

Зміст

Передмова	4
Тема 1. Базовий синтаксис мови програмування Kotlin	5
Тема 2. Вступ до функціонального програмування мовою Kotlin	9
Тема 3. Основи об'єктно-орієнтованого програмування в мові Kotlin	9

Система оцінювання

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

Тема 4. Анатомія мобільного додатку для платформи Android	14
Тема 5. Матеріальний дизайн та стилізація графічного інтерфейсу мобільних додатків	18
Тема 6. Макетування інтерфейсу мобільного додатку на основі фрагментів	24
Тема 7. Навігаційні елементи управління в Android-додатках	29
Тема 8. Використання інформації з локальних джерел даних в мобільних додатках	37
Тема 9. Проектування Android-додатків на базі архітектурних компонентів від компанії Google	41
Тема 10. Служби та асинхронні операції в Android-додатках	43
Список рекомендованої літератури	75
Додатки	76

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 IntelliJ IDEA дозволяла генерувати для Java-класів, автоматично додаються в Kotlin. Поля класу з точки зору внутрішньої реалізації є *властивостями*, подібними до властивостей з мови 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 ключового 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).

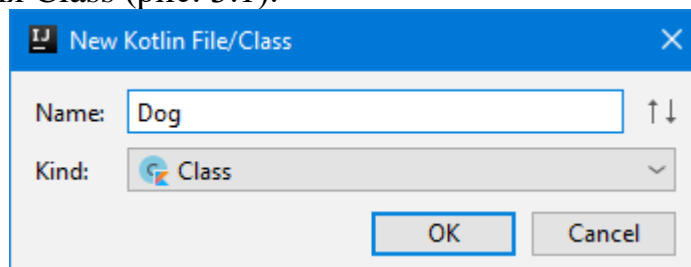


Рис. 3.1. Додавання класу в проєкт

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 синглетного об'єкта (ключове слово `object`) або об'єкта-помічника (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:

- ліниву 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 функцію `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:

Лінійні обчислення

Змінна лінійно ініціалізована

Змінна лінійно ініціалізована

Лістинг 3.3. Делеговані та пізньоініціалізовані властивості

```
1  import kotlin.properties.Delegates
2
3  val text1 by lazy {
4      println("Лінійні обчислення")
5      "Змінна лінійно 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 лінійних 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 оператора `===`.

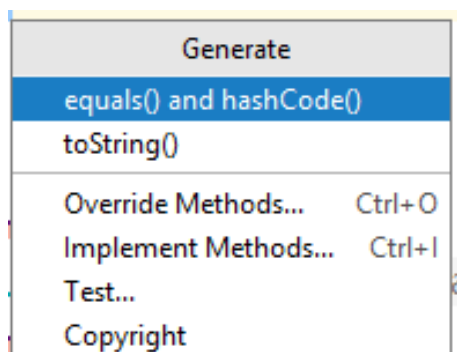


Рис. 3.2. Доступні для генерування методи



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()` переозначено для зручнішого виводу інформації про об'єкт класу;
- автоматично переозначається метод `hashCode()`, який тепер буде залежати від полів об'єкта;
- метод `equals()` теж автоматично переозначається 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()` переозначено 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.

 <p>Завдання</p>	<p><i>Порівняйте роботу методу <code>copy()</code> для аналогічних класу даних та звичайного класу. Сконструйте об'єкти таких класів, здійсніть копіювання та порівняйте їх з оригіналами шляхом виведення результатів викликів вище згаданих методів.</i></p>
--	--

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 + '\n' +  
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, 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
<code>public</code>	Доступність повсюди	Доступність повсюди
<code>internal</code>	Доступність тільки в модулі	Доступність у модулі
<code>protected</code>	Доступність у підкласах	-
<code>private</code>	Доступність у класі	Доступність у файлі

Як і в Java, мова програмування Kotlin дозволяє вкладати один клас в інший. Це може бути корисним для приховування допоміжної функціональності або просто для розташування коду якомога ближче до місця його застосування. Основна відмінність від мови Java – відсутність доступу до екземпляра зовнішнього класу без явного запиту на цей доступ. Таблиця 3.2 демонструє відповідність між синтаксисом вкладених та внутрішніх класів у мовах Java та Kotlin.

<https://www.geeksforgeeks.org/kotlin-nested-class-and-inner-class/>

Як уже згадувалось, мова Kotlin має цікавий синтаксичний інструмент – ключове слово `object`. Він використовується для позначення користувацького

типу, з якого можна створити тільки один (синглетний) іменований об'єкт. 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 3.5. На рис. 3.3 показано вигляд байткоду до декомпіляції. Його можна переглянути після переходу по меню `Tools → Kotlin → Show Kotlin Bytecode`.

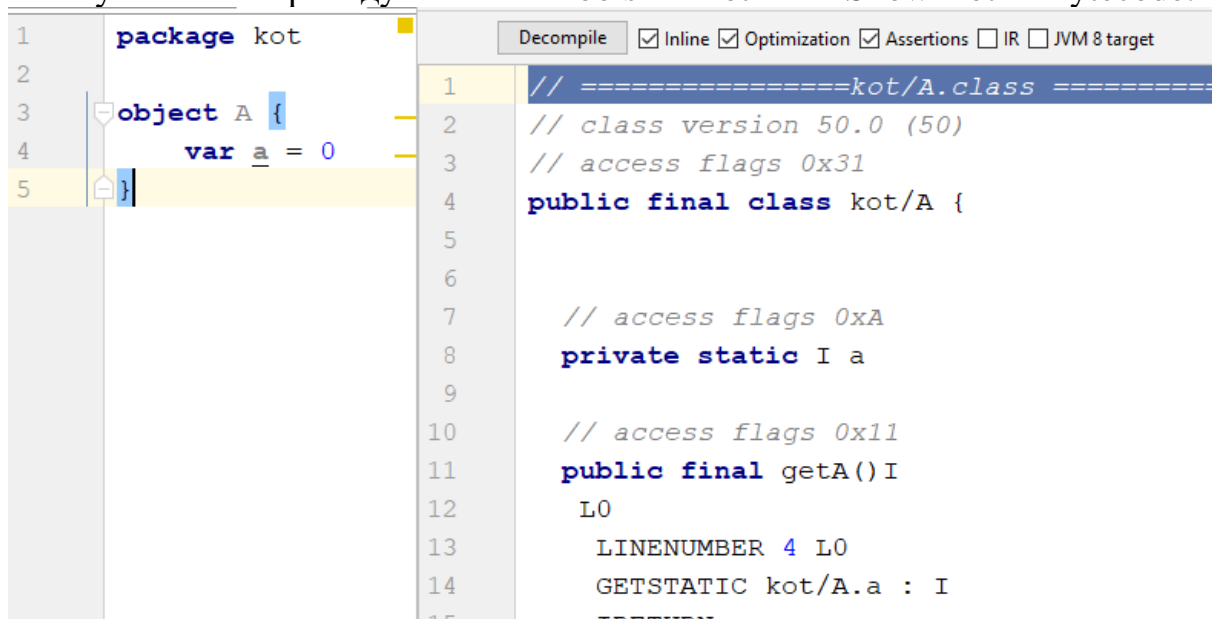


Рис. 3.3. Байткод для синглетного об'єкта

Лістинг 3.5. 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 на новий екземпляр класу. Крім того, додаються геттер і сеттер для статичного поля *a*.

<https://kotlinlang.org/docs/tutorials/kotlin-for-py/objects-and-companion-objects.html>

Основні відношення між класами

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 (inheritance modifiers), представлених у таблиці 3.3.

Таблиця 3.3. Модифікатори наслідування в Kotlin [2]

Модифікатор	Відповідний член	Примітки
final	Неможливо переозначити	Використовується за умовчанням
open	Можливо переозначити	Потрібно явно задавати
abstract	Необхідно переозначити	Лише в абстрактних класах
override	Переозначає член суперкласу або інтерфейсу	Переозначений 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 класу через двокрапку виклик конструктора базового класу, як у рядку 9 лістинга 3.4.

```

class Button {
    fun click() = print("Клік!")
}

class MyButton : Button() { // помилка
    // This type is final, so it cannot be inherited from

    override fun click() = print("Мій клік!") // помилка
}
    
```

Рис. 3.3. Неможливість наслідування через фінальність базового класу

Лістинг 3.4. 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 3.4 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 переозначити 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. Аналогом ключових слів `extends` та `implements` з мови `Java` є один оператор мови `Kotlin` – `“:”`, тому можлива множинна реалізація інтерфейсів (як у рядку 11 лістинга 3.5). Аналогічно через кому можна записувати інтерфейси та виклик конструктора базового класу.

Лістинг 3.5. 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 3.5 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!

Узагальнене програмування та параметризовані типи

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

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} Змодельуйте 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* Змодельуйте 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

