I/O Streams in Java

Produced by:

Eamonn de Leastar (edeleastar@wit.ie)

Dr. Siobhán Drohan (sdrohan@wit.ie)

Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

Essential Java

- **Overview**
  - Introduction
  - Syntax
  - Basics
  - Arrays
- **Classes**
  - Classes Structure
  - Static Members
  - Commonly used Classes
- **Control Statements**
  - Control Statement Types
  - If, else, switch
  - For, while, do-while

- **Inheritance**
  - Class hierarchies
  - Method lookup in Java
  - Use of this and super
  - Constructors and inheritance
  - Abstract classes and methods
  - Interfaces
- **Collections**
  - ArrayList
  - HashMap
  - Iterator
  - Vector
  - Enumeration
  - Hashtable

- **Exceptions**
  - Exception types
  - Exception Hierarchy
  - Catching exceptions
  - Throwing exceptions
  - Defining exceptions

  Common exceptions and errors
- **Streams**
  - Stream types
  - Character streams
  - Byte streams
  - Filter streams
  - Object Serialization

Road Map

- Introduction to I/O Streams
- Byte-oriented I/O Streams
- Character-oriented I/O Streams
- Layered I/O Streams (e.g. buffering)
- Line-oriented I/O Streams
- Scanning
- Data Streams
- Object Streams
- Pacemaker I/O
- Command Line I/O

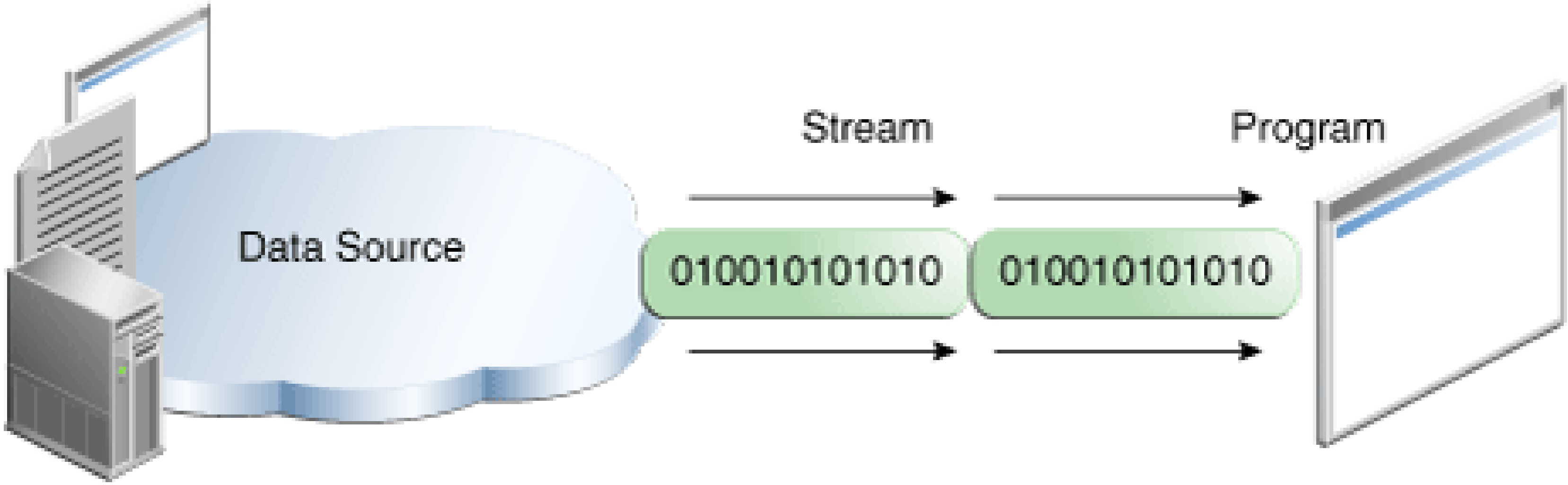
Introduction

- An I/O Stream represent a sequence of data; a one way, sequential flow of data.
- Conceptualise it as water flowing through a pipe.
- A stream can represent different kinds of sources (inputs) and destinations (outputs i.e. sinks) e.g.
  - disk files
  - devices
  - other programs
  - etc.

https://docs.oracle.com/javase/tutorial/essential/io/streams.html

Input Stream

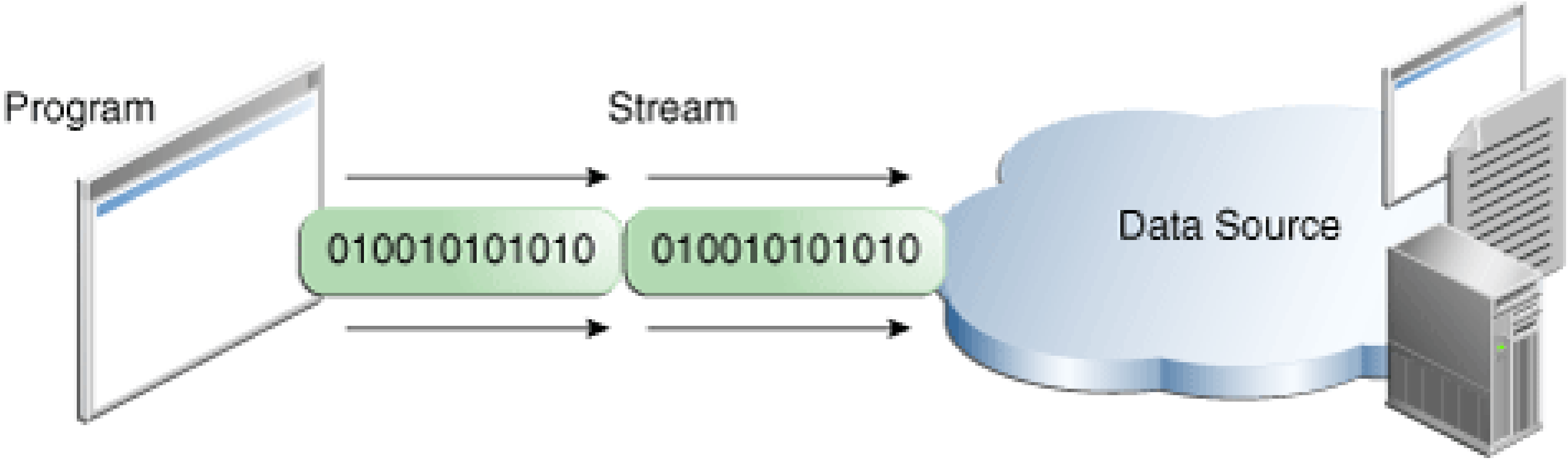
- A program uses an *input stream* to read data from a source, one item at a time:



https://docs.oracle.com/javase/tutorial/essential/io/streams.html

Output Stream

- A program uses an *output stream* to write data to a destination, one item at time:



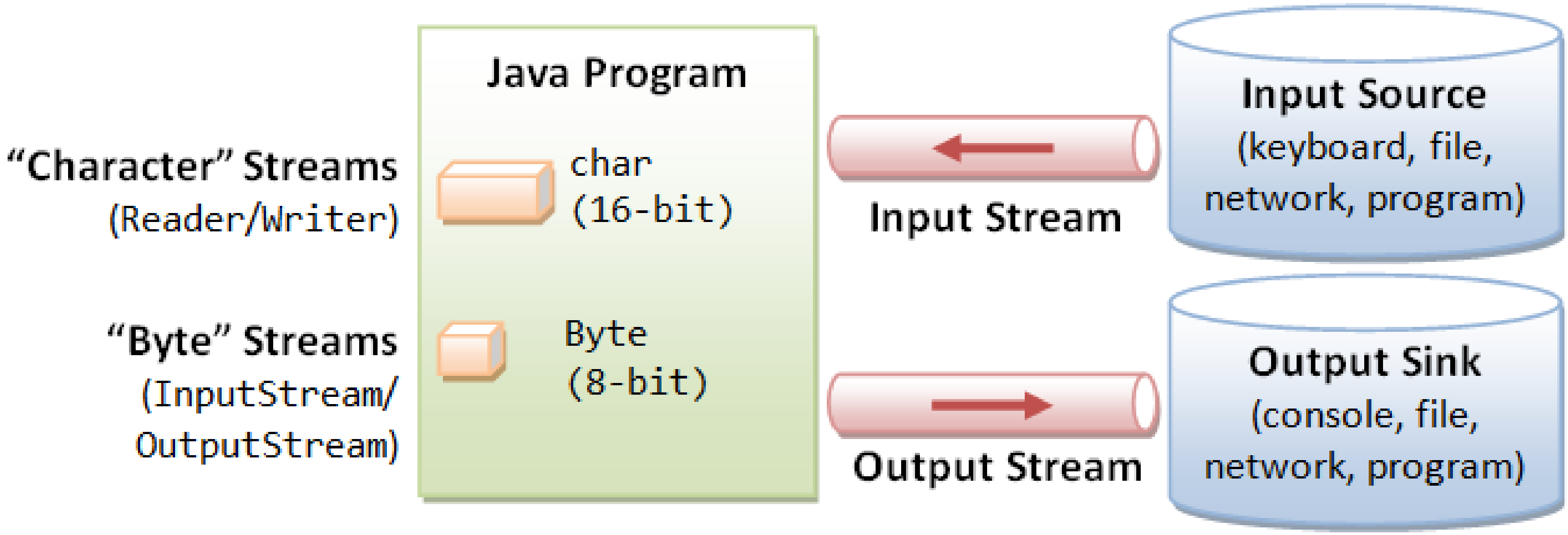
https://docs.oracle.com/javase/tutorial/essential/io/streams.html

Stream Data Types

- Streams support many different types of data
  - simple bytes
  - primitive data types
  - localized characters (サーバに関するお知らせ)
  - objects
- **java.io** package supports many different data types.

https://docs.oracle.com/javase/tutorial/essential/io/streams.html

I/O Stream



"Character" Streams
(Reader/Writer)

"Byte" Streams
(InputStream/
OutputStream)

Java Program

char
(16-bit)

Byte
(8-bit)

Input Stream

Output Stream

Input Source
(keyboard, file,
network, program)

Output Sink
(console, file,
network, program)

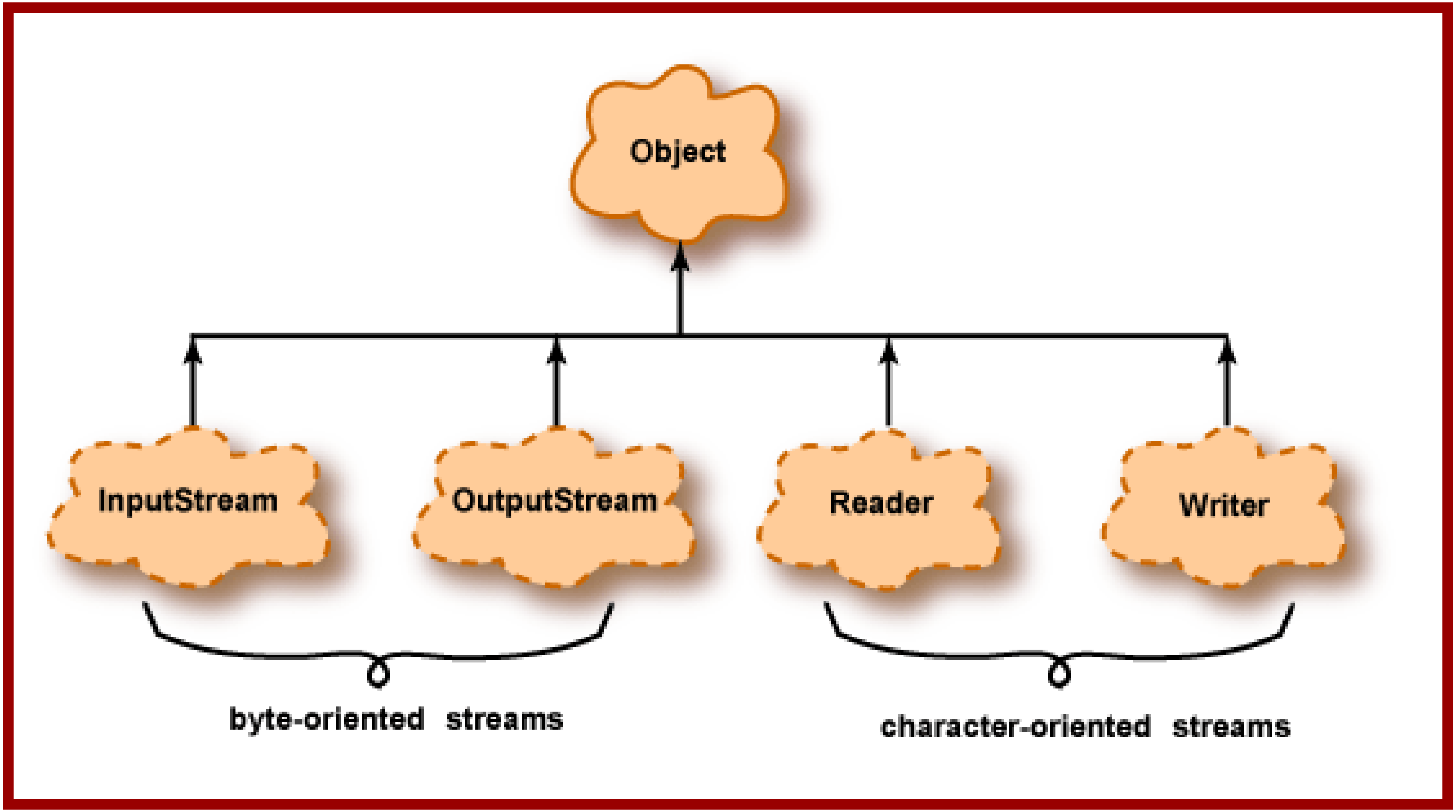
Internal Data Formats:

- Text (char): UCS-2
- int, float, double, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

Abstract classes in I/O Streams



http://chortle.ccsu.edu/java5/notes/chap82/ioHierarchyTop.gif

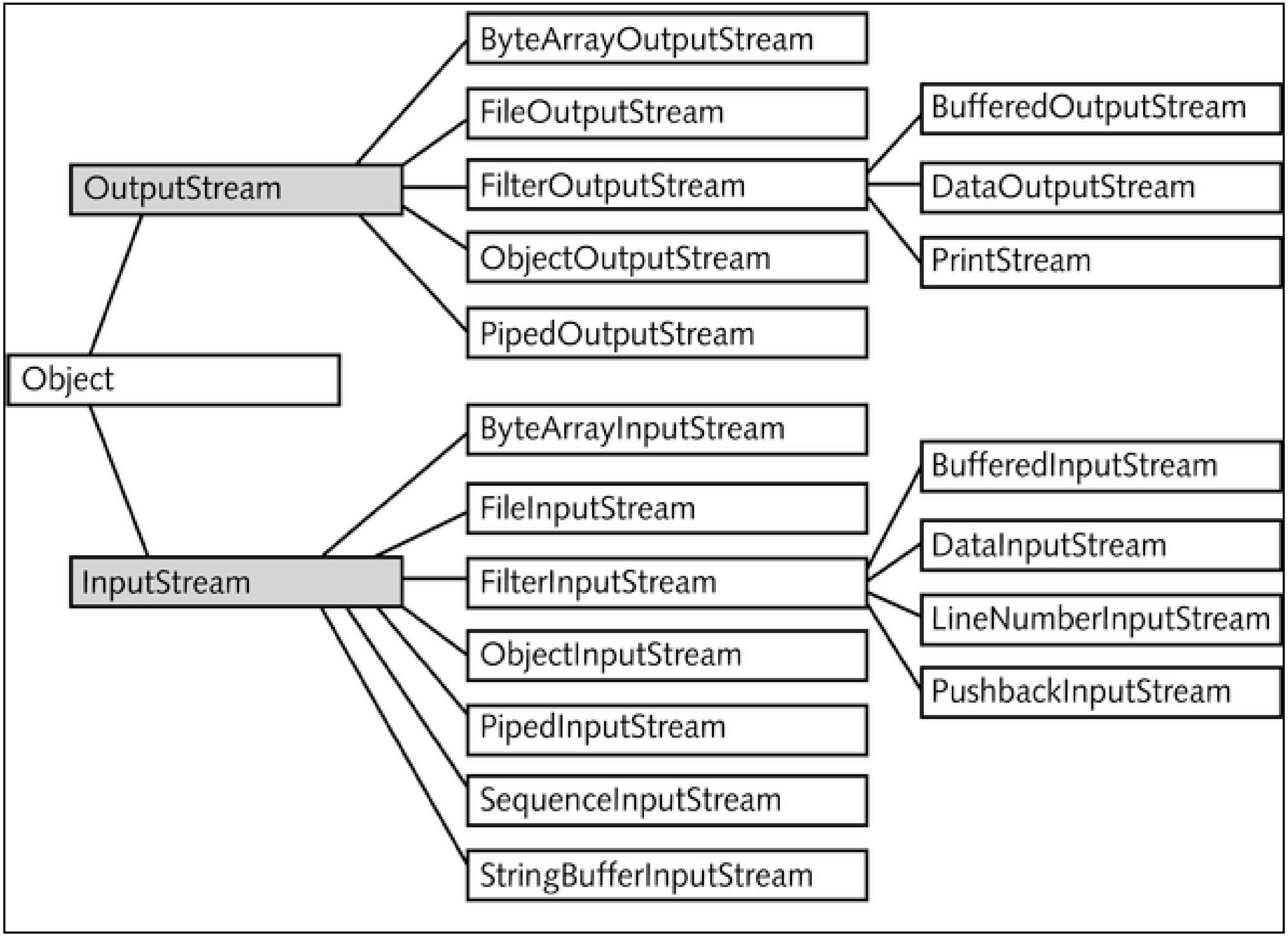
Road Map

- Introduction to I/O Streams
- Byte-oriented I/O Streams
- Character-oriented I/O Streams
- Layered I/O Streams (e.g. buffering)
- Line-oriented I/O Streams
- Scanning
- Data Streams
- Object Streams
- Pacemaker I/O
- Command Line I/O

Byte-oriented Streams

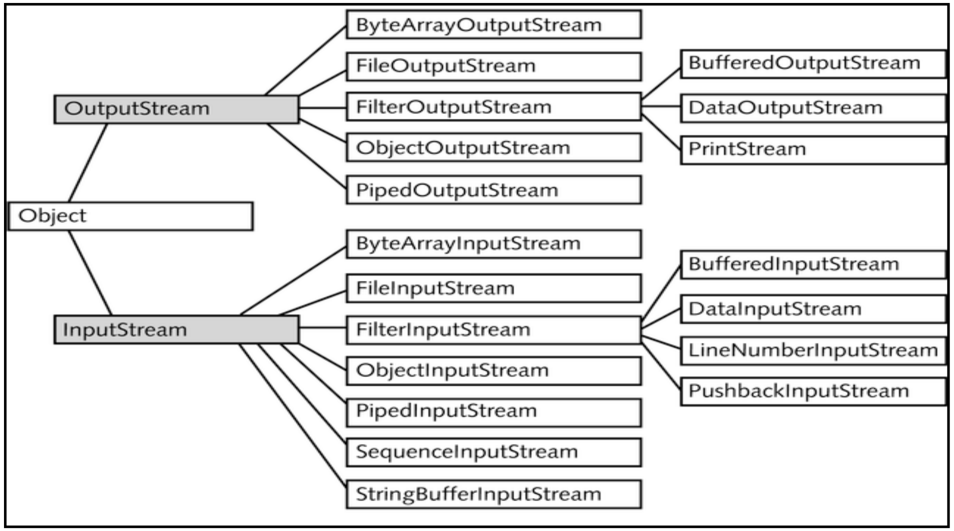
Programs use *byte streams* to perform input and output of 8-bit bytes.

Byte-oriented Streams Hierarchy



Byte Streams

- I/O of 8-bit bytes.
- InputStream & OutputStream are abstract, descendants are concrete.
- To read/write from files, use FileInputStream and FileOutputStream.
- Byte streams should only be used for the most primitive I/O.
- However, all other stream types are built on byte streams.



xanadu.txt

In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
through caverns measureless to man
Down to a sunless sea.

Sample file that we will use
to explain Byte Streams

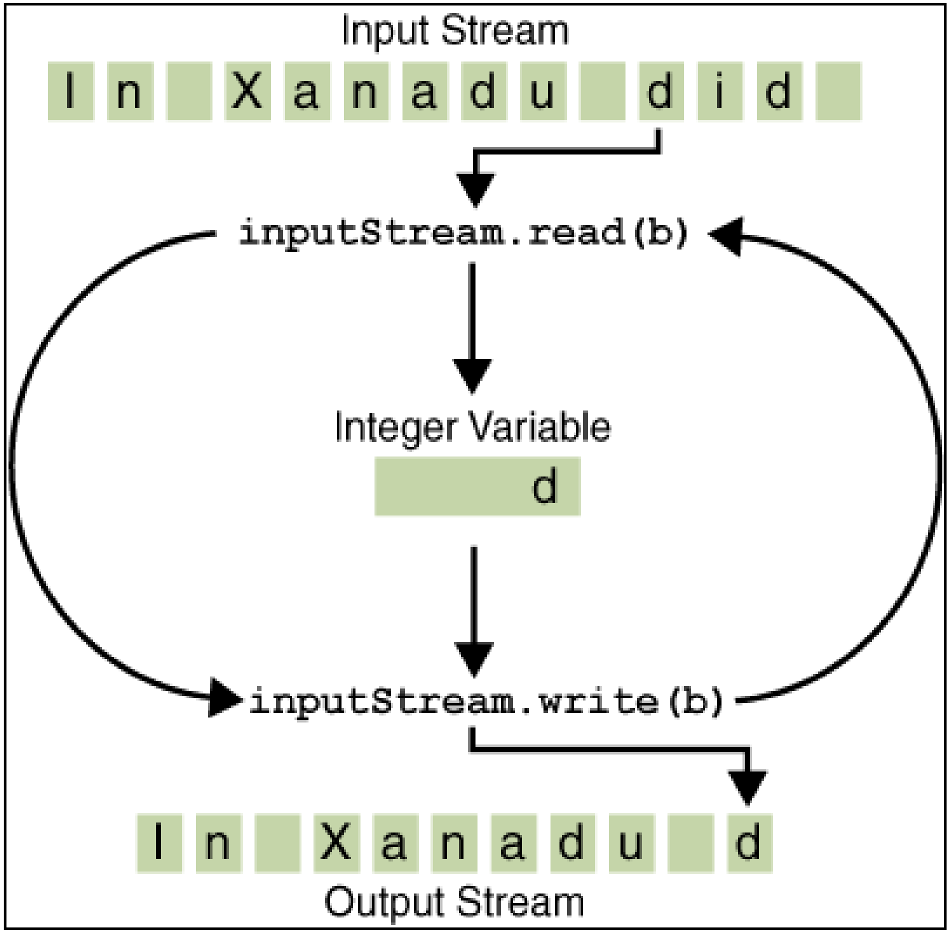
https://docs.oracle.com/javase/tutorial/essential/io/streams.html

Byte Stream I/O steps

- *Open* an input/output stream associated with a physical device.
- *Read* from the opened input stream until "end-of-stream" encountered, or *write* to the opened output stream.
- *Close* the input/output stream.

Byte Streams

In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
through caverns measureless to man
Down to a sunless sea.



Byte Streams – CopyBytes Example

```
public class CopyBytes
{
  public static void main(String[] args) throws IOException
  {
    FileInputStream in = null;
    FileOutputStream out = null;
    try{
      in = new FileInputStream("xanadu.txt");
      out = new FileOutputStream("outagain.txt");
      int c;
      while ((c = in.read()) != -1){
        out.write(c);
      }
    }
    finally{
      if (in != null){
        in.close();
      }
      if (out != null){
        out.close();
      }
    }
  }
}
```

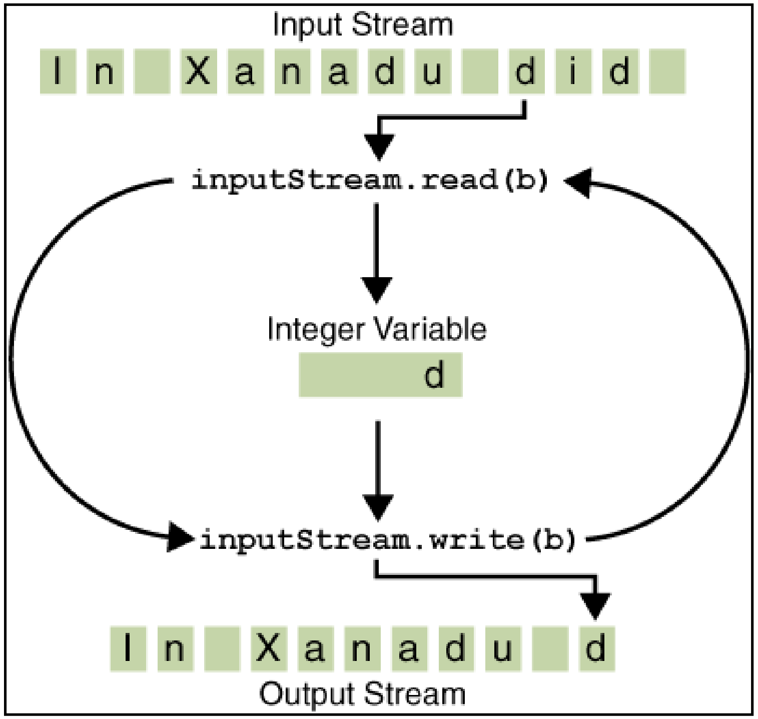
Byte Streams – CopyBytes Example

- An int return type allows read() to use -1 to indicate end of stream.

- A finally block is used to guarantee that both streams will be closed even if an error occurs; this helps avoid resource leaks.

- If Java was unable to open one or both files, the associated file stream variable won't from its initial null value; hence the test for null in the finally block.

- Java 7's *try-with-resources* would be useful here.



Before using try-with-resources

```
public class CopyBytes
{
  public static void main(String[] args) throws IOException
  {
    FileInputStream in = null;
    FileOutputStream out = null;
    try{
      in = new FileInputStream("xanadu.txt");
      out = new FileOutputStream("outagain.txt");
      int c;
      while ((c = in.read()) != -1){
        out.write(c);
      }
    }
    finally{
      if (in != null){
        in.close();
      }
      if (out != null){
        out.close();
      }
    }
  }
}
```

After using try-with-resources

```
public class CopyBytes
{
  public static void main(String[] args) throws IOException
  {
    try (FileInputStream  in  = new FileInputStream("xanadu.txt");
         FileOutputStream out = new FileOutputStream("outagain.txt"))
    {
      int c;
      while ((c = in.read()) != -1){
        out.write(c);
      }
    }
  }
}
```

try-with-resources is a new construct in Java 7.

When the try block finishes, the resources instantiated in the try clause are closed automatically.

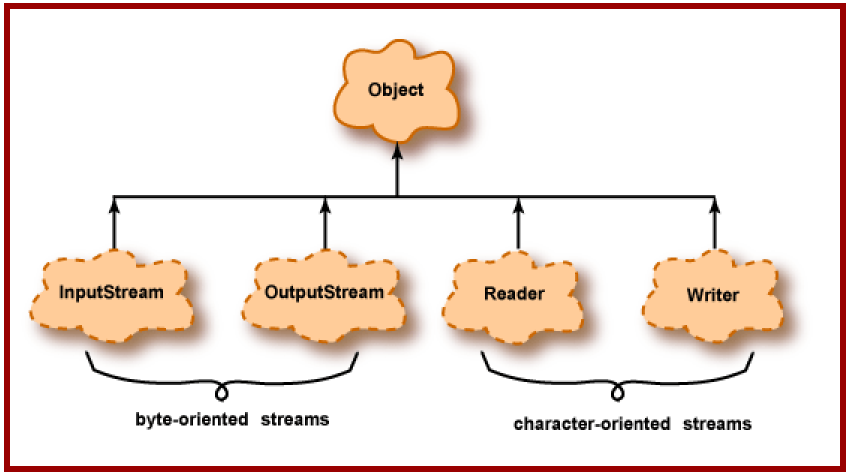
All classes implementing the java.lang.AutoCloseable interface can be used inside the try-with-resources construct.

Road Map

- Introduction to I/O Streams
- Byte-oriented I/O Streams
- Character-oriented I/O Streams
- Layered I/O Streams (e.g. buffering)
- Line-oriented I/O Streams
- Scanning
- Data Streams
- Object Streams
- Pacemaker I/O
- Command Line I/O

Character-oriented Streams

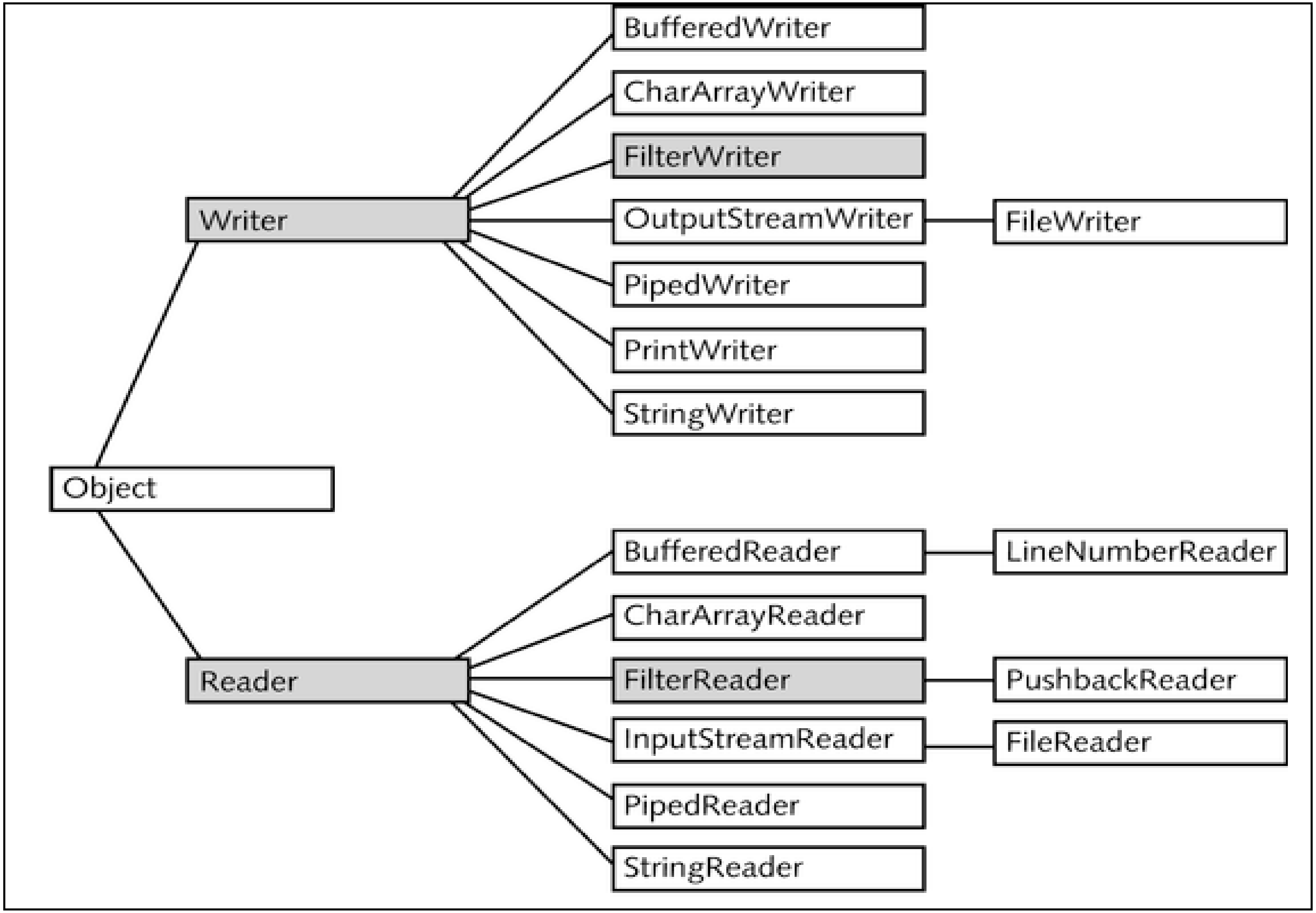
Programs use *character streams* to perform input and output of 16-bit bytes.



Character-oriented Streams

- Java internally stores characters (char types) using Unicode.
- But the external data source could store characters in other character sets e.g. US-ASCII, UTF-8, etc.
- Character stream I/O automatically translates Unicode character values to and from the local character set.
- Working with character streams is no more complicated than I/O with byte streams.

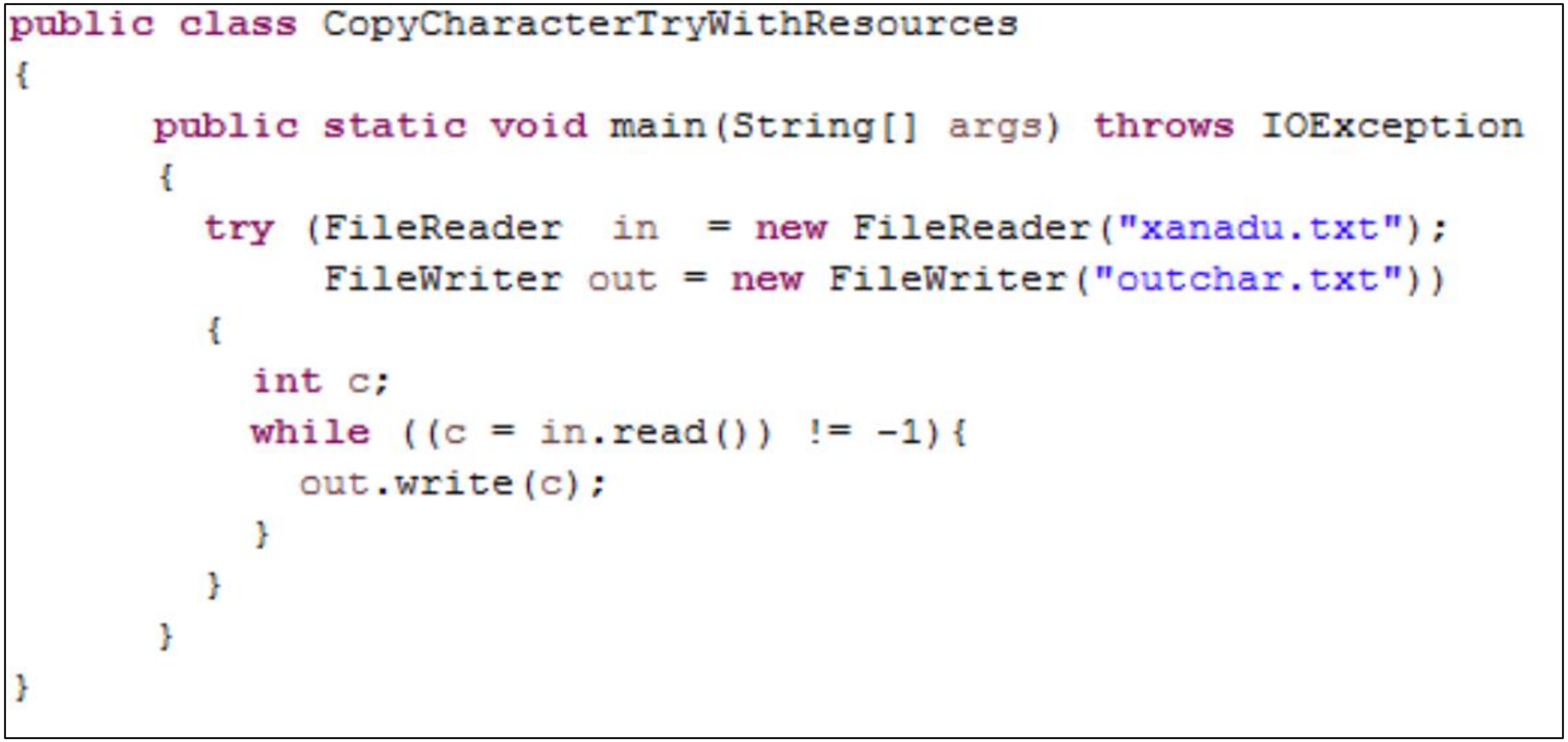
Character-oriented Streams



Character Stream – CopyCharacters Example

```
public class CopyCharacters
{
  public static void main(String[] args) throws IOException
  {
    FileReader in = null;
    FileWriter out = null;
    try{
      in = new FileReader("xanadu.txt");
      out = new FileWriter("outchar.txt");
      int c;
      while ((c = in.read()) != -1){
        out.write(c);
      }
    }
    finally{
      if (in != null){
        in.close();
      }
      if (out != null){
        out.close();
      }
    }
  }
}
```

CopyCharacters – using try-with-resources



```
public class CopyCharacterTryWithResources
{
      public static void main(String[] args) throws IOException
      {
        try (FileReader  in  = new FileReader("xanadu.txt");
             FileWriter out = new FileWriter("outchar.txt"))
        {
          int c;
          while ((c = in.read()) != -1){
            out.write(c);
          }
        }
      }
}
```

CopyCharacters vs CopyBytes

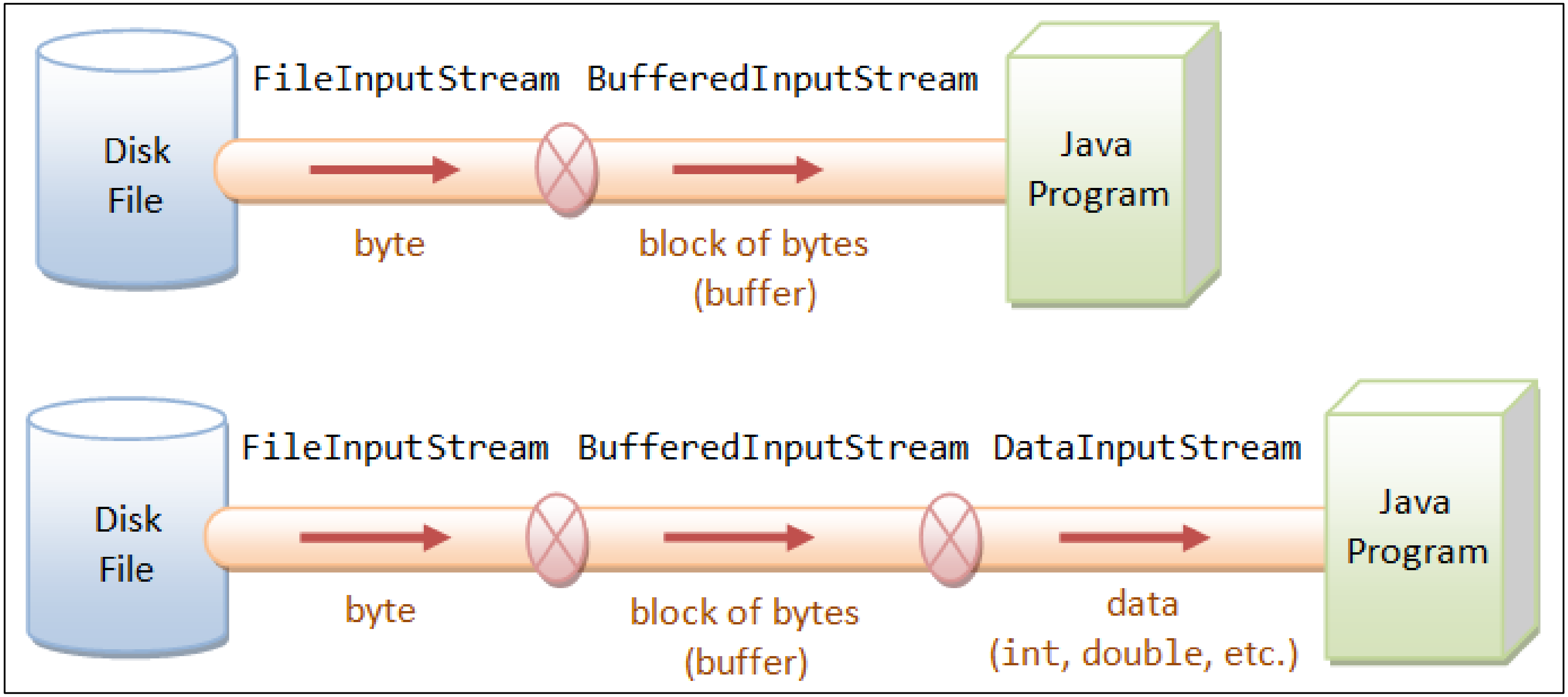
- CopyCharacters is very similar to CopyBytes.
  - CopyCharacters uses FileReader and FileWriter.
  - CopyBytes uses FileInputStream and FileOutputStream.
- Both use an int variable to read to and write from.
  - CopyCharacters int variable holds a character value in its last 16 bits
  - CopyBytes int variable holds a byte value in its last 8 bits
- Character streams are often "wrappers" for byte streams.
  - A byte stream to perform the physical I/O
  - The character stream handles translation between characters and bytes.
- e.g. FileReader uses FileInputStream, while
  FileWriter uses FileOutputStream.

Road Map

- Introduction to I/O Streams
- Byte-oriented I/O Streams
- Character-oriented I/O Streams
- Layered I/O Streams (e.g. buffering)
- Line-oriented I/O Streams
- Scanning
- Data Streams
- Object Streams
- Pacemaker I/O
- Command Line I/O

Layered I/O Streams

- I/O streams are often layered (chained) with other I/O streams e.g. for buffering, data-format conversion, etc.



http://www3.ntu.edu.sg/home/ehchua/programming/java/j5b_io.html

Buffered I/O

- So far, we have only looked at reading/writing a single character of data:

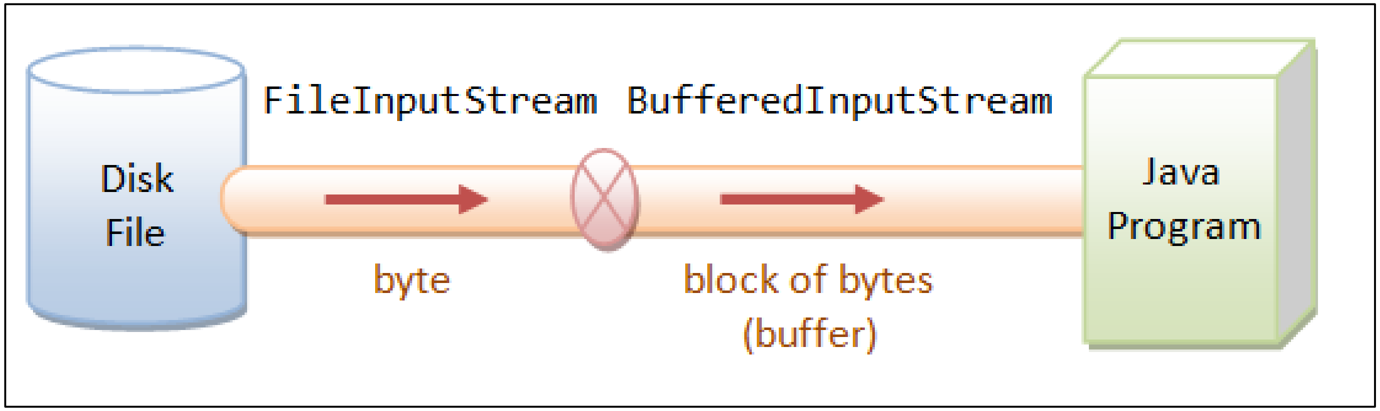
  → grossly inefficient e.g. each call can trigger a disk read/write.

- To speed up the I/O, we can read/write blocks of bytes into a memory buffer in one single I/O operation.

http://www3.ntu.edu.sg/home/ehchua/programming/java/j5b_io.html

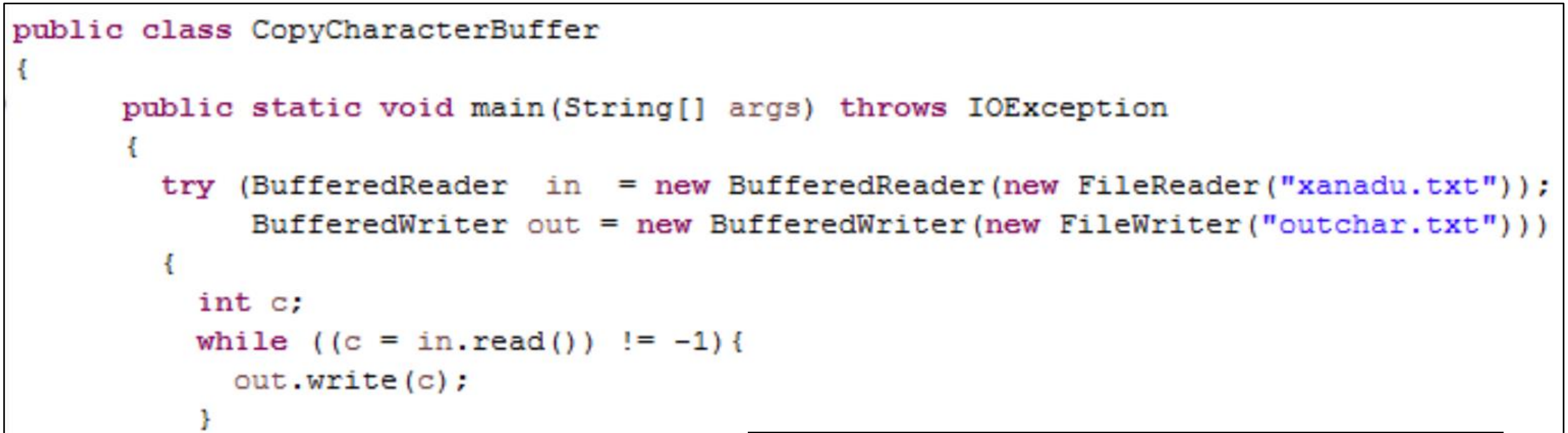
Buffered I/O

- FileInputStream/FileOutputStream is not buffered.
- **But** you can chain it to a BufferedInputStream/ BufferedOutputStream to provide the buffering.
- To chain streams, pass the instance of one stream to the constructor of another.

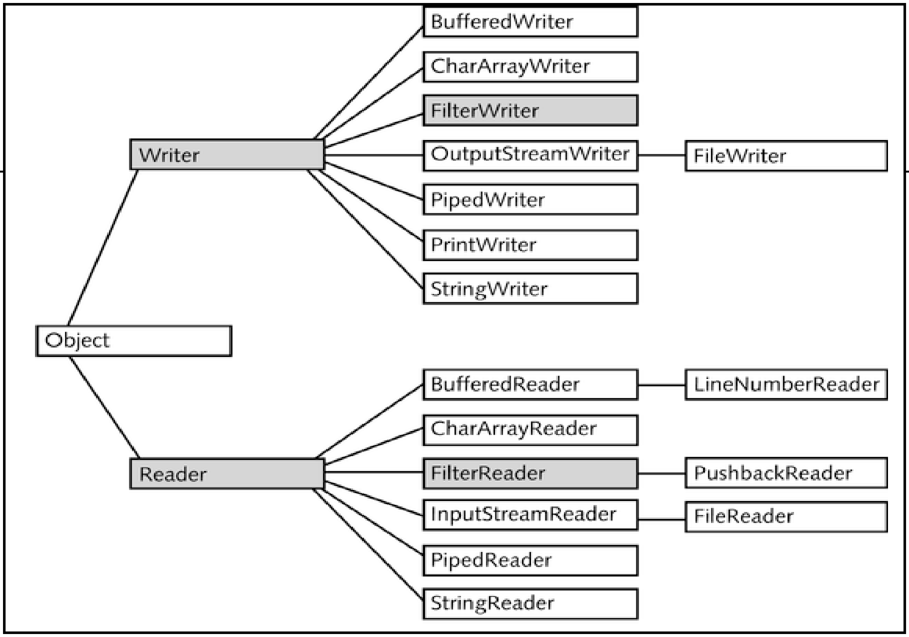


http://www3.ntu.edu.sg/home/ehchua/programming/java/j5b_io.html

Buffered I/O - CopyCharacter



```
public class CopyCharacterBuffer
{
      public static void main(String[] args) throws IOException
      {
        try (BufferedReader  in  = new BufferedReader(new FileReader("xanadu.txt"));
             BufferedWriter out = new BufferedWriter(new FileWriter("outchar.txt")))
        {
          int c;
          while ((c = in.read()) != -1){
            out.write(c);
          }
        }
      }
}
```



Object

Writer

BufferedWriter

CharArrayWriter

FilterWriter

OutputStreamWriter

FileWriter

PipedWriter

PrintWriter

StringWriter

Reader

BufferedReader

LineNumberReader

CharArrayReader

FilterReader

PushbackReader

InputStreamReader

FileReader

PipedReader

StringReader

Flushing Buffers

- There are four buffered stream classes used to wrap unbuffered streams:
  - BufferedInputStream and BufferedOutputStream for byte streams
  - BufferedReader and BufferedWriter for character streams
- It often makes sense to write out a buffer at critical points, without waiting for it to fill.
  - This is known as flushing the buffer.

Flushing Buffers

- Some buffered output classes support autoflush, specified by an optional constructor argument.

- When autoflush is enabled, certain key events cause the buffer to be flushed.
  - For example, an autoflush PrintWriter object flushes the buffer on every invocation of println or format.

- To flush a stream manually, invoke its flush method.

- The flush method is valid on any output stream (that implements Flushable interface), but has no effect unless the stream is buffered.

Road Map

- Introduction to I/O Streams
- Byte-oriented I/O Streams
- Character-oriented I/O Streams
- Layered I/O Streams (e.g. buffering)
- Line-oriented I/O Streams
- Scanning
- Data Streams
- Object Streams
- Pacemaker I/O
- Command Line I/O

Line-Oriented I/O

- Character I/O usually occurs in bigger units than single characters.
- One common unit is the line:
  - a string of characters with a line terminator at the end.
- A line terminator can be, depending on the OS:
  - a carriage-return and line-feed sequence ("\r\n")
  - a single carriage-return ("\r")
  - a single line-feed ("\n").

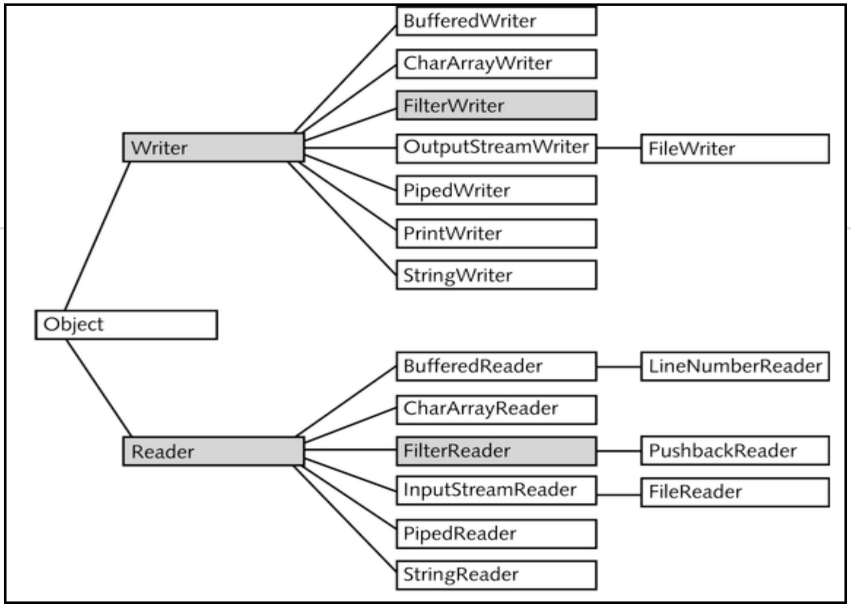
Line-Oriented I/O

- Supporting all possible line terminators, the **BufferedReader readLine()** method allows programs to read text files created on any of the widely used operating systems.

- **PrintWriter println(String)** method prints the String and then terminates the line; without chaining this stream, all output read with the BufferedReader would appear on one line in the output file.

CopyLines Example

```
public class CopyLines
{
  public static void main(String[] args) throws IOException
  {
    try(BufferedReader in =
               new BufferedReader(new FileReader("xanadu.txt"));
        PrintWriter out =
               new PrintWriter(new FileWriter("characteroutput.txt")))
    {
       String l;
       while ((l = in.readLine()) != null){
           out.println(l);
       }
    }
  }
}
```



BufferedWriter

- An unbuffered stream can be converted into a buffered stream using the wrapper idiom.
- The unbuffered stream object is passed to the constructor for a buffered stream class.

```
public static void main(String[] args) throws IOException
  {
    try(BufferedReader in =
               new BufferedReader(new FileReader("xanadu.txt"));
        PrintWriter out =
               new PrintWriter(
                   new BufferedWriter(
                      new FileWriter("characteroutput.txt"))))
    {
       String l;
       while ((l = in.readLine()) != null){
           out.println(l);
       }
    }
  }
```

Road Map

- Introduction to I/O Streams
- Byte-oriented I/O Streams
- Character-oriented I/O Streams
- Layered I/O Streams (e.g. buffering)
- Line-oriented I/O Streams
- Scanning
- Data Streams
- Object Streams
- Pacemaker I/O
- Command Line I/O

Scanning

- Programming I/O often involves translating to and from the neatly formatted data humans like to work with.
  - **Scanner** objects can be useful for breaking input into individual tokens according to their data type.

Scanning

- By default, a Scanner uses white space to separate tokens.

- To use a different token separator, invoke useDelimiter(), specifying a regular expression (i.e. a sequence of symbols and characters expressing a string/pattern).

- Even though a scanner is not a stream, you need to close it to indicate that you're done with its underlying stream.

ScanFile

```
public class ScanFile
{
  public static void main(String[] args) throws IOException
  {
    Scanner s = null;
    try
    {
      s = new Scanner(new BufferedReader(
                                  new FileReader("xanadu.txt")));
      while (s.hasNext())
      {
        System.out.println(s.next());
      }
    }
    finally
    {
      if (s != null)
      {
        s.close();
      }
    }
  }
}
```

This class reads in the individual words in the xanadu.txt file and prints them out to the console, one per line.

Translating Individual Tokens

```
public class ScanSum
{
  public static void main(String[] args) throws IOException
  {
    Scanner s = null;
    double sum = 0;

    try{
      s = new Scanner(new BufferedReader(new FileReader("usnumbers.txt")));

      while (s.hasNext()){
        if (s.hasNextDouble()){
          sum += s.nextDouble();
        }
        else{
          s.next();
        }
      }
    }
    finally{
      s.close();
    }
    System.out.println(sum);
  }
}
```

usnumbers.txt

```
45
3
4
6
rogue text
8.4
3
more rogue text
6.46
```

Console output

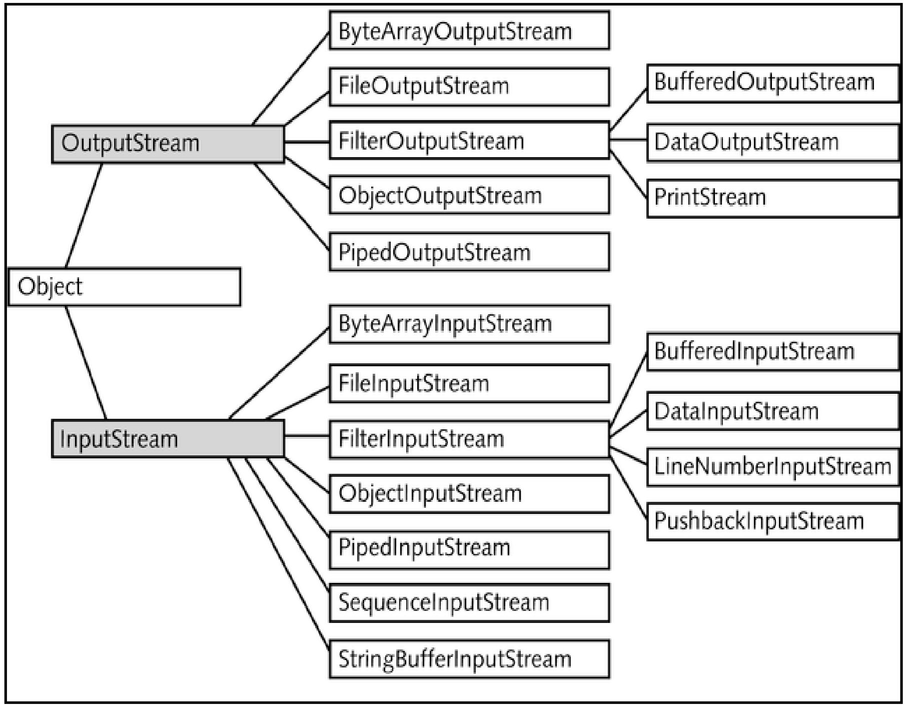
```
75.86
```

Road Map

- Introduction to I/O Streams
- Byte-oriented I/O Streams
- Character-oriented I/O Streams
- Layered I/O Streams (e.g. buffering)
- Line-oriented I/O Streams
- Scanning
- Data Streams
- Object Streams
- Pacemaker I/O
- Command Line I/O

Data Streams

- Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values.
- All data streams implement either the DataInput interface or the DataOutput interface.
- The most widely-used implementations of these interfaces are DataInputStream and DataOutputStream.



DataStream (1)

```java
public class DataStream
{
  static final String dataFile = "invoicedata";
  static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
  static final int[] units     = { 12, 8, 13, 29, 50 };
  static final String[] descs = { "Java T-shirt", "Java Mug",
                                  "Duke Juggling Dolls",
                                  "Java Pin", "Java Key Chain"};

  public static void main(String[] args) throws IOException
  {
    DataOutputStream out = new DataOutputStream(
             new BufferedOutputStream(new FileOutputStream(dataFile)));

    for (int i = 0; i < prices.length; i++)
    {
      out.writeDouble(prices[i]);
      out.writeInt(units[i]);
      out.writeUTF(descs[i]);
    }
    out.close();

    //…continued
```

DataStream (2)

```
    DataInputStream in = new DataInputStream(
                           new BufferedInputStream(
                             new FileInputStream(dataFile)));
    double price;
    int unit;
    String desc;
    double total = 0.0;
    try
    {
      while (true)
      {
        price = in.readDouble();
        unit = in.readInt();
        desc = in.readUTF();
        System.out.format("You ordered %d units of %s at $%.2f%n",
                                                          unit, desc, price);
        total += unit * price;
      }
    }
    catch (EOFException e)
    {
      System.out.println("End of file");
    }
  }
}
```

Data Streams Observations

- The writeUTF method writes out String values in a modified form of UTF-8.
  - A variable-width character encoding that only needs a single byte for common Western characters.
- Generally, we detect an end-of-file condition by catching EOFException, instead of testing for an invalid return value.
- Each specialized write in DataStreams is exactly matched by the corresponding specialized read.
- Floating point numbers not recommended for monetary values
  - In general, floating point is bad for precise values.
  - The correct type to use for currency values is java.math.BigDecimal.
- Unfortunately, BigDecimal is an object type, so it won't work with data streams – need Object Streams.

Road Map

- Introduction to I/O Streams
- Byte-oriented I/O Streams
- Character-oriented I/O Streams
- Layered I/O Streams (e.g. buffering)
- Line-oriented I/O Streams
- Scanning
- Data Streams
- Object Streams
- Pacemaker I/O
- Command Line I/O

Object Streams

- Data streams support I/O of primitive data types
- Object streams support I/O of objects
  - A class that can be serialized implements the marker interface Serializable.

- The object stream classes are ObjectInputStream and ObjectOutputStream.
  - An object stream can contain a mixture of primitive and object values

- If readObject() doesn't return the object type expected, attempting to cast it to the correct type may throw a ClassNotFoundException.

ObjectSteams

```
public class ObjectStreams
{
  static final String dataFile = "invoicedata";
  static final BigDecimal[] prices = {new BigDecimal("19.99"),
                                      new BigDecimal("9.99"),
                                      new BigDecimal("15.99"),
                                      new BigDecimal("3.99"),
                                      new BigDecimal("4.99") };
  static final int[] units = { 12, 8, 13, 29, 50 };
  static final String[] descs = { "Java T-shirt", "Java Mug",
                                  "Duke Juggling Dolls",
                                  "Java Pin", "Java Key Chain" };
  public static void main(String[] args)
                        throws IOException, ClassNotFoundException
  {
    ObjectOutputStream out = null;
    try
    {
      out = new ObjectOutputStream(
             new BufferedOutputStream(new FileOutputStream(dataFile)));
      out.writeObject(Calendar.getInstance());
      for (int i = 0; i < prices.length; i++)
      {
        out.writeObject(prices[i]);
        out.writeInt(units[i]);
        out.writeUTF(descs[i]);
      }
    }
    finally
    {
      out.close();
    }
  //…
  }
```

ObjectStreams(2)

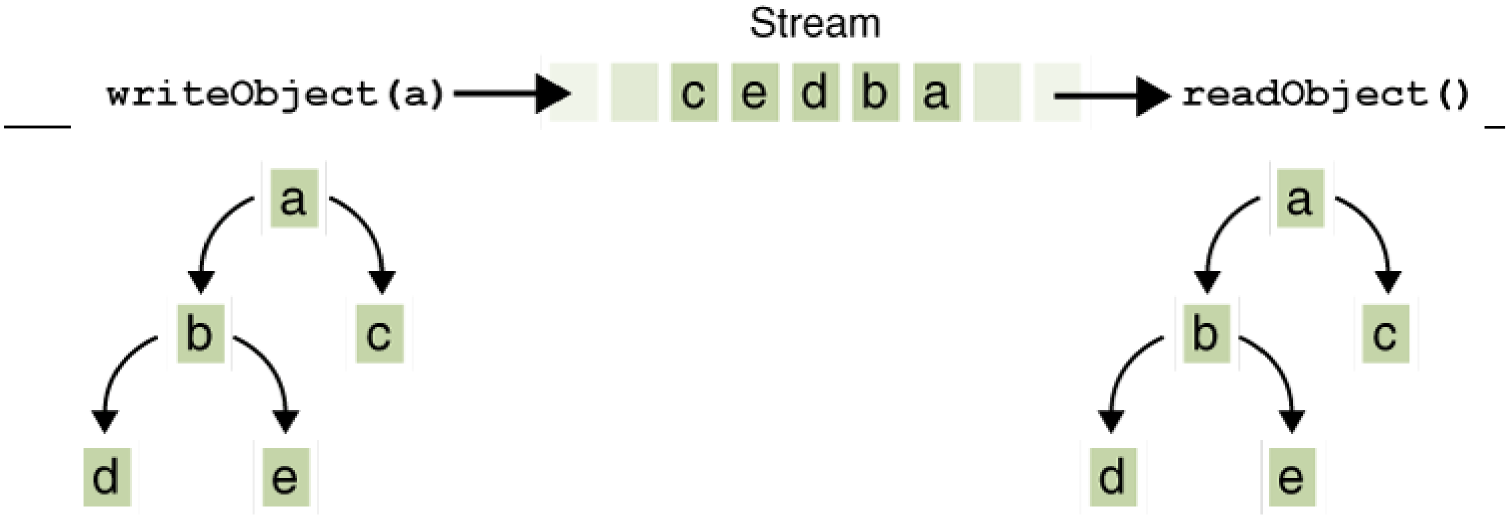
```
ObjectInputStream in = null;
try
{
  in = new ObjectInputStream(
         new BufferedInputStream(new FileInputStream(dataFile)));
  Calendar date = null;
  BigDecimal price;
  int unit;
  String desc;
  BigDecimal total = new BigDecimal(0);

  date = (Calendar) in.readObject();

  System.out.format("On %tA, %<tB %<te, %<tY:%n", date);
  try
  {
    while (true)
    {
      price = (BigDecimal) in.readObject();
      unit = in.readInt();
      desc = in.readUTF();
      System.out.format("You ordered %d units of %s at $%.2f%n",unit, desc, price);
      total = total.add(price.multiply(new BigDecimal(unit)));
    }
  }
  catch (EOFException e)
  {
  }
  System.out.format("For a TOTAL of: $%.2f%n", total);
}
finally
{
  in.close();
}
```

readObject() and writeObject()

- The writeObject and readObject methods contain some sophisticated object management logic.
- This is particularly important for objects that contain references to other objects.
- If readObject is to reconstitute an object from a stream, it has to be able to reconstitute all the objects the original object referred to.
  - These additional objects might have their own references, and so on.
- In this situation, writeObject traverses the entire web of object references and writes all objects in that web onto the stream. Thus a single invocation of writeObject can cause a large number of objects to be written to the stream.



- Suppose:
  - If writeObject is invoked to write a single object named a.
  - This object contains references to objects b and c,
  - while b contains references to d and e.
- Invoking writeobject(a) writes a and all the objects necessary to reconstitute a
- When a is read by readObject, the other four objects are read back as well, and all the original object references are preserved.

Road Map

- Introduction to I/O Streams
- Byte-oriented I/O Streams
- Character-oriented I/O Streams
- Layered I/O Streams (e.g. buffering)
- Line-oriented I/O Streams
- Scanning
- Data Streams
- Object Streams
- Pacemaker I/O – lab03 exercises
- Command Line I/O

Abstract the mechanism

```
package utils;

public interface Serializer
{
  void push(Object o);
  Object pop();
  void write() throws Exception;
  void read() throws Exception;
}
```

Defining this interface will allow us to build different serialization strategies e.g. XML, JSON, etc.

We can decide which to use at compile time, or at run time.

Different Serializers

```
public class XMLSerializer implements Serializer
{

  private Stack stack = new Stack();
  private File file;

  public XMLSerializer(File file)
  {
    this.file = file;
  }

//more code
```

```
public class BinarySerializer
                  implements Serializer
{

  private Stack stack = new Stack();
  private File file;

  public BinarySerializer(File file)
  {
    this.file = file;
  }
```

```
public class JSONSerializer implements Serializer
{

  private Stack stack = new Stack();
  private File file;

  public JSONSerializer(File file)
  {
    this.file = file;
  }

//more code
```

Deciding at compile time

```
public Main() throws Exception
 {
   //XML Serializer
   //File  datastore = new File("datastore.xml");
   //Serializer serializer = new XMLSerializer(datastore);

   //JSON Serializer
   //File  datastore = new File("datastore.json");
   //Serializer serializer = new JSONSerializer(datastore);

  //Binary Serializer
  File  datastore = new File("datastore.txt");
  Serializer serializer = new BinarySerializer(datastore);
```

Deciding at runtime

```
Welcome to pacemaker-console - ?help for instructions
pm> ?la
abbrev  name                              params
lu      list-users                        ()
cu      create-user                       (first name, last name, email, password)
lu      list-user                         (email)
lius    list-user                         (id)
la      list-activities   (userid, sortBy: type, location, distance, date,
duration)
la      list-activities   (user id)
du      delete-user                       (id)
aa      add-activity                      (user-id, type, location, distance,
datetime, duration)
al      add-location                      (activity-id, latitude, longitude)
cff     change-file-format                (file format: xml, json)
l       load                              ()
s       store                             ()
pm>
```

Binary Strategy

```
public class BinarySerializer implements ISerializationStrategy
{
  public Object read(String filename) throws Exception
  {
    ObjectInputStream is = null;
    Object obj = null;

    try
    {
      is = new ObjectInputStream(new BufferedInputStream(
                                             new FileInputStream(filename)));
      obj = is.readObject();
    }
    finally
    {
      if (is != null)
      {
        is.close();
      }
    }
    return obj;
  }
  //..
}
```

Binary Strategy (contd.)

```
public class BinarySerializer implements ISerializationStrategy
{
 //..

  public void write(String filename, Object obj) throws Exception
  {
    ObjectOutputStream os = null;
    try
    {
      os = new ObjectOutputStream(new BufferedOutputStream(
                                            new FileOutputStream(filename)));
      os.writeObject(obj);
    }
    finally
    {
      if (os != null)
      {
        os.close();
      }
    }
  }
}
```

XML Strategy

```
public class XMLSerializer implements ISerializationStrategy
{
  public Object read(String filename) throws Exception
  {
    ObjectInputStream is = null;
    Object obj = null;

    try
    {
      XStream xstream = new XStream(new DomDriver());
      is = xstream.createObjectInputStream(new FileReader(filename));
      obj = is.readObject();
    }
    finally
    {
      if (is != null)
      {
        is.close();
      }
    }
    return obj;
  }
  //...
}
```

XML Strategy (contd.)

```
public class XMLSerializer implements ISerializationStrategy
{
  //...
  public void write(String filename, Object obj) throws Exception
  {
    ObjectOutputStream os = null;

    try
    {
      XStream xstream = new XStream(new DomDriver());
      os = xstream.createObjectOutputStream(new FileWriter(filename));
      os.writeObject(obj);
    }
    finally
    {
      if (os != null)
      {
        os.close();
      }
    }

  }
}
```

Road Map

- Introduction to I/O Streams
- Byte-oriented I/O Streams
- Character-oriented I/O Streams
- Layered I/O Streams (e.g. buffering)
- Line-oriented I/O Streams
- Scanning
- Data Streams
- Object Streams
- Pacemaker I/O
- Command Line I/O

Command Line I/O

- A program is often run from the command line, and interacts with the user in the command line environment.

- The Java platform supports this kind of interaction in two ways:
  - Standard Streams
  - Console

Standard Streams

- A feature of many operating systems, they read input from the keyboard and write output to the display.
- They also support I/O on files and between programs.
- The Java platform supports three Standard Streams:
  - Standard Input, accessed through System.in;
  - Standard Output, accessed through System.out;
  - Standard Error, accessed through System.err.
- These objects are defined automatically (do not need to be opened)
- Standard Output and Standard Error are both for output
- Having error output separately allows the user to divert regular output to a file and still be able to read error messages.

System.in, System.out, System.err

- For historical reasons, the standard streams are byte streams (more logically character streams).
- System.out and System.err are defined as PrintStream objects.
- Although it is technically a byte stream, PrintStream utilises an internal character stream object to emulate many of the features of character streams.
- By contrast, System.in is a byte stream with no character stream features.
- To utilise Standard Input as a character stream, wrap System.in in InputStreamReader.

```
InputStreamReader cin = new InputStreamReader(System.in);
```

Console

- New in Java 6 - a more advanced alternative to the Standard Streams
- This is a single pre-defined object of type Console that has most of the features provided by the Standard Streams.
- The Console object also provides input and output streams that are true character streams, through its reader and writer methods.
- Before a program can use the Console, it must attempt to retrieve the Console object by invoking System.console().
  - If the Console object is available, this method returns it.
  - If it returns NULL, then Console operations are not permitted, either because the OS doesn't support them, or because the program was launched in a non-interactive environment.

Password Entry

- The Console object supports secure password entry through its readPassword method.

- This method helps secure password entry in two ways:
  - It suppresses echoing, so the password is not visible on the users screen.
  - readPassword returns a character array, not a String, so that the password can be overwritten, removing it from memory as soon as it is no longer needed.

Password (1)

```
public class Password
{
  public static void main(String[] args) throws IOException
  {
    Console c = System.console();

    if (c == null)
    {
      System.err.println("No console.");
      System.exit(1);
    }

    String login = c.readLine("Enter your login: ");
    char[] oldPassword = c.readPassword("Enter your old password: ");
    //..

  }
}
```

Password (2)

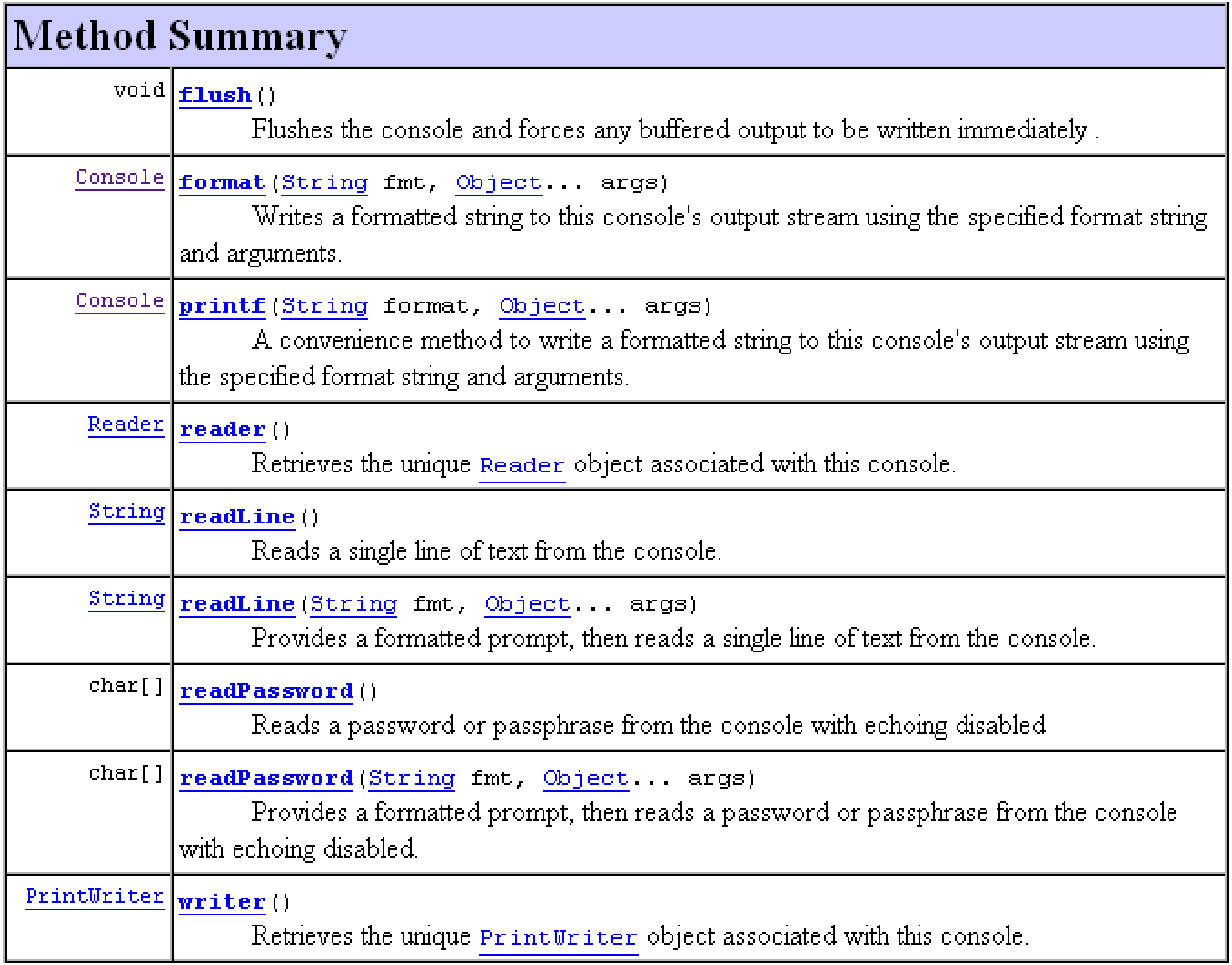
```
//..
if (verify(login, oldPassword))
  {
    boolean noMatch;
    do
    {
      char[] newPassword1 = c.readPassword("Enter your new password: ");
      char[] newPassword2 = c.readPassword("Enter new password again: ");
      noMatch = !Arrays.equals(newPassword1, newPassword2);
      if (noMatch)
      {
        c.format("Passwords don't match. Try again.%n");
      }
      else
      {
        change(login, newPassword1);
        c.format("Password for %s changed.%n", login);
      }

      Arrays.fill(newPassword1, ' ');
      Arrays.fill(newPassword2, ' ');
    }
    while (noMatch);
  }
  Arrays.fill(oldPassword, ' ');
}
```

format method

```
System.out.format("The value of "
     + "the float variable is "
     + "%f, while the value of the "
     + "integer variable is %d, "
     + "and the string is %s",
     floatVar, intVar, stringVar);
```

- Format specifiers begin with a percent sign (%) and end with a *converter*.



Method Summary	
void	**flush**() Flushes the console and forces any buffered output to be written immediately .
Console	**format**(String fmt, Object... args) Writes a formatted string to this console's output stream using the specified format string and arguments.
Console	**printf**(String format, Object... args) A convenience method to write a formatted string to this console's output stream using the specified format string and arguments.
Reader	**reader**() Retrieves the unique Reader object associated with this console.
String	**readLine**() Reads a single line of text from the console.
String	**readLine**(String fmt, Object... args) Provides a formatted prompt, then reads a single line of text from the console.
char[]	**readPassword**() Reads a password or passphrase from the console with echoing disabled
char[]	**readPassword**(String fmt, Object... args) Provides a formatted prompt, then reads a password or passphrase from the console with echoing disabled.
PrintWriter	**writer**() Retrieves the unique PrintWriter object associated with this console.

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning
support unit