Transparency and Asynchronous Method Invocation

Pierre Vignéras

pierre@vigneras.name

Assistant Professor at the
Ghulam Ishaq Khan Institute of
Engineering Sciences and Techonology,
Pakistan

Member of the Java Community Process
Expert Group JSR 236/237

Transparency, Asynchronous Method Invocation, Concurrency

Outline

I. Introduction
II. Full-Transparency
III. Semi-Transparency
IV. Conclusion

Focus of our work
(Mandala: http://mandala.sf.net)

Consider *Object Oriented Languages* only

	local	remote
synchronous	MI	RMI
concurrent	Threads Event-driven AMI	ARMI

ARMI → Leads naturally to ***parallelism***

MI: Method Invocation
AMI: Asynchronous Method Invocation
ARMI: Asynchronous Remote Method Invocation

Dealing with concurrency

- AMI is just a way to express concurrency
  - it does not solve problems commonly found in concurrency (locks, mutex, condition variables...)
- Distinguish the mechanism that provides *concurrency* from the one that provides *transparency*
  - actors, active objects, guardians, separates, active containers, asychronous references, ... can be used for the former
  - we call the *concurrency* mechanism the *abstraction* in this work

Introduction

- Transparency: hiding a non-functional aspect to the developper
- Example: Java-RMI

```
try{
    MyInterface object = java.rmi.Naming.lookup(...);
    MyResult result = object.myMethod(myArgs);
}catch(RemoteException e) {...}
```

What makes this code appear a remote call?

Very easy to hide!

```
try{
    MyInterface object = getInstance(...);
    MyResult result = object.myMethod(myArgs);
}catch(Exception e) {...}
```

Common Coding Practice!

Introduction

- Transparency is usefull in the context of distributed programming (success of RMI, CORBA and .NET)
  - Transparency is usefull in other contexts: *persistency* (serialization), *transaction*, ...
- Is it also true in the context of **concurrent programming?**
  - Does the hiding of (some) complexities of concurrent aspect ease the making of highly concurrent applications?
- Concurrency: a major concern in the next decade
  - new *n-ways* processor (dual-core, HT, CMT)
  - Best performance with highly concurrent applications

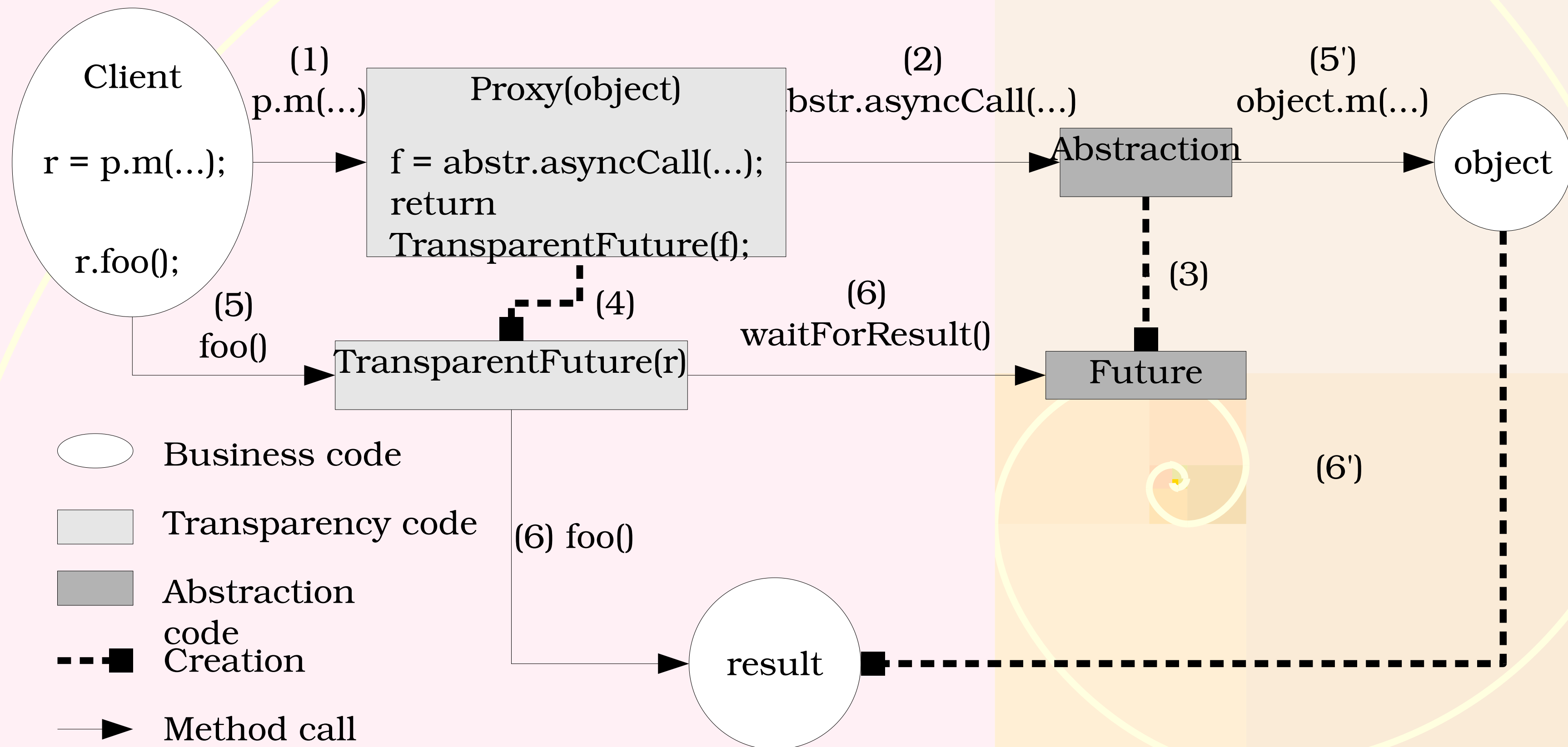
Outline

I. Introduction
II. Full-Transparency
III. Semi-Transparency
IV. Conclusion

Full Transparency


```
Type p = new Type();
Result r = p.m(args);
doingSomething();
Info i = r.foo();
```

Syntax is not distinguishable from a standard MI



Inheritance (ProActive)

- For a given class C produce a class ProxyC that inherits C and overwrites all its public methods so they become asynchronous

```
class Type {
    public Result m(A arg) {
        ...
    }
}
```

```
class ProxyType extends Type {
    private Abstraction a;
    public Result m(A arg) {
        Future f = a.call(...);
        return new ProxyResult(future);
    }
}
```

Limitations:

- final classes (9 %) cannot be extended
- final methods (5 %) cannot be overwritten
- primitive type method return (27 %)
- access to public fields (8 % non final)

Inherits also the returned class to provide *wait by necessity* mechanism

Composition (Mandala)

- For a given **interface** I provide an asynchronous implementation that uses a business implemention of I

```
interface Type {
    public Result m(A arg);
}
```

```
class ProxyType implements Type {
    private Abstraction a;
    public Result m(A arg) {
        Future f = a.call(...);
        return new ProxyResult(future);
    }
}
```

Previous limitations solved:

- interfaces can always be implemented
- their methods can always be implemented
- no public field in interfaces (or final)

Constraints:

- Only for interfaces (20 %)
- return type of interface methods (5 %) must be interfaces (4 %)

Implements also the **interface** declared as the return type to provide *wait by necessity* mechanism

Full-Transparency and Exceptions

```
byte[] b;
java.io.FileOutputStream fos = ...;
try{
    while((b = getBytes()) != null) {
        fos.write(b);
    }
}catch(IOException ioe) {
    ...
}
```

If the call `fos.write()` is asynchronous (hence fully-transparent), the thrown of an exception may not be catched:

- the caller thread may have reached the `catch` statement for a while!

Solution (ProActive): any method that is declared with a 'throws' statement becomes synchronous

- does not solve the problem for unchecked exceptions

The *Developper Consciousness* Problem

Sequential Code

```
Type o = new Type();
Result r = o.m(args);
doingSomething();
Info i = r.getInfo();
```

Natural Code

```
Type o = new Type();
Result r = o.m(args);
Info i = r.getInfo();
doingSomething();
```

- Sequential Programming Schema:
  - ➔ invoke method as needed
  - ➔ use their result as soon as possible
  - ➔ ***reduce variable scope***

- AMI Programming Schema:
  - ➔ invoke method as soon as possible
  - ➔ use their result as late as possible
  - ➔ ***increase concurrrency***

Full-transparency does not lead to any concurrency in legacy code!

Outline

I. Introduction
II. Full-Transparency
III. Semi-Transparency
IV. Conclusion

Proposition: semi-transparency

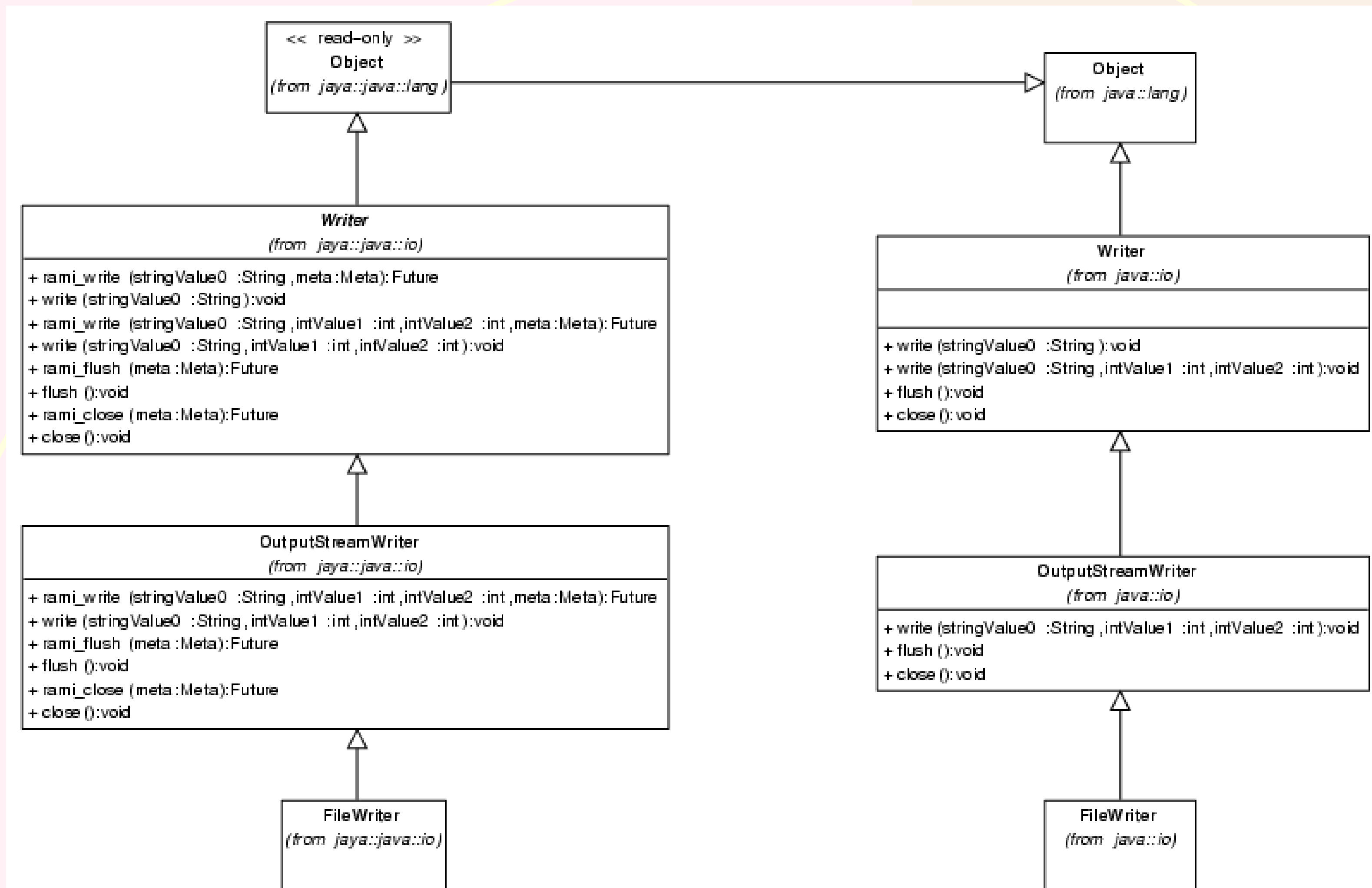
- For a given method 'Result m(A arg)', its AMI version has the following signature

```
public Future<Result> #m(A arg, Meta m);
```

  - Future has the "usual" meaning
  - Meta contains at least an
    - exception handler
    - a callback
- For a given class `p.C` produce its asynchronous version `jaya.p.C` (consisting of the AMI version of all its public methods)
  - Provides also the synchronous version for convenience
  - Produce all the hierarchy of type until `jaya.java.lang.Object`
  - No need for source Java file, use reflection.

Proposition: semi-transparency

- Homonym short class names enforce ***developper consciousness***



Example #1
write optimisations

Making up the writes

```
java.io.File file = ...
java.io.Writer writer =
           new java.io.FileWriter(file);
try{
```

Creation of the business object (FileWriter)

Business code

```
    while(true) {
        String s = getStringToWrite();
        if (s == null) break;
        writer.write(s);
    }

    doSomethingElse();
    writer.flush();
```

Write on each iteration

Independant operations

Real writes

Termination

```
}finally {
  if (writer != null) writer.close();
}
```

Freeing ressources

Example #1
write optimisations

III. Semi-Transparency

Making up the writes
Making up the writes

```
java.io.File file = ...
java.io.Writer writer =
           new java.io.FileWriter(file);
try{
```

Business code
Business code

```
    while(true) {
        String s = getStringToWrite();
        if (s == null) break;
        writer.write(s);
    }

    doSomethingElse();
    writer.flush();
```

Termination
Termination

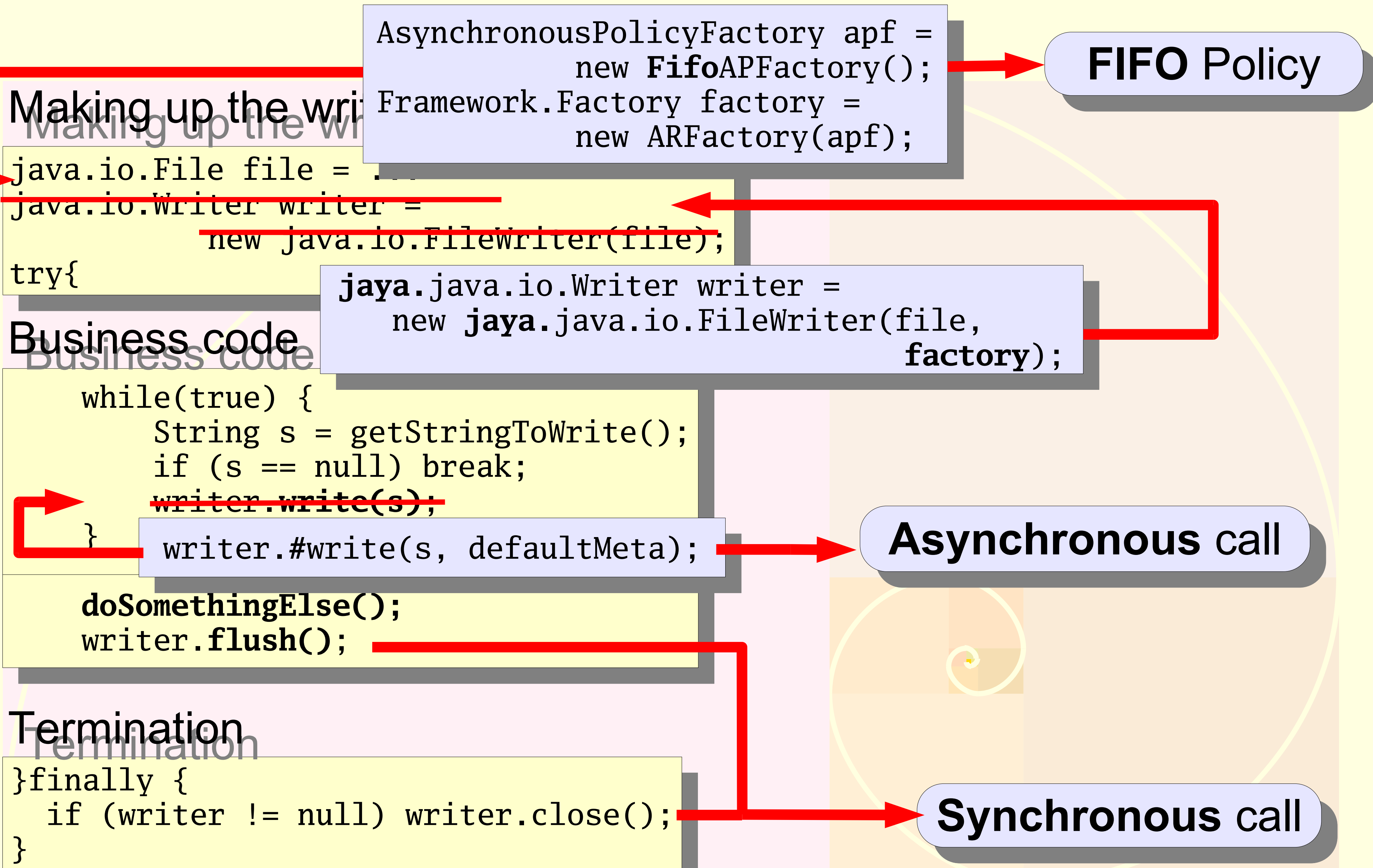
```
}finally {
  if (writer != null) writer.close();
}
```

How to decrease the time of the `while()` loop?

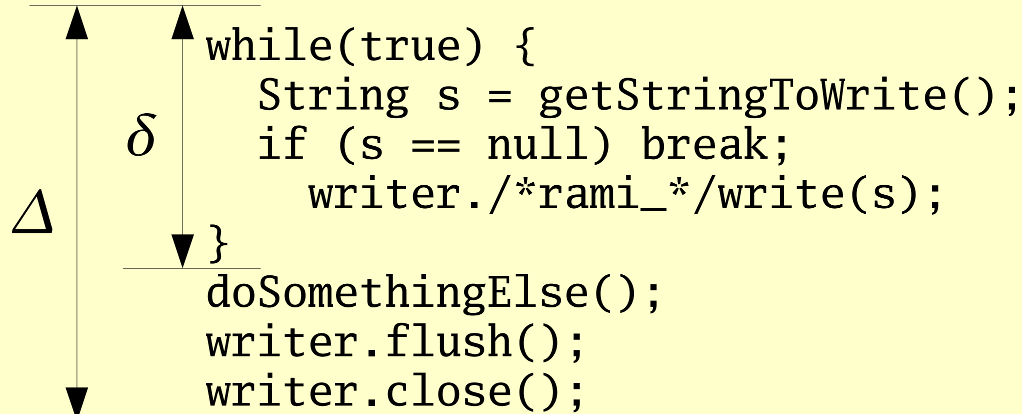
Using a buffer (`java.io.BufferedWriter`)

Knowing the good buffer size is not easy (may depend on data)

Example #1
write optimisations



Example #1
write optimisations



```
while(true) {
  String s = getStringToWrite();
  if (s == null) break;
    writer./*rami_*/write(s);
}
doSomethingElse();
writer.flush();
writer.close();
```

$$E=\frac{\Delta}{\delta}, D=\frac{|s|*n}{\Delta}$$

$$S=\frac{E_{asynchronous}}{E_{synchronous}}$$

$$O=\frac{\Delta_{asynchronous}}{\Delta_{synchronous}}-1$$

Test: 100 executions
10 Ko ≤ |s| ≤ 20 Ko
1000 ≤ n ≤ 2000

Higher E is better
$E_{synchrone} \simeq 1$,
$S \simeq E_{asynchrone}$

Call number (n)	1487
String size (|s|)	15.5 Ko
Total size (|s| * n)	23 Mo
Rate (D)	850 Ko/s
***Speedup* (S)**	**854**
***Overhead* (O)**	**2.82 %**

Example #2
Event-Driven GUI Programming

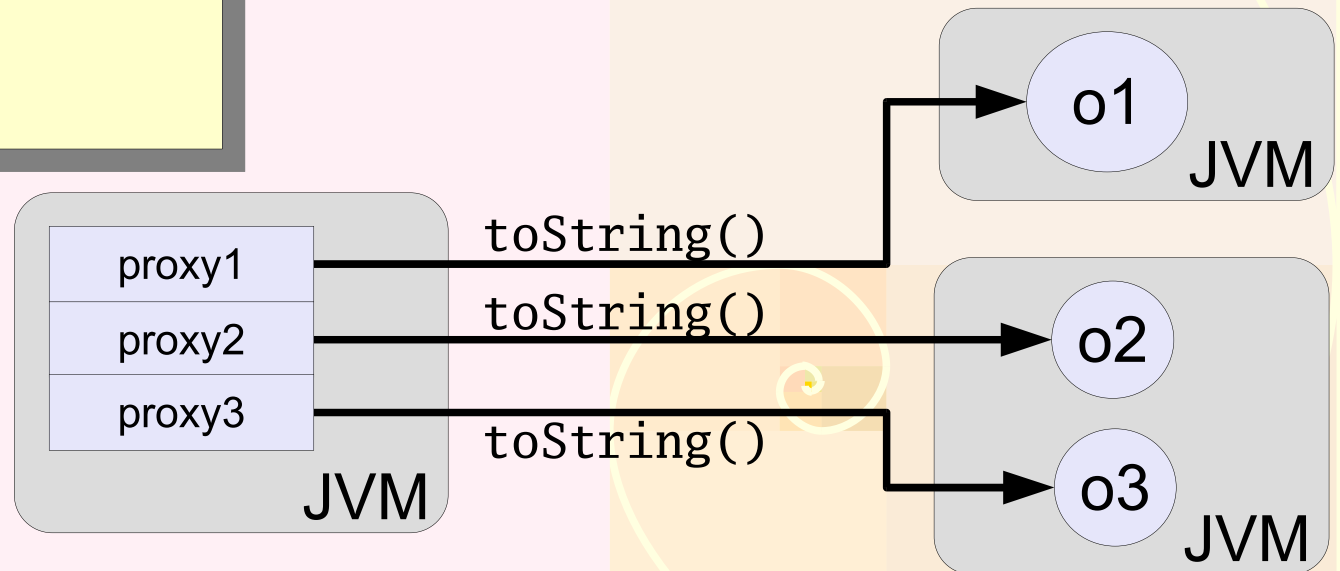
Printing an object list

```
JPanel panel = new JPanel();
final Collection c = ...;
Iterator i = c.iterator();
while(i.hasNext()) {
  Object o = i.next();
  JLabel label = new JLabel();
  String s = o.toString();
  label.setText(s);
  panel.add(label);
}
```

Local case

o1.toString()
o2.toString()
o3.toString()

Remote case



Problem related to **network latency** in the remote case!

Example #2

Event-Driven GUI Programming

Printing an object list

```
JPanel panel = new JPanel();
final Collection c = ...;
Iterator i = c.iterator();
while(i.hasNext()) {
  Object o = i.next();
  JLabel label = new JLabel();
  String s = o.toString();
  label.setText(s);
  panel.add(label);
}
```

```
jaya.java.lang.Object o =
   (jaya.java.lang.Object) i.next();
```

```
JLabel label = new JLabel("Waiting...");
```

```
o.rami_toString(new Callback() {
  public void done(InvocationInfo info,
                   MethodResult result){
    String s = result.getResult();
    label.setText(s);
    label.revalidate();
    label.repaint();
  }
});
```

Outline

I. Introduction
II. Full-Transparency
III. Semi-Transparency
IV. Conclusion

Conclusion

- Focused on mechanism that *hide the complexity* of AMI
  - Our study is independent of the underlying abstraction used to provide the concurrency
- Full-transparency
  - can be provided by
    - inheritance with limitations
    - composition with high constraints
  - is very complex when dealing with exceptions
  - is useless as far as efficiency is concerned

Conclusion

- We propose semi-transparency
  - Hides most complexity of AMI (abstractions)
  - Enforces the developper to focus on concurrency
  - ***Helps the making of highly concurrent applications***
- Dealing with exceptions
  - We already provide some solutions
  - Must be further studied
- This study is part of the Mandala framework

  http://mandala.sf.net

Outline

I. Introduction
II. Full-Transparency
III. Semi-Transparency
IV. Conclusion

Thank You.