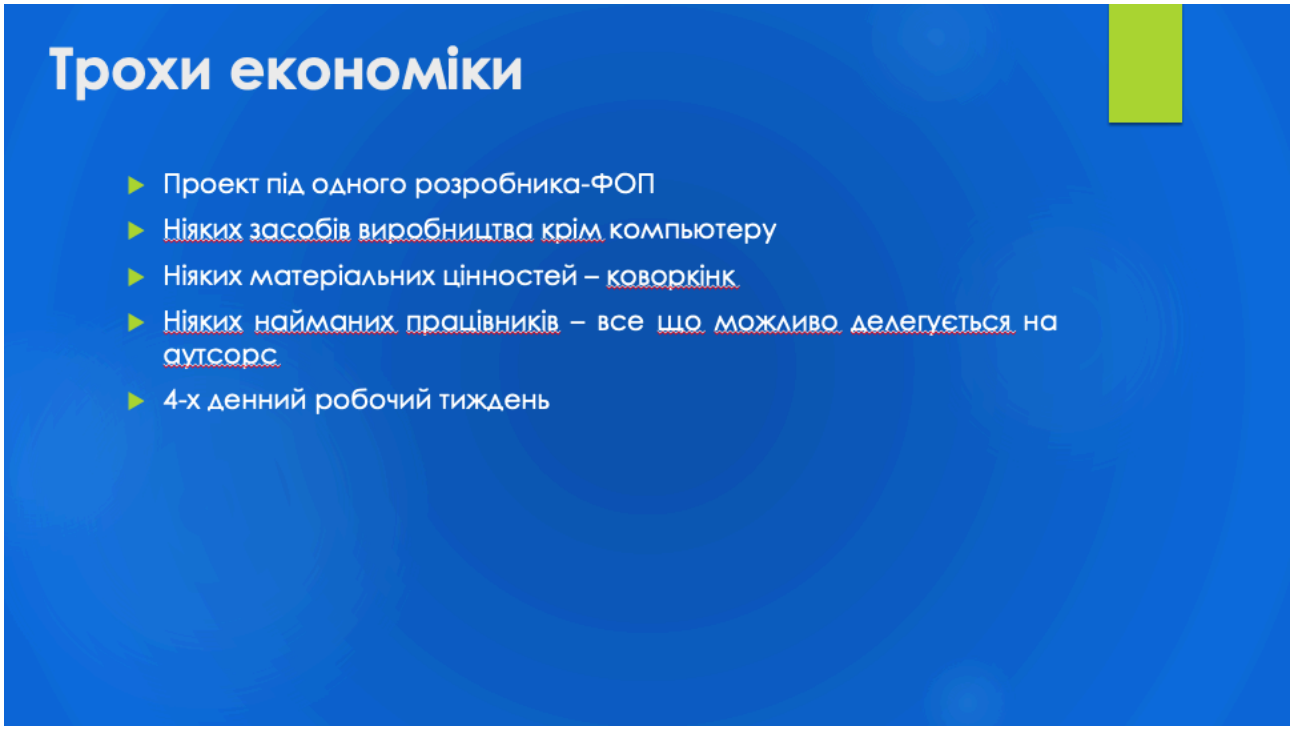

Рисунок Е.17 – Слайд презентації №17

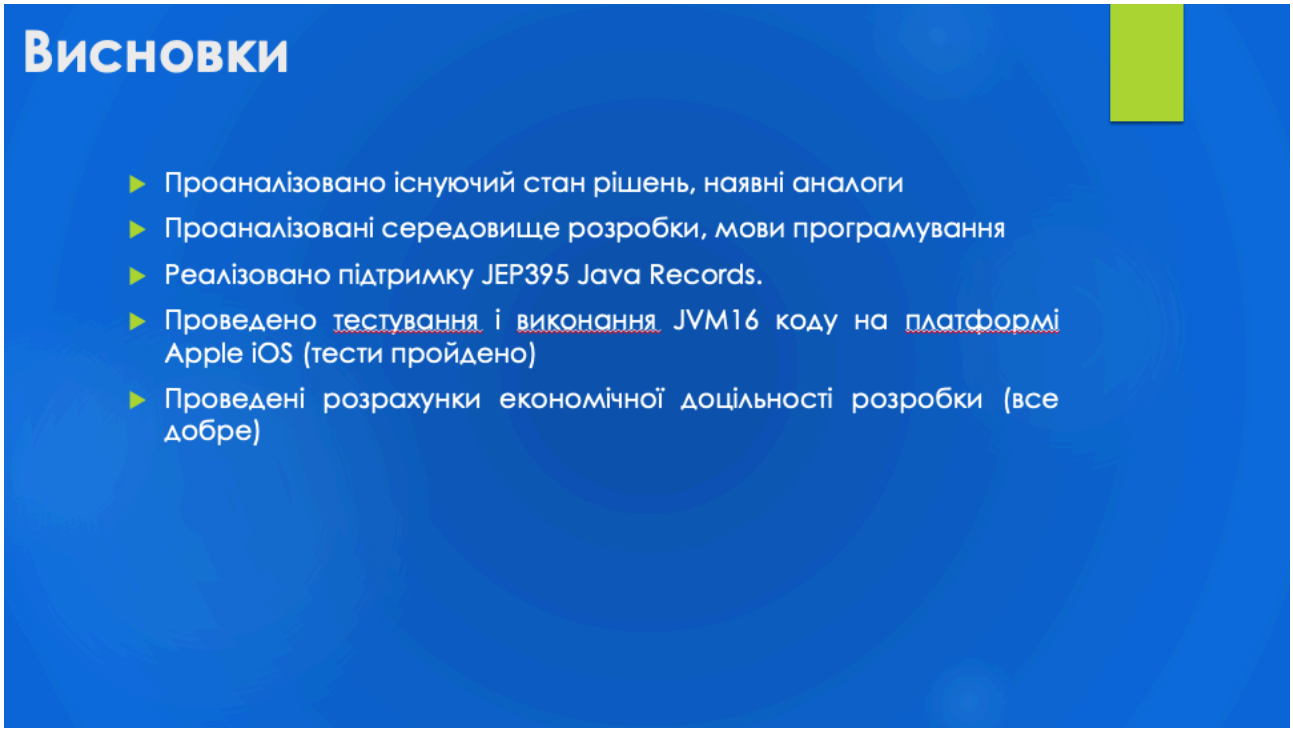

Рисунок Е.18 – Слайд презентації №18

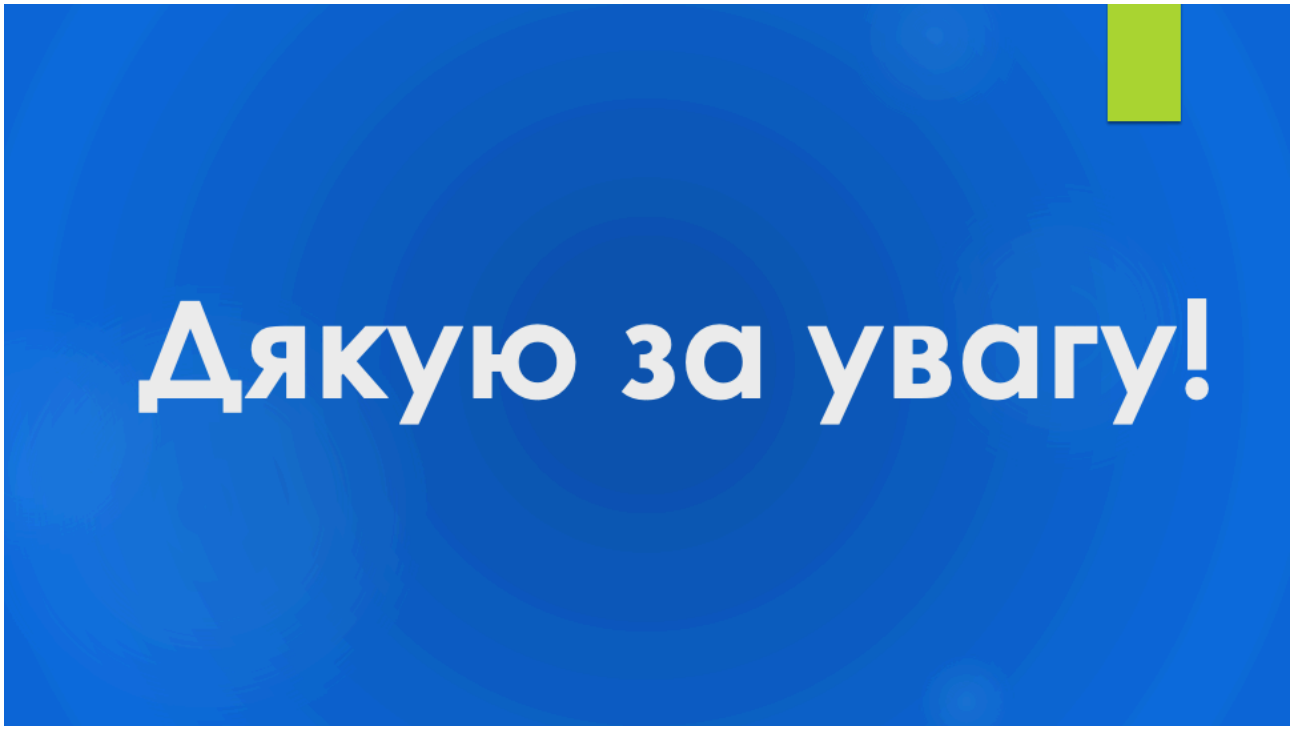

Рисунок Е.19 – Слайд презентації №19