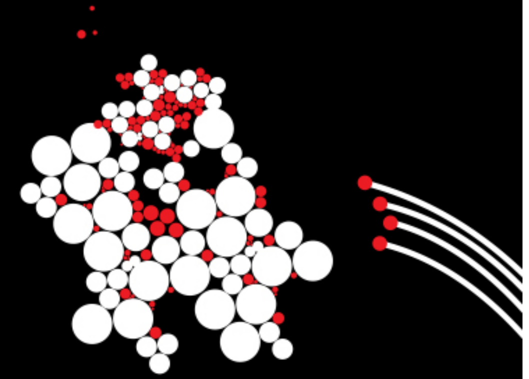
UNIVERSITY OF TWENTE.

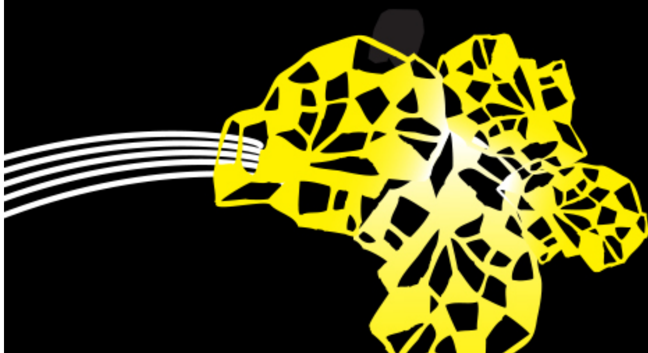


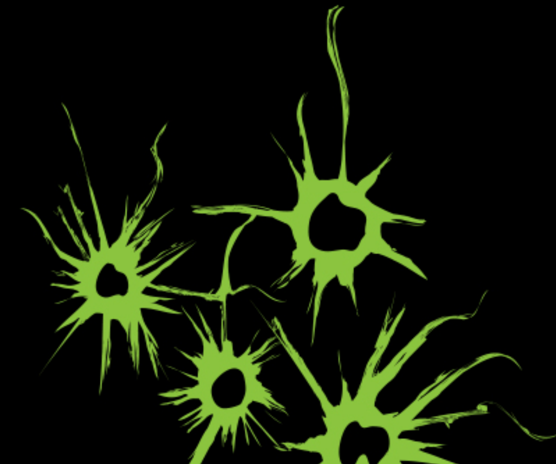
P5.2: I/O STREAMS AND MVC

LUÍS FERREIRA PIRES

201700117-1B MODULE 2: SOFTWARE SYSTEMS

10 DECEMBER 2019



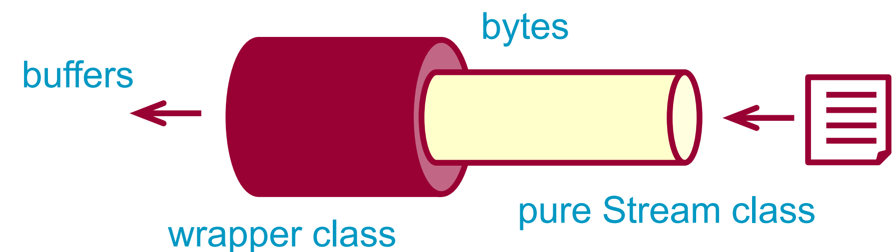


PROGRAMMING LINE OVERVIEW

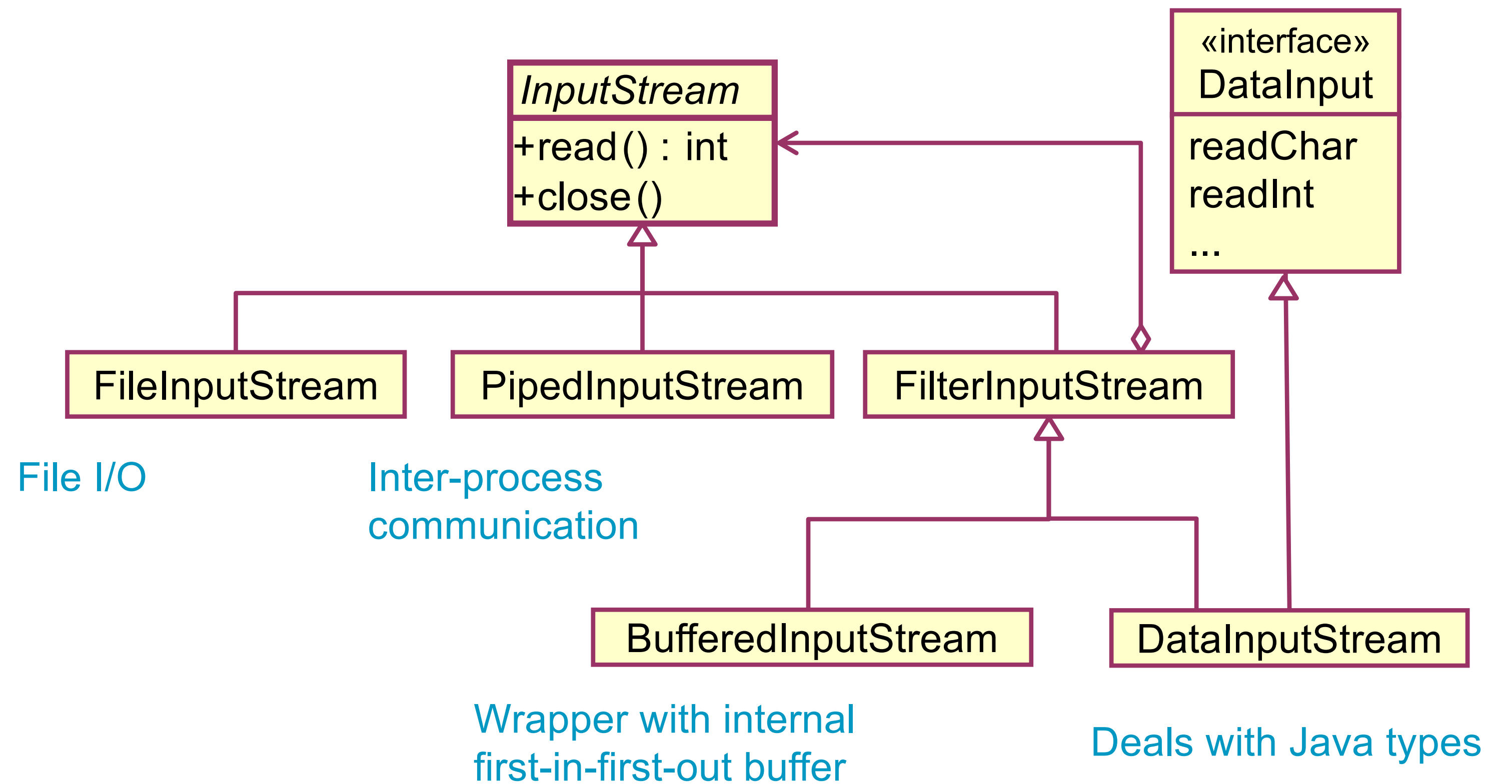
Week 1	Week 2	Week 3
Values and variables Control flow	Classes and objects Testing	Interfaces and Inheritance Subtyping Security 1
Week 4	**Week 5**	**Week 6**
Arrays and Lists List implementations Collections	Exceptions I/O Streams and MVC Security 2	Concurrency Project kick-off IDE Tips & Tricks
Week 7	**Week 8/9**	**Week 10**
Basic Networking Networking and Multithreading GUIs	Advanced Java facilities Test	Project Test resit

DATA INPUT TO A PROGRAM

- From user, file system, program components, other programs, etc.
- Exchange data in the form of
  - Bytes (8 bits): Stream classes
  - Characters: Reader/writer classes
- Extra functionality wrapped around pure stream classes
  - Buffer, data handling, etc.



INPUT STREAM HIERARCHY



CLASS INPUTSTREAM

```
//Abstract superclass
abstract class InputStream implements AutoCloseable {
        // Returns next byte from stream
        /*@ensures (0 <= result && result <= 255) ||
                result == -1;  // end of stream*/
        int read() throws IOException {  }

        // Closes the stream
        void close() throws IOException {  }
}
```

Necessary to release resources
(specified in `AutoCloseable`)

BYTE STREAM EXAMPLE

```
String text = "";
boolean b = true;
double salary = 0.0;
DataInputStream datIn = null;

try {
    InputStream in = new FileInputStream("data.dat");
    InputStream bufIn = new BufferedInputStream(in);
    datIn = new DataInputStream(bufIn);
    text = datIn.readUTF();
    b = datIn.readBoolean();
    salary = datIn.readDouble();
} catch (IOException e) {
    e.printStackTrace();
}
finally {
    try {
        datIn.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Define variables outside try-catch and initialise them

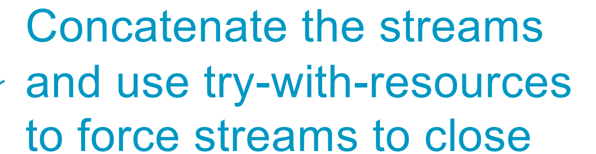
Define try-catch to handle I/O exceptions

Read data from file

Close the stream in `finally` to avoid memory leaks!

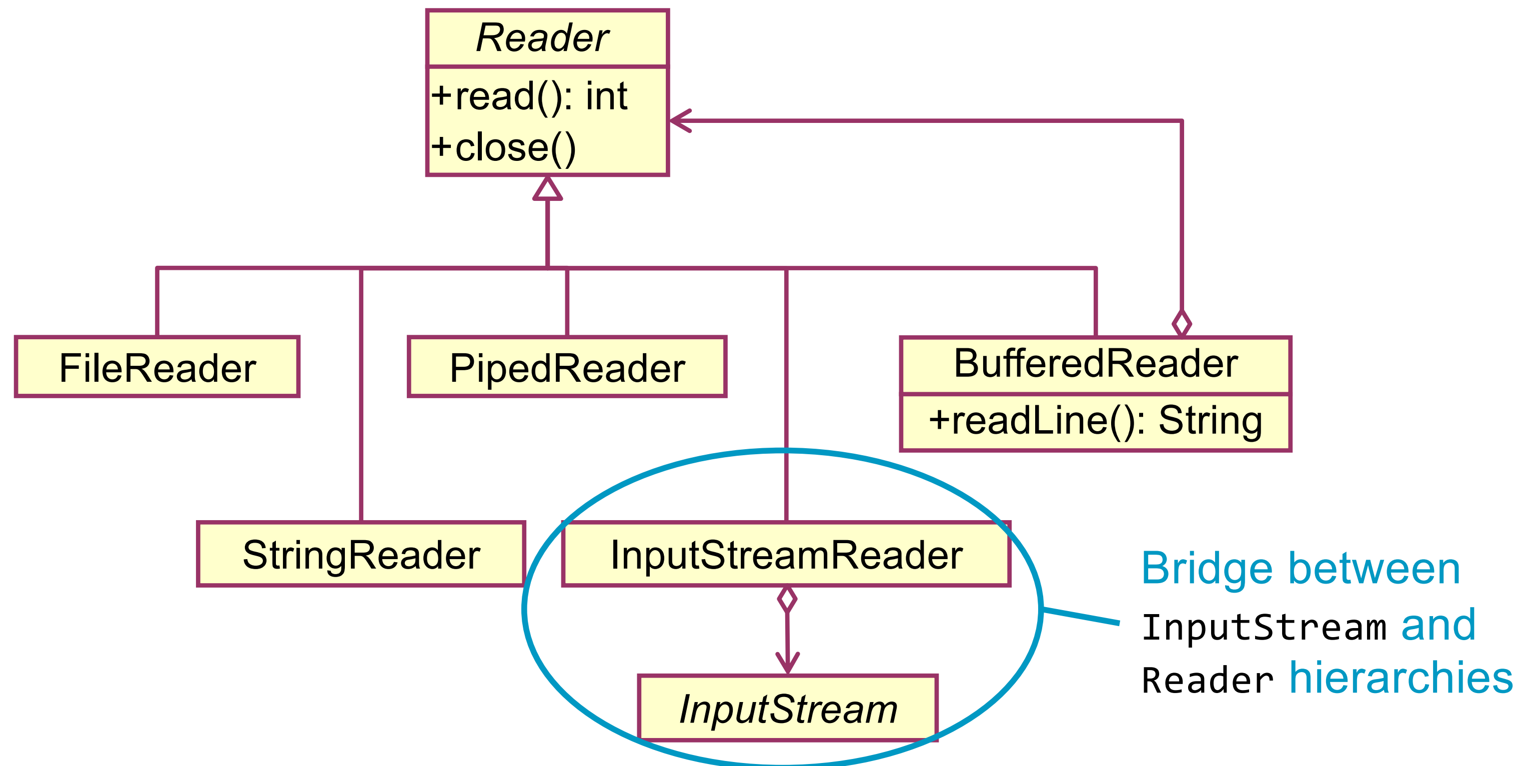
BYTE STREAM EXAMPLE

MORE COMPACT SOLUTION




```
String text = "";
boolean b = true;
double salary = 0.0;

try (DataInputStream datIn = new DataInputStream(
        new BufferedInputStream(new FileInputStream("data.dat")))) {
    text = datIn.readUTF();
    b = datIn.readBoolean();
    salary = datIn.readDouble();
} catch (IOException e) {
    e.printStackTrace();
}
```

READER (= CHARACTER INPUT) HIERARCHY



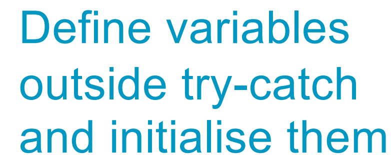
CLASS READER

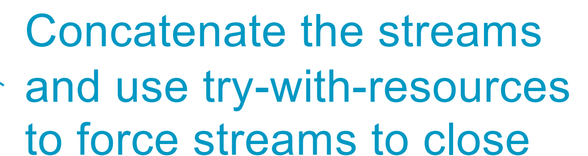
```
// abstract superclass
abstract class Reader implements AutoCloseable {
        // Reads a single character, as an integer
        // in the range 0 to 65535,
        // or -1 if the end of the stream has been reached
        int read() throws IOException { ... }

        // Closes the stream
        void close() throws IOException { ... }
}
```

Necessary to release resources
(specified in AutoCloseable)

TEXT STREAM EXAMPLE

```
String text = "";
boolean b = true;
double salary = 0.0;

    try (BufferedReader bufIn = new BufferedReader(
            new FileReader("data.txt"))) {
        text = bufIn.readLine();
        b  = Boolean.parseBoolean(bufIn.readLine());
        salary = Double.parseDouble(bufIn.readLine());
    } catch (IOException e) {
        e.printStackTrace();
    }
```



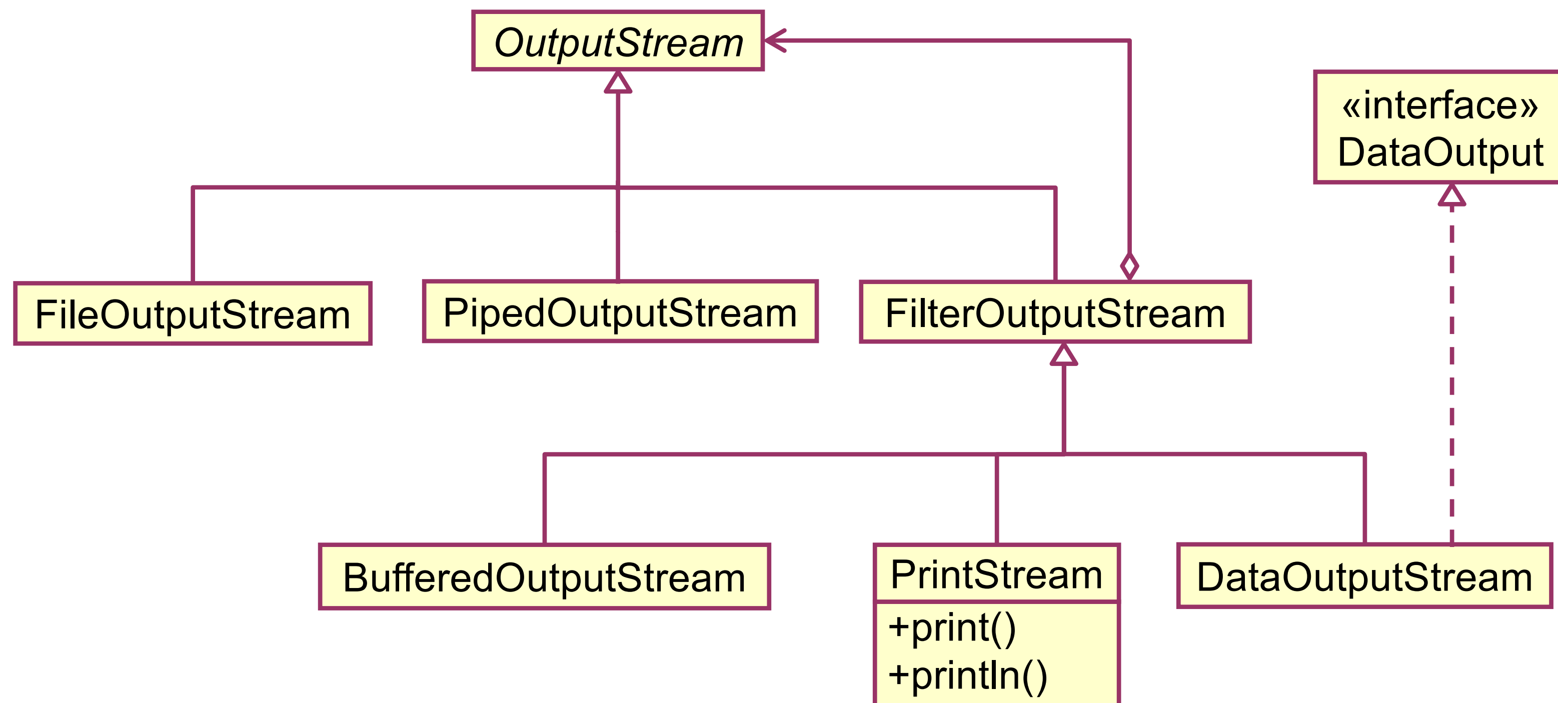


OUTPUT CLASSES

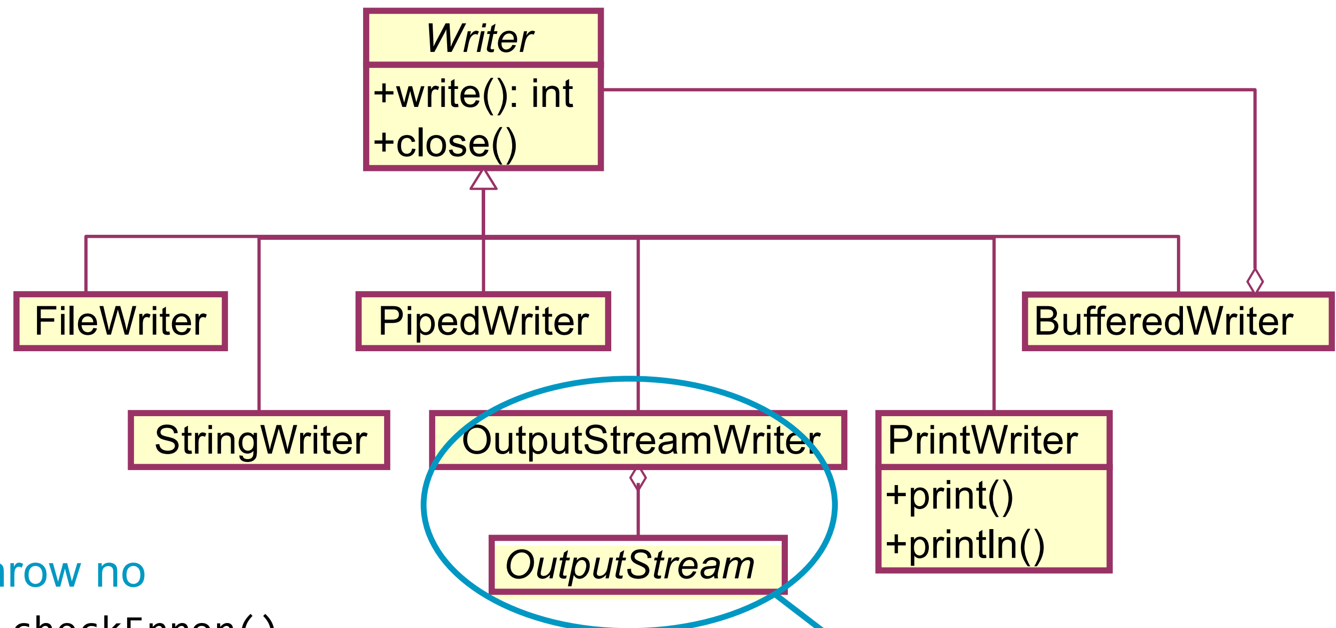
- Input and output are symmetric → corresponding classes
  - Byte streams: `InputStream` ↔ `OutputStream`
  - Char streams: `Reader` ↔ `Writer`
- Extra classes: `PrintWriter` and `PrintStream`
  - All kinds of methods: `print(char)`, `print(int)`, `print(boolean)`, `println(char)`, `printf(String, Object,…)`, etc.

print `Object` as formatted text, extremely handy, seen before

OUTPUT STREAM HIERARCHY



WRITER (= CHARACTER OUTPUT) HIERARCHY



`PrintWriter` methods throw no exceptions. Instead, `pw.checkError()` returns `true` if something when wrong

Bridge between `OutputStream` and `Writer` hierarchies

OBJECT I/O

READ AND WRITE COMPLETE OBJECTS

Interface without methods

- Objects should be made serializable → implement marker interface `Serializable`
- Java can read and write all values of object's fields with methods
  - `Object readObject()` of `ObjectInputStream` (`InputStream` subclass)
  - `void writeObject(Object)` of `ObjectOutputStream` (`OutputStream` subclass)

OBJECT I/O EXAMPLE

```
public class Student implements java.io.Serializable {
    private String name;
    private String number;
    ...
```

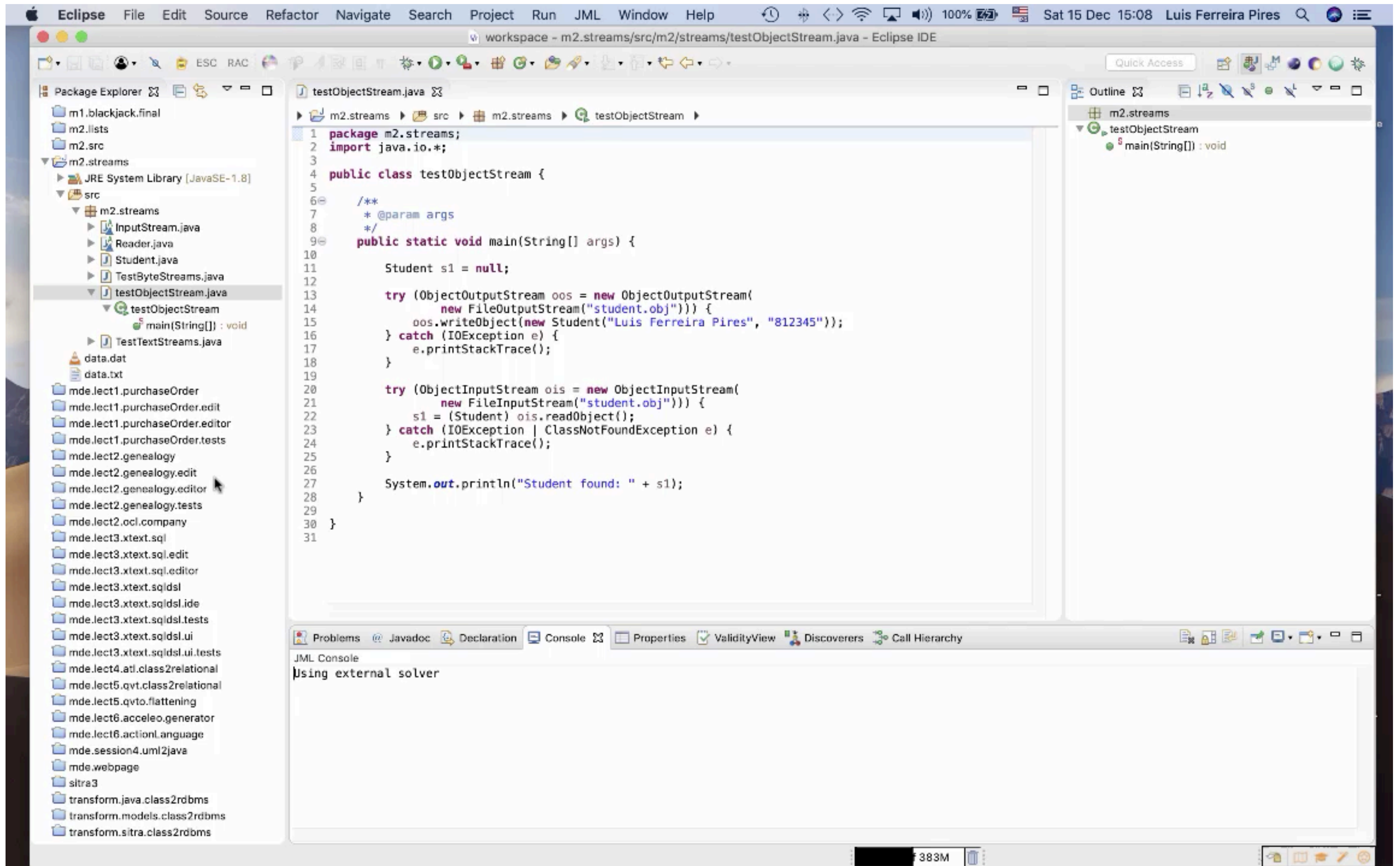
Serializable class Student

```
Student s1 = null;

try (ObjectOutputStream oos = new ObjectOutputStream(
        new FileOutputStream("student.obj"))) {
    oos.writeObject(new Student("Luis Ferreira Pires", "812345"));
} catch (IOException e) {
    e.printStackTrace();
}
```

Create output stream and write object to file

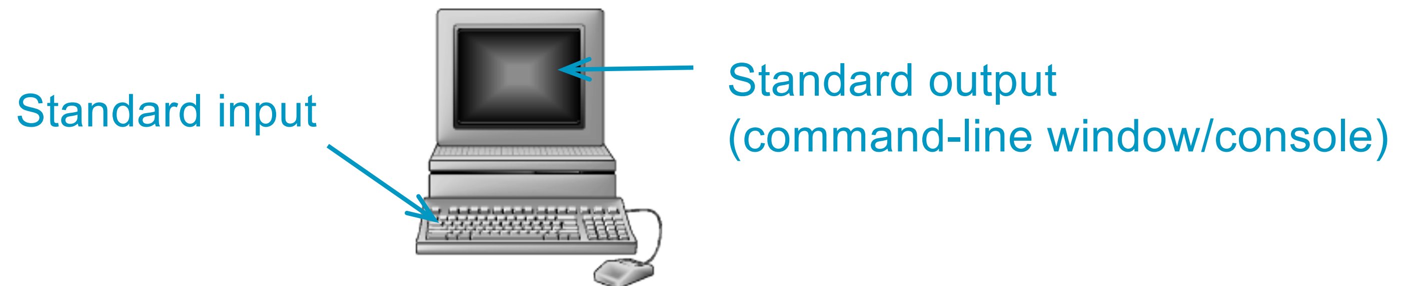
```
try (ObjectInputStream ois = new ObjectInputStream(
        new FileInputStream("student.obj"))) {
    s1 = (Student) ois.readObject();
} catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
}
```

Create input stream and read object from file



Eclipse File Edit Source Refactor Navigate Search Project Run JML Window Help
Sat 15 Dec 15:08 Luis Ferreira Pires
workspace - m2.streams/src/m2/streams/testObjectStream.java - Eclipse IDE
Package Explorer
testObjectStream.java
Outline
package m2.streams;
import java.io.*;
public class testObjectStream {
/**
* @param args
*/
public static void main(String[] args) {
Student s1 = null;
try (ObjectOutputStream oos = new ObjectOutputStream(
new FileOutputStream("student.obj"))) {
oos.writeObject(new Student("Luis Ferreira Pires", "812345"));
} catch (IOException e) {
e.printStackTrace();
}
try (ObjectInputStream ois = new ObjectInputStream(
new FileInputStream("student.obj"))) {
s1 = (Student) ois.readObject();
} catch (IOException | ClassNotFoundException e) {
e.printStackTrace();
}
System.out.println("Student found: " + s1);
}
}
Problems Javadoc Declaration Console Properties ValidityView Discoverers Call Hierarchy
JML Console
Using external solver

TEXTUAL USER INTERFACE



- Standard output
  - Static constant `java.lang.System.out` (or just `System.out`) of type `java.io.PrintStream`
  - Important methods: `print`, `println`, `printf`

STANDARD INPUT

- Static constant `System.in` of type `java.io.InputStream`
- Input should not only be read, but also interpreted
  - Split in lines, split lines in numbers, commands, etc.
- Alternative: auxiliary class `java.util.Scanner`

Already with `DataInputStream`

```
Scanner input = new Scanner(System.in);
while (input.hasNextLine()) {
    String line = input.nextLine();
    doSomething(line);
}
input.close();
```

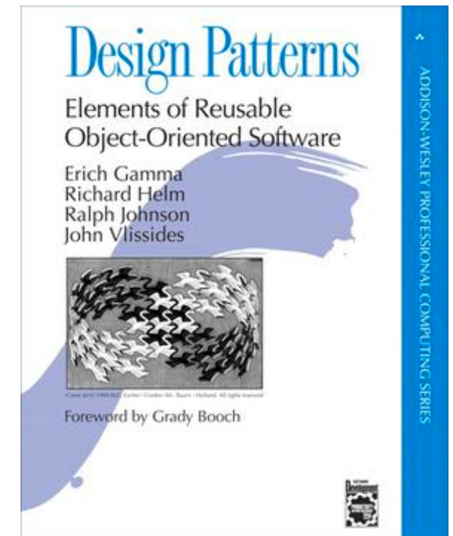
Also to parse `String`

Also `nextInt`, `nextBoolean`, etc.

DESIGN PATTERNS
ORIGIN

- Design (program) structure in which each class has a well-defined responsibility
- Often it is generally proven to solve a specific problem
- Reusability of solutions!

Christopher Alexander



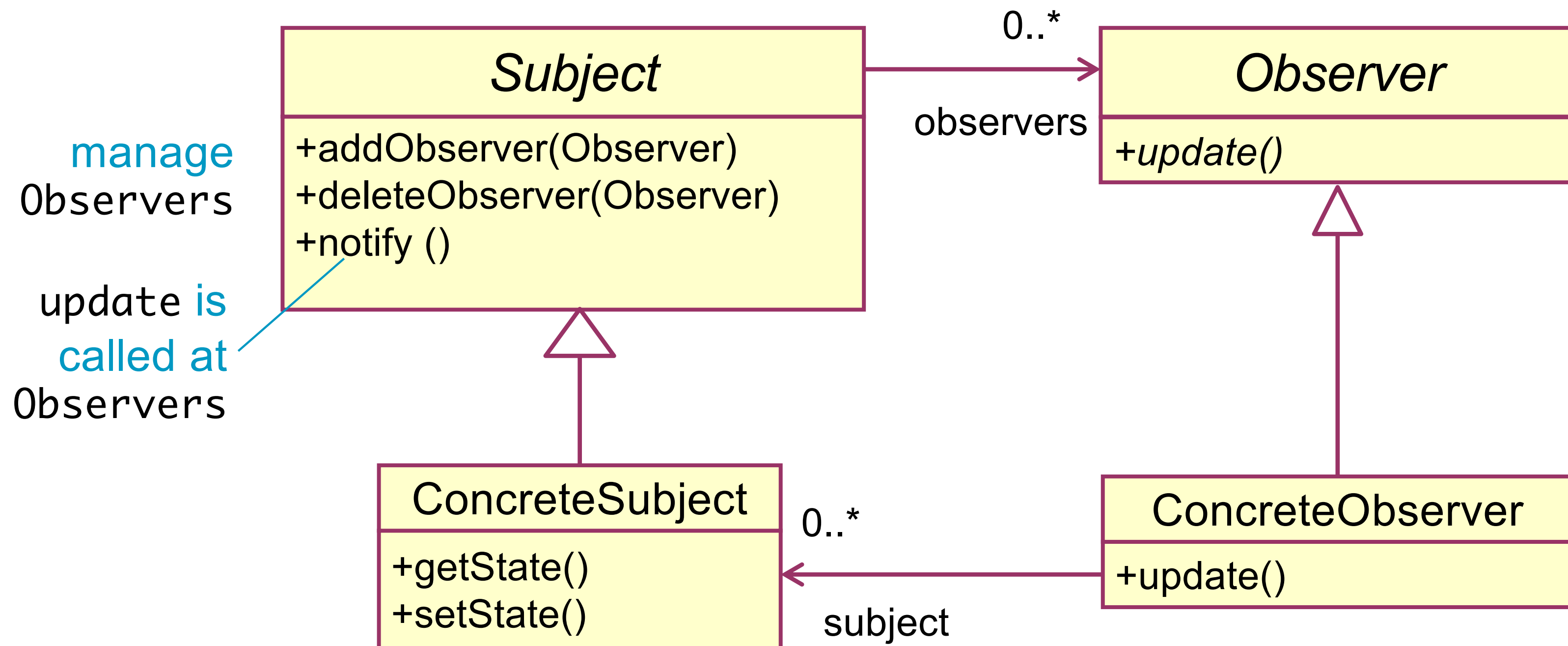
GoF Design patterns

OBSERVER PATTERN
CLASS RESPONSIBILITIES

- `Subject`: class where something happens (i.e., a change)
- `Observer`: class that should react to this change
- `Subject` does not know its `Observers`
  - Compare with Twitter
    - Tweet = shout into the ether, don’t care who listens
    - Follower = observer
- Also know as Publish/subscribe pattern or Notification service

OBSERVER PATTERN (GOF)

SUBJECT AND OBSERVER



OBSERVER PATTERN IN JAVA

- Java had an `Observer` interface and an `Observable` class to implement this pattern
- They were deprecated in Java 9 due to their limitations
  - Not thread-safe
  - `update()` method indicated that something was changed, not what
  - `Observable` was not serialisable
- Alternative is to use `Listeners`, like the `PropertyChangeListener`

OBSERVER PATTERN EXAMPLE

NEWSCHANNELS LISTENING TO NEWS AGENCIES

```
public class NewsAgency {
        private String news;
        private PropertyChangeSupport support;

        public NewsAgency() {
            support = new PropertyChangeSupport(this);
        }

        public void addPropertyChangeListener(PropertyChangeListener pcl) {
            support.addPropertyChangeListener(pcl);
        }

        public void removePropertyChangeListener(PropertyChangeListener pcl) {
            support.removePropertyChangeListener(pcl);
        }

        public void setNews(String value) {
            support.firePropertyChange("news", this.news, value);
            this.news = value;
        }
```

Implementation choice: delegation instead of extension

Notifies listeners when change occurs

OBSERVER PATTERN EXAMPLE

NEWS CHANNELS LISTENING TO NEWS AGENCIES

```
public class NewsChannel implements PropertyChangeListener {

    private String news;

    public void propertyChange(PropertyChangeEvent evt) {
        this.setNews((String) evt.getNewValue());
    }

    public String getNews() {
        return news;
    }

    public void setNews(String news) {
        this.news = news;
    }
}
```

Update state with value from event!
Called by `PropertyChangeSupport` object → callback

OBSERVER PATTERN EXAMPLE

NEWS CHANNELS LISTENING TO NEWS AGENCIES

```
NewsChannel channel1 = new NewsChannel();
NewsChannel channel2 = new NewsChannel();
NewsAgency agency1 = new NewsAgency();

// channel1 listens to agency1
agency1.addPropertyChangeListener(channel1);
NewsAgency agency2 = new NewsAgency();

// channels1 and 2 listen agency2
agency2.addPropertyChangeListener(channel1);
agency2.addPropertyChangeListener(channel2);

// changed the model state in agency1
agency1.setNews("fire");
System.out.println("Channel 1: " + channel1.getNews());

// changed the model state in agency2
agency2.setNews("earthquake");
System.out.println("Channel 1: " + channel1.getNews());
System.out.println("Channel 2: " + channel2.getNews());
```

Execution

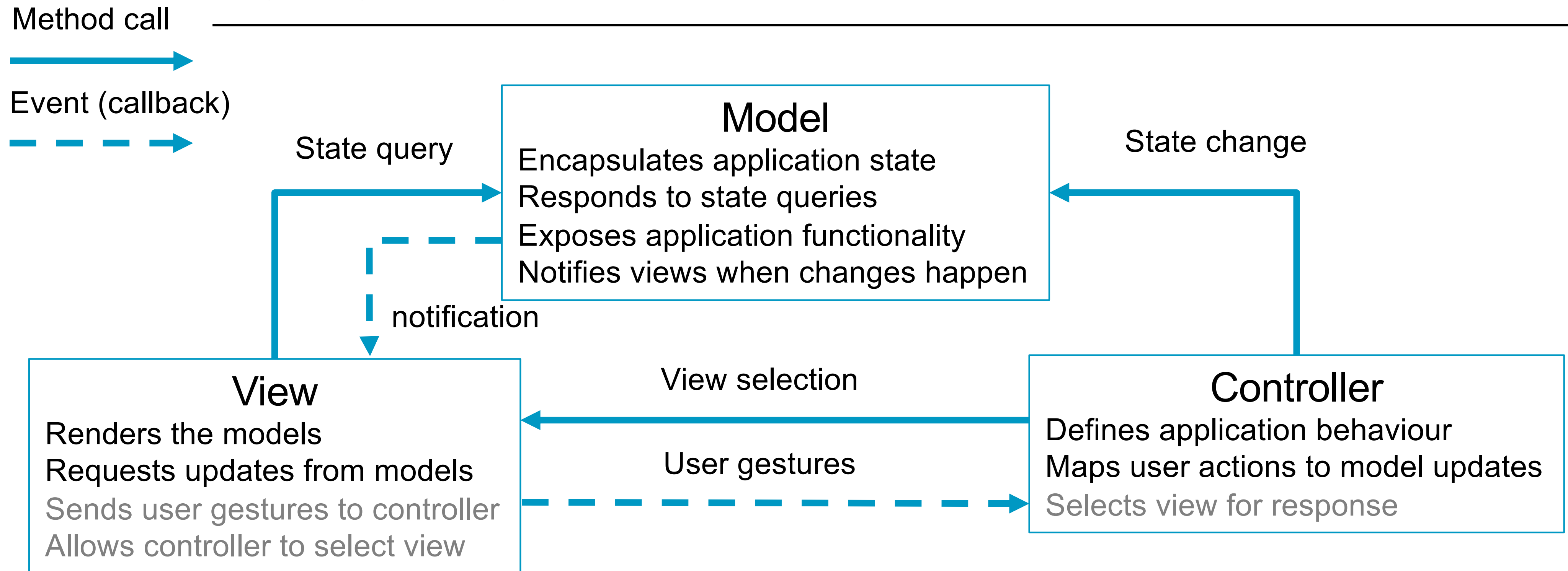
```
Channel 1: fire
Channel 1: earthquake
Channel 2: earthquake
```

MODEL-VIEW-CONTROLLER (MVC) PATTERN

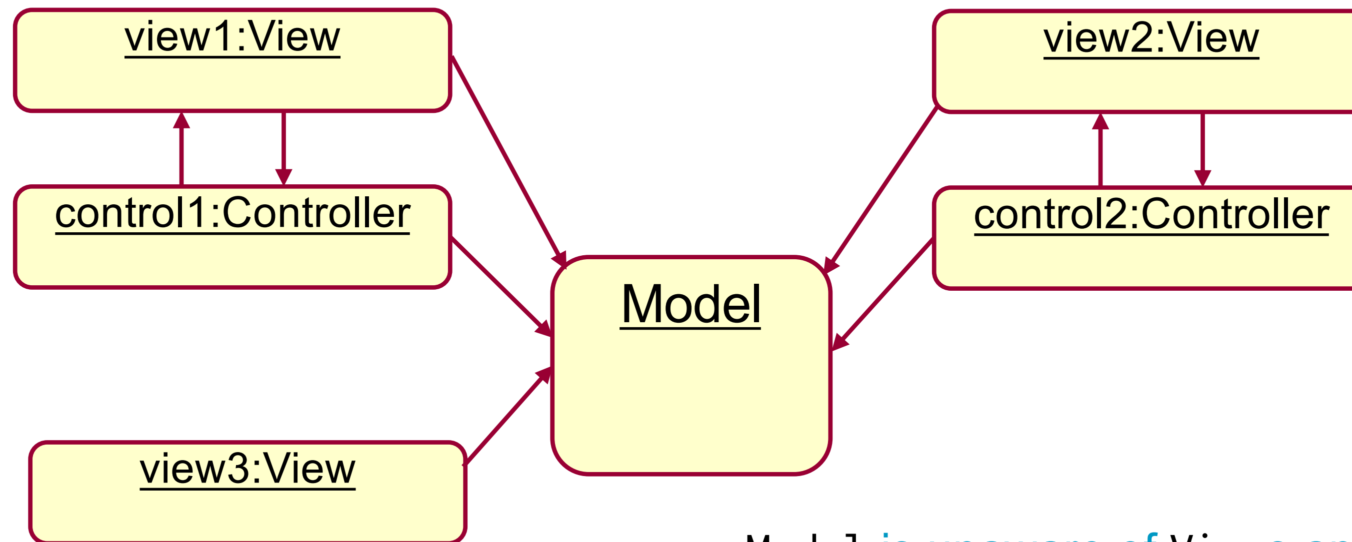
- Model objects: model the domain of the application
- View objects: visual representation of the model
  - Principle: Observer pattern
- Controller objects: register and process user input
  - Previous example: `main` method (calls `NewsAgency.setNews()`)
- View and controller may coincide
  - Especially in graphical user interface (GUI) elements

MODEL-VIEW-CONTROLLER (MVC) PATTERN

RESPONSIBILITIES



MVC PATTERN
ILLUSTRATION



Model is unaware of Views and Controllers

Views are Observers

Later: GUIs

TAKE HOME MESSAGES

- Classes to facilitate input and output
  - `Readers` and `Writers`: textual input/output
  - `Streams`: binary data
  - `ObjectStreams`: binary serialisable objects
- Standard input/output for Textual User Interfaces (TUI)
- Patterns for decoupling functionality
  - Observer pattern
  - MVC pattern, especially useful for reactive applications