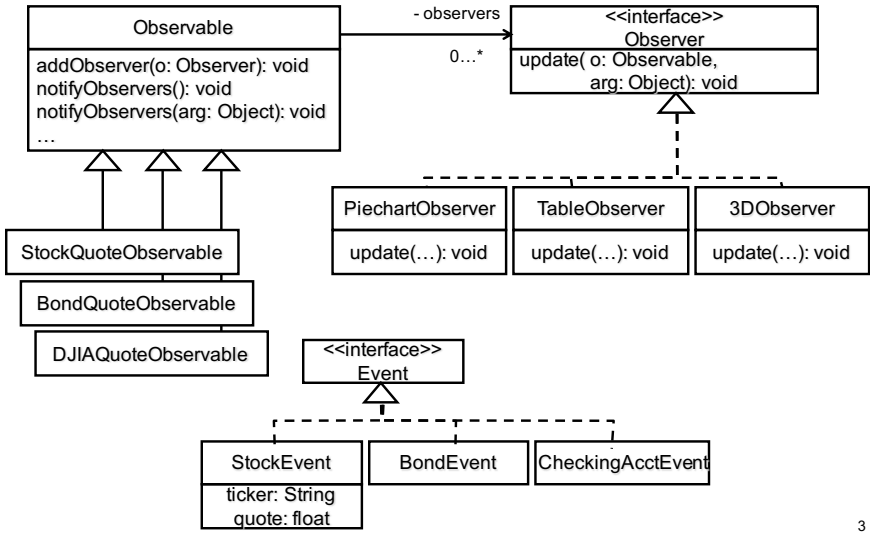
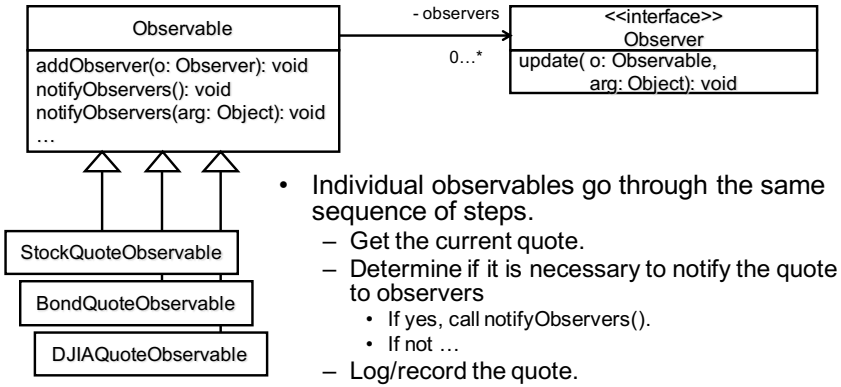
Template Method Design Pattern

Template Method

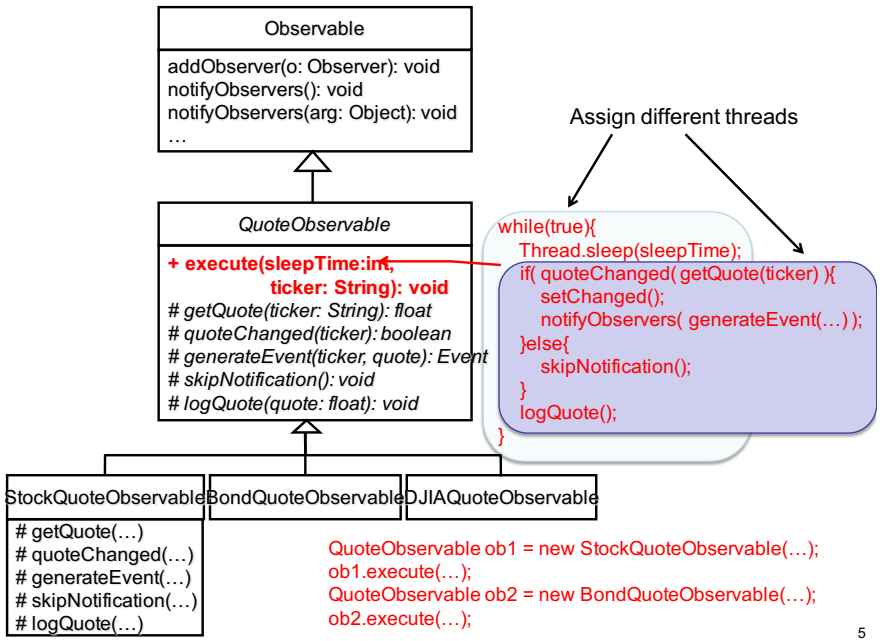
- Intent
  - Define the template (or abstract flow) of an algorithm (or logic/procedure) in a superclass's method
    - Leave implementation details of some steps to subclasses.
    - Subclasses implements those steps without changing the template/flow that their superclass defines.

Recap: Observer





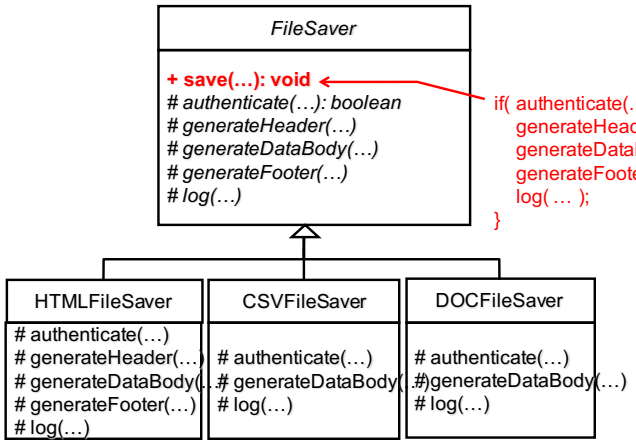
- Individual observables go through the same sequence of steps.
  - Get the current quote.
  - Determine if it is necessary to notify the quote to observers
    - If yes, call notifyObservers().
    - If not …
  - Log/record the quote.
- However, different observables may want to implement certain steps in different ways.
  - Where to get a quote from?
  - What to do if no events are notified?
  - How to log/record a quote?
  - How often to run this sequence?



Points

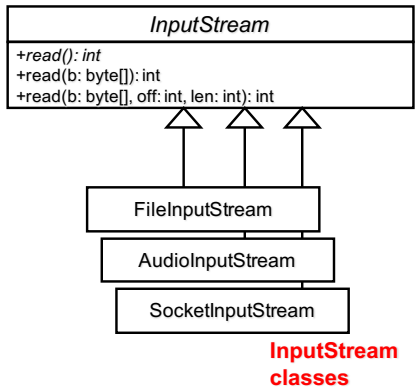
- Can avoid copying and pasting similar yet slightly different code across different Observable classes.
  - Intend to improve maintainability
- A template method can be a “final” method.

Saving Data to a File

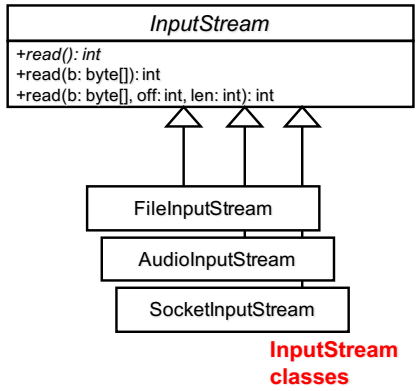


InputStream in Java API

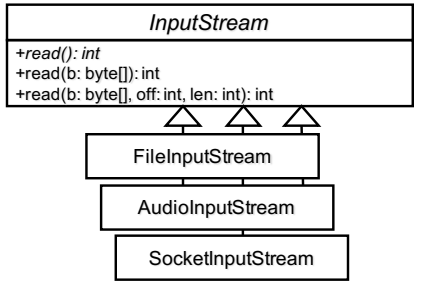
- InputStream
  - Abstract class that represents input byte streams
- InputStream classes
  - AudioInputStream
  - FileInputStream
  - ObjectInputStream
  - PipedInputStream
  - …etc.
  - SocketInputStream (hidden)



- read()
  - Reads the next byte of data from an input stream. The value byte is returned in the range from 0 to 255.
  - Returns -1 if no byte is available because the end of the stream has been reached.
  - Defined as an abstract method. Each subclass must implement it.



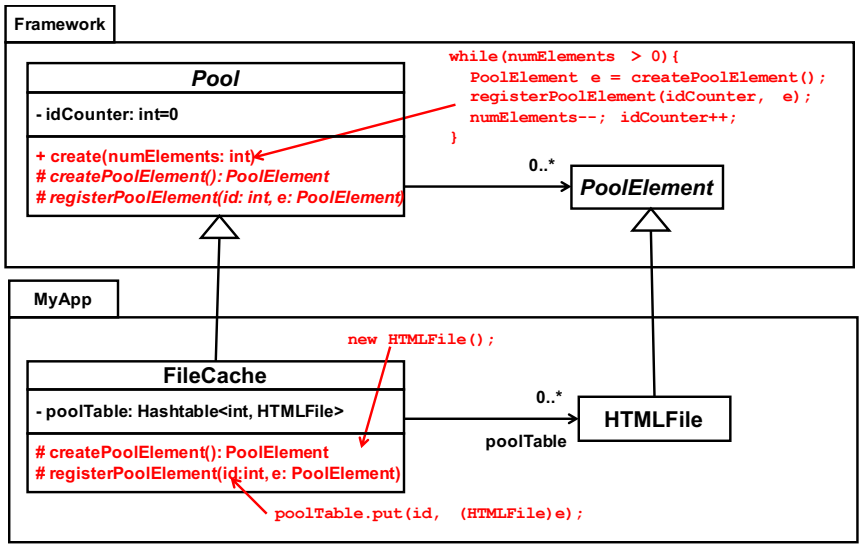
- read()
  - Reads the next byte of data from an input stream. The value byte is returned in the range from 0 to 255.
  - Returns -1 if no byte is available because the end of the stream has been reached.



- Template methods
  - read(byte[] b)
    - Reads multiple bytes
    - int i = 0;
      while(i < b.length){
        read the next byte with read();
        if (EOS is reached) break;
        copy the byte to b[i];
        i++;
      }
      return i;
    - Equivalent to call read(b, 0, b.length)
  - read(byte[] b, int off, int len)
    - Reads multiple bytes
    - int i = 0;
      while(i < len){
        if (EOS is reached) break;
        read the next byte with read();
        copy the byte to b[off+i];
        i++;
      }
      return i;

Template Method and *Factory Method*

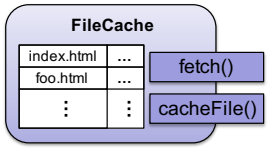
- *Factory method* is a variant of *Template Method*
  - Specializing in instance creation/initialization
    - Factory method: defining a template for instance creation/instantiation.
    - Template method: defining a template for any logic.

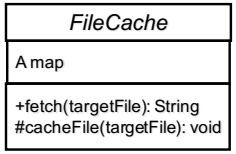



```
FileCache cache = new FileCache();
cache.create(10);
```

File Caching

- **`FileCache`**
  - Maintains a map that pairs a relative file path and string data of the file.
    - Assume **`java.util.HashMap`**
  - **`fetch()`**
    - accepts a file path and gets the content of the requested file from the HashMap.
  - **`cacheFile()`**
    - accepts a file path and its content to the HashMap.

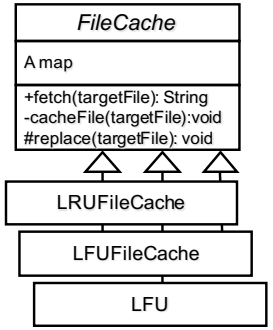




- **`FileCache`**
  - **`public String fetch(targetFile)`**
    - targetFile: a relative file path
      - String or java.nio.Path
    - Returns a requested file's content

```
- if ( targetFile is cached ){
      return targetFile's (cached) content.
  } else if {
      obtain targetFile from the disk.
      if( cache is not full )
          cacheFile(targetFile);
      else if
         replace( targetFile ); }
```

- C.f. Lecture note #12
  - Deisgn w/ *Strategy*



Decorator Design Pattern

Code Reuse in OOP

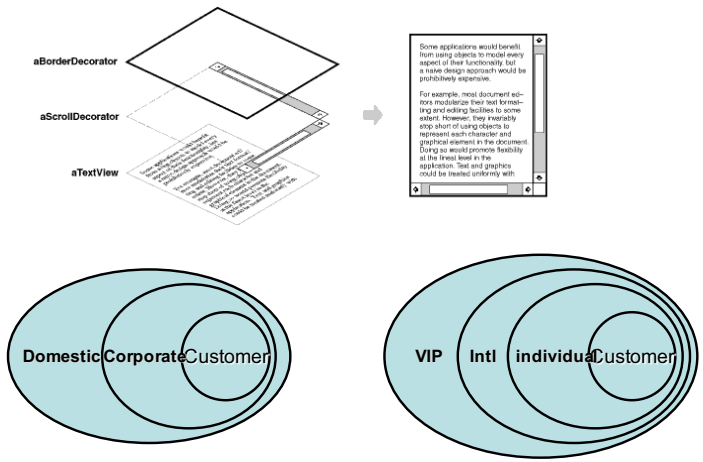
- Three categories of code reuse
  - Instantiate an existing class
    - new Thread (new MyRunnableClass())
  - Extend an existing class
    - SSLServerSocket extends ServerSocket
  - Combine instances at runtime
    - Decorator design pattern

When is Instance Combination Useful?

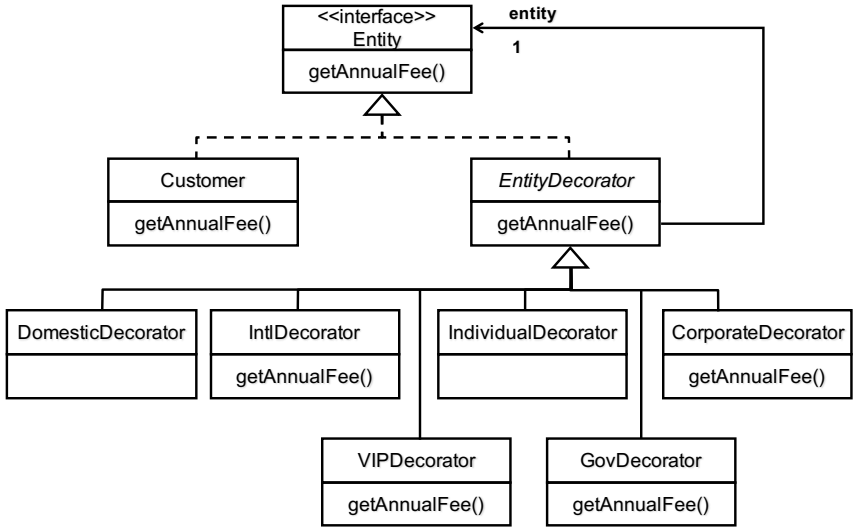
- class VisualString{
  private String string;
  public void draw(){
  // draw a string on the screen } }
- If you want to draw a box around the string…
  - class BoxedString extends VisualString{
    public void draw(){
    // redefine how to draw a string } }
- If you want to draw a string in color…
  - Class ColoredString extends VisualString{
    public void draw(){
    // redefine how to draw a string } }
- What if you want to draw a box around a colored string?
- What if you want to draw a box around an icon?

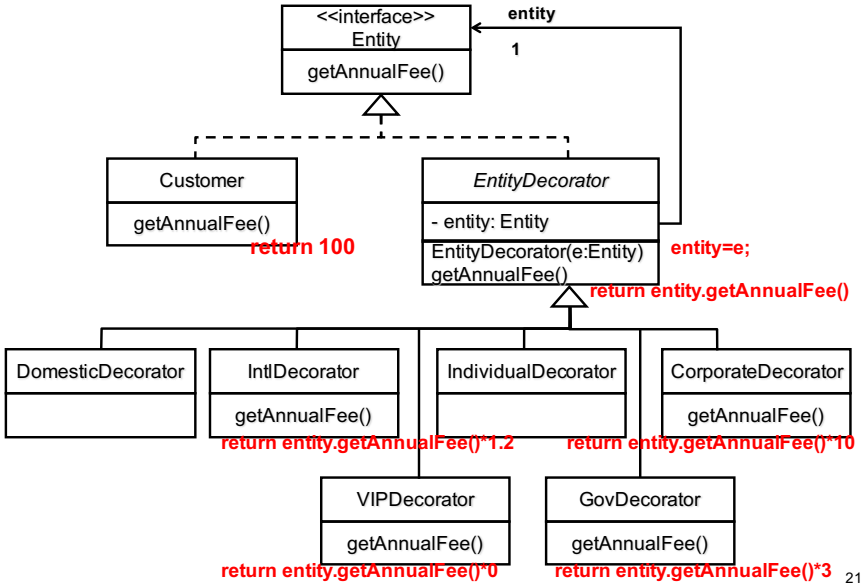
- class Customer{
  public double getAnnualFee(){ … } }
- class DomesticCustomer extends Customer{
  public double getAnnualFee(){ … } }
- class InternationalCustomer extends Customer{
  public double getAnnualFee(){ … } }
- What if you want to set up
  - Domestic/international corporate customers and domestic/international government customers
  - VIP discount rate?, employee discount rate??
- What if you want to change the status of a customer from non-VIP to VIP or domestic to international?

Decorator Design Pattern

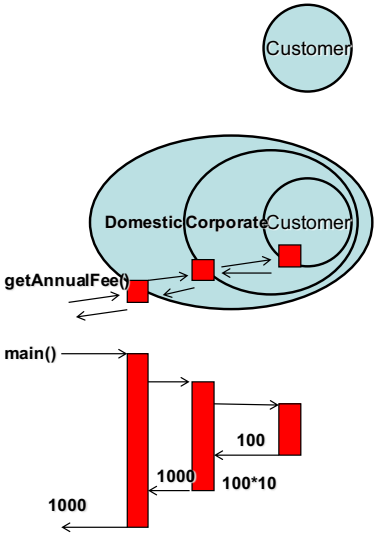


Sample Code: Class Structure





Sample Code: Instance Combination

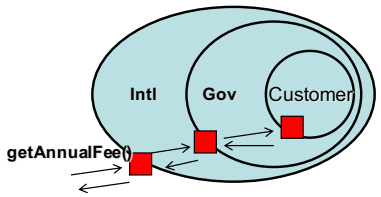


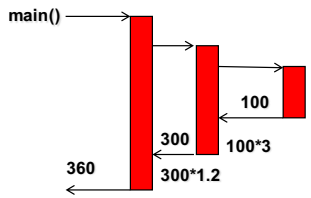
- Customer customer =
  new Customer();
  customer.getAnnualFee();
  --> 100.0

- Entity customer =
  new DomesticDecorator(
  new CorporateDecorator(
  new Customer()));
  customer.getAnnualFee();
  --> 1000.0

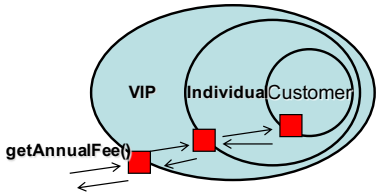
  – The kernel + 2 skins (decorators)

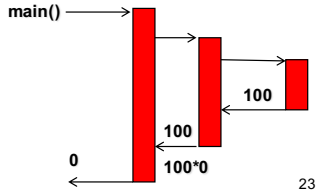
new InternationalDecorator(
new GovernmentDecorator(
new Customer()));





- new VIPDecorator(
  new IndividualDecorator(
  new Customer()));





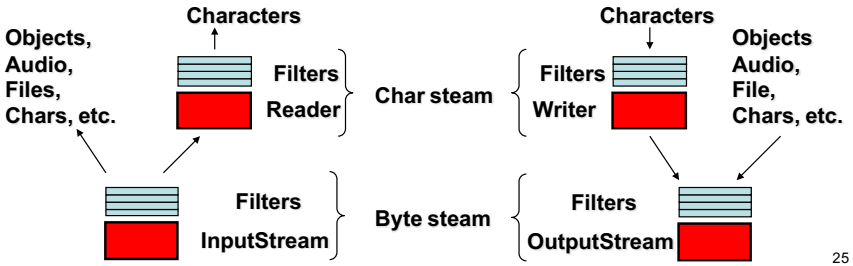
Decorator Design Pattern

- This is a good design strategy if…
  – you have many skins/decorators to combine.
  – you expect to add extra skins/decorators in the future.

- Otherwise, be careful.
  – The number of classes increases.
  – Your fellow developers might feel puzzled.

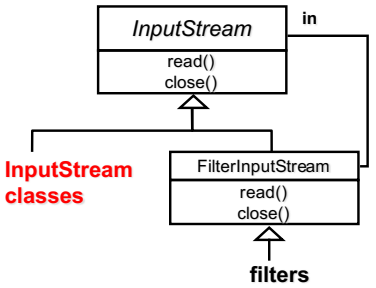
Example: Java I/O API

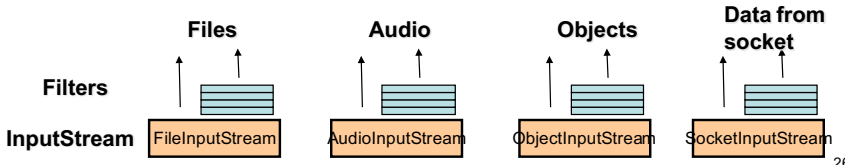
- InputStream and OutputStream
  - Abstract classes that represent input and output byte streams.
- Reader and Writer
  - Abstract classes to read and write character streams
- Many filters are defined to tailor/specialize input/output streams and readers/writers.



InputStream

- InputStream objects can be decorated with an arbitrary number of filters.
- InputStream classes
  - AudioInputStream
  - FileInputStream
  - ObjectInputStream
  - PipedInputStream
  - …etc.
  - SocketInputStream (hidden)



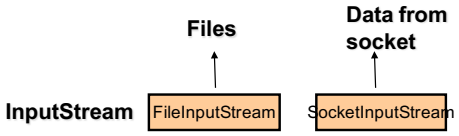


Initialization of InputStream

```
InputStream is = new FileInputStream("foo.txt");
is.read();
```

```
Socket socket = new Socket(…);
InputStream is = socket.getInputStream();
is.read();
```

- `System.out.println(socket.getInputStream().toString() );`
  - `java.net.SocketInputStream@10d81b`

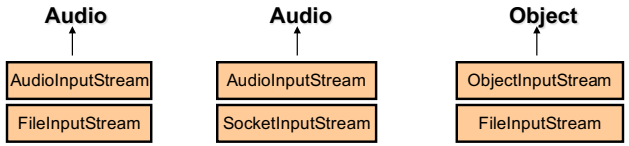


- Some InputStream objects can be connected with each other.

```
InputStream is = new AudioInputStream(
    new FileInputStream( "audiofile") );
is.read();
```

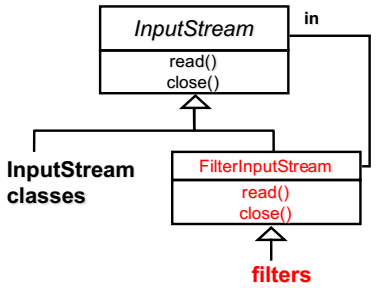
```
InputStream is = new AudioInputStream(
    socket.getInputSteam() );
is.read();
```

```
InputStream is = new ObjectInputStream(
    new FileInputStream( "serializedObjectFile") );
is.read();
```

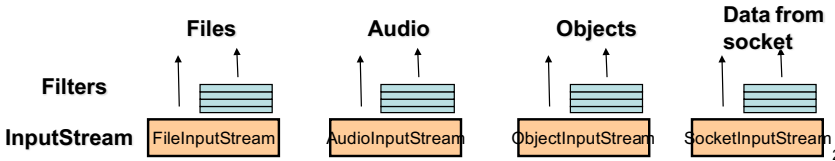


Filters of InputStream

- Filters
  - BufferedInputStream
  - CheckedInputStream (java.util.zip)
  - ZipInputStream (java.util.zip)
  - GZIPInputStream (java.util.zip)
  - JarInputStream (java.util.zip)
  - CipherInputStream (javax.crypto)
  - DigestInputStream (java.security)
  - …etc.



- An InputStream object can be decorated with multiple filters.



- `InputStream is = new BufferedInputStream( socket.getInputStream() ); is.read();`

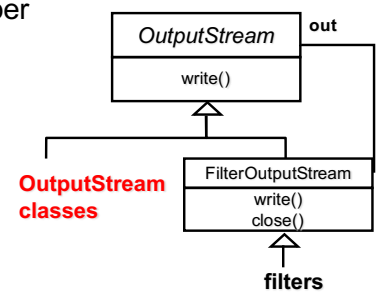
- `InputStream in = new ZipInputStream ( new FileInputStream(…) ); in.read();`

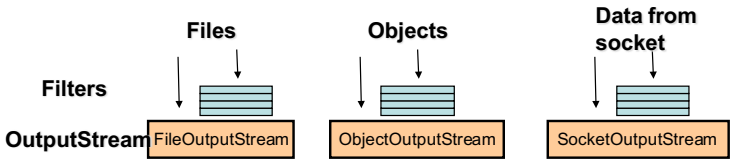
- `InputStream in = new BufferredInputStream( new ZipInputStream( socket.getInputStream() )); in.read();`

OutputStream

- OutputStream objects can be decorated with an arbitrary number of filters.

- OutputStream classes
  - FileOutputStream
  - ObjectOutputStream
  - PipedOutputStream
  - …etc.
  - SocketOutputStream (hidden)

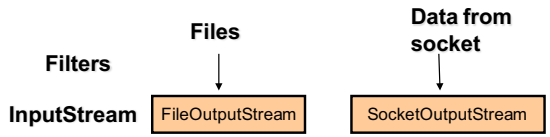




Initialization of OutputStream

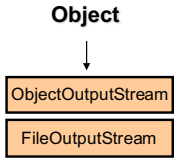
- `OutputStream os = new FileOutputStream("foo.txt"); os.write();`

- `Socket socket = new Socket(…); OutputStream os = socket.getOutputStream(); os.write();`
  - System.out.println(socket.getOutputStream().toString());
    - java.net.SocketOutputStream@11d32c



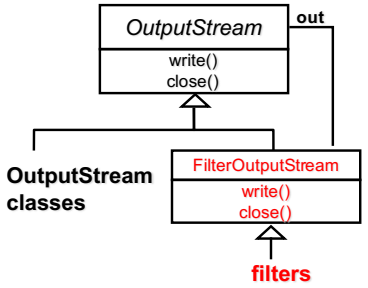
- Some OutputStream objects can be connected with each other.

- ```
  OutputStream os = new ObjectOutputStream(
      new FileOutputStream( "objectfile") );
  ```



Filters of OutputStream

- Filters
  - BufferedOutputStream
  - PrintStream
  - CheckedOutputStream (java.util.zip)
  - ZipOutputStream (java.util.zip)
  - GZIPOutputStream (java.util.zip)
  - JarOutputStream (java.util.zip)
  - CipherOutputStream (javax.crypto)
  - DigestOutputStream (java.security)
  - …etc.



- An OutputStream object can be decorated with multiple filters.


- ```
  OutputStream os = new ZipOutputStream(
              new FileOutputStream(…) );
  os.write(…);
  ```

- ```
  OutputStream os = new BufferredOutputStream(
              new ZipOutputStream(
                    socket.getOutputStream() ));
  os.write(…);
  ```

- ```
  OutputStream os = new PrintStream(
                socket.getOutputStream() );
  os.write(…);
  ```


Readers, Writers and Utility Classes

- PrintWriter
  - Character interface for PrintStream
  - print() println(), printf(), etc.
- FileWriter
  - Convenience class for writing character files.
- FileReader
  - Convenience class for reading character files.
- Scanner
  - Not a subclass of FilterInputStream nor FilterOutputStream
  - A text scanner that can parse primitive types and strings using regular expressions.

HW18

- The Decorator design pattern can eliminate a long sequence of conditional statements.
- Show and explain how your code looks like if the Decorator design pattern is not used.
  - It can be pseudo code. It doesn't have to be complete/compilable code.
- Explain how the Decorator design pattern eliminates conditional statements.
  - Explain why Java API designers decided to use *Decorator* in java.io.