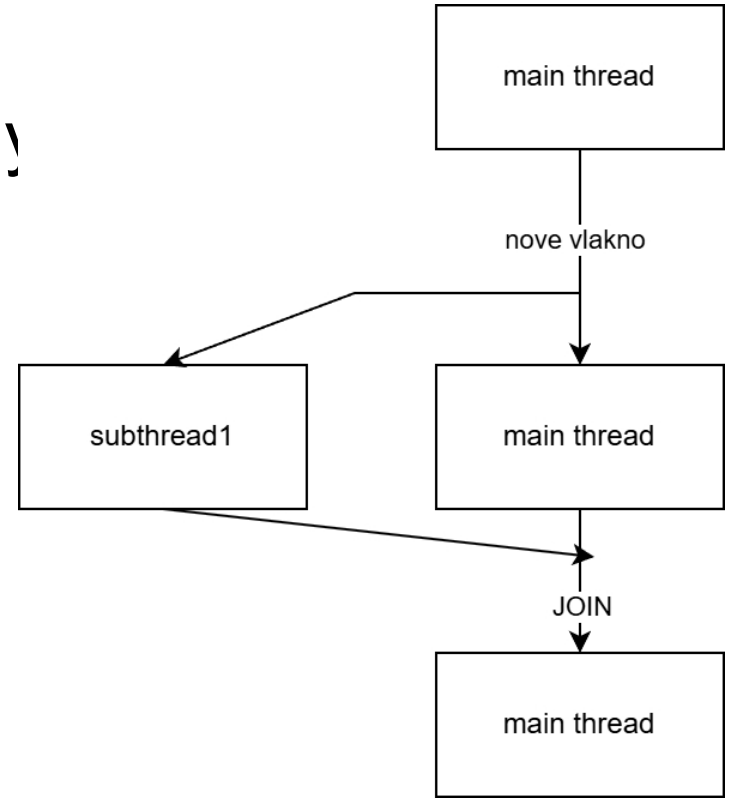
Object-Oriented Programming

7. Lecture
LS 2025/2026
Author: Juraj Petrík

Parallel processing using threads

- We do two or more things at once (concurrently
- Threads:
  - “Lightweight processes”
  - **Shared memory**
- Beware of determinism



- Implementing Runnable
- Swing, e.g. javax.swing.SwingUtilities.invokeLater

javax.swing.SwingUtilities.invokeLater

- Swing event handling runs on a separate thread (Event Dispatch Thread)
- Swing is not thread-safe
- https://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html
- SwingWorker ensures proper interaction with EDT

Streams

- Based on functional programming, declarative
- Easy parallel processing possible
- We start with a source, then we work (intermediate operations), and at the end there is a terminal expression; a stream can **only** be "used/passed" **once**
- They are lazy
- We do not modify the original data
- Collections support streams (.stream(), .parallelStream())

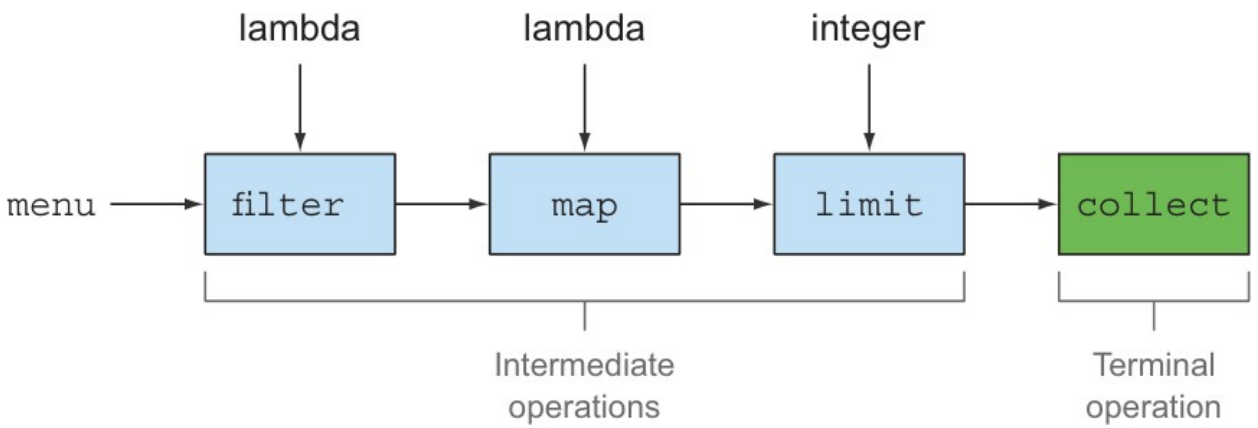
Streams

- Sequence of elements - source
- Data-processing operations

- Pipelining
- Internal iteration

- Computed on demand (DVD vs Netflix)

- Filter - we filter
- Map - transform
- Limit – we reduce
- Reduce – "we want the result"
- Collect – convert stream to another form



- List<Integer> steveList2 = steveList.stream().filter((steve) -> steve.getName().equals("Steve")).sorted(comparing(Steve::getName)).map(Steve::getHunger).toList();

- Declarative
- Composable
- Parallelizable

- distinct()
- .forEach(System.out::println)
- .takeWhile()/.dropWhile()
- .skip()
- .limit()
- .flatMap()
- .allMatch()/noneMatch()
- .findAny() (Optional<T>)

Streams (reduce)

- .reduce(initial_value, Operator)
- int product = numbers.stream().reduce(1, (a, b) -> a * b)

- Optional! (also with flatten, for example) - .isPresent(), .isEmpty()

Storing objects

- Saving the state of an object
- Using:
  - Serialization:□□ sr minecraft.player.SteveJHu□□h Z godModeI hungerI maxHungerx
  - Plain Text: e.g. (csv like).: Juraj,1,1,1..

Serialization

- The entire object graph is saved automatically
- All or nothing
- Implements Serializable
- If I don't want to/don't know how to serialize - transient

Serialization and **IO** streams

- 1. Create FileOutputStream – where we write
- 2. Create ObjectOutputStream – what we write
- 3. Write object .writeObject()
- 4. Close ObjectOutputStream – FileOutputStream will also close automatically

Deserialization

- The constructor is **not executed**
- 1. Create FileInputStream – where we read from
- 2. Create ObjectInputStream – where we read
- 3. Read the object .readObject()
- 4. Close ObjectInputStream – FileInputStream will also close automatically

Deserialization

- 1. The object is read from the stream
- 2. JVM determines the class type (stored)
- 3. JVM finds and loads the appropriate class
- 4. Creates an object on the heap without calling the constructor
- 5. If there is a non-deserializable class somewhere in the graph, the constructor(s) are executed, beware of chain reactions (super)
- 6. Class attributes are assigned; if they were transient, they are null, 0, etc.

What if the class changes? Problems

- Deleting an attribute
- Changing types
- Change from non-transient to transient
- Moving up in the inheritance hierarchy
- Removing Serializable
- Changing an attribute to a static attribute

What if the class changes? What is (usually) okay

- Adding a new attribute
- Adding a new class to the inheritance hierarchy
- Removing classes from the inheritance hierarchy
- Changing access modifiers
- Transient to non-transient

Serialization - serialVersionUID

- serialver Steve
- static final long serialVersionUID = -5849794470654667210L;

- However, this does not solve anything for the programmer; the programmer has to deal with everything else themselves.

!Quiz time