Оглавление

[Что такое сериализация, и как она реализована в Java?](#_Toc47424297)

[Для чего нужна сериализация?](#_Toc47424298)

[Процесс сериализации/десериализации с использованием Serializable.](#_Toc47424299)

[Как изменить стандартное поведение сериализации/десериализации?](#_Toc47424300)

[Какие поля не будут сериализованы при сериализации (static и final)?](#_Toc47424301)

[Как не допустить сериализацию?](#_Toc47424302)

[Как создать собственный протокол сериализации?](#_Toc47424303)

[Какая роль поля serialVersionUID в сериализации?](#_Toc47424304)

[Когда стоит изменять значение поля serialVersionUID?](#_Toc47424305)

[В чем проблема сериализации Singleton?](#_Toc47424306)

[Расскажите про клонирование объектов](#_Toc47424307)

[В чем отличие между поверхностным и глубоким клонированием?](#_Toc47424308)

[Какой способ клонирования предпочтительней?](#_Toc47424309)

[Почему метод clone() объявлен в классе Object, а не в интерфейсе Cloneable?](#_Toc47424310)

[Как создать глубокую копию объекта? (2 способа)](#_Toc47424311)

# Что такое сериализация, и как она реализована в Java?

Сериализация — процесс преобразования структуры данных в линейную последовательность байтов для дальнейшей передачи или сохранения.

Десериализация — обратный процесс.

Стандартные способы сериализации (согласно спецификации Java Object Serialization):  
∙ стандартная сериализация (java.io.Serializable);  
∙ «расширенная» сериализация (java.io.Externalizable).

# Для чего нужна сериализация?

Сериализация позволяет в определенных пределах изменять класс. Вот наиболее важные изменения, с которыми спецификация Java Object Serialization может справляться автоматически:  
∙ добавление в класс новых полей;  
∙ изменение полей из статических в нестатические;  
∙ изменение полей из транзитных в нетранзитные.

Обратные изменения (из нестатических полей в статические и из нетранзитных в транзитные) или удаление полей требуют определенной дополнительной обработки в зависимости от того, какая степень обратной совместимости необходима.

# Процесс сериализации/десериализации с использованием Serializable.

При использовании Serializable применяется алгоритм сериализации, который с помощью рефлексии (Reflection API) выполняет:  
∙ запись в поток метаданных о классе, ассоциированном с объектом (имя класса, идентификатор SerialVersionUID, идентификаторы полей класса);  
∙ рекурсивную запись в поток описания суперклассов до класса Object (не включительно);  
∙ запись примитивных значений полей сериализуемого экземпляра, начиная с полей самого верхнего суперкласса;  
∙ рекурсивную запись объектов, которые являются полями сериализуемого объекта.

При этом ранее сериализованные объекты повторно не сериализуются, что позволяет алгоритму корректно работать с циклическими ссылками.

Для выполнения десериализации под объект выделяется память, после чего его поля заполняются значениями из потока. Конструктор объекта при этом не вызывается. Однако при десериализации будет вызван конструктор без параметров родительского несериализуемого класса, а его отсутствие повлечет ошибку десериализации.

# Как изменить стандартное поведение сериализации/десериализации?

Реализовать интерфейс java.io.Externalizable, который позволяет применение пользовательской логики сериализации. Способ сериализации и десериализации описывается в методах writeExternal() и readExternal(). Во время десериализации вызывается конструктор без параметров, а потом уже на созданном объекте вызывается метод readExternal.

Если у сериализуемого объекта реализован один из следующих методов, то механизм сериализации будет использовать его, а не метод по умолчанию:  
∙ writeObject() — запись объекта в поток;  
∙ readObject() — чтение объекта из потока;  
∙ writeReplace() — позволяет заменить себя экземпляром другого класса перед записью;  
∙ readResolve() — позволяет заменить на себя другой объект после чтения.

# Какие поля не будут сериализованы при сериализации (static и final)?

При помощи ключевого слова transient можно исключить поля из общего процесса сериализации.

Обычно в таких полях хранится промежуточное состояние объекта, которое, к примеру, проще вычислить. Другой пример такого поля — ссылка на экземпляр объекта, который не требует сериализации или не может быть сериализован.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Serializable | | Externalizable | |
|  | сериал. | десериал. | сериал. | десериал. |
| static | Нет | — | трудноуловимые ошибки | |
| final | да | | да | нет |

final и Externalizable: final поля должны быть инициализированы в конструкторе, а после этого в readExternal() изменить значение этого поля будет невозможно. Соответственно, если необходимо сериализовать объект с final полем необходимо использовать только стандартную сериализацию.

# Как не допустить сериализацию?

Чтобы не допустить автоматическую сериализацию можно переопределить private методы для создания исключительной ситуации NotSerializableException.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | private void writeObject(ObjectOutputStream out)  throws IOException {  throw new NotSerializableException();  }  private void readObject(ObjectInputStream in)  throws IOException {  throw new NotSerializableException();  } |

Любая попытка записать или прочитать этот объект теперь приведет к возникновению исключительной ситуации.

# Как создать собственный протокол сериализации?

Для создания собственного протокола сериализации достаточно реализовать интерфейс Externalizable, который содержит два метода:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | public void writeExternal(ObjectOutput out)  throws IOException;  public void readExternal(ObjectInput in)  throws IOException,  ClassNotFoundException; |

# Какая роль поля serialVersionUID в сериализации?

serialVersionUID используется для указании версии сериализованных данных.

Когда мы не объявляем serialVersionUID в нашем классе явно, среда выполнения Java делает это за нас, но этот процесс чувствителен ко многим метаданным класса включая количество полей, тип полей, модификаторы доступа полей, интерфейсов, которые реализованы в классе и пр.

Рекомендуется явно объявлять serialVersionUID, т. к. при добавлении, удалении атрибутов класса динамически сгенерированное значение может измениться и в момент выполнения будет выброшено исключение InvalidClassException.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | private static final long serialVersionUID =  20161013L; |

# Когда стоит изменять значение поля serialVersionUID?

serialVersionUID нужно изменять при внесении в класс несовместимых изменений, например при удалении какого-либо его атрибута.

# В чем проблема сериализации Singleton?

Проблема в том, что после десериализации мы получим другой объект. Т. о., сериализация дает возможность создать Singleton еще раз, что недопустимо. Существует два способа избежать этого:  
∙ явный запрет сериализации;  
∙ определение метода с сигнатурой (default/public/private/protected/) Object readResolve() throws ObjectStreamException, назначением которого станет возврат замещающего объекта вместо объекта, на котором он вызван.

# Расскажите про клонирование объектов

Использование оператора присваивания не создает нового объекта, а лишь копирует ссылку на объект. Т. о., две ссылки указывают на одну и ту же область памяти, на один и тот же объект. Для создания нового объекта с таким же состоянием используется клонирование объекта.

Класс Object содержит protected метод clone(), осуществляющий побитовое копирование объекта производного класса. Однако сначала необходимо переопределить метод clone() как public для обеспечения возможности его вызова. В переопределенном методе следует вызвать базовую версию метода super.clone(), которая и выполняет собственно клонирование.

Чтобы окончательно сделать объект клонируемым, класс должен реализовать интерфейс Cloneable. Интерфейс Cloneable не содержит методов относится к маркерным интерфейсам, а его реализация гарантирует, что метод clone() класса Object возвратит точную копию вызвавшего его объекта с воспроизведением значений всех его полей. В противном случае метод генерирует исключение CloneNotSupportedException. Следует отметить, что при использовании этого механизма объект создается без вызова конструктора.

Это решение эффективно только в случае, если поля клонируемого объекта представляют собой значения базовых типов и их оберток или неизменяемых объектных типов. Если же поле клонируемого типа является изменяемым ссылочным типом, то для корректного клонирования требуется другой подход. Причина заключается в том, что при создании копии поля оригинал и копия представляют собой ссылку на один и тот же объект. В этой ситуации следует также клонировать и сам объект поля класса.

Такое клонирование возможно только в случае, если тип атрибута класса также реализует интерфейс Cloneable и переопределяет метод clone(). Т. к., если это будет иначе вызов метода невозможен из‑за его недоступности. Отсюда следует, что если класс имеет суперкласс, то для реализации механизма клонирования текущего класса-потомка необходимо наличие корректной реализации такого механизма в суперклассе. При этом следует отказаться от использования объявлений final для полей объектных типов по причине невозможности изменения их значений при реализации клонирования.

Помимо встроенного механизма клонирования в Java для клонирования объекта можно использовать:

∙ Специализированный конструктор копирования — в классе описывается конструктор, который принимает объект этого же класса и инициализирует поля создаваемого объекта значениями полей переданного.

∙ Фабричный метод — представляет собой статический метод, возвращающий экземпляр своего класса.

∙ Механизм сериализации — сохранение и последующее восстановление объекта в/из потока байтов.

# В чем отличие между поверхностным и глубоким клонированием?

Поверхностное копирование копирует настолько малую часть информации об объекте, насколько это возможно. По умолчанию, клонирование в Java является поверхностным, т. е. класс Object не знает о структуре класса, которого он копирует. Клонирование такого типа осуществляется JVM по следующим правилам:

∙ Если класс имеет только члены примитивных типов, то будет создана совершенно новая копия объекта и возвращена ссылка на этот объект.

∙ Если класс помимо членов примитивных типов содержит члены ссылочных типов, то тогда копируются ссылки на объекты этих классов. Следовательно, оба объекта будут иметь одинаковые ссылки.

Глубокое копирование дублирует абсолютно всю информацию объекта:

∙ Нет необходимости копировать отдельно примитивные данные;

∙ Все члены ссылочного типа в оригинальном классе должны поддерживать клонирование. Для каждого такого члена при переопределении метода clone() должен вызываться super.clone();

∙ Если какой-либо член класса не поддерживает клонирование, то в методе клонирования необходимо создать новый экземпляр этого класса и скопировать каждый его член со всеми атрибутами в новый объект класса, по одному.

# Какой способ клонирования предпочтительней?

Наиболее безопасным и, следовательно, предпочтительным способом клонирования является использование специализированного конструктора копирования:

∙ Отсутствие ошибок наследования (не нужно беспокоиться, что у наследников появятся новые поля, которые не будут склонированы через метод clone());

∙ Поля для клонирования указываются явно;

∙ Возможность клонировать даже final поля.

# Почему метод clone() объявлен в классе Object, а не в интерфейсе Cloneable?

Метод clone() объявлен в классе Object с указанием модификатора native, чтобы обеспечить доступ к стандартному механизму поверхностного копирования объектов. Одновременно он объявлен и как protected, чтобы нельзя было вызвать этот метод у не переопределивших его объектов. Непосредственно интерфейс Cloneable является маркерным (не содержит объявлений методов) и нужен только для обозначения самого факта, что данный объект готов к тому, чтобы быть клонированным. Вызов переопределенного метода clone() у не Cloneable объекта вызовет выбрасывание CloneNotSupportedException.

# Как создать глубокую копию объекта? (2 способа)

Все поля реализовавывали интерфейс Cloneable.

Какие существуют способы контроля за значениями десериализованного объекта

Если есть необходимость выполнения контроля за значениями десериализованного объекта, то можно использовать интерфейс ObjectInputValidation с переопределением метода validateObject().

// Если вызвать метод validateObject() после десериализации объекта, то будет вызвано исключение InvalidObjectException при значении возраста за пределами 39...60.

public class Person implements java.io.Serializable,

java.io.ObjectInputValidation {

...

@Override

public void validateObject() throws InvalidObjectException {

if ((age < 39) || (age > 60))

throw new InvalidObjectException("Invalid age");

}

}

Так же существуют способы подписывания и шифрования, позволяющие убедиться, что данные не были изменены:

с помощью описания логики в writeObject() и readObject().

поместить в оберточный класс javax.crypto.SealedObject и/или java.security.SignedObject. Данные классы являются сериализуемыми, поэтому при оборачивании объекта в SealedObject создается подобие «подарочной упаковки» вокруг исходного объекта. Для шифрования необходимо создать симметричный ключ, управление которым должно осуществляться отдельно. Аналогично, для проверки данных можно использовать класс SignedObject, для работы с которым также нужен симметричный ключ, управляемый отдельно.