14 Binary I/O

Motivations

Data stored in a text file is represented in human-readable form. Data stored in a binary file is represented in binary form. You cannot read binary files. They are designed to be read by programs. For example, Java source programs are stored in text files and can be read by a text editor, but Java classes are stored in binary files and are read by the JVM. The advantage of binary files is that they are more efficient to process than text files.

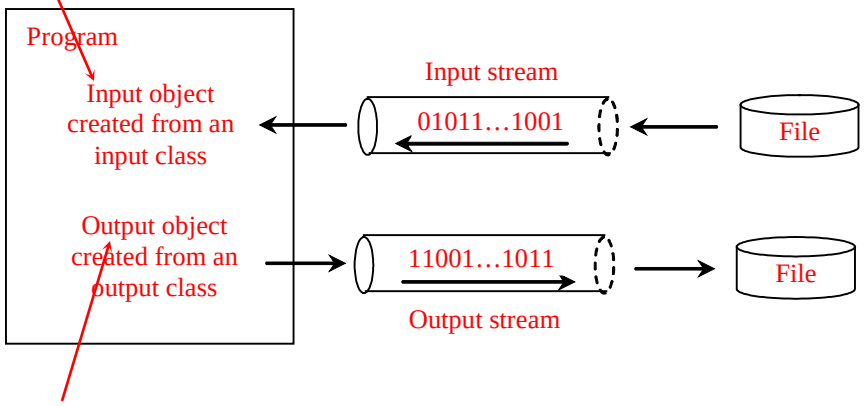
Objectives

- To discover how I/O is processed in Java (§17.2).
- To distinguish between text I/O and binary I/O (§17.3).
- To read and write bytes using FileInputStream and FileOutputStream (§17.4.1).
- To read and write primitive values and strings using DataInputStream/DataOutputStream (§17.4.3).
- To store and restore objects using ObjectOutputStream and ObjectInputStream, and to understand how objects are serialized and what kind of objects can be serialized (§17.6).
- To implement the Serializable interface to make objects serializable (§17.6.1).
- To serialize arrays (§17.6.2).
- To read and write the same file using the RandomAccessFile class (§17.7).

How is I/O Handled in Java?

A File object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file. In order to perform I/O, you need to create objects using appropriate Java I/O classes.

```
Scanner input = new Scanner(new File("temp.txt"));
System.out.println(input.nextLine());
```

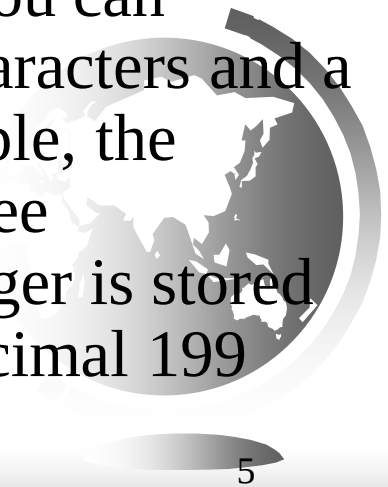



```
PrintWriter output = new PrintWriter("temp.txt");
output.println("Java 101");
output.close();
```

Text File vs. Binary File

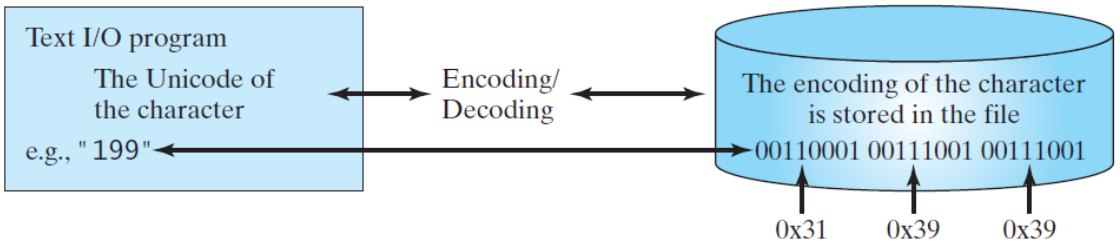
- Data stored in a text file are represented in human-readable form. Data stored in a binary file are represented in binary form. You cannot read binary files. Binary files are designed to be read by programs. For example, the Java source programs are stored in text files and can be read by a text editor, but the Java classes are stored in binary files and are read by the JVM. The advantage of binary files is that they are more efficient to process than text files.

- Although it is not technically precise and correct, you can imagine that a text file consists of a sequence of characters and a binary file consists of a sequence of bits. For example, the decimal integer 199 is stored as the sequence of three characters: '1', '9', '9' in a text file and the same integer is stored as a byte-type value C7 in a binary file, because decimal 199 equals to hex C7.

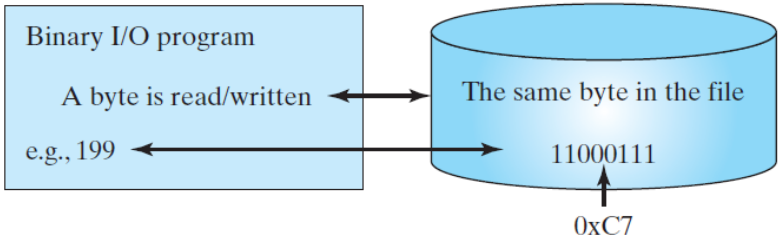


Binary I/O

Text I/O requires encoding and decoding. The JVM converts a Unicode to a file specific encoding when writing a character and coverts a file specific encoding to a Unicode when reading a character. Binary I/O does not require conversions. When you write a byte to a file, the original byte is copied into the file. When you read a byte from a file, the exact byte in the file is returned.

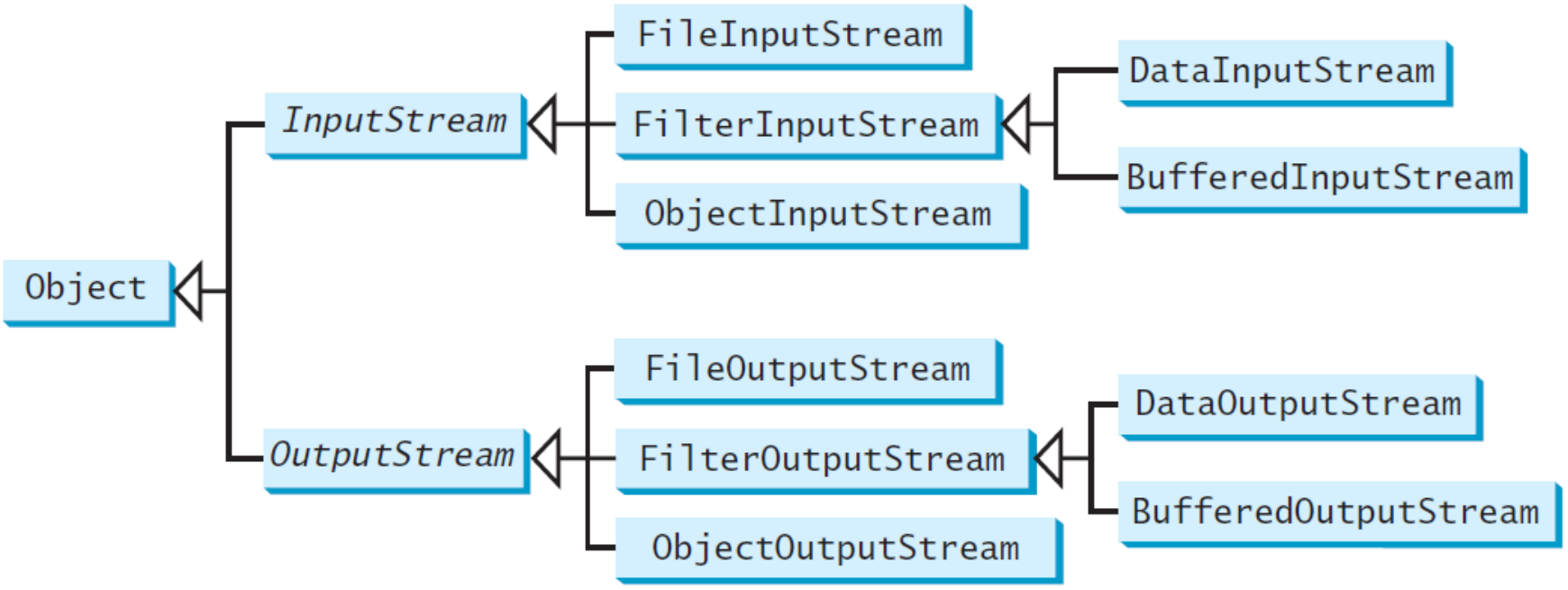


(a)



(b)

Binary I/O Classes



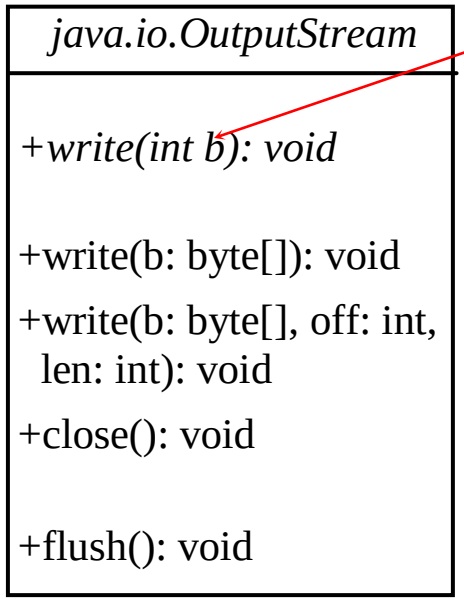
InputStream

The value returned is a byte as an int type.

java.io.InputStream	
+read(): int	Reads the next byte of data from the input stream. The value byte is returned as an int value in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value –1 is returned.
+read(b: byte[]): int	Reads up to b.length bytes into array b from the input stream and returns the actual number of bytes read. Returns -1 at the end of the stream.
+read(b: byte[], off: int, len: int): int	Reads bytes from the input stream and stores into b[off], b[off+1], …, b[off+len-1]. The actual number of bytes read is returned. Returns -1 at the end of the stream.
+available(): int	Returns the number of bytes that can be read from the input stream.
+close(): void	Closes this input stream and releases any system resources associated with the stream.
+skip(n: long): long	Skips over and discards n bytes of data from this input stream. The actual number of bytes skipped is returned.
+markSupported(): boolean	Tests if this input stream supports the mark and reset methods.
+mark(readlimit: int): void	Marks the current position in this input stream.
+reset(): void	Repositions this stream to the position at the time the mark method was last called on this input stream.

OutputStream

The value is a byte as an int type.



Writes the specified byte to this output stream. The parameter b is an int value. (byte)b is written to the output stream.

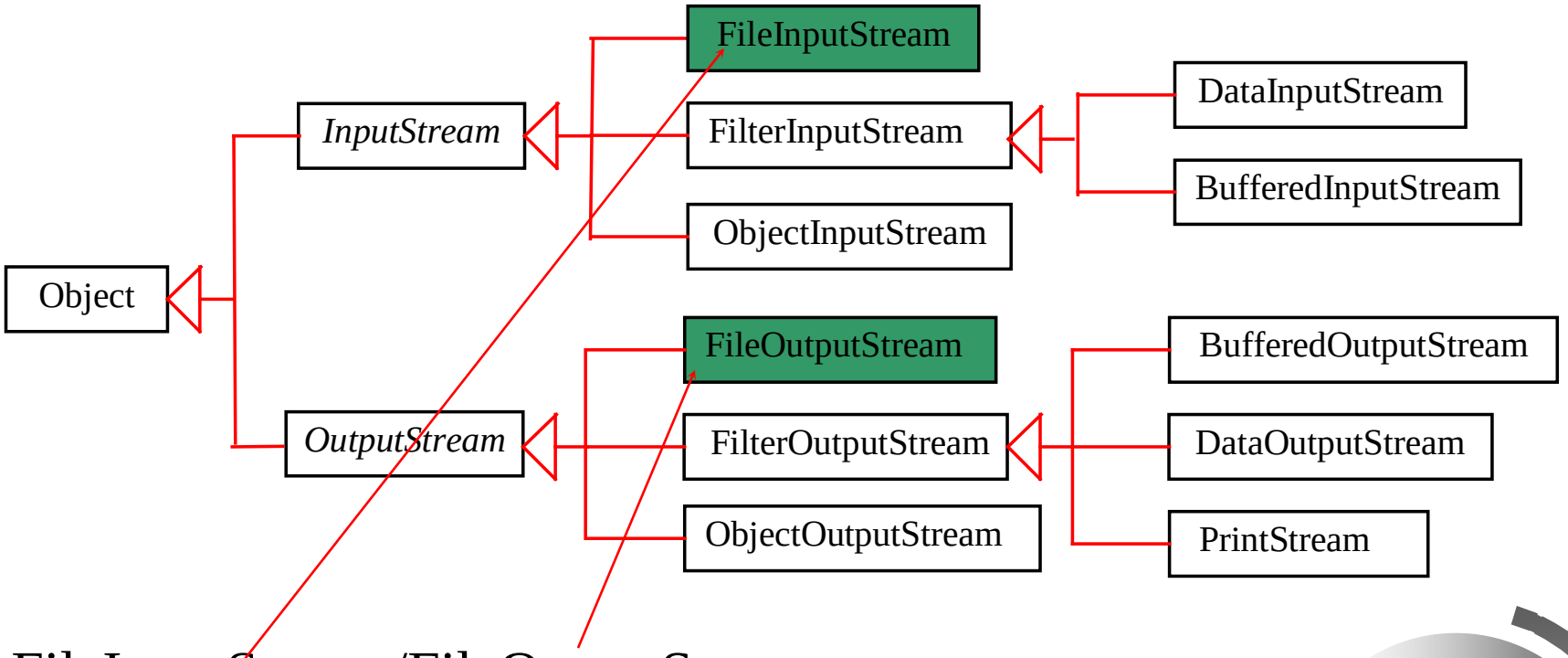
Writes all the bytes in array b to the output stream.

Writes b[off], b[off+1], …, b[off+len-1] into the output stream.

Closes this output stream and releases any system resources associated with the stream.

Flushes this output stream and forces any buffered output bytes to be written out.

FileInputStream/FileOutputStream



FileInputStream/FileOutputStream associates a binary input/output stream with an external file. All the methods in FileInputStream/FileOuptputStream are inherited from its superclasses.

FileInputStream

To construct a FileInputStream, use the following constructors:

public FileInputStream(String filename)

public FileInputStream(File file)

A java.io.FileNotFoundException would occur if you attempt to create a FileInputStream with a nonexistent file.

FileOutputStream

To construct a FileOutputStream, use the following constructors:

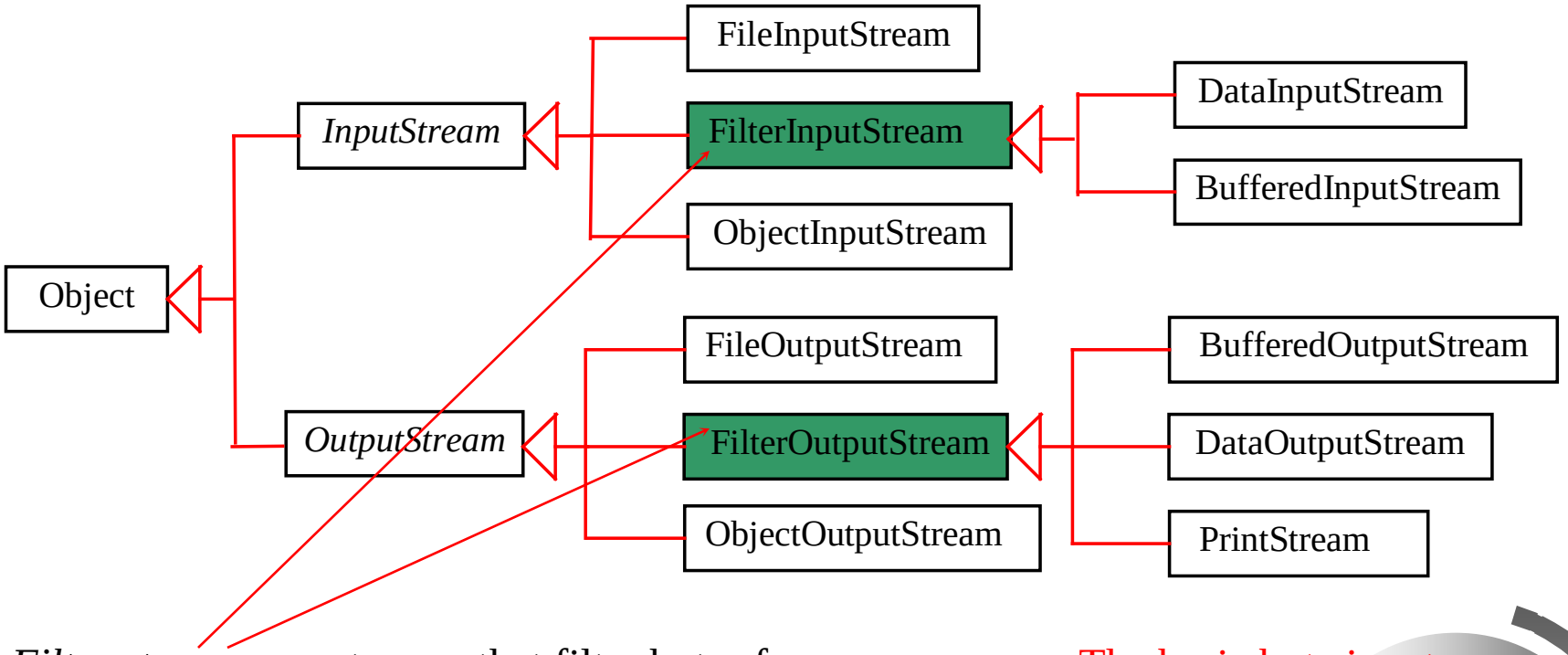
```
public FileOutputStream(String filename)
public FileOutputStream(File file)
public FileOutputStream(String filename, boolean append)
public FileOutputStream(File file, boolean append)
```

If the file does not exist, a new file would be created. If the file already exists, the first two constructors would delete the current contents in the file. To retain the current content and append new data into the file, use the last two constructors by passing true to the append parameter.

TestFileStream

Run

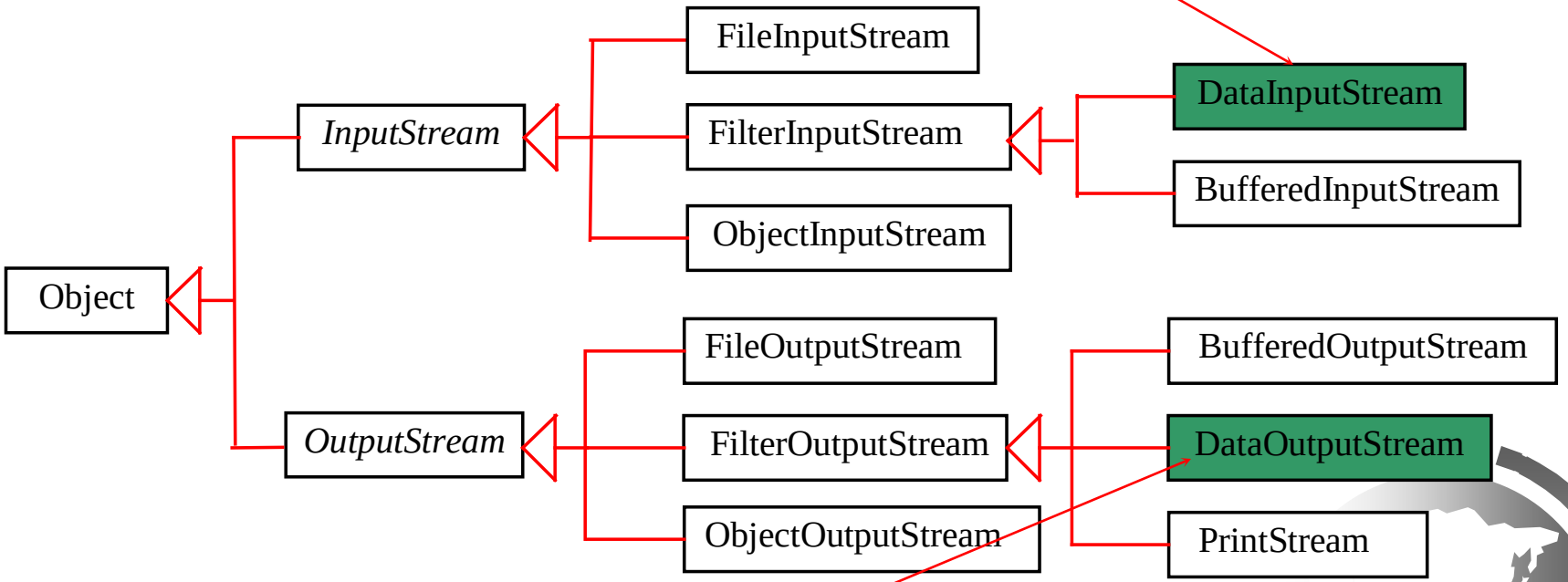
FilterInputStream/FilterOutputStream



Filter streams are streams that filter bytes for some purpose. The basic byte input stream provides a read method that can only be used for reading bytes. If you want to read integers, doubles, or strings, you need a filter class to wrap the byte input stream. Using a filter class enables you to read integers, doubles, and strings instead of bytes and characters. FilterInputStream and FilterOutputStream are the base classes for filtering data. When you need to process primitive numeric types, use DataInputStream and DataOutputStream to filter bytes.

DataInputStream/DataOutputStream

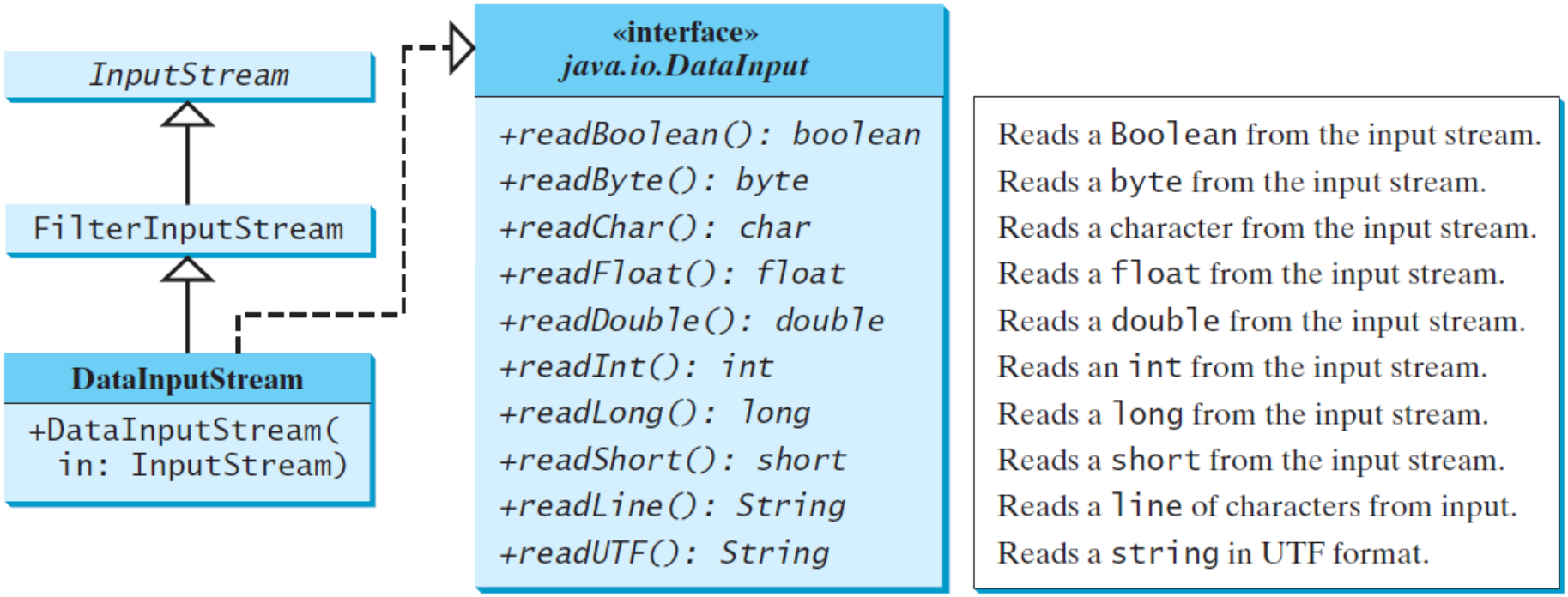
DataInputStream reads bytes from the stream and converts them into appropriate primitive type values or strings.



DataOutputStream converts primitive type values or strings into bytes and output the bytes to the stream.

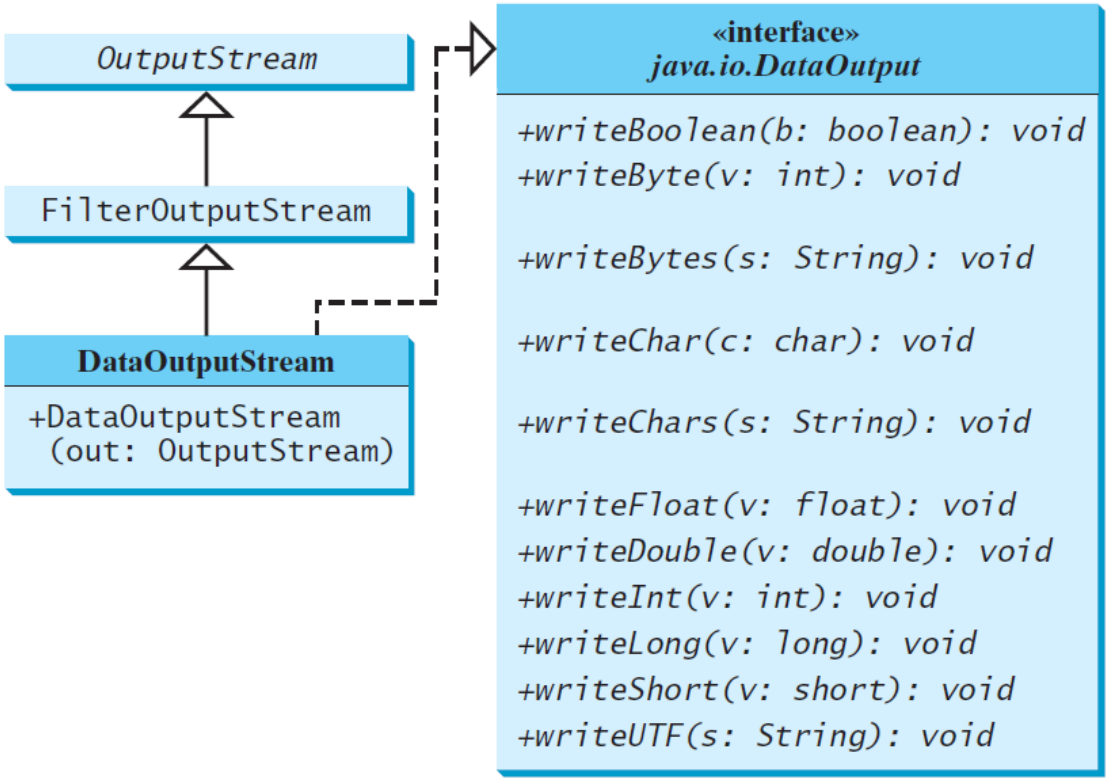
DataInputStream

DataInputStream extends FilterInputStream and implements the DataInput interface.



DataOutputStream

DataOutputStream extends FilterOutputStream and implements the DataOutput interface.



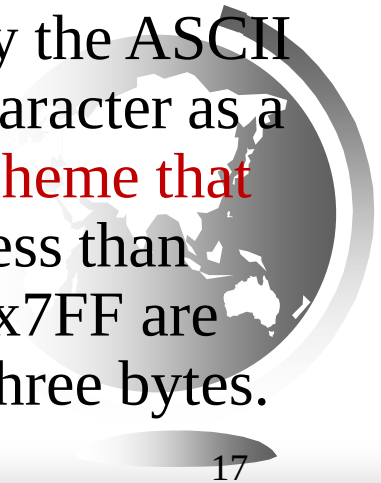
Writes a Boolean to the output stream.	
Writes the eight low-order bits of the argument `v` to the output stream.	
Writes the lower byte of the characters in a string to the output stream.	
Writes a character (composed of 2 bytes) to the output stream.	
Writes every character in the string `s` to the output stream, in order, 2 bytes per character.	
Writes a `float` value to the output stream.	
Writes a `double` value to the output stream.	
Writes an `int` value to the output stream.	
Writes a `long` value to the output stream.	
Writes a `short` value to the output stream.	
Writes `s` string in UTF format.	

Characters and Strings in Binary I/O

A Unicode consists of two bytes. The writeChar(char c) method writes the Unicode of character c to the output. The writeChars(String s) method writes the Unicode for each character in the string s to the output.

Why UTF-8? What is UTF-8?

UTF-8 is a coding scheme that allows systems to operate with both ASCII and Unicode efficiently. Most operating systems use ASCII. Java uses Unicode. The ASCII character set is a subset of the Unicode character set. Since most applications need only the ASCII character set, it is a waste to represent an 8-bit ASCII character as a 16-bit Unicode character. The UTF-8 is an alternative scheme that stores a character using 1, 2, or 3 bytes. ASCII values (less than 0x7F) are coded in one byte. Unicode values less than 0x7FF are coded in two bytes. Other Unicode values are coded in three bytes.



Using DataInputStream/DataOutputStream

Data streams are used as wrappers on existing input and output streams to filter data in the original stream. They are created using the following constructors:

```
public DataInputStream(InputStream instream)
public DataOutputStream(OutputStream outstream)
```

The statements given below create data streams. The first statement creates an input stream for file **in.dat**; the second statement creates an output stream for file **out.dat**.

```
DataInputStream infile =
  new DataInputStream(new FileInputStream("in.dat"));
DataOutputStream outfile =
  new DataOutputStream(new FileOutputStream("out.dat"));
```

TestDataStream

Run

```
public class TestDataStream {
  public static void main(String[] args) throws IOException {
    try ( // Create an output stream for file temp.dat
      DataOutputStream output =
        new DataOutputStream(new FileOutputStream("temp.dat"));
    ) {
      // Write student test scores to the file
      output.writeUTF("John");
      output.writeDouble(85.5);
      output.writeUTF("Jim");
      output.writeDouble(185.5);
      output.writeUTF("George");
      output.writeDouble(105.25);
    }

    try ( // Create an input stream for file temp.dat
      DataInputStream input =
        new DataInputStream(new FileInputStream("temp.dat"));
    ) {
      // Read student test scores from the file
      System.out.println(input.readUTF() + " " + input.readDouble());
      System.out.println(input.readUTF() + " " + input.readDouble());
      System.out.println(input.readUTF() + " " + input.readDouble());
    }
  }
}
```

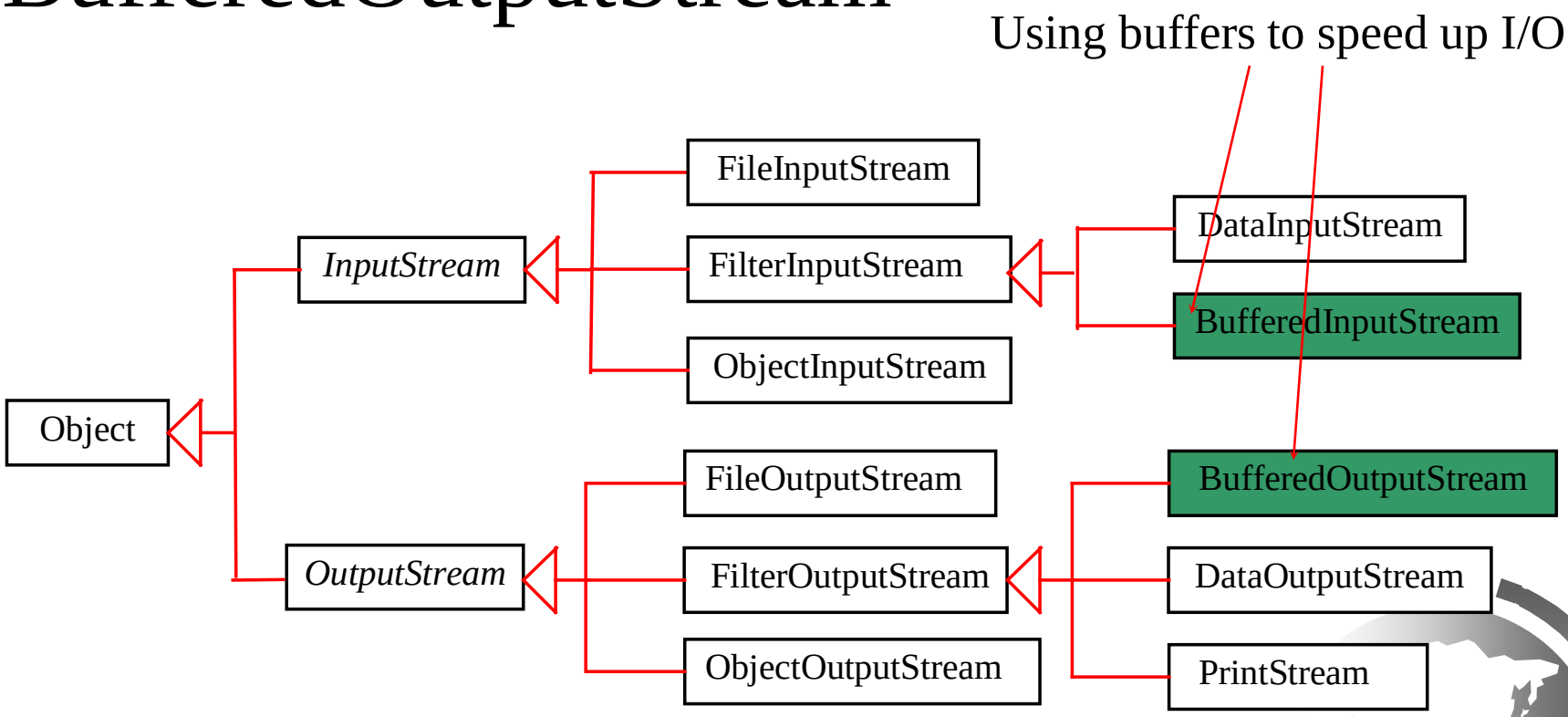
Order and Format

CAUTION: You have to read the data in the same order and same format in which they are stored. For example, since names are written in UTF-8 using writeUTF, you must read names using readUTF.

Checking End of File

TIP: If you keep reading data at the end of a stream, an EOFException would occur. So how do you check the end of a file? You can use input.available() to check it. input.available() == 0 indicates that it is the end of a file.

BufferedInputStream/ BufferedOutputStream



BufferedInputStream/BufferedOutputStream does not contain new methods. All the methods BufferedInputStream/BufferedOutputStream are inherited from the InputStream/OutputStream classes.

Concept of pipe line

int, double, string …

01000110011 …

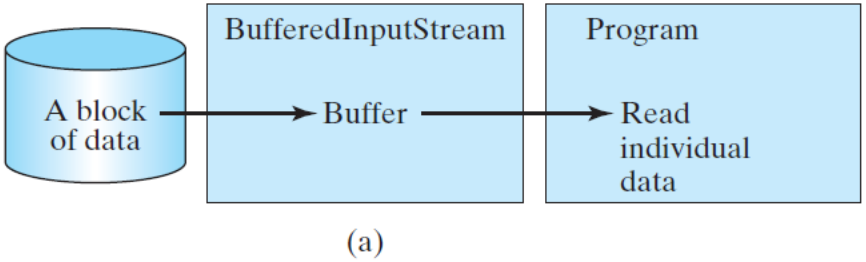
int, double, string …

01000110011 …

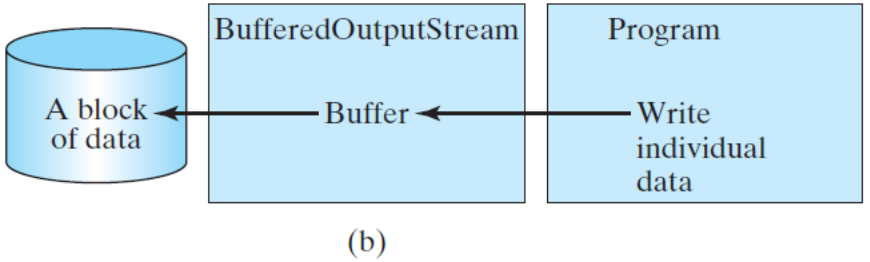
Constructing BufferedInputStream/BufferedOutputStream

```
// Create a BufferedInputStream
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int bufferSize)

// Create a BufferedOutputStream
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStreamr out, int bufferSize)
```



(a)

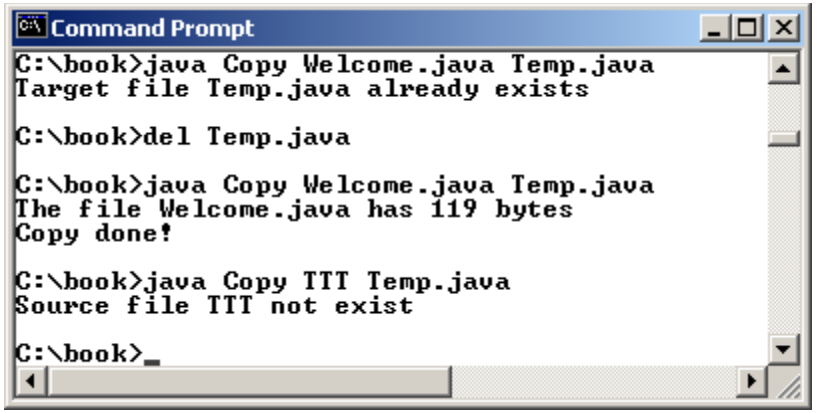


(b)

Case Studies: Copy File

This case study develops a program that copies files. The user needs to provide a source file and a target file as command-line arguments using the following command:

java Copy source target



The program copies a source file to a target file and displays the number of bytes in the file. If the source does not exist, tell the user the file is not found. If the target file already exists, tell the user the file already exists.

FilterInputStream 与 装饰者模式

- 一般有两种方式可以实现给一个类或对象增加行为：
  - 继承机制，使用继承机制是给现有类添加功能的一种有效途径，通过继承一个现有类可以使得子类在拥有自身方法的同时还拥有父类的方法。但是这种方法是静态的，用户不能控制增加行为的方式和时机。
  - 关联机制，即将一个类的对象嵌入另一个对象中，由另一个对象来决定是否调用嵌入对象的行为以便扩展自己的行为，我们称这个嵌入的对象为装饰器(Decorator)。

FilterInputStream 与 装饰者模式

```
package java.io;

public class FilterInputStream extends InputStream {
    /**
     * The input stream to be filtered.
     */
    protected volatile InputStream in;

    /**
     * Creates a <code>FilterInputStream</code>
     * by assigning the  argument <code>in</code>
     * to the field <code>this.in</code> so as
     * to remember it for later use.
     *
     * @param   in   the underlying input stream, or <code>null</code> if
     *          this instance is to be created without an underlying stream.
     */
    protected FilterInputStream(InputStream in) {
        this.in = in;
    }
```

```
public int read() throws IOException {
    return in.read();
}
```

```
public int read(byte b[]) throws IOException {
    return read(b, 0, b.length);
}
```

```
public int read(byte b[], int off, int len) throws IOException {
    return in.read(b, off, len);
}
```

```
public void close() throws IOException {
    in.close();
}
```

FilterInputStream 与 装饰者模式

- FilterInpustStream子类可以分成两类：
  - 1） DataInputStream能以一种与机器无关的方式，直接从地从字节输入流读取JAVA基本类型和String类型的数据。
  - 2 ）其它的子类使得能够对InputStream进行改进，即在原有的InputStream基础上可以提供了新的功能特性。日常中用的最多的就是BufferedInputStream，使得inputStream具有缓冲的功能。

FilterInputStream 与 装饰者模式

- **DataInputStream：**之前的InputStream我们只能读取byte，这个类使得我们可以直接从stream中读取int，String等类型。

```
public final int readInt() throws IOException {
    int ch1 = in.read();
    int ch2 = in.read();
    int ch3 = in.read();
    int ch4 = in.read();
    if ((ch1 | ch2 | ch3 | ch4) < 0)
        throw new EOFException();
    return ((ch1 << 24) + (ch2 << 16) + (ch3 << 8) + (ch4 << 0));
}
```

```
501     public final String readLine() throws IOException {
502         char buf[] = lineBuffer;
503 
504         if (buf == null) {
505             buf = lineBuffer = new char[128];
506         }
507 
508         int room = buf.length;
509         int offset = 0;
510         int c;
511 
512 loop:   while (true) {
513             switch (c = in.read()) {
514               case -1:
515               case '\n':
516                 break loop;
517 
518               case '\r':
519                 int c2 = in.read();
520                 if ((c2 != '\n') && (c2 != -1)) {
521                     if (!(in instanceof PushbackInputStream)) {
522                         this.in = new PushbackInputStream(in);
523                     }
524                     ((PushbackInputStream)in).unread(c2);
525                 }
526                 break loop;
527 
528               default:
529                 if (--room < 0) {
530                     buf = new char[offset + 128];
531                     room = buf.length - offset - 1;
532                     System.arraycopy(lineBuffer, 0, buf, 0, offset);
533                     lineBuffer = buf;
534                 }
535                 buf[offset++] = (char) c;
536                 break;
537             }
538         }
539         if ((c == -1) && (offset == 0)) {
540             return null;
541         }
542         return String.copyValueOf(buf, 0, offset);
543     }
```

FilterInputStream 与 装饰者模式

- **BufferedInputStream：**在read时，干脆多读一部分数据进来，放在内存里，等你每次操作流的时候，读取的数据直接从内存中就可以拿到，这就减少了io次数

```
public class BufferedInputStream extends FilterInputStream {
//....省略部分源码
private static int DEFAULT_BUFFER_SIZE = 8192;

private static int MAX_BUFFER_SIZE = Integer.MAX_VALUE - 8;

protected volatile byte buf[];

public synchronized int read() throws IOException {
    if (pos >= count) {
        fill();
        if (pos >= count)
            return -1;
    }
    return getBufIfOpen()[pos++] & 0xff;
}
```

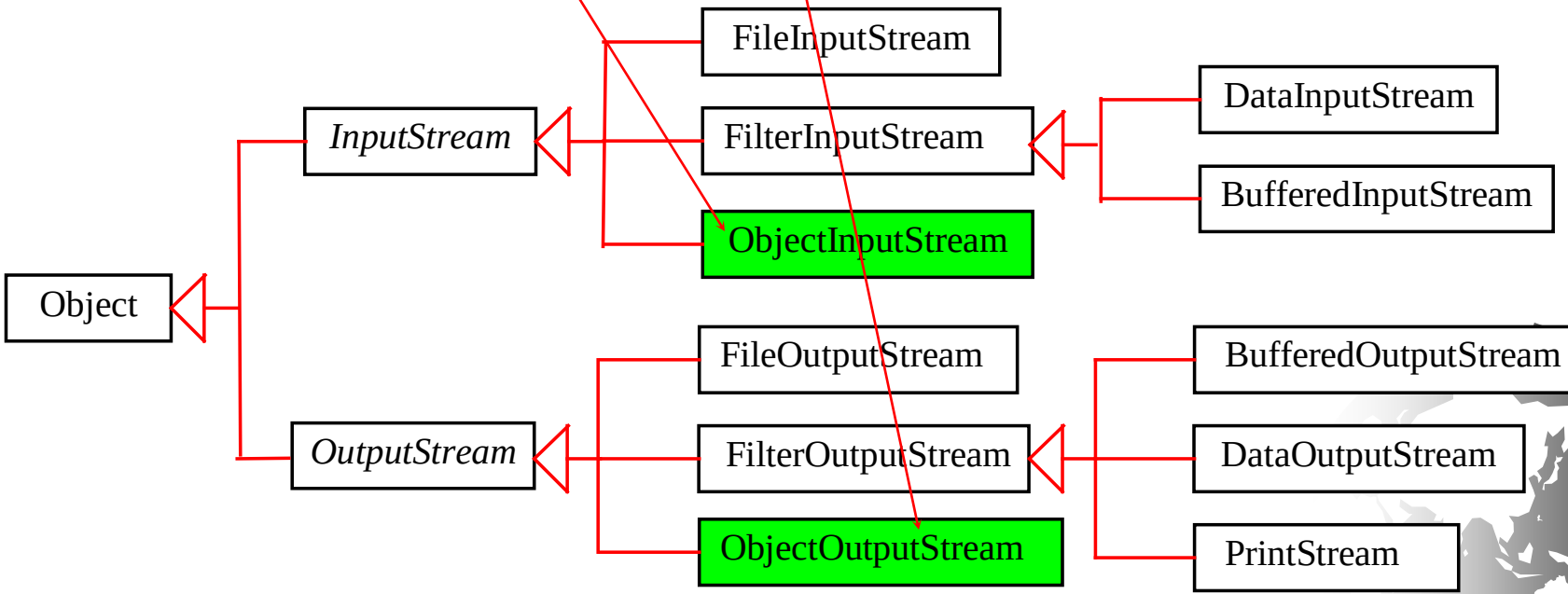
每次调用read读取数据时，先查看要读取的数据是否在缓存中，如果在缓存中，直接从缓存中读取；如果不在缓存中，则调用fill方法，从InputStream中读取一定的存储到buf中。

```
private void fill() throws IOException {
    byte[] buffer = getBufIfOpen();
    if (markpos < 0)
        pos = 0;            /* no mark: throw away the buffer */
    else if (pos >= buffer.length)  /* no room left in buffer */
        if (markpos > 0) {  /* can throw away early part of the buffer */
            int sz = pos - markpos;
            System.arraycopy(buffer, markpos, buffer, 0, sz);
            pos = sz;
            markpos = 0;
        } else if (buffer.length >= marklimit) {
            markpos = -1;   /* buffer got too big, invalidate mark */
            pos = 0;        /* drop buffer contents */
        } else if (buffer.length >= MAX_BUFFER_SIZE) {
            throw new OutOfMemoryError("Required array size too large");
        } else {            /* grow buffer */
            int nsz = (pos <= MAX_BUFFER_SIZE - pos) ?
                    pos * 2 : MAX_BUFFER_SIZE;
            if (nsz > marklimit)
                nsz = marklimit;
            byte nbuf[] = new byte[nsz];
            System.arraycopy(buffer, 0, nbuf, 0, pos);
            if (!bufUpdater.compareAndSet(this, buffer, nbuf)) {
                // Can't replace buf if there was an async close.
                // Note: This would need to be changed if fill()
                // is ever made accessible to multiple threads.
                // But for now, the only way CAS can fail is via close.
                // assert buf == null;
                throw new IOException("Stream closed");
            }
            buffer = nbuf;
        }
    count = pos;
    int n = getInIfOpen().read(buffer, pos, buffer.length - pos);
    if (n > 0)
        count = n + pos;
}
```

Optional

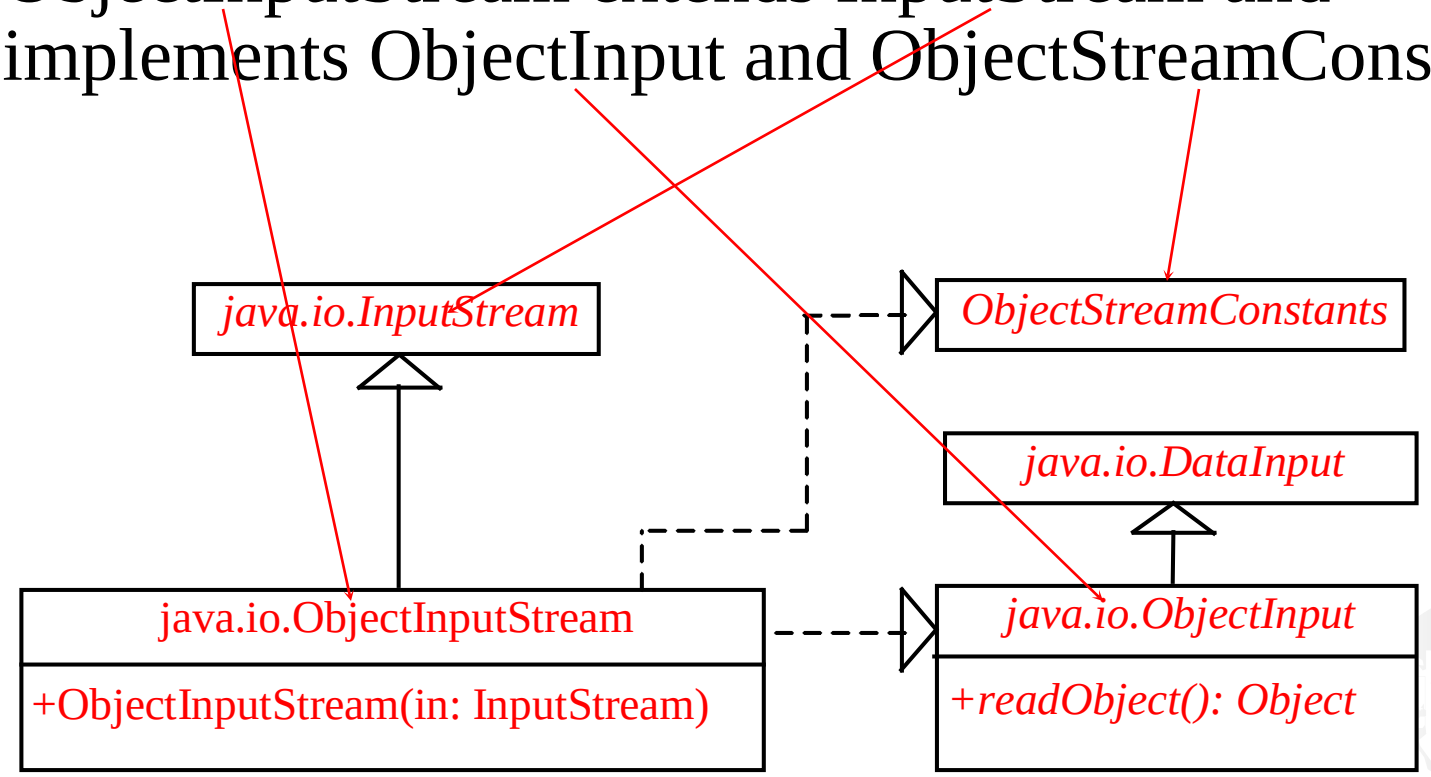
Object I/O

DataInputStream/DataOutputStream enables you to perform I/O for primitive type values and strings.
ObjectInputStream/ObjectOutputStream enables you to perform I/O for objects in addition for primitive type values and strings.



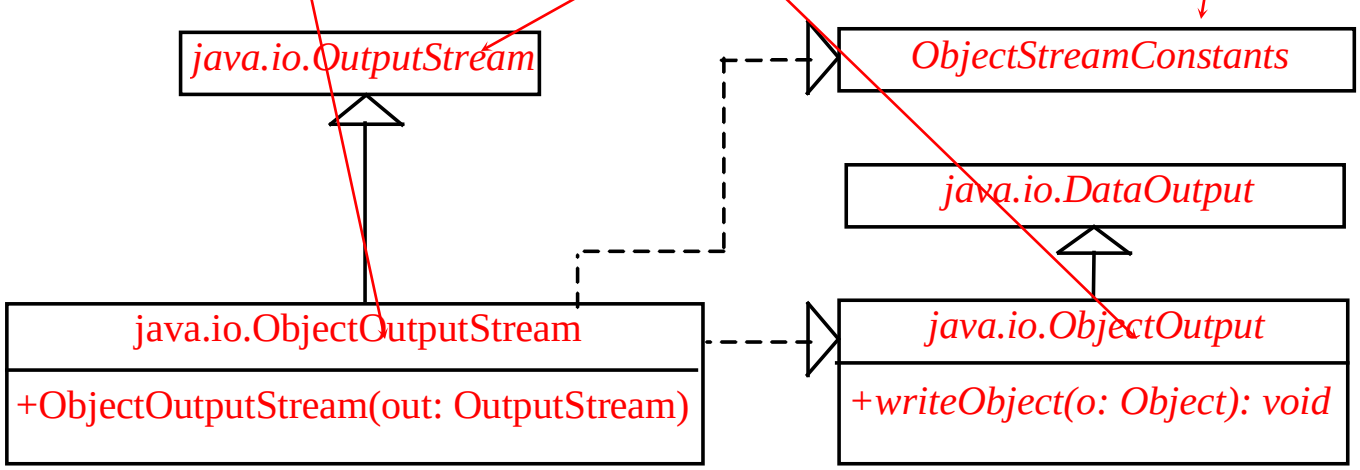
ObjectInputStream

ObjectInputStream extends InputStream and implements ObjectInput and ObjectStreamConstants.



ObjectOutputStream

ObjectOutputStream extends OutputStream and implements ObjectOutput and ObjectStreamConstants.

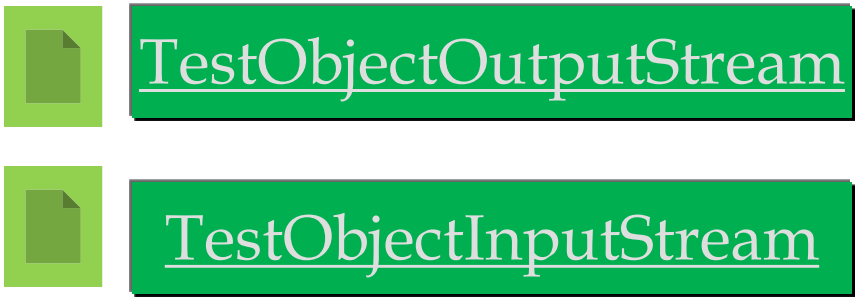


Using Object Streams

You may wrap an ObjectInputStream/ObjectOutputStream on any InputStream/OutputStream using the following constructors:

```
// Create an ObjectInputStream
public ObjectInputStream(InputStream in)

// Create an ObjectOutputStream
public ObjectOutputStream(OutputStream out)
```



```java
public class TestObjectOutputStream {
  public static void main(String[] args) throws IOException {
    try ( // Create an output stream for file object.dat
      ObjectOutputStream output =
        new ObjectOutputStream(new FileOutputStream("object.dat"));
    ) {
      // Write a string, double value, and object to the file
      output.writeUTF("John");
      output.writeDouble(85.5);
      output.writeObject(new java.util.Date());
    }
  }
}
```

```java
public class TestObjectInputStream {
  public static void main(String[] args)
    throws ClassNotFoundException, IOException {
    try ( // Create an input stream for file object.dat
      ObjectInputStream input =
        new ObjectInputStream(new FileInputStream("object.dat"));
    ) {
      // Read a string, double value, and object from the file
      String name = input.readUTF();
      double score = input.readDouble();
      java.util.Date date = (java.util.Date)(input.readObject());
      System.out.println(name + " " + score + " " + date);
    }
  }
}
```

The `Serializable` Interface

Not all objects can be written to an output stream. Objects that can be written to an object stream is said to be *serializable*. A serializable object is an instance of the java.io.Serializable interface. So the class of a serializable object must implement Serializable.

The Serializable interface is a marker interface. It has no methods, so you don't need to add additional code in your class that implements Serializable.

Implementing this interface enables the Java serialization mechanism to automate the process of storing the objects and arrays.

The `transient` Keyword

If an object is an instance of Serializable, but it contains non-serializable instance data fields, can the object be serialized? The answer is no.

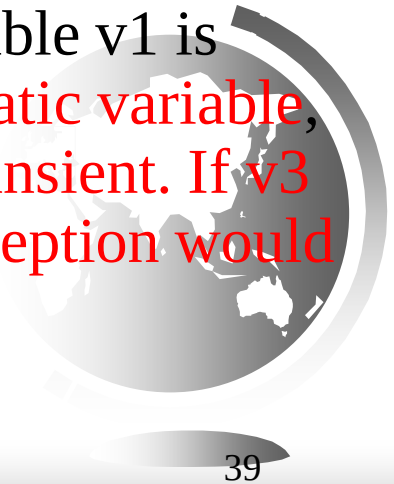
To enable the object to be serialized, you can use the transient keyword to mark these data fields to tell the JVM to ignore these fields when writing the object to an object stream.

The `transient` Keyword, cont.

Consider the following class:

```
public class Foo implements java.io.Serializable {
  private int v1;
  private static double v2;
  private transient A v3 = new A();
}
class A { } // A is not serializable
```

When an object of the Foo class is serialized, only variable v1 is serialized. Variable v2 is not serialized because it is a static variable, and variable v3 is not serialized because it is marked transient. If v3 were not marked transient, a java.io.NotSerializableException would occur.



Serializing Arrays

An array is serializable if all its elements are serializable. So an entire array can be saved using writeObject into a file and later restored using readObject. Here is an example that stores an array of five int values and an array of three strings, and reads them back to display on the console.

TestObjectStreamForArray

Run

```java
public class TestObjectStreamForArray {
  public static void main(String[] args)
      throws ClassNotFoundException, IOException {
    int[] numbers = {1, 2, 3, 4, 5};
    String[] strings = {"John", "Susan", "Kim"};

    try ( // Create an output stream for file array.dat
      ObjectOutputStream output = new ObjectOutputStream(new
        FileOutputStream("array.dat", true));
    ) {
      // Write arrays to the object output stream
      output.writeObject(numbers);
      output.writeObject(strings);
    }

    try ( // Create an input stream for file array.dat
      ObjectInputStream input =
        new ObjectInputStream(new FileInputStream("array.dat"));
    ) {
      int[] newNumbers = (int[])(input.readObject());
      String[] newStrings = (String[])(input.readObject());

      // Display arrays
      for (int i = 0; i < newNumbers.length; i++)
        System.out.print(newNumbers[i] + " ");
      System.out.println();

      for (int i = 0; i < newStrings.length; i++)
        System.out.print(newStrings[i] + " ");
    }
  }
}
```

1. 反序列化后的对象，需要调用构造函数重新构造吗？

- 答案：不需要。对于Serializable对象，对象完全以它存储的二进制位作为基础来构造，而不调用构造器。

```
package test.serializable;

import java.io.Serializable;
import java.util.Date;

/**
 *
 * @author chenfei
 *
 * 用于测试序列化时的deep copy
 *
 */
public class House implements Serializable {
    private static final long serialVersionUID = -6091530420906090649L;

    private Date date = new Date(); //记录当前的时间

    public String toString() {
        return "House:" + super.toString() + ".Create Time is:" + date;
    }

}
```

```java
package test.serializable;

import java.io.Serializable;

public class Animal implements Serializable {
    private static final long serialVersionUID = -2132211891929620 74L;

    private String name;

    private House house;

    public Animal(String name , House house) {
        this.name = name;
        this.house = house;
        System.out.println("调用了构造器");
    }

    public String toString() {
        return  name + "[" +super.toString() + "']" + house;
    }

}
```

```
public static void main(String[] args) throws IOException, ClassNotFoundException {
    House house = new House();
    System.out.println("序列化前");
    Animal animal = new Animal("test",house);
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(out);
    oos.writeObject(animal);
    oos.flush();
    oos.close();

    System.out.println("反序列化后");
    ByteArrayInputStream in = new ByteArrayInputStream(out.toByteArray());
    ObjectInputStream ois = new ObjectInputStream(in);
    Animal animal1 = (Animal)ois.readObject();
    ois.close();
}
```

运行结果如下所示：

```
序列化前
调用了构造器
反序列化后
```

- 问题：序列前的对象与序列化后的对象是什么关系？是"=="还是equal？是浅复制还是深复制？

2. 序列前的对象与序列化后的对象是什么关系？是"=="还是equal？是浅复制还是深复制？

```
public static void main(String[] args) throws IOException, ClassNotFoundException {
    House house = new House();
    System.out.println("序列化前");
    Animal animal = new Animal("test",house);
    System.out.println(animal);
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(out);
    oos.writeObject(animal);
    oos.writeObject(animal);//在写一次，看对象是否是一样，
    oos.flush();
    oos.close();

    ByteArrayOutputStream out2 = new ByteArrayOutputStream();//换一个输出流
    ObjectOutputStream oos2 = new ObjectOutputStream(out2);
    oos2.writeObject(animal);
    oos2.flush();
    oos2.close();

    System.out.println("反序列化后");
    ByteArrayInputStream in = new ByteArrayInputStream(out.toByteArray());
    ObjectInputStream ois = new ObjectInputStream(in);
    Animal animal1 = (Animal)ois.readObject();
    Animal animal2 = (Animal)ois.readObject();
    ois.close();

    ByteArrayInputStream in2 = new ByteArrayInputStream(out2.toByteArray());
    ObjectInputStream ois2 = new ObjectInputStream(in2);
    Animal animal3 = (Animal)ois2.readObject();
    ois2.close();

    System.out.println("out流: " +animal1);
    System.out.println("out流: " +animal2);
    System.out.println("out2流: " +animal3);


    System.out.println("测试序列化前后的对象 == : "+ (animal==animal1));
    System.out.println("测试序列化后同一流的对象: "+ (animal1 == animal2));
    System.out.println("测试序列化后不同流的对象==:" + (animal1==animal3));

}
```

2. 序列前的对象与序列化后的对象是什么关系？是("==" 还是equal？是浅复制还是深复制？)

```
序列化前
调用了构造器
test[test.serializable.Animal@bb7465']House:test.serializable.House@d6c16c.Create Time is:Sat Apr 06 00:11:30 CST 2013
反序列化后
out流: test[test.serializable.Animal@4f80d6']House:test.serializable.House@193722c.Create Time is:Sat Apr 06 00:11:30 CST 2013
out流: test[test.serializable.Animal@4f80d6']House:test.serializable.House@193722c.Create Time is:Sat Apr 06 00:11:30 CST 2013 (与上面的相同)
out2流: test[test.serializable.Animal@12cc95d']House:test.serializable.House@157fb52.Create Time is:Sat Apr 06 00:11:30 CST 2013 (与上面只是值相同，但是地址不一样。)
测试序列化前后的对象 == : false
测试序列化后同一流的对象: true
测试序列化后不同流的对象==:false
```

- 答案：深复制，反序列化还原后的对象地址与原来的的地址不同。
- 通过序列化操作，我们可以实现对任何可Serializable对象的”深度复制（deep copy）"——这意味着我们复制的是整个对象网，而不仅仅是基本对象及其引用。对于同一流的对象，他们的地址是相同，说明他们是同一个对象，但是与其他流的对象地址却不相同。

3. serialVersionUID 的作用？

- Serializable接口的代价一：若一个类实现Serializable接口，会大大降低了“改变这个类的实现”的灵活性。

  如果不设计一种自定义的序列化形式，而是采用默认的序列化形式，则会被束缚在该类最初的内部表示法上。就是说，若之后改变该类的内部表示法，结果会导致不兼容。用旧版本来序列化一个类，而用新版本来反序列化，会导致程序失效。

  凡是实现Serializable接口的类都有一个表示序列化版本标识符的静态long类型变量：private static final long serialVersionUID;

  如果没有设置这个值，在序列化一个对象之后，改动了该类的字段或者方法名之类的，再反序列化想取出之前的那个对象时就可能会抛出异常，因为serialVersionUID是根据类名、接口名、成员方法及属性等来生成一个64位的哈希字段，当修改后的类去反序列化时，该类的serialVersionUID值和之前保存在文件中的serialVersionUID值不一致，所以就会抛出异常。

```java
public static void main(String[] args)  throws Exception {
    SerializeCustomer();// 序列化Customer对象
    Customer customer = DeserializeCustomer();// 反序列Customer对象
    System.out.println(customer);
}
 private static void SerializeCustomer() throws FileNotFoundException,
    IOException {
     Customer customer = new Customer("gacl",25);
     // ObjectOutputStream 对象输出流
     ObjectOutputStream oo = new ObjectOutputStream(new FileOutputStream(
             new File("D:/TEMP/Customer.txt")));
     oo.writeObject(customer);
     System.out.println("Customer对象序列化成功! ");
     oo.close();
 }

 private static Customer DeserializeCustomer() throws Exception, IOException {
     ObjectInputStream ois = new ObjectInputStream(new FileInputStream(
             new File("D:/TEMP/Customer.txt")));
     Customer customer = (Customer) ois.readObject();
     System.out.println("Customer对象反序列化成功! ");
     return customer;
 }
```

```java
public class Customer implements Serializable {
    private String name ;
    private int age;
    public Customer(String name, int age){
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "name=" + name + ", age=" + age;
    }
}
```

```
Customer对象序列化成功!
Customer对象反序列化成功!
name=gacl, age=25
```

```java
public class Customer implements Serializable {
    private String name ;
    private int age;
    private String sex;
    public Customer(String name, int age){
        this.name = name;
        this.age = age;
    }
    public Customer(String name,int age, String sex){
        this.name = name;
        this.age = age;
        this.sex = sex;
    }
    @Override
    public String toString() {
        return "name=" + name + ", age=" + age;
    }
```

修改Customer后，对原来的文件进行反序列化

```java
    public static void main(String[] args)  throws Exception {
//        SerializeCustomer();// 序列化Customer对象
        Customer customer = DeserializeCustomer();// 反序列Customer对象
        System.out.println(customer);
    }
```

```
Exception in thread "main" java.io.InvalidClassException: testForSerializable.Customer; local class incomp
        at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:621)
        at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1623)
        at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1518)
        at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1774)
        at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1351)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java:371)
        at testForSerializable.TestSerialversionUID.DeserializeCustomer(TestSerialversionUID.java:32)
        at testForSerializable.TestSerialversionUID.main(TestSerialversionUID.java:15)
```

```java
public class Customer implements Serializable {
    private static final long serialVersionUID = -5182532647273106745L;
    private String name ;
    private int age;
//  private String sex;
    public Customer(String name, int age){
        this.name = name;
        this.age = age;
    }
/*  public Customer(String name,int age, String sex){
        this.name = name;
        this.age = age;
        this.sex = sex;
    }*/
    @Override
    public String toString() {
        return "name=" + name + ", age=" + age;
    }
}
```

```java
public static void main(String[] args)  throws Exception {
    SerializeCustomer();// 序列化Customer对象
    Customer customer = DeserializeCustomer();// 反序列Customer对象
    System.out.println(customer);
}
```

```
Customer对象序列化成功!
Customer对象反序列化成功!
name=gacl, age=25
```

```java
public class Customer implements Serializable {
    private static final long serialVersionUID = -5182532647273106745L;
    private String name ;
    private int age;
    private String sex;
    public Customer(String name, int age){
        this.name = name;
        this.age = age;
    }
    public Customer(String name,int age, String sex){
        this.name = name;
        this.age = age;
        this.sex = sex;
    }
    @Override
    public String toString() {
        return "name=" + name + ", age=" + age;
    }
}
```

修改Customer后，对原来的文件进行反序列化

```java
    public static void main(String[] args)  throws Exception {
//        SerializeCustomer();// 序列化Customer对象
        Customer customer = DeserializeCustomer();// 反序列Customer对象
        System.out.println(customer);
    }
```

```
Customer对象反序列化成功!
name=gacl, age=25
```

4. writeObject 和 readObject

- 当 ObjectOutputStream对一个Customer对象进行序列化时，如果该对象具有writeObject()方法，那么就会执行这一方法， 否则就按默认方式序列化。在该对象的writeObject()方法中，可以先调用ObjectOutputStream的 defaultWriteObject()方法，使得对象输出流先执行默认的序列化操作。同理可得出反序列化的情况，不过这次是 defaultReadObject()方法。

```
public class Person implements Serializable {
   ...
   transient private Integer age = null;
   ...

   private void writeObject(ObjectOutputStream out) throws IOException {
      out.defaultWriteObject();
      out.writeInt(age);
   }

   private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundExce
      in.defaultReadObject();
      age = in.readInt();
   }
}
```

虽然age为transient，不会被默认序列化，但可以用writeObject和readObject进行序列化

5. Externalizable

- Java默认的序列化机制非常简单，而且序列化后的对象不需要再次调用构造器重新生成，但是在实际中，我们可以会希望对象的某一部分不需要被序列化，或者说一个对象被还原之后，其内部的某些子对象需要重新创建，从而不必将该子对象序列化。 在这些情况下，我们可以考虑实现Externalizable接口从而代替Serializable接口来对序列化过程进行控制（通过transient的方式更简单）。
- 相当于完全自己控制序列化和反序列化，之前基于Serializable接口的序列化机制就将失效。
- Externalizable接口extends Serializable接口，而且在其基础上增加了两个方法：writeExternal()和readExternal()。这两个方法会在序列化和反序列化还原的过程中被自动调用，以便执行一些特殊的操作。

```
public interface Externalizable extends java.io.Serializable {

    void writeExternal(ObjectOutput out) throws IOException;

    void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;
}
```

```
public class Blip implements Externalizable {

    private int i ;

    private String s;//没有初始化

    public Blip() {
         //默认构造函数必须有，而且必须是public
        System.out.println("Blip默认构造函数");
    }

    public Blip(String s ,int i) {
        //s,i只是在带参数的构造函数中进行初始化。
        System.out.println("Blip带参数构造函数");
        this.s = s;
        this.i = i;
    }

    public String toString() {
        return s  + i ;
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException,
            ClassNotFoundException {
        System.out.println("调用readExternal () 方法");
        s = (String)in.readObject();//在反序列化时，需要初始化s和i，否则只是调用默认构造函数，得不到s和i的值
        i = in.readInt();
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        System.out.println("调用writeExternal () 方法");
        out.writeObject(s); //如果我们不将s和i的值写入的话，那么在反序列化的时候，就不会得到这些值。
        out.writeInt(i);
    }

}
```

```java
public class ExternalizableTest {

    /**
     * @param args
     * @throws IOException
     * @throws ClassNotFoundException
     */
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        System.out.println("序列化之前");
        Blip b = new Blip("This String is " , 47);
        System.out.println(b);

        System.out.println("序列化操作, writeObject");
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(out);
        oos.writeObject(b);
        System.out.println("反序列化之后,readObject");
        ByteArrayInputStream in = new ByteArrayInputStream(out.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(in);
        Blip bb = (Blip)ois.readObject();
        System.out.println(bb);
    }
}
```

```
序列化之前
Blip带参数构造函数
This String is 47
序列化操作, writeObject
调用writeExternal () 方法
反序列化之后,readObject
Blip默认构造函数
调用readExternal () 方法
This String is 47
```

1） 如果我们只修改writeExternal（）方法如下：

```
@Override
    public void writeExternal(ObjectOutput out) throws IOException {
        System.out.println("调用writeExternal () 方法");
//        out.writeObject(s);
//        out.writeInt(i);
    }
```

那么运行的结果为：

```
序列化之前
Blip带参数构造函数
This String is 47
序列化操作, writeObject
调用writeExternal () 方法
反序列化之后,readObject
Blip默认构造函数
调用readExternal () 方法
Exception in thread "main" java.io.OptionalDataException
    at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1349)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:351)
    at test.serializable.Blip.readExternal(Blip.java:34)
    at java.io.ObjectInputStream.readExternalData(ObjectInputStream.java:1792)
    at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1751)
    at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1329)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:351)
    at test.serializable.ExternalizableTest.main(ExternalizableTest.java:28)
```

原因是因为，我们在ObjectOutPutStream中没有writeObject,而在ObjectInputStream中readObject导致的

2）如果我们修改writeExternal（）方法如下：

```
@Override
    public void writeExternal(ObjectOutput out) throws IOException {
        System.out.println("调用writeExternal（）方法");
        out.writeObject("自己定义的");
        out.writeInt(250);
    }
```

那么运行的结果为：

```
序列化之前
Blip带参数构造函数
This String is 47
序列化操作，writeObject
调用writeExternal（）方法
反序列化之后，readObject
Blip默认构造函数
调用readExternal（）方法
自己定义的250
```

反序列化后得到的s和i是我们在writeExternal（）中自定义的数据

3） 如果我们只是修改readExternal()方法

```
    @Override
    public void readExternal(ObjectInput in) throws IOException,
            ClassNotFoundException {
        System.out.println("调用readExternal () 方法");
//          s = (String)in.readObject();
//          i = in.readInt();
    }
```

那么运行的结果为：

```
序列化之前
Blip带参数构造函数
This String is 47
序列化操作, writeObject
调用writeExternal () 方法
反序列化之后,readObject
Blip默认构造函数
调用readExternal () 方法
null0
```

4）如果我们删除Blip的默认构造函数，或者将其权限不设置为public

```
//    public Blip() {
//         //默认构造函数必须有，而且必须是public
//        System.out.println("Blip默认构造函数");
//    }
//    or
     Blip() {
         //默认构造函数必须有，而且必须是public
        System.out.println("Blip默认构造函数");
    }
```

运行结果如下：

```
序列化之前
Blip带参数构造函数
This String is 47
序列化操作，writeObject
调用writeExternal ()方法
反序列化之后，readObject
Exception in thread "main" java.io.InvalidClassException: test.serializable.Blip; test.serializable.Blip; no valid constructor
    at java.io.ObjectStreamClass.checkDeserialize(ObjectStreamClass.java:713)
    at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1733)
    at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1329)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:351)
    at test.serializable.ExternalizableTest.main(ExternalizableTest.java:28)
Caused by: java.io.InvalidClassException: test.serializable.Blip; no valid constructor
    at java.io.ObjectStreamClass.<init>(ObjectStreamClass.java:471)
    at java.io.ObjectStreamClass.lookup(ObjectStreamClass.java:310)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1106)
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:326)
    at test.serializable.ExternalizableTest.main(ExternalizableTest.java:24)
```

必须有权限为public的默认的构造器

总结Externalizable对象的用法

- 与Serizable对象不同，使用Externalizable，就意味着没有任何东西可以自动序列化， 为了正常的运行，我们需要在writeExtenal()方法中将自对象的重要信息写入，从而手动的完成序列化。对于一个Externalizabled对象，对象的默认构造函数都会被调用（包括哪些在定义时已经初始化的字段），然后调用readExternal()，在此方法中必须手动的恢复数据。

6. 对单例模式的影响

- 单例模式的序列化

```
public class Customer implements Serializable {
    private static final long serialVersionUID = -5182532647273106745L;
    private String name ;
    private int age;
    private String sex;
    private static class InstanceHolder {
        private static final Customer instatnce = new Customer("John", 31);
    }

    public static Customer getInstance() {
        return InstanceHolder.instatnce;
    }

    private Customer(String name, int age){
        this.name = name;
        this.age = age;
    }
    private Customer(String name,int age, String sex){
        this.name = name;
        this.age = age;
        this.sex = sex;
    }
    @Override
    public String toString() {
        return "name=" + name + ", age=" + age;
    }
}
```

```java
public class TestSerialSingleton {

    public static void main(String[] args)throws IOException, ClassNotFoundException {
        File file = new File("D:/TEMP/Customer.txt");
        ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(file));
        out.writeObject(Customer.getInstance()); // 保存单例对象
        out.close();

        ObjectInputStream oin = new ObjectInputStream(new FileInputStream(file));
        Object newCustomer = oin.readObject();
        oin.close();
        System.out.println(newCustomer);

        System.out.println(Customer.getInstance() == newCustomer);

    }

}
```

```
name=John, age=31
false
```

破坏了单例性！

- 为了能在序列化过程仍能保持单例的特性，可以在Customer类中**添加一个readResolve()方法**，在该方法中直接返回Customer的单例对象

```
private Object readResolve() throws ObjectStreamException {
        return InstanceHolder.instatnce;
}
```

```
public class TestSerialSingleton {

    public static void main(String[] args)throws IOException, ClassNotFoundException {
        File file = new File("D:/TEMP/Customer.txt");
        ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(file));
        out.writeObject(Customer.getInstance()); // 保存单例对象
        out.close();

        ObjectInputStream oin = new ObjectInputStream(new FileInputStream(file));
        Object newCustomer = oin.readObject();
        oin.close();
        System.out.println(newCustomer);

        System.out.println(Customer.getInstance() == newCustomer);

    }

}
```

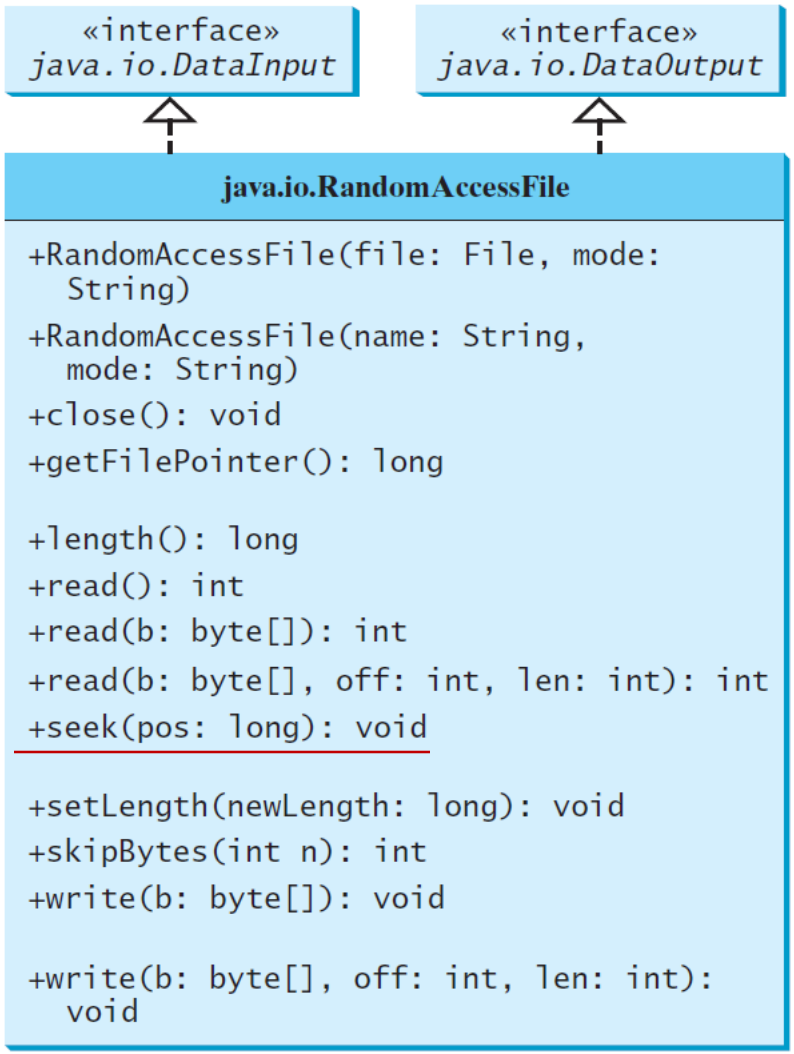
```
name=John, age=31
true
```

无论是实现Serializable接口，或是Externalizable接口，当从I/O流中读取对象时，readResolve()方法都会被调用到。实际上就是用readResolve()中返回的对象直接替换在反序列化过程中创建的对象。

Random Access Files

All of the streams you have used so far are known as *read-only* or *write-only* streams. The external files of these streams are *sequential* files that cannot be updated without creating a new file. It is often necessary to modify files or to insert new records into files. Java provides the RandomAccessFile class to allow a file to be read from and write to at random locations.

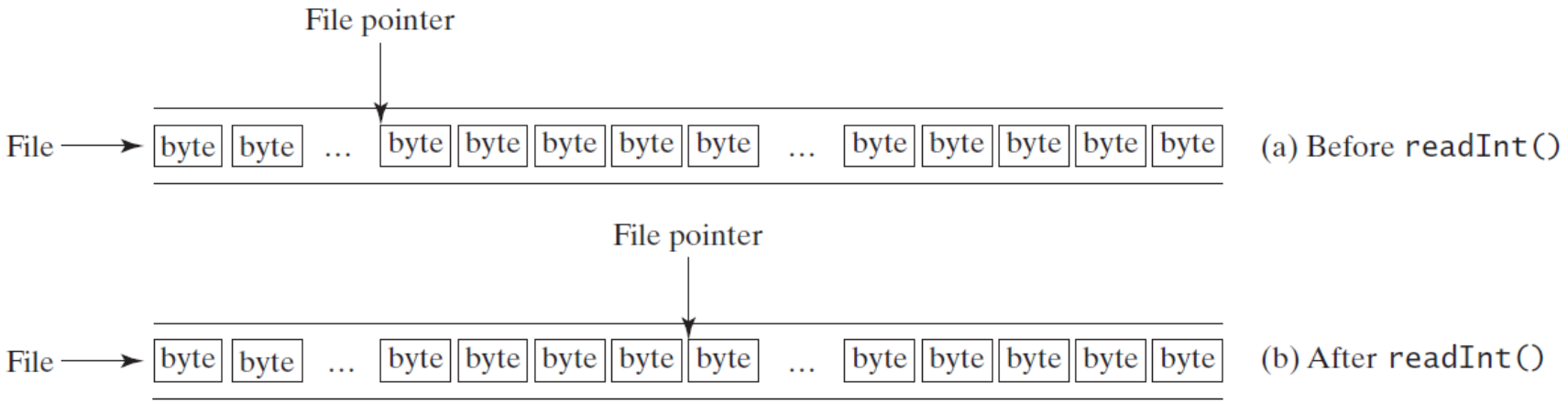
RandomAccessFile



`+RandomAccessFile(file: File, mode: String)`	Creates a `RandomAccessFile` stream with the specified `File` object and mode.
`+RandomAccessFile(name: String, mode: String)`	Creates a `RandomAccessFile` stream with the specified file name string and mode.
`+close(): void`	Closes the stream and releases the resource associated with it.
`+getFilePointer(): long`	Returns the offset, in bytes, from the beginning of the file to where the next `read` or `write` occurs.
`+length(): long`	Returns the number of bytes in this file.
`+read(): int`	Reads a byte of data from this file and returns –1 at the end of stream.
`+read(b: byte[]): int`	Reads up to `b.length` bytes of data from this file into an array of bytes.
`+read(b: byte[], off: int, len: int): int`	Reads up to `len` bytes of data from this file into an array of bytes.
`+seek(pos: long): void`	Sets the offset (in bytes specified in `pos`) from the beginning of the stream to where the next `read` or `write` occurs.
`+setLength(newLength: long): void`	Sets a new length for this file.
`+skipBytes(int n): int`	Skips over `n` bytes of input.
`+write(b: byte[]): void`	Writes `b.length` bytes from the specified byte array to this file, starting at the current file pointer.
`+write(b: byte[], off: int, len: int): void`	Writes `len` bytes from the specified byte array, starting at offset `off`, to this file.

File Pointer

A random access file consists of a sequence of bytes. There is a special marker called *file pointer* that is positioned at one of these bytes. A read or write operation takes place at the location of the file pointer. When a file is opened, the file pointer sets at the beginning of the file. When you read or write data to the file, the file pointer moves forward to the next data. For example, if you read an int value using readInt(), the JVM reads four bytes from the file pointer and now the file pointer is four bytes ahead of the previous location.



`RandomAccessFile` Methods

Many methods in `RandomAccessFile` are the same as those in `DataInputStream` and `DataOutputStream`. For example, `readInt()`, `readLong()`, `writeDouble()`, `readLine()`, `writeInt()`, and `writeLong()` can be used in data input stream or data output stream as well as in `RandomAccessFile` streams.

`RandomAccessFile` Methods, cont.

```
void seek(long pos) throws IOException;
```

Sets the offset from the beginning of the `RandomAccessFile` stream to where the next read or write occurs.

```
long getFilePointer() IOException;
```

Returns the current offset, in bytes, from the beginning of the file to where the next read or write occurs.

`RandomAccessFile` Methods, cont.

`long length()IOException`

Returns the length of the file.

`final void writeChar(int v) throws IOException`

Writes a character to the file as a two-byte Unicode, with the high byte written first.

`final void writeChars(String s) throws IOException`

Writes a string to the file as a sequence of characters.

RandomAccessFile Constructor

```
RandomAccessFile raf =
  new RandomAccessFile("test.dat", "rw");
  // allows read and write


RandomAccessFile raf =
  new RandomAccessFile("test.dat", "r");
  // read only
```

A Short Example on RandomAccessFile

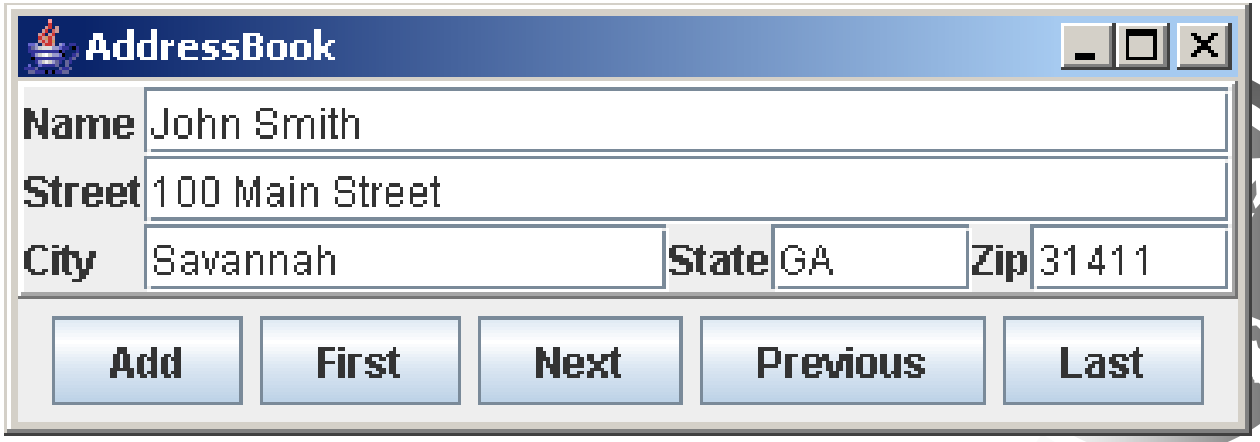
TestRandomAccessFile

Run

Companion Website

Case Studies: Address Book

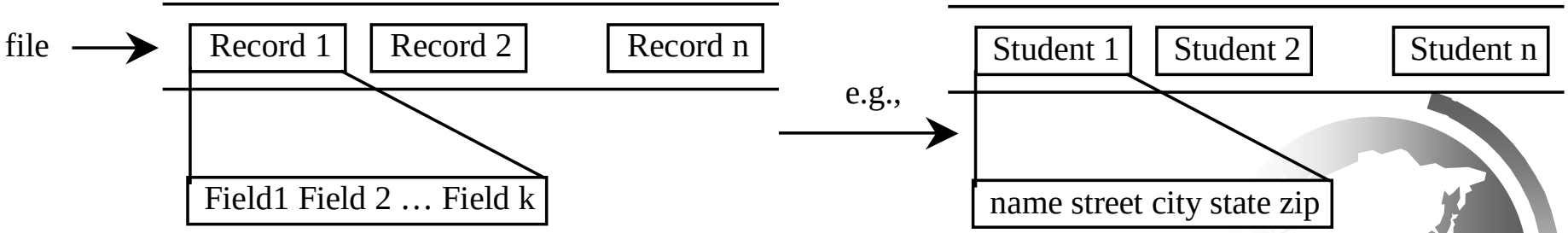
Now let us use RandomAccessFile to create a useful project for storing and viewing and address book. The *Add* button stores a new address to the end of the file. The *First*, *Next*, *Previous*, and *Last* buttons retrieve the first, next, previous, and last addresses from the file, respectively.



Companion Website

Fixed Length String I/O

Random access files are often used to process files of records. For convenience, fixed-length records are used in random access files so that a record can be located easily. A record consists of a fixed number of fields. A field can be a string or a primitive data type. A string in a fixed-length record has a maximum size. If a string is smaller than the maximum size, the rest of the string is padded with blanks.



FixedLengthStringIO

Companion Website

Address Implementation

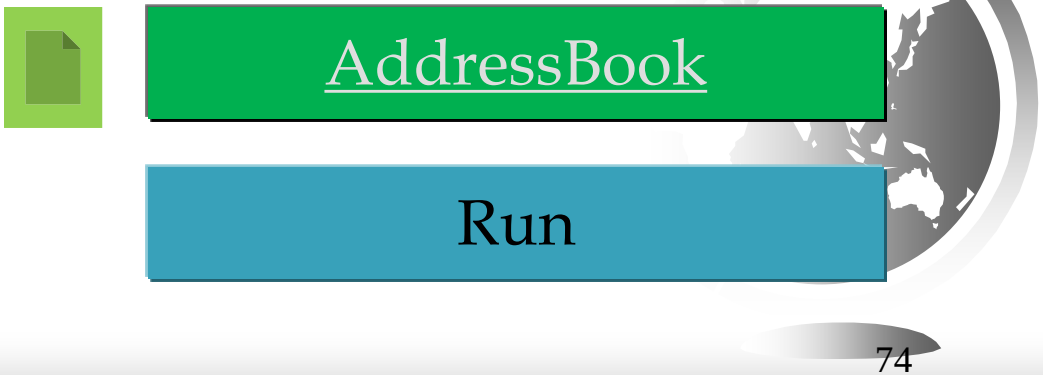
The rest of the work can be summarized in the following steps:

Create the user interface.

Add a record to the file.

Read a record from the file.

Write the code to implement the button actions.



```
/** Write a record at the end of the file */
public void writeAddress() {
  try {
    raf.seek(raf.length());
    FixedLengthStringIO.writeFixedLengthString(
      jtfName.getText(), NAME_SIZE, raf);
    FixedLengthStringIO.writeFixedLengthString(
      jtfStreet.getText(), STREET_SIZE, raf);
    FixedLengthStringIO.writeFixedLengthString(
      jtfCity.getText(), CITY_SIZE, raf);
    FixedLengthStringIO.writeFixedLengthString(
      jtfState.getText(), STATE_SIZE, raf);
    FixedLengthStringIO.writeFixedLengthString(
      jtfZip.getText(), ZIP_SIZE, raf);
  }
  catch (IOException ex) {
    ex.printStackTrace();
  }
}

/** Read a record at the specified position */
public void readAddress(long position) throws IOException {
  raf.seek(position);
  String name = FixedLengthStringIO.readFixedLengthString(
    NAME_SIZE, raf);
  String street = FixedLengthStringIO.readFixedLengthString(
    STREET_SIZE, raf);
  String city = FixedLengthStringIO.readFixedLengthString(
    CITY_SIZE, raf);
  String state = FixedLengthStringIO.readFixedLengthString(
    STATE_SIZE, raf);
  String zip = FixedLengthStringIO.readFixedLengthString(
    ZIP_SIZE, raf);
```

```
public class FixedLengthStringIO {
  /** Read fixed number of characters from a DataInput stream */
  public static String readFixedLengthString(int size,
      DataInput in) throws IOException {
    // Declare an array of characters
    char[] chars = new char[size];

    // Read fixed number of characters to the array
    for (int i = 0; i < size; i++)
      chars[i] = in.readChar();

    return new String(chars);
  }

  /** Write fixed number of characters to a DataOutput stream */
  public static void writeFixedLengthString(String s, int size,
      DataOutput out) throws IOException {
    char[] chars = new char[size];

    // Fill in string with characters
    s.getChars(0, Math.min(s.length(), size), chars, 0);

    // Fill in blank characters in the rest of the array
    for (int i = Math.min(s.length(), size); i < chars.length; i++)
      chars[i] = ' ';

    // Create and write a new string padded with blank characters
    out.writeChars(new String(chars));
  }
}
```