# In the beginning…

- Class ATM

```
package banking;

public class ATM {
    private String _id;
    private String _address;

    private transient ATMCard _currentATMCard;
    private transient Account _currentAccount;

    public ATM(String id, String address) {
        _id = id;
        _address = address;
    }

    public String getID() {
        return _id;
    }

    public String getAddress() {
        return _address;
    }

    public void setCurrentATMCard(ATMCard currentATMCard) {
        _currentATMCard = currentATMCard;
    }

    public void setCurrentAccount(Account currentAccount) {
        _currentAccount = currentAccount;
    }

    public void credit(float amount) {
         _currentATMCard.getBankLiaison()
              .credit(_currentAccount, amount);
    }

    public void debit(float amount) throws InsufficientBalanceException {
        _currentATMCard.getBankLiaison()
             .debit(_currentAccount, amount);
    }

    public float getBalance() {
        return _currentATMCard.getBankLiaison()
             .getBalance(_currentAccount);
    }
}
```

- Class Account

```
package banking;

public class Account {
    private int _accountNumber;
    private float _balance;

    public Account(int accountNumber) {
        _accountNumber = accountNumber;
    }

    public int getAccountNumber() {
        return _accountNumber;
    }

    public void credit(float amount) {
        _balance = _balance + amount;
    }

    public void debit(float amount)
        throws InsufficientBalanceException {
        if (_balance < amount) {
            throw new InsufficientBalanceException(
                    "Total balance not sufficient");
        } else {
            _balance = _balance - amount;
        }
    }

    public float getBalance() {
        return _balance;
    }

    public String toString() {
        return "Account: " + _accountNumber;
    }
}
```

# Let's add logging…

- Class ATM

```
package banking;

import java.util.logging.*;

public class ATM {

    ...

     private static Logger _logger
         = Logger.getLogger("banking");

    ...

    public void credit(float amount) {
        _logger.logp(Level.INFO,
                     "ATM", "credit", "Entering");
        _currentATMCard.getBankLiaison()
              .credit(_currentAccount, amount);
    }

    public void debit(float amount)
        throws InsufficientBalanceException {
        _logger.logp(Level.INFO,
                     "ATM", "debit", "Entering");
         _currentATMCard.getBankLiaison()
              .debit(_currentAccount, amount);
    }

    ...

}
```

- Class Account

```
package banking;

import java.util.logging.*;

public class Account {

   ...

    private static Logger _logger
        = Logger.getLogger("banking");

    ...

    public void credit(float amount) {
        _logger.logp(Level.INFO,
                     "Account", "credit", "Entering");
        _balance = _balance + amount;
    }

    public void debit(float amount)
        throws InsufficientBalanceException {
        _logger.logp(Level.INFO,
                     "Account", "debit", "Entering");
        if (_balance < amount) {
            throw new InsufficientBalanceException(
                      "Total balance not sufficient");
        } else {
            _balance = _balance - amount;
        }
    }

    ...

}
```

# Let's add logging…

- Class ATM

```
package banking;

public class ATM {

    ...

    ...

    public void credit(float amount) {

        _currentATMCard.getBankLiaison()
              .credit(_currentAccount, amount);
    }

    public void debit(float amount)
        throws InsufficientBalanceException {

         _currentATMCard.getBankLiaison()
              .debit(_currentAccount, amount);
    }

    ...

}
```

- Class Account

```
package banking;

public class Account {

   ...

    ...

    public void credit(float amount) {

        _balance = _balance + amount;
    }

    public void debit(float amount)
        throws InsufficientBalanceException {

        if (_balance < amount) {
            throw new InsufficientBalanceException(
                      "Total balance not sufficient");
        } else {
            _balance = _balance - amount;
        }
    }

    ...

}
```

# …and access control

- Class ATM

```
package banking;

import java.security.*;
import javax.security.auth.*;

import java.util.logging.*;

public class ATM {
    ...

    private Subject _atmSubject;

    private static Logger _logger = Logger.getLogger("banking");

    ...

    public void credit(final float amount) {
        _logger.logp(Level.INFO, "ATM", "credit", "Entering");
        PrivilegedAction worker
            = new PrivilegedAction() {
                public Object run() {
                    _currentATMCard.getBankLiaison().credit(_currentAccount, amount);
                    return null;
                }};
        Subject.doAsPrivileged(_atmSubject, worker, null);
    }

    public void debit(final float amount) throws InsufficientBalanceException {
        _logger.logp(Level.INFO, "ATM", "debit", "Entering");

        PrivilegedExceptionAction worker
            = new PrivilegedExceptionAction() {
                public Object run() throws Exception {
                    _currentATMCard.getBankLiaison().debit(_currentAccount, amount);
                    return null;
                }};

        try {
            Subject.doAsPrivileged(_atmSubject, worker, null);
        } catch (PrivilegedActionException ex) {
            Throwable cause = ex.getCause();
            if (cause instanceof InsufficientBalanceException) {
                throw (InsufficientBalanceException)ex.getCause();
            }
        }
    }

    ...

}
```

- Class Account

```
package banking;

import java.util.logging.*;
import java.security.AccessController;

public class Account {
    ...

    private static Logger _logger = Logger.getLogger("banking");

    ...

    public void credit(float amount) {
        AccessController.checkPermission(
            new BankingPermission("accountOperation"));
        _logger.logp(Level.INFO,
                     "Account", "credit", "Entering");
        _balance = _balance + amount;
    }

    public void debit(float amount) throws InsufficientBalanceException {
        AccessController.checkPermission(
            new BankingPermission("accountOperation"));
        _logger.logp(Level.INFO,
                     "Account", "debit", "Entering");
        if (_balance < amount) {
            throw new InsufficientBalanceException(
                      "Total balance not sufficient");
        } else {
            _balance = _balance - amount;
        }
    }

    ...

}
```

# …and access control

- Class ATM

```
package banking;

import java.security.*;
import javax.security.auth.*;


public class ATM {
    ...

    private Subject _atmSubject;


    ...

    public void credit(final float amount) {

        PrivilegedAction worker
            = new PrivilegedAction() {
                public Object run() {
                    _currentATMCard.getBankLiaison().credit(_currentAccount, amount);
                    return null;
                }};
        Subject.doAsPrivileged(_atmSubject, worker, null);
    }

    public void debit(final float amount) throws InsufficientBalanceException {


        PrivilegedExceptionAction worker
            = new PrivilegedExceptionAction() {
                public Object run() throws Exception {
                    _currentATMCard.getBankLiaison().debit(_currentAccount, amount);
                    return null;
                }};

        try {
            Subject.doAsPrivileged(_atmSubject, worker, null);
        } catch (PrivilegedActionException ex) {
            Throwable cause = ex.getCause();
            if (cause instanceof InsufficientBalanceException) {
                throw (InsufficientBalanceException)ex.getCause();
            }
        }
    }

    ...

}
```

- Class Account

```
package banking;


import java.security.AccessController;

public class Account {
    ...


    ...

    public void credit(float amount) {
        AccessController.checkPermission(
            new BankingPermission("accountOperation"));

        _balance = _balance + amount;
    }

    public void debit(float amount) throws InsufficientBalanceException {
        AccessController.checkPermission(
            new BankingPermission("accountOperation"));

        if (_balance < amount) {
            throw new InsufficientBalanceException(
                    "Total balance not sufficient");
        } else {
            _balance = _balance - amount;
        }
    }

    ...

}
```

# …and access control

- Class ATM

```
package banking;

public class ATM {
    ...

    ...

    public void credit(final float amount) {

                    _currentATMCard.getBankLiaison().credit(_currentAccount, amount);

    }

    public void debit(final float amount) throws InsufficientBalanceException {

                    _currentATMCard.getBankLiaison().debit(_currentAccount, amount);

    }

    ...
}
```

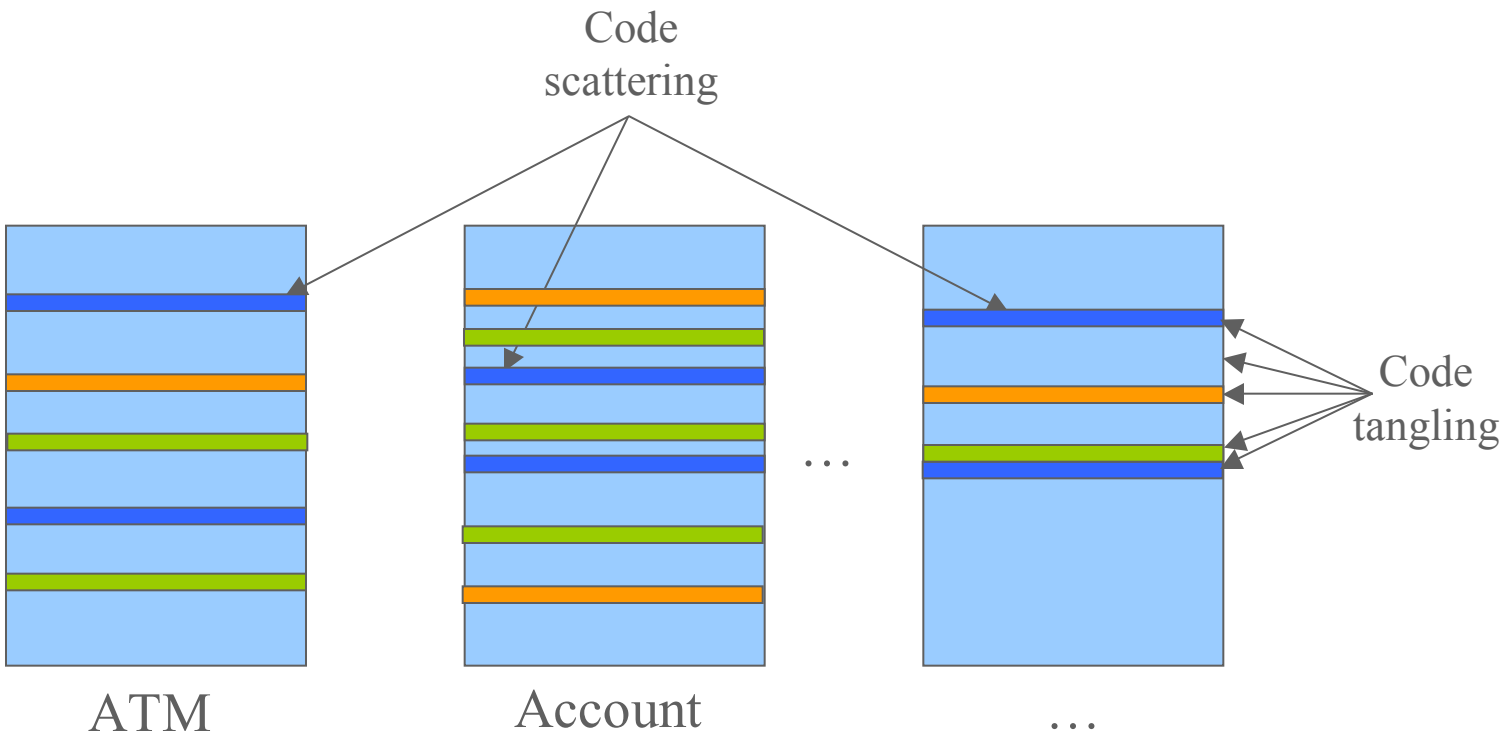
- Class Account

```
package banking;

public class Account {
    ...

    ...

    public void credit(float amount) {

        _balance = _balance + amount;
    }

    public void debit(float amount) throws InsufficientBalanceException {

        if (_balance < amount) {
            throw new InsufficientBalanceException(
                    "Total balance not sufficient");
        } else {
            _balance = _balance - amount;
        }
    }

    ...

}
```

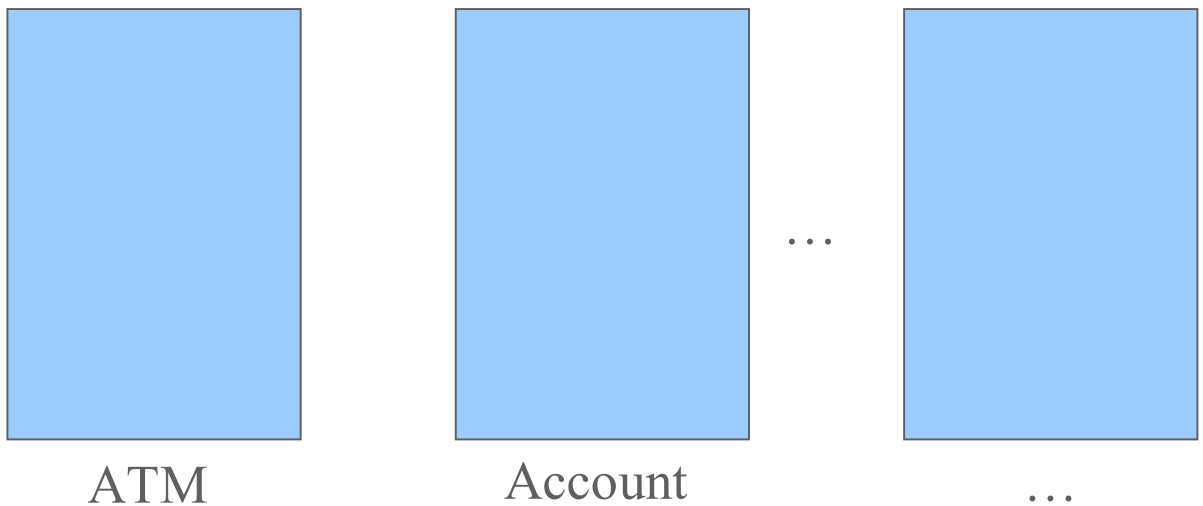
# …and so it goes

- Transaction management
- Thread safety
- Caching
- Resource pooling
- Profiling
- Policy enforcement
- Monitoring
- Testing
- Statement generation
- Service level agreement enforcement
- Business rules

# System Evolution: Conventional



# System Evolution: AOP based



# AOP

- A new programming methodology that facilitates modularization of crosscutting concerns
  - Also a new way of thinking about crosscutting concerns
- Introduces a new modularization unit: **Aspect**
  - Individual aspects are coded separately
    - Individual concern is dealt with one-at-a-time
  - Weaving rules are specified to combine different modules together
    - Weaver uses these to compose the system.

# Core concepts: Join point model

- Join point
  - An identifiable point in the execution of a program
  - Central, distinguishing concept in AOP
  - Exposed join points
    - Not all join points are exposed
    - Restricts the power
      - Therefore makes AOP powerful!
- Pointcut
  - Program construct that selects join points and collects context at those points.
  - The “when” part

# Core concepts: Dynamic crosscutting

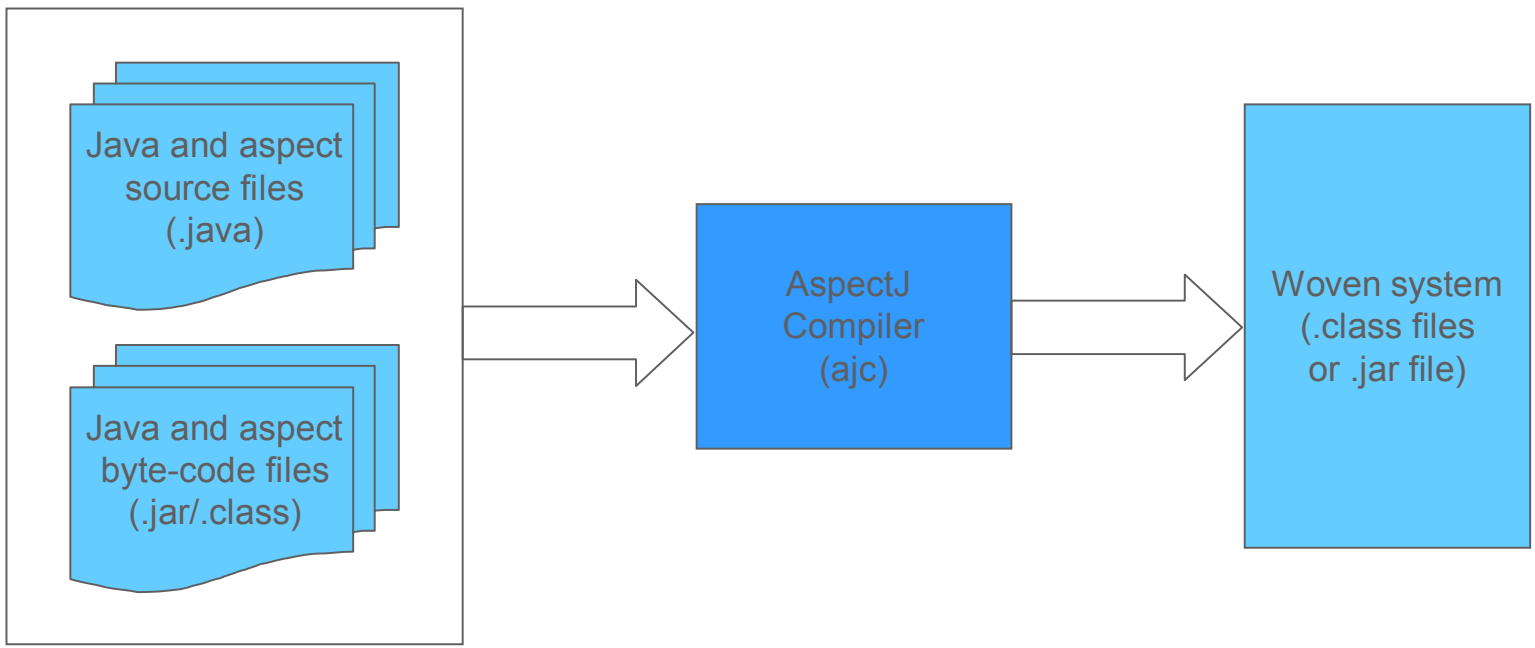
- Advice
  - Code to be executed at a join point that has been selected by a pointcut
  - The “what to do” part
  - Three kinds:
    - Before
    - After
    - Around

# Introducing AspectJ

- Aspect-oriented extension to Java
  - Compiles to standard Java byte-code
- Open source project at Eclipse.org
  - Originated at Xerox PARC
- Extensive tools support
  - IDE integration: Eclipse, NetBeans, JBuilder, Emacs
  - Ant, maven
  - Spring framework integration
  - BEA Weblogic integration
- Currently the leading AOP implementation
  - Numerous articles and papers
  - Four published books (one in Japanese)

A language that doesn't affect the way you think about programming, is not worth knowing. -- Alan J. Perlis

# AspectJ compilation



# AspectJ Pointcuts

- Recap: Pointcut is a program construct that selects join points and collects context at those points
- AspectJ's pointcut language is very powerful
  - Join points to capture can be precisely defined
  - Simple pointcuts may be combined to form composite pointcuts: !, &&, and || operators

# Pointcut supported

- Method/constructor execution

```
pointcut debitOp()
   : execution(void Account.debit(float))


pointcut accountCreation()
   : execution(Account.new())
```

- Method/constructor call

```
pointcut release()
   : call(void Connection.close())
```

- Exception handler

```
pointcut remoteHandler()
   : handler(RemoteException)
```

# Pointcut supported (contd…)

- Field read-access

```
pointcut amountRead()
   : get(float Account._amount)
```

- Field write-access

```
pointcut amountWrite()
    : set(float Account._amount)
```

- Many more…
  - Class initialization
  - Object initialization/preinitialization
  - Advice execution

# Pointcut and wildcards

- Capture the method debit() in Account that takes a float argument

```
pointcut accountDebitOp()
  : execution(void Account.debit(float))
```

- Capture ***all*** methods in Account that take a float argument

```
pointcut accountFloatArgOps()
  : execution(* Account.*(float))
```

- Capture all methods in Account that take ***any arguments***

```
pointcut accountAllOps()
  : execution(* Account.*(..))
```

- Capture all methods in Account ***and its subclass*** that take any arguments

```
pointcut accountAndSubAllOps()
  : execution(* Account+.*(..))
```

# Advice Examples

- Before advice

```
before(ATM atm) : execution(* ATM.*(..)) && this(atm) {
  System.out.println("Making ATM request"
                     + thisJoinPointStaticPart.getSignature()
                                              .getName()
                     + " from " + atm);
}
```

- After advice

```
after(Account account) returning
  : accountOperation(account) {
     observer.accountOperationSucceeded
       (new PropertyChangeSupport(account));
}
```

# Advice examples

- Around advice

```
Object around()
    : call(* Remote+.*(..) throws RemoteException){
      int retry = 0;
      while(true){
        try{
          return proceed();
        } catch(RemoteException ex){
        // log ex
        if (++retry > MAX_RETRIES) {
          throw ex;
        }
      }
    }
}
```

# A Logging Aspect: AspectJ

```
public aspect BankLoggingAspect {
  private static Logger _logger
    = Logger.getLogger("banking");

  public pointcut loggedOperations()
    : execution(* *.* (..))
      && !within(BankLoggingAspect);

  before() : loggedOperations() {
    Signature sig =
        thisJoinPointStaticPart.getSignature();
    _logger.logp(Level.INFO,
        sig.getDeclaringType().getName(),
        sig.getName(), "Entering");
  }
}
```

# Putting all together: Aspect

- Crosscutting type comprised of AspectJ primitives
- Like classes, an aspect
  - Can have access specification
  - Can extend another aspect, class or implement interfaces
  - Can be marked abstract
  - Can contain Java methods, fields etc.
- Unlike classes, an aspect
  - Cannot be instantiated
  - Cannot extend concrete aspect