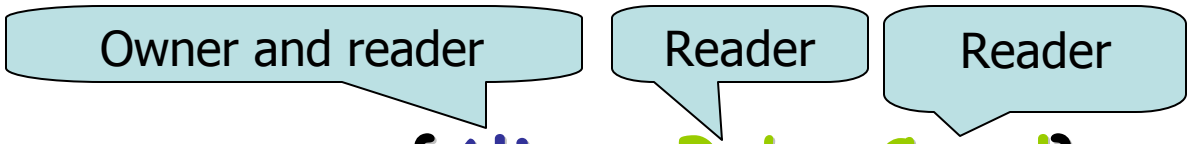
# Introduction to Jif
## or how to survive the Lab

Aslan Askarov

24 March 2006

# Decentralized Label Model (DLM)

- **Principals** (e.g. Alice, Bob)
- **Privacy policies**: {owner: reader list}




- **Labels** consist of a set of policies

{Alice:Bob, Carol; Bob:Alice}

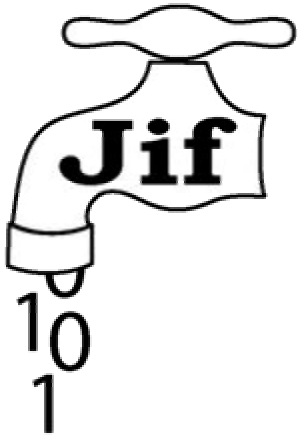
A principal is allowed to read data **iff** it is contained in the reader sets of all policies

# Labels: more examples

| | |
|---|---|
| {Bob::Alice:Bob} | Only Bob can read |
| {} | No policies. The most public label (bottom) |
| {Alice:Bob;Bob:Carol;Carol:Alice} | Nobody can read |

# Jif [Cornell University,1999-2006]

- Based of Java
- Implements DLM
- Every variable has a **labelled type:** Java type+security label
  - ex. variable declaration in Jif:



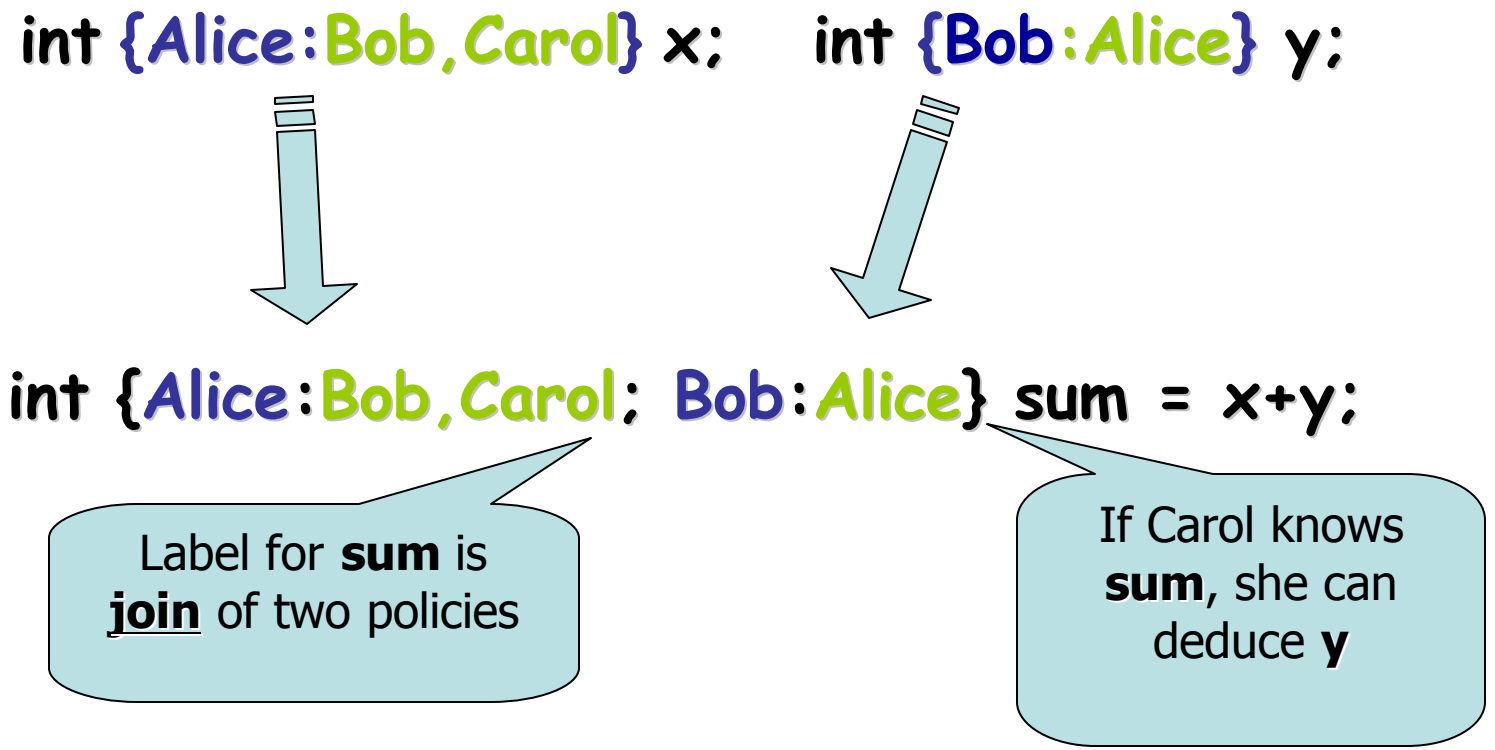

```
int {Alice:} x;
```

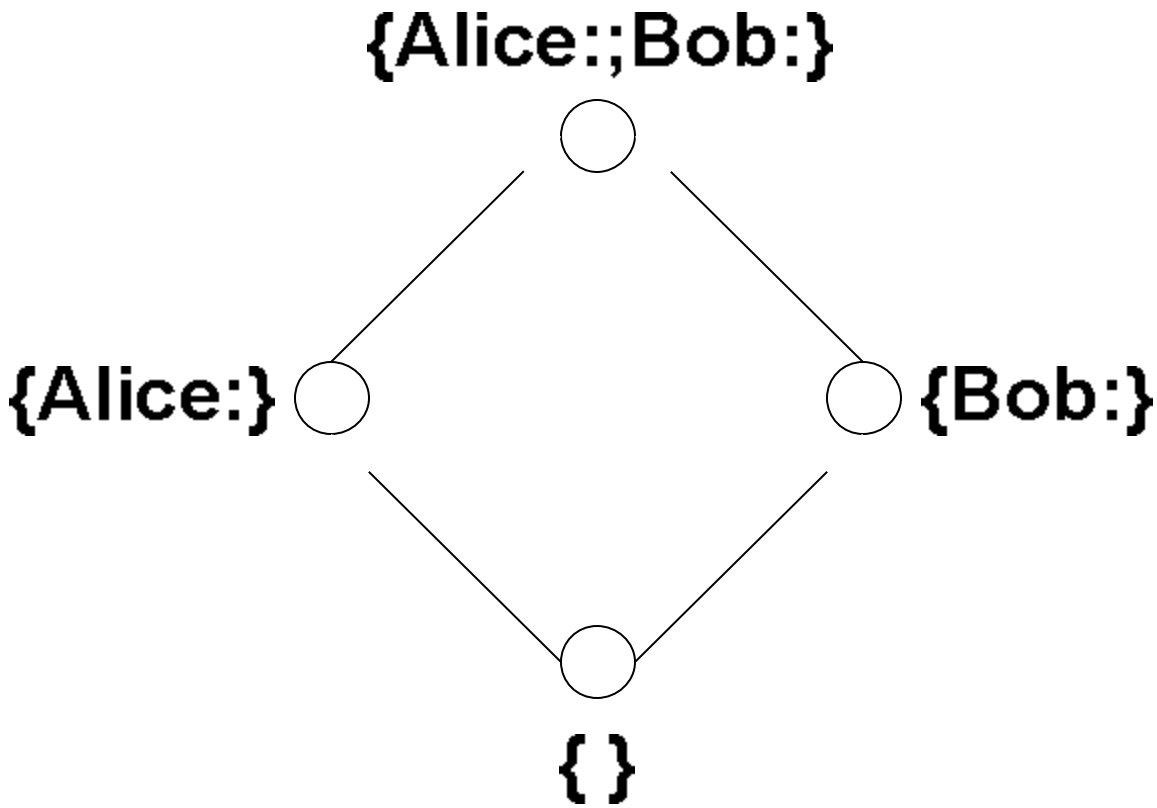
Ordinary Java type

Security label

Variable name

# How labels propagate




{Alice:;Bob:}
{Alice:}
{Bob:}
{ }

# Explicit and implicit flows

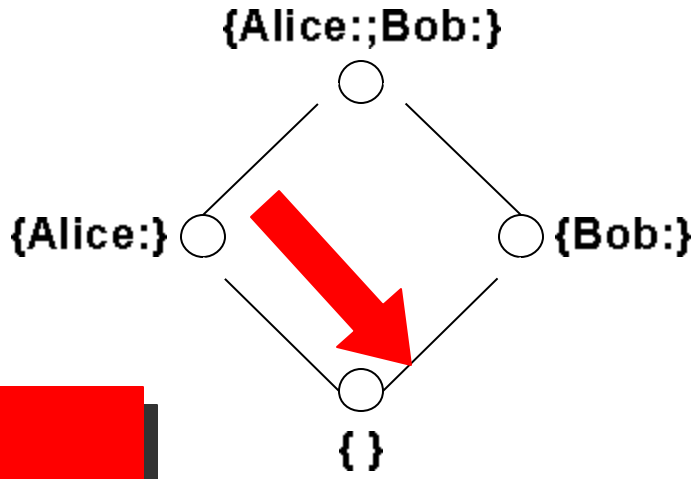
```
boolean {Alice:} secret;
boolean { } pub;
```

High

Low

```
pub = secret;
```

Explicit flow

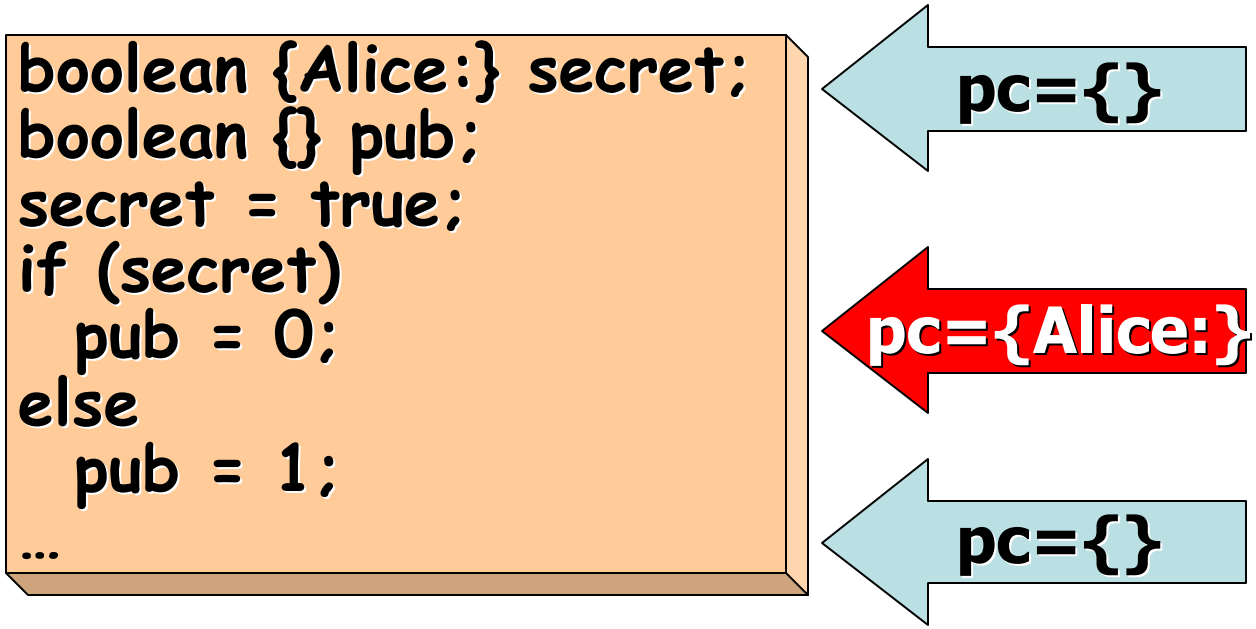



Implicit flow

```
if (secret)
  pub = 0;
else
  pub = 1;
```

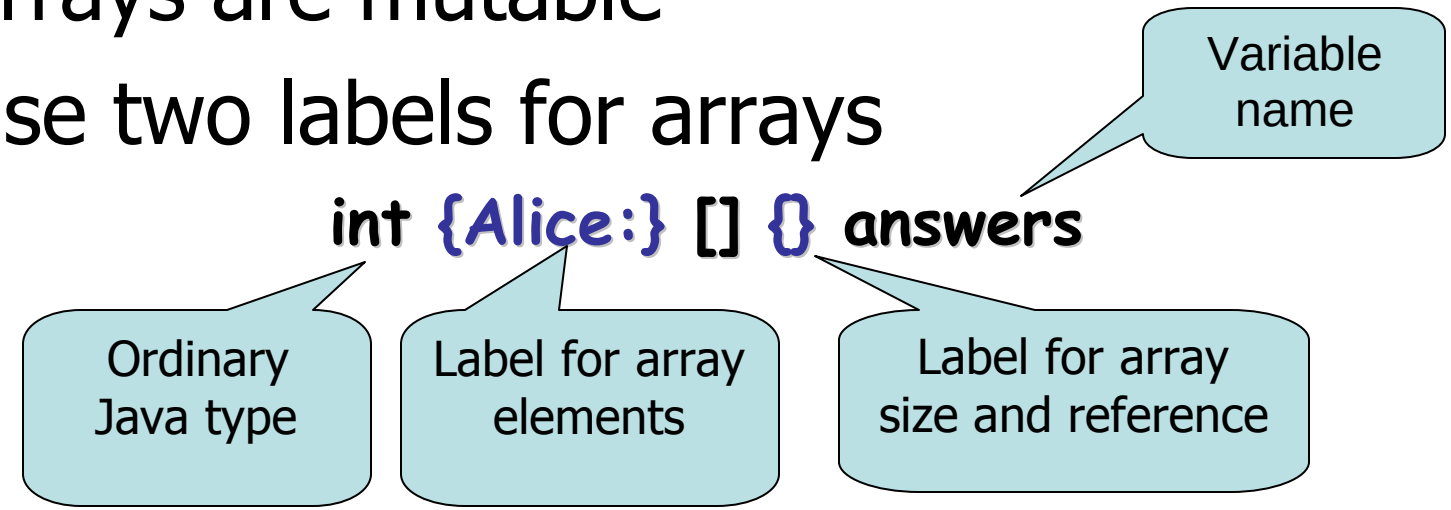
# Tracking implicit flows – pc label

- Program-counter label – what can be learned by knowing that the statement is evaluated



# Arrays

- Arrays are mutable
- Use two labels for arrays

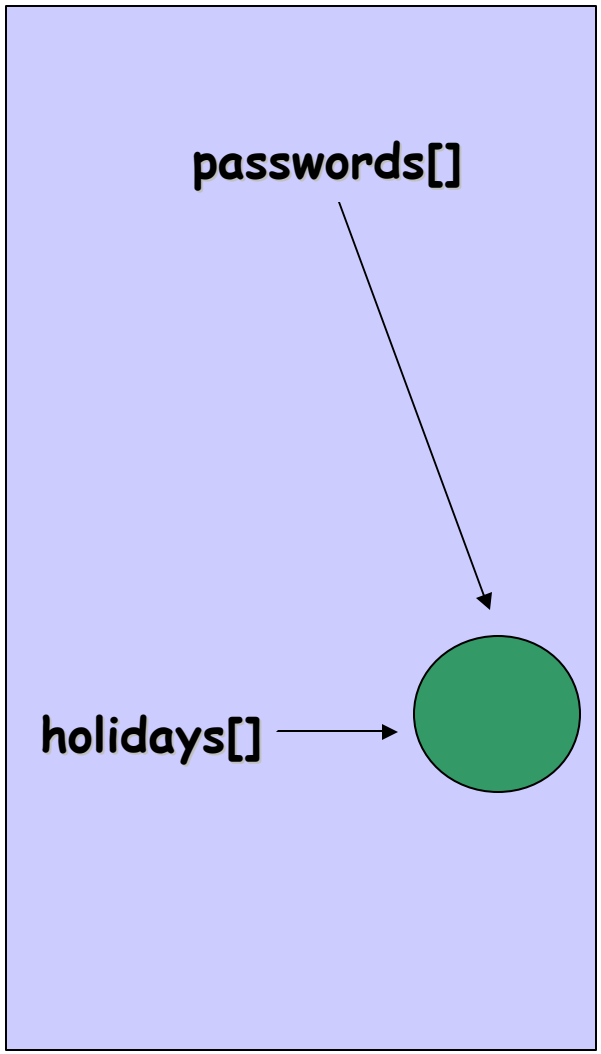



- In assignments **element labels** should be the same

```
int {Alice:} [] {} guess = new int [2];

guess = answer
```

# Arrays (con't)




```
string{root:}[] passwords;
string{}[] holidays;
passwords = holidays;
```

{root:}
|
{}

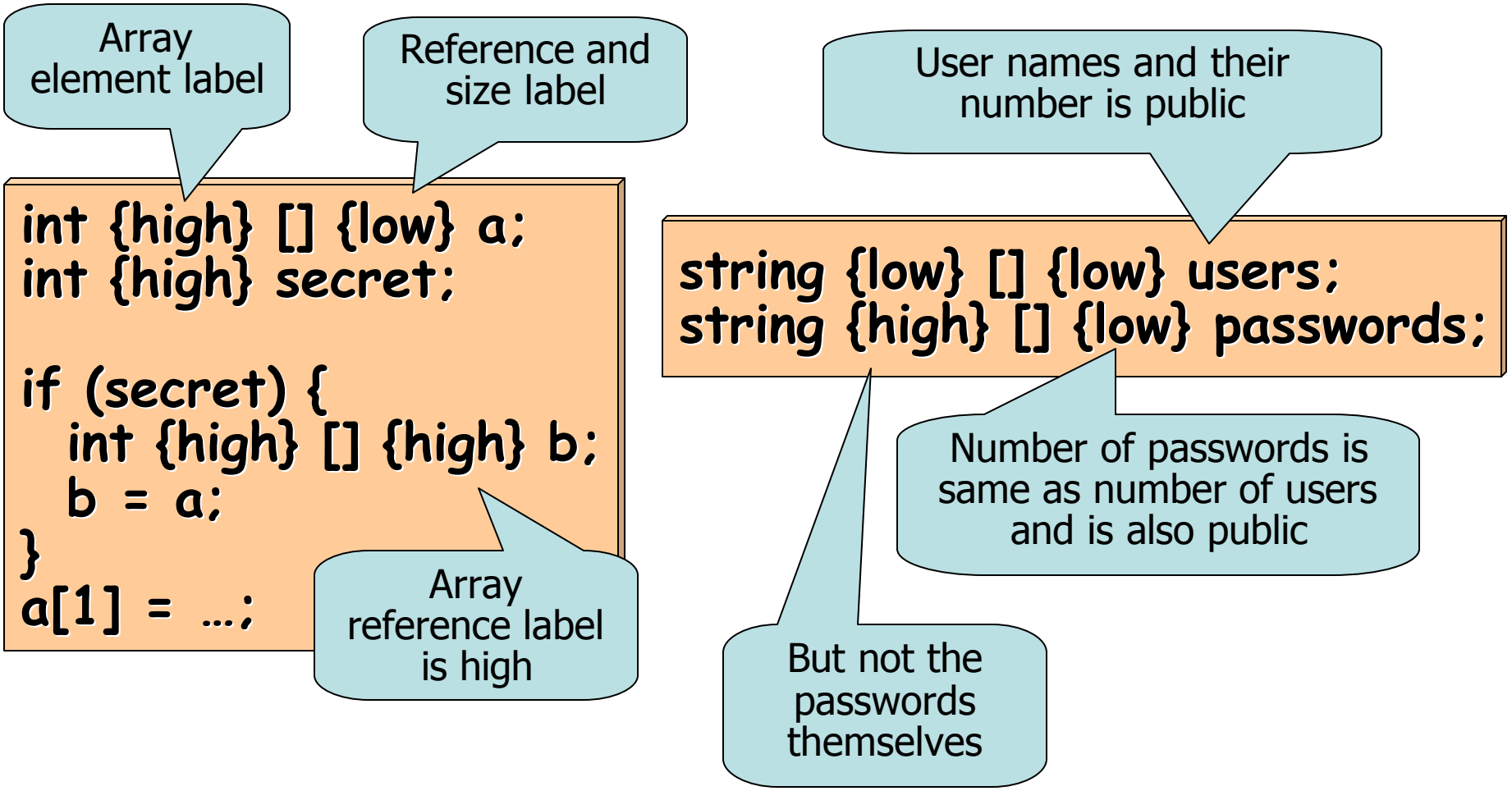
Looks safe, isn't it?

```
string {root:} newpwd = "!o#*a[ic/x1"
passwords [0] = newpwd;
```

What is the first public holiday of the year?

Or maybe the password of the first user?

# Two labels for Arrays



# Side effects

What are side-effects?

- Modifying mutable data structures (e.g.: arrays, classes)
- Assigning to class fields
- Printing a message to console
- Calling a method with side effects

# Does the method foo() has side-effects?

```
class T {
      public void foo() {
      }
  }
```

```
class T {
      public void foo() {
          int x = 0;
          x = 1;
      }
  }
```

```
class T {
      int x = 0;
      public void foo() {
          x = 1;
      }
  }
```

# Method labels

- Begin-label:
  - upper bound on the pc of the caller
  - lower bound on the side effects of the method
- End labels carry information about what can be learned by observing the method's (ab)normal termination
- Arguments may have labels just as other variables

# Begin-labels

```
class T {
    int {Alice:} k;
    int {} y;
    void f {Alice:} () {
        k=1;
    }
```

Begin-label

Side effect

```
    void g {} () {
        f();
        y = 0;
    }
}
```

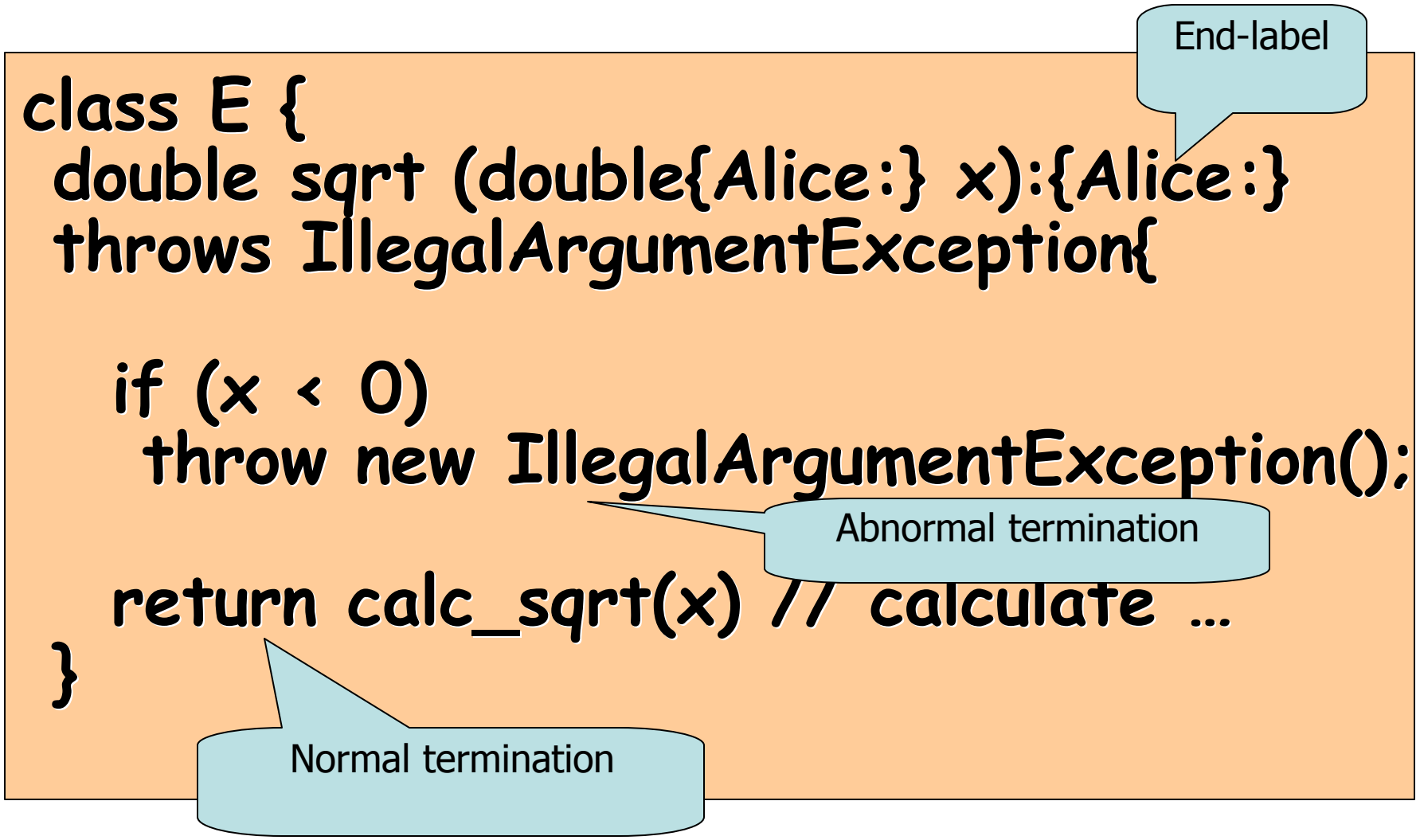
Begin-label

The lowest side effect

# Exceptions

- Exceptions may terminate method execution
- How exactly method terminates is information flow.
  - Terminating normally, e.g.:
    - **`return 0;`**
  - Throwing an exception, e.g.:
    - **`throw new IllegalArgumentException();`**
    - **`throw new NullPointerException();`**
- Declared exceptions affect end-labels

# End-labels


End-label
class E {
double sqrt (double{Alice:} x):{Alice:}
throws IllegalArgumentException{
if (x < 0)
throw new IllegalArgumentException();
return calc_sqrt(x) // calculate …
}
Abnormal termination
Normal termination

# End-labels (con't)

```
int {} low;
public void foo {} () {
    try {
      double {Alice:} x = 10;
      double {Alice:} sqrtX = this.sqrt(x);

      this.low = 0;

    } catch (IllegalArgumentException e) {}
  }
}
```

pc = {Alice:}

If this statement is executed, it is the case that call to **sqrt()** returned normally, i.e. x > 0

# Exceptions (con't)

- Runtime exceptions must be handled

```
int {Alice:} [] {Alice:} q;
void h {Alice:} (): {Alice:}
    throws (NullPointerException,
   ArrayIndexOutOfBoundsException) {
        q[1]=1;
}
```

end-label affect pc-label of the caller

NullPointerExeception
ArrayIndexOutOfBoundsException

# Exceptions (con't)

```
public boolean validate(A o) {
  if ( ! o.foo () ) return false;
}
```

Java code

```
public boolean validate(A{L} o):{L}
  throws NullPointerException {
    if ( ! o.foo()) return false;
}
```

Jif code version #1

Variable **o** may be **null**

# Exceptions (con't)

```
public boolean validate(A{L} o):{L}
   throws IllegalArgumentException {
      if (o == null)
        throw new IllegalArgumentException();
      if ( ! o.foo())
        return false;
}
```

Jif code version #2

Use **null pointer analysis** and throw **IllegalArgumentException** in the beginning

# Default labels

| | |
|---|---|
| Class fields | {} |
| Method arguments | <top> |
| Begin-label | <top> - no side effects |
| End-label | Join of exception labels, {} if no exceptions |
| Result label | Join of arguments and the end-label |
| Exception label | Method's end-label |
| Local variables | <pc-label> |

# {this} label

- **{this}** label corresponds to the label of the current class instance
- Maybe used for final fields

```
class A {
  final int {this} var1;
}
```

```
static void g() {
  A{Alice:} a;

  int {Alice:} x = a.var1;
}
```

Now **a.var1** has label {Alice:}

# Parameterized Classes

- Classes may be parameterized over labels and principals
- Can reuse the same class for different principals and labels

# Parameterized Classes

```
class Address[label L] {
    String {L} street;
    String {L} zip;
}
class Person[principal P] {
   String {P:} personNumber;
   Address[{P:}] address;
}
```

Name of the parameter

**Address** requires label as parameter

```
Address[{Alice:}] addr=new Address[{Alice:}]()
Person[Alice] alice = new Person[Alice] ();
```

**Person** requires principal as parameter

# Parameterized Classes

```
class T[principal P, label L] {
  int {P:} x;
  void foo {P:} (int {L} arg) {
      …
  }
}
```

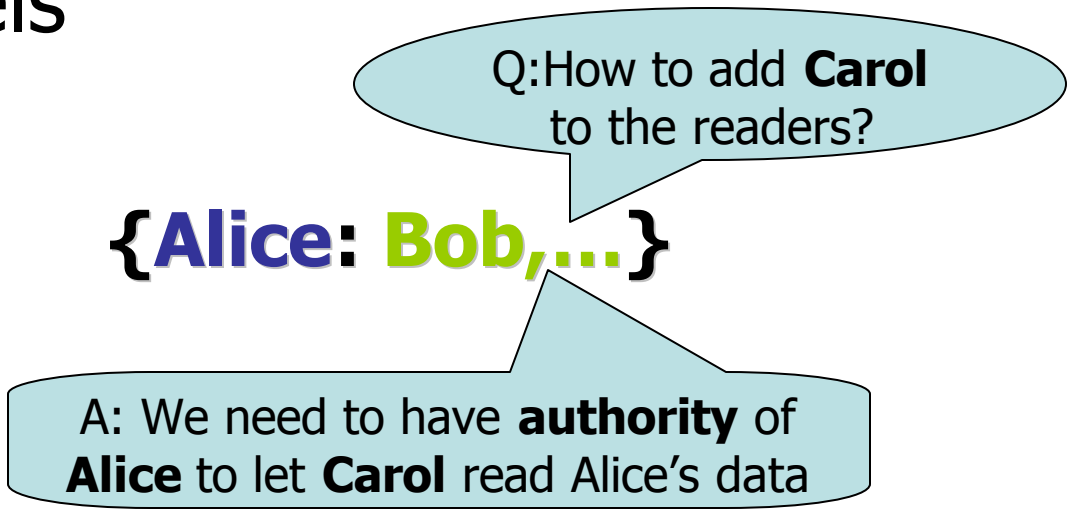
If needed, a class may have many parameters

Instantiation with actual principals and labels

```
T[Alice, {}] t1 = new T[Alice, {}] ();
T[Bob,{Alice:Bob}] t2=new T[Bob,{Alice:Bob}]()
```

# Authorities

Methods need authority of policy owners to modify labels



# Declassification

- Pure non-interference is too restrictive
- Often need to release some secrets
  - Result of password check
  - Crypto signature
- Declassification is **intentional information release**
- Jif supports declassification
- Methods need authority of principal whose policy is affected

# Declassification

```
class T [principal P] authority (P) {
  int {P:} x;

  void f {P:} () where authority(P) {

      int {} y = declassify (x, {P:}, {})
        …
  }
}
```

The class has an authority of P

This method has authority of P to perform declassification

Policy {P:} is removed from the label

# Authority vs. Caller

Authority clause grants authority and is dangerous

```
class T [principal P]  {
   int {P:} x;
   void f {P:} () where caller(P)
      int {} y = declassify(x,{P:},{})
      …
   }
}
```

Caller constraint requires the process at call site to have sufficient authority rather than grant it here

# Declassifying arrays and objects

```
class Declassifier[principal P, label L] {
   public static String{L}[]{L}
   declassifyStringArray{L}(String{P:}[]{P:} x_0)
   where caller (P) {
     String{P:}[]{L} x = declassify(x_0, {P:}, {L});
     if (x == null) return null;
     String{L}[]{L} y = new String[x.length];
     try {
        for (int i = 0; i < x.length; i++)
            y[i] = declassify(x[i], {P::L}, {L});
     } catch (Exception ignored) {}
     return y;
   }
}
```

# Lab

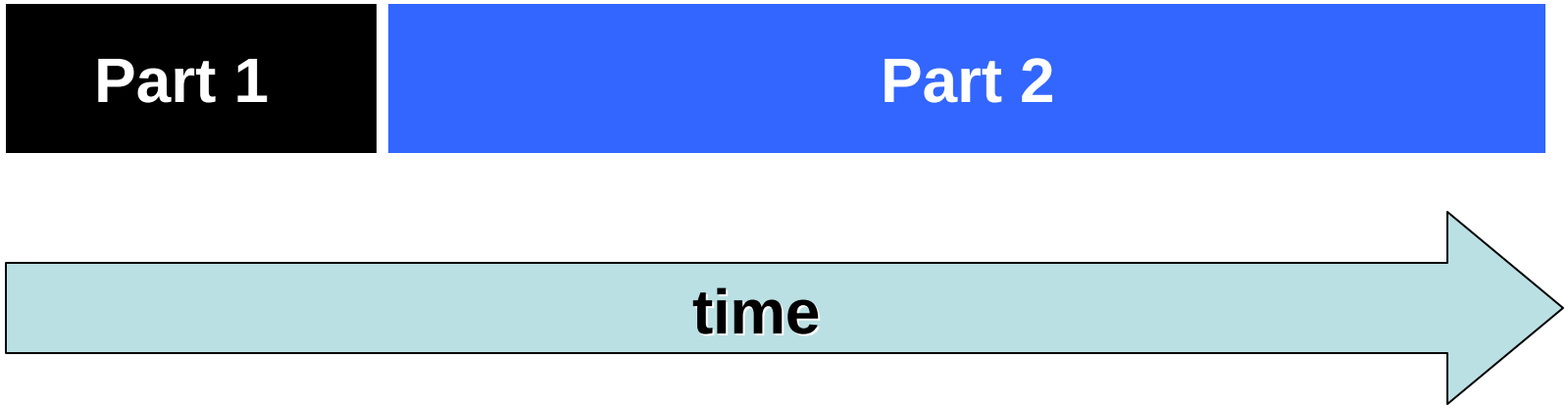
Scenario: Alice and Bob are taking multiple choice exam

Two parts

1. Write a malicious Java implementation for Student
2. Implement the lab in Jif

# Lab Timing

Be aware of timing




Work in groups of 2