Зміст

Система оцінювання

№	Тема	К-ть балів
1.	*Захист принаймні одного завдання з роботи*	1
2.	Завдання в тексті	0,8*
3.	Практичні завдання	2,8*
4.	*Здача звіту*	0,4
	Всього	**5**

Тема 3. Об’єктно-орієнтоване програмування в мові Kotlin

У даній темі будуть розглянуті основні питання стосовно створення об’єктно-орієнтованого коду мовою програмування Kotlin:

- класи та об'єкти в мові Kotlin;
- основні відношення між класами;
- узагальнене програмування та параметризовані типи.

 **Доповідь**	1. *Породжуючі шаблони проектування в Kotlin* 2. *Структурні шаблони проектування в Kotlin* 3. *Поведінкові шаблони проектування в Kotlin*

Класи та об’єкти в мові Kotlin

Робота з класами та об’єктами в мові програмування Kotlin багато в чому схожа на аналогічні дії в мові Java, проте присутні й суттєві відмінності та додаткові можливості. Наприклад, стандартні геттери та сеттери, які середовище розробки IntelliJ IDEA дозволяло генерувати для Java-класів, автоматично додаються в Kotlin. Поля класу з точки зору внутрішньої реалізації є *властивостями*, подібними до властивостей у мові C#. Таким чином, оголошення публічного поля насправді призводить до появи властивості, яка є приватним полем зі стандартними геттером та сеттером.

Для прикладу розглянемо створення класу для представлення собаки з кличкою, віком та вагою (лістинг 3.1). Також переозначимо геттери та сеттери так, щоб вік та вагу неможливо було задати від’ємними, а кличка завжди була з великої літери.

Зверніть увагу, що полям класу обов’язково потрібно задати значення за умовчанням. На відміну від Java, модифікатором доступу за умовчанням є public, тому при оголошенні явно не описуємо його. Переозначені геттери та сеттери записуються відразу після оголошення поля, причому для доступу до значення цього поля використовується ключове слово field. Застосування ключово слова var виправдане наявністю сеттерів, проте за потреби можна зробити змінну доступною тільки для зчитування, застосувавши ключове слово val. Додавання класу відбувається таким же чином, як і додавання файлу, проте в полі Kind обирається значення Class (рис. 3.1).

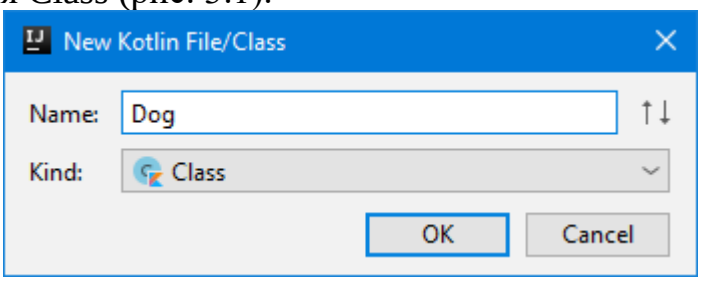


Рис. 3.1. Додавання класу в проєкт

Лістинг 3.1. Клас Dog

```
1   class Dog {
2       var nickname = ""
3           get() = field.toLowerCase().capitalize()
4
5       var age = 0
6           set(value) {
7               if (value >= 0) {
8                   field = value
9               }
10          }
11
12      var weight = 0.0
13          set(value) {
14              if (value >= 0) {
15                  field = value
16              }
17          }
18  }
```

Властивості, значення яких відоме на момент компіляції, можна позначити як константи часу компіляції (compile time constants) за допомогою модифікатора const:

```
const val TAG: String = "MainActivity"
```

До подібних властивостей висувається набір вимог: відсутність власної реалізації геттера, ініціалізація значенням примітивного типу або String та оголошення на верхньому рівні (top-level – поза класами), всередині оголошення об'єкта (ключове слово object) допоміжного об'єкта (ключове слово companion).

Створення екземпляру класу в Kotlin відбувається за допомогою конструкторів, проте ключове слово new не використовується. Крім того, розрізняють основний (primary) та вторинний (secondary) конструктори. Основний конструктор задається набором параметрів у дужках біля назви класу. Наприклад, створимо клас для представлення книги з назвою, роком випуску та тиражем. Для цього достатньо записати один рядок:

```
class Book( val name: String, var year: Int = 2000,
            var circulation: Int? = null )
```

Як і для будь-якої функції в Kotlin, доступні аргументи за умовчанням, які дозволяють зняти потребу у вторинних конструкторах (перевантажених версіях основного конструктора). Проте для більш складних випадків пропонується спеціальний синтаксис для вторинних конструкторів, які обов'язково мають звертатись до основного конструктора за допомогою виклику this(), як у рядку 3 лістинга 3.2. Вторинний конструктор є безаргументним та викликає основний конструктор з одним аргументом.

Лістинг 3.2. Основний та вторинний конструктори

```
1   class Book(val name: String, var year: Int = 2000,
2              var circulation: Int? = null) {
3       constructor(): this("")
4   }
5
6   fun main() {
7       var book1 = Book()
8       println("Назва: " + book1.name)
9       var book2 = Book("Kotlin в действии", 2018)
10      println("Назва: " + book2.name)
11      println("Тираж: " + book2.circulation)
12  }
```

Виклик вторинного конструктора відбувається в рядку 7 та основного – в рядку 9. Результатом запуску стане наступний вивід:

Назва:
Назва: Kotlin в действии
Тираж: null

Якщо розробник не бажає ініціалізувати властивість у конструкторі, мова Kotlin пропонує два важливих способи ініціалізації:

- лініву ініціалізацію за допомогою *делегованих властивостей*;
- відкладену ініціалізацію за допомогою ключового слова *lateinit*.

Делеговані властивості дають розробникові можливість виконувати їх одноразову ініціалізацію та в подальшому, наприклад, додати до деякої бібліотеки. До прикладу можна навести ліниві властивості, значення яких обчислюється лише один раз при першому зверненні. Також делеговані властивості є інструментом введення властивостей, на події щодо зміни яких можна підписатись (observable properties). Крім того, за допомогою делегованих властивостей можна представляти властивості, які зберігаються в асоціативному списку, а не в окремих полях. Загальний синтаксис делегованих властивостей наступний:

```
val/var <назва_властивості>: <тип> by <вираз>
```

Вираз після by є делегатом, саме він оброблятиме звернення до геттеру та сеттеру властивості. Власне делегат не зобов'язаний реалізовувати певний інтерфейс, для нього достатньо наявності методів getValue() і setValue() зі спеціально визначеною сигнатурою.

Ліниві властивості використовуються функцією lazy(), яка прийматиме лямбда-вираз та повертатиме екземпляр класу Lazy<T> – делегат для їх реалізації. Приклад такої поведінки наведено в лістингу 3.3. У рядках 3-6 описано ліниву

властивість text1, яка ініціалізується лише після виклику функції printText() та виконання рядків 20 та 21. Дана частина коду виведе наступний текст:

Ліниві обчислення
Змінна ліниво ініціалізована
Змінна ліниво ініціалізована

Лістинг 3.3. Делеговані та пізньоініціалізовані властивості

```
1  import kotlin.properties.Delegates
2
3  val text1 by lazy {
4      println("Ліниві обчислення")
5      "Змінна ліниво ініціалізована"
6  }
7
8  var name: String by Delegates.observable("<no name>") {
9      prop, old, new -> println("$old -> $new")
10 }
11
12 class User(val map: Map<String, Any?>) {
13     val name: String by map
14     val age: Int     by map
15 }
16
17 lateinit var text2: String
18
19 fun printText() {
20     println(text1)
21     println(text1)
22
23     name = "FirstName"
24     name = "LastName"
25
26     val user = User(mapOf(
27             "name" to "FirstName LastName",
28             "age" to 20
29     ))
30     println(user.name)
31     println(user.age)
32
33     text2 = "Змінна пізньоініціалізована"
34     println("Довжина тексту2: " + text2.length)
35 }
36
37 fun main() {
38     printText()
39 }
40
```

За умовчанням обчислення лінивих властивостей синхронізоване та виконується тільки в одному потоці, причому решта потоків можуть бачити одне і те ж значення. Якщо синхронізація не потрібна, можна передати додатковий параметр LazyThreadSafetyMode.PUBLICATION у функцію lazy(). Якщо ж немає впевненості в тому, що ініціалізація відбуватиметься лише в одному потоці, можна застосувати режим LazyThreadSafetyMode.NONE, який не гарантує потокобезпечності в даному контексті.

Observable-властивості також використовують делегати в своїй основі, проте задають початкове значення властивості, як у рядках 8-10 лістингу 3.3. Зміна значення в рядках 23 і 24 призводить до реакції – виклику обробника, представленого лямбда-виразом у рядку 9. Результатом виводу цієї частини коду буде таке:

<no name> -> FirstName
FirstName -> LastName

Таким чином, відбувається стеження за значенням observable-властивості та реагування на його зміну. Якщо існує потреба в забороні присвоєння певних значень, метод observable() заміняють на метод vetoable().

Ще один приклад застосування делегованих властивостей – збереження властивостей в асоціативному списку. Такий підхід доречний в «динамічному» коді, зокрема, при роботі з JSON. До прикладу в рядках 12-15 лістингу 3.3 описано клас User, конструктор якого приймає асоціативний список, з якого в подальшому делеговані властивості беруть значення (рядки 26-31). Аналогічні дії можна виконувати і зі змінюваними мепами (MutableMap), застосовуючи ключове слово var замість val. Детальніше про вимоги до делегованих властивостей можна дізнатись на офіційній сторінці документації мови Kotlin [1].

Іншою опцією для відокремлення оголошення властивості від її ініціалізації є відкладена (late) ініціалізація за допомогою ключового слова lateinit. Це означає, що розробник сам визначатиме, коли та яким чином проініціалізувати властивість пізніше. Звичайно ж, звернення до ще неініціалізованої властивості викличе виняток kotlin.UninitializedPropertyAccessException. У лістингу 3.3 відкладена ініціалізація показана в рядках 17 та 33-34.

Таким чином, маємо два подібних поведінкових механізми для відокремлення оголошення та ініціалізації змінних і властивостей. Рекомендації щодо вибору конкретного з них наступні:

– ключове слово lateinit може застосовуватись лише до var-властивостей, а функція lazy() – лише до val-властивостей;
– lateinit не працює з примітивними типами (Int, Long та ін.);
– використовуючи шаблон «Одиночка» (Singleton, ключове слово object у мові Kotlin), рекомендується застосовувати функцію lazy(), оскільки така властивість буде ініціалізуватись при першому зверненні до неї;

– для lateinit тип властивості повинен бути ненулабельним;

– lateinit можна використовувати лише тоді, коли властивість не має кастомізованого геттера або сеттера;

– для lateinit коректна ініціалізація властивості в багатопоточному середовищі покладається на користувача. Лінива ініціалізація за умовчанням є потокобезпечною та забезпечує її єдиноразове виконання.

Клас може містити багато інших методів, зокрема, й характерних для Java-класів. Якщо натиснути в коді класу комбінацію Alt + Insert, буде показано набір доступних для генерації методів (рис. 3.2). Зверніть увагу на те, що різновиди порівнянь у Java та Kotlin мають різні позначення. Для порівняння об'єктів у мові Java зазвичай використовується метод equals(), проте Kotlin частіше використовує оператор "==". У той же час, цей оператор у Java означав порівняння за посиланнями. Мова Kotlin дозволяє порівняння за посиланнями за допомогою оператору "===".

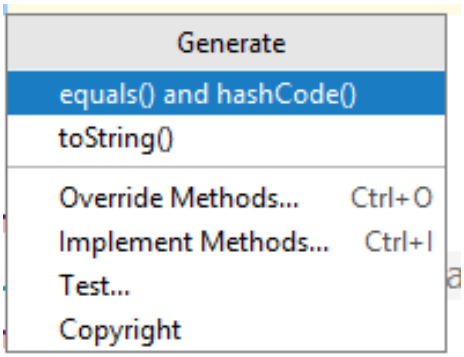


Рис. 3.2. Доступні для генерування методи

Завдання	*Згенеруйте стандартні методи equals(), hashCode() та toString() для запропонованого раніше класу Book. Які дії виконують ці методи та для чого вони вводяться в клас?*

Підсумовуючи стандартне наповнення класу, в мові Kotlin було запропоновано концепцію ***класу даних (data class)***. Від звичайного класу він відрізняється наступним:

– перед ключовим словом class дописується слово data;

– метод toString() заміщено для зручнішого виводу інформації про об'єкт класу;

– автоматично заміщується метод hashCode(), який тепер буде залежати від властивостей об'єкта;

– метод equals() теж автоматично заміщається для порівняння об'єктів за значеннями їх полів, а не за посиланнями на них;

– введено деструктор (деструктивний оператор) для класу, який дозволяє виокремити всі значення полів об'єкту. Якщо якесь значення не буде

потрібно, замість назви поля ставлять знак підкреслення. За бажанням деструктивні оператори можна переозначити і для звичайного класу. Вони матимуть назви component1(), component2() і т. д., залежно від кількості полів класу;

– метод copy() заміщено так, щоб отримувати не поверхневу копію (посилання на той же об'єкт), а глибоку – повноцінне дублювання функціональності. Крім того, за допомогою іменованих аргументів методу copy() можна змінювати значення відповідних полів у копії об'єкта.

Завдання

Порівняйте роботу методу copy() для аналогічних класу даних та звичайного класу. Сконструюйте об'єкти таких класів, здійсніть копіювання та порівняйте їх з оригіналами шляхом виведення результатів викликів вище згаданих методів.

Відповідно до перелічених особливостей, можемо порівняти код класу даних у мові Kotlin:

```
data class Person(val name:String, var age:Int)
```

з відповідним Java-кодом (лістинг 3.4).

Лістинг 3.4. Java-код, який відповідає класу даних у мові Kotlin

```
1   public class Person {
2       private String name;
3       private int age;
4
5       public Person(String name, int age){
6           this.name = name;
7           this.age = age;
8       }
9       public String getName() {
10          return name;
11      }
12      public void setName(String name) {
13          this.name = name;
14      }
15      public int getAge() {
16          return age;
17      }
18      public void setAge(int age) {
19          this.age = age;
20      }
21      @Override
22      public boolean equals(Object o) {
23          if (this == o) return true;
24          if (o == null || getClass() != o.getClass())
25              return false;
```

```
26          Person person = (Person) o;
27          return age == person.age &&
28                  name.equals(person.name);
29      }
30      @Override
31      public int hashCode() {
32          int result = name != null ? name.hashCode(): 0;
33          result = 31 * result;
34          return result;
35      }
36      @Override
37      public String toString() {
38          return "Person{" +
39                  "name='" + name + '\'' +
40                  ", age='" + age +
41                  '}';
42      }
43  }
```

Інкапсуляція функціональності за допомогою класів підкріплюється можливістю приховування даних, яка забезпечується за допомогою модифікаторів видимості (таблиця 3.1). Як уже зазначалось, модифікатором видимості за умовчанням є public. На відміну від мови Java, тут відсутній модифікатор видимості package-private, оскільки пакети в мові Kotlin не використовуються для управління видимістю. Замість нього пропонується модифікатор видимості internal – видимість у межах модуля (набору скомпільованих разом файлів). Це забезпечує дієву інкапсуляцію деталей реалізації модуля, оскільки сторонній код, який порушуватиме інкапсуляцію, буде важче інтегрувати.

Таблиця 3.1. Модифікатори видимості в мові Kotlin [1]

Модифікатор	Член класу	Оголошення верхнього рівня
public	Доступність повсюди	Доступність повсюди
internal	Доступність тільки в модулі	Доступність у модулі
protected	Доступність у підкласах	-
private	Доступність у класі	Доступність у файлі

Як і в Java, мова програмування Kotlin дозволяє вкладати один клас в інший. Це може бути корисних для приховування допоміжної функціональності або просто для розташування коду якомога ближче до місця його застосування.

У мові Kotlin оперують поняттями ***вкладених (nested)*** та ***внутрішніх (inner)*** класів. Вкладені класи у мові Kotlin подібні до статичних вкладених класів у мові Java, а внутрішні класи – до нестатичних вкладених класів. Основна відмінність від мови Java – відсутність доступу до екземпляра зовнішнього класу без явного запиту на цей доступ. Для мови Kotlin характерно те, що вкладені класи не мають доступу до членів зовнішнього класу, проте з зовнішнього класу можуть отримуватись властивості вкладеного без створення відповідного об’єкту. Внутрішні класи (ключове слово inner), у свою чергу, дозволяють доступ до властивостей зовнішніх класів.

Доступ до методів вкладеного класу вимагає створення об’єкту цього класу та виклику відповідних методів. Зазначені можливості продемонстровано в лістингу 3.5. Зверніть увагу, що прибирання ключового слова inner (перетворення класу у вкладений) у описі внутрішнього класу спричинить появу помилки доступу: Unresolved reference: str.

Лістинг 3.5. Вкладені та внутрішні класи в мові Kotlin

```
// зовнішній клас
class outerClass {
    var str = "Зовнішній клас"
    // вкладений клас
    class nestedClass {
        var s1 = "Вкладений клас"
        // не має доступу до властивості str зовнішнього класу
        fun nestfunc(str2: String): String {
            var s2 = s1.plus(str2)
            return s2
        }
    }
    // внутрішній клас
    inner class innerClass {
        var s1 = "Внутрішній клас"
        fun nestfunc(): String {
            // має доступ до властивості str зовнішнього класу
            var s2 = str
            return s2
        }
    }
}

fun main() {
    // створення об'єкту вкладеного класу
    val nested = outerClass.nestedClass()
    // створення об'єкту внутрішнього класу
    val inner = outerClass().innerClass()
    // виклик методу вкладеного класу
    var result = nested.nestfunc(". Метод успішно викликано.")
    println(result)
    // виклик методу внутрішнього класу
    println(inner.nestfunc() +
     ". Отримано доступ до властивості із внутрішнього класу.")
}
```

Оголошення об’єктів та допоміжні об’єкти

Як уже згадувалось, мова Kotlin не підтримує статику в стилі мови Java. Це зумовлено бажанням краще відповідати вимогам об’єктно-орієнтованого програмування, мінімізувавши залучення глобальних даних та методів. Більш дружній для даної парадигми підхід – впровадження синглетних та супутніх (companion) об’єктів.

Ключове слово object використовується для позначення користувацького типу, з якого можна створити тільки один (синглетний) іменований об’єкт. По суті, воно дозволяє одночасно оголосити клас та створити його єдиний екземпляр. Також з використанням ключового слова object можливе створення анонімних об’єктів.

Для типу, визначеного за допомогою object, не передбачено конструкторів, оскільки об’єкт створюється в єдиному екземплярі, а за своєю суттю, вже є об’єктом. Для розуміння механізму створення таких об’єктів слід зробити декомпіляцію JVM-байткоду, результат якого (без метаданих) наведено в лістингу 3.6. На рис. 3.3 показано вигляд байткоду до декомпіляції. Його можна переглянути після переходу по меню Tools → Kotlin → Show Kotlin Bytecode.

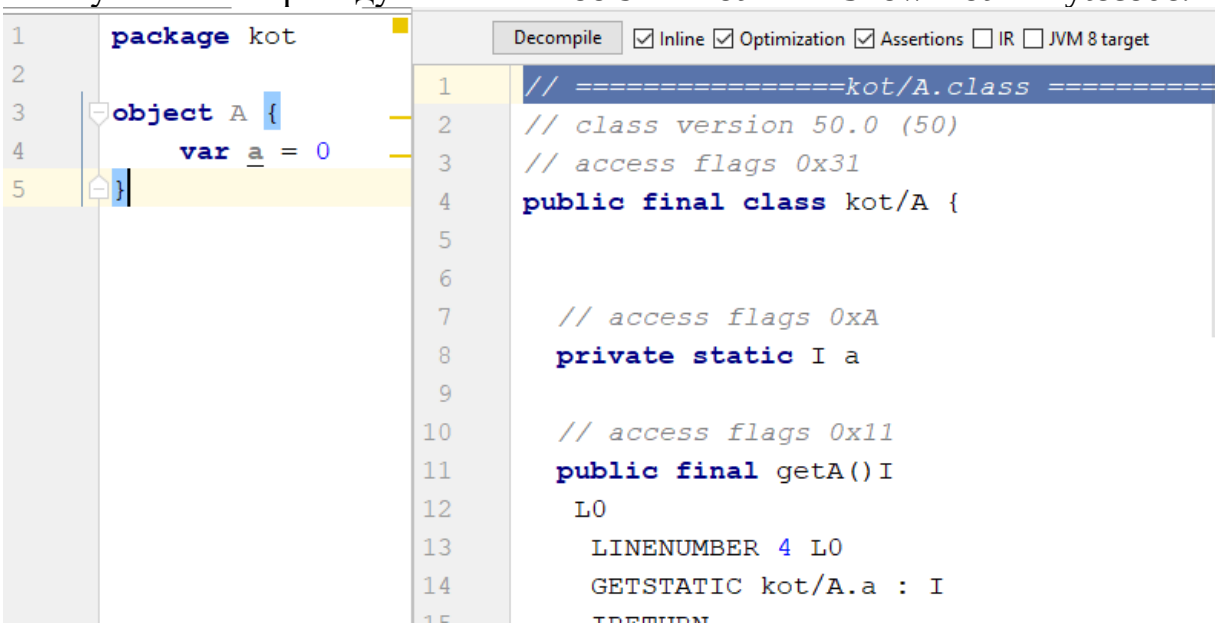


Рис. 3.3. Байткод для синглетного об'єкта

Лістинг 3.6. Декомпільований Java-код для синглетного об’єкта

```
1  public final class A {
2     private static int a;
3     public static final A INSTANCE;
4
5     public final int getA() {
6        return a;
7     }
8
9     public final void setA(int var1) {
10       a = var1;
11    }
12
13    private A() {
14    }
15
16    static {
17       A var0 = new A();
18       INSTANCE = var0;
```

```
19      }
20  }
```

Код мовою Java застосовує стандартний підхід для створення синглетних об'єктів: вводиться статичне константе поле INSTANCE, якому всередині статичного ініціалізатора присвоюється посилання но новий екземпляр класу. Крім того, додаються геттер і сеттер для статичного поля *a*.

За аналогією до оголошення змінних, оголошення об'єктів за допомогою ключового слова object не є виразом та не може використовуватись у правій частині оператору присвоєння. Для безпосереднього посилання на об'єкт застосовують його назву. Лістинг 3.7 демонструє синглетні об'єкти в дії.

Лістинг 3.7. Робота з оголошенням об'єкту та об'єктом-виразом

```
1   object Counter {
2       private var count: Int = 0
3
4       fun currentCount() = count
5
6       fun increment() {
7           ++count
8       }
9   }
10
11  fun main() {
12      for(i in 0 until 5) {
13          Counter.increment()
14          println(Counter.currentCount())
15      }
16
17      val counter2 = object {
18          private var count: Int = 0
19
20          fun currentCount() = count
21
22          fun increment() {
23              ++count
24          }
25      }
26
27      for(i in 0 until 5) {
28          counter2.increment()
29          println(counter2.currentCount())
30      }
31  }
```

Обидва цикли виведуть у ствопчик числа від 1 до 5. Таким чином, маємо можливість створювати як аналоги статичних полів класу з мови Java, так і анонімні об'єкти (рядок 17-25) – екземпляри внутрішніх анонімних класів, які, зокрема, можна передавати в якості аргументів чи представляти як вихідний тип для функцій. На позначення таких особливостей вводяться спеціальні терміни – ***оголошення об'єктів (object declarations)*** та ***об'єкти-вирази (object expressions)***

відповідно. Семантично це різні конструкції: об'єкти-вирази ініціалізуються та виконуються негайно при використанні, а оголошення об'єктів ініціалізуються ліниво, при першому доступі до них.

Загалом слід бути обережним з використанням синглетних об'єктів. Враховуючи, що вони «живуть» стільки ж, скільки й процес, у якому вони були сформовані, їх надмірне наповнення функціональністю може призвести до витоків пам'яті.

Забезпечення аналогів статичних полів та методів з мов Java/C# також покладається в мові Kotlin на ***допоміжні об'єкти (companion objects)***. Ідея їх використання залишається тією ж: статика представляється як функції рівня пакету (package-level functions). Допоміжні об'єкти ініціалізуються при завантаженні відповідного класу, що семантично відрізняє їх від об'єктів-виразів та оголошення об'єктів. Допоміжні об'єкти впроваджуються за допомогою ключового слова companion object і можуть бути анонімними, як у випадку оголошення об'єктів. Приклад використання допоміжного об'єкту описано в лістингу 3.8.

Лістинг 3.8. Допоміжний об'єкт

```
1   class ToBeCalled {
2       companion object Test {
3           var callsNumber: Int = 7
4           fun callMe() = println("Ви дзвоните мені")
5           fun callByNumber(s: String) =
6                   println("Ви дзвоните на номер $s")
7           fun multipleCallMe() =
8                   println("Ви дзвоните мені $callsNumber разів")
9       }
10  }
11
12  fun main() {
13      ToBeCalled.callMe()   // Ви дзвоните мені
14      ToBeCalled.callByNumber("01234567890")// Ви дзвоните
15                                            // на номер 0123456789
16      ToBeCalled.callsNumber = 5
17      ToBeCalled.multipleCallMe()// Ви дзвоните мені 5 разів
18  }
```

Зауважте, що вигляд роботи з допоміжними об'єктами дуже подібний до статичних атрибутів класу в інших мовах, проте в ході виконання це все ще члени реальних об'єктів. Також допоміжні об'єкти мають важливу перевагу над високорівневими функціями (top-level functions), оскільки мають доступ до приватних функцій та властивостей їх зовнішнього класу.

Реалізація наслідування у мові Kotlin

До основних відношень між класами відносять генералізацію, реалізацію та асоціацію з різновидами – композицією та агрегацією. Розпочнемо з генералізації, оскільки вона демонструє один з базових принципів ООП – наслідування коду.

Загалом мова Kotlin пропонує кілька модифікаторів наслідування (inheritence modifiers), представлених у таблиці 3.3.

Таблиця 3.3. Модифікатори наслідування в Kotlin [2]

Модифікатор	Відповідний член	Примітки
final	Неможливо замістити	Використовується за умовчанням
open	Можливо замістити	Потрібно явно задавати
abstract	Необхідно замістити	Лише в абстрактних класах
override	Заміщає член суперкасу або інтерфейсу	Заміщений член класу відкритий за умовчанням

Неможливість наслідування від звичайного класу показана на рис. 3.3. Для виправлення даної помилки потрібно позначити клас Button та метод click() модифікаторами open. Синтаксично наслідування описується шляхом дописування до заголовку класа через двокрапку виклик конструктора базового класу, як у рядку 9 лістинга 3.9.

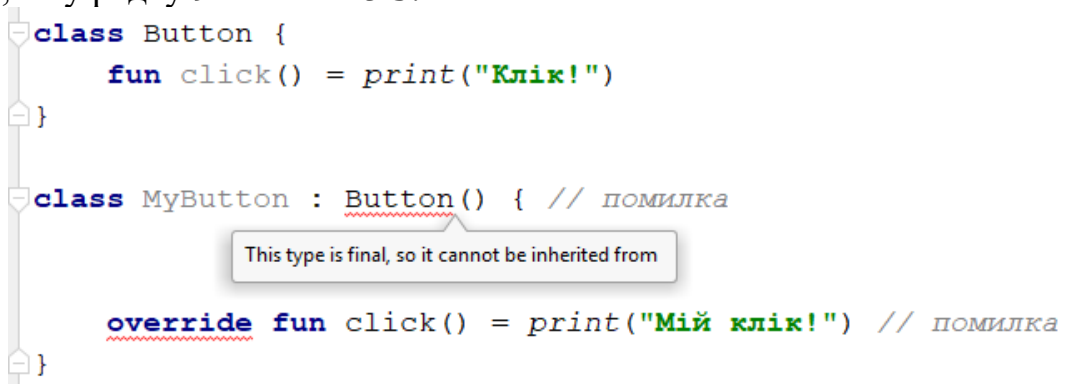


Рис. 3.3. Неможливість наслідування через фінальність базового класу

Лістинг 3.9. Модифікатори наслідування в мові Kotlin

```
1   abstract class Widget {
2       abstract fun draw()
3       open fun focus() {}
4       fun hide() {
5           println("Віджет сховано!")
6       }
7   }
8
9   class MyButton : Widget() {
10      override fun draw() {
11          println("Кнопка = текст + рамка!")
12      }
13
14      override fun focus() {
15          super.focus()
16          println("Фокус на кнопці!")
17      }
18  }
19
20  fun main() {
21      val btn = MyButton()
```

```
22      btn.draw()
23      btn.focus()
24      btn.hide()
25  }
```

Лістинг 3.9 також демонструє роботу з абстрактним класом Widget, у якого абстрактним є метод draw(). Модифікатор abstract автоматично передбачає відкритість для наслідування, тобто модифікатор open непотрібний. Підклас MyButton повинен забезпечити реалізацію даного метода, щоб бути конкретним (неабстрактним) класом, на основі якого можна конструювати об'єкти. Для цього реалізацію потрібно замістити за допомогою модифікатора override (рядки 10-12). Аналогічна ситуація з заміною функціональності методу focus() з базового класу всередині підкласу MyButton. Оскільки базова реалізація вже існує, за потреби можемо звернутись до неї за допомогою ключового слова super, що характерно й для мови програмування Java. Розробники мови Kotlin рекомендують *завжди* використовувати модифікатор override у підкласах.

Відношення реалізації забезпечується за допомогою того ж механізму, що й у мові Java, – інтерфейсів. Функціонально вони нагадують інтерфейси з Java 8, проте методи з реалізацією за умовчанням не вимагають ключового слова default у своїй сигнатурі. Для синтаксичного позначення наслідування та реалізації інтерфейсів мова Kotlin пропонує оператор «:», тому можлива множинна релізація інтерфейсів (як у рядку 11 лістинга 3.10). Аналогічно через кому можна записувати інтерфейси та виклик конструктора базового класу.

Лістинг 3.10. Множинна реалізація інтерфейсів

```
1   interface Drawable {
2       fun draw()
3       fun display() = println("Drawable!")
4   }
5   interface Focusable {
6       fun setFocus(b: Boolean) =
7           println("${if(b) "Отримано" else "Втрачено"}
8                    фокус")
9       fun display() = println("Focusable!")
10  }
11  class Button : Drawable, Focusable {
12      override fun draw() = println("Кнопку натиснено!")
13
14      override fun display() {
15          super<Drawable>.display()
16          super<Focusable>.display()
17      }
18  }
19
20  fun main() {
21      val btn = Button()
22      btn.display()
23      btn.setFocus(true)
24      btn.draw()
```

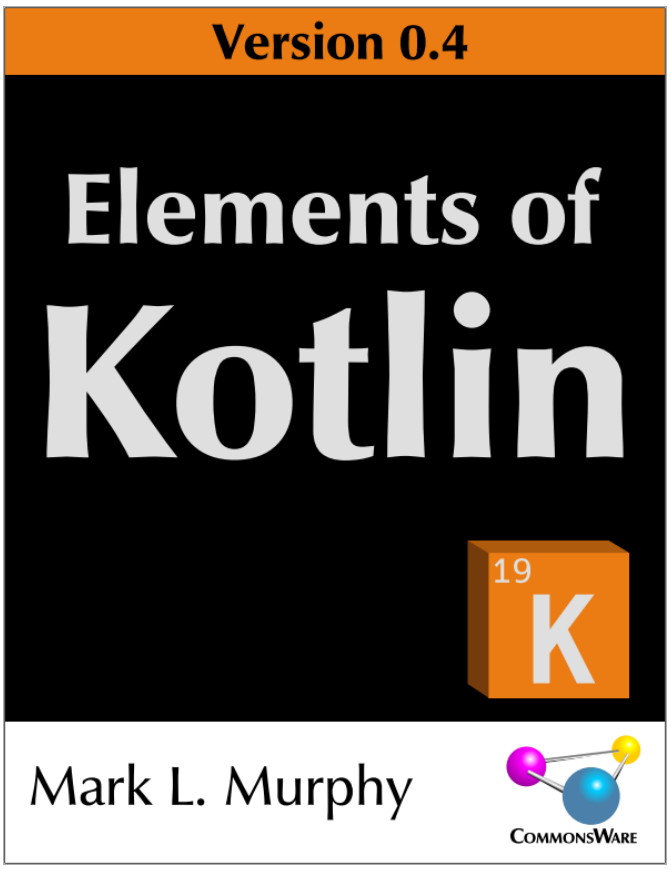
```
25  }
```

Особливістю лістинга 3.8 є конфлікт множинних реалізацій для методу display() та його вирішення за допомогою явного вибору конкретної реалізації (рядки 15-16). У результаті запуску даного коду на екран буде виведено наступний текст:

Drawable!
Focusable!
Отримано фокус
Кнопку натиснено!

Дерево дочірніх класів називають ***ієрархією наслідування***.

+ АБСТРАКТНІ КЛАСИ + ІНТЕРФЕЙСИ



Перелічення та запечатані класи

Функціональність перелічень у мові Kotlin повторює

Узагальнене програмування та параметризовані типи

https://habr.com/ru/company/redmadrobot/blog/301174/

https://medium.com/learn-kotlin/generics-variance-in-java-vs-kotlin-c964df6f649b

Дженерики дозволяють оголошувати тип у якості параметру класу чи методу. Прикладом подібних класів є ArrayList<Integer>, Set<String> тощо. Використання примітивних типів у дженериках неможливе, слід передавати типи об'єктів або класи-обгортки. Тобто заборонено створювати типи на зразок таких: HashMap<int>, ArrayList<char> та ін.

Для оголошення дженериків мова Kotlin пропонує наступний синтаксис:

```
class Box<T>(t: T) {
    var value = t
}
```

Загалом базове використання дженериків у мові Kotlin мало відрізняється від аналогічної функціональності в Java (не враховуючи підстановочні символи – wildcards).

Java	Kotlin

`public class SomeGenericClass <T> {`
`  private T mSomeField;`
 	
`  public void setSomeField(T someData) {`
`  mSomeField = someData;`
`  }`
 	
`  public T getSomeField() {`
`  return mSomeField;`
`  }`
`}`
	`class SomeGenericClass <T> {`
	`  private var mSomeField: T? = null`
	
	`  fun setSomeField(someData: T?) {`
	`  mSomeField = someData`
	`  }`
	
	`  fun getSomeField(): T? {`
	`  return mSomeField`
	`  }`
	`}`

Варіантність коду також має подібні позначення у мовах Java:

? – підстановочний символ (wildcard), який дозволяє усі типи змінної;

? extends T – підстановочний символ з верхньою межею типу T. Це означає, що параметричний тип може бути лише одним із підтипів типу T;

? super T – підстановочний символ з нижньою межею. Визначає параметричний тип, який може бути лише супертипом відносно типу T.

Реіфіковані дженерики

Ієрархія винятків та обробка помилок у мові Kotlin

Часто синтаксичні можливості об'єктно-орієнтованої мови програмування для побудови ієрархії наслідування демонструються на прикладі ієрархії винятків (exceptions). Загалом виняток слід розглядати як подію, що сигналізує про проблему в ході виконання програми. У мові Java винятки представлені батьківським класом Exception. Це ж характерно і для мови Kotlin. Іншим видом критичних подій є помилки, представлені класом Error та його підкласами. Помилки слід не обробляти, а виправляти; вони є результатом серйозних проблем у програмі, зокрема, некоректного використання пам'яті.

У контексті мови Java говорять про *винятки, що перевіряються (checked exceptions)*, та *винятки, що не перевіряються (unchecked exceptions)*. Перші повинні оброблятись або за допомогою ключового слова throws оголошуватись після сигнатури методу. На противагу їм, винятки, що не перевіряються, можна ігнорувати, проте за відсутності їх обробки може відбуватись збій у роботі додатку.

Для мови Kotlin всі винятки є такими, що не перевіряються, що дозволяє самостійно обирати, чи варто їх відловлювати та обробляти. Ще однією відмінністю від мови Java є можливість використання оператора try як виразу.

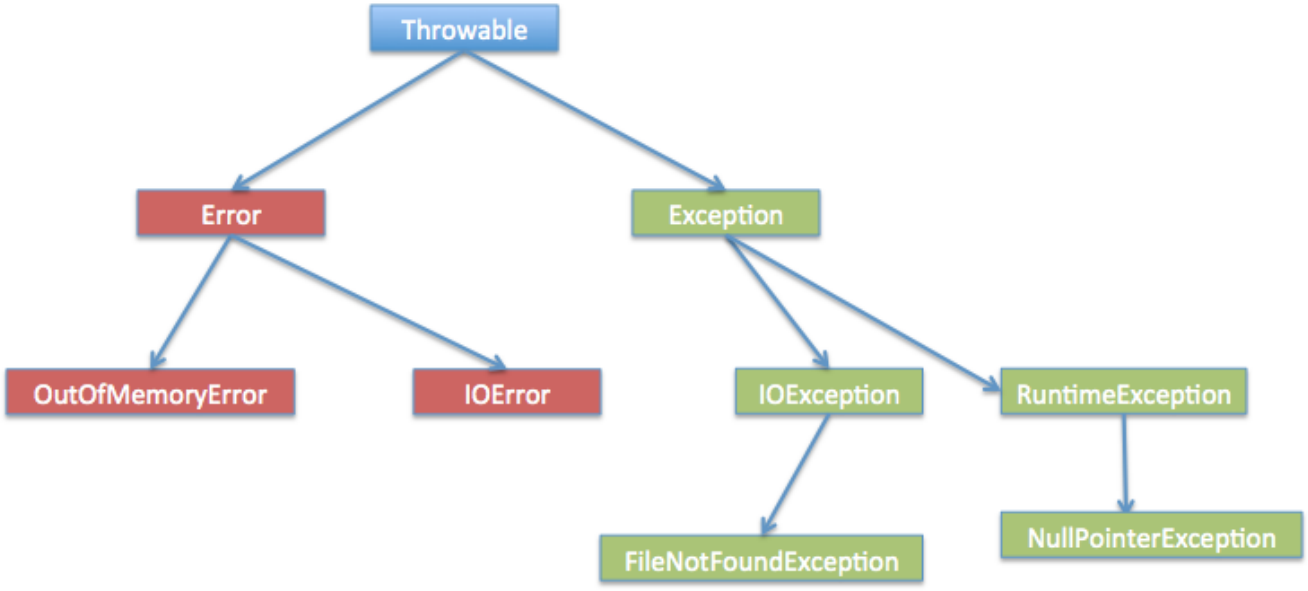


Рис. 3.4. Елементи ієрархії винятків у мові Java

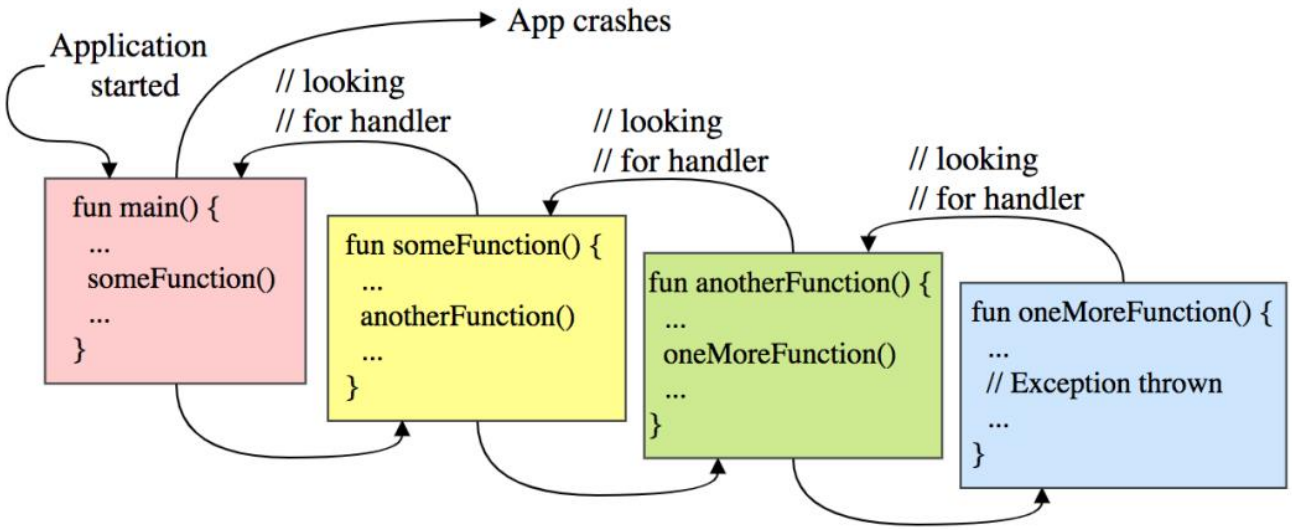
https://habr.com/ru/company/maxilect/blog/447380/
https://habr.com/ru/company/funcorp/blog/471766/

```
fun main() {
  someFunction()
}

fun someFunction() {
  anotherFunction()
}

fun anotherFunction() {
  oneMoreFunction()
}

fun oneMoreFunction() {
  throw Exception("Something went wrong")
}
```



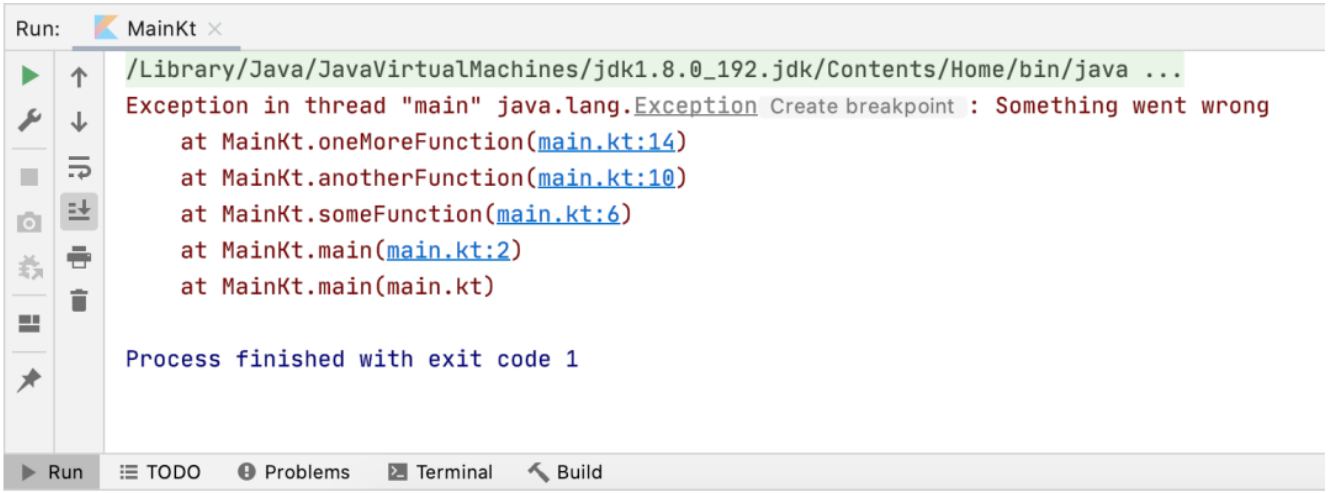
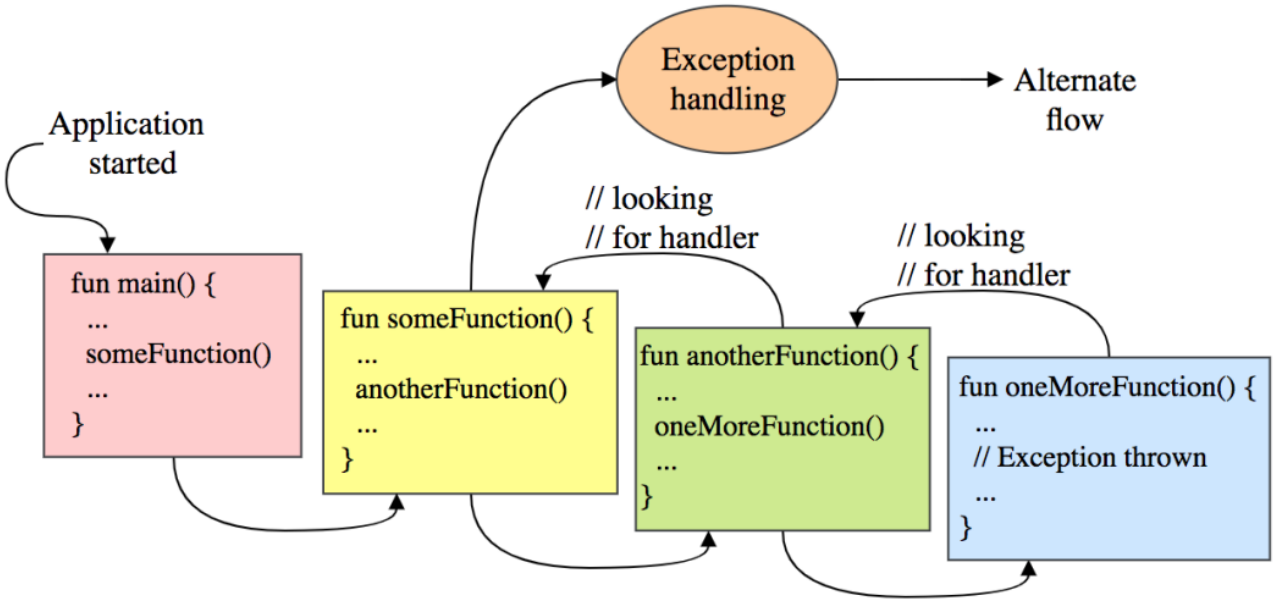


Рис. 3.5. Стектрейс необробленого винятку

Without handling the exception, the process ends up in the entry point of your app — the main() function — and then the app crashes, your user sees an annoying error message. To prevent the app from crashing, you should handle an exception; you can do that in any function in the chain that led to the exception. Now look how things change if you handle an exception (рис. 3.6). While rolling up your program, it finds a handler inside someFunction() and, after handling an alternate execution, the program flow re-starts and your app doesn't crash.



https://kotlin.christmas/2019/17

Практичні завдання

1. ***0,5 бала*** Створіть клас для представлення робітника за його іменем, посадою та роком прийняття на роботу. Додайте властивість для обчислення стажу роботи (різниця між поточним роком та роком працевлаштування) та метод для імітації роботи, який буде виводити на екран текст "Працюю…". Створіть функцію-розширення для розробленого класу, яка виведе всю інформацію про робітника.
2. ***0,5 бала*** Уявіть, що Ви розробляєте movie-viewing додаток. Користувачі можуть створювати списки фільмів та ділитись ними з іншими користувачами. Створіть класи User та MovieList, які забезпечуватимуть такі списки для користувачів. У класі User повинен бути метод addList(), який додає заданий список у змінюваний меп з MovieList-об'єктами (ім'я користувача виступатиме ключем) та метод list(): MovieList?, який

повертатиме MovieList для заданого імені. Клас MovieList міститиме назву списку та змінюваний список з назвами фільмів. Додатково впровадьте метод print(), який буде виводити всі фільми зі списку.
Створіть двох користувачів та заповніть їх списки фільмів. Поділіться списками між користувачами.

3. 0,5 бала Змоделюйте роботу магазину, який торгує футболками. Додаток повинен містити наступні класи:
   - TShirt: описує стиль футболки для покупки: розмір, колір, ціну, (опційно) зображення та ін;
   - User: зареєстрований клієнт мобільного додатку для цього магазину. Має ПІБ, електронну пошту, номер телефону та платіжну картку (об'єкт класу ShoppingCart);
   - Address: представляє адресу доставки: ПІБ, вулиця, номер будинку, номер квартири, населений пункт, індекс;
   - ShoppingCart: містить поточне замовлення (список футболок, які клієнт бажає придбати), а також метод для обчислення загальної вартості товарів. Додатково присутня адреса, на яку слід надіслати замовлення.
4. ***0,8 бала*** Уявіть, що ви розробляєте науково-фантастичну гру, персонажі якої можуть володіти однією, двома або трьома (визначається випадковим чином) спеціальними навичками із запропонованого переліку:
   - авторитетність,
   - дипломатичність,
   - інженерні навички,
   - військові навички,
   - медичні навички,
   - наукові навички.

Згенеруйте 10 таких персонажів, кожна навичка матиме випадково визначений з діапазону від 5 до 20 рівень. Також персонаж повинен мати ім'я, запропоноване користувачем.

Кожну хвилину має генеруватись бонус для однієї з навичок. У командний рядок потрібно буде виводити перелік персонажів, для яких його можна застосувати, після чого пропонувати користувачеві обрати, кому цей бонус надати (від 1 до 8).

Для зручності слід виводити добре відформатовану статистику щодо команди персонажів. Також програма повинна передбачати бої між персонажами один-на-один за двома випадково обраними навичками (до початку бою). Бій проходить шляхом генерування випадкових чисел для відповідної навички (від 0 до максимального поточного значення). Для кожної навички тричі генерується результат, а загальна сума балів визначає переможця.

Наприклад, можемо мати наступних гравців:

	Гравець1	Гравець2	Гравець3
Авторитет	7	0	5
Дипломатичність	0	12	0
Інженер	5	0	0
Військовий	6	0	0
Медик	0	0	8
Науковець	0	0	0

Генерується бій між Гравець1 та Гравець3 за навичками «Авторитет» та «Медик». Перший гравець генерує: 4+2+7+0+0+0=13, другий гравець – 1+1+0+5+6+2=15. Перемога другого гравця дозволяє йому розвинути певний з доступних навиків на половину від різниці балів. У випадку нічиєї слід додавати +1 до відповідних навичок (якщо значення навички не 0). Бій також може проходити лише через 20 секунд після завершення попереднього, про доступність бою та бонусу слід сповіщати користувача.

Пограйтесь у гру 3 хвилини та виведіть статистику щодо команди на початку гри та після її завершення.

5. ***0,6 бала*** Змоделюйте систему відповідно до заданих класів та їх відповідальностей:

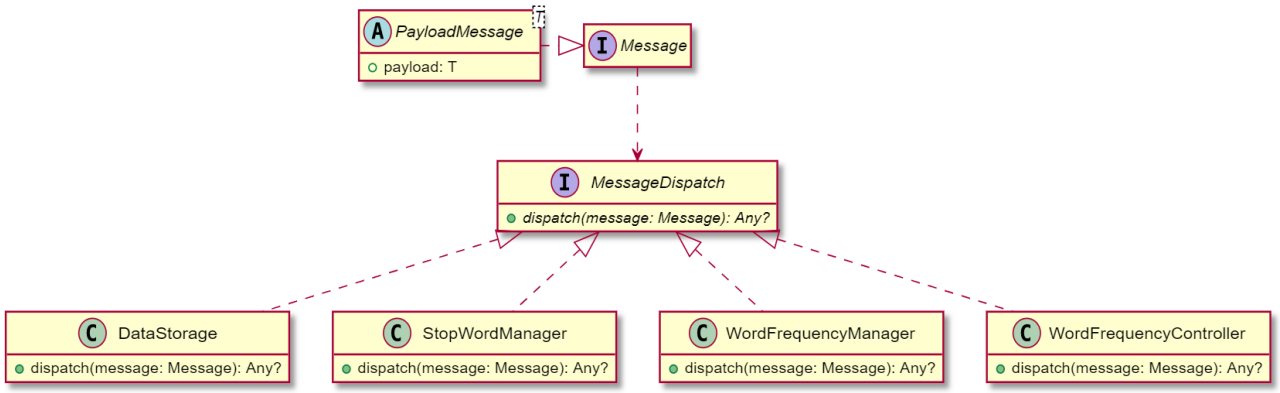
Клас DataStorage
- Зчитує текстовий файл
- Розбирає (parse) вміст файлу на слова
- Зберігає слова
- Повертає слова

Клас StopWordManager
- Зчитує файл зі стоп-словами
- Розбирає вміст файлу
- Зберігає стоп-слова
- Перевіряє, чи є слово стоп-словом

Клас WordFrequencyManager
- Зберігає меп «слово-частота зустрічності»
- Управляє частотою, коли слово надходить (submitted)
    - Якщо слово ще не існувало, додати його з частотою 1
    - Інакше інкрементувати частоту на 1

Клас WordFrequencyController
- Керує та впорядковує потік повідомлень між previous об’єктами

Схема такої системи може бути зображена на діаграмі класів


PayloadMessage
payload: T
Message
MessageDispatch
dispatch(message: Message): Any?
DataStorage
dispatch(message: Message): Any?
StopWordManager
dispatch(message: Message): Any?
WordFrequencyManager
dispatch(message: Message): Any?
WordFrequencyController
dispatch(message: Message): Any?