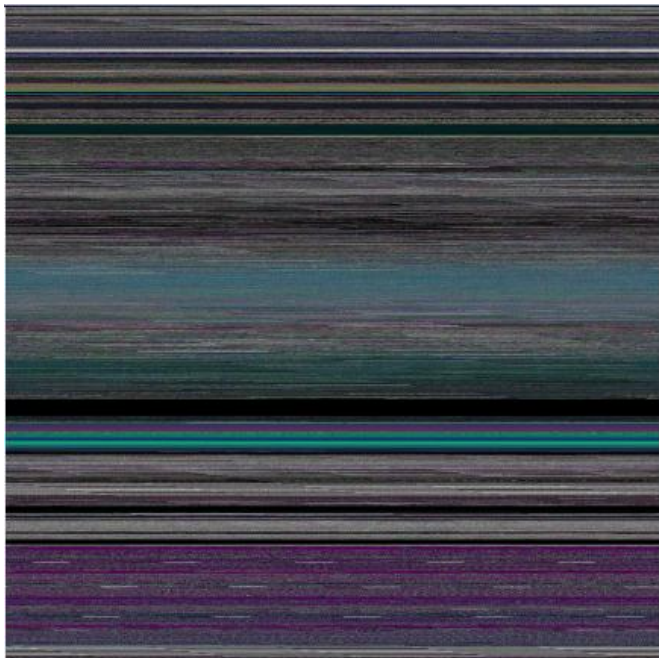


# Voortgezet Programmeren

## Lecture 2: Errors, exceptions and streams

Tommi Tervonen

Econometric Institute, Erasmus School of Economics



- Type system forms the core of all programming languages
- Strong typing reduces bugs by detecting errors at compile-time
- Primitive values have a single type (int, double)
- Objects can be of multiple types (Student is also an Object)

- Method calls can fail due to
  - external factors uncontrollable for the programmer
  - violation of a pre-condition
- How to signal failure?

# Return value outside the valid set

```
public int lengthOfString(String str) { ... }
```

```
public int parseInt(String str) { ... }
```

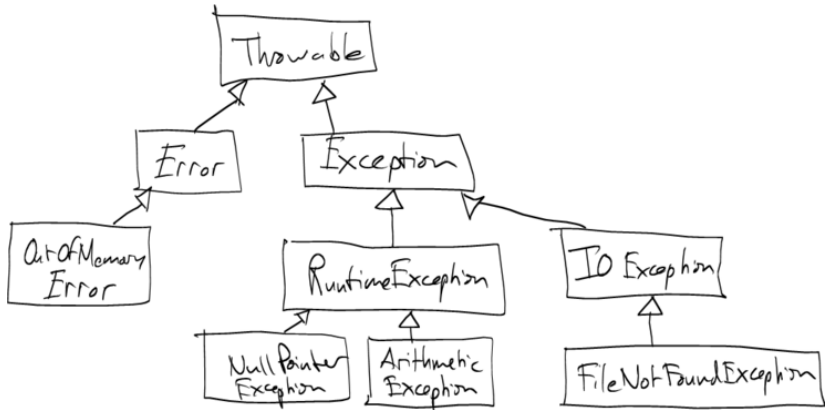
# Abstraction of failure: Exception

```
public int parseInt(String str)
    throws NumberFormatException { ... }
...
try {
    int myint = parseInt("1");
    myint = parseInt("5.5");
} catch (NumberFormatException e) {
    System.out.println("Cannot parse: "
        + e.getMessage());
}
```

# Checked exceptions

- **Checked** exceptions are ones that the program has to be able to recover from (caught)
- **Unchecked** exceptions are serious errors (out of memory) or logic errors (method call with invalid input values) that the program should not recover from (note: rule not always adhered to)
- Unchecked exceptions in Java are in the type hierarchy below RuntimeException or Error

# Exception hierarchy (part)





```
public int add(int a, int b) {  
    int c = a;  
    c += b;  
    if (c != (a + b)) {  
        throw new VirtualMachineError();  
    }  
    return c;  
}
```

# If you can't handle it, delegate

```
/**
 * Loads in the file for doing something.
 *
 * @param filename The file to read contents from
 * @throws IOException If the file could not be
 *                   read successfully
 */
public void loadFromFile(String filename)
    throws IOException {

    FileInputStream f =
        new FileInputStream(filename);
    ... // read file in and do something
}
```

```
public class MyDatabase {  
    ...  
  
    public MyDatabase(String filename)  
        throws FileNotFoundException, IOException {  
  
        FileInputStream f =  
            new FileInputStream(filename);  
  
        ... // read file in and do something  
  
    }  
    ...  
}
```

# Making your own exceptions

```
package fi.smaa.libror;  
  
@SuppressWarnings("serial")  
public class SamplingException extends Exception {  
  
    public SamplingException(String reason) {  
        super(reason);  
    }  
}
```

# Using your own exceptions

```
package fi.smaa.libror;  
  
public class RejectionValueFunctionSampler ... {  
    ...  
    private FullValueFunction sampleValueFunction()  
        throws SamplingException {  
        ...  
  
        throw new SamplingException("Cannot sample a"  
+ "VF within " + maxTries + " iterations");  
    }  
}
```

## Too many exceptions

```
public QResult execQuery(Connection c, String q)
    throws ServerbusyException ,
        QueryIncorrectException ,
        QuerySyntaxException
{ ... }
```

```
public QResult dbQuery(String serv, String query)
    throws ServerbusyException ,
        QueryIncorrectException ,
        QuerySyntaxException ,
        IncorrectServerAddressException ,
        NetworkException
{
    Connection c = openConnection(serv);
    QResult r = c.execQuery(c, "SELECT * FROM db");
    ...
}
```

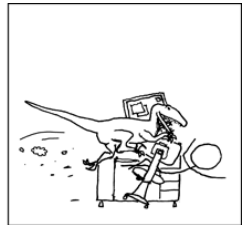
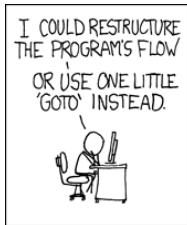
# Exception wrapping

```
public class QueryException extends Exception {  
    public QueryException(Exception e) {  
        super(e);  
    }  
}
```

```
public QResult execQuery(Connection c, String q)  
    throws QueryException { ... }
```

```
public QResult dbQuery(String serv, String query)  
    throws QueryException { ... }
```

# Aren't exceptions like GOTO?

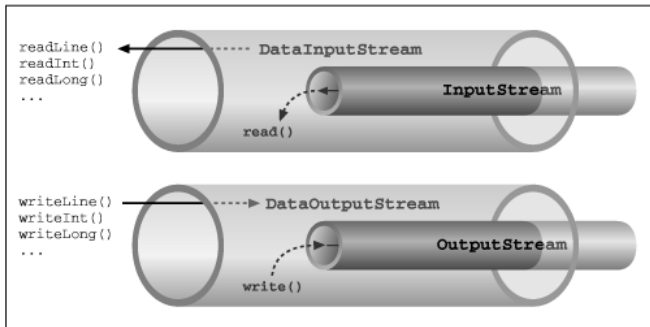




- Streams are abstraction of input/output in OOP
- Allow transparently to change transmission/storage media
- `System.out` is a `PrintStream`
- Character sources/sinks can be read/written with `Readers/Writers`

# Stream combination

- Lower level streams can be used by higher level streams to provide additional functionality
- Streams can perform input conversion transparently on the fly



```
BufferedReader rdr = null;
try {
    rdr = new BufferedReader(
        new FileReader("file.txt"));
    String s = null;
    do {
        s = rdr.readLine();
        if (s!=null){System.out.println("Read:" + s);}
    } while (s != null);
} catch (FileNotFoundException e) {
    System.out.println("file.txt not found");
} catch (IOException e) {
    System.out.println("Error reading file: "
        + e.getMessage());
} finally {
    try {
        if (rdr != null) {
            rdr.close();
        } } catch (IOException e) { }
}
```

Do not ever use Scanner

# Separation of concerns

Premature optimization is the root of  
all evil

D.E. Knuth