CSE 331

Memento Pattern and Serialization

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia
http://www.cs.washington.edu/331/

Pattern: Memento

a memory snapshot
of an object's state

MEMENTO
SOME MEMORIES ARE BEST FORGOTTEN.
GUY PEARCE CARRIE-ANNE MOSS JOE PANTOLIANO
MEMENTO
otnemem.com

The problem situation

- *Problem*: Sometimes we want to remember a version of an important object's state at a particular moment.
  - example: Writing an Undo / Redo operation.
  - example: Ensuring consistent state in a network.
  - example: persistency; save / load state between runs of a program.

- Poor solutions to this problem:
  - Writing out the object's state as a formatted text file, reading it back in and parsing it again later.
  - Making many deep copies of the object.

Memento pattern

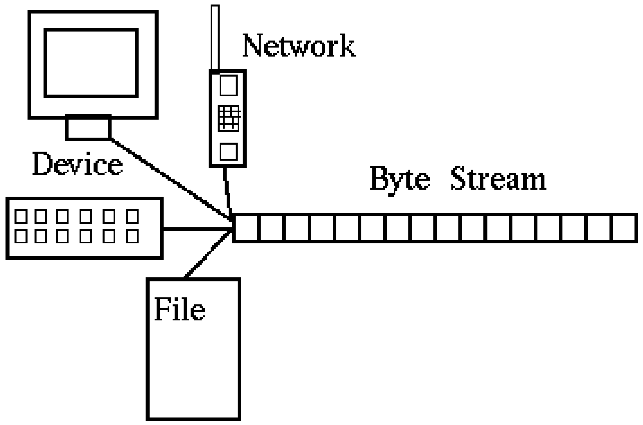
- **memento**: A saved "snapshot" of the state of an object or objects for possible later use.
  - Often involves a mechanism for transporting an object from one location to another and back again.

- We'll examine Memento in the context of saving an object to disk using input/output streams.
  - Also very useful for implementing Undo/Redo functionality.

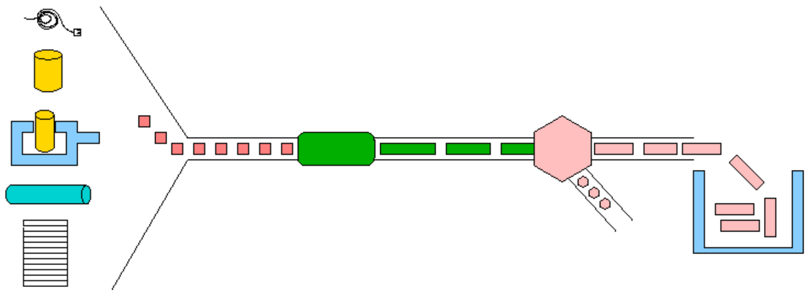
I/O streams, briefly

- **stream**: An abstraction of a source or target of data.
  - 8-bit bytes flow to (output) and from (input) streams.

- Can represent many data sources:
  - files on hard disk
  - another computer on network
  - web page
  - input device (keyboard, mouse, etc.)

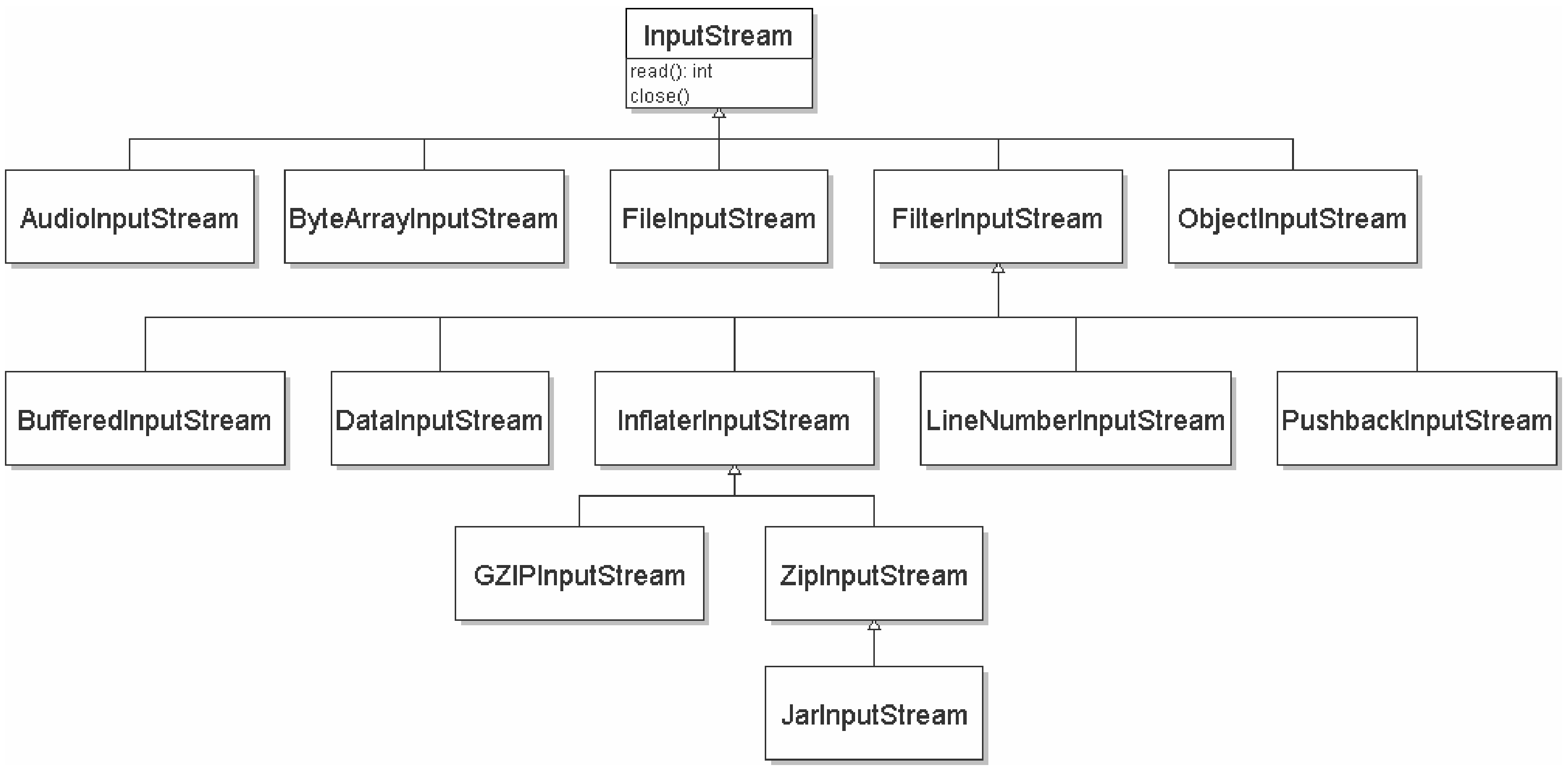


- Represented by `java.io` classes:
  - `InputStream`
  - `OutputStream`



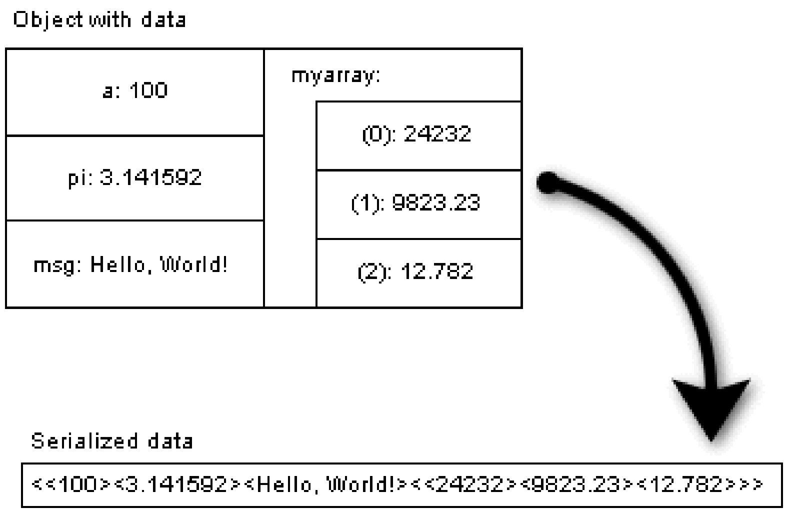
Streams and inheritance

- all input streams extend common superclass `InputStream`; all output streams extend common superclass `OutputStream`
  - Guarantees that all sources of data have the same methods.
  - Provides minimal ability to read/write one byte at a time.



Serialization

- **serialization**: Reading / writing objects and their exact state into a linear format using I/O streams.
  - Entire objects can be written to files, a network, a database, etc.
  - Lets you save your objects to disk and restore later.
  - Avoids converting object's state into an arbitrary text format.



Classes for serialization

- `ObjectOutputStream` : A connection to write (save) objects.
  - `public class` **`ObjectOutputStream`**
    - `public ObjectOutputStream(OutputStream out)`
    - `public void` **`writeObject`**`(Object o)`
      `throws IOException`
- `ObjectInputStream` : A connection to read (load) objects.
  - `public class` **`ObjectInputStream`**
    - `public ObjectInputStream(InputStream in)`
    - `public Object` **`readObject`**`() throws Exception`
- Common read/write target: A file.
  - A `FileInputStream` or `FileOutputStream` can be constructed by passing a file name string.

Serialization example

```
// write the given object to the given file
try {
    OutputStream os = new FileOutputStream("filename");
    ObjectOutputStream oos = new ObjectOutputStream(os);
    oos.writeObject(object);
    oos.close();
} catch (IOException e) { ... }


// load the object named someObject from file "file.dat"
try {
    InputStream is = new FileInputStream("filename");
    ObjectInputStream ois = new ObjectInputStream(is);
    Type name = (Type) ois.readObject();
    ois.close();
} catch (Exception e) { ... }
```

Making a class serializable

- You must implement the (methodless) `java.io.Serializable` interface for your class to be compatible with streams.

```
public class BankAccount implements Serializable {
    ...
```

  - *(Recall: Methodless "tagging" interfaces (Serializable, Cloneable) pre-date better techniques such as annotations.)*

serialVersionUID

- There is a versioning issue with serializing / deserializing objects.
  - You might save a `BankAccount` object, then edit and recompile the class, and later try to load the (now obsolete) object.
  - Serializable objects should have a field inside named `serialVersionUID` that marks the "version" of the code.
    - (If your class doesn't change, you can set it to 1 and never change it.)

```
public class BankAccount implements Serializable {
    private static final long serialVersionUID = 1;
    ...
```

Serializable fields

- When you make a class serializable, all of its fields must be serializable as well.
  - All primitive types are serializable.
  - Many built-in objects are serializable:
    - String, URL, Date, Point, Random
    - all collections from java.util (ArrayList, HashMap, TreeSet, etc.)
  - But your own custom types might not be serializable!

- If you try to save an object that is not serializable or has non-serializable fields, you will get a `NotSerializableException`.

Transient fields

- **transient**: Will not be saved during serialization.

  ```
  private transient type name;
  ```

  Example:

  ```
  private transient PrintStream out;
  ```

- Ensure that all instance variables inside your class are either serializable *or* declared `transient`.
  - A `transient` field won't be saved when object is serialized.
  - When deserialized, the field's value will revert back to `null` .

Custom serialization

- The object in/out streams have a default notion of how objects should be serialized and saved.
  - If this is unsatisfactory for your object for some reason, you can override it by writing these methods in your class:

```
private void writeObject(ObjectOutputStream out)
    throws IOException

private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException

private void readObjectNoData()
    throws ObjectStreamException
```

  - (You don't usually need to write these methods.)

Serialization exercise

- Let's make our Rock-Paper-Scissors game serializable.
  - Save the state of past rock-paper-scissors games played and games won by the first player.
  - When the game loads again, restore that state.

- If you have time, implement an Undo feature.
  - This feature will go back to the previous game.