# Streams and Lambdas

Introduction to Java 8 Streams and Lambdas

# Java as a pipeline - introducing Streams

- You have a collection and want to operate on each element
- A pipeline is a sequence of elements and consists of
  - A source (an array, a collection etc)
  - Optionally, some intermediate operations (filtering, sorting, crunching)
    - produces a new stream/pipeline
  - A terminal operation which produces a normal result again (not a stream)
    - a primitive value (int, double etc)
    - a collection
    - even void (nothing, but with a side-effect, like println)
- Similar to shell (bash) pipes (where std out is the stream typically)

# Sorting the arguments and printing them

Let's sort the arguments to a main method and then print all of them:

```
import java.util.*;

public class ArgSorter {
  public static void main(String[] args) {
    Arrays.stream(args) // create a stream (source)
      .sorted() // intermediate op, creates a new stream
      .forEach(System.out::println); // terminal operation
  }
}
// Bash:
cat file | sort | cat        # actually | cat is not needed
```

# Filtering the arguments and printing them

Let's filter the arguments to a main method and then print all of them:

```
import java.util.*;

public class ArgFilterer {
  public static void main(String[] args) {
    Arrays.stream(args) // create a stream (source)
      .filter(s -> s.startsWith("i")) // intermediate op
      .forEach(System.out::println); // terminal operation
  }
}
// Bash:
cat file | grep '^i' | cat   # actually | cat is not needed
```

# New weird syntax detected!

What was the following in the previous examples?

```
.filter(s -> s.startsWith("i"))
.forEach(System.out::println);
```

# Lambda expression

The following is a lambda expression:

```
s -> s.startsWith("i")
```

The filter() method wants a Predicate, which is a functional interface with one single abstract non-static method

```
public boolean test(T t)
```

# Lambda expression

Java accepts this, as a summary of a Predicate instance:

```
s -> s.startsWith("i")
```

It wanted an object with this method:

```
public boolean test(T t)
```

It understands that “s” is the argument, and **`s.startsWith("i")`**

can be used as the boolean value the method should return.

# Lambda syntax - arguments

A lambda is used to summarize an object with only one method.

If the method doesn't take any arguments, you write:

```
() ->
```

If the method takes one argument, you can use:

```
a ->
```

More than one argument (like two for instance):

```
(a1, a2) ->
```

# Lambda syntax - arguments and body

The body of the method is summarized to its return expression.

A method which takes one argument and returns its value times 2 can be written:

a -> a * 2

The type of the argument and return value are inferred. So, if you need a method which produces an int, the above works fine.

# Lambda syntax - multi-line method bodies

If you need more than just the expression to return, you need to use { } around your method body, semicolons and return statement:

```
SomeStringInterface ssi = a -> {
    if (a == null) {
      return "";
    } else {
      return a.toLowerCase();
    }
};
```

# We need an object which can fix a string

Let's say we have an interface defining only one method:

```
public interface CaseFixer {
  // returns the String with its first letter in upper case
  public String ucFirst(String s);
}
```

Now, we can use a lambda to represent an object which has such a method:

```
s -> Character.toUpperCase(s.charAt(0)) + s.substring(1);
```

# Example using a lambda for CaseFixer

```
public class Uc {
  public static void main(String[] args) {
    CaseFixer caseFixer =
      s -> Character.toUpperCase(s.charAt(0)) + s.substring(1);
    for (String arg : args) {
      System.out.println(fixIt(arg, caseFixer));
    }
  }
                                  // will accept lambda here
  static String fixIt(String s,  CaseFixer cf) {
    return cf.ucFirst(s);
  }
}
```

# Example using a lambda for CaseFixer

```
// We could even write this:
System.out.println(
  fixIt(arg, s -> Character.toUpperCase(s.charAt(0)) + s.substring(1))
);

/* Becuase:
s -> Character.toUpperCase(s.charAt(0)) + s.substring(1) works as a CaseFixer!
One argument (s), and an expression of type String! Just like
ucFirst(String)!
*/

                                // will accept lambda here
static String fixIt(String s,  CaseFixer cf) {
  return cf.ucFirst(s);
}
```

# New weird syntax detected!

The following is a method reference:

```
System.out::println
```

Anywhere we'd need a void method as a terminator for instance, we can give the reference to the println method of the System.out object.

Method references are very useful, for instance when creating a comparator.

# Recap - Comparator interface

- Functional interface (only one abstract method!) in java.util
- Method to override: int compare(T t1, T t2)
- Such a simple method (taking two arguments, returning an int) can be written as a method reference sometimes

# Recap - Comparator interface

Let’s say we have a Book class with a method year() returning an int

- OK, int is not perfect for representing a year, but come on, play along a little
- We’d like a Comparator<Book> which compares two books using year()
- How would you write such a Comparator?

# Book class

```
public class Book {
  private String author;
  private String title;
  private int year;

  public Book(String author, String title, int year) {
    this.author = author;
    this.title = title;
    this.year = year;
  }

  public String author() { return author; }
  public String title() {return title; }
  public int year() { return year; }
  public String toString() { return author + " " + title + " " + year; }
}
```

# BookYearComparator - first attempt

```
import java.util.Comparator;

public class BookYearComparator implements Comparator<Book> {
  public int compare(Book first, Book second) {
    return first.year() - second.year();
  }
}
```

# BookYearComparator - in use

```
public static void main(String[] args) {
  List<Book> books = someBooks();
  System.out.println(books);
  Collections.sort(books, new BookYearComparator());
  System.out.println(books);
}
```

# Using a lambda instead of BookYearComparator

```
public static void main(String[] args) {
  List<Book> books = someBooks();
  System.out.println(books);
  Collections.sort(books,  (b1, b2) -> b1.year() - b2.year());
  System.out.println(books);
}
```

# Using a method reference to year()

```
public static void main(String[] args) {
  List<Book> books = someBooks();
  System.out.println(books);
  Collections.sort(books, Comparator.comparing(Book::year));
  System.out.println(books);
}
```

# Different types of method references

| | |
|---|---|
| Reference to a static method: | ContainingClass::staticMethodName |
| Reference to an instance method of a particular object: | containingObject::instanceMethodName |
| Reference to an instance method of an arbitrary object of a particular type: | ContainingType::methodName |
| Reference to a constructor | ClassName::new |

Source: https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

# Further reading

- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html
- https://docs.oracle.com/javase/tutorial/collections/streams/index.html

# Source listings

```
import java.util.*;

public class ArgSorter {
  public static void main(String[] args) {
    Arrays.stream(args) // create a stream (source)
      .sorted() // intermediate op, creates a new stream
      .forEach(System.out::println); // terminal operator, void
  }
}
import java.util.*;

public class ArgFilterer {
  public static void main(String[] args) {
    Arrays.stream(args) // create a stream (source)
      .filter(s -> s.startsWith("i")) // intermediate op, creates a new stream
      .forEach(System.out::println); // terminal operator, void
  }
}
```

# Source listings

```
interface CaseFixer {
  public String ucFirst(String s);
}

public class Uc {
  public static void main(String[] args) {
    CaseFixer caseFixer =
      s -> Character.toUpperCase(s.charAt(0)) + s.substring(1);
    for (String arg : args) {
      System.out.println(fixIt(arg, caseFixer));
    }
  }
  static String fixIt(String s, CaseFixer cf) {
    return cf.ucFirst(s);
  }
}
```

# Source listings - Uc with multi-line-lambda

```
public class Uc {
  public static void main(String[] args) {
    CaseFixer caseFixer = s -> {
      if (s == null || s.equals("")) {
        return "";
      } else {
        return Character.toUpperCase(s.charAt(0)) + s.substring(1);
      }
    };
    for (String arg : args) {
      System.out.println(fixIt(arg, caseFixer));
    }
    // Test "" and null:
    System.out.println(fixIt("", caseFixer));
    System.out.println(fixIt(null, caseFixer));
  }
  static String fixIt(String s, CaseFixer cf) {
    return cf.ucFirst(s);
  }
}
```

# Source listings

```
public class Book {
  private String author;
  private String title;
  private int year;

  public Book(String author, String title, int year) {
    this.author = author;
    this.title = title;
    this.year = year;
  }

  public String author() { return author; }
  public String title() {return title; }
  public int year() { return year; }
  public String toString() { return author + " " + title + " " + year; }
}
```

# Source listings

```
import java.util.Comparator;

public class BookYearComparator implements Comparator<Book> {
  public int compare(Book first, Book second) {
    return first.year() - second.year();
  }
}
```

# Source listings

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Collections;
import java.util.Comparator;

public class BookTest {
  public static void main(String[] args) {
    List<Book> books = someBooks();
    System.out.println(books);
    //Collections.sort(books, new BookYearComparator());
    //Collections.sort(books, (b1, b2) -> b1.year() - b2.year());
    Collections.sort(books, Comparator.comparing(Book::year));
    System.out.println(books);
  }

// continues on next slide
```

# Source listings

```
  static List<Book> someBooks() {
    Book[] books = {
      new Book("Java for dummies", "Henrik", 2012),
      new Book("Programming in the future", "Jan", 2039),
      new Book("Java for weirdos", "Henrik", 2013),
      new Book("C for university teachers", "Henrik", 2009),
      new Book("Ada for Beda", "Rikard", 2010)
    };
    return new ArrayList<Book>(Arrays.asList(books));
  }
} // end class BookTest
```

# Download source code

https://github.com/progund/java-extra-lectures/tree/master/functional-streams/intro-lambda-streams