Потоки ввода-вывода и сериализация объектов

Пользоваться форматом записей фиксированной длины, безусловно, удобно, если сохранять объекты одинакового типа. Но ведь объекты, создаваемые в объектно-ориентированных программах, редко бывают одного и того же типа. Например, может существовать массив staff, номинально представляющий собой массив записей типа Employee, но фактически содержащий объекты, которые, по существу, являются экземплярами какого-нибудь подкласса вроде Manager.

Конечно, можно было бы подобрать такой формат данных, который позволял бы сохранять подобные полиморфные коллекции, но, к счастью, в этом нет никакой необходимости. В Java поддерживается универсальный механизм, называемый сериализацией объектов и предоставляющий возможность записать любой объект в поток ввода-вывода, а в дальнейшем считать его снова.

Для сохранения объектных данных необходимо прежде всего открыть поток вывода объектов типа ObjectOutputStream следующим образом:

```
ObjectOutputStream out =
new ObjectOutputStream(new FileOutputStream("employee.dat"));
```

Далее для сохранения объекта остается лишь вызвать метод writeObject() из класса ObjectOutputStream, как показано в приведенном ниже фрагменте кода.

```
Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
out.writeObject(harry);
out.writeObject(boss);
```

А для того чтобы считать объект обратно, нужно сначала получить объект типа ObjectInputStream, т.е. поток ввода объектов, следующим образом:

```
ObjectInputStream in =
new ObjectInputStream(new FileInputStream("employee.dat"));
```

И затем извлечь объекты в том порядке, в котором они записывались, вызвав метод readObject (), как показано ниже.

```
Employee el = (Employee) in.readObject();
Employee e2 = (Employee) in.readObject()
```

Имеется, однако, одно изменение, которое нужно внести в любой класс, объекты которого требуется сохранить и восстановить в потоке ввода-вывода объектов, а именно: каждый такой класс должен обязательно реализовать интерфейс Serializable следующим образом:

```
class Employee implements Serializable { . . . }
```

У интерфейса Serializable отсутствуют методы, поэтому изменять каким-то образом свои собственные классы не нужно. Для того, чтобы сделать класс пригодным для сериализации, ничего больше делать не нужно.

НА ЗАМЕТКУ! Записывать и считывать только объекты можно и с помощью методов writeObject() и readObject(). Что же касается значений простых типов, то для их ввода-вывода следует применять такие методы, как writeInt () и readInt () или

writeDouble () и readDouble(). (Классы потоков ввода-вывода объектов реализуют интерфейсы DataInput и DataOutput.)

Класс ObjectOutputStream просматривает подспудно все поля объектов и сохраняет их содержимое. Так, при записи объекта типа Employee в поток вывода записывается содержимое полей Ф.И.О., даты зачисления на работу и зарплаты сотрудника.

Необходимо, однако, рассмотреть очень важный вопрос: что произойдет, если один объект совместно используется рядом других объектов? Чтобы проиллюстрировать важность данного вопроса, внесем одно небольшое изменение в класс Manager. В частности, допустим, что у каждого руководителя имеется свой секретарь, как показано в приведенном ниже фрагменте кода.

```
class Manager extends Employee
{
private Employee secretary;
• • •
}
```

Теперь каждый объект типа Manager будет содержать ссылку на объект типа Employee, описывающий секретаря. Безусловно, два руководителя вполне могут пользоваться услугами одного и того же секретаря, как показано на рис. 1. и в следующем фрагменте кода:

```
harry = new Employee("Harry Hacker", . . .);
Manager carl = new Manager("Carl Cracker", . . .);
carl.setSecretary(harry);
Manager tony = new Manager("Tony Tester", . . .);
tony.setSecretary(harry);
```

Сохранение такой разветвленной сети объектов оказывается непростой задачей.

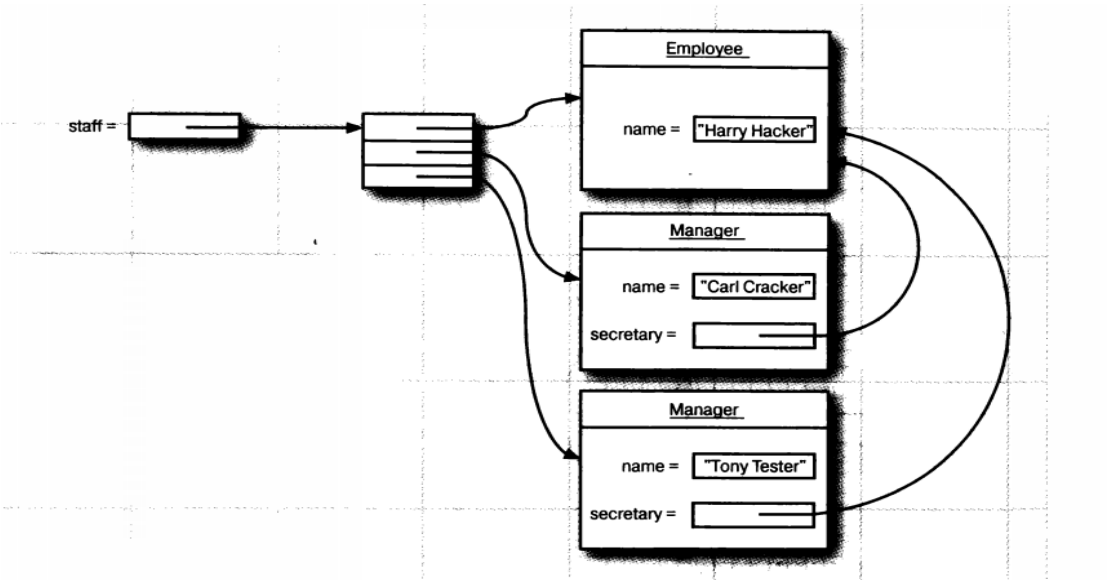

Рисунок 1. Два руководителя могут совместно пользоваться услугами одного и того же сотрудника в качестве секретаря

Разумеется, сохранять и восстанавливать адреса ячеек памяти для объектов секретарей нельзя. При повторной загрузке каждый такой объект, скорее всего, будет занимать уже совершенно другую ячейку памяти по сравнению с той, которую он занимал первоначально.

Поэтому каждый такой объект сохраняется под серийным номером, откуда, собственно говоря, и происходит название механизма *сериализации объектов*. Ниже описывается порядок действий при *сериализации* объектов.

1. Серийный (т.е. порядковый) номер связывается с каждой встречающейся ссылкой на объект, как показано на рис. 2.

2. Если ссылка на объект встречается впервые, данные из этого объекта сохраняются в потоке ввода-вывода объектов.

3. Если же данные были ранее сохранены, просто добавляется метка "same as previously saved object with serial number x" (совпадает с объектом, сохраненным ранее под серийным номером x).

При чтении объектов обратно из потока ввода-вывода объектов порядок действий меняется на обратный.

1. Если объект впервые указывается в потоке ввода-вывода объектов, он создается и инициализируется данными из потока, а связь серийного номера с ссылкой на объект запоминается.

2. Если же встречается метка "same as previously saved object with serial number x", то извлекается ссылка на объект по данному серийному номеру.

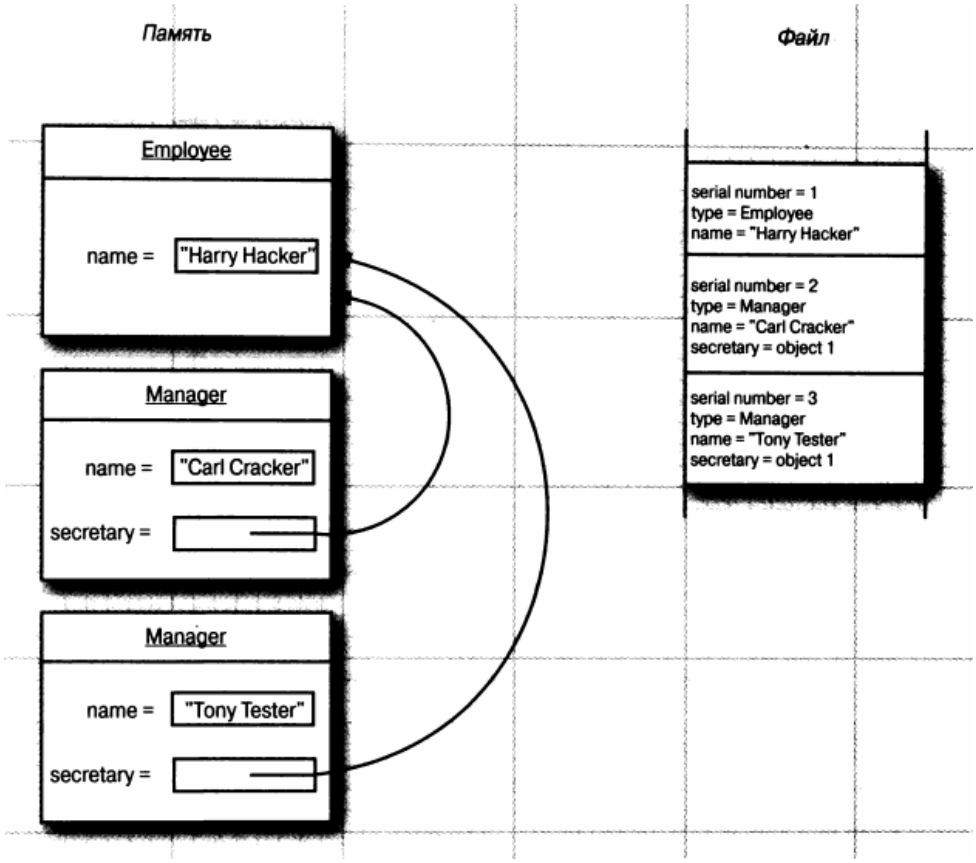

Рис. 2. Пример сериализации объектов.

В листинге 1. приведен исходный код примера программы, способной сохранять и перезагружать сеть объектов типа Employee и Manager (некоторые из руководителей пользуются услугами общего сотрудника в качестве секретаря). Обратите внимание на то, что объект секретаря остается однозначным после повторной перезагрузки, т.е. когда сотрудник, представленный объектом, хранящимся в элементе массива newStaff [1], получает повышение, это отражается в полях secretary объектов типа Manager.

Листинг 1. Код ObjectStreamTest.java

```
package objectStream;
import java.io.*;
class ObjectStreamTest
{
   public static void main(String[] args) throws IOException, 
ClassNotFoundException
   {
    Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
    Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
      carl.setSecretary(harry);
      Manager tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
      tony.setSecretary(harry);
      Employee[] staff = new Employee[3];
      staff[0] = carl;
      staff[1] = harry;
      staff[2] = tony;
      // save all employee records to the file employee.dat
      try (ObjectOutputStream out = new ObjectOutputStream(new
                    FileOutputStream("employee.dat")))    {
         out.writeObject(staff);
      }
      try (ObjectInputStream in = new ObjectInputStream(
               new FileInputStream("employee.dat")))      {
         // retrieve all records into a new array

         Employee[] newStaff = (Employee[]) in.readObject();

         // raise secretary's salary
         newStaff[1].raiseSalary(10);

         // print the newly read employee records
         for (Employee e : newStaff)
            System.out.println(e);
      }
   }}
```

Листинг 2. Код Employee.java

```
package objectStream;
import java.io.*;
import java.util.*;
public class Employee implements Serializable{
   private String name;
   private double salary;
   private Date hireDay;

   public Employee()   {
   }

   public Employee(String n, double s, int year, int month, int day)  {
      name = n;
      salary = s;
      GregorianCalendar calendar =
                  new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()   {
      return name;
   }

   public double getSalary()   {
      return salary;
   }

   public Date getHireDay()   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)  {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public String toString()   {
      return getClass().getName() + "[name=" + name + ",salary=" + salary
+ ",hireDay=" + hireDay
               + "]";
   }
}
```

Листинг 3. Код Manager.java

```
package objectStream;
public class Manager extends Employee{
   private Employee secretary;
   /**
    * Constructs a Manager without a secretary
    * @param n the employee's name
    * @param s the salary
    * @param year the hire year
    * @param month the hire month
    * @param day the hire day
    */
   public Manager(String n, double s, int year, int month, int day)   {
      super(n, s, year, month, day);
      secretary = null;
   }
   /**
    * Assigns a secretary to the manager.
    * @param s the secretary
    */
   public void setSecretary(Employee s)   {
      secretary = s;
   }

   public String toString()   {
      return super.toString() + "[secretary=" + secretary + "]";
   }
}
```