ЛЯМБДА-ВИРАЗИ ТА ФУНКЦІОНАЛЬНІ ТИПИ

Питання 3.4.

Лямбда-вираз — це блок коду, який передається для наступного виконання один або кілька разів

- У мові Java відсутні функціональні типи даних.
  - Замість цього функції виражаються в якості об'єктів – екземплярів класів, які реалізують конкретний інтерфейс.
  - Лямбда-вирази надають зручний синтаксис для отримання таких екземплярів:

```
public interface Comparator<T> {
  int compare(T first, T second);
}
```

```
class LengthComparator implements Comparator<String> {
  public int compare(String first, String second) {
    return first.length() - second.length();
  }
}
```

Аналогічний запис за допомогою лямбда-виразу:

```
(String first, String second) -> first.length() - second.length()
```

Лямбда-вирази

- Якщо в тілі лямбда-виразу виконуються обчислення, тіло охоплюється фігурними дужками:

```
(String first, String second) -> {
  int difference = first.length() < second.length();
  if (difference < 0) return -1;
  else if (difference > 0) return 1;
  else return 0;
}
```

Якщо у лямбда-виразу відсутні параметри, слід вказати порожні круглі дужки:

```
Runnable task = () -> { for (int i = 0; i < 1000; i++) doWork(); }
```

Якщо типи параметрів лямбда-виразу можуть бути виведеними, їх можна опустити:

```
Comparator<String> comp =
    (first, second) -> first.length() - second.length();
     // То же, что и (String first, String second)
```

- Якщо у метода наявний єдиний параметр вивідного типу, можна навіть опустити дужки

```
EventHandler<ActionEvent> listener = event ->
  System.out.println("Oh noes!");
    // Вместо выражения (event) -> или (ActionEvent event) ->
```

Результат обчислення лямбда-виразу взагалі не вказується.
Проте компілятор виводить його з тіла лямбда-виразу та перевіряє, чи відповідає він очікуваному результату.

Наприклад, вираз
`(String first, String second) -> first.length() - second.length()`
може використовуватись у контексті, де очікується результат типу int (або сумісного з ними типу на зразок Integer, long, double)

Функціональні інтерфейси

- В Java присутні багато інтерфейсів, що виражають дію, зокрема Runnable і Comparator.
  - Лямбда-вирази сумісні з ними.
  - Лямбда-вирази можна надати всякий раз, коли очікується об'єкт класу, що реалізує інтерфейс з *єдиним абстрактним методом*.
  - Такий інтерфейс називається *функціональним*.
- Розглянемо метод Arrays.sort().
  - У якості другого параметру йому потрібен екземпляр типу Comparator — інтерфейсу з єдиним методом.
  - Замість нього достатньо надати лямбда-вираз:
  - Arrays.sort(words, (first, second) -> first.length() - second.length());
  - Змінна другого параметру методу Arrays.sort() автоматично приймає об'єкт деякого класу, що реалізує інтерфейс Comparator <String> .

Функціональні інтерфейси

- Управління такими об’єктами та класами повністю залежить від конкретної реалізації та достатньо оптимізоване.
- У більшості МП, які підтримують функціональні літерали, можна оголосити типи функцій на зразок (String, String) -> int, оголошувати змінні подібних типів, присвоювати цим змінним функції та викликати їх.
  - З лямбда-виразом у Java можна робити *лише одне:* присвоювати його змінній, типом якої є функціональний інтерфейс, щоб перетворити його в екземпляр даного інтерфейсу.
- Лямбда-вираз неможливо присвоїти змінній типу Object, оскільки це не функціональний інтерфейс.

Посилання на методи

- Іноді вже існує метод, що виконує саме ту дію, яку потрібно передати іншому коду.
  - Нехай символьні рядки потрібно відсортувати незалежно від регістру букв.
  - Arrays.sort(strings, (x, y) -> x.compareToIgnoreCase(y));
  - Альтернатива:
  - Arrays.sort(strings, String::compareToIgnoreCase);
- У класі Objects визначається метод isNull().
  - Після виклику Objects.isNull(x) повертається значення x == null.
  - Спецыально призначений для передачі йому посилання на метод.
  - Наприклад, у результаті виклику: `list.removelf(Objects::isNull);`
    зі списку видаляються всі порожні значення.
- Нехай потрібно вивести всі елементи списку.
  - У класі ArrayList наявний метод forEach(), який застосовує передану йому функцію до кожного елементу списку.
  - list.forEach(x -> System.out.println(x));
  - Більш елегантне рішення - передати метод println() методу forEach().
  - list.forEach(System.out::println);

Різновиди операції

- Операція :: відокремлює назву методу від імені класу чи об’єкта і має 3 різновиди:
  - ***Клас::МетодЕкземпляру*** – перший параметр стає отримувачем методу, а решта параметрів передаються методу.
    - Посилання String::compareToIgnoreCase рівнозначне лямбда-виразу (x, y) -> x.compareToIgnoreCase(y)
  - ***Клас::СтатичнийМетод*** – усі параметри передаються статичному методу.
    - Посилання на метод Objects::isNull рівнозначне лямбда-виразу x -> Objects.isNull(x).
  - ***Об’єкт*::*МетодЕкземпляру*** – метод викликається для заданого об’єкта, а параметри передаються методу екземпляра.
    - Посилання на метод System.out::println рівнозначне лямбда-виразу x -> System.out.println(x).
- Якщо є кілька переозначуваних одноіменних методів, компілятор намагатиметься їх призначення з контексту.
  - Наприклад, коли посилання на метод **println()** передається методу **forEach()** з класу **ArrayList<String>,** то обирається варіант **println(String).**
  - В посиланні на метод допускається вказувати посилання this: «this::equals» = «x -> this.equals(x)»
  - У внутрішньому класі можна вказати посилання **this** на зовнішній клас: ***ЗовнішнійКлас*::this::*Метод.***
  - Аналогічно можна вказати й посилання **super**.

Посилання на конструктори

- Діють так же, як і посилання на методи, проте замість назви методу вказується оператор new: **`Employee::new`**.
- Якщо у класа є кілька конструкторів, конкретний з них обирається з контексту.
  - Нехай матимемо список ріядків: **`List<String> names = . . .;`**
  - Потрібно скласти список працівників за їх іменами.
  - Не застосовуючи цикли, можна спочатку перетворити список у потік даних, а потім викликати метод map().
  - Цей метод приймає в якості параметру функцію та накопичує отримані результати.
  - Потік даних names.stream() містить об’єкти типу String, тому компілятору відомо, що посилання Employee::new вказує на конструктор класу
  - Employee(String).Stream<Employee> stream = names.stream().map(Employee::new);
- Посилання на масиви можна сформувати за допомогою типів масивів.
  - Наприклад, **`int[ ]::new`** — посилання на конструктор з 1 параметром, що позначає довжину масиву.
  - Рівнозначний лямбда-вираз: **`n->new int[n]`**.

- Посилання на конструктори у вигляді масивів зручні для преодоления следующего ограничения: в Java нельзя построить массив обобщенного типа.
  - По этой причине такие методы, как Stream.toArray(), возвращают массив типа Object, а не массив типа его элементов, как показано ниже.
  - Object[] employees = stream.toArray();
- Но этого недостаточно: пользователю требуется массив работников, а не объектов.
  - Можно вызвать другой вариант метода toArray(), принимающий ссылку на конструктор: Employee[] buttons = stream.toArray(Employee[]::new);
  - Он сначала вызывает этот конструктор, чтобы получить массив правильного типа.
  - А затем он заполняет массив и возвращает его.

Обробка лямбда-виразів. Реалізація відкладеного виконання

- Лямбда-вирази застосовуються для ***відкладеного виконання.***
  - Якщо деякий код треба виконати відразу, це можна зробити без лямбда-виразів.
- Для відкладеного виконання коду имеется немало причин:
  - Выполнение кода в отдельном потоке.
  - Неоднократное выполнение кода.
  - Выполнение кода в нужный момент по ходу алгоритма (например, выполнение операции сравнения при сортировке).
  - Выполнение кода при наступлении какого-нибудь события (щелчка на экранной кнопке, поступления данных и т.д.).
  - Выполнение кода только по мере надобности.

Розглянемо простий приклад

- Нехай деяку дію потрібно повторити n разів.
  - Цю дію та кількість повторень передаються в метод repeat():
    `repeat(10, () -> System.out.println("Hello, World!"));`
  - Для передачі лямбда-виразу в якості параметру потрібно обрати (інколи – надати) функціональний інтерфейс.
- У даному прикладі достатньо застосувати інтерфейс Runnable.
  - Зверніть увагу: тіло лямбда-виразу виконується при виклику action.run().

```
public static void repeat(int n, Runnable action)
  for (int i = 0; i < n; i++) action.run();
}
```

- Потрібно сповістити дію, на якому самк кроці циклу вона повинна відбутись.
  - Треба обрати функціональний інтерфейс з методом, що примає параметр типу int та нічого не повертає.

Розглянемо простий приклад

- Стандартний інтерфейс для обробки значень типу int .

```
public interface IntConsumer {
  void accept(int value);
}
```

- Удосконалений вигляд методу repeat():
  - ```
    public static void repeat(int n, IntConsumer action) {
        for (int i = 0; i < n; i++) action.accept(i);
    }
    ```
- Виклик:
  - ```
    repeat(10, i -> System.out.println("Countdown: " + (9 - i)));
    ```

Виклик функціонального інтерфейсу

- У більшості мов ФП типи функцій є ***структурними***.
  - Для вказування функції, що перетворює два символьні рядки в ціле число, слугує тип, аналогічний одному з наступних:
  - Function2 <String, String, Integer>
  - (String, String) -> int.
- В языке Java цель функции объявляется с помощью функционального интерфейса вроде Comparator <String> .
  - В теории языков программирования это называется *номинальной типизацией*.
- Имеется немало случаев, когда требуется принять любую функцию без конкретной семантики.
  - И для этой цели имеется целый ряд обобщенных функциональных типов.
  - Пользоваться ими рекомендуется при всякой возможности.

Функциональный интерфейс	Типы параметров	Возвращаемый тип	Имя абстрактного метода	Описание	Другие методы
`Runnable`	Отсутствуют	`void`	`run`	Выполняет действие без аргументов или возвращаемого значения	
`Supplier<T>`	Отсутствуют	`T`	`get`	Предоставляет значение типа `T`	
`Consumer<T>`	`T`	`void`	`accept`	Употребляет значение типа `T`	`andThen`
`BiConsumer<T, U>`	`T, U`	`void`	`accept`	Употребляет значения типа `T` и `U`	`andThen`
`Function<T, R>`	`T`	`R`	`apply`	Функция с аргументом `T`	`compose`, `andThen`, `identity`
`BiFunction<T, U, R>`	`T, U`	`R`	`apply`	Функция с аргументами `T` и `U`	`andThen`
`UnaryOperator<T>`	`T`	`T`	`apply`	Унарная операция над типом `T`	`compose`, `andThen`, `identity`
`BinaryOperator<T>`	`T, T`	`T`	`apply`	Двоичная операция над типом `T`	`andThen`, `maxBy`, `minBy`
`Predicate<T>`	`T`	`boolean`	`test`	Булевозначная функция	`and`, `or`, `negate`, `isEqual`
`BiPredicate<T, U>`	`T, U`	`boolean`	`test`	Булевозначная функция с аргументами	`and`, `or`, `negate`

Найбільш використовувані функціональні інтерфейси

- Допустим, что требуется написать метод для обработки файлов, соответствующих определенному критерию.
  - Следует ли для этого воспользоваться описательным классом java.io.FileFilter или функциональным интерфейсом Predicate <FiIe>?
  - В этом случае настоятельно рекомендуется воспользоваться функциональным интерфейсом Predicate <File>.
  - Единственная причина не делать этого — наличие многих полезных методов, получающих экземпляры типа FileFilter.

- У большинства стандартных функциональных интерфейсов имеются неабстрактные методы получения или объединения функций.
  - Например, вызов метода Predicate.isEqual(a) равнозначен ссылке на метод a::equals, но он действует и в том случае, если аргумент a имеет пустое значение null.
  - Для объединения предикатов имеются методы по умолчанию and(), or(), negate().
  - Например, вызов метода Predicate.isEqual(a).or(Predicate.isEqual(b)) равнозначен выражению x -> a.equals (x) || b.equals(x).

Функциональные интерфейсы для примитивных типов

- обозначения **p, *q*** относятся к типам int, long, double; а обозначения *P* , *Q* — к типам Int, Long, Double

Функциональный интерфейс	Типы параметров	Возвращаемый тип	Имя абстрактного метода
BooleanSupplier	Отсутствует	boolean	getAsBoolean
*P*Supplier	Отсутствует	*p*	getAs*P*
*P*Consumer	*p*	void	accept
Obj*P*Consumer<T>	T, *p*	void	accept
*P*Function<T>	*p*	T	apply
*P*To*Q*Function	T	*q*	applyAs*Q*
To*P*Function<T>	T	T	applyAs*P*
To*P*BiFunction<T, U>	T, U	*p*	applyAs*P*
*P*UnaryOperator	*p*	*p*	applyAs*P*
*P*BinaryOperator	*p*, *p*	*p*	applyAs*P*
*P*Predicate	*p*	boolean	test

Реализация собственных функциональных интерфейсов

- Нередко бывают случаи, когда стандартные функциональные интерфейсы не подходят для решения конкретной задачи.
  - В таком случае придется реализовать собственный функциональный интерфейс.
- Допустим, что требуется заполнить изображение образцами цвета, где пользователь предоставляет функцию, возвращающую цвет каждого пикселя.
- Для преобразования (int, int) -> Color отсутствует стандартный тип.
  - В таком случае можно было бы воспользоваться функциональным интерфейсом BiFunction<Integer, Integer, Color> , но это подразумевает автоупаковку.
  - В данном случае целесообразно определить новый интерфейс :

```
@FunctionalInterface
public interface PixelFunction {
  Color apply(int x, int y);
}
```

Реализация собственных функциональных интерфейсов

- А теперь можно реализовать метод следующим образом:

```
BufferedImage createImage(int width, int height, PixelFunction f) {
  BufferedImage image = new BufferedImage(width, height,
          BufferedImage.TYPE_INT_RGB);
  for (int x = 0; x < width; x++)
  for (int y = 0; y < height; y++) {
    Color color = f.apply(x, y);
    image.setRGB(x, y, color.getRGB());
  }
  return image;
}
```

- При вызове этого метода предоставляется лямбда-выражение, возвращающее значение цвета, соответствующее двум целочисленным значениям:

```
BufferedImage frenchFlag = createImage(150, 100, (x, y) ->
x < 50 ? Color.BLUE : x < 100 ? Color.WHITE : Color.RED);
```

Область действия лямбда-выражений и переменных

- Тело лямбда-выражения имеет ту же самую область действия, что и вложенный блок кода.
  - В ней соблюдаются те же самые правила для разрешения конфликтов и сокрытия имен.
  - В теле лямбда-выражения не допускается объявлять параметр или локальную переменную с таким же именем, как и у локальной переменной:

```
int first = 0;
Comparator<String> comp =
    (first, second) -> first.length() - second.length();
// ОШИБКА: переменная first уже определена!
```

В теле метода нельзя иметь две локальные переменные с одинаковым именем. Следовательно, такие переменные нельзя внедрить и в лямбда-выражении.

Область действия лямбда-выражений и переменных

- Ще один наслідок з правила: ключове слово this в лямбда-виразі позначає параметр this методу, що створює лямбда-вираз:

```
public class Application() {
  public void doWork() {
    Runnable runner =
              () -> { ...; System.out.println(this.toString()); ... };
  ...
  }
}
```

У виразі this.toString() викликається метод toString() для об’єкту типу Application, а не экземпляра типа Runnable.

Область действия лямбда-выражения вложена в тело метода doWork(), а ссылка this имеет одно и то же назначение повсюду в этом методе.

Доступ к переменным из объемлющей области действия

- Нередко в лямбда-выражении требуется доступ к переменным из объемлющего метода или класса.

```
public static void repeatMessage(String text, int count) {
  Runnable r = () -> {
    for (int i = 0; i < count; i++) {
      System.out.println(text);
    }
  };
  new Thread(r).start();
}
```

Обратите внимание на доступ из лямбда-выражения к переменным параметров, определяемым в объемлющей области действия, а не в самом лямбда-выражении.

Розглянемо виклик repeatMessage("Hello", 1000);
// выводит слово Hello 1000 раз в отдельном потоке исполнения

Обратите внимание на переменные count и text в лямбда-выражении

- Код лямбда-выражения может быть выполнен спустя немало времени после возврата из вызванного метода repeatMessage(), когда переменные параметров больше не существуют.
- Лямбда-выражение имеет следующие три составляющие:
  1. Блок кода.
  2. Параметры.
  3. Значения трех свободных переменных, т.е. таких переменных, которые не являются параметрами и не определены в коде.
- В структуре данных, представляющей лямбда-выражение, должны храниться значения этих переменных (в приведенном выше примере — символьная строка "Hello" и число 1000).
  - В таком случае говорят, что эти значения захвачены лямбда-выражением.
  - Механизм захвата значений зависит от конкретной реализации.
  - Например, лямбда-выражение можно преобразовать в объект единственным методом, чтобы скопировать значения свободных переменных в переменные экземпляра этого объекта.

Дійсно кінцеві змінні

- Лямбда-выражение может захватывать значение переменной из объемлющей области действия.
  - Но для того чтобы захваченное значение было вполне определено, существует ограничение: в лямбда-выражении можно ссылаться только на те переменные, значения которых не изменяются.
  - Лямбда-выражения захватывают значения, а не переменные.
- Так, компиляция следующего фрагмента кода приведет к ошибке:

```
for (int i = 0; i < n; i++) {
  new Thread(() -> System.out.println(i)).start();
  // ОШИБКА: захватить переменную i нельзя
}
```

В лямбда-выражении могут быть доступны только локальные переменные из объемлющей области действия, называемые в данном случае действительно конечными.

- Действительно конечная переменная вообще не изменяется.
- Она является конечной или может быть объявлена как final.
- Это же правило распространяется и на переменные, захватываемые локальными внутренними классами.

- Переменная расширенного цикла for является действительно конечной: ее область действия распространяется на единственный шаг цикла.

```
for (String arg : args) {
  new Thread(() -> System.out.println(arg)).start();
  // Захватить переменную arg допустимо!
}
```

- Новая переменная arg создается на каждом шаге цикла и присваивается следующему значению из массива args.
  - С другой стороны, область действия переменной *i* в предыдущем примере распространяется на весь цикл.

- Как следствие из правила "действительно конечных переменных", в лямбда-выражении нельзя изменить ни одну из захватываемых переменных.

```
public static void repeatMessage(String text, int count, int threads) {
  Runnable r = () -> {
    while (count > 0) {
      count--; // ОШИБКА: изменить захваченную переменную нельзя!
      System.out.println(text);
    }
  };
  for (int i = 0; i < threads; i++) new Thread(r).start();
}
```

На самом деле это даже хорошо.
Если переменная count обновляется одновременно в двух потоках исполнения, ее значение не определено.

Обхід проблеми

- Проверку неуместных изменений можно обойти, используя массив единичной длины следующим образом:
  - int[] counter = new int[1];
  - button.setOnAction(event -> counter[0]++);
- Переменная counter является действительно конечной.
  - Она вообще не изменяется, поскольку всегда ссылается на один и тот же массив.
  - Следовательно, она доступна в лямбда-выражении.
- Безусловно, подобного рода код не является потокобезопасным и поэтому малопригодным, разве что для обратного вызова в однопоточном пользовательском интерфейсе.

Функции высшего порядка. Методы, возвращающие функции

- В языках функционального программирования функции, которые обрабатывают или возвращают другие функции, называются функциями высшего порядка.
  - хотя Java не является языком функционального программирования, основной принцип остается прежним.
- Допустим, что в одних случаях требуется отсортировать массив символьных строк в возрастающем порядке, а в других — в убывающем порядке.
  - С этой целью можно создать метод, получающий нужный компаратор следующим образом:

```
public static Comparator<String> compareInDirecton(int direction) {
  return (x, y) -> direction * x.compareTo(y);
}
```

В результате вызова compareInDirection(1) получается компаратор по возрастанию, а в результате вызова compareInDirection(-1) — компаратор по убыванию.
Полученный результат может быть передан другому методу, ожидающему подобный интерфейс, например, методу Arrays.sort():
Arrays.sort(friends, compareInDirection(-1));

Методы, изменяющие функции

- Узагальнимо попередній приклад для будь-якого компаратора

```
public static Comparator<String> reverse(Comparator<String> comp) {
  return (x, y) -> comp.compare(x, y);
}
```

- Этот метод оперирует функциями.
- Он получает исходную функцию и возвращает видоизмененную.
- Так, если требуется отсортировать символьные строки в убывающем порядке с учетом регистра, достаточно сделать следующий вызов:
- reverse(String::compareToIgnoreCase)
  - У интерфейса Comparator имеется метод по умолчанию reversed() , обращающий заданный компаратор точно таким же образом.

Приклади зміни коду

- Реалізація інтерфейсу Runnable за допомогою лямбда-виразу:
- До Java 8:

```
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Before Java8, too much code for too little to do");
    }
}).start();
```

- Після Java 8:

```
new Thread( () -> System.out.println("In Java8, Lambda expression rocks !!") ).start();
```

Приклади зміни коду

- Обробка подій за допомогою лямбда-виразів (Swing API)
- До Java 8:

```
JButton show =  new JButton("Show");
show.addActionListener(new ActionListener() {
     @Override
     public void actionPerformed(ActionEvent e) {
           System.out.println("Event handling without lambda expression is boring");
        }
     });
```

- Після Java 8:

```
show.addActionListener((e) -> {
    System.out.println("Light, Camera, Action !! Lambda expressions Rocks");
});
```

Приклади зміни коду. Ітерування по списку

- До Java 8:

```
List features = Arrays.asList("Lambdas", "Default Method", "Stream API", "Date and Time API");
for (String feature : features) {
   System.out.println(feature);
}
```

- Після Java 8:

```
List features = Arrays.asList("Lambdas", "Default Method", "Stream API", "Date and Time API");
features.forEach(n -> System.out.println(n));

// Even better use Method reference feature of Java 8
// method reference is denoted by :: (double colon) operator
// looks similar to score resolution operator of C++
features.forEach(System.out::println);
```

Використовуємо лямбда-вираз та функціональний інтерфейс Predicate

```
public static void main(args[]){
  List languages = Arrays.asList("Java", "Scala", "C++", "Haskell", "Lisp");

  System.out.println("Languages which starts with J :");
  filter(languages, (str)->str.startsWith("J"));

  System.out.println("Languages which ends with a ");
  filter(languages, (str)->str.endsWith("a"));

  System.out.println("Print all languages :");
  filter(languages, (str)->true);

   System.out.println("Print no language : ");
   filter(languages, (str)->false);

   System.out.println("Print language whose length greater than 4:");
   filter(languages, (str)->str.length() > 4);
}

 public static void filter(List names, Predicate condition) {
    for(String name: names)  {
       if(condition.test(name)) {
          System.out.println(name + " ");
       }
    }
  }
}
```

Вибірка та Stream API

```
public static void filter(List names, Predicate condition) {
    names.stream().filter((name) -> (condition.test(name))).forEach((name) -> {
        System.out.println(name + " ");
    });
}
```

```
Output:
Languages which starts with J :
Java
Languages which ends with a
Java
Scala
Print all languages :
Java
Scala
C++
Haskell
Lisp
Print no language :
Print language whose length greater than 4:
Scala
Haskell
```

Включення Predicate у лямбда-вирази

Інтерфейс java.util.function.Predicate дозволяє комбінувати кілька предикатів у один.

Він постачає методи, подібні до логічних операторів AND та OR, які називаються and(), or() та xor() відповідно.

```
Predicate<String> startsWithJ = (n) -> n.startsWith("J");
Predicate<String> fourLetterLong = (n) -> n.length() == 4;

names.stream() .filter(startsWithJ.and(fourLetterLong)) .forEach((n) ->
                  System.out.print("\nName, which starts with 'J' and four letter long is : " + n));
```

ДЯКУЮ ЗА УВАГУ!

Наступна тема: принципи побудови та тестування об’єктно-орієнтованого коду