**Pros of this Approach**

1. Easier to maintain and readable.
2. Simple, hence easy to debug and write tests.

**Cons of this Approach:**

1. Non-generic code and needs new methods on every addition of new fields.

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collections;

import java.util.Date;

import java.util.List;

public class Person {

      public String ssn;

      public Date dateOfBirth;

      public String firstName;

      public String lastName;

      public Double heightIn;

      public Double weightLb;

      public Person(String ssn, Date date, String fname, String lname, double height, double weight) {

        this.ssn = ssn;

        this.dateOfBirth = date;

        this.firstName = fname;

        this.lastName = lname;

        this.heightIn = height;

        this.weightLb = weight;

    }

    static List<Person> sort(List<Person> people, String sortField, String ascending){

        List<Person> sortedList = new ArrayList<>();

        switch(sortField.toLowerCase()){

            case "ssn":

                 sortedList = SORT\_BY\_SSN(people, ascending);

            case "height":

                return SORT\_BY\_HEIGHT(people, ascending);

            //other cases here.

            default:

                return people;

        }

      }

    public static List<Person> SORT\_BY\_SSN(List<Person> people, String ascending){

        if(ascending.equals("true"))

            Collections.sort(people, (person1, person2) -> person1.ssn.compareToIgnoreCase(person2.ssn));

        else

            Collections.sort(people, (person1, person2) -> person2.ssn.compareToIgnoreCase(person1.ssn));

        return people;

    }

    public static List<Person> SORT\_BY\_HEIGHT(List<Person> people, String ascending){

        if(ascending.equals("true"))

            Collections.sort(people, (person1, person2) -> person1.heightIn.compareTo(person2.heightIn));

        else

            Collections.sort(people, (person1, person2) -> person2.heightIn.compareTo(person1.heightIn));

        return people;

    }

    //If you want to use iterable

    public static <Person> List<Person> iterableToCollection(Iterable<Person> iterable)

    {

        List<Person> collection = new ArrayList<>();

        iterable.forEach(collection::add);

        return collection;

    }

      public static void main(String args[]){

          Person p1 = new Person("1234", new ~~Date~~(1990, 1, 1), "pari", "mj", 2D, 3D);

          Person p2 = new Person("1235", new ~~Date~~(1990, 1, 1), "pari", "mj", 2D, 3D);

          Person p3 = new Person("1231", new ~~Date~~(1990, 1, 1), "pari", "mj", 2D, 3D);

          Person[] people = {p1, p2, p3};

          System.out.println(Arrays.asList(people));

          System.out.println(sort(Arrays.asList(people),"ssn","false"));

      }

}

**Other way of Achieving the Same Results**  
  
We can use the Fields class in java which uses reflection and write one generic method but it is never a good practice to reflections in java.

**Pros of this Approach**

1. One generic method to Sort by any of the Fields.

**Cons of this Approach**

1. The stack trace will not give the right picture in case of reflections.
2. Hard to debug and maintain the code.