Почти всички типове в Java се моделират като класове. Следователно при работа с файлове има възможност да се записват и четат цели обекти. Процесът по преобразуване на обект към поток от байтове се нарича сериализация. Java предоставя два интерфейса за постигане на тази цел – Serializable и Externalizable.

Нека създадем един примерен клас, който бихме искали да запишем във файл:

```
public class Person
{
   private String name;
   private LocalDate birth;
   private int age;
   …
```

Следва да се имплементира Serializable. При разглеждане на сорс-кода на интерфейса се забелязва, че в него няма декларирани методи или променливи. Такъв „празен“ интерфейс се нарича маркер. Това е така, защото при използване на Serializable, виртуалната машина има грижата да генерира инструкциите за запис и четене на полетата на класа. Достатъчно е само да се маркира, че даден клас ще има тази функционалност:

```
public class Person implements Serializable { …
```

Операциите се извършват чрез специализирани потоци за работа с обекти – ObjectOutputStream и ObjectInputStream. Примерен код за запис на обект от тип Person във файл:

```
final String filePath = “person.bin”;
try(ObjectOutputStream out =new ObjectOutputStream(newFileOutputStream(filePath)))
    {
       out.writeObject(person1);
    }
    catch (IOException e)
    {
       // handle IO Exception
    }
```

Четенето е аналогично:

```
try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(filePath)))
    {
       personFromFile = (Person) in.readObject();
    }
    catch (IOException | ClassNotFoundException e)
    …
```

Една особеност е, че тъй като методът readObject() връща Object, е необходимо да се извърши кастване към очаквания тип.

Почти всички стандартни класове в Java по подразбиране имплементират интерфейса Serializable. За сериализация на полето birth (клас LocalDate) не се налага писане на допълнителен код. Маркер интерфейсът е необходим единствено в създадените от програмиста класове.

Serializable дава ограничен контрол над това, кои полета ще бъдат сериализирани. Например полето на Person age лесно може да се изчисли спрямо рождената дата. За да не се хаби дисково пространство при запис във файл, то би могло да бъде маркирано с ключовата дума **transient**. Transient полетата не се записват и при четене приемат стойност по подразбиране за типа:

private transient int age;

За инициализиране на коректната възраст при четене на обекта се налага ръчно да се извика метод, който да я преизчисли:

personFromFile = (Person) in.readObject();

personFromFile.recalculateAge();

Интерфейсът Externalizable наследява Serializable и дава пълен контрол върху процедурите по запис и четене. В предоставените от него методи се указва кои полета да се сериализират, в какъв ред, стойности по подразбираме и т.н. Недостатъкът е, че вече се налага програмистът да създаде кода за сериализация, вместо той да бъде генериран от средата:

```
public class Person implements Externalizable
{
   private String name;
   private LocalDate birth;
   private int age;
   @Override
   public void writeExternal(ObjectOutput out) throws IOException
   {
      out.writeObject(name);
      out.writeObject(birth);
   }
@Override
   public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
   {
      name = (String) in.readObject();
      birth = (LocalDate) in.readObject();
      recalculateAge();
   }
```

Предимството тук е, че изчисляването на възрастта спрямо рождената дата вече е част от четенето и не се налага да се прави ръчно. По аналогичен начин би могла да се зададе специализирана логика за всички полета на класа. Типичен пример е поддържане на различни версии на един и същ обект с добавени или премахнати полета без да се налага презаписване на по-стари файлове. Това прави интерфейса Externalizable по-гъвкав от Serializable.

Както бе споменато по-горе Externalizable наследява Serializable. Следователно кодът за запис и четене остава същият – отново се използват ObjectInput и Output Stream. Това се вижда и от параметрите на методите writeExternal() и readExternal(). Те работят именно с тези потоци.

При четене на Externalizable обектите се появява допълнителна особеност – кои полета ще бъдат четени не е предварително известно на средата. Това означава, че те не могат да бъдат инициализирани при създаване на обекта. Необходимо е първо да се създаде инстанцията и след това да се изпълни метода readExternal(). Следователно при използване на интерфейса Externalizable за четене, програмистът е задължен да предостави публичен конструктор без параметри:

```
/*
Default constructor required by Externalizable.
 */
public Person() {  }
```

Това е често срещана практика и при използване на библиотеки за сериализация на обекти към други популярни формати (например JSON и XML).

Важно е да се отбележи, че използването на ObjectOutputStream и ObjectInputStream не е ограничено до файлови потоци. Те могат да бъдат използвани при всякакъв вход/изход (например мрежови).

Задача:

Да се реализира като работеща програма гореописаният пример с клас Person.

Да се модифицира програмата така, че да записва/чете масив от Person.