Course Application Design

Creating beautiful and reliable applications
Multithreading and Functional Programming

Michiel Noback
Institute for Life Sciences and Technology
Hanze University of Applied Sciences

Java™

Part one

Classic Multithreading

Why Multithreading

- Because you want to use more than one of the 8 (16, 32, …) available CPU cores on your machine…
- Because you don’t want your GUI application to “hang” when a (long-running) process is running

What is a Thread, anyway

- A Thread is a “thread of execution” – an independent call stack within a single instance of the Java Virtual Machine
- It shares main memory (heap) with other threads in the JVM

The threat of Threads

- There are three major issues to consider when creating multithreaded applications:
  - Concurrent modification of shared objects – **DATA CORRUPTION**
  - You never know which thread will be selected to run, and you do not know the order in which they will finish - **UNCERTAINTY**
  - Deadlocks – threads endless waiting on each other – **HANGING APP**

The basic method

- In its most basic form, you need two classes/objects
  - A Thread instance
  - A class implementing the Runnable interface
- Pass the Runnable to the Thread and call start.
- Here is the code.

A Runnable worker

```java
package nl.bioinf.multithreading;

public class SimpleWorker implements Runnable {
    private String name;

    public SimpleWorker(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        System.out.println("S Worker " + name + " doing my thing");
    }
}
```

Running Threads with a Worker

```java
package nl.bioinf.multithreading;

public class SimpleMultithreadingDemo {
  public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
      SimpleWorker worker = new SimpleWorker("_" + i + "_");
      Thread t = new Thread(worker);
      t.start();
    }
  }
}
```

Note the order of execution – this may be different every time!

```
SimpleRunnableWorker _0_ doing my thing
SimpleRunnableWorker _2_ doing my thing
SimpleRunnableWorker _3_ doing my thing
SimpleRunnableWorker _1_ doing my thing
SimpleRunnableWorker _4_ doing my thing
```

A Runnable worker with shared data

```
public class SimpleWorker implements Runnable {
    private static int operationCount;

    public static void incrementOperationCount() {
        operationCount++;
    }

    @Override
    public void run() {
        incrementOperationCount();
        System.out.println("S Worker " + name + " doing my thing");
        System.out.println("Operation count =  " + operationCount );
    }
}
```

Running Worker Threads with shared data

```
for (int i = 0; i < 5; i++) {
  SimpleWorker worker = new SimpleWorker("_" + i + "_");
  Thread t = new Thread(worker);
  t.start();
}
```

```
S Worker _0_ doing my thing
S Worker _2_ doing my thing
Operation count =  3
S Worker _1_ doing my thing
S Worker _3_ doing my thing
Operation count =  3
Operation count =  4
Operation count =  4
SimpleWorker _4_ doing my thing
Operation count =  5
```

Not good!

A synchronized block to define “atomic transactions”

```java
public void run() {
    synchronized (SimpleWorker.class) {
        incrementOperationCount();
        System.out.println("S Worker " + name + " doing my thing");
        System.out.println("Operation count =  " + operationCount);
    }
}
```

```
S Worker _0_ doing my thing
Operation count =  1
S Worker _4_ doing my thing
Operation count =  2
S Worker _3_ doing my thing
Operation count =  3
S Worker _2_ doing my thing
Operation count =  4
S Worker _1_ doing my thing
Operation count =  5
```

Synchronized

- Putting the synchronized keyword on a method or block makes it ***executable by only one thread at a time***
- Have a look at http://www.baeldung.com/java-synchronized for further info on where and how to use the synchronized keyword.

Thread to Thread communication

- Have a look at class `InterThreadCommunication` in the repo accompanying this course. Run it a few times and see the different outputs.
- Key players in this game are methods
  - **`Object.wait()`** Causes the current thread to wait until another thread invokes the notify()
  - **`Object.notify()`** Wakes up a single thread that is waiting on this object's monitor
  - **`Thread.sleep()`** Causes the current thread to suspend execution for a specified period in milliseconds

Thread to Thread communication

- When designing an application with a single controller object and a larger set of worker objects, it is a good idea to implement the *Observer pattern* to get notified when a worker is finished with a job - you do this using a `synchronized` setting of course

Thread pools

- Whenever you are going to some serious multicore programming you should make use of thread pools to handle them
- Have a look at this tutorial https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice

Parallel streams

- With Java 8+, you can also make use of parallelStream (see next section), e.g.

```
myList.parallelStream().sum();
```

Part two

Lambdas

What is a Lambda

- A Lambda expression is a block of code with parameters
- The general syntax is this

```
(args...) -> {<Block to execute with args>}
```

A first example

- Consider this Comparator of Strings

```
Comparator<String> comp = new Comparator<String>() {
    @Override
    public int compare(String first, String second) {
        return Integer.compare(first.length(), second.length();
    }
};
```

- Is *exactly* the same as this

```
Comparator<String> comp =
    (String first, String second) ->
        Integer.compare(first.length(), second.length());
```

Several equivalent notations

- When type can be inferred you can omit these
- When multiple statements, use braces and a return statement

```
Comparator<String> comp;
comp =
(String f, String s) -> Integer.compare(f.length(), s.length());
//Same as
comp = (f, s) -> Integer.compare(f.length(), s.length());
//Same as
comp = (f, s) -> {return Integer.compare(f.length(), s.length());};
```

Lambdas usually represent *functional interfaces*

- A functional interface is an interface with a single abstract method.
- The Comparable interface is an example, but you can also define your own

A custom *functional interfaces*

- Here is an interface that is supposed to do something with two numbers; what it is remains for the implementers

```
@FunctionalInterface
public interface NumberCombiner {
    int combine(int i, int j);
}

NumberCombiner nc = (i, j) -> i*j;
System.out.println("2 * 3 = " + nc.combine(2,3));
nc = (i, j) -> (int)Math.pow(i, j);
System.out.println("2 ^ 3 = " + nc.combine(2,3));
```

Accepting *functional interfaces*

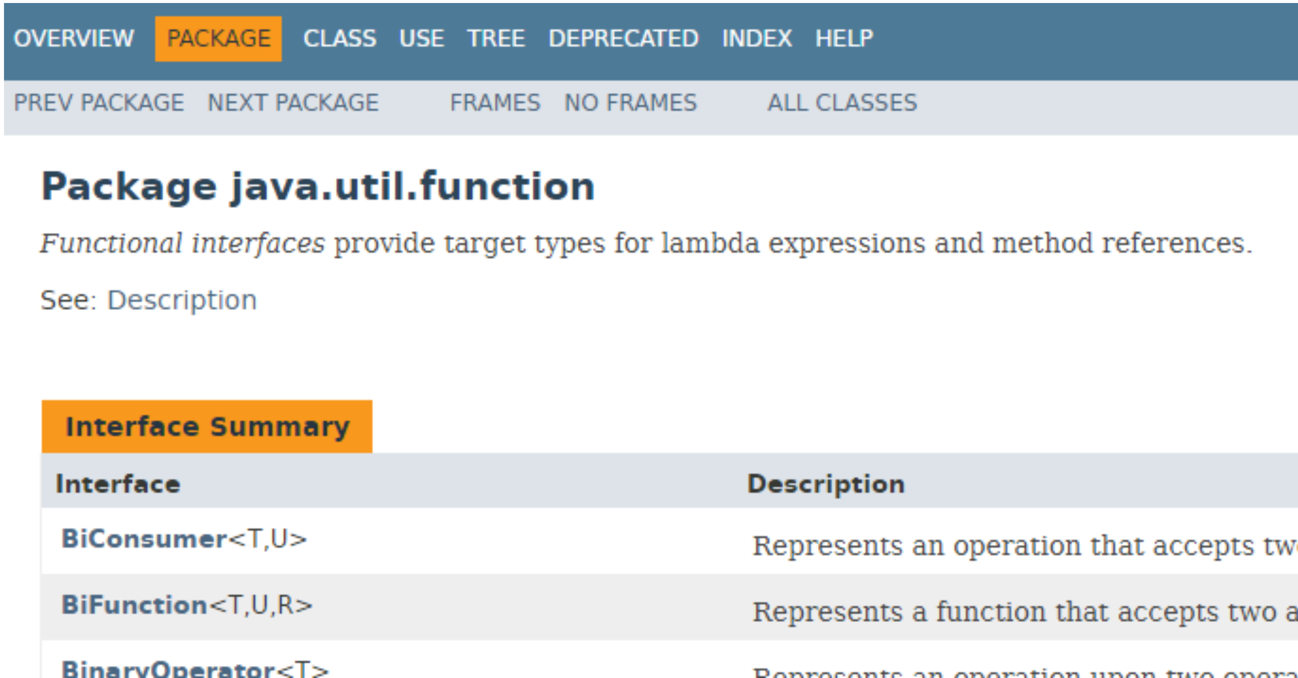
- Here is an regular method that accepts as argument a functional interface

```
public void combineAndPrint(int i, int j, NumberCombiner combiner) {
    System.out.println("combining i = " + i
              + " and j = " + j
              + " to " + combiner.combine(i, j));
}

//call
combineAndPrint(3, 4, (i, j) -> (int)(i * j * Math.random()));
```

Functional interfaces of `java.util.function`

- Java probably has already defined most interfaces you may be interested in
- Here follows an example of one of the simplest Fis – the Predicate


OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES ALL CLASSES

Package java.util.function

Functional interfaces provide target types for lambda expressions and method references.

See: Description

Interface Summary

Interface	Description
BiConsumer<T,U>	Represents an operation that accepts tw
BiFunction<T,U,R>	Represents a function that accepts two a
BinaryOperator<T>	Represents an operation upon two opera

Filtering using Predicate<T>

- Given a list of User objects (see package `nl.bioinf.fp_exercise`), and this method

```
public static void filterUsers(Predicate<User> p) {<implementation>}
```

- You can pass it something like this (argument user needs no type because inferred from context

```
filterUsers((user) -> user.getAddress() != null);
filterUsers((user) -> user.getNumberOfLogins() > 10);
```

public interface Function<T, R> { … }

- Given a list of User objects (see package `nl.bioinf.fp_exercise`), and this method

```
public static void filterUsers(Predicate<User> p) {<implementation>}
```

- You can pass it something like this (argument user needs no type because inferred from context

```
filterUsers((user) -> user.getAddress() != null);
filterUsers((user) -> user.getNumberOfLogins() > 10);
```

Some examples of FIs

Interface	Description
BiFunction<T,U,R>	Represents a function that accepts two arguments and produces a result.
BiPredicate<T,U>	Represents a predicate (boolean-valued function) of two arguments.
Consumer<T>	Represents an operation that accepts a single input argument and returns no result.
Function<T,R>	Represents a function that accepts one argument and produces a result.
Predicate<T>	Represents a predicate (boolean-valued function) of one argument.
Supplier<T>	Represents a supplier of results.
UnaryOperator<T>	Represents an operation on a single operand that produces a result of the same type as its operand.

Part three

Streams (and Lambdas)

The Stream API

- The main area of application of Lambdas is the Stream API
- Streams let you process, change, transform and collect objects from collections and arrays in a series of coupled operations (in parallel!)
- The next few examples use this array

```
String[] words =
   {"arg", "bah", "yeah", "howwie", "aw", "whoa", "yuck"};
```

A first stream

- First, a stream should be coupled to your collection

```
Arrays.stream(words)
```

- The stream itself does nothing; you need to *chain* operations to it

```
Arrays.stream(words)
    .forEach(w -> System.out.println(w));
```

Stream method types

The Stream interface specifies roughly two classes of methods; those that return another stream, such as map(), peek(), skip()

```
Arrays.stream(words)
      .map(w -> w.length())
      .forEach(System.out::println);
```

The other class is those that do NOT return a stream, such as reduce(), max(), min()

```
Arrays.stream(words)
      .map(w -> w.length())
      .max(Integer::compare).get()
```

Chaining constraints

Only those Stream methods that return another stream can be chained. The other methods always form the end of the line.

Chaining possible

```
Arrays.stream(words)
     .map(w -> w.length())
     .max(Integer::compare).get()
```

No stream produced!

Get() is a method of class Optional!

Collecting a collection

If your stream produces a collection of elements (not a single object), you probably want to collect these into a List or something

```
List<Integer> collected = Arrays.stream(words)
        .map(String::length)
        .distinct()
        .collect(Collectors.toList());
System.out.println("collected = " + collected);
```

It is good form to put your stream operations indented on new lines

Reductions

- Of course, max() and min() are already reductions, but so much more is possible.
- Here is the sum of all lengths

```
Optional<Integer> sum = Arrays.stream(words)
        .map(String::length)
        .reduce(Integer::sum);
         //same as
         //.reduce((x, y) -> x + y);
System.out.println("sum.get() = " + sum.get());
```

More complex Reductions

- But how about counting the frequencies of numbers?

```
Map<Integer, Long> collect = Arrays.stream(words)
    .map(String::length)
    .collect(Collectors.groupingBy(
        Function.identity(), Collectors.counting())
     );
```

Want to know more?

- Read this excellent intro