

תוכנה 1

תרגול 10: IO ועוד
רובי בוים ומתי שמרת

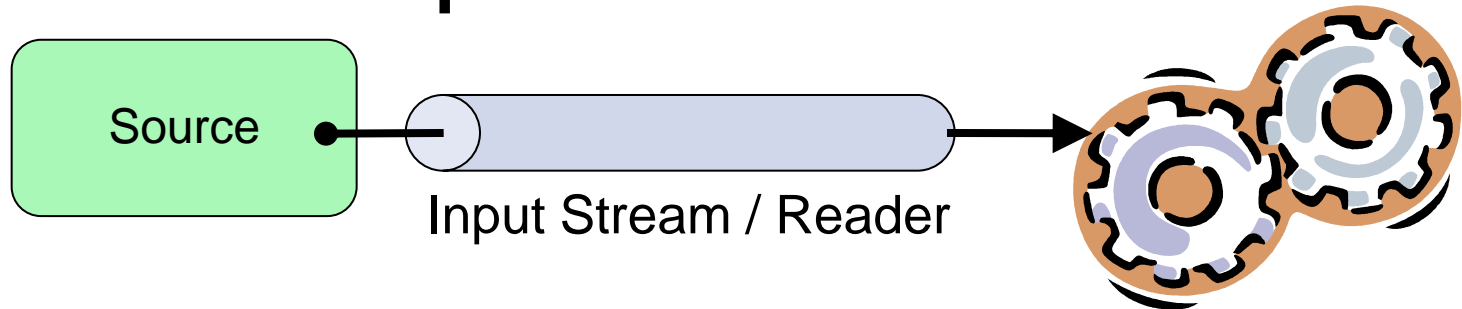
Today

- IO
 - Class hierarchy
 - Exceptions
 - Serialization

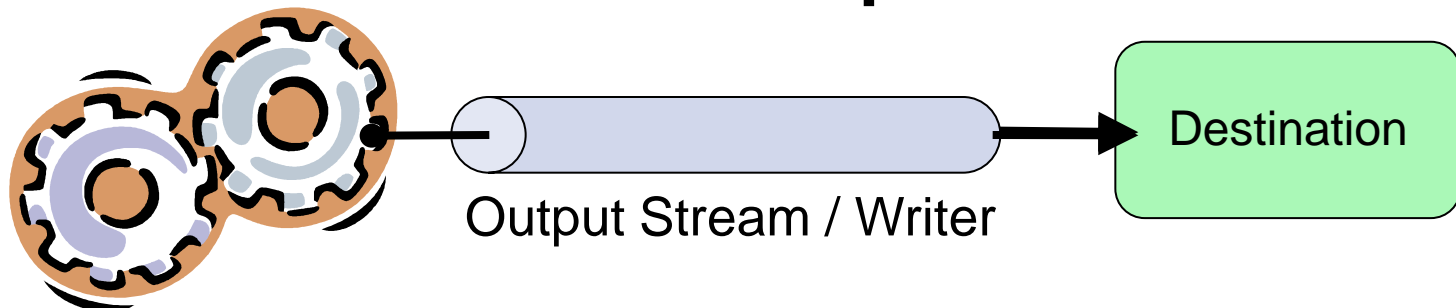
- Demystifying Enums

Streams Reminder

- A program that needs to read data a source needs an **input stream** or **reader**



- A program that needs to write data to a destination needs an **output stream** or **writer**

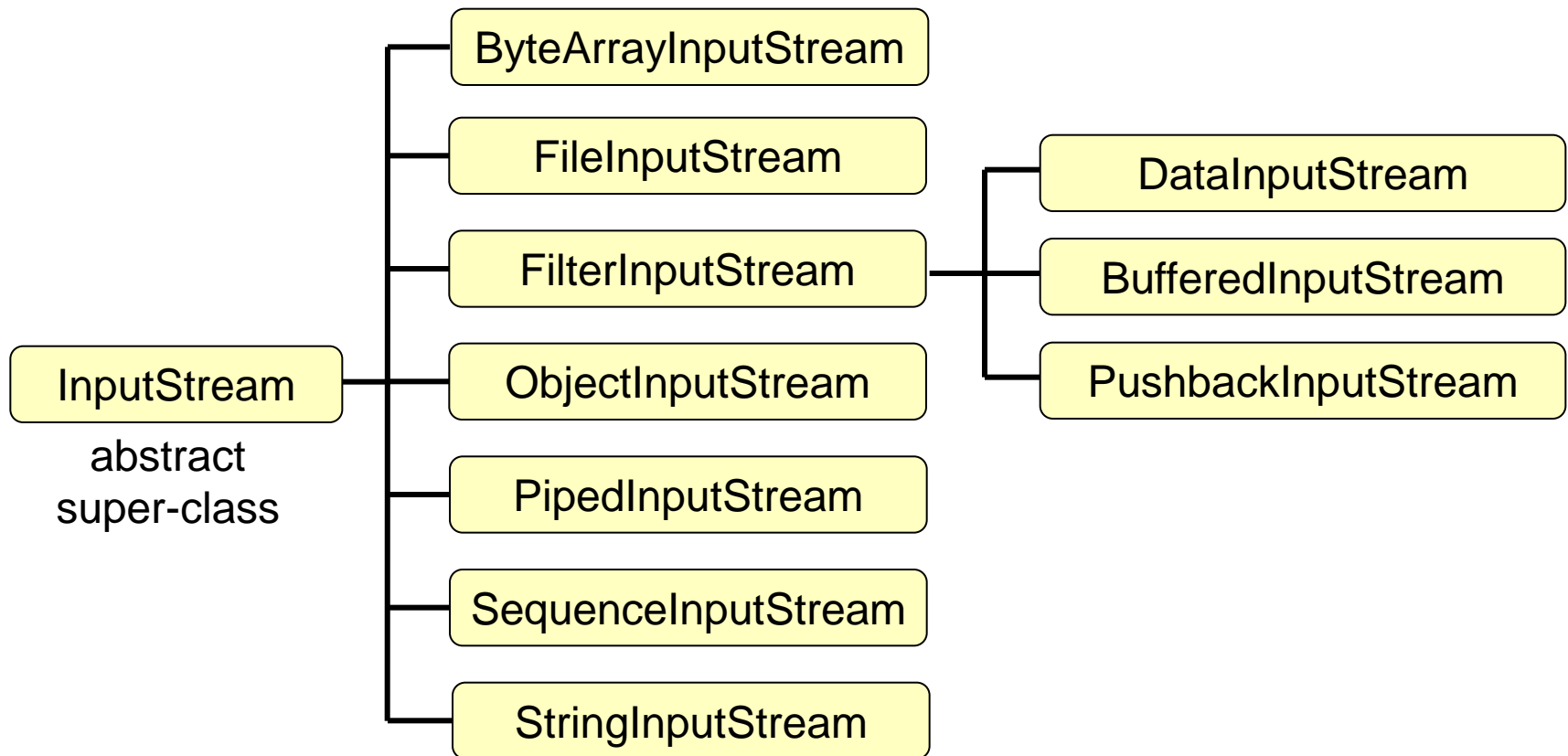


Streams

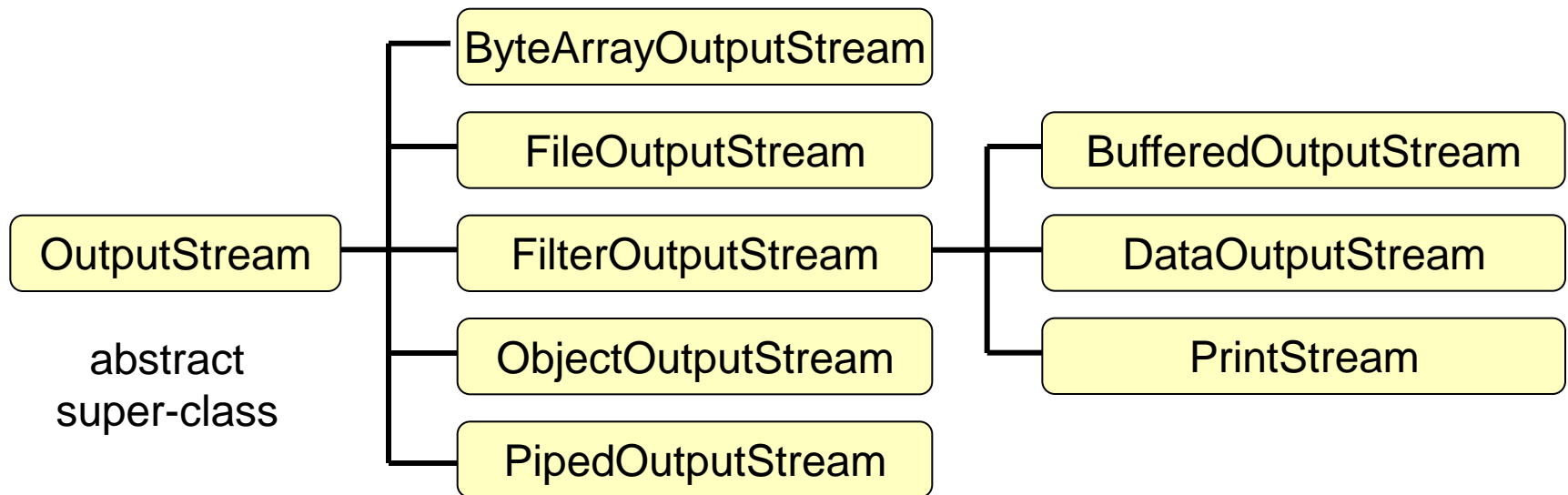
- There are two categories of streams:
 - **Byte streams** for reading/writing binary data
 - **Character streams** for reading/writing text
- Suffix Convention:

direction \ category	Byte	Character
Input	InputStream	Reader
Output	OutputStream	Writer

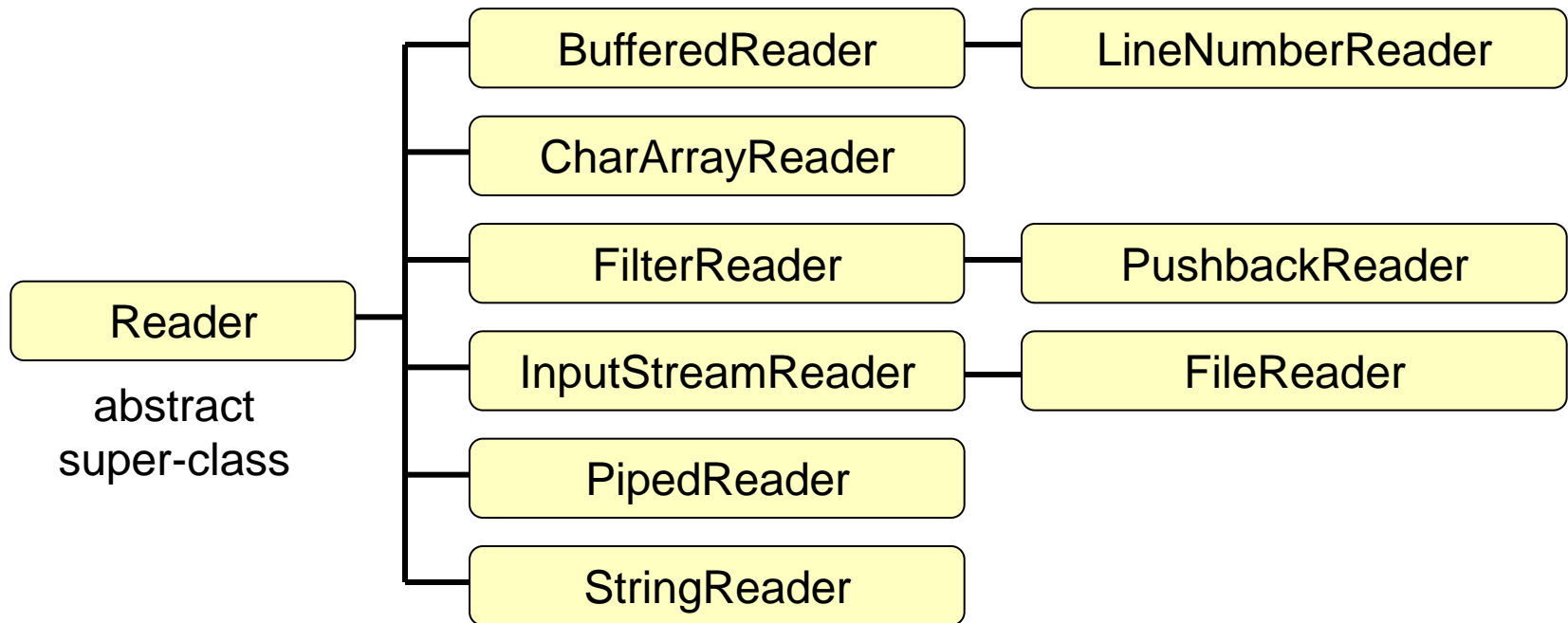
InputStream Class Hierarchy



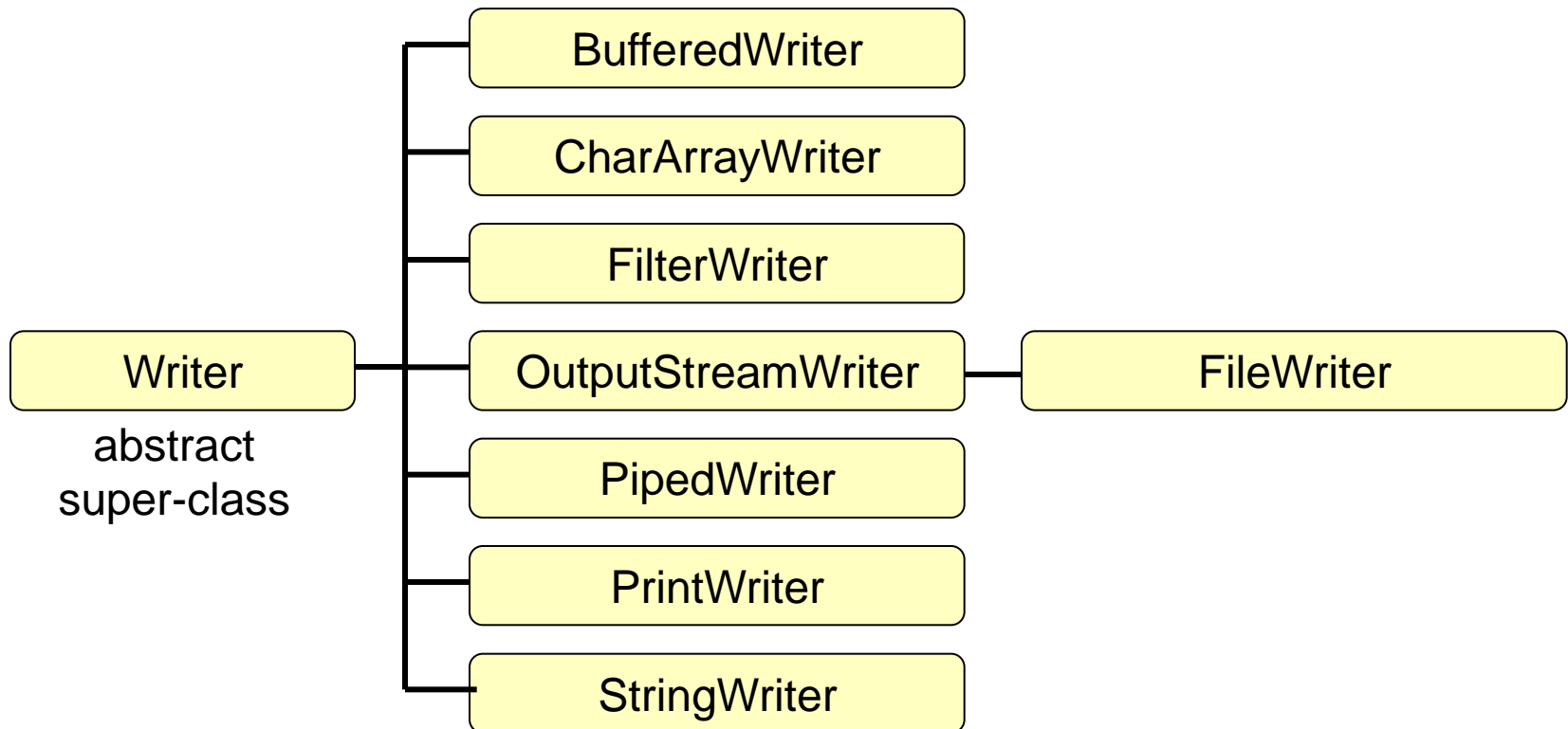
OutputStream Class Hierarchy



Reader Class Hierarchy



Writer Class Hierarchy



Handling Exceptions

- Handle exception
 - using a try-catch block
- Propagate the exception to the caller
 - Add throws declaration
- finally block is always executed at the end of the try block

Character Stream Example

```
public static void copy(String src, String dst)
    throws IOException {
    FileReader in = null;
    FileWriter out = null;
```

```
    try {
        in = new FileReader(src);
        out = new FileWriter(dst);

        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
    } finally {
        in.close();
        out.close();
    }
}
```

might throw an exception

might throw an exception

might throw an exception

Almost

```
public static void copy(String src, String dst)
    throws IOException {
    // copy input to output
} finally {
    closeIgnoringException(in);
    closeIgnoringException(out)
}
}

private static void closeIgnoringException(Closeable c) {
    if (c != null) {
        try {
            c.close();
        } catch (IOException e) {
            // Deliberately left empty; There is nothing we
            // can do if close fails
        }
    }
}
}
```

Stream Wrappers

- Some streams wrap other streams and add new features.
- A wrapper stream accepts another stream in its constructor:

```
DataInputStream din =  
    new DataInputStream(System.in) ;  
double d = din.readDouble() ;
```



Stream Wrappers Example

- Reading a line of text from a file:

```
try {  
    FileReader in =  
        new FileReader("FileReaderDemo.java");  
  
    BufferedReader bin = new BufferedReader(in);  
  
    String text = bin.readLine();  
    ...  
} catch (IOException e) {...}
```



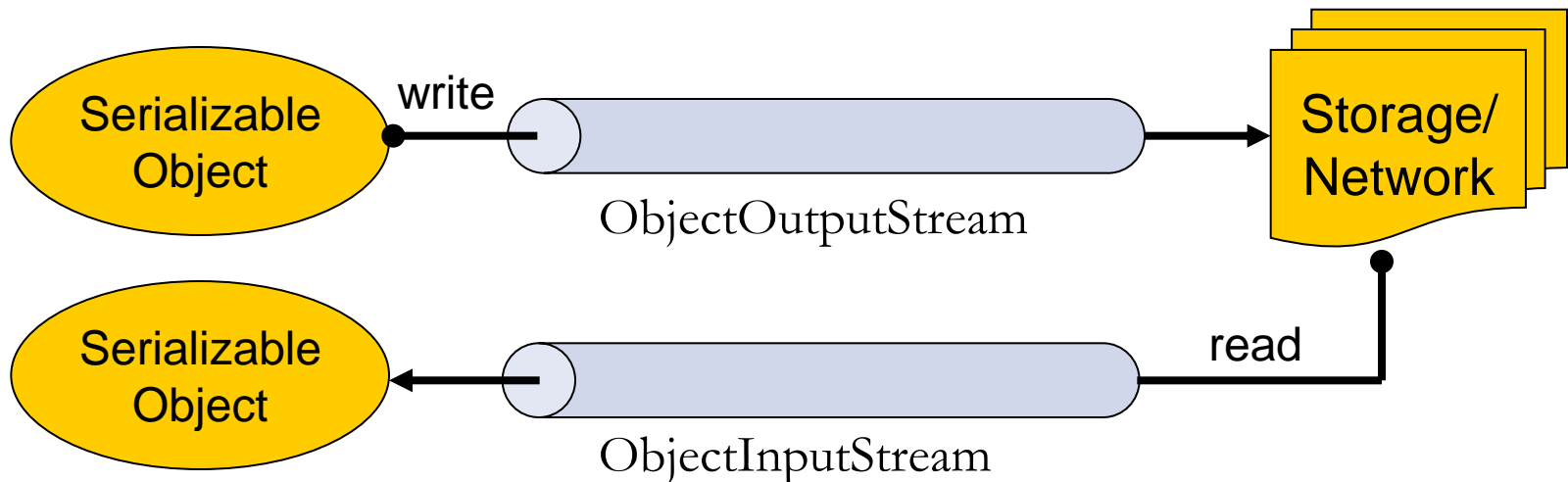
Object Serialization

- A mechanism that enable objects to be:
 - saved and restored from byte streams
 - persistent (outlive the current process)

- Useful for:
 - persistent storage
 - sending an object to a remote computer

The Default Mechanism

- The default mechanism includes:
 - The Serializable interface
 - The ObjectOutputStream
 - The ObjectInputStream



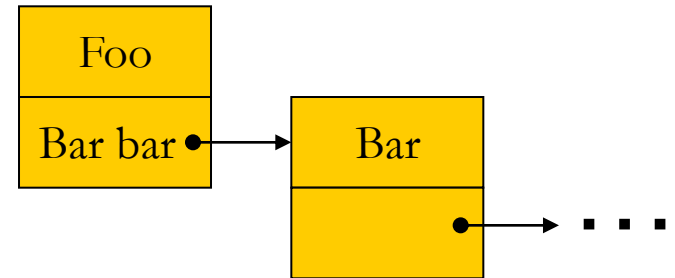
The Serializable Interface

- Objects to be serialized must implement the `java.io.Serializable` interface
- An empty interface
- Some types are `Serializable`:
 - Primitives, Strings, GUI components etc.
- Subtypes of `Serializable` types are also `Serializable`

Recursive Serialization

- Can we serialize a Foo object?

```
public class Foo implements Serializable {  
    private transient Bar bar;  
    ...  
}
```



```
public class Bar implements Serializable {...}
```

- No, since Bar is not Serializable
- Solutions:
 1. Implement Bar as Serializable
 2. Mark the bar field of Foo as transient
 3. Customize the serialization process

HashMap Serialization

```
Map<Integer, String> map = new HashMap<...>();  
...  
ObjectOutputStream out = null;  
try {  
    out = new ObjectOutputStream(  
        new FileOutputStream("map.s"));  
    out.writeObject(map);  
} catch (IOException e) {  
    ...  
} finally {  
    ...  
}
```

HashMap is Serializable, so are all the other **concrete** collection types we've seen

Reading Objects

```
ObjectInputStream in = null;
try {
    in = new ObjectInputStream(
        new FileInputStream("map.s"));
    Map<Integer, String> map =
        (Map<Integer, String>) in.readObject();
    System.out.println(map);
} catch (Exception e) {
    ...
} finally {
    ...
}
```

Demystifying Enums

- Enums are just syntactic sugar
- We could emulate an Enum with a class
 - This is what the compiler does

```
public enum Operation {
    PLUS("+") { public double apply(double x, double y) {return x + y;} },
    MINUS("-") { public double apply(double x, double y) {return x - y;} },
    TIMES("*") { public double apply(double x, double y) {return x * y;} },
    DIVIDE("/") { public double apply(double x, double y) {return x / y;} };

    private final String symbol;

    Operation(String symbol) { this.symbol = symbol; }
    public String toString() { return symbol; }

    public abstract double apply(double x, double y);
}
```

Disassembling Operation

```
public abstract class Operation extends Enum {
    private Operation(String s, int i, String symbol) {
        super(s, i);
        this.symbol = symbol;
    }

    public static Operation[] values() {
        Operation aoperation[];
        int i;
        Operation aoperation1[];
        System.arraycopy(aoperation = ENUM$VALUES, 0,
        aoperation1 = new Operation[i = aoperation.length], 0, i);
        return aoperation1;
    }

    ...
}
```

See the code on the course site.