ENTERPRISE JAVA PROGRAMMING / PROGRAMMING II

MODULES CT545 / CT875, SEMESTER 2, ACADEMIC YEAR 2020-2021
NATIONAL UNIVERSITY OF IRELAND, GALWAY

Lecture 6

Lecturer: Dr Patrick Mannion

TOPICS

- Java 8 Lambda Expressions
- Java 8 Streams
- Local variable type inference using 'var' (Java >=10)

Java >=8

- A significant improvement of Java >=8 over older Java versions is its better support for *functional programming*
- Functional programming (FP) concepts are crucial for modern programming, in particular with a focus on large-scale data processing and parallel computing
- Java is not a FP language and Java 8 is still far from optimal for FP (look at, e.g., Scala, Haskell, Swift, F# instead for better support), but improved a lot over Java <=7
- A few benefits & use cases of FP in general (but some aren't enforced by Java 8):
    - Easier programming if programmer doesn't need to care about *side effects*
    - Much easier parallelization of task, e.g., because (pure) FP doesn't allow for modification of shared resources (-> *immutability*)
    - Modern "Big data" frameworks like Apache Spark and "Big data" programming models such as MapReduce are heavily geared towards the FP paradigm
    - FP makes it easier to formulate -->bulk, -->aggregate and -->pipelined operations on data collections and streams
    - Easier optimization of complex chains of operations (often, but not always, supported in FP)

- This module doesn't cover real FP, except for a few important aspects introduced in Java 8: Lambda expressions ("lambdas") and Java 8 Streams…
- For anyone interested in reading up on functional programming:
    - https://www.educba.com/functional-programming-vs-oop/
    - https://www.codenewbie.org/blogs/object-oriented-programming-vs-functional-programming

Java >=8

- Java >=8 isn't a FP language, but you can use the following guidelines to use it in a somewhat functional way
- At the same time, the guidelines below are good advice for working with parallel/distributed operations on data (avoidance of race conditions)
- Guidelines
    - Prefer immutability, avoid mutation (in-place modification of existing data, e.g., variables, instead of creating a new (modified) object), where possible
    - Avoid global variables (e.g., public static fields, apart from constants)
    - Avoid *side effects* of functions. The results of functions/methods should ideally only depend on their arguments. Methods/functions should not manipulate shared state, as far as possible.
    - Where possible, rely on inherently "parallel" data structures instead of designing parallel solutions manually (e.g., use *parallel streams* with Java 8 -> later today in the lecture)

Anonymous classes

- We motivate Lambda Expressions by quickly revising an already known concept from Java <=7
- You have seen that Java allows for inner classes without a name - a.k.a. *anonymous classes*
- They are often used in the context of GUI programming (e.g., for event handling callbacks), but are also important more generally for creating function objects (sometimes called "functors", although that term is ambiguous): objects whose sole purpose is to store or pass around a function:

1) Suppose you want to call a method with some other function fn as an *argument* (*), e.g.,

```
x = someMethod(a,b,fn,c);
```

Not *directly* possible with "traditional" (pre-8) Java, since methods/functions aren't objects!

2) Solution: we "wrap" fn inside of a new object o (a *functor*) whose class C has fn as a method.
   Inside the body of someMethod, we then simply can call o.fn(...)

3) Since we need class C only for this purpose, it doesn't need a name -> anonymous classes...

(*) Note: we are talking about using the *entire* function fn as argument, not about using the result of calling fn as argument. In Java pre-8, functions are not *first-class citizens*, so this is not directly possible there.

Anonymous classes

- Such function objects using anonymous classes are the Java <=7 precursor of lambda expressions.

- Recall, e.g.,:

Creates a new object of an anonymous class which implements interface ActionListener

```
menuItem.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent event) {   // the function "wrapped" inside the object
          ...Code of the function...
   }
});
```

- A function (like addActionListener) which takes another function (like actionPerformed) as an argument or returns a function as result is called *higher-order function*

  (No need to understand the purpose of addActionListener at this time - but important to see that it indirectly takes another method as argument!)

- Many contemporary JVM-based frameworks, such as Apache Spark, make heavy use of higher-order functions.

Anonymous classes

- Recall:
- We want to pass function actionPerformed into method addActionListener as an argument
- Since this is not directly possible in Java pre-8, a new object (the functor which "wraps" actionPerformed) is created and used as argument for method addActionListener
- The new object is an instance of an unnamed (i.e., anonymous) class which implements interface ActionListener
- The interface needs to declare the function which we want to pass into addActionListener as an argument, that is, function actionPerformed
- The only purpose of this anonymous class is to provide a method implementation (of method actionPerformed(...) which is specified in interface ActionListener) and to pass this method on to some other method (here: addActionListener(...)).
- The example is from GUI programming (Java Swing), but it is a universal pattern which is seen in many other use cases too

Anonymous classes

This is how the interface from the previous example is defined:

```
public interface ActionListener extends EventListener {
        public void actionPerformed(ActionEvent e); // abstract method
 }
```

When you create the function object as an instance of the anonymous class (that is, using new ActionListener() {…}), you provide an implementation of abstract method actionPerformed.

- In Java 8, such an interface is called a *functional interface*
- Functional interfaces as the one above are also used for determining the type of lambda expressions. I.e., even with lambda expression, in Java 8 we unfortunately will still require some of the machinery of anonymous classes…

Anonymous classes

- You could likewise use an ordinary class (in the example before, you'd create a normal class which implements ActionListener and pass on an object of this class to menuItem.addActionListener())
- But that would require even more coding overhead. And since you don't require this class elsewhere, the class name would be unnecessary.

- Remark: anonymous classes could also be used to specify more than one method. Also, instead of an interface, the anonymous class could also extend a class, using exactly the same syntax pattern:

```
new Superclass() {
        (...overriding/implementing methods of SuperClass...)
}
```

Lambda expressions

- A good thing about anonymous classes is that they can be used with knowledge of "traditional" (pre-8) Java only
- However, they have shortcomings:
    - Verbose, difficult to read, not very intuitive
    - Runtime overhead, object creation from anonymous class is quite costly
- In Java >=8, we typically use *Lambda expressions* instead to "pass around functions"...

Lambda expressions

- Lambda expressions can be seen as anonymous functions (functions without names) which are also objects (e.g., they can be stored in variables, object and class fields, arrays, lists and other data structures, used as method arguments…)
- Since we can use them everywhere where also an ordinary object can be used (e.g., store them in variables, fields, use them as method arguments…), they are first-class citizens of Java, making functions also first-class citizens in Java >=8 (well, sort of).
- They are based on one of the oldest concepts in Computer Science (*lambda abstractions*) and they are also available in many other modern programming languages, e.g. Python, Scala or C# (in other forms)
- A first example: a Lambda expression in Java 8 which represents the *addition* operation:

```
(int x, int y) -> x + y
```

This function takes two integer arguments. Its result is the sum of these arguments.

Lambda expressions

- A closer look...

Result expression (i.e., how to compute the result from the given arguments)

```
(int x, int y) -> x + y
```

Parameters (with types, here: int)

- Parameters are optional. If there are no parameters, write () before the arrow.
- The result expression is actually evaluated only when we apply the Lambda expression to some arguments.
- Lambda expressions can use arbitrary blocks of code to produce a result, e.g.,

```
x -> {
        double d = 5.0;
        System.out.println("x: " + x);
        doubld dx = d * x;
        return dc - 7.2;
      }
```

Lambda expressions

- Effectively, Lambda expression represent methods directly *as objects*, i.e., they can be seen as a form of Java functors (although they are usually not called "functors" in this context, as the term "functor" has a different meaning in Functional Programming, where Lambda expressions originate from. A FP functor is anything that can be “mapped over”. “Mapping” in FP is a 1:1 transformation over a list to a new version of that list)
- lambda expressions can be used basically like any ordinary Java object. E.g., they can be assigned to variables and fields, passed on as arguments of methods, or even used as arguments into other Lambda expressions…

Lambda expressions

- You can provide a Lambda expression everywhere where an object with a matching *functional interface* as type is expected
- A functional interface is an interface with a single abstract method (just as the interfaces which we have used to create anonymous classes)
- Functional interfaces serve as the *types* of the Lambda expression (recall that in Java, types are primitive types (such as int) or references types (i.e., types of objects), the latter corresponding to classes or interfaces).
- A Lambda expression can be seen as an instance of the implementation of its functional interface.
- Functional interfaces are often, but not necessarily, generic (e.g., using the types of the Lambda expression's parameters as type parameters).
- The abstract method declared in the functional interface corresponds to the respective Lambda expression in terms of its parameters and result type (see examples on the following slides).

Lambda expressions

- Java 8 has a number of pre-defined functional interfaces, e.g., for *predicates* (Boolean Lambda expressions, see below), *consumers*, or java.lang.Runnable (!). E.g.,

The *type* of the Lambda expression,
a (generic) functional interface

Lambda expression (in blue)

```
Predicate<String> pred = (String s) -> { s.equals("Tom") };
```

- You can find these standard functional interfaces in package java.util.function

- You can also create your own functional interfaces, as follows:

```
@FunctionalInterface
```

an annotation

```
public interface MyFnInterface {
        public Double doSomething(Double x);  // the method to which the Lambda
                                              // expression, whose type is MyFnInterface,
                                              // corresponds.
}
```

The next slide shows how to use interface **MyFnInterface** as the type of a corresponding Lambda expression...

Lambda expressions

- Once we know/have the functional interface, we can use it as the type of the Lambda expression. E.g., as type of a variable which can store the Lambda expression, or as the type of a parameter of some other method which takes the Lambda expression as an argument (as in the example below).

Example (using the functional interface from the previous slide):

```
Double someOtherMethod(MyFnInterface fn) {
        return fn.doSomething(4.3);  // we evaluate the Lambda expression
}

Double r = someOtherMethod((Double x) -> x*2.0);
System.out.println(r);
```

- Keep in mind that we are talking about the Lambda expression itself here, not about its result!

Lambda expressions

- We can also evaluate Lambda expressions almost directly (*)...
- Example:

```
import java.util.function.Function;
// https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html
```

A built-in functional interface

```
Function<Double, Double> squareFn = a -> a * a;
```

Type of the parameter

Type of the result

```
System.out.println(squareFn.apply(5.0));
```

(prints 25)

(*) ...but there's no way around functional interfaces in Java, which makes the use of Java's Lambda expressions more "wordy" than Lambda abstractions in real FP languages.

Lambda expressions

- In Computer Science, Lambda expressions (a Java 8 term) are more commonly known as *Lambda abstractions* (but these are a mathematical concept and have a somewhat different syntax and semantics)
- Lambda abstractions are a corner stone of functional programming
- Their first use was as the constituents of the Lambda calculus
- They are supported by many modern programming languages (including, e.g., Python, JavaScript and C++ 11), not just "real" functional programming languages

Lambda expressions

- What can you do with Lambda expressions?
- Lots of things... Just a few examples:
- Use cases similar to function objects of anonymous classes but easier to use.
- E.g., they are useful for using (unevaluated) functions as if they where ordinary objects. E.g., if you want to create a callback (e.g., for event handling), or to deploy a task to another, remote, computer, you can wrap this task in a Lambda expression and submit it to the remote machine.
- They are used with *higher-order functions (functions which take functions as arguments)*, in particular for manipulation, filtering and aggregation of collections (aggregate and other bulk operations).
  Pipelines of such manipulation functions are a crucial concept of modern Data Analytics.
- They generally lead to more compact and readable code
- Several modern "Big data" processing frameworks are based on functional programming principles (e.g., Apache Spark) and make heavy use of Lambda expressions for data parallelism...

Lambda expressions

- *Data parallelism* is a concrete use case for Lambda expressions.
- Assume you can split your (large amount of) data into several segments…
- Next, you want to apply the same function fn to each of these segments (all applications of fn to be performed *in parallel*, independently from each other)
- You can use *Java 8 streams* (-->more details later in this lecture) for the actual parallelization (*). You need to tell them which function fn to apply. We can represent fn conveniently as a Lambda expression.
- Pseudocode (not real Java):

```
foreEachInParallel(segment in data) {
      apply(fn, segment);   //applies function fn to the data segment
}
```

Use a Lambda expression to represent fn

(*) Of course, you could alternatively parallelize this pseudo-code example "manually" using conventional multithreading, i.e., data parallelism is also possible with older Java versions, it is just somewhat more cumbersome to program

Question: do you need to care about race conditions in the above scenario?

Lambda expressions

- Another application of Lambda expressions is in the context of *concurrency*: creating instances of task classes ("runnable object") (which are then used with conventional Java threads for parallelism)…
- You can do this using a named class (see lectures about concurrency), but often you want to avoid the overhead of creating a new task class. Anonymous classes could also be used, but still very verbose.
- So we instead use Lambda expressions now, using interface Runnable directly as their functional interface…

Lambda expressions

```
public class RunnableTest {
  public static void main(String[] args) {

     // Anonymous class which implements interface Runnable (old-style Java)
     Runnable r1 = new Runnable() {
        @Override public void run(){
            System.out.println("Hello world one!");
       }
     };

     // Lambda Runnable (Java 8) as a better alternative to the above:
     Runnable r2 = () -> System.out.println("Hello world two!");       // much more concise and easier to read

      ... (we can now take object r1 or r2 and pass it as argument into a Thread constructor as usual - omitted)
  } }
```

Lambda expressions

// Another example, also using another new concept: *Java 8 Streams*.
// (For full example see http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html)

```
// Calculate average age of pilots old style (Java <=7):

System.out.println("== Calc Old Style ==");
int sum = 0;
int count = 0;

for (Person p:pl){
  if (p.getAge() >= 23 && p.getAge() <= 65){
    sum = sum + p.getAge();
    count++;
  }
}
long average = sum / count;
System.out.println("Total Ages: " + sum);
System.out.println("Average Age: " + average);
```

Lambda expressions

```
// Calculating the sum of persons' ages new style (with Java 8 Streams (*)):

Predicate<Person> allPilots = p -> p.getAge() >= 23 && p.getAge() <= 65;

long totalAge = pl.stream()
        .filter(allPilots)
        .mapToInt(p -> p.getAge())
        .sum();

// Get average of ages, using a "parallel stream"
OptionalDouble averageAge = pl
        .parallelStream()
        .filter(allPilots)
        .mapToDouble(p -> p.getAge())
        .average();
```

A Lambda expression with Boolean result type (so-called predicate) - ideal for *filtering* things

(*) we will see on the next slides in more detail what stream(), parallelStream(), map(), etc mean. Until then, think of Java 8 Streams as a kind of sequence of objects.

Java 8 Streams

- On the previous slide, you have seen an example for another important addition which came with Java 8: *Java 8 Streams*
- **Not to be confused with Java's Input/Output (I/O) streams!** (although there are a few similarities)
- Java 8 Streams are part of java.util.stream. In this lecture, we refer to them simply as "streams".
- They are not actual data structures (streams don't store their data items), but a way of accessing and computing with data iteratively (or in parallel)
- Each stream has a *source* where its actual data comes from (e.g., a collection data structure such as an ArrayList or a Set)
- A stream doesn't replicate and store the data from its source but makes it accessible and transforms it in specified ways
- The original data is being operated on in a sequence of stream *operations*, transforming the data in the stream step-by-step and finally computing some end result

Java 8 Streams

- Features of stream over ordinary data structures (arrays, collections and iterators):
- Many stream operations work *lazily*:
  They *materialize* (=actually compute) intermediate results only as far as required for computing the end result!
- They don't manipulate their respective data sources - the original data is untouched, the stream just iteratively computes a result from the data or some other stream.
- They can be unbounded, even infinite (due to their laziness)

Java 8 Streams

- Stream *operations* provide computations on the stream elements
- E.g., *filtering*, *mapping* or *traversing*
- *Intermediate* stream operations create a new stream from an existing stream, using some specified stream transformation operation (e.g., filtering or mapping). Typically, multiple intermediate operations are chained together
- *Terminal* operations return a final non-stream result (e.g., individual data items, such as a number, or a collection, such as alist), or nothing / void (*)
- Once traversed or otherwise operated on, existing streams cannot be reused (**), but you can create a new stream from an existing stream, see above.
- Stream operations are typically parameterized with Lambda expressions. The Lambda expression specifies what the stream operation should do with individual items in the stream (e.g., the filter condition or the map expression), see examples on the next slides.

(*) What could such a "nothing" result be good for? Is it a problem if the terminal operation doesn't yield an actual result?

(**) Another thing Java 8 Streams have in common with iterators! However, iterators are much more restricted.

Java 8 Streams

- Examples for stream operations:

  - *filter*, e.g.,
    ```
    Stream<Person> personsOver18 = persons.stream()
        .filter(p -> p.getAge() > 18);
    ```
    Creates a stream from the source collection stored in variable persons (e.g., an ArrayList)

    A Lambda expression which parameterizes the filter operation

  - *map*, e.g.,
    ```
        Stream<Student> myStream = persons.stream()
            .map(person -> new Student(person));
    ```

  (assume persons is a list of objects of some class Person)

  - *forEach*, e.g.,
    ```
    someStream.forEach(p -> p.setLastName("Doe"))    // forEach is a terminal operation
    ```

    Careful - uses a side-effect (why might this be a problem?)

Java 8 Streams

- The operation is typically parameterized with a Lambda expression, but can alternatively also use as parameter an ordinary method, using a *method reference*, e.g., Person::compareAge
- The result of the overall map, filter, etc. stream operation is determined by the outcome of the iterative application of its Lambda expression on all the stream's data items.
- E.g., if the outcome of p -> p.getAge() > 18 in the *filter* operation on the previous slide is "false" for some concrete person p, that person does not become part of the result (the new stream) produced by filter(p -> p.getAge() > 18)
- map(person -> new Student(person)) converts a stream of person objects into a new stream of Student objects (with this conversion specified by the lambda expression)
- forEach simply traverses a stream and applies the lambda expression to each item

Java 8 Streams

- Instead of a Lambda expression, we could also provide a block of code, consisting of plain Java statements with a return statement to parameterize map, filter, forEach, etc, but this would not be very "functional" (but sometimes unavoidable).

- Because most stream operations are parameterized with Lambda expressions (i.e., functions), the respective methods (such as map or filter) need to be *higher-order functions*, that is, functions which take other functions (here: Lambda expressions) as arguments.

Java 8 Streams

- Chains of operations such as filter(...).map(...).sum() such as in

```
personsList
    .stream()
    .filter(allPilots)
    .mapToInt(p -> p.getAge())
    .sum();
```

...are called *pipelines.*

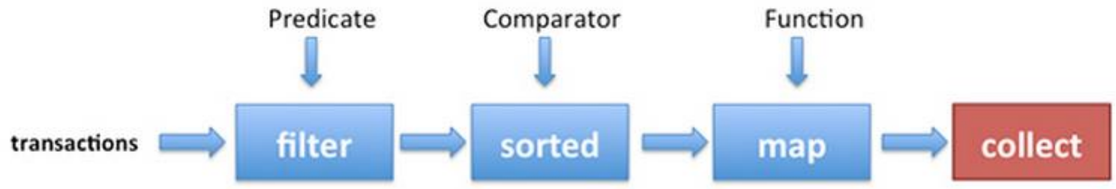
- "Bulk" stream operations such as sum(), average() or forEach() are called *aggregate operations* or *aggregates* (because they work on a bulk of data items over which they iterate) (*)
- Each intermediate aggregate operation produces a new stream from the previous one
- Aggregate operations and pipelines (but not normally using Java 8 Streams,) are also the basic programming techniques with many "Big Data" Data Analytics frameworks such as in the *MapReduce* approach in Apache Hadoop or Spark.
- Also observe the similarity to SQL operations (e.g., SELECT ... FROM ... WHERE ...)

(*) Normally, all "bulk" operations are called "aggregate operations", but sometimes only intermediate or reduction operations (the latter are terminal operations which generate a single non-stream value, e.g., sum()).

Java 8 Streams

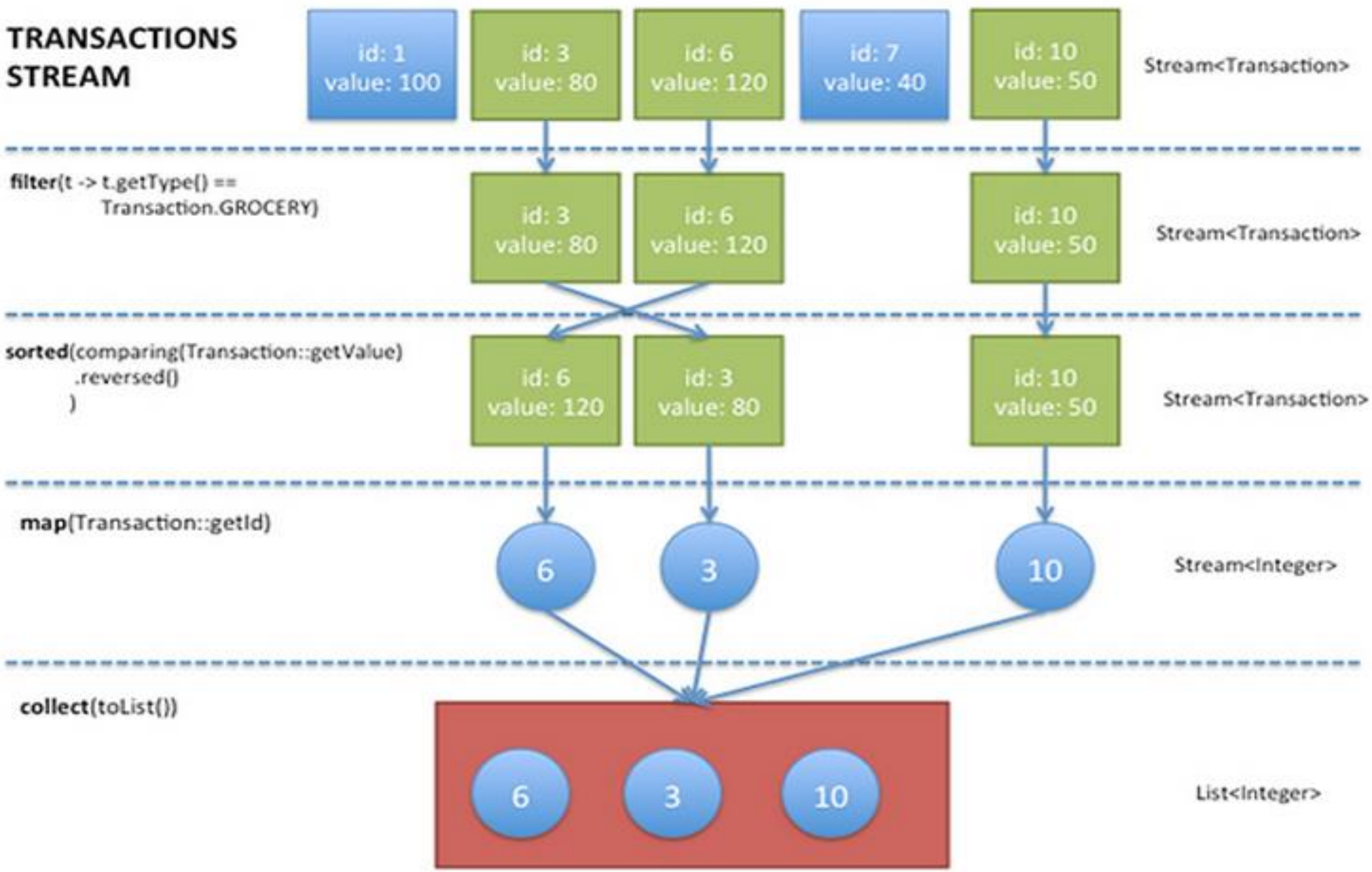
Another, more complex, example for such a pipeline, now using a stream of shopping transactions:

```
List<Integer> transactionsIds =
    transactions.stream()  // we obtain a stream from collection transactions
                .filter(t -> t.getType() == Transaction.GROCERY)
                .sorted(comparing(Transaction::getValue).reversed())
                .map(Transaction::getId)
                .collect(toList()); // collect creates a list (i.e., a collection) from the stream (terminal operation)
```



Source: Oracle technetwork

Java 8 Streams



Source: Oracle technetwork

Java 8 Streams

- https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html
- Stream operations can *automatically* make use of parallel computing (on multi-core processors), which is much easier than working manually with threads (although parallel streams also make use of threads internally, of course)

```
List<Integer> transactionsIds =
        transactions.parallelStream()
          .filter(t -> t.getType() == Transaction.GROCERY)
          .sorted(comparing(Transaction::getValue).reversed())
          .map(Transaction::getId)
          .collect(toList());
```

- Pipelines of operations can be automatically optimized (so-called *short-circuiting* - merging of operations)
- Streams can even be *infinite* (how?):

```
Stream<Integer> numbers = Stream.iterate(0, n -> n + 10);   // 0, 10, 20, 30, ...
```

Note: Race conditions can still occur with parallel streams if the streams are not used correctly! So you should still try to ensure that your data is immutable (or even better: not shared at all among operations), and that the functions applied on the data don't have any side effects, etc. Generally, avoid side-effects in Lambda expressions used with streams.

Java 8 Streams

- As mentioned before, *intermediate* stream operations (e.g., map, filter) return a new stream, whereas *terminal* operations such as forEach "close" the stream and return some ordinary data item(s) (or void)
- Because streams are *lazy*, the "materialize" their intermediate operation results only with terminal operations. E.g.,

```
Stream.of("a", "b", "c")
  .filter(x -> {
     System.out.println("filter: " + x);
     return true;
  });
```

doesn't print anything, because there is no terminal operation! But if you add

```
.forEach(x -> System.out.println("forEach: " + x));
```

at the end, it prints the items (even twice, both in filter and forEach!).

Java 8 Streams

- Similar to iterators, a closed stream cannot be "reused". You cannot iterate over the same stream again using aggregate or other stream operations
- However, we can

  1) create a new stream from an existing one (that's what intermediate operations such as "map" or "filter" do)

or

  2) terminally convert a stream into an ordinary collection (or even just a scalar value such as a single number) at the end of all previous stream operations. Ordinary collections like ArrayLists can be traversed and modified infinitely often.

Example:

```
myStream.collect(Collectors.toList());   // converts a stream into an ordinary list
```

Java 10 'var' (local variable type inference)

- A recent addition to Java is the possibility to declare local variables without explicitly mentioning their type on the left hand side of the declaration.
- The type is automatically determined by Java (*type inference*)
- Similar idea to dynamically typed languages e.g. JavaScript, Python etc.
- Keyword 'var'
- NB: these variables are still statically typed! The only difference is that with 'var' the type is automatically inferred instead of manually provided.
- What benefits and shortcomings do you see…?
- Examples:

Instead of

```
String h = "Hello " + 5;
```

we can since Java 10 write

```
var h = "Hello " + 5;
```

Instead of

```
ArrayList<Integer> myList = new ArrayList<Integer>();
```

we can write

```
var myList = new ArrayList<Integer>();
```

Java 10 'var' (local variable type inference)

- In simple cases like the examples before, the benefits are less boilerplate and higher readability.
- But there's also a danger: the inferred type might not be the intended type, so use with care. If you manually provide the type instead, you allow Java to check whether the real type is the type you had in mind --> fewer bugs.

- A real benefit of 'var' is where the type might not be obvious for beginners, e.g.:

```
var myList = List.of(67, 3.8, 4, 0);
```

(in that case I recommend letting Eclipse insert the type using "Change type of..." after clicking the mouse over the yellow "!" symbol in the left sidebar)

Remark (1): 'var' works only for local variables, not for, e.g., field declarations.

Remark (2): When you try out 'var' examples in Eclipse or IntelliJ, make sure the Java compiler and Java runtime library are set to Java 10 or higher. This might not work with older versions of Eclipse such as Oxygen.