Plan

2023-10-24

1. Predicate
2. Arrays methods

Theory

▶ **English**

▼ **На русском**

Предикаты в Java

Что это такое?
Предикаты — это функциональные интерфейсы, которые используются для проверки условий. Интерфейс `Predicate<T>` имеет метод `test`, который принимает объект типа `T` и возвращает `boolean`.

Пример с лямбдой:

```
Predicate<Integer> isEven = n -> n % 2 == 0; // лямбда
System.out.println(isEven.test(4)); // true
System.out.println(isEven.test(3)); // false
```

Пример с анонимным классом:

```
public class Main {
    public static void main(String[] args) {
        Predicate<Integer> isEven = new Predicate<Integer>() { // анонимный
            @Override
            public boolean test(Integer n) {
                return n % 2 == 0;
            }
        };

        System.out.println(isEven.test(4)); // true
        System.out.println(isEven.test(3)); // false
```

```
    }
}
```

Пример с реализацией интерфейса:

```
import java.util.function.Predicate;


public class IsevenPredicate implements Predicate<Integer> { // реализацией
    @Override
    public boolean test(Integer n) {
        return n % 2 == 0;
    }
}


public class Main {

    public static void main(String[] args) {
        Predicate<Integer> isEven = new IsevenPredicate();
        System.out.println(isEven.test(4)); // true
        System.out.println(isEven.test(3)); // false
    }
}
```

класс Arrays

В Java класс `Arrays` из пакета `java.util` предоставляет набор статических методов для работы с массивами. Вот некоторые из них:

1. **Сортировка**: `Arrays.sort(array)` сортирует массив в порядке возрастания.
    - Arrays.sort(int[] a): Сортирует целочисленный массив в порядке возрастания.
    - Arrays.sort(int[] a, int fromIndex, int toIndex): Сортирует часть массива от fromIndex до toIndex-1.
    - Arrays.sort(Object[] a): Сортирует объекты, реализующие интерфейс **Comparable**
        - public static void sort(Object[] a, int fromIndex, int toIndex): Эта перегрузка сортирует часть массива объектов, реализующих интерфейс Comparable, от индекса fromIndex до toIndex-1. Объекты сравниваются на основе их естественного порядка.
        - public static void sort(T[] a, int fromIndex, int toIndex, Comparator c): Эта версия позволяет сортировать часть массива с использованием специального

компаратора. Comparator с определяет, как будут сравниваться объекты.
    - public static void sort(T[] a, Comparator c): Эта версия сортирует весь массив объектов с использованием заданного компаратора. Это удобно, когда естественный порядок сортировки объектов вам не подходит.
2. **Поиск**: `Arrays.binarySearch(array, value)` выполняет бинарный поиск значения в отсортированном массиве.
3. **Копирование**: `Arrays.copyOf(array, newLength)` создаёт копию массива с новой длиной.
4. **Заполнение**: `Arrays.fill(array, value)` заполняет все элементы массива заданным значением.
5. **Сравнение**: `Arrays.equals(array1, array2)` проверяет, равны ли два массива.
6. **Преобразование в строку**: `Arrays.toString(array)` возвращает строковое представление массива.

Методы `Arrays.copyOf` и `System.arraycopy` оба предназначены для копирования массивов, но есть несколько ключевых различий:

Arrays.copyOf:

1. **Создание нового массива**: `Arrays.copyOf` возвращает новый массив, который может иметь другую длину.
2. **Тип возвращаемого массива**: Может быть изменён, если используется перегрузка с параметром типа.
3. **Простота использования**: Очень прост в использовании, так как вам нужно указать только исходный массив и новую длину.

```
int[]original={1,2,3};
int[]copied=Arrays.copyOf(original,5); // [1, 2, 3, 0, 0]
```

System.arraycopy:

1. **Использует существующий массив**: Этот метод не создаёт новый массив, а копирует данные в уже существующий массив.
2. **Больше параметров**: Требует указания исходного и целевого массивов, позиций в этих массивах и количества копируемых элементов.
3. **Быстродействие**: Обычно быстрее, так как работает напрямую с памятью.

```
int[]original={1,2,3};
int[]destination=new int[5];
System.arraycopy(original,0,destination,0,original.length); // dest
```

В общем, `Arrays.copyOf` удобнее и проще в использовании для создания новых массивов, тогда как `System.arraycopy` обычно используется для копирования данных в уже существующие массивы и может быть быстрее в некоторых сценариях.

Перегрузки методов Arrays.copyOf и System.arraycopy, детально

Методы `Arrays.copyOf` и `System.arraycopy` оба предназначены для копирования массивов, но есть несколько ключевых различий:

Arrays.copyOf:

1. **Создание нового массива**: `Arrays.copyOf` возвращает новый массив, который может иметь другую длину.
2. **Тип возвращаемого массива**: Может быть изменён, если используется перегрузка с параметром типа.
3. **Простота использования**: Очень прост в использовании, так как вам нужно указать только исходный массив и новую длину.

```
int[]original={1,2,3};
int[]copied=Arrays.copyOf(original,5); // [1, 2, 3, 0, 0]
```

System.arraycopy:

1. **Использует существующий массив**: Этот метод не создаёт новый массив, а копирует данные в уже существующий массив.
2. **Больше параметров**: Требует указания исходного и целевого массивов, позиций в этих массивах и количества копируемых элементов.
3. **Быстродействие**: Обычно быстрее, так как работает напрямую с памятью.

```
int[]original={1,2,3};
int[]destination=new int[5];
System.arraycopy(original,0,destination,0,original.length); // dest
```

В общем, `Arrays.copyOf` удобнее и проще в использовании для создания новых массивов, тогда как `System.arraycopy` обычно используется для копирования данных в уже существующие массивы и может быть быстрее в некоторых сценариях.

Практика:

1 спринт

Написать метод сортировки рабочих используя Comparator и Comparable.

- написать Comparator для полей
  - id
    - Для поля id используйте интерфейс Comparable
  - hireYear
    - Используйте анонимный класс для поля hireYear
  - name
    - Используйте лямбду для поле name
- написать метод фильтрации массива рабочих по условиям:
  - найти всех рабочих чья зарплата в диапазоне от и до.
  - найти всех рабочих кто был устроен в определенный период времени (fromHireYear, toHireYear).
  - и так далее для всех полей.

2 спринт

Применить полученные знания по Predicate

- внесите изменения в код, где это необходимо
- удалите не нужный код

Homework

▶ **English**

▼ **На русском**

Задание 1: Фильтрация массива чисел

Задача

1. Создайте массив целых чисел (например, `[1, 2, 3, 4, 5, 6]`).
2. Определите предикат `Predicate<Integer>`, который будет проверять, является ли число четным.
3. Пройдите по массиву и используйте предикат для фильтрации, чтобы оставить только четные числа.
4. Выведите отфильтрованные числа на экран.

Ожидаемый результат

На экране должны быть выведены только четные числа из исходного массива.

Задание 2: Фильтрация массива строк

Задача

1. Создайте массив строк (например, `["apple", "banana", "cherry"]`).

2. Определите предикат `Predicate<String>`, который будет проверять, начинается ли строка на определенную букву (например, `"a"`).
3. Пройдите по массиву и используйте предикат для фильтрации, чтобы оставить только строки, удовлетворяющие условию.
4. Выведите отфильтрованные строки на экран.

Ожидаемый результат

На экране должны быть выведены только строки, начинающиеся на заданную букву.

Задание 3: Композиция предикатов

Задача

1. Создайте массив целых чисел (например, `[1, 4, 5, 12, 15, 22]`).
2. Определите два предиката: один для фильтрации четных чисел и один для чисел, больших 10.
3. Создайте композицию этих предикатов, используя методы `and()`, `or()` и `negate()`.
4. Пройдите по массиву и примените каждую композицию предикатов.
5. Выведите числа, которые удовлетворяют каждой композиции предикатов, на экран.

Ожидаемый результат

На экране должны быть выведены числа, которые удовлетворяют условиям каждой из созданных композиций предикатов.

Задание 4: Обобщенный метод с предикатами

Задача

1. Создайте обобщенный метод `filterArray`, который принимает массив и предикат. Метод должен вернуть новый массив, в котором останутся только элементы, удовлетворяющие условию предиката.
   - Сигнатура метода может выглядеть так: `<T> T[] filterArray(T[] array, Predicate<T> predicate)`
2. Создайте два разных массива для тестирования: один с целыми числами и один со строками.
   - Например, массив целых чисел `[1, 2, 3, 4, 5]` и массив строк `["apple", "banana", "cherry"]`.
3. Определите предикаты для каждого типа массивов:
   - Для массива чисел предикат, который фильтрует четные числа.
   - Для массива строк предикат, который фильтрует строки, начинающиеся на определенную букву (например, "a").
4. Примените `filterArray` к каждому из массивов, используя определенные предикаты.
5. Выведите результаты на экран.

Ожидаемый результат

На экране должны быть выведены отфильтрованные массивы: один с числами, которые удовлетворяют предикату, и один со строками, которые удовлетворяют предикату.

- Будьте внимательны с типами данных. Обобщенный метод должен работать с массивами любого типа данных.

Code

code/employee/src/comparators/EmployeeHireYearComparator.java

```java
package comparators;

import entity.BaseEmployee;

import java.util.Comparator;

/**
 * @author Andrej Reutow
 * created on 23.10.2023
 */
public class EmployeeHireYearComparator implements Comparator<BaseEmployee>

    @Override
    public int compare(BaseEmployee o1, BaseEmployee o2) {
        return o1.getHireYear() - o2.getHireYear();
    }
}
```

code/employee/src/entity/BaseEmployee.java

```java
package entity;

import java.util.Calendar;
import java.util.Objects;

// Абстрактный класс BaseEmployee
public abstract class BaseEmployee implements Employee {
    protected String name;
    protected Integer id; // null
    protected int hireYear;
    protected double salary;
```

```
    public BaseEmployee(String name, int hireYear) {
        this.name = name;
        this.hireYear = hireYear;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public Integer getId() {
        return id;
    }

    @Override
    public void setId(Integer id) {
        this.id = id;
    }

    public int getHireYear() {
        return hireYear;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    public void adjustSalaryByExperience(int minExperience, int maxExperien
        // todo
    }

    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder("BaseEmployee{");
        sb.append("name='").append(name).append('\'');
        sb.append(", id=").append(id);
        sb.append(", hireYear=").append(hireYear);
        sb.append(", salary=").append(salary);
```

```
            sb.append('}');
            return sb.toString() + " ";
        }

        @Override
        public boolean equals(Object object) {
            if (this == object) return true;
            if (object == null || getClass() != object.getClass()) return false

            BaseEmployee that = (BaseEmployee) object;

            if (hireYear != that.hireYear) return false;
            if (Double.compare(salary, that.salary) != 0) return false;
            if (!Objects.equals(name, that.name)) return false;
            return Objects.equals(id, that.id);
        }

        @Override
        public int hashCode() {
            int result;
            long temp;
            result = name != null ? name.hashCode() : 0;
            result = 31 * result + (id != null ? id.hashCode() : 0);
            result = 31 * result + hireYear;
            temp = Double.doubleToLongBits(salary);
            result = 31 * result + (int) (temp ^ (temp >>> 32));
            return result;
        }
    }
```

code/employee/src/entity/Developer.java

```
    package entity;

    // Класс Developer
    public class Developer extends BaseEmployee {
        private double hourlyRate;
        private int hoursWorked;

        public Developer(String name, double hourlyRate, int hoursWorked, int h
            super(name, hireYear);
            this.hourlyRate = hourlyRate;
            this.hoursWorked = hoursWorked;
        }
```

```
    @Override
    public double calculateSalary() {
        return hourlyRate * hoursWorked;
    }

    @Override
    public boolean equals(Object object) {
        if (this == object) return true;
        if (object == null || getClass() != object.getClass()) return false
        if (!super.equals(object)) return false;

        Developer developer = (Developer) object;

        if (Double.compare(hourlyRate, developer.hourlyRate) != 0) return f
        return hoursWorked == developer.hoursWorked;
    }

    @Override
    public int hashCode() {
        int result = super.hashCode();
        long temp;
        temp = Double.doubleToLongBits(hourlyRate);
        result = 31 * result + (int) (temp ^ (temp >>> 32));
        result = 31 * result + hoursWorked;
        return result;
    }

    @Override
    public String toString() {
        return "Developer{" +
                "hourlyRate=" + hourlyRate +
                ", hoursWorked=" + hoursWorked +
                ", name='" + name + '\'' +
                ", id=" + id +
                ", hireYear=" + hireYear +
                ", salary=" + salary +
                '}';
    }
}
```

code/employee/src/entity/Employee.java

```
package entity;

// Интерфейс Employee
public interface Employee {
    String getName();

    Integer getId();

    double calculateSalary();

    void setId(Integer id);
}
```

code/employee/src/entity/Manager.java

```
package entity;

// Класс Manager
public class Manager extends BaseEmployee {
    private double baseSalary;
    private int numberOfProjects;

    public Manager(String name, double baseSalary, int numberOfProjects, in
        super(name, hireYear);
        this.baseSalary = baseSalary;
        this.numberOfProjects = numberOfProjects;
    }

    @Override
    public double calculateSalary() {
        return baseSalary + (numberOfProjects * 1000);
    }


    @Override
    public String toString() {
        return "Manager{" +
                "baseSalary=" + baseSalary +
                ", numberOfProjects=" + numberOfProjects +
                ", name='" + name + '\'' +
                ", id=" + id +
                ", hireYear=" + hireYear +
                ", salary=" + salary +
```

```
                '}';
    }
}
```

code/employee/src/entity/Salesperson.java

```
package entity;

// Класс Salesperson
public class Salesperson extends BaseEmployee {
    private double baseSalary;
    private int numberOfDeals;

    public Salesperson(String name, double baseSalary, int numberOfDeals, i
        super(name, hireYear);
        this.baseSalary = baseSalary;
        this.numberOfDeals = numberOfDeals;
    }

    @Override
    public double calculateSalary() {
        return baseSalary + (numberOfDeals * 200);
    }

    @Override
    public String toString() {
        return "Salesperson{" +
                "baseSalary=" + baseSalary +
                ", numberOfDeals=" + numberOfDeals +
                ", name='" + name + '\'' +
                ", id=" + id +
                ", hireYear=" + hireYear +
                ", salary=" + salary +
                '}';
    }
}
```

code/employee/src/predicate/EmployeeHireYearPredicate.java

```
package predicate;

import entity.BaseEmployee;
```

```
import java.util.function.Predicate;

/**
 * @author Andrej Reutow
 * created on 24.10.2023
 */
public class EmployeeHireYearPredicate implements Predicate<BaseEmployee> {

    private final int fromHireYear;
    private final int toHireYear;

    public EmployeeHireYearPredicate(int fromHireYear, int toHireYear) {
        this.fromHireYear = fromHireYear;
        this.toHireYear = toHireYear;
    }

    @Override
    public boolean test(BaseEmployee baseEmployee) {
        //copy[i].getHireYear() >= fromHireYear && copy[i].getHireYear() <=
        return baseEmployee.getHireYear() >= fromHireYear && baseEmployee.g
    }
}
```

code/employee/src/predicate/EmployeeSalaryPredicate.java

```
package predicate;

import entity.BaseEmployee;

import java.util.function.Predicate;

/**
 * @author Andrej Reutow
 * created on 24.10.2023
 */
public class EmployeeSalaryPredicate implements Predicate<BaseEmployee> {

    private final int fromSalary;
    private final int toSalary;

    public EmployeeSalaryPredicate(int fromSalary, int toSalary) {
        this.fromSalary = fromSalary;
        this.toSalary = toSalary;
```

```
        }

        @Override
        public boolean test(BaseEmployee baseEmployee) {
            return baseEmployee.calculateSalary() >= fromSalary && baseEmployee
        }
    }
```

code/employee/src/repo/EmployeeRepository.java

```
    package repo;

    import entity.BaseEmployee;

    import java.util.Arrays;
    import java.util.Comparator;
    import java.util.function.Predicate;

    public class EmployeeRepository implements EmployeeRepositoryInterface {
        private BaseEmployee[] employees = new BaseEmployee[10]; // Массив для
        private int size = 0; // количесвто работников
        private static int counterId = 46985;

        // employees {1, 2 ,3, null, null, ...  } // size = 3
        // employees {1, 2 ,3 }

        public boolean addEmployee(BaseEmployee employee) {
            if (size < employees.length) {
                employees[size] = employee; // employees[3] = employees {1, 2 ,
                // size = 2
                // {1, 2, null, 4, null, ...  }
                // employees[2] = 5
                // {1, 2, 5, 4, null, ...  }

                // size = 3
                // {1, 2, null, 4, null, ...  }
                // employees[3] = 6
                // {1, 2, 5, 6, null, ...  }
                size++;
                employee.setId(++counterId);
                return true;
            } else {
                System.out.println("Репозиторий работников заполнен.");
                return false;
```

```
        }
    }

    public boolean removeEmployee(int id) { // employees[3] = employees {1,
        for (int i = 0; i < size; i++) {
            if (employees[i].getId() == id) {
//                // Если найден работник с заданным ID, удаляем его и сдви
                employees[i] = null;
                for (int j = i; j < size; j++) {
                    employees[j] = employees[j + 1];
                }
                // {1, 2 ,3, 4, null, ...  }
                // {1, 2 , null, 4, null, ...  }
                // {1, 2 , 4, null, ...    }
                // employees[2] = employees {1, 2 ,null, 4, null, ...  }
                size--;             // size = 3, -> size 2
                return true;
            }
        }
        System.out.println("Работник с ID " + id + " не найден.");
        return false;
    }

    public boolean removeEmployee2(int id) {
        for (int i = 0; i < size; i++) {
            if (employees[i] != null && employees[i].getId() == id) {
                employees[i] = null;
                System.out.println("Работник с ID " + id + " уволен");
                return true;
            }
        }
        System.out.println("Работник с ID " + id + " не найден.");
        return false;
    }

    public BaseEmployee findEmployeeById(int id) {
        for (int i = 0; i < size; i++) {
            if (employees[i].getId() == id) {
                return employees[i];
            }
        }
        return null;
    }
```

```
    public BaseEmployee[] getAllEmployees() {
        BaseEmployee[] result = new BaseEmployee[size];
        for (int i = 0; i < size; i++) {
            result[i] = employees[i];
        }
        return result;
    }

    public int countEmployees() {
        return size;
    }

    @Override
    public BaseEmployee[] sortByComparator(Comparator<BaseEmployee> compara
//        BaseEmployee[] sortedArray = new BaseEmployee[size];
//        for (int i = 0; i < size; i++) {
//            sortedArray[i] = employees[i];
//        }
        BaseEmployee[] sortedArray = Arrays.copyOf(employees, size);
        Arrays.sort(sortedArray, comparator);
        return sortedArray;
    }

//    @Override
//    public BaseEmployee[] filterByHireYear(int fromYear, int toYear) {
//        BaseEmployee[] copy = Arrays.copyOf(employees, size);
//
//        int filterCounter = 0;
//        for (int i = 0; i < copy.length; i++) {
//            if (copy[i].getHireYear() >= fromYear && copy[i].getHireYear(
//                filterCounter++;
//            }
//        }
//
//        BaseEmployee[] result = new BaseEmployee[filterCounter];
//        for (int i = 0, j = 0; j < result.length; i++) {
//            if (copy[i].getHireYear() >= fromYear && copy[i].getHireYear(
//                result[j] = copy[i];
//                j++;
//            }
//        }
//
```

```
//        return result;
//    }

    @Override
    public BaseEmployee[] filterBy(Predicate<BaseEmployee> predicate) {
        BaseEmployee[] copy = Arrays.copyOf(employees, size);

        int filterCounter = 0;
        for (int i = 0; i < copy.length; i++) {
            if (predicate.test(copy[i])) {
                filterCounter++;
            }
        }

        BaseEmployee[] result = new BaseEmployee[filterCounter];
        for (int i = 0, j = 0; j < result.length; i++) {
            if (predicate.test(copy[i])) {
                result[j] = copy[i];
                j++;
            }
        }

        return result;
    }

//    @Override
//    public BaseEmployee[] filterBySalary(double fromSalary, double toSala
//        BaseEmployee[] copy = Arrays.copyOf(employees, size);
//
//        int filterCounter = 0;
//        for (int i = 0; i < copy.length; i++) {
//            if (copy[i].calculateSalary() >= fromSalary && copy[i].calcul
//                filterCounter++;
//            }
//        }
//
//        BaseEmployee[] result = new BaseEmployee[filterCounter];
//        for (int i = 0, j = 0; j < result.length; i++) {
//            if (copy[i].calculateSalary() >= fromSalary && copy[i].calcul
//                result[j] = copy[i];
//                j++;
//            }
//        }
```

```
    //
    //        return result;
    //    }

        public BaseEmployee[] getAll() {
            return employees;
        }
    }
```

code/employee/src/repo/EmployeeRepositoryInterface.java

```
    package repo;

    import entity.BaseEmployee;

    import java.util.Comparator;
    import java.util.function.Predicate;

    /**
     * Интерфейс для репозитория работников.
     */
    public interface EmployeeRepositoryInterface {

        /**
         * Добавляет работника в репозиторий.
         *
         * @param employee Добавляемый работник.
         */
        boolean addEmployee(BaseEmployee employee);

        /**
         * Удаляет работника из репозитория по его ID.
         *
         * @param id ID работника, которого необходимо удалить.
         */
        boolean removeEmployee(int id);

        /**
         * Ищет работника в репозитории по его ID.
         *
         * @param id ID работника, которого необходимо найти.
         * @return Найденный работник или null, если работник не найден.
         */
        BaseEmployee findEmployeeById(int id);
```

```
    /**
     * Получает массив всех работников в репозитории.
     *
     * @return Массив всех работников в репозитории.
     */
    BaseEmployee[] getAllEmployees();

    /**
     * Возвращает количество работников в репозитории.
     *
     * @return Количество работников в репозитории.
     */
    int countEmployees();

    /**
     * Метод сортирует массив работников на основе компаратора
     *
     * @param comparator определят как сравнивать объекты
     * @return отсортированный массив
     */
    BaseEmployee[] sortByComparator(Comparator<BaseEmployee> comparator);

    /**
     * Метод возвращает список рабтников которые были устроены на работу в у
     *
     * @param fromYear от какого годо
     * @param toYear   до какого годо
     * @return отфильтрованный массив
     */
//    BaseEmployee[] filterByHireYear(int fromYear, int toYear);

//    BaseEmployee[] filterBySalary(double fromSalary, double toSalary);

    BaseEmployee[] filterBy(Predicate<BaseEmployee> predicate);
}
```

code/employee/src/repo/EmployeeRepositoryTest.java

```
package repo;

import entity.BaseEmployee;
import entity.Developer;
import org.junit.jupiter.api.Assertions;
```

```java
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import predicate.EmployeeHireYearPredicate;
import predicate.EmployeeSalaryPredicate;

import java.util.function.Predicate;

/**
 * @author Andrej Reutow
 * created on 18.10.2023
 */
class EmployeeRepositoryTest {

    private EmployeeRepository repository;

    @BeforeAll // tests
    public static void init() {
        System.out.println("@BeforeAll");
    }

    @BeforeEach // test
    public void setUp() {
        System.out.println("@BeforeEach");
        repository = new EmployeeRepository();
    }

    @Test
    void test_countEmployees() {

        int result = repository.countEmployees();

        Assertions.assertEquals(0, result);
    }

    @Test
    void test_removeEmployee_() {
        //Дано
        BaseEmployee developer1 = new Developer("dev1", 100, 180, 2023);
        BaseEmployee developer2 = new Developer("dev2", 200, 150, 2023);
        BaseEmployee developer3 = new Developer("dev3", 300, 200, 2023);

        repository.addEmployee(developer1);
```

```
        repository.addEmployee(developer2);
        repository.addEmployee(developer3);

        BaseEmployee[] employees = repository.getAll();
        Assertions.assertEquals(developer2.getName(), employees[1].getName(

        // Когда
        boolean isRemoved = repository.removeEmployee(developer2.getId());

        // Тогда
        Assertions.assertTrue(isRemoved);
        Assertions.assertEquals(2, repository.countEmployees());

        Assertions.assertEquals(developer1.getName(), employees[0].getName(
        Assertions.assertEquals(developer3.getName(), employees[1].getName(
        for (int i = 2; i < employees.length; i++) {
            Assertions.assertNull(employees[i]);
        }
    }

    @Test
    public void test_filterBySalary() {
        BaseEmployee developer1 = new Developer("dev1", 100, 180, 2012); //
        BaseEmployee developer2 = new Developer("dev2", 200, 150, 2018); //
        BaseEmployee developer3 = new Developer("dev3", 300, 200, 2024); //
        BaseEmployee developer4 = new Developer("dev3", 400, 200, 2020); //

        repository.addEmployee(developer4);
        repository.addEmployee(developer1);
        repository.addEmployee(developer3);
        repository.addEmployee(developer2);

        Predicate<BaseEmployee> predicate = new EmployeeSalaryPredicate(30_
        BaseEmployee[] filteredBySalary = repository.filterBy(predicate);

//        Assertions.assertEquals(2, filteredBySalary.length);
        BaseEmployee[] expected = {developer3, developer2};
        Assertions.assertArrayEquals(expected, filteredBySalary);
    }

    @Test
    public void test_filterByHireYear() {
        BaseEmployee developer1 = new Developer("dev1", 100, 180, 2012); //
```

```
        BaseEmployee developer2 = new Developer("dev2", 200, 150, 2018); //
        BaseEmployee developer3 = new Developer("dev3", 300, 200, 2024); //
        BaseEmployee developer4 = new Developer("dev3", 400, 200, 2020); //

        repository.addEmployee(developer4);
        repository.addEmployee(developer1);
        repository.addEmployee(developer3);
        repository.addEmployee(developer2);

        Predicate<BaseEmployee> predicate = new EmployeeHireYearPredicate(2
        BaseEmployee[] filteredBySalary = repository.filterBy(predicate);

//        Assertions.assertEquals(2, filteredBySalary.length);
        BaseEmployee[] expected = {developer1, developer2};
        Assertions.assertArrayEquals(expected, filteredBySalary);
    }

    @Test
    public void test_filterBy_predicateFilterByNameStartsWith() {
        BaseEmployee developer1 = new Developer("dev1", 100, 180, 2012); //
        BaseEmployee developer2 = new Developer("dev2", 200, 150, 2018); //
        BaseEmployee developer3 = new Developer("dev3", 300, 200, 2024); //
        BaseEmployee developer4 = new Developer("dev3", 400, 200, 2020); //

        repository.addEmployee(developer4);
        repository.addEmployee(developer1);
        repository.addEmployee(developer3);
        repository.addEmployee(developer2);

        Predicate<BaseEmployee> predicate = e -> e.getName().startsWith("d"
        BaseEmployee[] filteredBySalary = repository.filterBy(predicate);

//        Assertions.assertEquals(2, filteredBySalary.length);
        BaseEmployee[] expected = {developer4, developer1, developer3, deve
        Assertions.assertArrayEquals(expected, filteredBySalary);
    }
}
```

code/employee/src/test_entity/Animal.java

```
package test_entity;

import tools.Id;
```

```java
/**
 * @author Andrej Reutow
 * created on 24.10.2023
 */
public class Animal implements Id {
    @Override
    public long getId() {
        return 1;
    }
}
```

code/employee/src/test_entity/Car.java

```java
package test_entity;

import tools.Id;

import java.util.Objects;

/**
 * @author Andrej Reutow
 * created on 24.10.2023
 */
public class Car implements Id {

    private final int id;
    private String brand;

    public Car(int id, String brand) {
        this.id = id;
        this.brand = brand;
    }

    @Override
    public long getId() {
        return this.id;
    }

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
```

```
    }

    @Override
    public boolean equals(Object object) {
        if (this == object) return true;
        if (object == null || getClass() != object.getClass()) return false

        Car car = (Car) object;

        return Objects.equals(brand, car.brand);
    }

    @Override
    public int hashCode() {
        return brand != null ? brand.hashCode() : 0;
    }

    @Override
    public String toString() {
        return "Car{" +
                "brand='" + brand + '\'' +
                '}';
    }


}
```

code/employee/src/tools/ArrayTools.java

```
package tools;

import entity.BaseEmployee;

/**
 * @author Andrej Reutow
 * created on 24.10.2023
 */
public class ArrayTools {

    private ArrayTools() {
    }

    public static <T> void print(T[] array) {
        for (int i = 0; i < array.length; i++) {
```

```
                System.out.println(array[i]);
            }
        }
    //
    //    public static <T> T search(T[] source, T value) {
    //
    //    }

        public static <T extends Id> T searchById(T[] source, long id) {
            for (int i = 0; i < source.length; i++) {
                if (id == source[i].getId()) {
                    return source[i];
                }
            }
            return null;
        }

    //    public static <T> boolean removeById(T[] source, long id) {
    //
    //    }
    }
```

code/employee/src/tools/ArrayToolsTest.java

```
package tools;

import entity.BaseEmployee;
import entity.Developer;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import test_entity.Animal;
import test_entity.Car;

/**
 * @author Andrej Reutow
 * created on 24.10.2023
 */
public class ArrayToolsTest {

    @Test
    public void testPrintArray() {
        BaseEmployee developer1 = new Developer("dev1", 100, 180, 2012); //
        BaseEmployee developer2 = new Developer("dev2", 200, 150, 2018); //
        BaseEmployee developer3 = new Developer("dev3", 300, 200, 2024); //
```

```
        BaseEmployee developer4 = new Developer("dev3", 400, 200, 2020); //
        BaseEmployee[] employees = {
                developer1,
                developer2,
                developer3,
                developer4
        };

        ArrayTools.print(employees);

        Car[] cars = {
                new Car(1, "Bmw"),
                new Car(2, "Audi"),
                new Car(3, "VW"),
        };

        ArrayTools.print(cars);
    }

    @Test
    void test_searchById() {
        BaseEmployee developer1 = new Developer("dev1", 100, 180, 2012); //
        BaseEmployee developer2 = new Developer("dev2", 200, 150, 2018); //
        BaseEmployee developer3 = new Developer("dev3", 300, 200, 2024); //
        BaseEmployee developer4 = new Developer("dev3", 400, 200, 2020); //
        BaseEmployee[] employees = {
                developer1,
                developer2,
                developer3,
                developer4
        };

        Car[] cars = {
                new Car(1, "Bmw"),
                new Car(2, "Audi"),
                new Car(3, "VW"),
        };

        Car result = ArrayTools.searchById(cars, 2);
        Assertions.assertEquals(cars[1], result);

        Animal[] animals = {
                new Animal(),
```

```
                new Animal(),
                new Animal(),
                new Animal()
        };
        Animal animalResult = ArrayTools.searchById(animals, 1);
        Assertions.assertEquals(animals[0], animalResult);

        Animal animalResult2 = ArrayTools.searchById(animals, 2);
        Assertions.assertNull(animalResult2);
    }
}
```

code/employee/src/tools/Id.java

```
package tools;

/**
 * @author Andrej Reutow
 * created on 24.10.2023
 */
public interface Id {

   long getId();
}
```

code/employee/src/Main.java

```
import entity.BaseEmployee;
import entity.Developer;
import entity.Manager;
import entity.Salesperson;
import repo.EmployeeRepository;

import java.util.Comparator;

public class Main {
    public static void main(String[] args) {
        EmployeeRepository repository = new EmployeeRepository();

        BaseEmployee developer1 = new Developer("John", 25.0, 160, 2022);
        BaseEmployee developer2 = new Developer("Alice", 30.0, 150, 2020);
        BaseEmployee manager1 = new Manager("Bob", 3000.0, 5, 2019);
        BaseEmployee salesperson1 = new Salesperson("Eve", 2000.0, 10, 2021
```

```
        repository.addEmployee(developer1);
        repository.addEmployee(developer2);
        repository.removeEmployee(46987);
        repository.addEmployee(manager1);
        repository.addEmployee(salesperson1);


//        // Повысить зарплату для работников с опытом от 2 до 5 лет на 10%
//        BaseEmployee[] allEmployees = repository.getAllEmployees();
//        for (int i = 0; i < allEmployees.length; i++) {
//            Employee employee = allEmployees[i];
//            if (employee instanceof BaseEmployee) {
//                BaseEmployee baseEmployee = (BaseEmployee) employee;
//                baseEmployee.adjustSalaryByExperience(2, 5, 10);
//            }
//        }

        // Вывести информацию о работниках
        BaseEmployee[] allEmployees = repository.getAllEmployees();
        for (int i = 0; i < allEmployees.length; i++) {
            BaseEmployee employee = allEmployees[i];
            System.out.println(employee);
        }

        System.out.println("Sort by...");
        System.out.println("\nSort by hire year");
//        Comparator<BaseEmployee> comparatorHireYear = new EmployeeHireYea
        Comparator<BaseEmployee> comparatorHireYear = new Comparator<BaseEm
            @Override
            public int compare(BaseEmployee o1, BaseEmployee o2) {
                return o1.getHireYear() - o2.getHireYear();
            }
        };
//        BaseEmployee[] sortedByHireYear = repository.sortByComparator(com
        BaseEmployee[] sortedByHireYear = repository.sortByComparator((o1, 
        printArray(sortedByHireYear);

        System.out.println("\nSort by id");
//        Comparator<BaseEmployee> comparatorById = new Comparator<BaseEmpl
//
//            @Override
//            public int compare(BaseEmployee o1, BaseEmployee o2) {
//                return o1.getId() - o2.getId();
```

```
//            }
//
//        };
//        Comparator<BaseEmployee> comparatorById = (o1, o2) ->  o1.getId()
        Comparator<BaseEmployee> comparatorById = (o1, o2) -> o1.getId().co
        BaseEmployee[] sortedById = repository.sortByComparator(comparatorB
        BaseEmployee[] sortedByIdReversed = repository.sortByComparator(com
        printArray(sortedById);
        System.out.println("Reversed");
        printArray(sortedByIdReversed);


        System.out.println("\nSort by name");
        Comparator<BaseEmployee> comparatorByName = (o1, baseEmployee2) ->
        BaseEmployee[] sortedByName = repository.sortByComparator(comparato
        printArray(sortedByName);
    }


    private static void printArray(BaseEmployee[] array) {
        for (int i = 0; i < array.length; i++) {
            BaseEmployee employee = array[i];
            System.out.println(employee);
        }
    }
}
```

code/employee/src/Main2.java

```
import entity.BaseEmployee;
import entity.Developer;
import entity.Manager;
import entity.Salesperson;
import repo.EmployeeRepository;

import java.util.Arrays;
import java.util.Comparator;
import java.util.function.Predicate;

public class Main2 {
    public static void main(String[] args) {
        EmployeeRepository repository = new EmployeeRepository();

        BaseEmployee developer1 = new Developer("John", 25.0, 160, 2022);
```

```
        BaseEmployee developer2 = new Developer("Alice", 30.0, 150, 2020);
        BaseEmployee manager1 = new Manager("Bob", 3000.0, 5, 2019);
        BaseEmployee salesperson1 = new Salesperson("Eve", 2000.0, 10, 2021

        repository.addEmployee(developer1);
        repository.addEmployee(developer2);
        //repository.removeEmployee(developer2.getId());
        repository.addEmployee(manager1);
        repository.addEmployee(salesperson1);


        System.out.println("Sort by...");

        System.out.println("\nSort by hire year");
        printArray(repository.sortByComparator((o1, o2) -> o1.getHireYear()

        System.out.println("\nSort by id");
        BaseEmployee[] sortedById = repository.sortByComparator((o1, o2) ->
        printArray(sortedById);

        System.out.println("\nSort by name");
        Comparator<BaseEmployee> comparatorByName = (o1, o2) -> o1.getName(
        BaseEmployee[] sortedByName = repository.sortByComparator(comparato
        printArray(sortedByName);


        System.out.println("\nFilter By");
        System.out.println("\nFilter by hire year");
//        Predicate<BaseEmployee> filterByHireYear = new EmployeeHireYearPr
//        Predicate<BaseEmployee> filterByHireYear = new Predicate<BaseEmpl
//            @Override
//            public boolean test(BaseEmployee baseEmployee) {
//                return baseEmployee.getHireYear() >= 2019 && baseEmployee
//            }
//        };
        Predicate<BaseEmployee> filterByHireYear = entity -> entity.getHire


        BaseEmployee[] filteredByHireYear = repository.filterBy(filterByHir
        Arrays.sort(filteredByHireYear, Comparator.comparingInt(baseEmploye
        printArray(filteredByHireYear);

        System.out.println("\nFilter by Name");
```

```
//        Predicate<BaseEmployee> filterByName = (be -> be.getName().endsWi
        Predicate<BaseEmployee> filterByName = new Predicate<BaseEmployee>(
            @Override
            public boolean test(BaseEmployee baseEmployee) {
                return baseEmployee.getName().endsWith("e");
            }
        };
        BaseEmployee[] filteredByName = repository.filterBy(filterByName);
        printArray(filteredByName);
    }


    private static void printArray(BaseEmployee[] array) {
        for (int i = 0; i < array.length; i++) {
            BaseEmployee employee = array[i];
            System.out.println(employee);
        }
    }
}
```