Advanced Java

Lambda Expressions

Java local-variable type inference

- Added in Java 10
- No need to specify type of local variables in some cases
- Type of variable is specified using *var* keyword
- Compiler automatically infers the type of local variables
- Restricted to
  - Local variables, including
    - index variables of for-each loops
    - resource variables of the try-with-resources statement

Examples

- var list = new ArrayList<Integer>(); // infers ArrayList<Integer>
- for(var v : list) { …}   // infers Integer
- var myNum = new Integer(123);   // infers Integer
- var myClassObj = new MyClass(); // infers MyClass

Additional Restriction

- Is this possible?

```
var v;

v = new Object();
```

- Java is still a statically typed language
- When using *var*
  - There should be enough information to infer the type of local variable.
  - If not, the compiler will throw an error.
  - The type of the variable is inferred from the type of the initializer.

- What happens with “var v = null;”

```
error: cannot infer type for local variable v
 var v = null;
 ^ (variable initializer is 'null')
```

instanceof with pattern matching

- Avoid *instanceof*
- Very useful in particular cases:

```
public boolean equals(Object obj) {
    if (obj instanceof SomeClass) {
        SomeClass sc = (SomeClass)obj;
        return this.att1.equals(sc.att1) && …// checking other fields
    }
    return false;
}
```

- There is some repetition in this type of code:
  - Extra variable, cast and assignment

instanceof with pattern matching - 2

- *instanceof* operator is extended to take a type pattern instead of just a type
- Type pattern consists of a predicate that specifies a type, along with a single pattern variable

```
public boolean equals(Object obj) {
    if (obj instanceof SomeClass sc) {
        return this.att1.equals(sc.att1) && …// checking other fields
    }
    return false;
}
```

instanceof with pattern matching - 3

- A pattern variable is in scope where it has definitely matched

```
if (obj instanceof String s && s.length() > 5) {
    flag = s.contains("jdk");
}
```

```
if (obj instanceof String s || s.length() > 5) {    // Error!
```

- Reduce number of casts
- Can simplify code

```
public boolean equals(Object o) {
    if (!(o instanceof Point))
        return false;
    Point other = (Point) o;
    return x == other.x
        && y == other.y;
}
```

```
public boolean equals(Object o) {
    return (o instanceof Point other)
        && x == other.x
        && y == other.y;
}
```

Lambda Expressions - Introduction

- Prior to Java SE 8, Java supported three programming paradigms:
  - Procedural programming
  - Object-oriented programming
  - Generic programming
- Java SE 8 adds Functional programming.

Introduction - 2

- Without functional programming
  - First, you typically determines what you want to accomplish
  - Then, specify the precise steps to accomplish that task.
  - Usually applies external iteration
    - Using a loop to iterate over a collection of elements.
    - Requires accessing the elements sequentially
- With functional programming
  - Specify what you want to accomplish in a task (a.k.a. as function), but not how to accomplish it
  - Internal iteration
    - Let the data structure determine how to iterate over a collection of elements
    - Internal iteration is easier to parallelize

Motivation Example

- Consider the following entity
- And users of the application are stored in a List<Person> attribute
- Want to be able to filter the members and only print those that satisfy a given criteria

```
public enum Sex {
    MALE, FEMALE
}

public class Person {
   private String name;
   private LocalDate birthday;
   private Sex gender;
   private String emailAddress;

   public int getAge() {
      // ...
   }

   public void print() {
      // ...
   }
}
```

Filter Members - 1

- Create a filtering method for criteria

```
public static void printPersonsOlderThan(List<Person> members, int age) {
    for (Person p : members) {
        if (p.getAge() >= age) {
            p.print();
        }
    }
}
```

- Additional criteria -> implement another *printPersonWithCriteria* method
- How to improve?
  - Separate the iteration code on Person from the filtering criteria code

Specify Filtering Criteria Code

Define abstraction

```
public interface CheckPerson {
  boolean test(Person p);
}
```

Define generic print

```
public static void printPersons(List<Person> members,
                                CheckPerson tester) {
   for (Person p : members)
      if (tester.test(p))
         p.print();
}
```

Advantages:

- Refactor repeated code
- Search criteria can be reused

Specify Filtering Criteria Code with a Class

Define abstraction

```
public interface CheckPerson {
  boolean test(Person p);
}
```

Consider only young male adults:

```
class CheckMaleAdults implements CheckPerson {
   public boolean test(Person p) {
      return p.gender == Sex.MALE &&
         p.getAge() >= 18 && p.getAge() <= 25;
   }
}
```

Define generic print

```
public static void printPersons(List<Person> members,
                                CheckPerson tester) {
   for (Person p : members)
      if (tester.test(p))
         p.print();
}
```

```
printPersons(members, new CheckMaleAdults());
```

Disadvantages:

- Additional interface
- Define class for each criteria

Specify Filtering Criteria Code with an Anonymous Class

Define abstraction

```
public interface CheckPerson {
  boolean test(Person p);
}
```

```
printPersons(members,
   new CheckPerson() {
      public boolean test(Person p) {
         return p.getGender() == Sex.MALE
            && p.getAge() >= 18 && p.getAge() <= 25;
      }
   });
```

Define generic print

```
public static void printPersons (List<Person> members,
                                 CheckPerson tester) {
   for (Person p : members)
      if (tester.test(p))
         p.print();
}
```

- Reduces amount of code
- But syntax of anonymous class

Syntax of Java 8 Lambdas

- Java 8 SE supports functions as first-class citizens
  - Lambda expression
- A lambda is basically a method in Java without a declaration usually written as
  (parameterList) -> body
- A lambda can have zero or more parameters separated by commas and their type can be explicitly declared or inferred from the context
  - (int x, int y) -> { return x + y; }
  - (x, y) -> { return x + y; }
- ( ) is used to denote zero parameters
  - () -> { System.out.println(“Hello World!”); }
- Parenthesis are not needed around a single parameter
  - x -> { return x * x; }

Syntax of Java 8 Lambdas – 2

- The body consists of a single expression or a statement block
- If you specify a single expression, then the Java runtime evaluates the expression and then returns its value (if needed)
- Braces are not needed around a single-statement body
  - x -> System.out.println(x);
  - x -> x + x;
- However, return statement is not an expression
  - In a lambda expression, you must enclose a return statement always in braces ({})
  - x -> { return x + x; }
- You can consider lambda expressions as anonymous methods

Implementation of Java 8 Lambdas

- The Java 8 compiler first converts a lambda expression into a function
- It then calls the generated function
- For example, `x -> System.out.println(x)` could be converted into a generated static function

```
public static void genName(Integer x) {
    System.out.println(x);
}
```

- But what type should be generated for this function?
- How should it be called?
- What class should it go in?

Solution: Functional Interfaces

- Design decision: Java 8 lambdas are assigned to functional interfaces
- A functional interface is a Java interface with ***exactly one*** **abstract method** that does not override a method in java.lang.Object

```
@FunctionalInterface
public interface Consumer<T> {
   void accept(T t);
}
```

- The package `java.util.function` defines many new useful functional interfaces.

Four Categories of Functional Interfaces

Supplier

```
interface Supplier<T> {
  T get();
}
```

Predicate

```
interface Predicate<T> {
  boolean test(T t);
}
```

```
interface BiPredicate<T, U> {
  boolean test(T t, U u);
}
```

Consumer

```
interface Consumer<T> {
  void accept(T t);
}
```

```
interface BiConsumer<T, U> {
  void accept(T t, U u);
}
```

Function

```
interface Function<T,R> {
  R apply(T t);
}
```

```
interface BiFunction<T, U, R> {
  R apply(T t, U u);
}
```

Properties of the Generated Method

```
public interface Consumer<T> {
  void accept(T t);
}
```

```
x -> System.out.println(x);
```

- The method generated from a Java 8 lambda expression has the same signature as the method in the functional interface
  - void accept(T t)
- The type is the same as that of the functional interface to which the lambda expression is assigned
  - Consumer<T>
- The lambda expression becomes the body of the method in the interface

```
Consumer<String> cons;
cons = x -> System.out.println(x);
cons.accept("aaa");
```

Assigning a Lambda to a Local Variable

How to print all elements in a List with Lambdas?

```
public class ArrayList<T> … {
  …
  void forEach(Consumer<T> action {
    for (T i:items) {
      action.accept(t);
  }
}
```

```
public interface Consumer<T> {
  void accept(T t);
}
```

Solution

```
class Main {
  public static void main(String[] args) {
    List<Integer> intSeq = Arrays.asList(1,2,3);

    Consumer<Integer> cnsmr = x -> System.out.println(x);
    intSeq.forEach(cnsmr):
  }
}
```

Assigning a Lambda to a method parameter

How to print all elements in a List with Lambdas?

```
public class ArrayList<T> … {
  …
  void forEach(Consumer<T> action {
    for (T i:items) {
      action.accept(t);
  }
}
```

```
public interface Consumer<T> {
  void accept(T t);
}
```

Solution

```
public class Main {
  public static void main(String[] args) {
    List<Integer> intSeq = Arrays.asList(1,2,3);

    intSeq.forEach(x -> System.out.println(x));
  }
}
```

Specify Filtering Criteria Code with a Lambda Expression

Define abstraction

```
public interface CheckPerson {
  boolean test(Person p);
}
```

```
printPersons(members,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Sex.MALE
                && p.getAge() >= 18 && p.getAge() <= 25;
        }
    });
```

Define generic print

```
public static void printPersons (
    List<Person> members, CheckPerson tester) {
    for (Person p : members)
        if (tester.test(p))
            p.print();
}
```

```
printPersons(members,
            p -> p.getGender() == Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25);
```

Specify filtering criteria code with a lambda expression-2

Reuse abstraction

```
public interface Predicate<T> {
  boolean test(T t);
}
```

```
printPersons(members,
        p -> p.getGender() == Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25);
```

Define generic print

```
public static void printPersons (
    List<Person> members, Predicate<Person> tester) {
    for (Person p : members)
        if (tester.test(p))
            p.print();
}
```

Specify filtering criteria code with a lambda expression-3

Reuse another abstraction

```
public interface Consumer<T> {
  void consumer(T t);
}
```

```
printPersons(members,
   p -> {if (p.getGender() == Sex.MALE && p.getAge() >= 18  && p.getAge() <= 25)
            p.print(); } );
```

Define a **more** generic print

```
public static void printPersons (
   List<Person> members, Consumer<Person> c) {
   members.forEach(c);
}
```

Accessing Local Variables of the Enclosing Scope

- Like anonymous classes, lambda expressions can access to:
  - to local variables (final or effectively final) of the enclosing scope
  - and fields and methods of the enclosing scope (static and non-static)
- Lambda expressions are lexically scoped:
  - Do not inherit any names from a supertype
  - Do not introduce a new level of scoping
  - Cannot define attributes
  - Use of **this** inside a lambda expression refers to the enclosing object and not to the lambda object
  - Declarations in a lambda expression are interpreted just as they are in the enclosing environment

Accessing variables - Example

```
public class A {
  private int x = 5;

  public void doSomething(int y) {
    B b = () -> System.out.println("this.toString() = " + this.toString() +
                                   "\ntoString = " +  toString() +
                                   "\nx = " + x + " y = " + y);
    System.out.println("A.toString() = " + this.toString());
    System.out.println("Lambda.toString()" + b.toString());
    x = 10;
    b.cc();
     x = 20;
     b.cc();
  }

  public static void main(String[] args) {
    new A().doSomething(3);
  }
}
```

```
interface B {
  void cc();
}
```

Result:

```
A.toString() = A@36baf30c
Lambda.toString()A$$Lambda$1/746292446@7a81197d
this.toString() = A@36baf30c
toString = A@36baf30c
x = 10 y = 3
this.toString() = A@36baf30c
toString = A@36baf30c
x = 20 y = 3
```

Lambdas as Objects

- A Java lambda expression is essentially an object

```
public class Person {
  private int _age;

  Person(int a) {
    _age = a;
  }


  public final int getAge() {
    return _age;
  }
}
```

```
import java.util.Comparator;
public class A {
  private int x = 5;

  public static void main(String[] args) {
    Comparator<Person> compareByAge =
        (p1, p2) -> { return p1.getAge() - p2.getAge(); };

    Person p = new Person(2);
    Person pp = new Person(5);

    int result = compareByAge.compare(p, pp);
  }
}
```

Method References as Lambdas

- A concise way to write lambda expression when:
  - Just call another method
  - With parameters given to the lambda

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

MyPrinter printer = s -> System.out.println(s);
```

```
MyPrinter printer = System.out::println;
```

  - Double colons :: signal to the Java compiler that this is a method reference
    - Format: Class or instance :: method
- Four kinds of method references

Method Reference Types

- *objectName*::*instanceMethodName* (**Instance Method Reference)**
  - Creates a lambda that:
    - invokes *instanceMethodName* on *objectName*
    - passes the lambda's arguments to the instance method
    - and returns the method's result
    - The argument types of *instanceMehodName* and lambda method must match
- *ClassName*::*staticMethodName (***Static Method Reference)**
  - Creates a lambda that
    - invokes *staticMethodName* on *ClassName*
    - passes the lambda's arguments to the static method
    - and returns the method's result
    - The argument types of *staticMethodName* and lambda method must match

Method Reference Types - 2

- *ClassName*::*instanceMethodName* (Parameter Method Reference)
  - Creates a lambda that
    - invokes the *instanceMethodName* on the first lambda's argument
    - passes the remaining parameters to the instance method
    - and returns the method's result
- *ClassName*::new (**Constructor Reference)**
  - Creates a lambda that
  - invokes one of the constructors of *ClassName*
  - passes the lambda's parameters to the constructor
  - The argument types of one of the constructors of *ClassName* and lambda method must match

```
public interface Factory {
  String create(char[] val);
}

Factory factory = String::new;
Factory factory = chars -> new String(chars);
```

Streams

- A stream is a limitless iterator that allows you to process data
- Streams are objects that implement interface `java.util.stream.Stream`
  - Enable you to perform functional programming tasks
  - Specialized stream interfaces for processing `int`, `long` or `double` values
- Streams move elements through a sequence of processing steps—known as a stream pipeline
- A pipeline contains the following components:
  - A data source: collection, array, generator function or I/O channel
  - Zero or more intermediate operations
    - Each intermediate operation consumes a stream and produces a new stream
  - And ends with a terminal operation
    - Produces a non-stream result: primitive value, collection, array or no value.
- A stream pipeline is formed by chaining method calls

Streams (Cont.)

- Streams do not have their own storage
  - Once a stream is processed, it cannot be reused, because it does not maintain a copy of the original data source.
- An intermediate operation specifies tasks to perform on the stream's elements and always results in a new stream.
  - Intermediate operations are lazy—they aren't performed until a terminal operation is invoked
  - returns a Stream<T>
  - Allows library developers to optimize stream-processing performance
- A terminal operation initiates processing of a stream pipeline's intermediate operations
  - Produces a result
  - Terminal operations are eager—they perform the requested operation when they are called
- May be parallelized and optimized across cores

Stream as a Sequence of Elements

- **Sequence of elements**— Like a collection, a stream provides an interface to a sequenced set of values of a specific element type
- Collections are data structures
  - Store and access elements

- Streams concern more with expressing computations as map, filter, sort, …
- Stream operations can be executed either **sequentially** or in **parallel**
- **Internal iteration**— In contrast to collections, which are iterated explicitly using an iterator,
- stream operations do the iteration behind the scenes for you.

- Note that generating a stream from an ordered collection preserves the ordering. The elements of a stream coming from a list will have the same order as the list.

Common Intermediate Operations

- Stream<T> **filter**(Predicate<T>)
  - Produces a new Stream that contains only the elements of the original Stream that pass a given test
- Stream<T> **distinct**()
  - Returns a stream consisting of the distinct elements of this stream
- Stream<T> **limit**(long maxSize)
  - Limit(n) returns a new stream of the first n elements of the original stream
- Stream<R> **map**(Function<T, R>)
  - Produces a new Stream that is the result of applying a Function to each element of original Stream
  - mapToInt, mapToDouble, mapToLong
- Stream<T> **sorted**(Comparator<T>), Stream<T> **sorted**()
  - Returns a new stream consisting of the elements of original stream, sorted according to the provided Comparator (or natural order)
  - No argument uses the natural order for the stream's element type
- Stream<T> **skip**(long n)
  - Returns a new stream starting with element n of original stream

```
interface Predicate<T> {
  Boolean test(T t);
}
```

```
interface Function<T,R> {
  R apply(T t);
}
```

Common Terminal Operations - Iteration

- void **forEach**(Consumer<T>) – consumes each element from the input stream and applies a function to each one
- Stream<T> **peek**(Consumer<T>) – similar to forEach but returns identical ro original one after applying provided action to each element

Common Terminal Operations - Search

- boolean **anyMatch**(Predicate<T>) – checks if any element in the input stream verifies the specified predicate
- boolean **noneMatch**(Predicate<T>) – checks if none element in the input stream verifies the specified predicate
- boolean **allMatch**(Predicate<T>) – checks if all elements in the input stream verify the specified predicate
- Optional<T> **findFirst**() – returns the first element of the input stream
- Optional<T> **findAny**() – returns an element of the input stream

Common Terminal Operations - Reduction

- Reduction operations
  - Take all values in a stream and return a single value
  - long **count**() – returns the number of elements in the input stream
  - Optional<T> **max**(Comparator<? super T>) – returns the maximum element of this stream according to the provided Comparator
  - Optional<T> **min**(Comparator<? super T>) – returns the minimum element of this stream according to the provided Comparator
  - OptionalDouble average() – returns the average value of this numeric stream
- Mutable Reduction operations
  - Create a container to store all elements in the stream
  - R **collect**(Collector<T, A, R>) – reduces the input stream into a collection
  - Collectors.toList(), Collectors.toSet(), …
  - Object<T> **toArray**(), A[] **toArray**(IntFunction<A[]>) – Returns an array containing the elements of this stream

Optional<T> Class

- A container which may or may not contain a non-null value
- Common methods
  - boolean **isPresent**() – returns true if value is present
  - T **get**() – returns value if present, otherwise throws NoSuchElementException
  - T **orElse**(T other) – returns value if present, or other
  - void **ifPresent**(Consumer<T>) – runs the lambda if value is present

Common Stream API Methods Used

- T reduce(T identity, BinaryOperator)
  - You start with a seed (identity) value, then combine this value with the first Entry in the Stream, combine the second entry of the Stream, etc.
  - Example

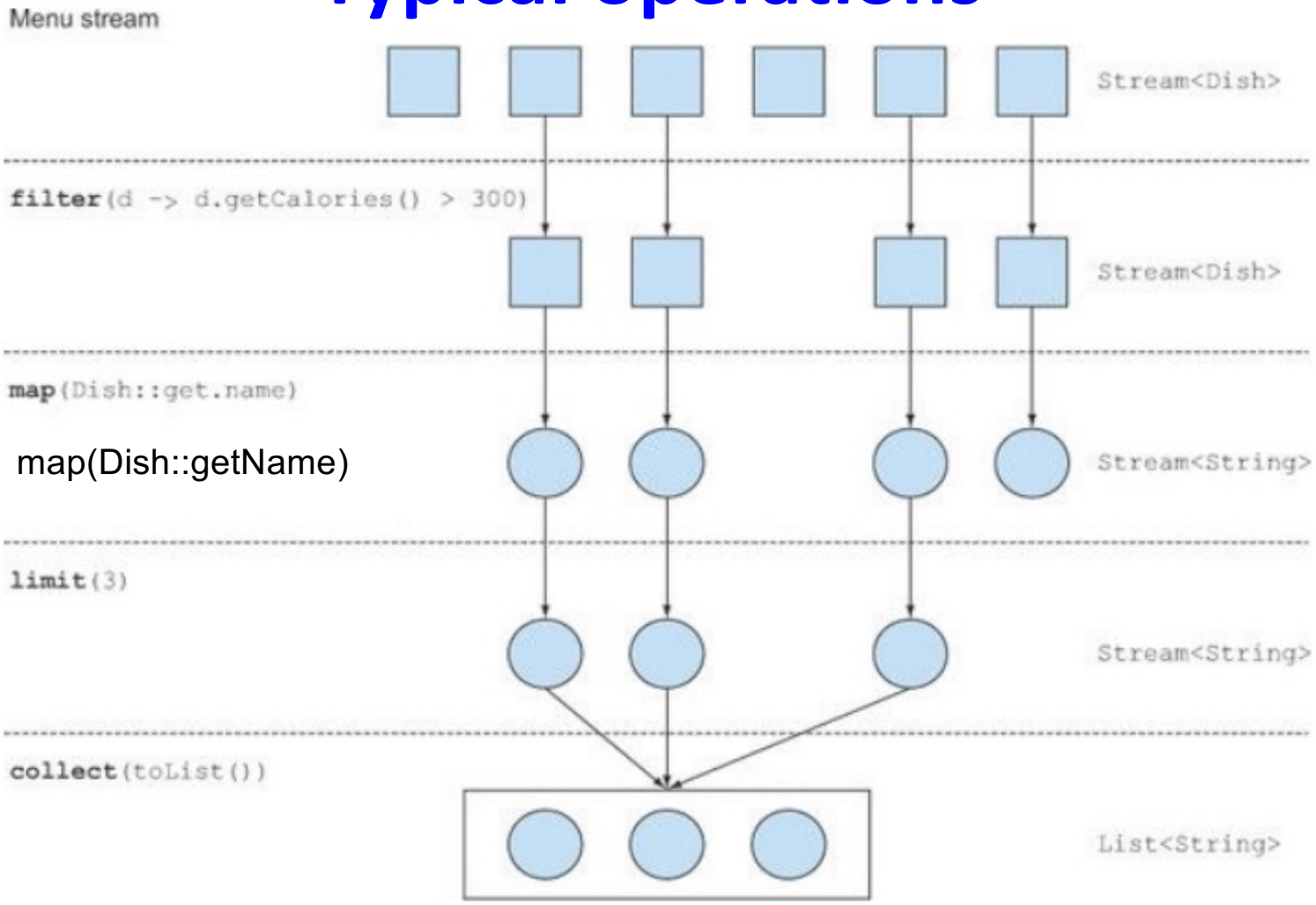
    Nums.stream().reduce(1, (n1,n2) -> n1*n2)

    Calculate the product of numbers
- Optional<T> reduce(BinaryOperator)
  - String reduced2 = items.stream()
    .reduce((acc, item) -> acc + " " + item)
    .get();
- IntStream (Stream on primitive int)
  - Built-in min, max and sum methods

Short-circuit Operations

- A Stream pipeline contains some short-circuit methods (which could be intermediate or terminal methods) that cause the earlier intermediate methods to be processed only until the short-circuit method can be evaluated
- Non-terminal short-circuit operations
  - Limit
- Terminal short-circuit operations
  - findFirst
  - findAny
  - anyMatch
  - allMatch
  - noneMatch

Typical operations



Example

```
public class Employee {
  private String _firstName;
  private String _lastName;
  private double _salary;
  private String _department;

  public Employee(String fn, String ln, double sal, String dep) { … }

  public getFirstName() { return _firstName; }
  public getLastName() { return _lastName; }

  public void setSalary(double newSalary} { _salary = newSalary; }
  public double getSalary() { return _salary; }

  public void setDepartment(String d { _department = d; }
  public String getDepartment() { return _department; }
  @Override
  public String toString() {
    return String.format("%s %s %s %f", _firstName, _lastName,
                                        _department, _salary);
  }
}
```

Example - 2

- Print all employees

```
List<Employee> employees;
…
employees.forEach(System.out::println);
```

- Filter employees by salary

```
Predicate<Employee> lowSalaries = e -> e.getSalary() < 1000;
employees.stream()
        .filter(lowSalaries)
        .sorted(Comparator.comparing(Employee::getSalary)
        .forEAch(System.out::println);
```

- Get first employee with a low salary

```
Employee e;
e = employees.stream()
        .filter(lowSalaries)
        .findFirst()
        .get();
```

Example - 3

- Create comparators

```
Comparator<Employee> byFirstName =
                Comparator.comparing(Employee::getFirstName);
Comparator<Employee> byLastName =
                Comparator.comparing(Employee::getLastName);
```

- Filter employees by salary ordered by first name

```
Predicate<Employee> lowSalaries = e -> e.getSalary() < 1000;
employees.stream()
                .filter(lowSalaries)
                .sorted(byFirstName)
                .forEach(System.out::println);
```

- Filter employees by salary ordered by first name and then by last name

```
employees.stream()
                .filter(lowSalaries)
                .sorted(byFirstName.thenComparing(byLastName))
                .forEAch(System.out::println);
```

Example - 4

- Print last names of employees (ordered)

```
employees.stream()
                .map(Employee::getLastName)
                .distinct()
                .ordered()
                .forEach(System.out::println);
```

- Print each department and corresponding employees

```
Map<Department, List<Employee>> groupedByDep ;
groupedByDep = employees.stream()
                        .collect(Collectors.groupingBy(Employee::getDepartment));

groupedByDep.forEach((key, value) -> {
                                        System.out.println("Department: " + key);
                                         value.forEach(System.out::println);
                                       });
```

Example - 5

- Compute average salary of all employees

```
employees.stream()
                .mapToDouble(Employee::getSalary)
                .average()
                 .getAsDouble();
```

- Compute the sum of salaries of all employees

```
employees.stream()
                .mapToDouble(Employee::getSalary)
                .sum();
```

```
employees.stream()
                .mapToDouble(Employee::getSalary)
                .reduce(0, (x1, x2) -> x1 + x2);
```

- Group Articles by author

```
public Map<String, List<Article>> groupByAuthor() {

   Map<String, List<Article>> result = new HashMap<>();

   for (Article article : articles) {
      if (result.containsKey(article.getAuthor())) {
         result.get(article.getAuthor()).add(article);
      } else {
         ArrayList<Article> articles = new ArrayList<>();
         articles.add(article);
         result.put(article.getAuthor(), articles);
      }
   }

   return result;
}
```

```
public Map<String, List<Article>> groupByAuthor() {
   return articles.stream()
      .collect(Collectors.groupingBy(Article::getAuthor));
}
```