אוניברסיטת
אריאל
בשומרון

תכנות מונחה עצמים – שיעור 7

סיכום מטלה 1 - בעיות
הרחבות והשלמות ב java:

- Lambda
- מחלקות פנימיות, סטטיות
- ממשקים גרפיים + אירועים
- שימוש בבסיסי נתונים JDBC

דיון במטלה 2 – גרפים

מטלה 1

מידול מדול מערכת שמתזמנת מעליות חכמות בבניין:
מבנה נתונים + אלג' offline

- הבנת המטלה?, אפיינו את הקשיים
- כתיבת תיעוד – Readme
- פרסום התוצאות – הבעיה המרכזית
- כתיבת Testers – נדרש תמיד

- המלצות למטלה הבאה (2):

Introduction

- Motivation: given a collection **c** and a function **f**, create a new collection **c'={f(x) for each x in c}**.
- Example: we have the collection called intStrings with the following elements: ("1", "2", "3", "4"), and the function Integer::parseInt which takes a string and parse it into an integer.

The old way

```
List<String> intStrings = Arrays.asList("1", "2", "3", "4");
List<Integer> integers = new ArrayList<>();
for(int i=0; i<intStrings.size(); i++){
    int parsed = Integer.parseInt(intStrings.get(i));
    integers.add(parsed);
}
```

Introduction

- This code works as intended and we get a new list, called “integers” with the elements (1, 2, 3, 4).
- However, as the amount of code and complexity of tasks scales up, such programming style can hurt our code’s readability.
- Instead, what if we could tell our collection to simply apply that function on each of it’s elements?
- Fortunately, we can do this, using the java Stream API!

Using stream API

```
List<String> intStrings = Arrays.asList("1", "2", "3", "4");
List<Integer> integers = intStrings.stream() // Create a stream of data out of intStrings
        .map(Integer::parseInt) // Apply the Integer::parseInt function to each element in the stream
        .collect(Collectors.toList()); // Collect the elements back into a list
```

Introduction

- **Functional programming** is a programming paradigm – a style of building the structure and elements of computer program.
  - Treats computation as the evaluation of mathematical functions.
  - Avoids changing-state and mutable data.
  - A different approach than the object oriented programming which relies on the object's state.

Functional vs Imperative Programming

אוניברסיטת אריאל בשומרון

Characteristic	Imperative approach	Functional approach
Programmer focus	How to perform tasks (algorithms) and how to track changes in state	What information is desired and what transformations are required?
State changes	Important	Almost non-existent
Order of execution	Important	Low importance
Primary flow control	Loops, conditionals and function calls	Function calls, including recursion
Primary manipulation unit	Instances of structures or classes	Functions as first-class objects and data collections

Functions as variables / parameters

- Some languages allows passing function as parameters to another functions, or instantiate functions as variables.
  - This is supported by C, C++ and C# among others
  - The Language-Integrated-Query (LINQ) of C# provides a simple mechanism, very similar to java's stream.
  - This mechanism works similarly in modern languages like python and javascript.

Prerequisites

אוניברסיטת אריאל בשומרון

In order to understand Stream API we need first:

1. Understand λ-expression
2. Be familiar with a group of java interfaces

Recall PS 5: Lambda Expressions

- Lambda (λ) expressions are supported since in Java SE 8.
- The λ expression is used to implement a **functional interface** – An interface with only one abstract method.
- In java, λ expressions are just a simple way (**syntactic sugar**) for instantiating a class and therefore are objects.
- Therefore, A lambda expression in java is a block of code that benefits from OO environment.
- Syntax: **(Argument list) -> {body}**
- **{body}** – Can be a single expression or a statement block
  - Single expression: The body is evaluated and returned

    Exp: (x, y) -> x + y // Takes 2 arguments, x and y, and returns x+y

    () -> 42 // Takes no arguments and returns 42.
  - Block: The body is evaluated like a method body with optional return statement

    Exp: (s) -> { System.out.println(s); } // Takes one argument and returns nothing
- λ-expression can be passed as an argument to another function.

Recall PS 5: Lambda Expressions

```
public interface Printer {
        void print(String
message);
}
```

```
public class PrinterImpl implements Printer{
   @Override
   public void print(String message) {
      System.out.println(message);
   }
}
public static void main(String[] args) {
   Printer p = new PrinterImpl()
   p.print("This is a message from PrinterImpl");
}
```

1. Define a class
2. Use the pre-defined class

```
public static void main(String[] args) {
   Printer p = new Printer() {
      @Override
      public void print(String message) {
         System.out.println(message);
      }
   };
   p.print("This is a message from p1");
}
```

Define and use at once

Equivalent, with lambda expression:

```
public static void main(String[] args) {
   Printer p = (message) ->
System.out.println(message);
   p.print("This is a message from p1");
}
```

The functional interface

Basic functional interfaces you should know (cont):

- Function<T, K>

  Simulates a function that takes an argument of type T and returns a value of type K by calling apply(T).

- Predicate<T>

  Simulates a function that takes an argument of type T and returns a boolean indicating whether T matches some specifications by calling test(T).

```
public interface
Function<T, K> {
    K apply(T t);
}
```

```
public interface
Predicate<T> {
    boolean test(T t);
}
```

```
Function<Integer, Double> sqrt = (x) ->
Math.sqrt(x);
System.out.println(sqrt.apply(36)); // Prints 6
```

```
Predicate<Integer> isEven = (a) -> a % 2 == 0;
System.out.println("isEven(2): " + isEven.test(2) + ", isEven(3): " +
isEven.test(3));   // isEven(2): true, isEven(3): false
```

The functional interface

```
public interface BinaryOperator<T> {
   T apply(T t1, T t2);}
```

Basic functional interfaces you should know (cont):

- BinaryOperator<T>

  Simulates a function that takes two arguments of type T and returns a value of type T by calling apply(T, T).

```
BinaryOperator<String> concat = (s1, s2) -> s1 + ' ' +  s2;
System.out.println(concat.apply("John", "Doe")); // Prints John Doe
```

The Stream API

- Stream is an API (Application Programming Interface) that supports functional-style operation on streams (or collections) of elements.
- Intermediate operations (such as map, filter, sorted) allows to perform manipulation on the stream object.
- Terminal operations (collect, forEach, reduce) allows to aggregate the elements of a manipulated stream into the desired result.
- Let’s take a look at some of these operations.
- Assuming the following definition:

```
List<Integer> lst = Arrays.asList(1, 2, 3, 4, 5);
```

Stream.map()

```
List<Integer> lst = Arrays.asList(1, 2, 3, 4, 5);
```

```
public interface Function<T, K> { K apply(T t); }
```

Stream<K> Stream.map(Function<T, K> f)

- Takes a Function<T, K> (let it be called **f**) as an argument.
- Returns a stream of elements composed of **f**(x) for each x in the stream data.
- Recall: Function<T, K>: takes T as an argument and returns K, where:
  - T is the type of the elements in our collection.
  - K is the type of the elements in the desired collection

```
Stream<Integer> squared = lst.stream().map((x) -> x * x); // Values are: (1, 4, 9, 16, 25)
```

Stream.filter()

```
List<Integer> lst = Arrays.asList(1, 2, 3, 4, 5);
```

```
public interface Predicate<T> {  boolean test(T t);  }
```

Stream<T> Stream.filter(Predicate<T> **f**)

- Takes a Predicate<T> **f** as an argument.
- Returns a stream of elements composed of each x in *lst* such that **f**(x)=true.
- Recall: Predicate<T>: takes T as an argument and returns a boolean, indicating whether that argument matches the predicate.
  - T is the type of the elements in our collection.

```
Stream<Integer> even = lst.stream().filter((x) -> x % 2 == 0); // Values are (2, 4)
```

Stream.forEach()

```
List<Integer> lst = Arrays.asList(1, 2, 3, 4, 5);
```

```
public interface Consumer<T> {  void accept(T t); }
```

void Stream.forEach(Consumer<T>)

- Takes a Consumer<T> as an argument.
- Recall: Consumer<T>: takes an element T and performs some operation.
- No value is returned (void function).
- Stream's way of performing 'for x : lst' loops.

```
lst.stream().forEach((x) -> System.out.println(x)); // Prints the elements of lst
```

Example

- Take a look at the **GameCharacter** class:

```
enum Continent {Kalimdor, Eastern_Kingdoms, Northrend}
```

```
public class GameCharacter {
    String name;
    String title;
    String city;
    int level;
    double hitPoints;
    Continent continent;

    public GameCharacter(String name, String title, String city, int level,
double hitPoints, Continent continent) {
        this.name = name;
        this.title = title;
        this.city = city;
        this.level = level;
        this.hitPoints = hitPoints;
        this.continent = continent;
    }
}
```

Example

- Assume the following definition:

```
List<GameCharacter> characters = Arrays.asList(
      new GameCharacter("Arthas Menethil", "Lich King", "Icecrown", 80, 1500,
Continent.Northrend),
      new GameCharacter("Thrall", "Warchief", "Orgrimmar", 90, 1200, Continent.Kalimdor),
      new GameCharacter("Jaina Proudmoore", "Lord Admiral", "Kul Tiras", 120, 1000,
Continent.Eastern_Kingdoms),
      new GameCharacter("Tyrande Whisperwind", "Priestess of Elune", "Teldrassil", 120,
1100, Continent.Kalimdor),
      new GameCharacter("Sylvanas Windrunner",  "Dark Ranger", "Undercity", 120, 1100,
Continent.Eastern_Kingdoms));
```

Example

- Create a list with the names of all the game characters from the continent of Kalimdor:

```
List<String> names = characters.stream()
    .filter((x) -> x.getContinent() == Continent.Kalimdor)
    .map(GameCharacter::getName) // This is java method reference, equivalent to: .map(c -> c.getName())
    .collect(Collectors.toList());
```

- Note that in order to create a list out of a stream we need to call:

```
.collect(Collectors.toList())
```

Example

- Find the average hit points of characters at level 120:

```
List<Double> hitPoints = characters.stream()
    .filter((character) -> character.getLevel() == 120) // Get all level 120 characters.
    .map(GameCharacter::getHitPoints) // Equivalent to .map(c -> c.getHitPoints()), returns hit points for each character.
    .collect(Collectors.toList());

double average = hitPoints.stream()
    .reduce(0.0, (acc, next) -> acc + next) / hitPoints.size();
// Identity (starting) element is 0, sum the elements in hitPoints and divide by hitPoints.size() (3) to get the average.
```

Example

- Print all characters, sorted by their hit points:

```
characters.stream()
        .sorted(Comparator.comparing(GameCharacter::getHitPoints)) // Equivalent to comparing(c -> c.getHitPoints())
        .forEach(System.out::println); // Equivalent to .forEach(c -> System.out.println(c))
```

- Note: we might want to implement Character::toString in order to get an informative result.
- Take a look at the usage of 'Comparator.comparing': this allows us to compare two object of type Character without actually implementing the 'Comparable' interface – useful when you would like to compare elements based on attributes different than the 'official' implementation.

Example

- Given a list of names, initialize a list of characters with the same names, title ‘Honorable Orc’, city ‘Orgrimmar’, level 15, continent ‘Kalimdor’ and hit points between 200 to 300:

```
List<String> names = Arrays.asList("Durotan", "Grom", "Garrosh",
"Garona", "Nazgrim", "Varok");
List<GameCharacter> honorableOrcs =
      names.stream().map((name)->new GameCharacter(name,
"Honorable Orc", "Orgrimmar", 15, 200 +(Math.random()*100),
Continent.Kalimdor)).collect(Collectors.toList());
```

דיון מקדים במטלה 2

מטלה 2 – עוסקת בגרפים

- נגדיר גרף: מכוון או לא מכוון, ממושקל או לא
- קודקודים, צלעות

פעולות

- קישורית
- מסלול קצר ביותר
- מספר מסלולים קצרים יחסית