

Agile Software Development

Produced
by

Eamonn de Leastar (edeleastar@wit.ie)



Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Streams

Java Language

Java Language

java	javac	javadoc	jar	javap	jdeps	Scripting
------	-------	---------	-----	-------	-------	-----------

Tools &
Tool APIs

Security	Monitoring	JConsole	VisualVM	JMC	JFR
JPDA	JVM TI	IDL	RMI	Java DB	Deployment
Internationalization		Web Services		Troubleshooting	

Deployment

Java Web Start

Applet / Java Plug-in

JavaFX

User Interface
Toolkits

Swing

Java 2D

AWT

Accessibility

Drag and Drop

Input Methods

Image I/O

Print Service

Sound

Integration
Libraries

IDL

JDBC

JNDI

RMI

RMI-IIOP

Scripting

Beans

Security

Serialization

Extension Mechanism

JMX

XML JAXP

Networking

Override Mechanism

JNI

Date and Time

Input/Output

Internationalization

lang and util

Math

Collections

Ref Objects

Regular Expressions

Logging

Management

Instrumentation

Concurrency Utilities

Reflection

Versioning

Preferences API

JAR

Zip

Java Virtual Machine

Java HotSpot Client and Server VM

Java SE
API

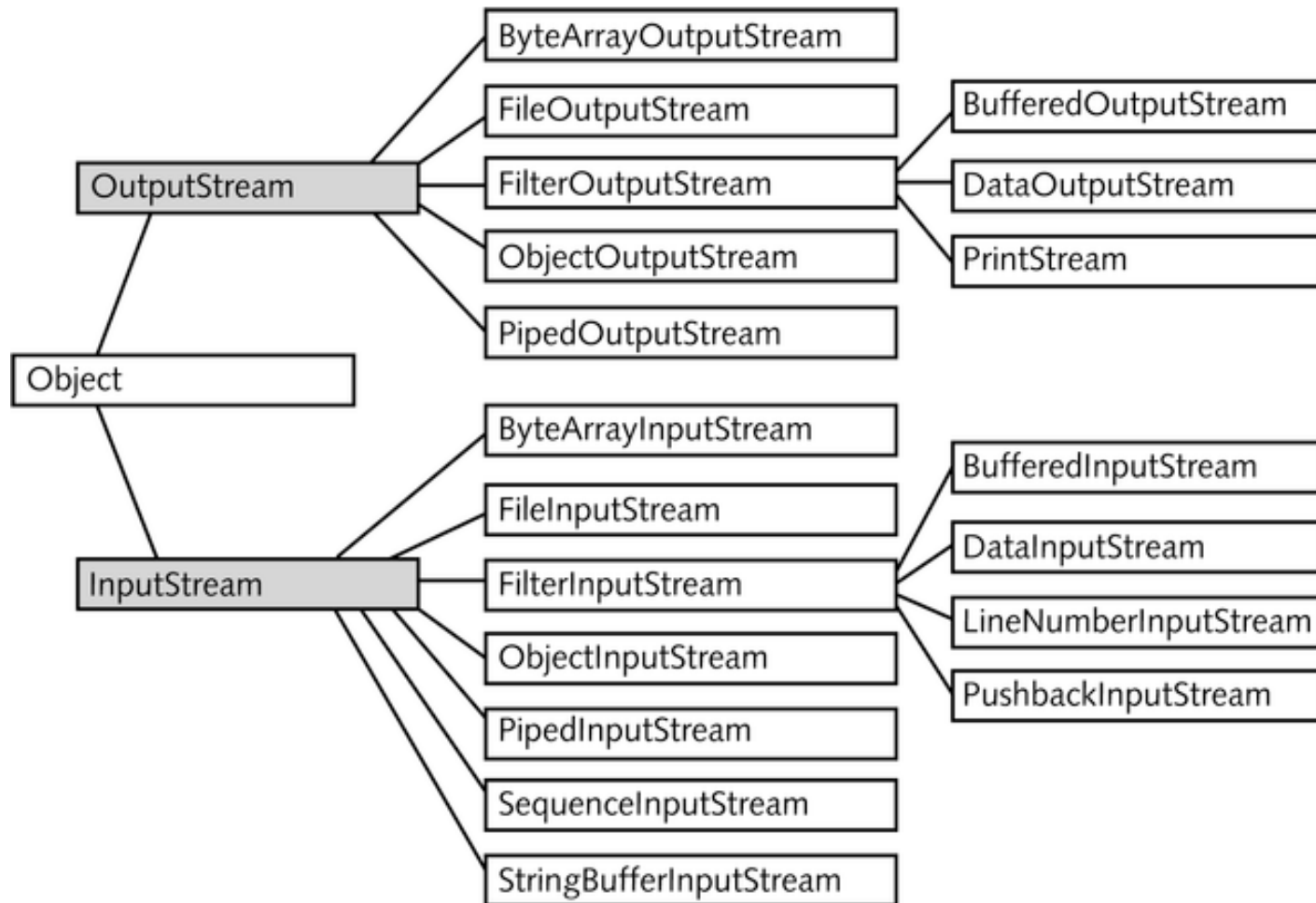
Compact
Profiles

<http://www.oracle.com/technetwork/java/javase/tech/index.html>

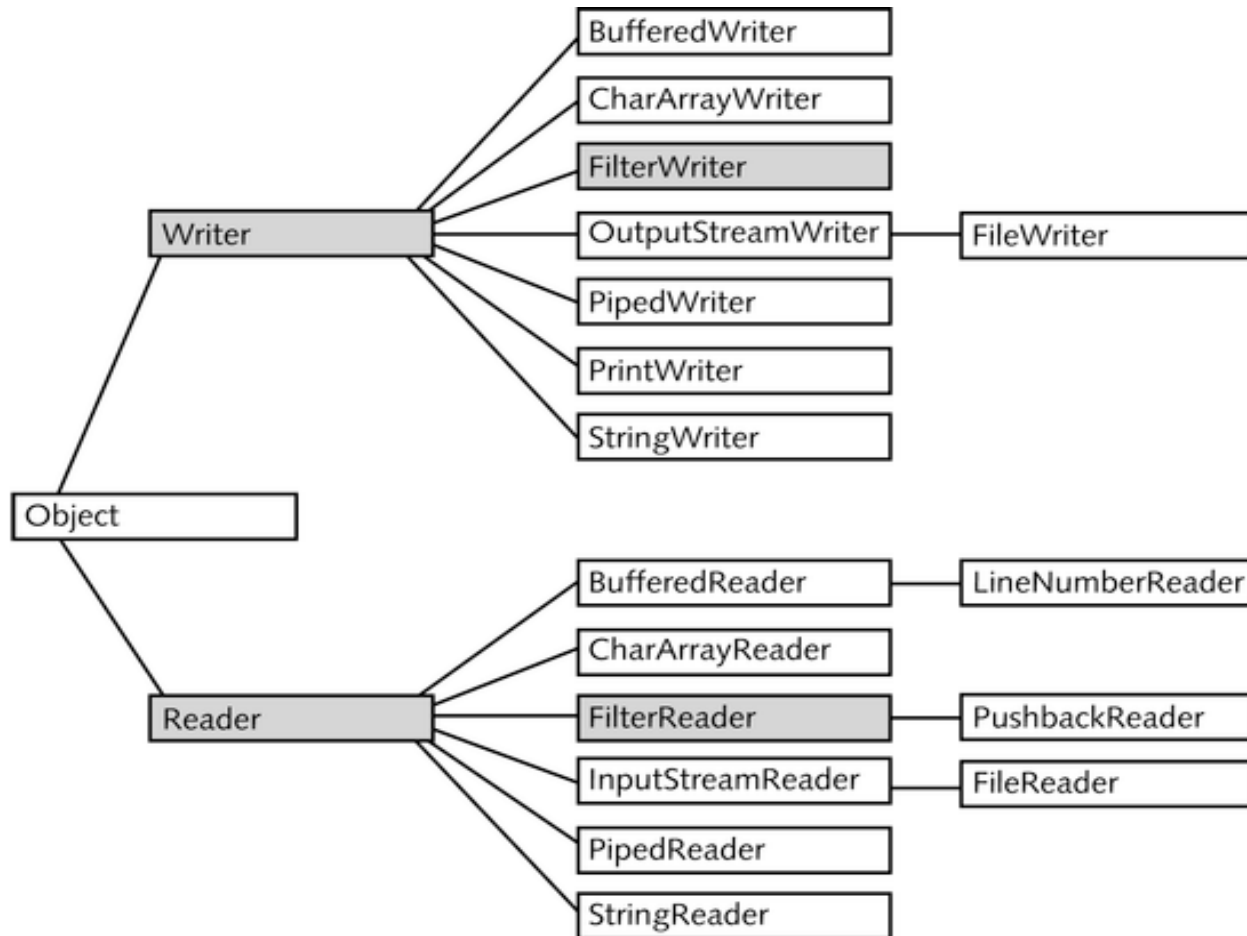
Introduction

- ⊕ An I/O Stream represents an input source or an output destination.
- ⊕ A stream can represent
 - ⊕ disk files
 - ⊕ devices
 - ⊕ other programs
- ⊕ Streams support
 - ⊕ simple bytes
 - ⊕ primitive data types
 - ⊕ localized characters (サーバに関するお知らせ)
 - ⊕ objects
- ⊕ Some streams simply pass on data, others manipulate and transform the data in useful ways.

Byte-Oriented Streams

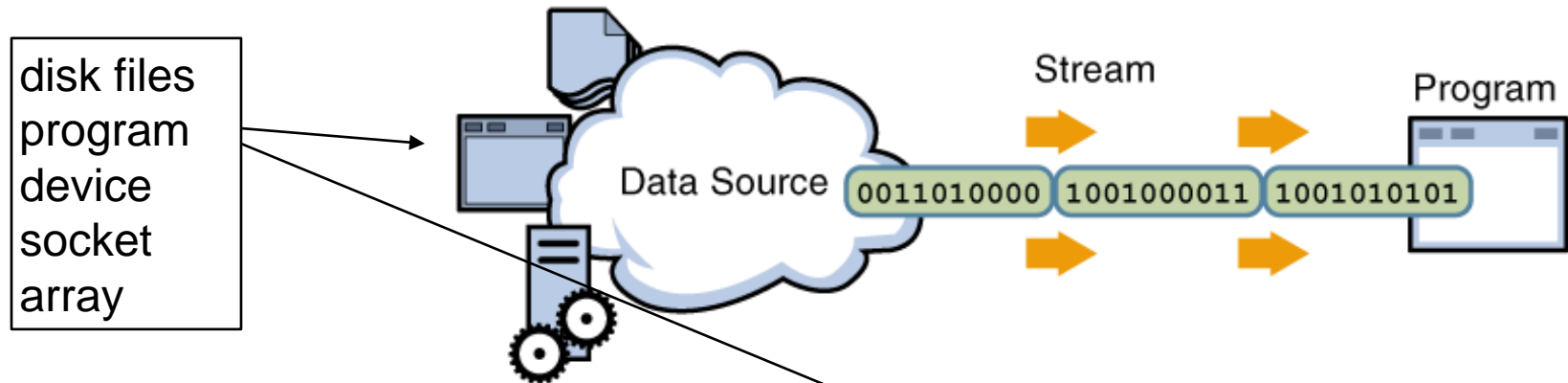


Text-Oriented Streams

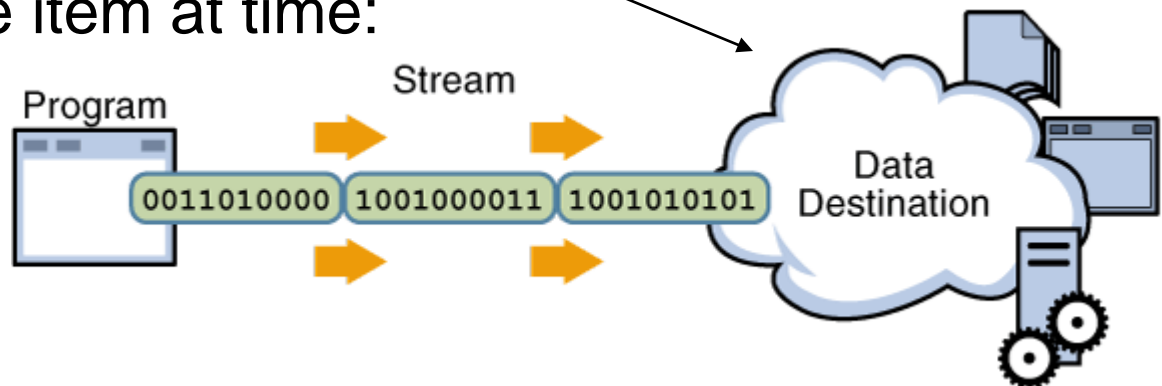


Input/Output Streams

- ⊕ A stream is a sequence of data.
- ⊕ A Java program uses an input stream to read data from a source, one item at a time:

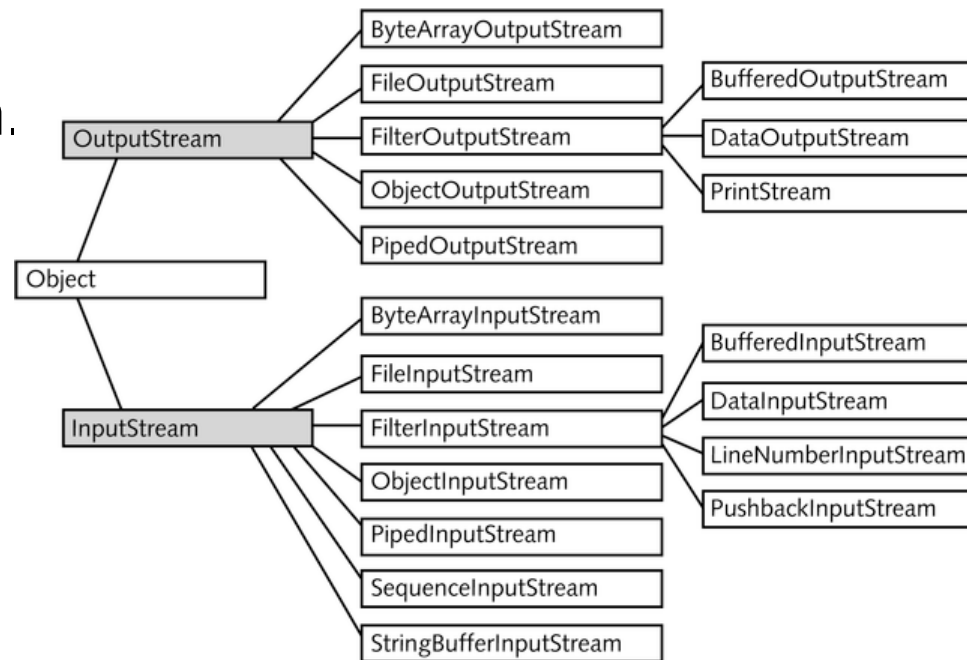


- ⊕ A Java program uses an output stream to write data to a destination, one item at a time:



Byte Streams

- ✦ Byte streams perform I/O of 8-bit bytes.
- ✦ All byte stream classes are descended from `InputStream` & `OutputStream`.
- ✦ 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.

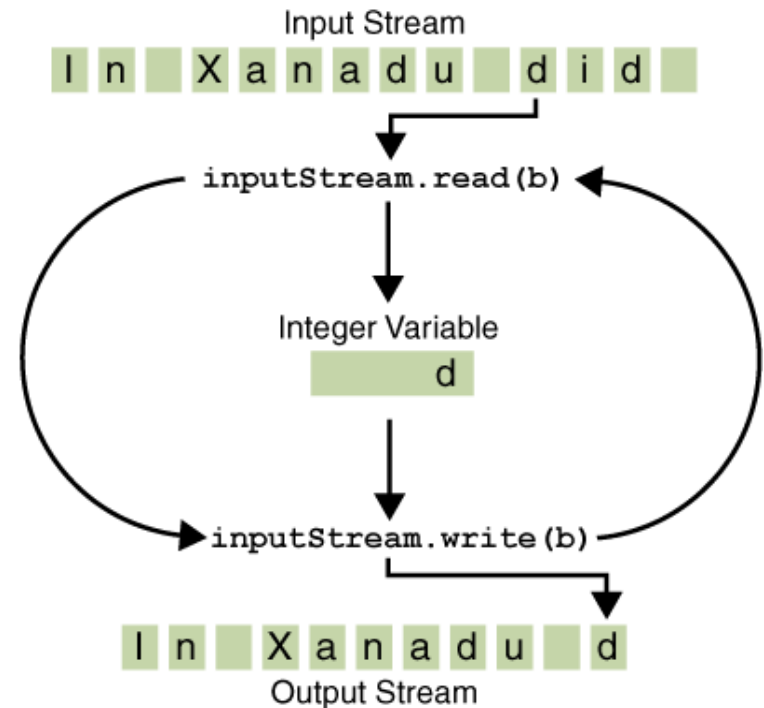


CopyBytes (Byte Streams)

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

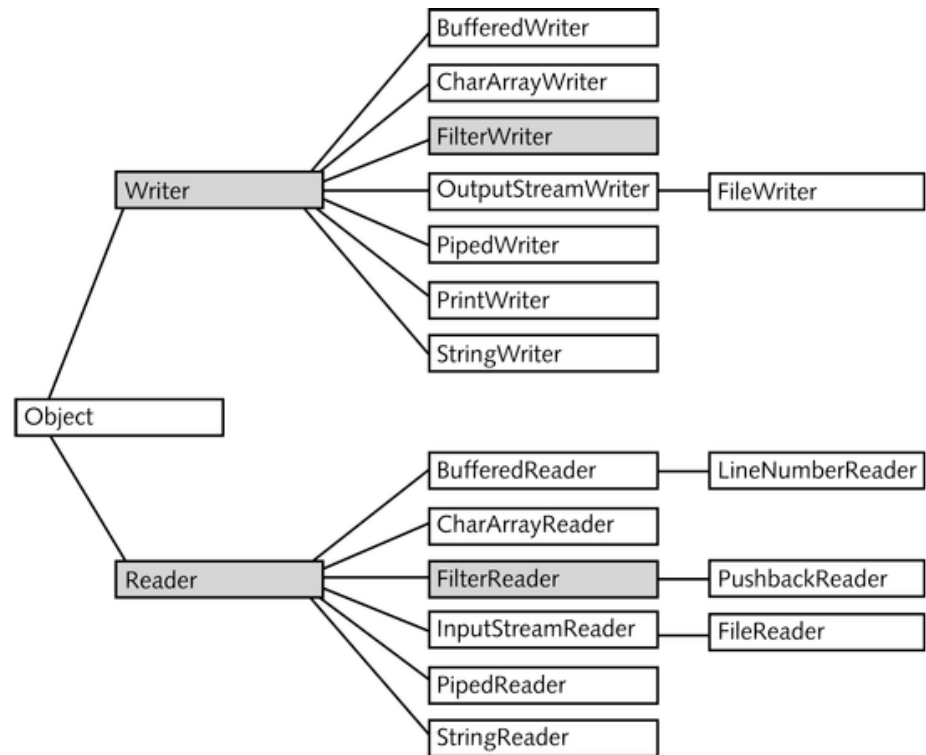
CopyBytes

- ✦ An int return type allows read() to use -1 to indicate end of stream.
- ✦ CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs; this helps avoid resource leaks.
- ✦ If CopyBytes was unable to open one or both files the associated file stream variable never changes from its initial null value; hence the test for null in the finally block.



Character Streams

- ⊕ Java stores character values using Unicode.
- ⊕ Character stream I/O automatically translates Unicode character values to and from the local character set.
- ⊕ In Western locales, the local character set is usually an 8-bit superset of ASCII.
- ⊕ I/O with character streams is no more complicated than I/O with byte streams.



CopyCharacters (Character Stream)

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

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.

Buffered IO

- ⊕ So far we have used unbuffered I/O:
 - ⊕ Each read or write request is handled directly by the underlying OS
 - ⊕ Can be less efficient, since each such request often triggers disk or network access → relatively expensive!

- ⊕ To reduce this kind of overhead use buffered I/O streams.
 - ⊕ Read data from a memory area known as a buffer
 - ⊕ Native input API is called only when the buffer is empty

 - ⊕ Buffered output streams write data to a buffer
 - ⊕ Native output API is called only when the buffer is full

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

CopyLines

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

            if (out != null) {
                out.close();
            }
        }
    }
}
```


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.

```
try
{
    in = new BufferedReader(new FileReader("input.txt"));
    out = new PrintWriter(
        new BufferedWriter(
            new FileWriter("characteroutput.txt")));

    String l;

    while ((l = in.readLine()) != null)
    {
        out.println(l);
    }
}
```

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, but has no effect unless the stream is buffered.

Scanning

- ⊕ Objects of type [Scanner](#) break input into tokens and translate individual tokens according to their data type.
- ⊕ 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("input.txt")));

            while (s.hasNext())
            {
                System.out.println(s.next());
            }
        }
        finally
        {
            if (s != null)
            {
                s.close();
            }
        }
    }
}
```

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);
    }
}
```

Translating Individual Tokens

- ⊕ ScanSum reads a list of double values and adds them up.
- ⊕ The ScanFile example treats all input tokens as simple String values.
- ⊕ Scanner also supports tokens for all of the Java language's primitive types as well as BigInteger and BigDecimal.

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 for 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

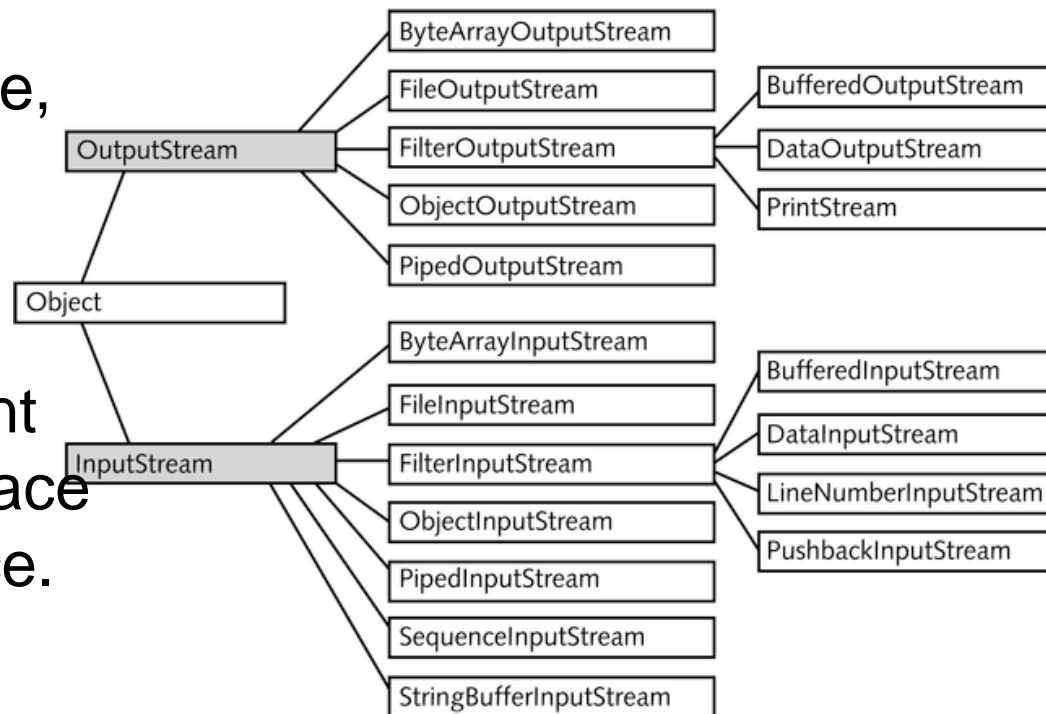
<code>void</code>	flush() Flushes the console and forces any buffered output to be written immediately .
Console	format (String fmt, Object ... args) Writes a formatted string to this console's output stream using the specified format string and arguments.
Console	printf (String format, Object ... args) 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.
<code>char[]</code>	readPassword () Reads a password or passphrase from the console with echoing disabled
<code>char[]</code>	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.

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)

```
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.

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](#).
 - ⊕ They implement [ObjectInput](#) and [ObjectOutput](#), which are subtypes of `DataInput` and `DataOutput`.
 - ⊕ Thus all the primitive data I/O methods covered in Data Streams are also implemented in object streams.
 - ⊕ 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](#).

ObjectStreams

```
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[] desc = { "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(desc[i]);
            }
        }
        finally
        {
            out.close();
        }
    }
    //...
}
```

ObjectStreams(2)

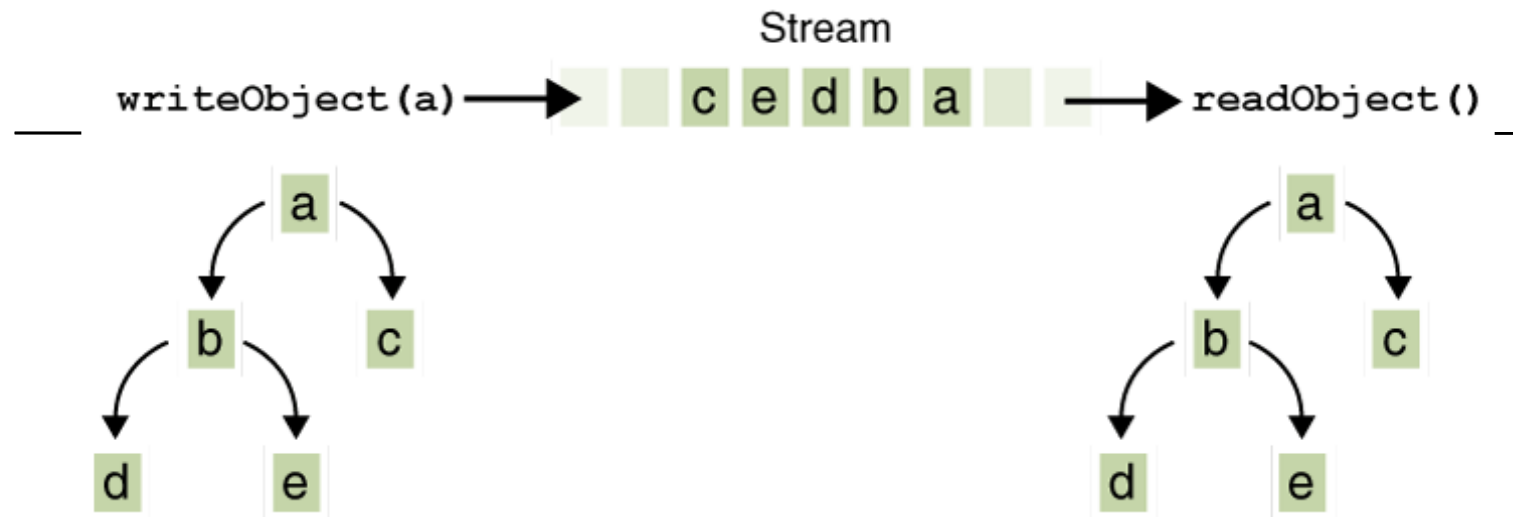
```
ObjectInputStream in = null;
try
{
    in = new ObjectInputStream(
        new BufferedInputStream(new FileInputStream(dataFile)));
    Calendar date = null;
    BigDecimal price;
    int unit;
    String desc;
    BigDecimal total = new BigDecimal(0);

    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.

Streams in AgileLab05

```
public class Pim implements IPim
{
    private AddressBookMap addressBook;

    public Pim()
    {
        newPim();
    }

    public IAddressBook getAddressBook()
    {
        return addressBook;
    }

    public void newPim()
    {
        addressBook = new AddressBookMap();
    }

    //...
}
```

open

```
public boolean open(String filename)
{
    boolean success = false;
    try
    {
        File source = new File(filename);
        ObjectInputStream is = new ObjectInputStream(new FileInputStream(source));
        addressBook = (AddressBookMap) is.readObject();
        is.close();
        success = true;
    }
    catch (ClassNotFoundException e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
    return success;
}
```

save

```
public boolean save(String filename)
{
    boolean success = false;
    try
    {
        File destination = new File(filename);
        ObjectOutputStream os
            = new ObjectOutputStream(new FileOutputStream(destination));
        os.writeObject(addressBook);
        os.close();
        success = true;
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
    return success;
}
```

Serializable Marker Interface

```
public class AddressBookMap implements IAddressBook, Serializable
{
    private static final long serialVersionUID = 1L;
    private Map<String, IContact> contacts;
    //...
}
```

```
public class Contact implements IContact, Serializable
{
    private static final long serialVersionUID = 1L;
    //...
}
```

- ⊕ The serialVersionUID should be incremented if the class structure changes.

transient

- ⊕ If a field is to be excluded from the serialisation mechanism it can be marked “transient”.
- ⊕ `writeObject()` will ignore these fields and `readObject()` will not attempt to read them.

```
public class AddressBookMap implements IAddressBook, Serializable
{
    private static final long serialVersionUID = 1L;
    private Map<String, IContact> contacts;
    private transient Map<String, IContact> removedContacts;
    //...
}
```

Abstract the Mechanism

```
public interface ISerializationStrategy
{
    void write(String filename, Object obj) throws Exception;
    Object read(String filename) throws Exception;
}
```

- ⊕ Defining this interface will allow us to build different serialization strategies.
- ⊕ We can decide which to use at compile time, or at run time.

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

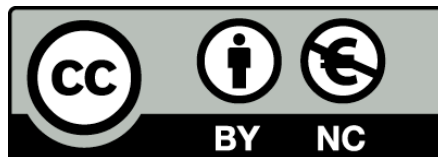
```
public class XMLSerializer implements ISerializationStrategy
{
    public Object read(String filename) throws Exception
    {
        ObjectInputStream is = null;
        Object obj = null;

        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();
            }
        }
    }
}
```



Except where otherwise noted, this content is licensed under a [Creative Commons Attribution-NonCommercial 3.0 License](http://creativecommons.org/licenses/by-nc/3.0/).

For more information, please see <http://creativecommons.org/licenses/by-nc/3.0/>



Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

