Механизмы ввода и вывода

NetCracker®

- Потоки данных
- Виды потоков и базовые классы
- Разновидности потоков
- Сериализация

- Система ввода/вывода не должна зависеть от платформы!

- Применяется модель потоков данных:
  - упорядоченная последовательность данных,
  - которой соответствует определенный источник (потоки ввода) или получатель (потоки вывода)

- Типы общего назначения
- Классы разновидностей потоков
- Специализированные классы и интерфейсы для ввода и вывода значений простых типов
- Классы и интерфейсы работы с файлами
- Классы и интерфейсы механизма сериализации

- `abstract int read()`
  `throws IOException`
- `int read(byte[] b, int off, int len)`
  `throws IOException`
- `int read(byte[] b)`
  `throws IOException`
- `long skip(long n)`
  `throws IOException`
- `int available()`
  `throws IOException`
- `void close()`
  `throws IOException`

- `abstract void write(int b)`
  `throws IOException`
- `void write(byte[] b, int off, int len)`
  `throws IOException`
- `void write(byte[] b)`
  `throws IOException`
- `void flush()`
  `throws IOException`
- `void close()`
  `throws IOException`

- `int read()`
  `throws IOException`
- `abstract int read(char[] b, int off, int len)`
  `throws IOException`
- `int read(char[] b)`
  `throws IOException`
- `long skip(long n)`
  `throws IOException`
- `boolean ready()`
  `throws IOException`
- `abstract void close()`
  `throws IOException`

- `void write(int ch) throws IOException`
- `abstract void write(char[] b, int off, int len) throws IOException`
- `void write(char[] b) throws IOException`
- `void write(String str, int off, int len) throws IOException`
- `void write(String str) throws IOException`
- `abstract void flush() throws IOException`
- `abstract void close() throws IOException`

- Уже знакомые потоки:

  `System.out`

  `System.in`

  `System.err`

- Какого они типа?

  `PrintStream`

  `InputStream`

  `PrintStream`

- Байтового!!! (для совместимости с версиями Java 1.0 и 1.1)

- Образуют 4 иерархии, в основе которых лежат базовые абстрактные классы
- Имя любого дочернего класса в иерархии имеет суффикс, совпадающий с именем корневого класса
- По сути делятся на 2 вида:
  - «Реальные» потоки: источник (получатель) данных реален
  - Потоки-обертки: источником (получателем) данных является другой поток

- Позволяют читать из байтового как из символьного и записывать в байтовый поток как в символьный (с учетом кодировки)
- **`InputStreamReader`**
  - **`InputStreamReader(InputStream in)`**
  - **`InputStreamReader(InputStream in, String encoding) throws UnsupportedEncodingException`**
- **`OutputStreamWriter`**
  - **`OutputStreamWriter(OutputStream out)`**
  - **`OutputStreamWriter(OutputStream out, String encoding) throws UnsupportedEncodingException`**

`FilterInputStream`, `FilterReader`
`FilterOutputStream`, `FiltrerWriter`

- Обертки, позволяют объединять потоки в цепочки для получения сложных потоков, обладающих расширенным набором функций
- Обладают дополнительными защищенными конструкторами
  `protected FilterInputStream(InputStream in)`
- В наследниках обычно переопределяются методы чтения/записи с добавлением новой функциональности

`BufferedInputStream`, `BufferedReader`
`BufferedOutputStream`, `BufferedWriter`

- Обертки, осуществляют буферизацию данных на программном уровне
- Размер буфера можно задать в конструкторе
- Символьные версии имеют методы чтения и записи строк

`PipedInputStream`, `PipedReader`
`PipedOutputStream`, `PipedWriter`

- Используются в виде пар ввода-вывода
- Данные, переданные в поток вывода, служат источником для потока ввода
- Например, реализуют механизм обмена данными между нитями
- Поток-пара задается параметром конструктора либо с помощью метода `connect()`

`ByteArrayInputStream`, `ByteArrayOutputStream`

- В качестве источника и получателя данных используются массивы байт
- В потоке вывода размер буфера может меняться динамически
- В потоке вывода существуют методы преобразования:
  - к массиву байт
    `byte[] toByteArray()`
  - к строке
    `String toString()`
  - вывода в другой поток
    `void writeTo(OutputStream out)`

- **`CharArrayReader`** и **`CharArrayWriter`** аналогичны **`ByteArrayInputStream`** и **`ByteArrayOutputStream`**, но оперируют с массивом символов

- **`StringReader`** и **`StringWriter`** имеют аналогичную функциональность, позволяют считывать символы из строки и записывать данные в строковый буфер

- Обертки **`PrintStream`** и **`PrintWriter`** содержат методы, упрощающие задачу вывода данных простых типов в текстовом виде
- Методы **`print()`** и **`println()`** не выбрасывают исключений
- **`System.out`** и **`System.err`** – единственные потоки **`PrintStream`**

- Не является потоком чтения, но позволяет обрабатывать информацию из них
- Содержит методы лексической обработки текста
- Ряд методов предназначен для настройки работы анализатора
- Метод **`nextToken()`** производит обработку очередной лексемы, после чего:
  - Поле **`ttype`** содержит константу типа лексемы
  - Поля **`nval`** и **`sval`** содержат числовое и строковое представление лексемы

- Интерфейсы **`DataInput`** и **`DataOutput`** содержат объявления методов ввода и вывода значений простых типов
  `void writeLong(long v)`, **`void writeFloat(float v)`**
  `boolean readBoolean()`, **`String readUTF()`**

- Обертки **`DataInputStream`** и **`DataOutputStream`**, соответственно, реализуют эти интерфейсы

- Класс **`RandomAccessFile`** реализует оба интерфейса Data и позволяет работать с файлами в режиме произвольного доступа

- Инкапсулирует платформенно-независимые методы работы с файлами и директориями:
  - создание
  - проверка атрибутов
  - удаление
  - переименование
- Позволяет создавать временные файлы, удаляемые при завершении работы программы
- API класса изучите самостоятельно

`FileInputStream`, `FileReader`
`FileOutputStream`, `FileWriter`

- Позволяют трактовать файл как поток, предназначенный для ввода и вывода данных
- Связаны с исключениями `FileNotFoundException` и `SecurityException`
- Конструкторы могут получать параметры:
  - Строку `String`, задающую имя файла
  - Объект класса `File`
  - Объект `FileDescriptor` (возвращается методом `getFD()` байтовых потоков)

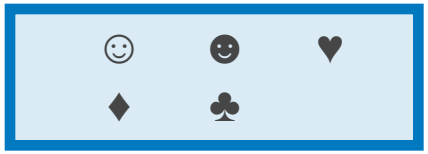
```
import java.io.*;

public class TextWrite {
  public static void main(String[] args) {
    int[] values = {1, 2, 3, 4, 5};
    try {
      PrintWriter out = new PrintWriter(new
        BufferedWriter(new FileWriter("out.txt")));
      for (int i = 0; i < values.length; i++) {
        out.println(values[i]);
      }
      out.close();
    }
    catch(IOException e) {
      System.out.println("Some error occurred!");
    }
  }
}
```

```
1
2
3
4
5
```

```
31 0D 0A
32 0D 0A
33 0D 0A
34 0D 0A
35 0D 0A
```

```
import java.io.*;

public class TextRead {
  public static void main(String[] args) {
    int[] values = new int[5];
    try {
      BufferedReader in = new BufferedReader(new
        FileReader("in.txt")); //InputStreamReader(System.in));
      for (int i = 0; i < values.length; i++) {
        values[i] = Integer.parseInt(in.readLine());
      }
      in.close();
    }
    catch(IOException e) {
      System.out.println("Some error occurred!");
    }
  }
}
```

```
import java.io.*;

public class ByteWrite {
  public static void main(String[] args) {
    int[] values = {1, 2, 3, 4, 5};
    try {
      DataOutputStream out = new
DataOutputStream(new FileOutputStream("out.bin"));
      for (int i = 0; i < values.length; i++) {
        out.writeInt(values[i]);
      }
      out.close();
    }
    catch(IOException e) {
      System.out.println("Some error occurred!");
    }
  }
}
```



```
00 00 00 01
00 00 00 02
00 00 00 03
00 00 00 04
00 00 00 05
```

```
import java.io.*;

public class ByteRead {
  public static void main(String[] args) {
    int[] values = new int[5];
    try {
      DataInputStream in = new DataInputStream(new
        FileInputStream("out.bin"));
      for (int i = 0; i < values.length; i++) {
        values[i] = in.readInt();
      }
      in.close();
    }
    catch(IOException e) {
      System.out.println("Some error occurred!");
    }
  }
}
```

- **Сериализация** – процесс преобразования состояния объекта в поток байтов
- **Десериализация** – восстановление состояния объекта из данных потока
- Не все объекты могут быть сериализованы
- Класс должен быть подготовлен к сериализации

- Класс **`ObjectOutputStream`** реализует сериализацию
- Класс **`ObjectInputStream`** реализует десериализацию
- Классы позволяют выводить и вводить графы объектов с сохранением структуры
- Результатом десериализации является объект, равнозначный исходному

```
import java.io.*;

public class SerializationWrite {
  public static void main(String[] args) {
    int[] values = {1, 2, 3, 4, 5};
    try {
      ObjectOutputStream out = new
        ObjectOutputStream(new
        FileOutputStream("out.bin"));
      out.writeObject(values);
      out.close();
    }
    catch(IOException e) {
      System.out.println("Some error occurred!");
    }
  }
}
```

```
┐H♣ur☻
[IMє`&vĸ__☻  xp
♣   ☺   ☻   ♥
♦   ♣
```

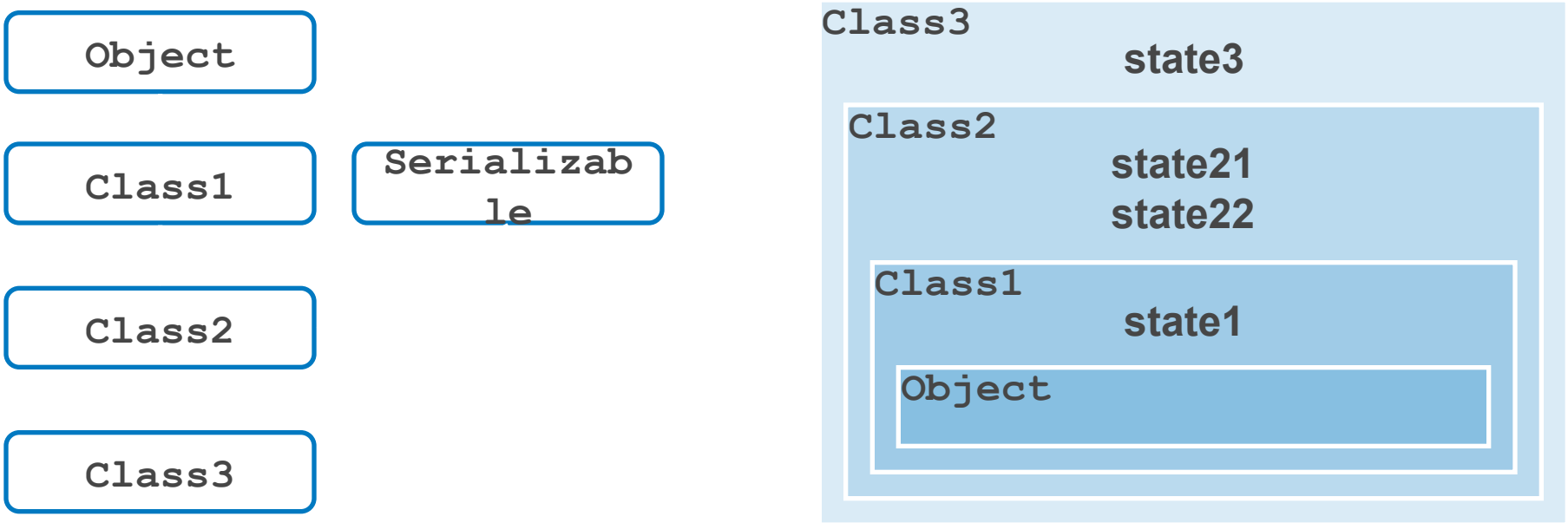
```
AC ED 00 05 75 72
00 02 5B 49 4D BA
60 26 76 EA B2 A5
02 00 00 78 70 00
00 00 05 00 00 00
01 00 00 00 02 00
00 00 03 00 00 00
04 00 00 00 05
```

```
import java.io.*;
public class SerializationRead {
  public static void main(String[] args) {
    int[] values;
    try {
      ObjectInputStream in = new ObjectInputStream(new
        FileInputStream("out.bin"));
      values = (int[])in.readObject();
      in.close();
    }
    catch(IOException e) {
      System.out.println("Some error occurred!");
    }
    catch(ClassNotFoundException e) {
      System.out.println("Wrong object type");
    }
  }
}
```

- Должен реализовываться интерфейс-маркер `java.io.Serializable`
- Все сериализуемые поля должны иметь сериализуемый тип
- Родительский класс должен иметь конструктор по умолчанию (без параметров) или быть подготовленным к сериализации
- Сериализуются поля объекта, не обозначенные как `transient` или `static`

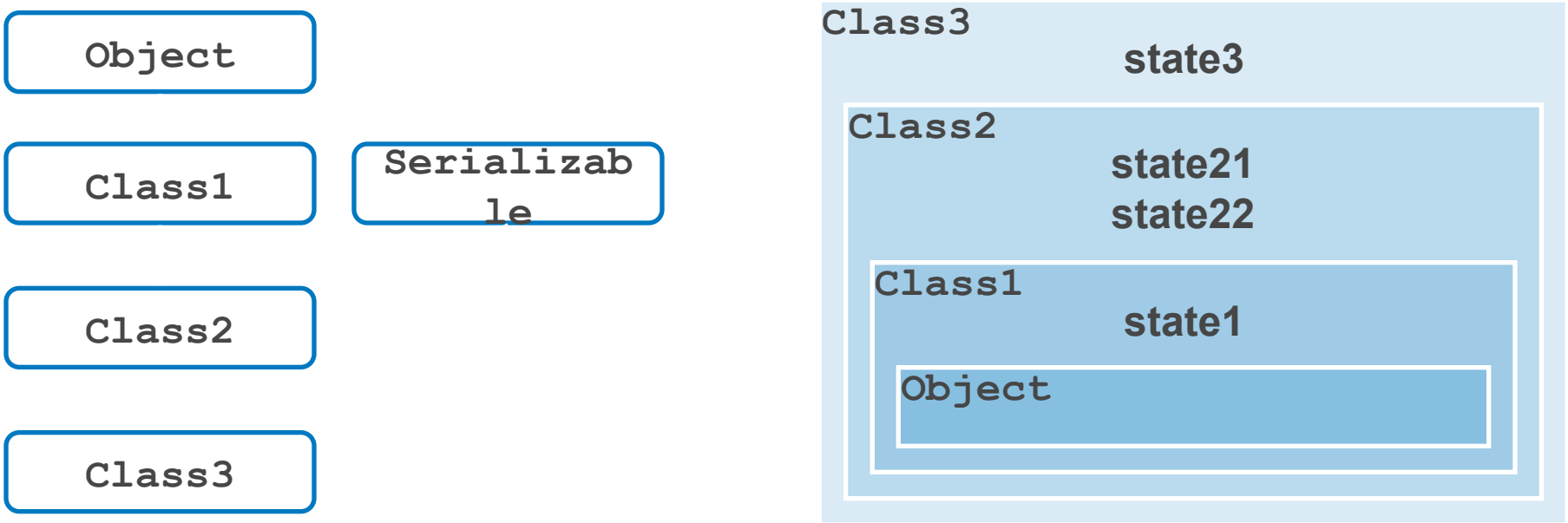
- В нисходящем порядке по древовидной иерархии типов: от первого сериализуемого класса до частного типа
- Объекты, на которые ссылаются поля, сериализуются в порядке обнаружения
- Перед десериализацией выполняется загрузка участвующих классов (возможен выброс исключения `ClassNotFoundException`)

```
class Class1 extends Object {
  private int state1 = 1;
}

class Class2 extends Class1 implements java.io.Serializable {
  protected int state21;
  private int state22;

  public Class2(int s1, int s2) {
    state21 = s1 + 15;
    state22 = s2 - 1;
  }
}

class Class3 extends Class2 {
  public int state 3 = 3;
}
```



Сериализованное состояние объекта класса Class3

Class3: (Class2) state21 state22 (Class3) state3



Сериализованное состояние объекта класса Class3

Class3: (Class2) state21 state22 (Class3) state3

- Для изменения работы механизма сериализации на уровне вашего класса в самом классе надо описать методы:
  - реализация сериализации
    `private void writeObject(ObjectOutputStream out) throws IOException`
  - реализация десериализации
    `private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException`
- Уровень доступа методов позволяет им независимо существовать в различных классах в иерархии наследования
- Можно не переписывать чтение/запись полностью, а лишь изменить порядок записи полей и их формат (см. методы `ObjectOutputStream.writeFields()` и `ObjectInputStream.readFields()`)

- Каждый класс имеет уникальный идентификатор номера версии – 64 битовое значение **`long`**
- По умолчанию значение рассчитывается как функция от кода класса (включая методы)
- Несовпадение версий при десериализации объекта выбрасывает исключение **`InvalidClassException`**
- Проблему можно обойти, явно введя в класс поле

```
private static final long
        serialVersionUID = ...;
```

Thank you!