

# Tutorial 12

November 28, 29

# Topics

- Java I/O
- Streams
- Files
- Serialization
- Pset 9

# What is I/O?

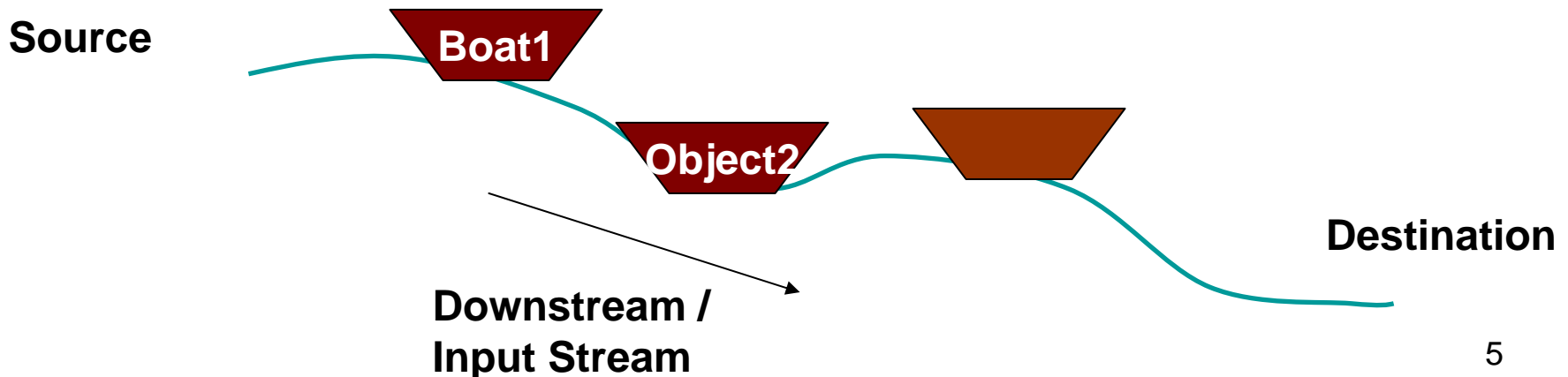
- All useful programs need to either “input” or “output” something
- There are many input sources:
  - Users (through a GUI interface or console application)
  - Files
  - Databases
  - Web URLs & Data Sockets
- These are also examples of places to output data
- Input and output need not occur at the same point  
e.g. A user input can be stored as an output to a database

# I/O in Java

- Uses java.io & java.nio packages
- Streams: Java programs communicate with outside world using streams.
- Files: Java I/O also can handle reading and writing to files for external, permanent storage
- Serialization: Use to save and load an object's state
- Catch IOException & any of its subclasses

# Streams

- A stream is an ordered sequence of bytes (or characters) of undetermined length.
- Unidirectional flow of data
  - Input stream: control data coming into the program
  - Output stream: control data leaving the program
  - Use 2 different streams if reading from & writing to same place
- Implemented using FIFO queues



# How to use streams for I/O?

- Reading
  - Open stream
  - While more data needs to be read
    - read data
  - Close stream (after all data has been read)
- Writing
  - Open a stream
  - While more data remains to be written
    - write data
  - Close stream

# Abstract Stream Classes

- 2 Classes for reading or writing an unstructured sequence of bytes. Other byte streams are built on top of these
  - **InputStream** : Reads bytes
  - **OutputStream** : Writes bytes
- 2 Classes for reading or writing a sequence of character data, with support for Unicode. Other character streams are built on top of these
  - **Reader** : Read chars
  - **Writer** : Write chars

# Basic Console I/O

- Simplest examples of input & output streams are System.in & System.out streams

```
InputStream stdin = System.in;
```

```
OutputStream stdout = System.out;
```

- These process data from the console window
- Read single byte of data using read() method

```
int val;
```

```
try {
```

```
    while( (val=System.in.read( )) != -1 )
```

```
        System.out.println((byte)val);
```

```
    } catch ( IOException e ) { ... }
```

- close() method shuts down stream & frees resources

# Extending Stream Functionality

- Stream Wrappers: Use these when you need to do more than just reading & writing to a stream. Some examples are:
  - **BufferedInputStream** (a type of `InputStream`) reads ahead and buffers data in memory. This increases efficiency by reducing number of read calls
  - **DataInputStream** (another type of `InputStream`) allows you to read strings & other primitive types that are more than a byte long
- Streams can also be coupled (piped) to make use of each stream's functionality

# Coupling Streams

- Java streams may be combined by using one stream as a constructor argument to another
- This is usually done to add functionality and/or convert the format or representation of the data
- Stream pipelines are constructed
  - from the data source to the program or
  - from the data destination back to the program

# Coupling Streams: Example

```
try
{
    Reader r = new FileReader("input.txt");
    int next = r.read();    // using the
                           // FileReader only

    // example of coupling
    BufferedReader b = new BufferedReader(r);
    next = b.readLine();
}
catch (IOException e)
{
}
```

# Files

- Use `java.io.File` class to work with files or directories
- Constructing a new `File` object

```
File f = new File ("filename.txt");
```

Remember to specify the full path
- Can obtain information about a file using methods such as `getName()` and `length()`
- Use `createNewFile()` method to make a new zero-length file at a particular location
- Read and write to files using streams.
- Need to catch the `FileNotFoundException`

# File Stream Classes in Java

- **FileInputStream**
  - Read data in binary format from files
- **FileOutputStream**
  - Write data in binary format to files
- **FileReader**
  - Read text data from files
- **FileWriter**
  - Write text data to files

# Files Exercise: Text Streams

- Write a program to copy the contents of the file “test.txt” to the file “output.txt”. Assume that both files are located in the root folder of the C:\ drive.
- You can either read the input file character by character and write it to the output file using the **FileWriter** class, or use the **writeln()** method of the **PrintWriter** class.
- Use an object of type **FileWriter** as argument to the constructor of the **PrintWriter**

# Files Exercise – Byte Streams

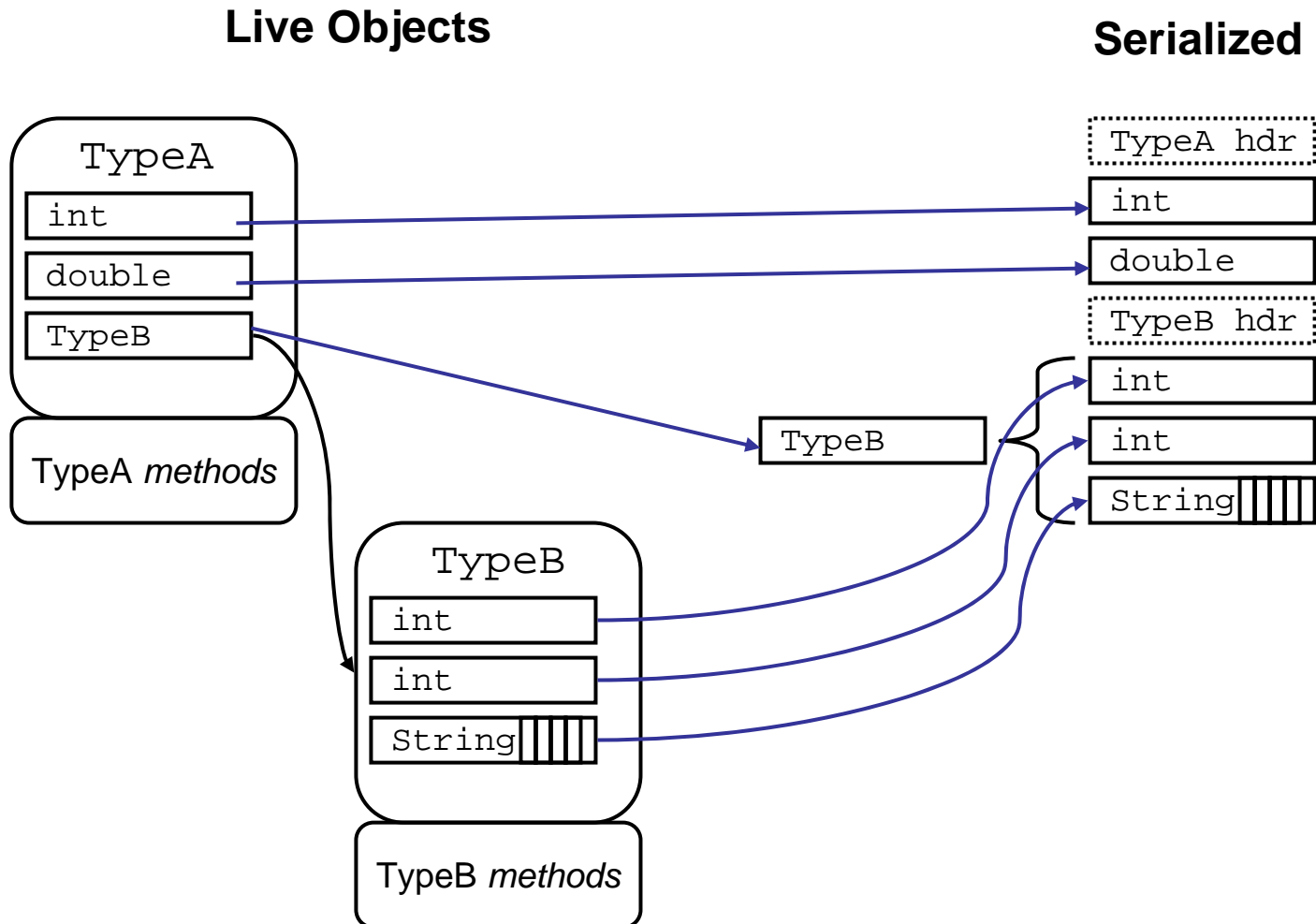
Download the `BinaryData.java` file and look at its `main()` method.

- The program writes some data in binary format to the file “binaryData”. It also records the number of data points in the file.
- Write code to read the data from the file, add them and print the total
- You need to use the `DataInputStream` class and the `FileInputStream` class

# Object Serialization

- Process of reading and writing objects to a stream is called object serialization
- Write objects to a stream using `ObjectOutputStream` and read objects to a stream using `ObjectInputStream`
- All object references within the object are also serialized
- The object needs to implement the `Serializable` interface

# Serialization Diagram



# Serialization Example

- Look at the main method in SerializeExample.java
- It first writes an object to an ObjectOutputStream

```
try{
    FileOutputStream out = new FileOutputStream
        ("theTime");
    ObjectOutputStream s = new ObjectOutputStream(out);
    SerializeExample b = new SerializeExample();
    s.writeObject("Today");
    s.writeObject(new Date());
    s.writeObject(b);
    s.close();
}catch(IOException e){ }
```

# Serialization Example

- It then reads an object using an `ObjectInputStream`

```
try{
    FileInputStream in = new FileInputStream
                        ("theTime");
    ObjectInputStream s = new ObjectInputStream(in);
    String today = (String)s.readObject();
    Date date = (Date)s.readObject();
    SerializeExample ex = (SerializeExample)
                        s.readObject();

    System.out.println(ex.a);
    s.close();
}
catch(IOException e){}
catch(ClassNotFoundException e){}
```

# Serialization Pitfalls

- It causes security loopholes
- Can change an object's properties from outside without calling any of its methods
- Also private fields values will be serialized
- So use serialization only if you absolutely need it
- Throw a `NotSerializableException` when you want to prevent serialization of a class
- **static**, **transient** data members are not serialized

# Files Summary

To read and write:

- Text data: Use `FileReader` and `FileWriter`
- Binary data: Use `DataInputStream` coupled to a `FileInputStream` and a `DataOutputStream` coupled to a `FileOutputStream`
- Objects: Use an `ObjectInputStream` coupled to a `FileInputStream` and an `ObjectOutputStream` coupled to a `FileOutputStream`

# Problem Set 9

- You will need to use text files to save and load your boards
- You can choose your own formats for your text files