FUNDAMENTAL PROGRAMMING TECHNIQUES

LAMBDA EXPRESSIONS AND STREAMS

MAIN BIBLIOGRAPHIC SOURCES:

- HTTPS://DOCS.ORACLE.COM/JAVASE/TUTORIAL/JAVA/JAVAOO/LAMBDAEXPRESSIONS.HTML
- HTTPS://DOCS.ORACLE.COM/JAVASE/8/DOCS/API/JAVA/UTIL/STREAM/COLLECTORS.HTML
- HTTPS://WWW.ORACLE.COM/TECHNICAL-RESOURCES/ARTICLES/JAVA/MA14-JAVA-SE-8-STREAMS.HTML
- HTTPS://JENKOV.COM/TUTORIALS/JAVA/LAMBDA-EXPRESSIONS.HTML
- K. SHARAN, BEGINNING JAVA 8 LANGUAGE FEATURES: LAMBDA EXPRESSIONS, INNER CLASSES, THREADS, I/O, COLLECTIONS, AND STREAMS 1ST EDITION, APRESS, 2014.

Lambda Expressions

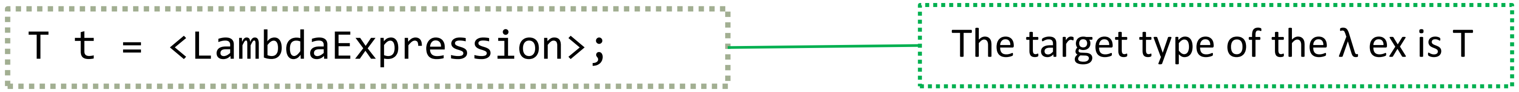
- Anonymous block of code
  - Describes an anonymous function that has no name, no return type, no throws clause and no generics
- Syntax

```
(<LambdaParametersList>) -> { <LambdaBody> }
```

  - **(<LambdaParametersList>)** - a comma-separated list of formal parameters enclosed in parentheses
  - **->** - the arrow token
  - **{ <LambdaBody> }** - consists of a single expression or a statement block
    - May declare local variables
    - May use statements including break, continue and return
    - May throw exceptions
- Classification: implicit typed and explicit typed lambda expressions

Lambda Expressions and Functional Interfaces

- Lambda expressions have different types in different contexts => are poly expressions
- Lambda expression type is Functional Interface, the exact type depends on the context in which it is used



```
T t = <LambdaExpression>;
```

The target type of the λ ex is T

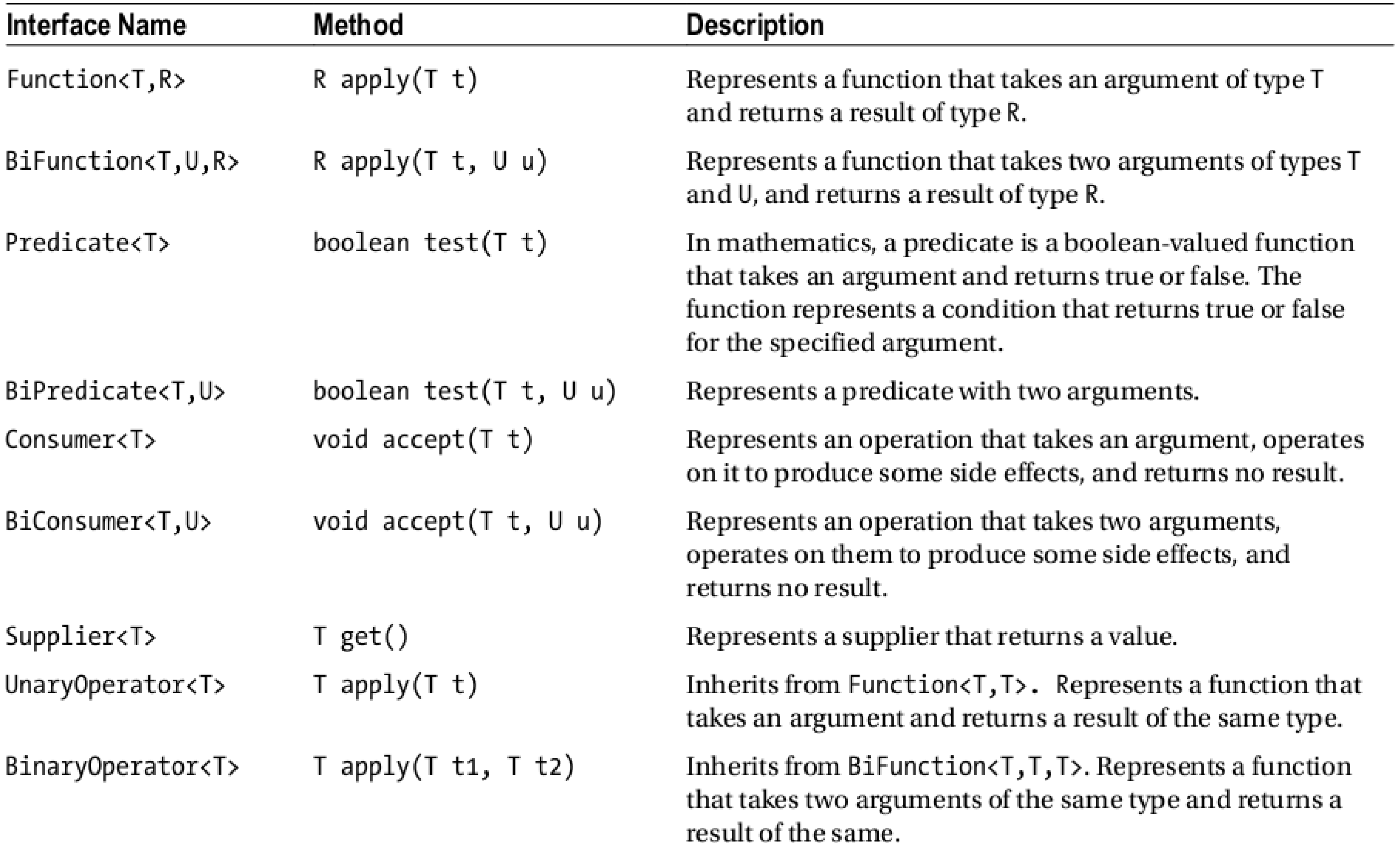
Inferring rules used by compiler (they are close related to the abstract method of the Functional Interface)

- T must be a Functional Interface type
- λ ex has the same number and type of parameters as the abstract method of T
- For an implicit λ ex, parameters types are inferred from the abstract method of T
- The type of the returned value from the body of the λ ex should be assignment compatible to the return type of the abstract method of T
- If the body of the λ ex throws any checked exceptions, they must be compatible with the declared throws clause of the abstract method of T
- It is a compile-time error to throw checked exceptions from the body of a λ ex, if its target type's method does not contain a throws clause

```
@FunctionalInterface
public interface Adder {
   double add(double n1, double n2);
}

Adder adder = (x, y) -> x + y;

double sum1 = adder.add(10.34, 89.11);
```

Lambda Expressions and Functional Interfaces

- Common functional interfaces defined in java.util.function



Interface Name	Method	Description
Function<T,R>	R apply(T t)	Represents a function that takes an argument of type T and returns a result of type R.
BiFunction<T,U,R>	R apply(T t, U u)	Represents a function that takes two arguments of types T and U, and returns a result of type R.
Predicate<T>	boolean test(T t)	In mathematics, a predicate is a boolean-valued function that takes an argument and returns true or false. The function represents a condition that returns true or false for the specified argument.
BiPredicate<T,U>	boolean test(T t, U u)	Represents a predicate with two arguments.
Consumer<T>	void accept(T t)	Represents an operation that takes an argument, operates on it to produce some side effects, and returns no result.
BiConsumer<T,U>	void accept(T t, U u)	Represents an operation that takes two arguments, operates on them to produce some side effects, and returns no result.
Supplier<T>	T get()	Represents a supplier that returns a value.
UnaryOperator<T>	T apply(T t)	Inherits from Function<T,T>. Represents a function that takes an argument and returns a result of the same type.
BinaryOperator<T>	T apply(T t1, T t2)	Inherits from BiFunction<T,T,T>. Represents a function that takes two arguments of the same type and returns a result of the same.

```
// Example using Function
Function<Long, Long> square = x -> x * x;
Function<Long, Long> addOne = x -> x + 1;

Function<Long, Long> squareAddOne =
         square.andThen(addOne);

System.out.println(squareAddOne.apply(5L));
```

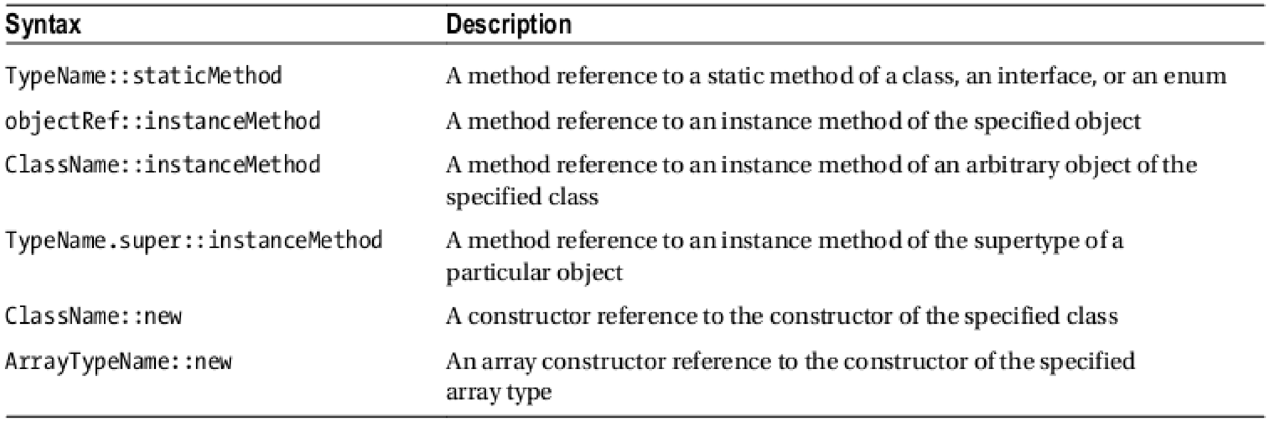
```
// Example using Predicate
Predicate<Integer> greaterThanTen = x -> x > 10;
Predicate<Integer> lessThanOrEqualToTen =
                   greaterThanTen.negate();

System.out.println(greaterThanTen.test(10));
System.out.println(lessThanOrEqualToTen.test(10));
```

Lambda Expressions and Functional Interfaces

- Method references
  - compact, easy-to-read lambda expressions for methods that already have a name
- Syntax

```
<Qualifier>::<MethodName>
```

  - **<Qualifier>** depends on the type of the method reference



Syntax	Description
`TypeName::staticMethod`	A method reference to a static method of a class, an interface, or an enum
`objectRef::instanceMethod`	A method reference to an instance method of the specified object
`ClassName::instanceMethod`	A method reference to an instance method of an arbitrary object of the specified class
`TypeName.super::instanceMethod`	A method reference to an instance method of the supertype of a particular object
`ClassName::new`	A constructor reference to the constructor of the specified class
`ArrayTypeName::new`	An array constructor reference to the constructor of the specified array type

  - **<MethodName>** is the name of the method

```
public interface MyPrinter{
    public void print(String s);
}

// Using lambda expressions
MyPrinter myPrinter = s ->
    System.out.println(s);

// Using method references
MyPrinter myPrinter = System.out::println;
```

Lambda Expressions and Functional Interfaces

- Method references – Comparing Objects
  - Methods of the Comparator interface

```
static <T,U extends Comparable<? super U>> Comparator<T> comparing (Function<? super T,? extends U> keyExtractor)
default <U extends Comparable<? Super U>>Comparator<T> thenComparing (Function<? super T,? extends U> keyExtractor)
```

  - Example - create a Comparator<Person> that sorts Person objects based on their last names and first names

```
Comparator<Person> lastFirstComp = Comparator.comparing(Person::getLastName)
                                             .thenComparing(Person::getFirstName);
```

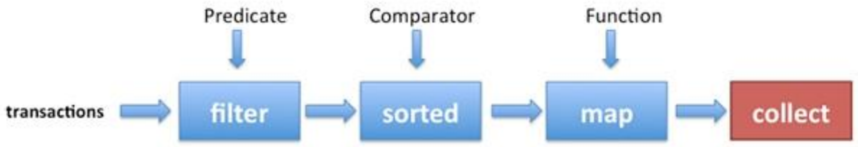
Streams

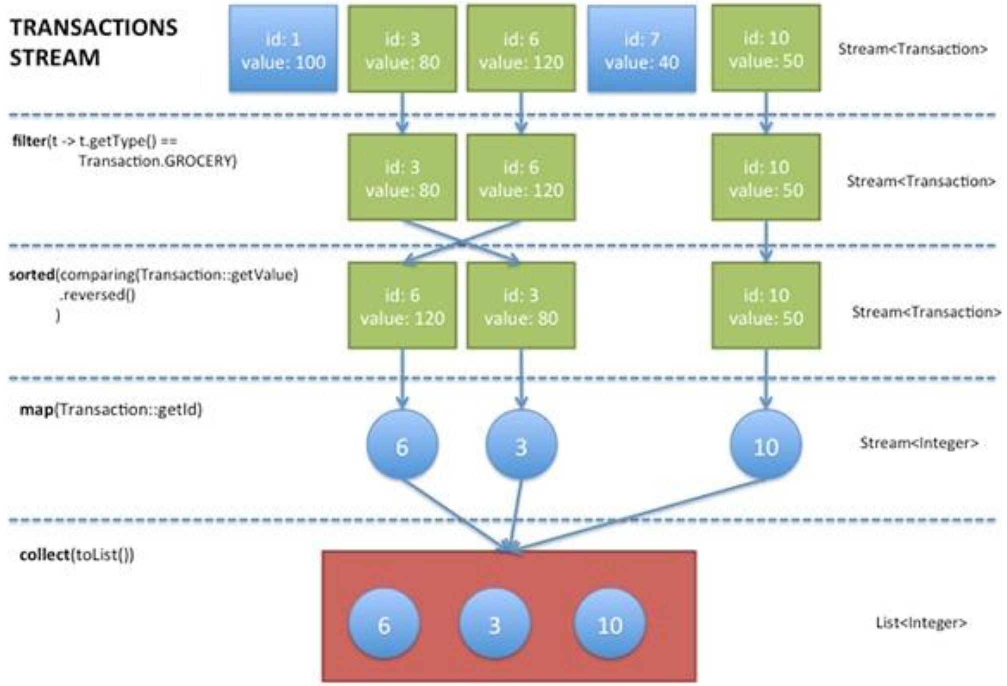
- Definition
  - a **sequence of elements** from a **source** that supports **aggregate operations**
    - **Sequence of elements**: A stream provides an interface to a sequenced set of values of a specific element type; streams do not actually store elements; they are computed on demand
    - **Source**: Streams consume from a data-providing source such as collections, arrays, or I/O resources
    - **Aggregate operations**: Streams support SQL-like operations and common operations from functional programing languages (e.g., filter, map, reduce, find, match, sorted, etc.)
  - Features
    - **Pipelining**: Many stream operations return a stream themselves => allows operations to be chained to form a larger pipeline
    - **Internal iteration**: In contrast to collections, which are iterated explicitly (external iteration), stream operations do the iteration behind the scenes for you

Streams

- Example - Find all transactions of type grocery and return a list of transaction IDs sorted in decreasing order of transaction value

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





Streams

- Example – Create a stream from files

```
public class Product {
    private String name;
    public Product(String name) { this.name = name; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

products.txt

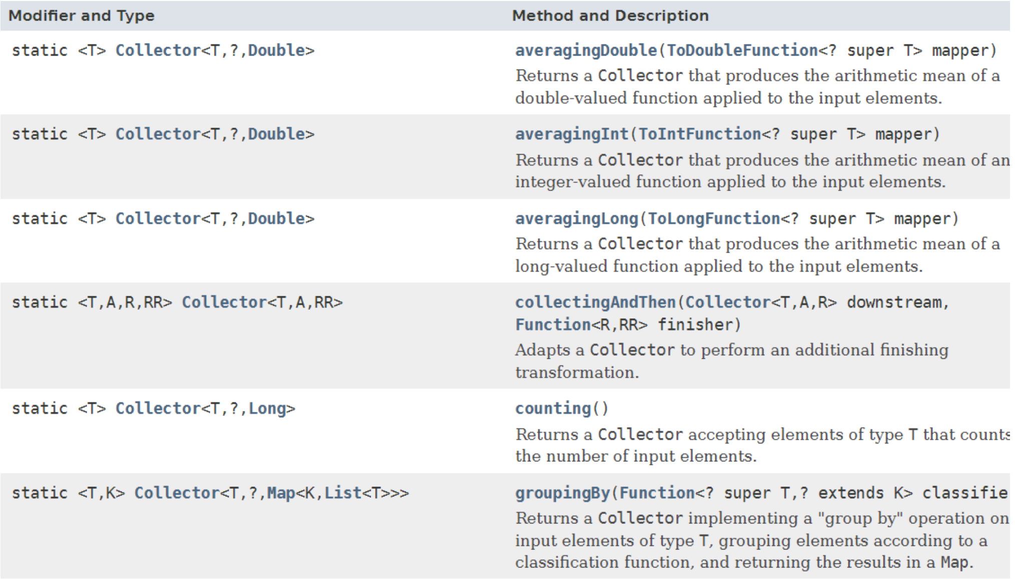
```
apple
juice
bread
…
```

```
public class StreamProcessing {
    public static void main(String[] args) throws IOException {
        Stream<String> stream = Files.lines(Paths.get("products.txt"));
        List<Product> productList = stream.map(line -> new Product(line))
                                            .collect(Collectors.toList());
        productList.stream()
                   .map(Product::getName)
                   .forEach(System.out::println);
    }
}
```

Streams

- **Collectors Class** (java.util.stream package) - implements various useful reduction operations

Fragment of the Collectors class' methods



Modifier and Type	Method and Description
static <T> Collector<T,?,Double>	averagingDouble(ToDoubleFunction<? super T> mapper) Returns a Collector that produces the arithmetic mean of a double-valued function applied to the input elements.
static <T> Collector<T,?,Double>	averagingInt(ToIntFunction<? super T> mapper) Returns a Collector that produces the arithmetic mean of an integer-valued function applied to the input elements.
static <T> Collector<T,?,Double>	averagingLong(ToLongFunction<? super T> mapper) Returns a Collector that produces the arithmetic mean of a long-valued function applied to the input elements.
static <T,A,R,RR> Collector<T,A,RR>	collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher) Adapts a Collector to perform an additional finishing transformation.
static <T> Collector<T,?,Long>	counting() Returns a Collector accepting elements of type T that counts the number of input elements.
static <T,K> Collector<T,?,Map<K,List<T>>>	groupingBy(Function<? super T,? extends K> classifie Returns a Collector implementing a "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a Map.

```
// Accumulate names into a List
List<String> list = people.stream().map(Person::getName)
                             .collect(Collectors.toList());
```

```
// Convert elements to strings and concatenate them,separated
// by commas
String joined = things.stream().map(Object::toString)
                           .collect(Collectors.joining(", "));
```

```
// Compute sum of salaries of employee
int total = employees.stream()
    .collect(Collectors.summingInt(Employee::getSalary)));
```

```
// Group employees by department
Map<Department, List<Employee>> byDept = employees.stream()
    .collect(Collectors.groupingBy(Employee::getDepartment));
```

```
// Compute sum of salaries by department
Map<Department, Integer> totalByDept = employees.stream()
    .collect(Collectors.groupingBy(Employee::getDepartment,
    Collectors.summingInt(Employee::getSalary)));
```