Лекция 9

Подготовка класса к использованию

Процесс подготовки класса к использованию имеет три стадии

1. **загрузка;**

   получение байт-кода, реализующего класс, и создание объекта Class;

2. **связывание;**

   проверка того, удовлетворяет ли байт-код синтаксису языка, выделение необходимой памяти для хранения статических данных и (эта функция не обязательна) разрешение всех ссылок, содержащихся в коде класса, с загрузкой, если это необходимо, классов, соответствующих таким ссылкам;

3. **инициализация;**

инициализация базового класса (с его предварительной загрузкой и связыванием, если это необходимо) и вычисление всех выражений и блоков статической инициализации класса.

Объект **Class**, возвращаемый методом **defineClass**, служит только цели представления загруженного класса — класс все еще остается не связанным. Выполнение процедуры связывания может быть принудительно спровоцировано вызовом метода **resolveClass**.

Этот метод имеет вид

protected final void resolveClass(Class c)

Связывает класс, если тот еще не связан.

Виртуальная машина, приступая к инициализации класса, проверяет, выполнено ли его связывание.

Класс должен быть инициализирован прежде, чем может быть создан объект класса, либо вызван статический метод класса, либо осуществлен доступ к статическому полю класса.

Особенности процесса зависят от конкретной реализации виртуальной машины Java.

Загрузка дополнительных ресурсов

Классы являются основными ресурсами, в которых нуждается программа, но некоторым классам необходимы также дополнительные ресурсы, такие как фрагменты текста, изображения или данные мультимедиа.

Рассмотрим пример загрузки дополнительных ресурсов

```
String b = “Str1.fl”;
 InputStream in;
 ClassLoader loader = this.getClass().getClassLoader();
 if (loader!= null)
    in = loader.getResourceAsStream(b);
 else
    in = ClassLoader.getSystemResourceAsStream(b);
```

Системные ресурсы связаны с системными классами, а таковые могут не обладать загрузчиками.

Статический метод **getSystemResourceAsStream** возвращает объект **InputStream** потока ввода для ресурса с заданным именем.

Реализация метода **getResourceAsStream**, предлагаемая классом **ClassLoader**, предусматривает возврат значения null.

Задача переопределения метода с поддержкой нужного алгоритма поиска ресурсов возлагается на производный класс загрузчика.

Обычно это значит, что ресурсы будут найдены в тех же местах и теми же способами, что и файлы классов.

Например, для класса PlayerLoader (определенного в предыдущей лекции) имеем:

```
public  InputStream
                    getResourceAsStream(String name){
  try{
     return streamFor(name);
   }catch (IOException e)  {  return null; }
}
```

Два других метода класса **ClassLoader**, имеющих отношение к проблеме поиска и загрузки ресурсов — **getResource** и **getSystemResource**, — возвращают объекты класса **URL**.

Метод **getContents** объекта URL, возвращаемого методами загрузчика классов, позволяет получить объект, представляющий содержимое, соответствующее адресу URL.

Существуют также методы:

public Enumeration findResources(String name)

Возвращает объект Enumeration, который позволяет просматривать объекты URL для всех ресурсов, отвечающих заданному имени.

Реализация метода, предлагаемая по умолчанию классом ClassLoader, возвращает объект Enumeration с нулевым числом ресурсов.

public Java.net.URL findResource(String name)

Возвращает объект java.net.URL для ресурса с заданным именем либо null, если таковой не найден.

Если существует несколько ресурсов, отвечающих заданному имени, правила выбора одного из них определяются конкретной реализацией.

Реализация метода, предлагаемая по умолчанию классом Class Loader, возвращает значение null.

Аннотации в Java

Аннотации представляют собой некие метаданные, которые могут добавляться в исходный код программы и семантически не влияют на нее, но могут использоваться в процессе анализа кода, компиляции и даже во время выполнения.

Основные варианты использования аннотаций:

- предоставлять необходимую информацию для компилятора;
- предоставлять метаданные различным инструментам для генерации кода, конфигураций и т.д.;
- использоваться в коде во время выполнения програмного кода (reflection).

Аннотации могут быть применены, например, к декларациям классов, полей, методов, и других аннотаций.

Для описания новой аннотации используется ключевое слово **@interface**.

Пример аннотации:

```
public @interface Description {
    String title();
    int version() default 1;
    String text();
}
```

Пример использования аннотации имеет вид:

```
@Description(title="title", version=2,  text="text")
 public class Clazz { /* */ }
```

В качестве типов у элементов аннотации могут использоваться только примитивные типы, перечисления и класс String.

В случае, когда аннотация указывается для другой аннотации, первую называют мета-аннотацией (meta-annotation type) (Одна из наиболее распространенных мета аннотаций является **Retention**).

Она показывает, как долго необходимо хранить аннотацию и инициализируется одним из трех значений:

- **RetentionPolicy.SOURCE** - аннотация используется на этапе компиляции и должна отбрасываться компилятором;
- **RetentionPolicy.CLASS** - аннотация будет записана в class-файл компилятором, но не должна быть доступна во время выполнения (runtime);
- **RetentionPolicy.RUNTIME** - аннотация будет записана в class-файл и доступна во время выполнения через reflection.

По умолчанию у всех аннотаций стоит RetentionPolicy.CLASS.

Рассмотрим пример аннотации.

Предположим, необходимо ограничить доступ к некоторым функциям веб-приложения для разных пользователей.

Иными словами необходимо реализовать права (permissions).

Для этого можно добавить следующее перечисление в класс пользователя:

```
public class User {
    public static enum Permission {
        USER_MANAGEMENT,
        CONTENT_MANAGEMENT
    }
    private List permissions;
    public List getPermissions() {
        return new ArrayList(permissions);
    }
    ...
}
```

Создадим аннотацию, которую затем можно использовать для проверки прав:

```
@Retention(RetentionPolicy.RUNTIME)
  public @interface PermissionRequired {
      User.Permission value();
  }
```

Предположим имеется некоторое действие, право на выполнение которого нужно ограничить, например, UserDeleteAction.

То добавляем аннотацию на это действие следующим образом:

```
@PermissionRequired(
                  User.Permission.USER_MANAGEMENT)
  public class UserDeleteAction {
       public void invoke(User user) { …. }
  }
```

Теперь используя reflection можно принимать решение, разрешать или не разрешать выполнение определенного действия:

```
User user = ...;
   Class<?> actionClass = ...;
   PermissionRequired permissionRequired =
         actionClass.getAnnotation(PermissionRequired.class);
   if (permissionRequired != null)
     if (user != null && user.getPermissions().contains(
                                        permissionRequired.value()) ){
      // выполнить действие
     }
```

Рассмотрим другой пример:

```
import java.lang.annotation.*;
import java.lang.reflect.Method;

enum Condition {TRUE, FALSE }

@Retention(RetentionPolicy.RUNTIME)
@interface Cond {
      Condition value();
 }
@Cond(Condition.TRUE)
class Myclass {
    public void func() { System.out.println("Hello");}}

public class Main {
     public static void main(String[] args)throws Exception {
            Myclass m=new Myclass();
           Class<?> cl= Myclass.class;
           Cond cd=cl.getAnnotation(Cond.class);
           System.out.println(cd.value());  //На экране true
      }
}
```

Пример.

Аннотация у метода.

Аннотация RequestForCustomer сопровождает объявление метода

……..

```
@RequestForCustomer (
   level = 2,
   description = "Enable time",
   date = "10/10/2007"
)
public void customerThroughTime () {
  ...
}
```

```
import java.lang.reflect.Method;
public class Request {
@RequestForCustomer (level = 2, description = "Enabletime",
                                                    date = "10/10/2007“ )
public void customerThroughTime () {
  try {
     Class c = this.getClass();
     Method m = c.getMethod("customerThroughTime");
     RequestForCustomer ann =
                          m.getAnnotation(RequestForCustomer.class);
                                                    //запрос аннотаций
     System.out.println(ann.level() + " " + ann.description () + " "
                                                    + ann.date ());
  }
  catch (NoSuchMethodException e) {
            System.out.println ("метод не найден");
   }
}
```

Сериализация

Сериализация это процесс сохранения состояния объекта в последовательность байт; десериализация это процесс восстановления объекта, из этих байт.

Если двум компонентам Java необходимо общаться друг с другом, то им необходим механизм для обмена данными. Есть несколько способов реализовать этот механизм.

Первый способ это разработать собственный протокол и передать объект.

Это означает, что получатель должен знать протокол, используемый отправителем для воссоздания объекта, что усложняет разработку сторонних компонентов.

Необходим универсальный и эффективный протокол передачи объектов между компонентами.

Сериализация создана для этого, и компоненты Java используют этот протокол для передачи объектов.

Сериализация объектов в Java

Класс сериализуемого объекта должен реализовать интерфейс java.io.Serializable:

```
import java.io.Serializable;
class TestSerial implements Serializable {
    public byte version = 100;
    public byte count = 0;
}
```

Интерфейс Serializable это интерфейс-маркер; в нём не задекларировано ни одного метода.

Он говорит сериализующему механизму, что класс может быть сериализован.

Сериализация делается вызовом метода **writeObject()** класса **java.io.ObjectOutputStream**

Рассмотрим пример:

```
public static void main(String args[])
                                                    throws IOException {
    FileOutputStream fos =
                                new FileOutputStream("temp.out");
    ObjectOutputStream oos =
                                      new ObjectOutputStream(fos);
    TestSerial ts = new TestSerial();
     oos.writeObject(ts);
     oos.flush();
     oos.close();
}
```

Воссоздать объект из файла можно следующим образом:

```
public static void main(String args[]) throws
            IOException,ClassNotFoundException {
   FileInputStream fis =
                  new FileInputStream("temp.out");
   ObjectInputStream oin =
                        new ObjectInputStream(fis);
   TestSerial ts = (TestSerial) oin.readObject();
   System.out.println("version="+ts.version);
}
```

Восстановление объекта происходит с помощью вызова метода **oin.readObject().**

В методе происходит чтение набора байт из файла и создание точной копии графа оригинального объекта. oin.readObject() может прочитать любой сериализованный объект, поэтому необходимо полученный объект приводить к конкретному типу.

Подготовка классов к сериализации

По умолчанию процесс сериализации заключается в сериализации каждого поля объекта, которое не обозначено как **transient** или **static,** также данный процесс предполагает, что все поля-объекты, подлежащие сериализации, должны указывать на типы, в свою очередь поддерживающие возможность сериализации.

Кроме того, требуется, чтобы класс, базовый по отношению к рассматриваемому, либо обладал конструктором без параметров (дабы таковой мог быть вызван в процессе десериализации), либо сам в свою очередь обеспечивал реализацию интерфейса Serializable (в таком случае реализация того же интерфейса в производном классе будет избыточным решением).

Рассмотрим пример:

```
public class Name implements Serializable {
    private String name;
    private long id;
    private transient boolean hashSet = false;
    private transient int hash;
    private static long nextiD = 0;
  public Name(String name)   {
    this.name = name;
    synchronized(Name.class){ id = nextID++;}
  }
  public int hashCode() {
    if (hashset==false) {
       hash = name.hashCode() ;
       hashSet = true;
    }
    return hash;
   }
    ...
}
```

Объект класса Name может быть сохранен в потоке ObjectOutputStream и непосредственно, с помощью вызова метода writeObject, и косвенно, если этот объект адресуется из другого объекта, подлежащего сериализации.

Содержимое полей **name** и **id** выводится в поток; поля **nextiD**, **hashSet** и **hash**, в процесс сериализации вовлечены не будут, поскольку **nextiD** объявлено как **static**, а два других поля обозначены модификатором **transient**.

Поле **hash** содержит хеш-код, который может быть заново пересчитан на основании содержимого **name**, нет никаких причин расходовать на его сериализацию дополнительные вычислительные ресурсы и время.

Схема десериализации, предлагаемая по умолчанию, предусматривает считывание из потока байтовых данных, сохраненных в процессе сериализации.

Статические поля класса остаются в неприкосновенности — в ходе загрузки класса будут выполнены все обычные процедуры инициализации, и статические поля получат требуемые исходные значения.

Каждому полю **transient** в восстановленном объекте присваивается значение по умолчанию, соответствующее типу этого поля.

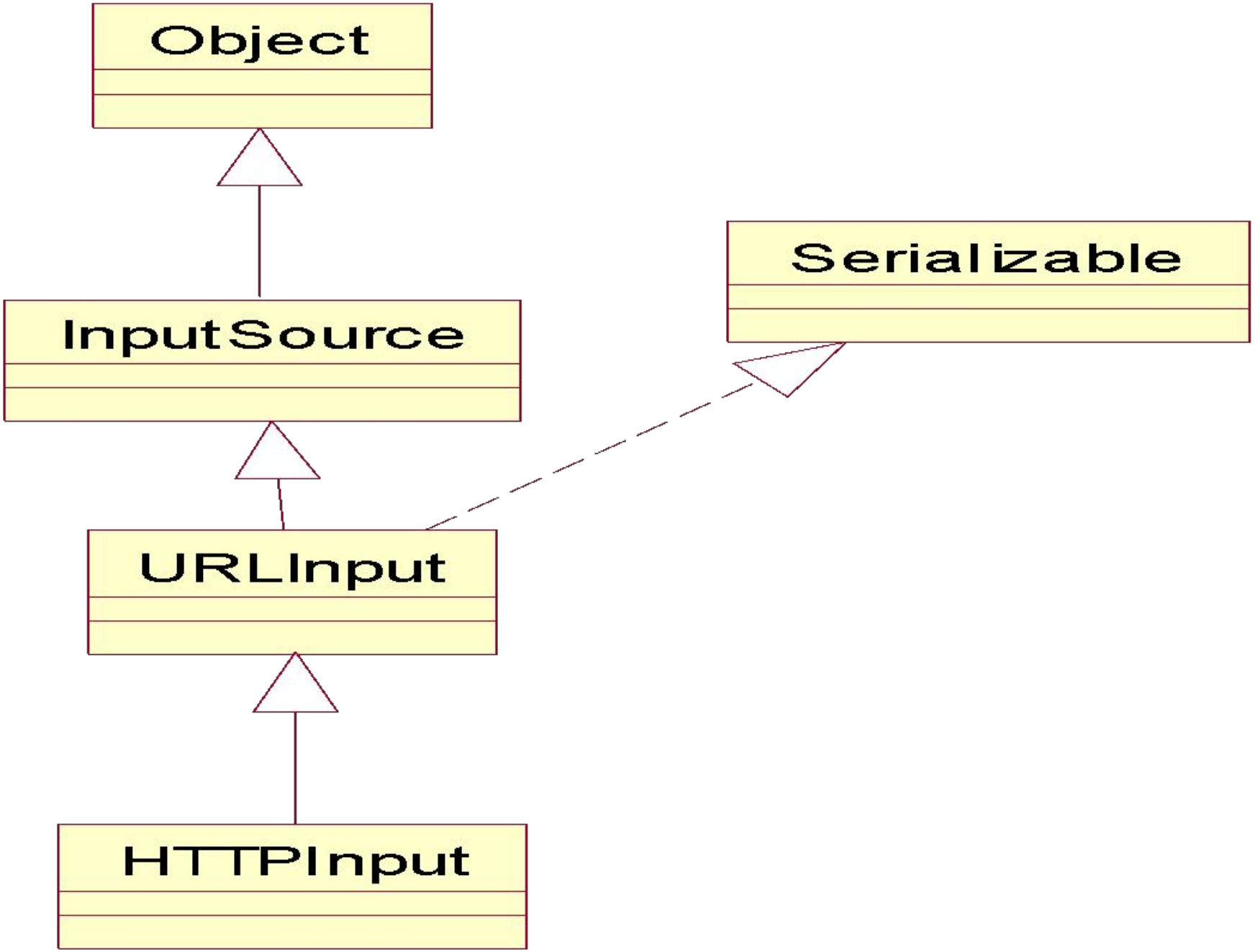
При десериализации объекта класса **Name** полям **name** и **id** вновь созданного объекта будут присвоены те же значения, что и в исходном объекте, содержимое статического поля **nextiD** останется нетронутым, а transient-поля hashSet и hash получат соответствующие значения по умолчанию (false и 0)— если значение hashSet равно false, величина hash будет пересчитана заново

Любая попытка осуществления сериализации объекта, не поддерживающего эту возможность, приводит к выбрасыванию исключения типа **NotSerializableException**.

Порядок сериализации и десериализации

Сериализация объектов выполняется в нисходящем порядке по древовидной иерархии типов— от первого из классов, поддерживающих интерфейс Serializable, до классов более частных типов.

Рассмотрим следующую иерархию типов


Object
Serializable
InputSource
URLInput
HTTPInput

Рассмотрим порядок десериализации для объекта HTTPInput.

1. Поток ObjectInputStream вначале выделяет память для нового объекта, а затем находит в иерархии типов ближайший из классов, поддерживающий интерфейс Serializable, т.е. URLInput.
2. Поток вызывает конструктор без параметров для класса, базового по отношению к найденному (ближайшего из числа тех, которые не поддерживают интерфейс Serializable), т.е. InputSource.

Примечание. Если должно быть сохранено иное состояние части объекта, относящейся к базовому классу (InputSource), ответственность за сериализацию и десериализацию этого состояния возлагается на URLInput.

Если же базовый класс, не обеспечивающий сериализацию, обладает собственным значимым состоянием, то придется отступить от схемы сериализации, предлагаемой по умолчанию, и осуществить дополнительную настройку ближайшего из классов, поддерживающих механизм сериализации.

3. После того, как ближайший из классов, поддерживающих механизм сериализации, завершает восстановление данных того подмножества полей, которые относятся к базовому классу, он приступает к воссозданию своего собственного состояния, считывая данные из потока.
4. Поток ObjectInputStream осуществляет просмотр иерархии объектов в нисходящем порядке и выполняет десериализацию каждого объекта посредством вызовов readObject.

Если в ходе десериализации объекта будут найдены ссылки на другие ранее сериализованные объекты.

Эти объекты подвергаются десериализации в том порядке, в каком они обнаруживаются.

Так, если бы объект URLInput, скажем, обладал ссылкой на объект HashMap, адресуемая хеш-таблица вместе со всем ее содержимым была бы подвергнута десериализации раньше, чем оставшаяся часть объекта URLInput как такового.

Прежде чем будут выполнены любые из рассмотренных операций десериализации, соответствующие классы должны быть загружены, в противном случае метод readObject выбросить исключение **ClassNotFoundException**.

Настройка механизма сериализации

Для некоторых классов обычные средства десериализации могут оказаться неадекватными или неэффективными.

В этом случае необходимо переопределить методы **readObject** и **writeObjec**t.

Подобные переопределенные варианты методов **writeObject** и **readObject** вызываются только в контексте объектов соответствующих классов, и методы несут ответственность исключительно за состояние объекта класса как такового, в том числе и в части, унаследованной от базового класса, не поддерживающего механизм сериализации.

Если в составе класса реализованы собственные варианты методов writeObject и readObject, они не должны обращаться к одноименным методам базового класса.

В качестве примера рассмотрим улучшенный вариант класса Name

```
public class BetterName implements Serializable {
   private String name;
   private long id;
   private transient int hash;
   private static long nextID = 0;
 public BetterName(String name)  {
      this.name = name;
      synchronized  (BetterName.class)  {
            id = nextID++;
      }
      hash = name.hashCode();
 }
```

```
  private void writeObject(ObjectOutputStream out)
                                          throws IOException {
    out.writeUTF(name);
    out.writeLong(id);
  }
  private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException{
    name = in.readUTF();
    id = in. readLong() ;
    hash = name.hashCode();
  }
  .....
}
```

- Существует ограничение: приступая к настройке механизма сериализации, нельзя присваивать значение полю final в теле метода readObject, поскольку поля final могут быть проинициализированы только в блоках инициализации или конструкторах.
- Переопределенные методы readObject и writeObject обязательно должны быть **private.**

Контроль версий объектов

Если реализация классов может быть изменена в промежутке времени между сериализацией и десериализацией объекта класса, то поток ObjectInputStream способен обнаружить факт внесения изменений.

При сохранении объекта вместе с ним записывается уникальный идентификатор номера версии (serial version unique identifier, UID), 64-битовое значение типа long.

По умолчанию идентификатор создается в виде хеш-кода, построенного на основе информации об именах класса, его членов и базовых интерфейсов; изменение таких данных служит сигналом о возможной несовместимости версий класса.

При вводе данных об объекте из потока

ObjectInputStream считывается также и идентификатор номера версии.

Затем предпринимается попытка загрузки соответствующего класса.

Если требуемый класс не найден или идентификатор загруженного класса не совпадает с тем, который считан из потока, метод readObject выбрасывает исключение типа **InvalidClassException**.

Если успешно загружены все нужные классы и выявлено совпадение всех идентификаторов, объект может быть подвергнут десериализации.

Существует возможность сохранить совместимость класса, снабженного дополнительными несущественными нововведениями, с данными ранее сериализованных объектов.

Для этого следует принудительно объявить в составе класса специальное поле, содержащее значение идентификатора номера версии, например:

```
private static final long
        serialversionUID = 1307795172754062330L;
```

Рассмотрим пример.

Пусть изначально имеется класс

```
public class BetterName implements Serializable {
    private String name;
    private long id;
    private transient int hash;
    private static long nextID = 1;

   public BetterName(String name){
        this.name = name;
        synchronized  (BetterName.class)  {
            id = nextID++;
        }
        hash = name.hashCode(); }
}
```

Используя следущий код сереализируем данный класс

```
public static void main(String args[])throws IOException{
   FileOutputStream fos = new FileOutputStream("temp.out");
   ObjectOutputStream oos = new ObjectOutputStream(fos);
    BetterName ts = new BetterName (“aaaa”);
    oos.writeObject(ts);
    oos.flush();
    oos.close();
}
```

Предположим, что перед десериализацией код класса BetterName был изменен

```
public class BetterName implements Serializable {
    private String name;
    private long id;
    private transient int hash;
    private static long nextID = 1;
    private int a=20;

   public BetterName(String name)  {
       this.name = name;

       this.a=20;

        synchronized(BetterName.class){ id = nextID++;}
        hash = name.hashCode();
   }
}
```

Если данный класс десериализировать с помощью кода

```
public static void main(String args[])
                    throws IOException,ClassNotFoundException{
  FileInputStream fis = new FileInputStream("temp.out");
  ObjectInputStream oin = new  ObjectInputStream(fis);
  BetterName ts = (BetterName) oin.readObject();
}
```

то будет выброшено исключение, которое связано с тем, что не совпадает **serialversionUID** сериализированного объекта и новой версии класса BetterName.

Чтобы избежать выброса исключения необходимо добавить в код класса BetterName поле serialversionUID

```
public class BetterName implements Serializable {
    private String name;
    private long id;
    private transient int hash;
    private static long nextID = 1;
    private int a=20;
    static final long serialVersionUID =
                                        4090868775906377719L;
    ……….
}
```

Получить значение поля serialVersionUID можно с помощью утилиты *serialver,* она находится там же, где и javac:

>> *serialver BetterName*

Эту утилиту нужно запустить перед изменением класса BetterName.

Интерфейс Externalizable

Интерфейс Externalizable является производным от Serializable.

Класс, реализующий интерфейс Externalizable, приобретает все полномочия по управлению состоянием объекта, подлежащего сериализации, но при этом несет и сугубую ответственность за корректную обработку данных, относящихся к базовым классам, обеспечение контроля версий и т.д.

Подобная возможность необходима, например, в условиях, когда при реализации хранилища данных об объектах следует учесть некие особые ограничения, касающиеся формы представления объектов и не совместимые с существующими механизмами сериализации.

В составе интерфейса Externalizable определены два метода:

```
public interface Externalizable extends Serializable {
   void writeExternal(ObjectOutput out) throws IOException;
   void  readExternal(ObjectInput in)
                    throws IOException, ClassNotFoundException;
}
```

Методы вызываются в процессе сериализации и десериализации объекта соответственно. Это обычные методы public, и выбор требуемой реализации каждого из них определяется типом текущего объекта.

Классам, производным от класса, реализующего интерфейс Externalizable, часто необходимо вызвать соответствующий метод базового класса прежде, чем будет осуществлена сериализация или десериализация их собственного состояния, — в отличие от классов, реализующих схему сериализации, принятую по умолчанию.

Рассмотрим пример:

```
import java.io.Externalizable;
import java.io.ObjectOutput;
import java.io.IOException;
import java.io.ObjectInput;
public class ItemExt implements Externalizable{

    private int fieldInt;
    private boolean fieldBoolean;
    private long fieldLong;
    private float fieldFloat;
    private double fieldDouble;
    private String fieldString;

    public ItemExt(){
        this(0,true,0,0,0,"");
    }
```

```
public ItemExt(int fieldInt,
               boolean fieldBoolean,
               long fieldLong,
               float fieldFloat,
               double fieldDouble,
               String fieldString) {

    this.fieldInt = fieldInt;
    this.fieldBoolean = fieldBoolean;
    this.fieldLong = fieldLong;
    this.fieldFloat = fieldFloat;
    this.fieldDouble = fieldDouble;
    this.fieldString = fieldString;
}
```

```
public int getFieldInt() {  return fieldInt;  }

public boolean getFieldBoolean() {
      return fieldBoolean;
}
public long getFieldLong() { return fieldLong;}

public float getFieldFloat() {return fieldFloat; }

public double getFieldDouble() {
      return fieldDouble;
}
public String getFieldString() {
     return fieldString;
}
```

```
public void writeExternal(ObjectOutput out)
                                    throws IOException{
    out.writeInt(fieldInt);
    out.writeBoolean(fieldBoolean);
    out.writeLong(fieldLong);
    out.writeFloat(fieldFloat);
    out.writeDouble(fieldDouble);
    out.writeUTF(fieldString);
}
```

```
public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException{

    fieldInt = in.readInt();
    fieldBoolean = in.readBoolean();
    fieldLong = in.readLong();
    fieldFloat = in.readFloat();
    fieldDouble = in.readDouble();
    fieldString = in.readUTF();
  }
}
```

Тогда сериализация класса ItemExt будет иметь вид:

```
public static void main(String[] args)throws IOException{

   FileOutputStream fos = new FileOutputStream("temp.out");
   ObjectOutputStream oos = new ObjectOutputStream(fos);
   ItemExt ts = new ItemExt();
   oos.writeObject(ts);
   oos.flush();
   oos.close();
}
```

Соответственно десериализация имеет вид:

```
public static void main(String[] args)
            throws IOException,ClassNotFoundException {

  FileInputStream fis = new FileInputStream("temp.out");
  ObjectInputStream oin = new ObjectInputStream(fis);
  ItemExt ts = (ItemExt) oin.readObject();
  System.out.println(ts.getFieldInt());
}
```