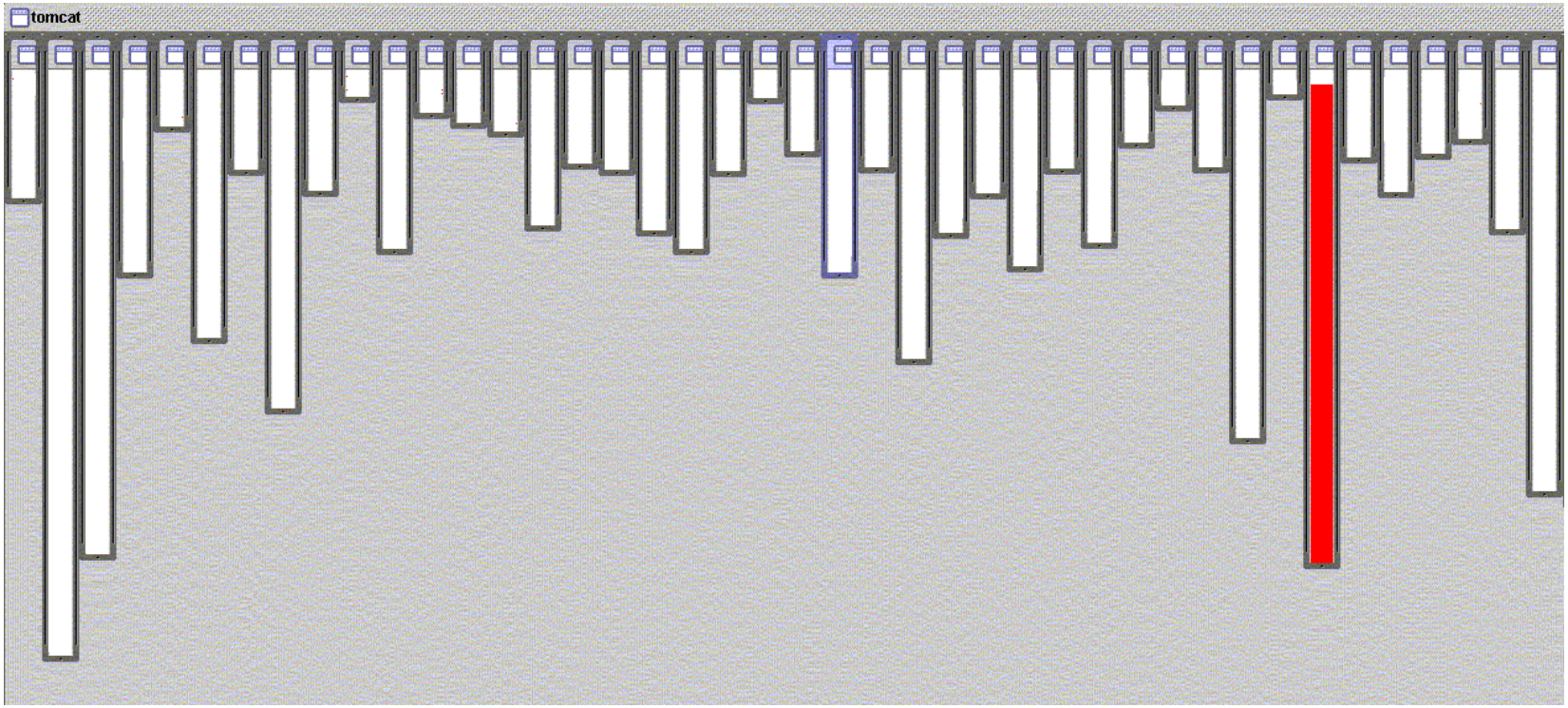
Adaptive distributed systems with AOP

Vania Marangozova-Martin

http://acs.forge.imag.fr

AOP Motivation
Good Modularity

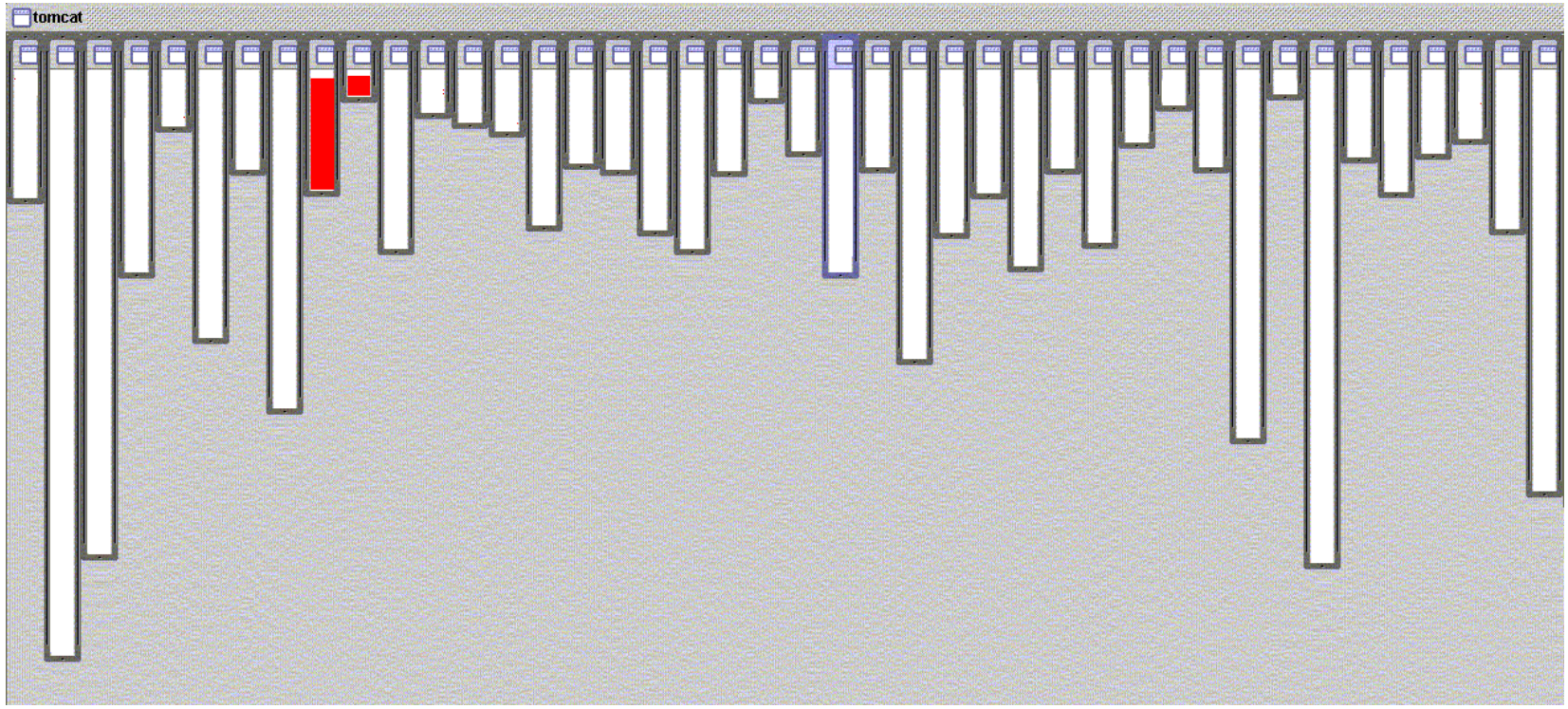
XML Parsing



- Parsing XML in org.apache.tomcat
  - red shows the code we are interested in
  - all the code is in one module (class) ☺

AOP Motivation
Good Modularity

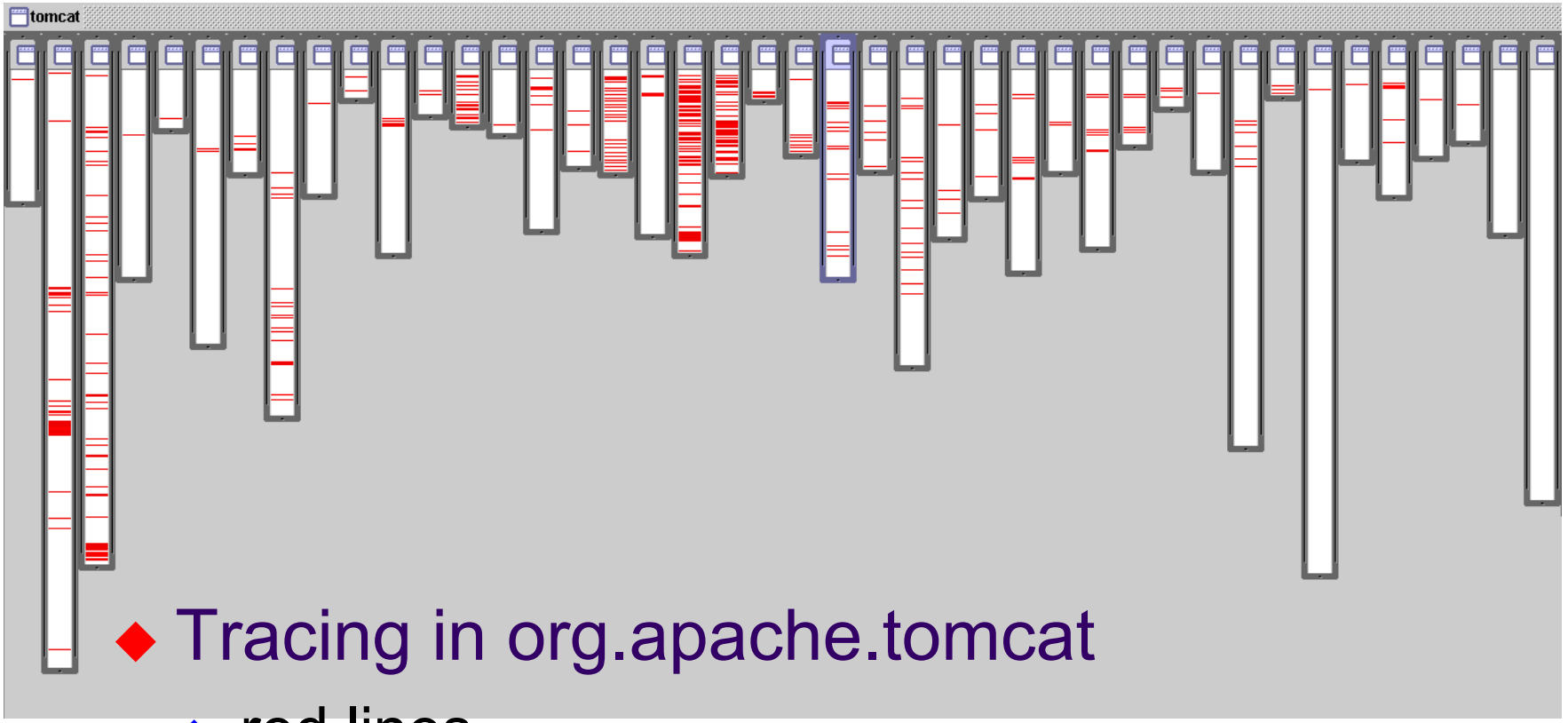
URL Pattern Recognition



- URL pattern recognition in org.apache.tomcat
  - the code we are interested in is in two classes ☺

AOP Motivation Problems

Tracing: Non-Modular



- Tracing in org.apache.tomcat
  - red lines
  - are not in one module
  - are almost everywhere ☹

Conslusion about the code

- Localized processing
  - Configuration files, XML parsing
- Processing implemented with few classes
  - HTTP requests, regular expressions
- Processing scattered everywhere
  - tracing, session management, …

What are the problems with non-modular code?

- Redundancy
  - Same processing repeated throughout the application code
- Obscure code
  - Code structure is not explicit
  - Global code architecture difficult to understand
- Difficult to change/maintain code
  - Need to find fragments that concern the feature we want to change
  - Need to make sure that changes are consistent i.e that a change in one place does not “hurt” a change in another place
  - Very few tools to help...

AOP Principles

- Separate tangled code
- Minimize inter-code dependancies
- All processings should have
  - a clear objective
  - a well-defined modular structure
- Aspects
  - Modular processings that can be “mixed”/ composed/tangled/organized together

Outline

- Introduction to AOP
- Introduction to AspectJ
- Syntax of AspectJ
- Software Development with AspectJ
- Conclusion and References

AOP Definition

- AOP = Aspect-Oriented Programming

- "Aspect-oriented programming allows the factorization of processings whose implementation is scattered through a system”

Types of Aspects

- A software system comprises:
  - Application aspects that correspond to core concerns. These are the software modules that implement the basic functionality of a system
  - Orthogonal aspects: cross-cutting concerns. Software modules that implement system code to be used by several other application aspects

AOP Methodology

- How do we do AOP?
  - Decompose a system into aspects
  - Implement the aspects
  - Recompose the aspects to form the overall application

AOP Methodology :
1. Decompose in aspects

- Separate the software processings into :
  - Application aspects.
    Example of a banking system : clients, accounts, transfers, etc.
  - Management aspects (NFR).
    Examples : data persistancy, authentification, tracing, ressource sharing, performance, etc.

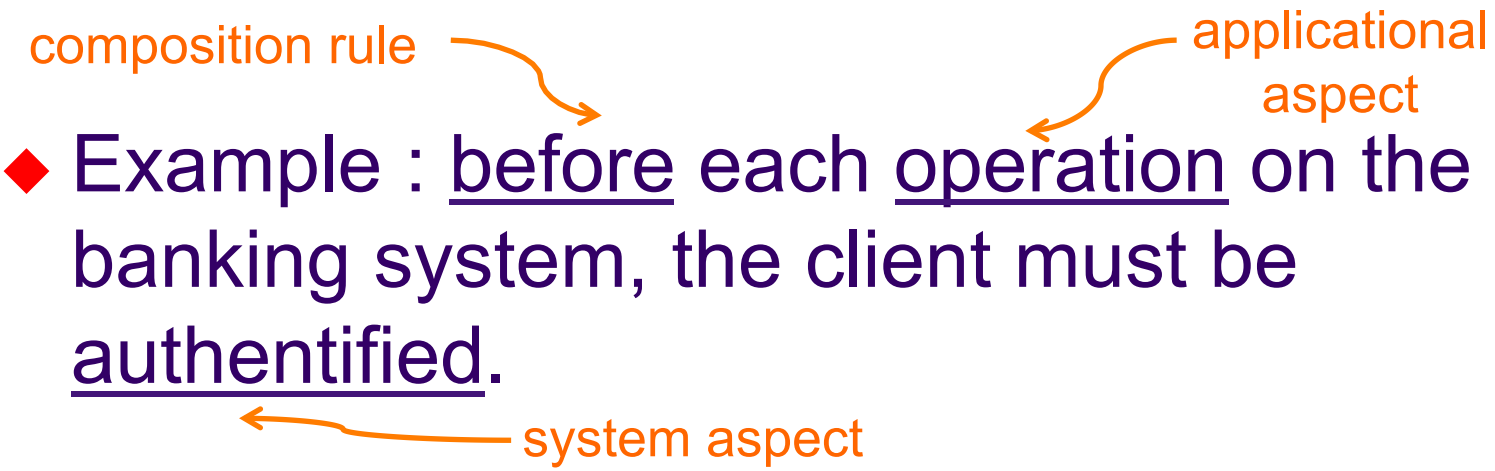
AOP Methodology :
2. Aspect Implementation

- Implementation of different aspects in an independent way i.e each aspect alone

- Implementation techniques
  - procedural languages (e.g. C)
  - object-oriented languages (e.g. Java)

AOP Methodology :
3. Aspects Composition (Weaving)

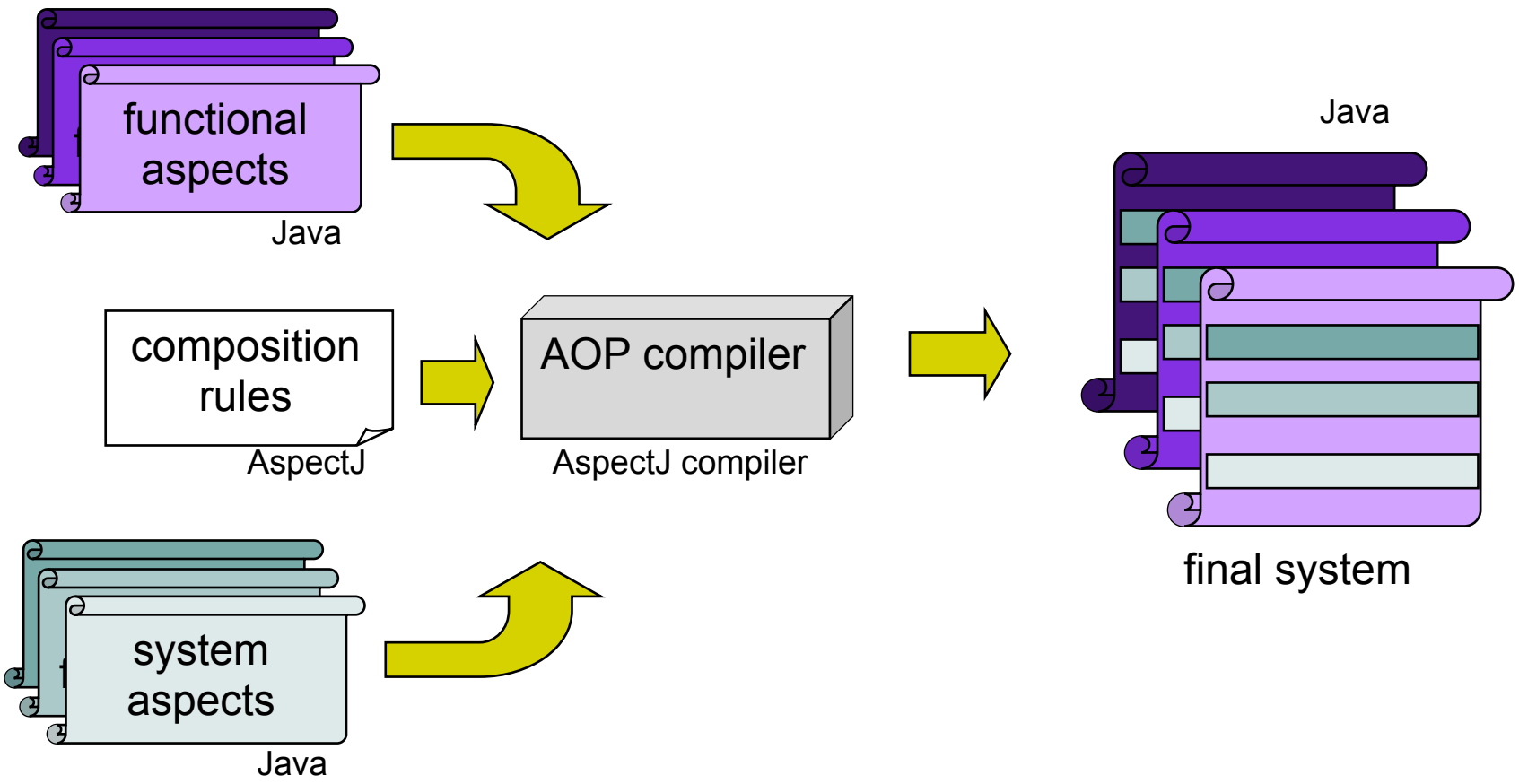
- Specify rules for composing aspects
- These rules are used in order to produce the final system
- Composition process: integration or weaving



How to AOP?

- AOP is a methodology, how do we put it into practice?
  - Aspect languages : C, C++, Java, etc.
  - Languages for specifying the rules for aspect weaving : AspectC, AspectJ, annotations Spring AOP…
  - Aspect integration tool : aspect weaver, AOP compiler (AspectC, AspectJ, etc.), run-time

AOP in Java


functional aspects
Java
composition rules
AspectJ
AOP compiler
AspectJ compiler
system aspects
Java
Java
final system

Outline

- Introduction to AOP
- **Introduction to AspectJ**
  - AspectJ Composition
  - Join point, pointcut, advice, introduction, declaration, aspect
  - Programming
  - Examples
- Syntax of AspectJ
- Development with AspectJ
- Conclusion and Refs

Composition in AspectJ

- Static composition
  - We can change the structure of the system: classes and interfaces
  - We can add methods and attributes
  - We can declare warnings or compile-time errors
- Dynamic composition
  - We can add new behavior to the normal behavior of the program
  - We can extend or replace an operation
  - We can operate before or after the execution of some operations

Composition elements in AspectJ

- AspectJ
  - is an extension of Java
  - for defining rules for dynamic and static composition of aspects

- Composition in AspectJ
  - Join point
  - Pointcut
  - Advice
  - Introduction
  - Declaration (at compile-time)
  - Aspect

Join point

- Definition
  - An well-identified point in the execution of a program
  - Can be a method call or the access of an attribute

```
public class Account {
    float balance;

    void credit(float amount) {
        this.balance += amount;
    }
}
```

- Example
  - The Account join points include
    - the execution of the credit method
    - the access to the balance attribute

Pointcut

- Definition
  - A pointcut is used to select a given join point and access the execution context of that join point
- Example

execution (void Account.credit(float))

  - Pointcut
    - select the join point corresponding to the call of the credit method of the Account class
    - get the execution context for this method: the object on which the method is called, the method's parameters

Advice

- Definition
  - The code associated to a pointcut i.e the code to be executed when the pointcut happens during the execution of a program
  - Execution before, after or in replacement of a pointcut
- Example
  - Advice
    - printing a message
    - before the execution of the credit method of the Account class

```
before() : execution (void Account.credit(float)) {
   System.out.println("About to perform credit operation");
}
```

Introduction

- Definition
  - Static change of a class or an interface
  - Add a method or an attribute, extend inheritance relations

```
declare parents : Account implements BankingEntity;
```

```
private float Account._minimalBalance;
```

- Example
  - Introduction
    - extension of the class hierarchy
    - from now on (i.e. after weaving) Account will implement the BankingAccount interface
    - add a new attribute _minimalBalance to Account

Compile-time declaration

- Definition
  - Add instructions to be composed statically
  - Add warnings or errors messages at compilation
- Example
  - Declaration
    - declare a warning
    - print a message whenever the save method of the Persistence class is called

```
declare warning : call (void
Persistence.save(Object)) :
    "Deprecated method Persistence.save(),
     Consider using Persistence.saveOptimized()";
```

Aspect

- Definition
  - A code module containing the rules for static and dynamic composition
  - Introductions + declarations + pointcuts + advices = aspect
  - An aspect in AspectJ is equivalent to a class in Java
- Example

```
public aspect ExampleAspect {
    declare parents : Account implements BankingEntity;

    before() : execution (void Account.credit(float)) {
        System.out.println("About to perform credit operation");
    }

    declare warning : call (void Persistence.save(Object)) :
        "Deprecated method Persistence.save(),
         Consider using Persistence.saveOptimized()";
}
```

Programming Methodology

- Design
  - Identify the join points where the program behavior should be changed/extended
  - Design the new behavior to be inserted at these join points

- Implementation
  - Declaration of an aspect:
    containing the implementation of the above design
  - Declaration of join points
  - Declaration of the advice:
    to be associated to the above join pojnts i.e define the new behavior to execute at these join points
  - Add rules for static composition if necessary (introductions, declarations)

Example Hello (1 / 4)

- Application

```
public class Communicator {
    public static void print(String message) {
        System.out.println(message);
    }
    public static void print(String person, String message) {
        System.out.println(person + ", " + message);
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Communicator.print("Want to learn AspectJ?");
        Communicator.print("Tom", "how are you?");
    }
}
```

Example Hello (2 / 4)

- Compilation Java

  - javac Communicator.java Test.java

- Execution

  - java Test

  Want to learn AspectJ?

  Tom, how are you?

Example Hello (3 / 4)

- Aspect

```
public aspect HelloAspect {          ❶ Declaration of an aspect

                                     ❷ Declaration of a pointcut
    pointcut printCall() : call (void Communicator.print(..));


    before() : printCall() {                ❸ Declaration of an advice
        System.out.print("Hello! ");
    }

}
```

Example Hello (4 / 4)

- Compilation AspectJ

  - ajc Communicator.java Test.java HelloAspect.java

- Execution

  - java Test

  Hello! Want to learn AspectJ?

  Hello! Tom, how are you?

Tracing Example (1/4)

- Initial application

```
package banking;
public class Account {
    float balance;
    void credit(float amount) {

        this.balance += amount;

    }
    void debit(float amount) {

        this.balance -= amount;

    }
}
```

Tracing Example (2 / 4)

- Hand-coded modifications

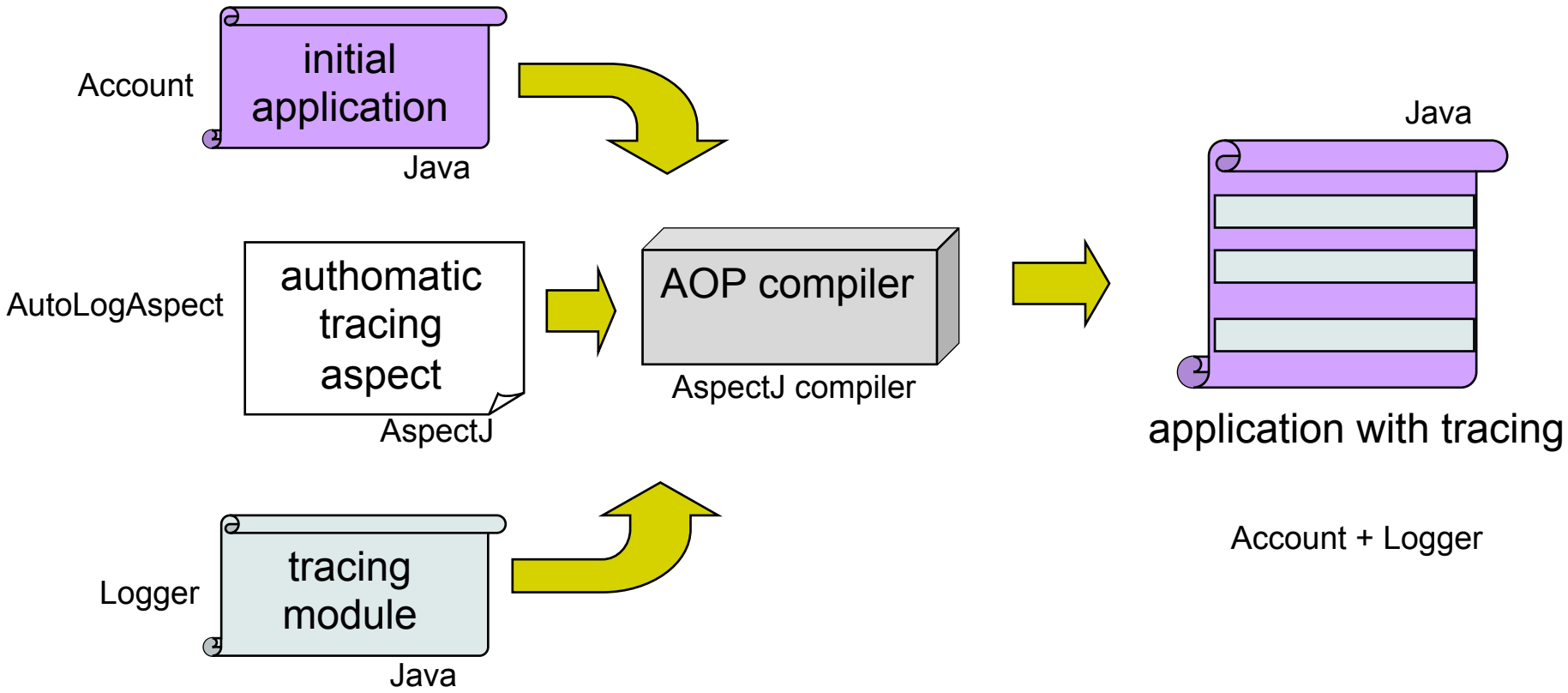
```
package banking;

public class Account {
    float balance;

    void credit(float amount) {
        Logger.entering("banking.Account", "credit(float)");
        this.balance += amount;
        Logger.exiting("banking.Account", "credit(float)");
    }

    void debit(float amount) {
        Logger.entering("banking.Account", "debit(float)");
        this.balance -= amount;
        Logger.exiting("banking.Account", "debit(float)");
    }
}
```

Tracing Example (3 / 4)

- Aspect-based automatic tracing

```
public aspect AutoLogAspect {

    pointcut methodExecution() : execution (*
banking.Account.*(..));

    before() : methodExecution() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        String className = sig.getDeclaringType().getName();
        String methodName = sig.getName();
        Logger.entering(className, methodName);
    }

    after() : methodExecution() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        String className = sig.getDeclaringType().getName();
        String methodName = sig.getName();
        Logger.exiting(className, methodName);
    } }
```

Tracing Example (4 / 4)



Outline

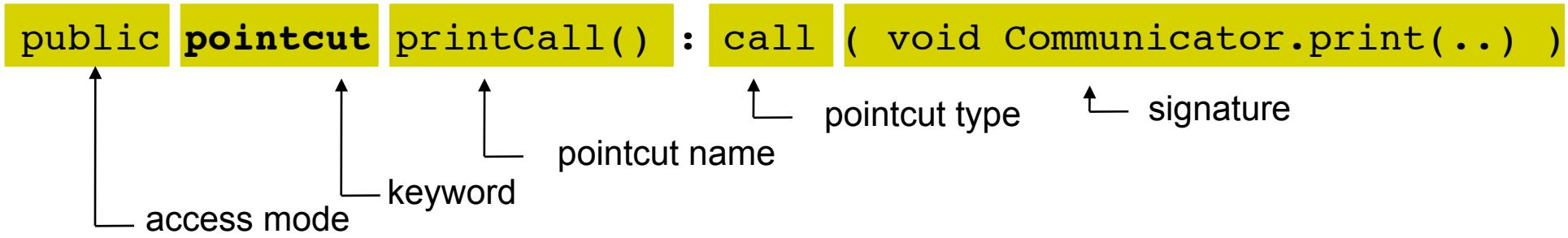
- Introduction to AOP
- Introduction to AspectJ
- **Syntaxe of AspectJ**
  - Pointcut
  - Call vs. Execution
  - Pointcuts of control flow, lexical structure
  - Advice
  - Passing parameters between pointcut and advice
- Software Development with AspectJ
- Conclusion and Refs

Pointcut

- pointcut syntax

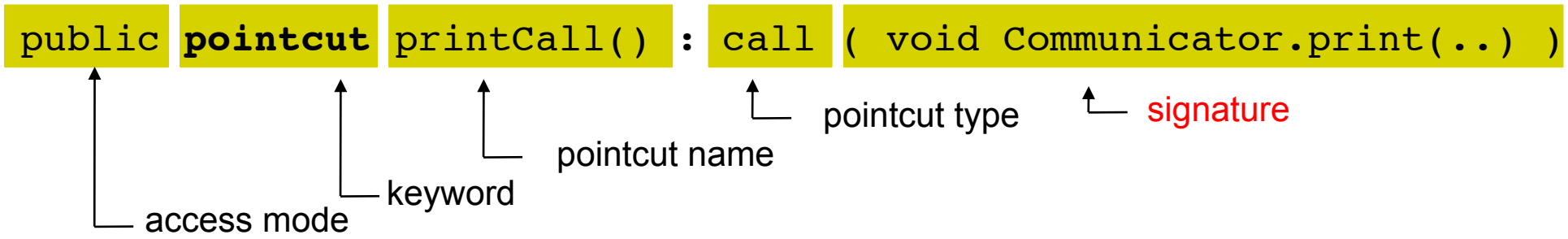
```
[access_mode] pointcut name_pointcut([args]) :
definition_pointcut
```

- Example



Signature of a pointcut

- Example



- Signature of a pointcut
  - Signature of a type
  - Signature of methods/constructors
  - Signature of an attribute

Pointcut: Type Signature

Signature	Associated types
Account	Type (classe / interface) called Account
*Account	Type whose name ends with Account, e.g SavingsAccount, CheckingAccount
java.*.Date	Type Date in a direct sub-package of the package java, e.g java.util.Date, java.sql.Date
java..Date	All types Date in the java package or in a (direct or indirect) sub-package of the java package
java.util.List+	Type inheriting the java.util.List interface

Pointcut: Method Signature

Signature	Associated methods
public void Collection.clear()	public method clear() of the Collection class, method with no parameters returning void
public void Account.set*(*)	Any public method of the Account class, whose name starts with set and returning void, whatever the parameters are
* Account.*()	Any method of the class Account, no parameters, whatever the returning type and access mode are
public void Account.*(..)	All public methods of the class Account, returning void, whatever the name and the parameters are
* *.*(..) throws RemoteException	All methods raising exception of type RemoteException

Pointcut: Constructor Signature

Signature	Associated constructors
public Account.new()	public constructor of the class Account, no parameters
public Account.new(int)	public constructor of the class Account, one int parameter
public Account.new(..)	All public constructors of the class Account, whatever the parameters are
public Account +.new(..)	All public constructors of the Account class or of one if its subclasses

Pointcut: Attribute Signature

Signature	Associated attributes
private float Account.balance	The private attribute balance of the class Account
* Account.*	All Account attributes, whatever their type, name and access mode are
public static * banking..*.*	All public static attributes of a class in the banking package or in one of its sub-packages
* *.*	All attributes of all classes

Pointcut Operators

- Unary Operator : !

public !final *.*	All public and non final attributes
!Vector	All types different from Vector

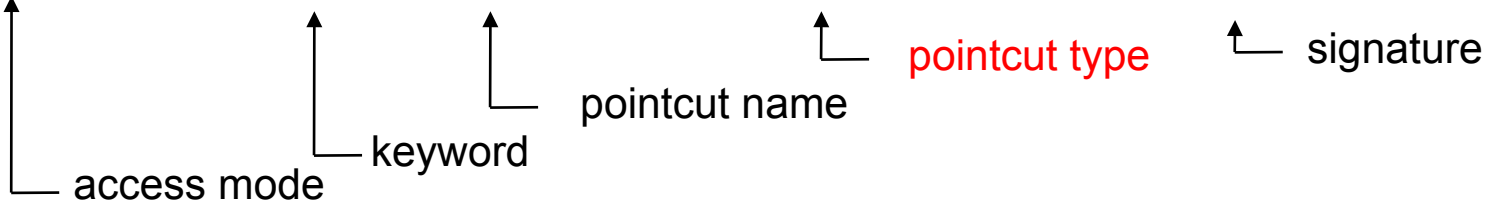
- Binary Operators : && (and) || (or)

Vector || Hashtable	Type Vector or Hashtable
java.util.RandomAccess+ && java.util.List	All type implementing both interfaces

Pointcut Type

- Example

```
public pointcut printCall() : call ( void Communicator.print(..) )
```



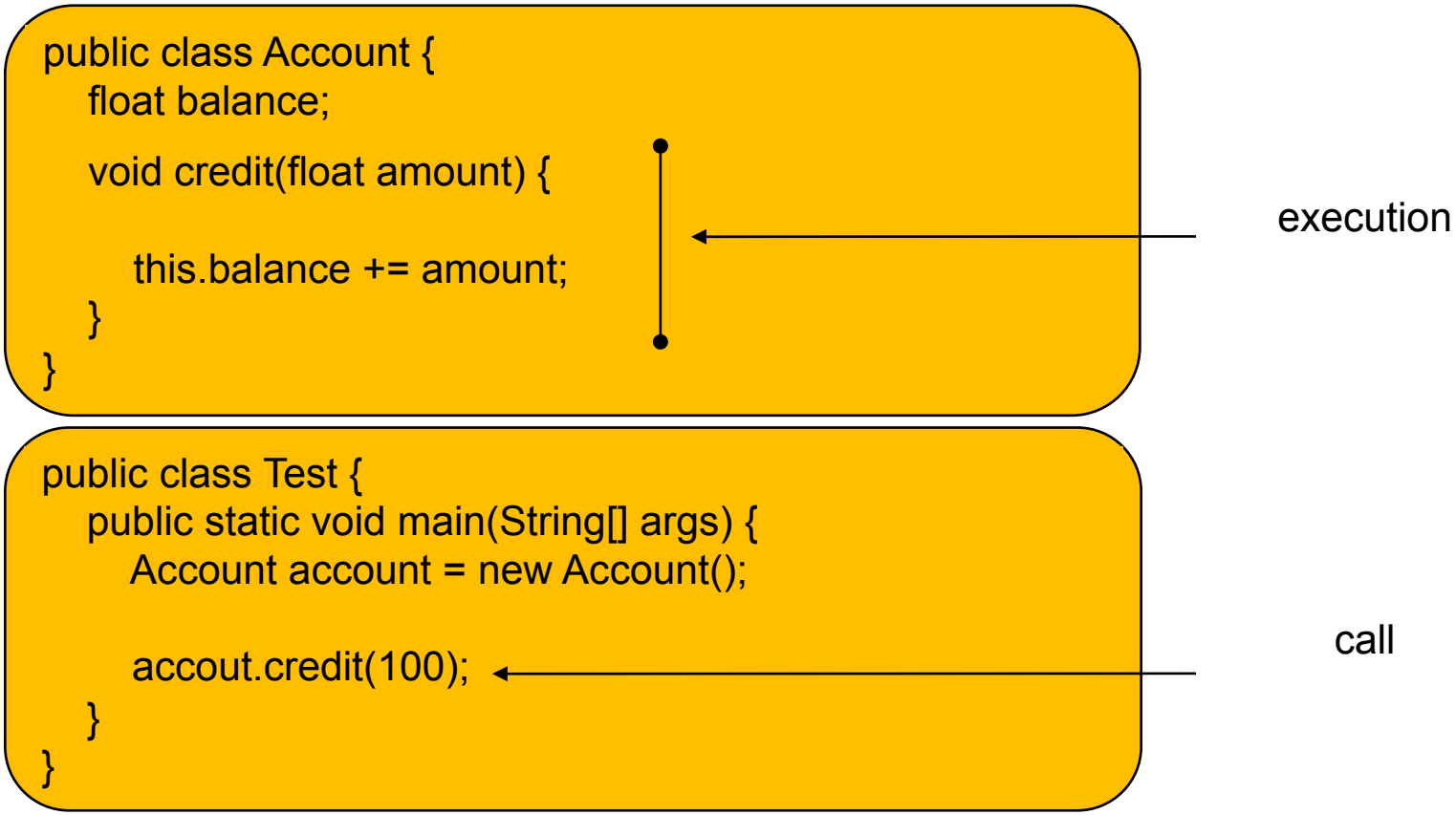
- Type of a pointcut
  - Call of a method, constructor
  - Execution of a method, constructor
  - Access to an attribute in read, write mode

Types of Pointcuts

Pointcut Type	Syntax
method call	call(signature_method)
method execution	execution(signature_method)
constructor call	call(signature_constructor)
constructor execution	execution(signature_constructor)
class initialization	staticinitialization(signature_type)
read access to an attribute	get(signature_attribute)
write access to an attribute	set(signature_attribute)
exception handler	handler(signature_type)

Call vs. Execution (1 / 5)

- Example



Call vs. Execution (2 / 5)

- Aspect using the call

```
public aspect AutoLogAspect_Call {
    pointcut creditMethodCall() : call (* Account.credit(..));

    before() : creditMethodCall() {
        …
        Logger.entering(className, methodName);
    }
}
```

Call vs. Execution (3 / 5)

- Application of AutoLogAspect_Call

```
public class Account {
    float balance;

    void credit(float amount) {

        this.balance += amount;
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Account account = new Account();
        Logger.entering("Account", "credit(float)");
        accout.credit(100);
    }
}
```

Call vs. Execution (4 / 5)

- Aspect using execution

```
public aspect AutoLogAspect_Execution {
    pointcut creditMethodExecution() : execution (* Account.credit(..));

    before() : creditMethodExecution() {
        …
        Logger.entering(className, methodName);
    }
}
```

Call vs. Execution (5 / 5)

- Application of AutoLogAspect_Execution

```
public class Account {
    float balance;

    void credit(float amount) {
        Logger.entering("Account", "credit(float)");
        this.balance += amount;
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Account account = new Account();

        accout.credit(100);
    }
}
```


Control Flow Pointcuts

Pointcut	Description
cflow(call (* Account.credit(..)))	All join points in the control flow of the method credit of the Account class, including the call of the method itself
cflowbelow(call (* Account.credit(..)))	All join points in the control flow of the method credit of the Account class, except the call of the method
cflow(creditMethodCall())	All join points in the control flow of the pointcut creditMethodCall
cflowbelow(execution (* Account.new(..)))	All join points in the control flow of a constructor of the class Account, except the execution of the constructor itself

cflow / cflowbelow

- Example

```
public class Test {
    public static void main(String[] args) {
        Account account = new Account();

        accout.credit(100);
    }
}
```

cflow / cflowbelow

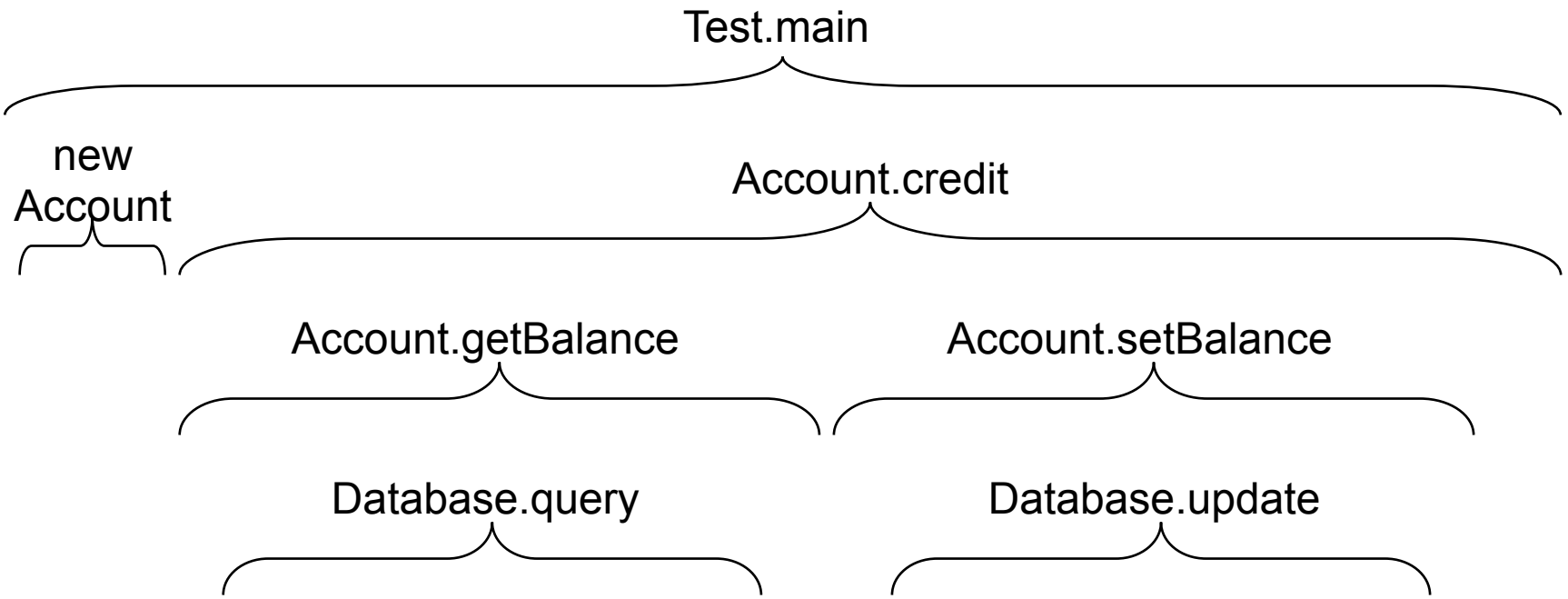
```
public class Account {
    int             id;             // account id
    Database database;  // associated database

    float getBalance() {

        return database.query("SELECT balance FROM accounts WHERE id=" + id);

    }

    void setBalance(float b) {

        database.update("UPDATE accounts SET balance="+b+" WHERE id=" + id);

    }

    void credit(float amount) {

        setBalance(getBalance() + amount);
    }
}
```

cflow / cflowbelow

```
public class Database {
    float query(String sqlQuery) {
        … reading data from the database
    }
    void update(String sqlQuery) {
        … writing data to the database
    }
}
```

cflow / cflowbelow

- Control flow



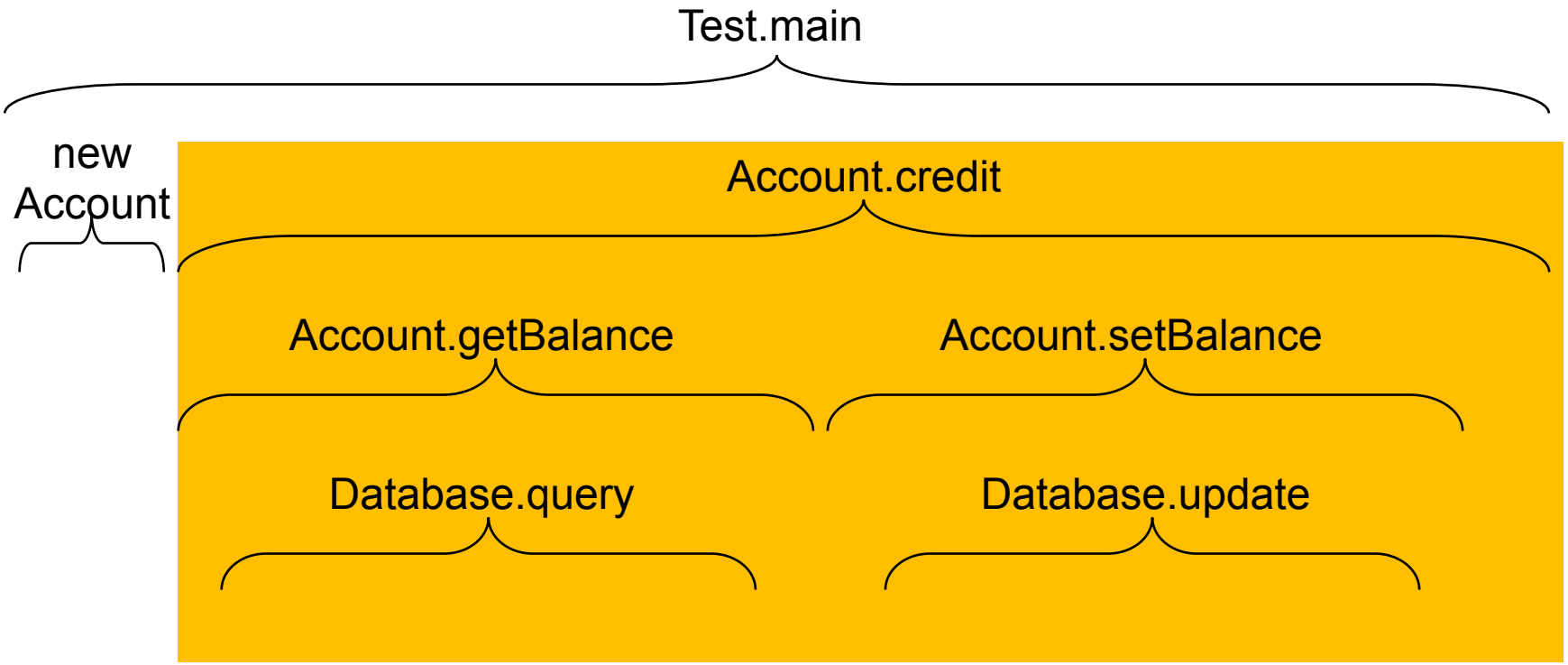
cflow

- Aspect using cflow

```
public aspect AutoLogAspect_CallF {
    pointcut creditMethodCallF() : cflow( call (* Account.credit(..)) );

    before() : creditMethodCallF() {
        System.out.println("Hello!");
    }
}
```

cflow

- cflow(call(* Account.credit(..)))



cflow

- Example

```
public class Test {
    public static void main(String[] args) {
        Account account = new Account();
        System.out.println("Hello!");
        accout.credit(100);
    }
}
```

cflow

```
public class Account {
    int             id;             // account id
    Database database;  // associated database

    float getBalance() {
        System.out.println("Hello!");
        return database.query("SELECT balance FROM accounts WHERE id=" + id);

    }

    void setBalance(float b) {
        System.out.println("Hello!");
        database.update("UPDATE accounts SET balance="+b+" WHERE id=" + id);

    }

    void credit(float amount) {
        System.out.println("Hello!");
        setBalance(getBalance() + amount);
    }
}
```

An error here; what is it?

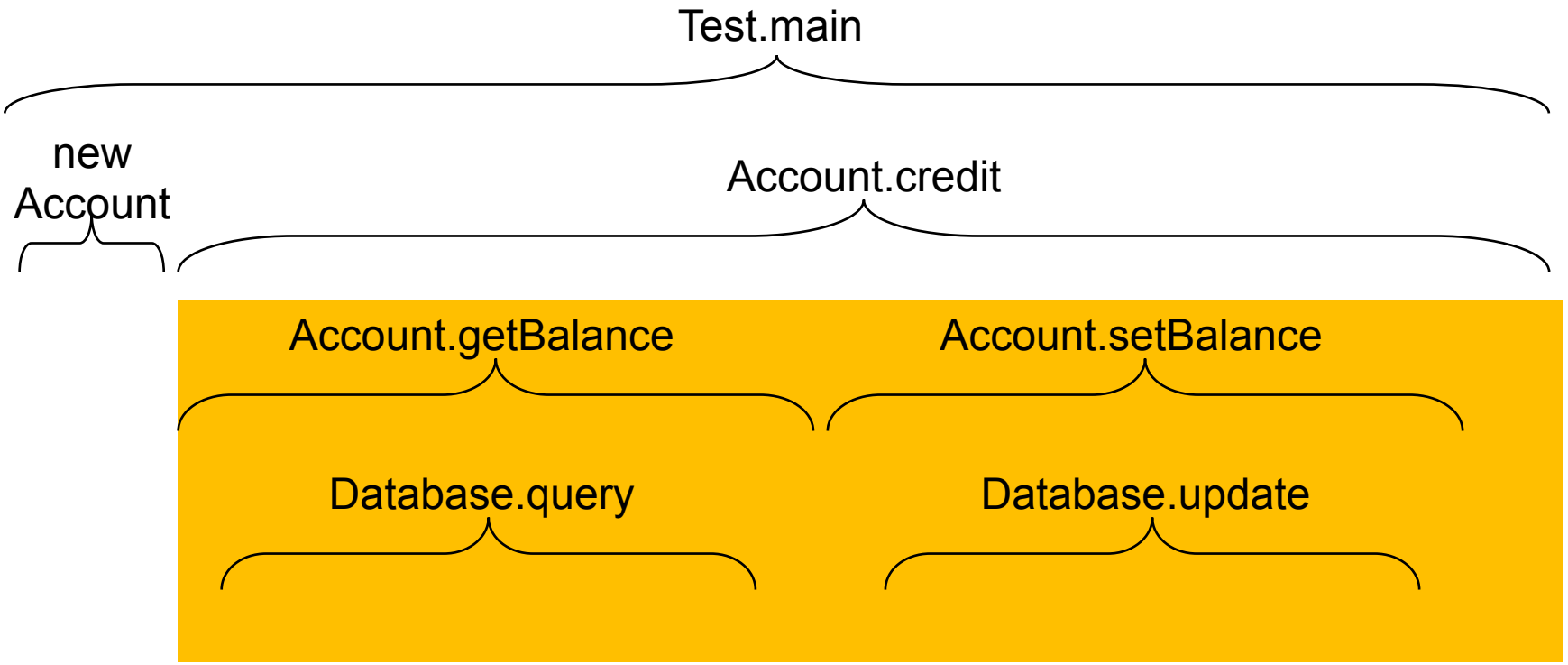
cflowbelow

- Aspect using cflowbelow

```
public aspect AutoLogAspect_CallFB {
    pointcut creditMethodCallFB() : cflowbelow( call (* Account.credit(..)) );

    before() : creditMethodCallFB() {
        System.out.println("Hello!");
    }
}
```

cflowbelow

- cflowbelow(call(* Account.credit(..)))



cflowbelow

- Example

```
public class Test {
    public static void main(String[] args) {
        Account account = new Account();
        System.out.println("Hello!");
        accout.credit(100);
    }
}
```

cflowbelow

```
public class Account {
    int            id;            // account id
    Database database;  // associated database

    float getBalance() {
        System.out.println("Hello!");
        return database.query("SELECT balance FROM accounts WHERE id=" + id);

    }

    void setBalance(float b) {
        System.out.println("Hello!");
        database.update("UPDATE accounts SET balance="+b+" WHERE id=" + id);

    }

    void credit(float amount) {
        System.out.println("Hello!");
        setBalance(getBalance() + amount);
    }
}
```

Pointcuts related to lexical structure

Pointcut	Description
within(Account)	All join points in the class Account
within(Account+)	All join point in the class Account and in its subclasses
withincode(* Account.credit(..))	All join points in the method credit of the class Account

Example with within

- Aspect with execution

```
public aspect AutoLogAspect_Execution {
    pointcut methodExecution() : execution (* Account.*(..))
                                 && within(banking..*);

    before() : methodExecution() {
        …
        Logger.entering(className, methodName);
    }
}
```

Pointcuts related to execution object

Pointcut	Description
this(Account)	Every join point where this instance is of the Account class or of one of its subclasses. This pointcut is used in combination with execution
target(Account)	Every join point where the object on which the method is called is instance of Account or of one of its subclasses. Pointcut used in combination with call

- this(Type) or this(ObjectIdentifier)
- target(Type) or target(ObjectIdentifier)

Pointcuts arguments

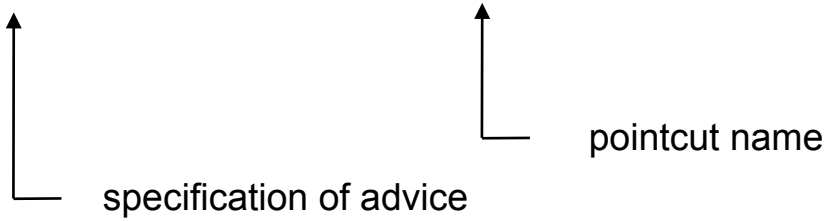
Pointcut	Description
args(String,..,int)	All join points of all the methods where the first argument is of type String and the last argument is of type int.

- args(Type or ObjectIdentifier,..)

Advice

- Advice :
  - Code to execute at a join point
- Syntax of an advice

```
specification_advice([args]) : name_pointcut([args})
{
        … advice body
}
```



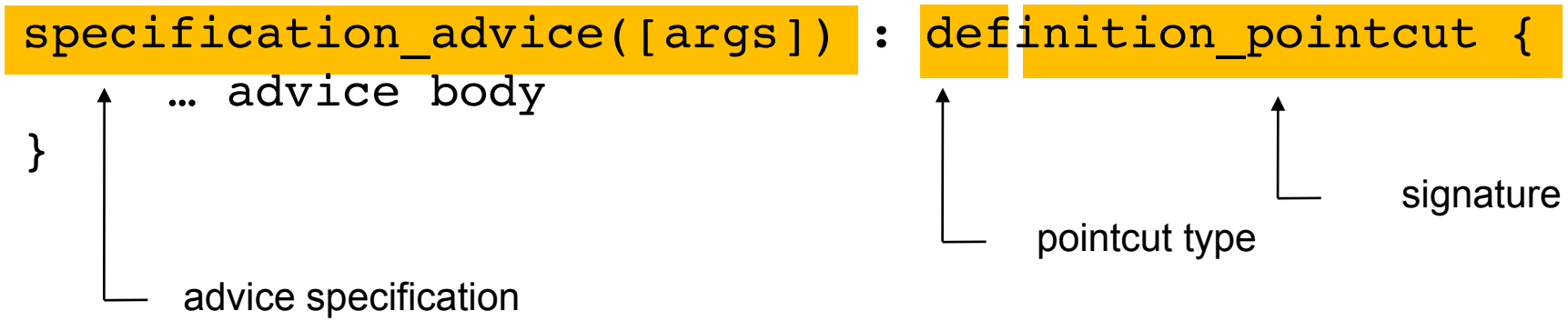
- Example

```
before () : printCall () {

        … advice body
}
```

Advice : Syntax

- Syntaxe of an advice

```
specification_advice([args]) : definition_pointcut {
    … advice body
}
```



- Example

```
before () : call ( void Communicator.print(..) ) {

    … advice body
}
```

Types of advice

- Advice before
  - executes before join point
- Advice after
  - executes after join point
- Advice around
  - surrounds execution of join point

Advice before

- Example

```
before () : call ( * Account.*(..)) {

    … authentify the user
}
```

- Particlular case
  - If the advice raises an exception, the method is not execute
- Examples
  - Tracing, authentification, etc.

Advice after

- Termination (normal or exception)

```
after () : call ( * Account.*(..)) {
    … trace all terminations
}
```

- Normal termination

```
after() returning () : call ( * Account.*(..)) {
    …
}
```

- Termination with an exception

```
after() throwing () : call ( * Account.*(..)) {
    … trace the exception
}
```

Advice around

- Replace a processing

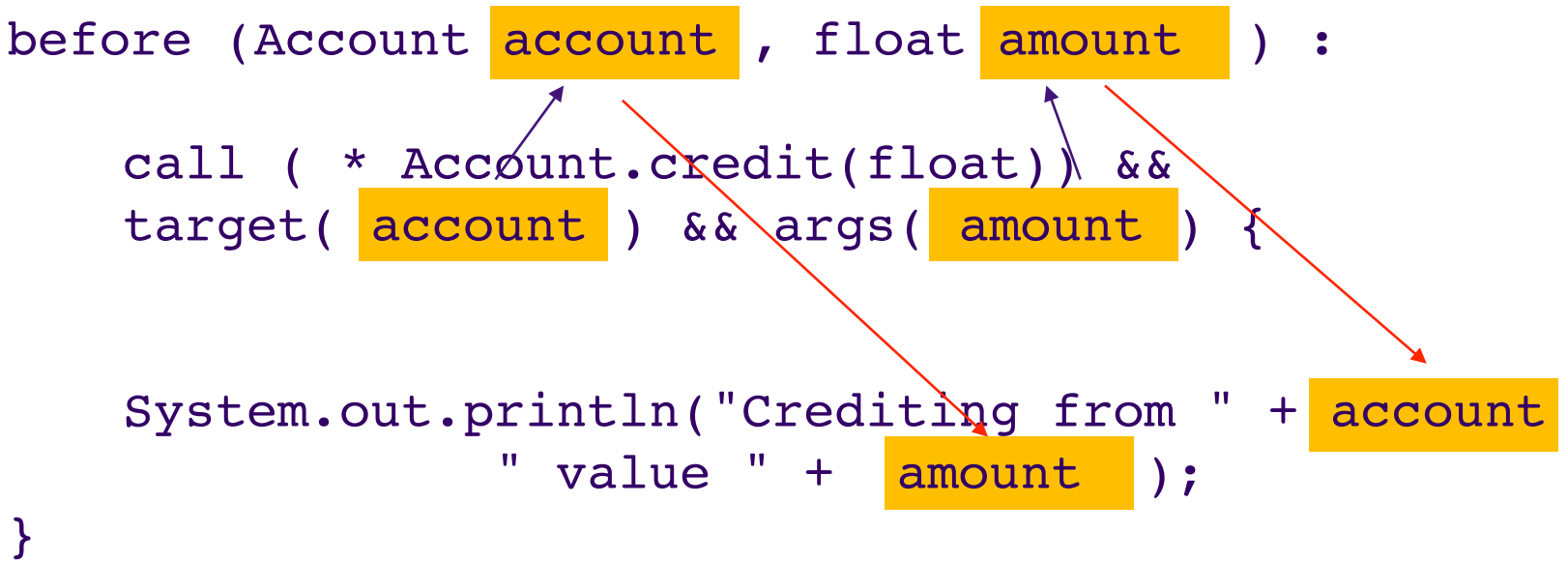
```
around () : call ( void Account.credit(float)) {

    … new implementation of the method
}
```

- Surround a processing

```
around (Account acount, float amount) :
  call ( * Account.credit(float)) && target(account)
&& args(amount) {

    … trace the beginning of the method execution
    proceed(account, amount); // method execution
    … trace the end of the method execution
}
```

Context passing between join point and advice (1 / 3)

- Parameter passing



Context passing between join point and advice (2 / 3)

- Get the result

```
after returning (Object object ) :

  call ( Object Account.*(..)) {

    System.out.println("Result " +  object );
}
```

Context passing between join point and advice (3 / 3)

- Get the exception

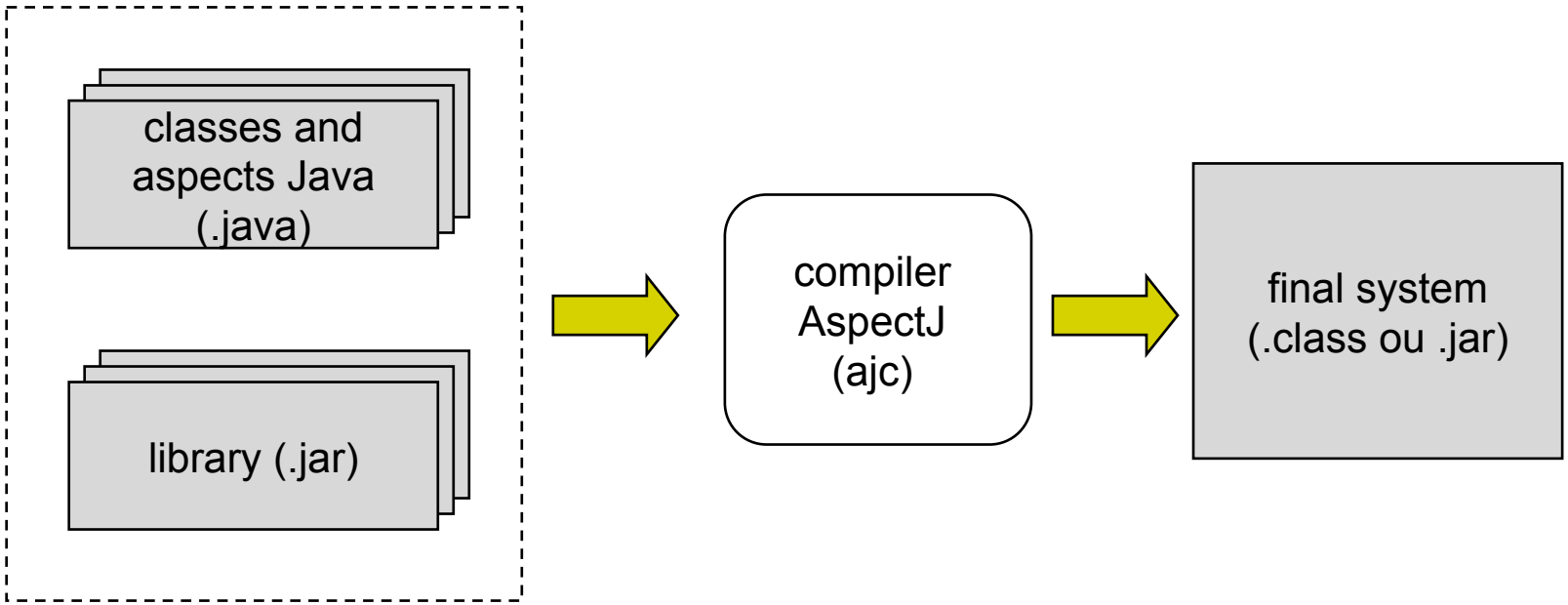
```
after throwing (RemoteException except ) :


call ( * Account.*(..) throws RemoteException ) {

    System.out.println("Exception " + except );
}
```

Outline

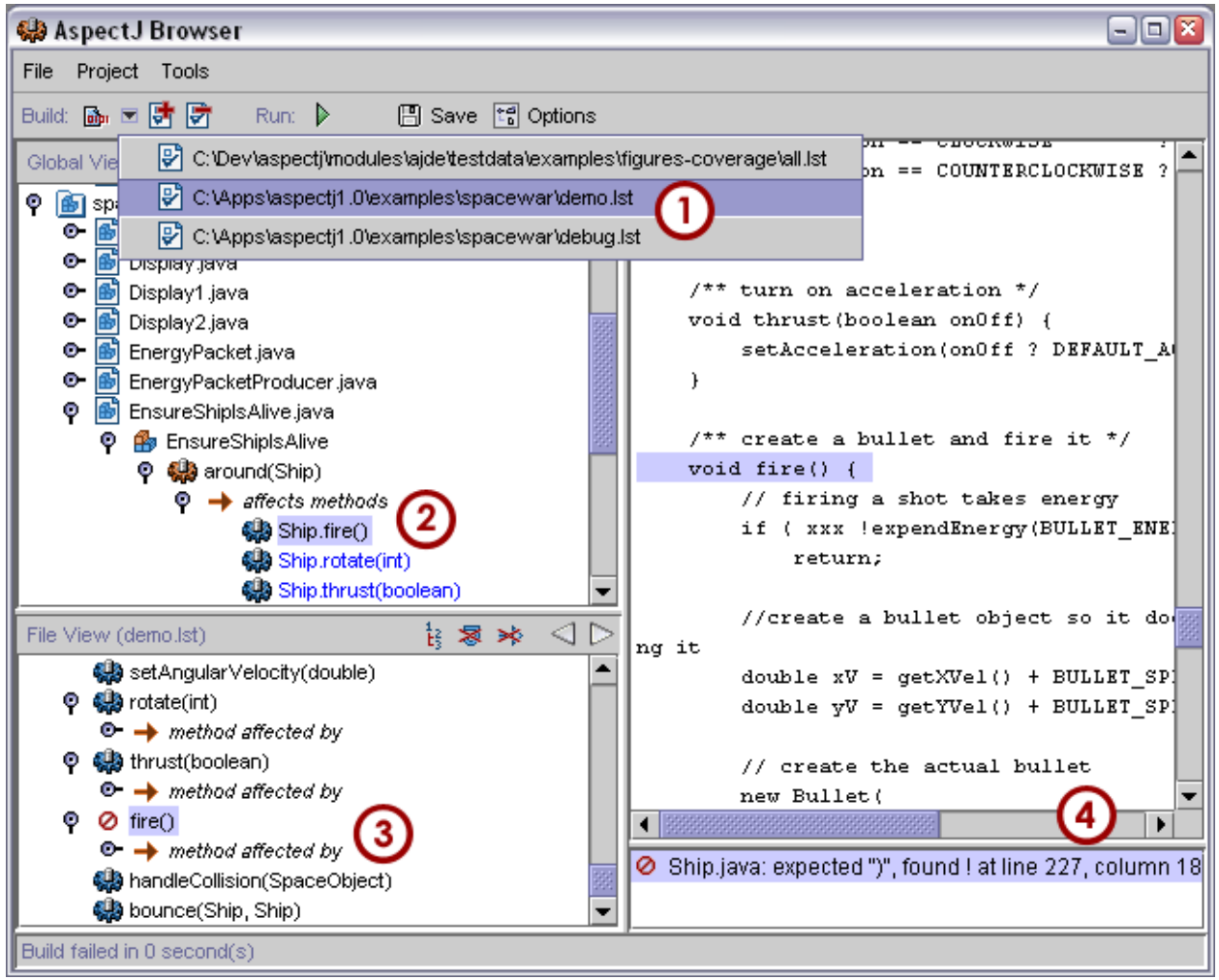
- Introduction to AOP
- Introduction to AspectJ
- Syntax of AspectJ
- Software Development with AspectJ
- Conclusion and Refs

AspectJ Compiler

- ajc command



Browser AspectJ (ajbrowser)



AOP in few words

- AOP: what for?
  - Extend an existing system so as to integrate new behaviors
  - Modular design of a system composed of multiple aspects (application and management)

Web

- AOP / AOSD
  http://www.aosd.net
- AspectJ
  http://www.eclipse.org/aspectj/
- AOP in C, C++
  http://www.aspectc.org/

- White Paper
- Spring

Big Players

```
pointcut anyCall() : call(* *.*(..));
```

- For Java
  - AspectJ
  - AspectWerkz
  - JBoss AOP
  - Spring AOP
- For non-Java
  - AspectC++
  - Others, including comparison, can be found here:

```
@Pointcut("call(* *.*(..))")
void anyCall() {}
```

@AspectJ

Spring 2.0 AOP includes AspectJ

Bibliography

- Aspect-Oriented Programming, Kiczales et. al., ECOOP'1997
- An Overview of AspectJ, Kiczales et. al., ECOOP'2001
- Getting Started with AspectJ, Kiczales et. al., CACM 44(10), Oct. 2001
- Aspect-Oriented Programming with AspectJ, Ivan Kiselev, SAMS 2002
- Mastering AspectJ: Aspect-Oriented Programming in Java, Joseph D. Gradecki, Nicholas Lesiecki, John Wiley & Sons 2003
- AspectJ in Action, R. Laddad, Manning, 2003.