ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of January 21, 2013

Abstract

- Review of object-oriented programming concepts:
  – Implementing a class.

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

Summary

- Object-oriented programming (OOP) has many facets;
- We have seen that OOP makes software design more concrete and helps with the development of complex software systems;
- OOP gives us the tools to better model a physical object such as the **Counter**, these tools are helping hiding implementation details;
- It is a way to organise (structure) programs such that the data and the methods that are transforming the data are grouped together in a single unit, called the object.

```
public class Counter {

    private int value = 0;

    public int getValue() {
      return value;
    }

    public void incr() {
        value++;
    }

    public void reset() {
        value = 0;
    }
}
```

Summary

Class and object, same thing?

Objects are entities that only exist at run-time.

Objects are “examples” (instances) of a class.

In a sense, the class is like a blue-print (specification) that characterizes a collection of objects.

In the case of a chess game application there can be a class which describes the properties and behaviours that are common to all the Pieces.

During the execution of the program, there will be many “instances” created: black king in D8, white queen in E1, etc.

Summary

Instance and object, same thing?

Instance and object refer to the same concept, the word instance is used in sentences of the of the form "the instance of the class . . . ", when talking about the role of an object.

The word object is used to designate a particular instance without necessarily referring to its role, "insert the object into the data structure".

You cannot design an object, you are designing a class that specifies the characteristics of a collection of objects. The class then serves to create the instances.

Package

In order to understand the **visibility modifiers**, we need to introduce the notion of **package** first.

What is a package?

A package is an organisational unit that allows to group classes;

A package contains one or more related classes.

E.g. The API (Application Programming Interface) is structured into packages.

```
java.io
java.lang
java.util
java.awt
```

Do you remember "import java.io.*;" from your ITI 1120 course?

Package

Packages may contain other packages (packages has members). This is convenient, but it has no semantic. E.g.

```
java.awt
java.awt.color
```

Package

A class has access to the other classes of the same package.

```
datebook
datebook.agenda
  Event
  Agenda
datebook.util
  Time
  TimeInterval
```

By default, a class is visible only to the other classes of the same package.

Package

The organisation of the packages also corresponds to the organization of the files and directories on the disk.

File: datebook/util/Time.java

```
package datebook.util;
class Time {
    ...
}
```

File: datebook/agenda/Event.java

```
package datebook.agenda;
public class Event {
     ...
}
```

Package

By default, a class is not accessible to classes from other packages.

File: datebook/util/Time.java

```
package datebook.util;
class Time {
    ...
}
```

File: datebook/agenda/Event.java

```
package datebook.agenda;
import datebook.util.Time;
public class Event {
    private Time start;
}
```

Cannot be compiled!

Package

You have to put some effort and add **public**.

File: datebook/util/Time.java

```
package datebook.util;
public class Time {
    ...
}
```

File: datebook/agenda/Event.java

```
package datebook.agenda;
import datebook.util.Time;
public class Event {
    private Time start;
}
```

Package

Why would you define a class that would not be public?

This would be a class used only by the classes of the package (an implementation detail).

Package

Packages are not only useful for regrouping classes they are extremely useful to avoid or resolve name conflicts!

E.g. **Event** is defined in **datebook.agenda.Event** as well as **java.awt.Event**!

Declaring a reference of type **java.awt.Event**.

```
java.awt.Event e;
```

Package

Finally, if a class is not assigned to a specific package, it belongs to an "unnamed package".

A package declaration is used to specify the name of the package to which the class belongs.

```
package datebook.util;

public class Time {
    ...
}
```

Class

What goes into a class?

Let's try to simplify things by looking at the elements of a class hierarchically.

Inside a class you can find:

- **variables**, **methods**, (nested) **classes**; (3)
- each belongs either to the **class** or the **instance**; (2)
- each can be **public**, **package**, (**protected**) or **private**; (4)
- each can be **final** or not. (2)

Class

How do you declare a class variable?

How do you declare an instance variable?

Did you notice that you have to put more efforts to create a class variable? By default, a variable, a method or a nested class belongs to the instance.

Class

How do you decide if you need to have an instance or a class variable?

Can you propose an example that would help understanding the difference between instance and class variables?

An instance variable defines the **state** of an object.

What do you mean by state?

I am defining the state of object as the current values of its instance variables.

Think about the class Counter! The class Time, etc.

Class variables

Ticket, serialNumber and lastSerialNumber.

Class variables

```
public class Ticket {
    private static int lastSerialNumber = 0;
    private int serialNumber;

    public Ticket() {
        serialNumber = lastSerialNumber;
        lastSerialNumber++;
    }
    public int getSerialNumber() {
        return serialNumber;
    }
}
```

Class variables

java.lang.Math provides many examples of static variables and methods.

```
public class Math {

 public static final double E = 2.71828182845904523536;

 static int  min( int a, int b )
 static double sqrt( double a )
 static double pow( double a, double b )

 public static double toDegrees(double angrad) {
     return angrad * 180.0 / PI;
 }
```

Class

How do you call a class method? Similarly, how do you access a class variable?

- Within the class where the member is defined, simply use the name of the method or variable;

- Otherwise, prefix the name of the member by the name of the class;

```
Math.sqrt( 36 );
Math.min( a, b );
```

- Otherwise, a reference to an object of the class that contains this member can also be used.

```
Counter c;
c = new Counter();
if ( c.getValue() < c.MAX_VALUE / 2 ) { ... }
```

Class vs instance method

What would be the impact of declaring a class method as an instance method?

What would be the impact of declaring an instance method as a class method?

Visibility modifiers

Members can be public, package or private (protected), what does this mean?

Final

The keyword final can also be placed in front a variable, what does this mean?

Modelling Time

Let's define a class to represent a Time in terms of the following,

hours: integer, 0 . . . 23 (inclusively)

minutes: integer, 0 . . . 59 (inclusively)

seconds: integer, 0 . . . 59 (inclusively)

$\Rightarrow$ two implementations will be considered.

Class

Are there mandatory parts?
(do we have to have variables? methods? constructor?)

A class starts with the keyword "class" followed by the name of the class; an identifier which is a singular noun that starts with a capital letter.

```
class Time {

}
```

Is this a valid class declaration?

Class

Yes, this can be saved to file, named Time.java and compiled.

Let's try:

```
> javac Time.java
```

produces **Time.class**.

Class

What does it buy us? Can it be used?

How about this:

```
class Test {
    public static void main( String[] args ) {
        Time t;
    }
}
```

Yes, a reference to an object of the class **Time** can be declared.

Class

How about this?

```
class Test {
    public static void main( String[] args ) {
        Time t;
        t = new Time();
    }
}
```

Constructor

Hum, the class **Time** does not even have a constructor.

Java provides you with a default constructor.

```
class Time {
  Time() {
  }
}
```

Attributes

Recall that **Time** should represent a time value, what are the attributes?

Hours, minutes and seconds (instance or class variables?)

Constructor

The default constructor exists unless you create your own constructor! Given the following definition:

```
class Time {
    int hours;
    int minutes;
    int seconds;
    Time( int h, int m, int s ) {
       hours = h;
       minutes = m;
       seconds = s;
    }
  }
```

The second statement will produce a compile time error.

```
  Time t;
  t = new Time();
```

Constructor

Are there situations where it would be useful to declare a constructor **private**?

```
public class {
  private Math() {}
}

class Test {
    public static void main( String[] args ) {
      Math m;
      m = new Math();
    }
}

// > javac Test.java
// Test.java:5: Math() has private access in
// java.lang.Math
//    m = new Math();
//            ^
```

```
// 1 error
```

Here the programmers who designed the class did not want you to create instances of the class **Math**.

Constructor

By the way, what is a constructor?

- A constructor is a block of statements that are executed when an object is created;
- A constructor has the same name as its class;
- A constructor can have arity 0, 1, 2, etc.
- Can only be called once, when the object is created, in the context "new . . . ";
- A constructor has no return type.

Since the constructor is called when the object is first created, a constructor generally serves to initialize the instance variables.

Interface

The public variables and methods are defining the **interface** of the class.

To use an object (or a class) all you need to know is its interface.

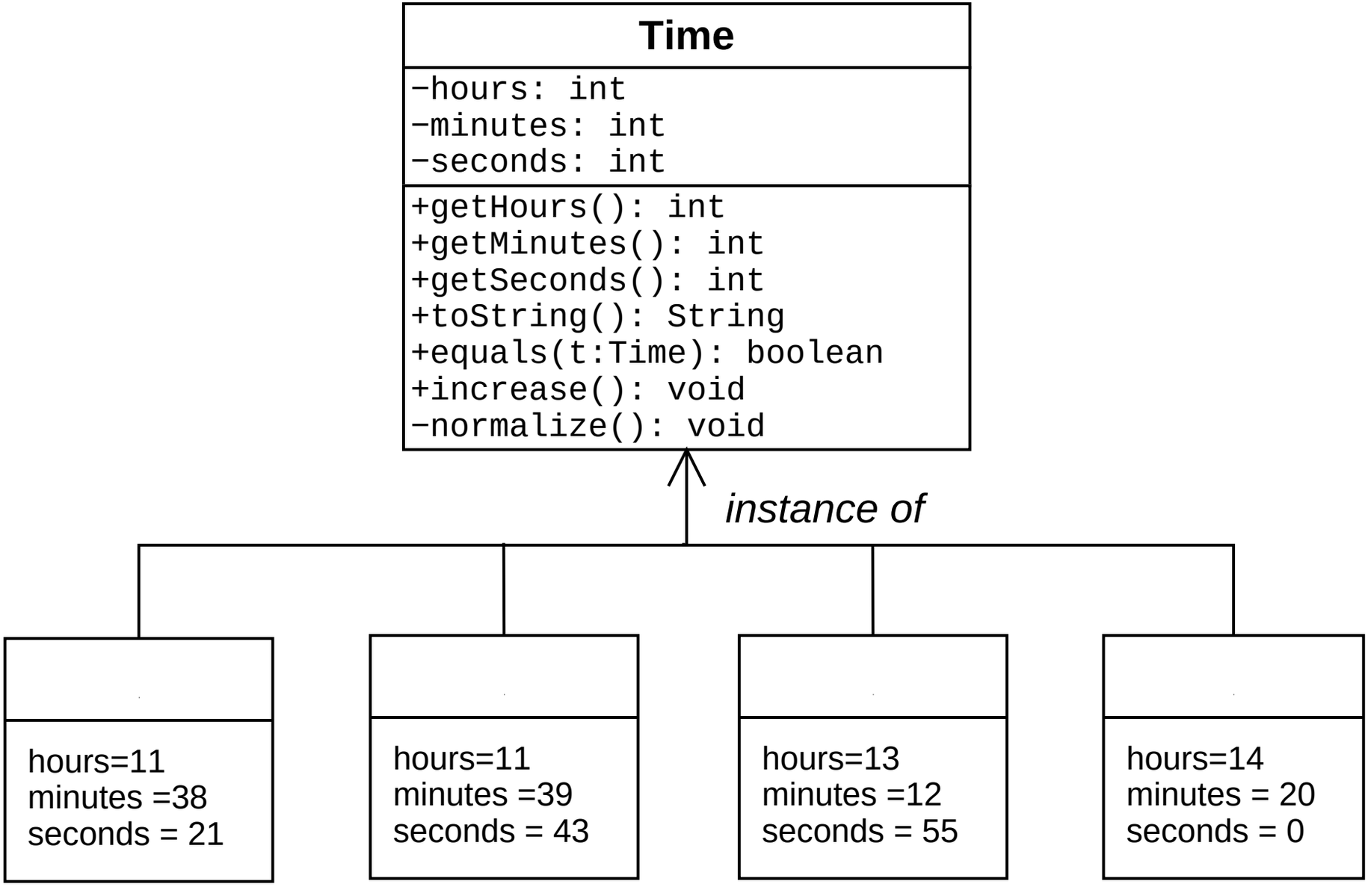
The interface must be carefully designed.

Changes to the public methods and variables (i.e. the interface) will affect the other classes that are making this software system.

The class as a specification

```
class Time {

    int hours;
    int minutes;
    int seconds;

    Time( int h, int m, int s ) {
        hours = h;
        minutes = m;
        secondes = s;
    }
}
```

$\Rightarrow$ All the objects specified by the class **Time** will have 3 variables: **hours**, **minutes** and **seconds**, as well as a constructor **Time()**.


Time
-hours: int
-minutes: int
-seconds: int
+getHours(): int
+getMinutes(): int
+getSeconds(): int
+toString(): String
+equals(t:Time): boolean
+increase(): void
-normalize(): void
instance of
hours=11
minutes =38
seconds = 21
hours=11
minutes =39
seconds = 43
hours=13
minutes =12
seconds = 55
hours=14
minutes = 20
seconds = 0

Creating objects

An object is created 1) with use of the reserved keyword **new** and 2) a call to a special method that has the same name as the class. That special method is called a constructor, it never returns a result, and possesses 0 or more formal parameters.

```
a = new Time(13,0,0);
b = new Time(14,30,0);
...
n = new Time(16,45,0);
```

n objects of the class **Time** have been created.

The class **Time** was used to create n objects.

There is a $1 : n$ relationship between the class and its objects.

Creating objects (contd)

```
a = new Time(13,0,0);
```

When creating an object, say a, from a class, here Time, we say that a is an instance of the class Time.

Here is Webster's definition of an instance:

> "(. . .) instance: an individual illustrative of a category"

$\Rightarrow$ . . . in a sense, an object is an example of a class.

Creating an object, let's be more precise

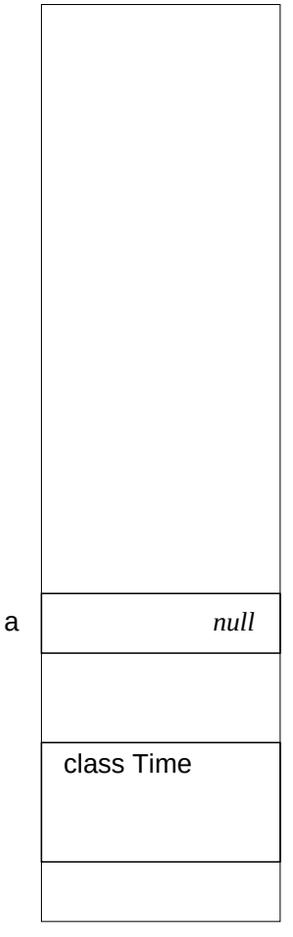
```
Time a;
a = new Time(13,0,0);
```

To be more precise, we should say that a is a reference variable that points to an instance of the class **Time**.

Variables have types!

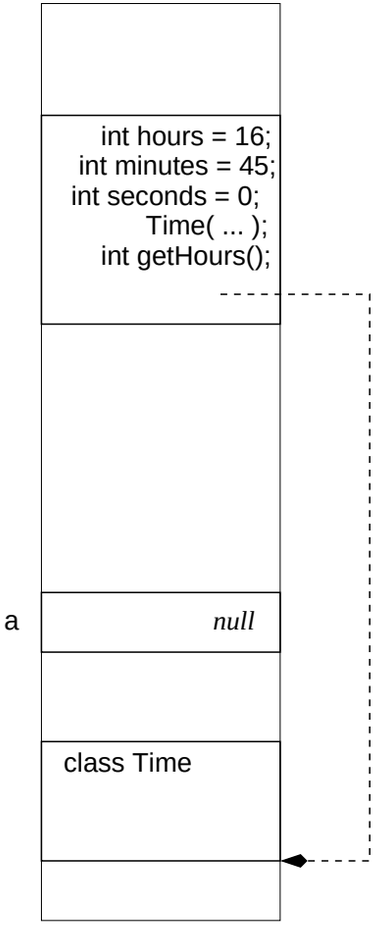
Objects have class(es)!

```
> Time a;
  a = new Time(16,45,0);
```



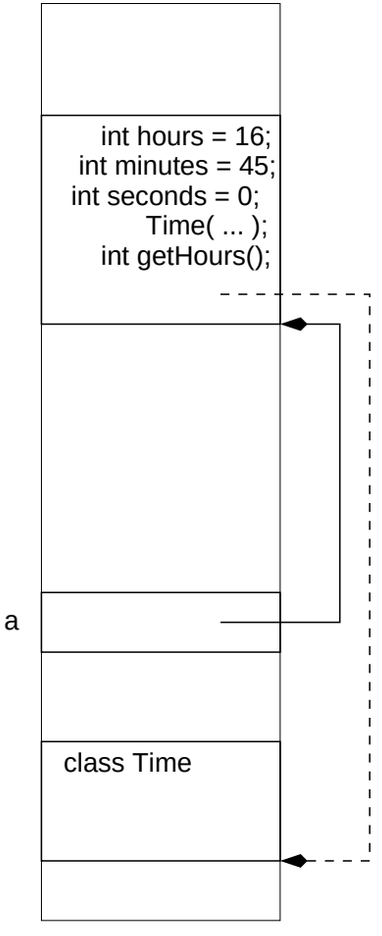
$\Rightarrow$ The declaration of a reference, here a reference to an oject of the class **Time**, tells the compiler to reserve enough space to hold a reference (an address, a pointer) toward an object.

```
  Time a;
> a = new Time(16,45,0);
```



$\Rightarrow$ At runtime, the execution of **new Time(16,45,0)** creates an object of the class **Time**, calls the constructor **Time(int,int,int)** which initializes the instance variables of the object; an object knows which class it belongs to, this is symbolized by the arrow here.

```
  Time a;
> a = new Time(16,45,0);
```



$\Rightarrow$ At runtime, the assignment **a = ...**, puts the reference ("address") of the object at the location designated by **a**, in other words, **a** now points toward the object.

Any particular problem?

With the current definition of the class **Time**, it's possible for the user to assign a value that would be out of range:

For example,

```
Time a = new Time( 13,0,0 );
a.hours = 55;
```

Solution

Object-oriented programming languages give us tools to control the "access to"/"visibility of" the data and methods, this principle is called **information hiding**.

```
class Time {
    private int hours;
    private int minutes;
    private int seconds;
}
```

What will happen?

```
public class Time {
    private int hours;
    private int minutes;
    private int seconds;
}

class Test {
    public static void main( String[] args ) {
        Time a = new Time( 1, 2, 3 );
        System.out.println( a.hours );
    }
}
```

The compiler will complain that hours has private access:

```
Main.java:6: hours has private access in Time
        System.out.println (a.hours);
```

^

Access methods

Since we no longer have access to the variables from outside the class, methods must be introduced to return the content of the instance variables:

```
int getHours () {
  return hours;
}
```

which will be used as follows:

```
a = new Time( 13,0,0 );
System.out.println( a.getHours() );
-> 13
```

```
public int getHours() {
    return hours ;
}

public int getMinutes() {
    return minutes ;
}

public int getSeconds() {
    return seconds ;
}
```

How about setters?

Should access methods be systematically created for each instance variable to assign a new value (setter)?

```
public void setHours( int h ) {
    hours = h;
}
```

No! Each problem has its own design issues. Here, we require that the value of a time object can only be changed by using the method **increment** that will increment the time represented by this object by one second.

Method equals

```
public class Time {

    private int hours;
    private int minutes;
    private int seconds;

    // ...

    public boolean equals( Time t ) {
        return  (( hours   == t.getHours() ) &&
                 ( minutes == t.getMinutes() ) &&
                 ( seconds == t.getSeconds() ) ) ;
    }

}
```

How about this?

```
public class Time {

    private int hours;
    private int minutes;
    private int seconds;

    // ...

    public boolean equals( Time t ) {
        return  (( hours   == t.hours ) &&
                 ( minutes == t.minutes ) &&
                 ( seconds == t.seconds ) ) ;
    }

}
```

Another problem to be fixed

There remains a problem with the current definition, it is possible that the initial values would be out of range:

```
Time a = new Time( 24,60,60 );
```

Solution

The constructor is a special method which is called (executed) when an object is first created, this is therefore the ideal place to put the necessary mechanisms that will ensure that the initial values are in the correct range.

Let's create a new method which will do the job of normalizing the values of the instance variables:

```
public Time(int hours, int minutes, int seconds) {
    this.seconds = seconds;
    this.minutes = minutes;
    this.hours = hours;
    normalise();
}
```

```
new Time (0,0,60) -> 0:1:0
new Time (0,59,60) -> 1:0:0
new Time (23,59,60) -> 0:0:0
```

normalize()

```
private void normalize () {
    int carry = seconds / 60;
    seconds = seconds % 60;
    minutes = minutes + carry ;
    carry = minutes / 60;
    minutes = minutes % 60;
    hours = (hours + carry) % 24;
}
```

The method is intended to be used only by the methods of the class Time, its visibility is set to **private**.

```
public void increase() {
    seconds++ ;
    normalise();
}
```

$\Rightarrow$ normalize () can be used at other places too.

public String toString()

Class

A **visibility modifier** can be put in front of the reserved keyword **class**.

public: any other class, either from the same compilation unit, from the same package, or any other packages, can use the class, i.e. can declare a reference of this type, access a class variable, etc.

By default, i.e. when no modifier is used, the access is package, i.e. other classes of the same package see this class.

Other reserved keywords, such as **final** and **abstract** could "qualify" the class, these concepts will be seen later.

Summary

We have considered the declarative aspect of a class, i.e. the class specifies the characteristics that are common to an ensemble of objects.

All the variables and methods that we have defined belong to the instance; they were instance methods and instance variables.

Consequently:

- each object (instance) has its own set of instance variables (sometimes called attributes), this means that each object reserves space for its own set of attributes;

- an instance method can access the content of the instance variables, you can think about this as if the method were executed inside the object.

Method and class variables

We now consider the dynamic aspect of a class.

The class is also a runtime entity, memory is allocated for it during the execution of the program.

Remember, we said earlier that there is a $1 : n$ relationship between the class and its instances. In particular, this means there is only one copy of a class in memory for n instances, a class variable is variable whose space is allocated into the class, and therefore this variable is unique and shared by all instances of the class.

Similarly, a class method is a method that belong to the class, there is only one copy it and it can only access the class variables (not the instance variables).

Class variables and methods are declared with the use of the reserved keyword **static**.

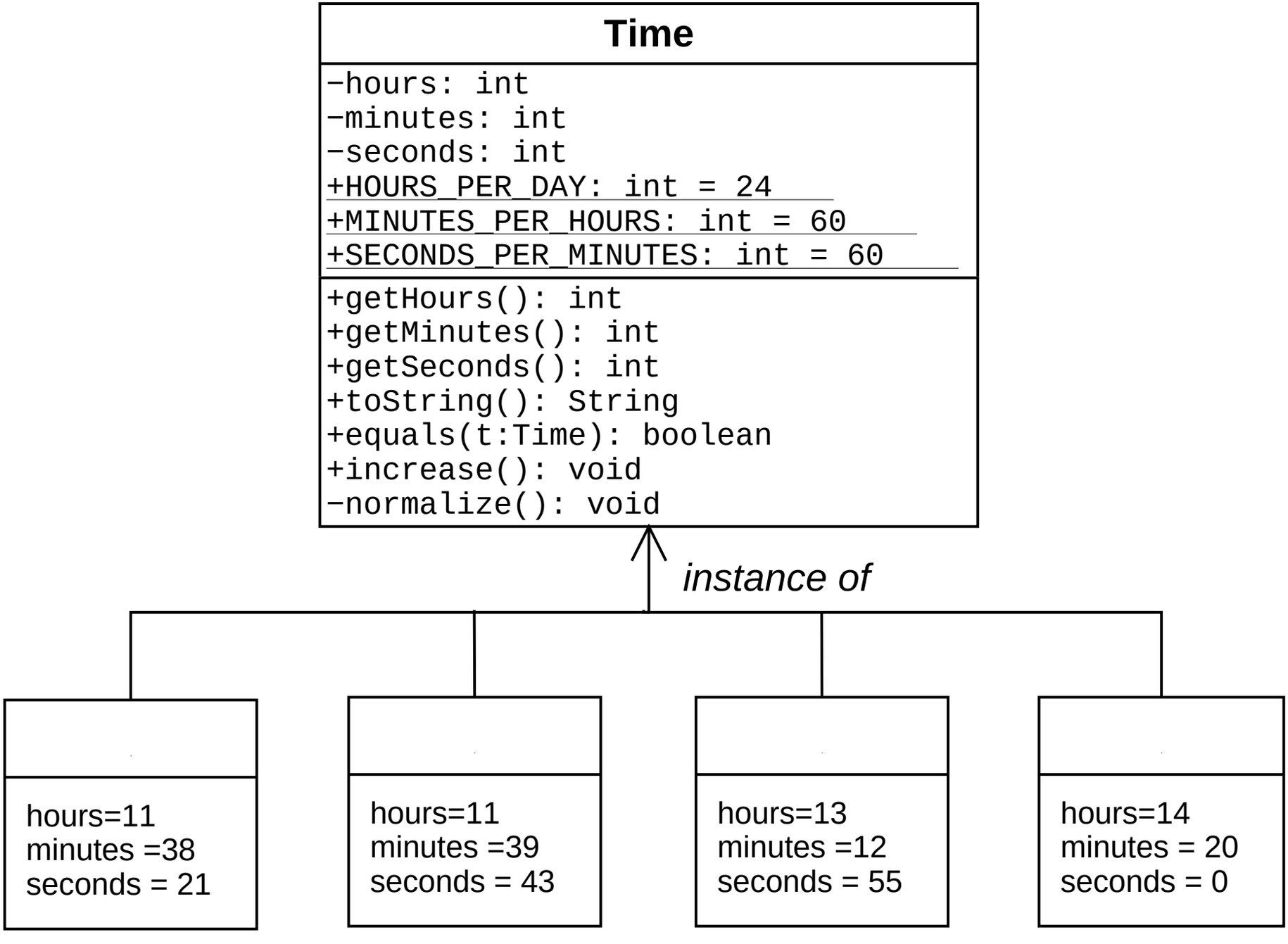
How to determine, if a variable should be a class variable or an instance variable?

Constants should always be class variables. By definition, a constant will not change throughout the execution of the program and therefore can be shared by

all the instances of a class.

Class variables

```
class Time {

    // class variables (these are constants,
    // notice the final keyword)

    static public final int HOURS_PER_DAY      = 24;
    static public final int MINUTES_PER_HOUR   = 60;
    static public final int SECONDS_PER_MINUTE = 60;

    // instance variables

    private int hours;
    private int minutes;
    private int seconds;

    ...
}
```


Time
-hours: int
-minutes: int
-seconds: int
+HOURS_PER_DAY: int = 24
+MINUTES_PER_HOURS: int = 60
+SECONDS_PER_MINUTES: int = 60
+getHours(): int
+getMinutes(): int
+getSeconds(): int
+toString(): String
+equals(t:Time): boolean
+increase(): void
-normalize(): void
instance of
hours=11
minutes =38
seconds = 21
hours=11
minutes =39
seconds = 43
hours=13
minutes =12
seconds = 55
hours=14
minutes = 20
seconds = 0

```
class Time {
    static public final int HOURS_PER_DAY      = 24;
    static public final int MINUTES_PER_HOUR   = 60;
    static public final int SECONDS_PER_MINUTE = 60;

    private int hours ;    // 0 - 23 (inclusively)
    private int minutes ;  // 0 - 59 (inclusively)
    private int seconds ;  // 0 - 59 (inclusively)
    ...
}
```

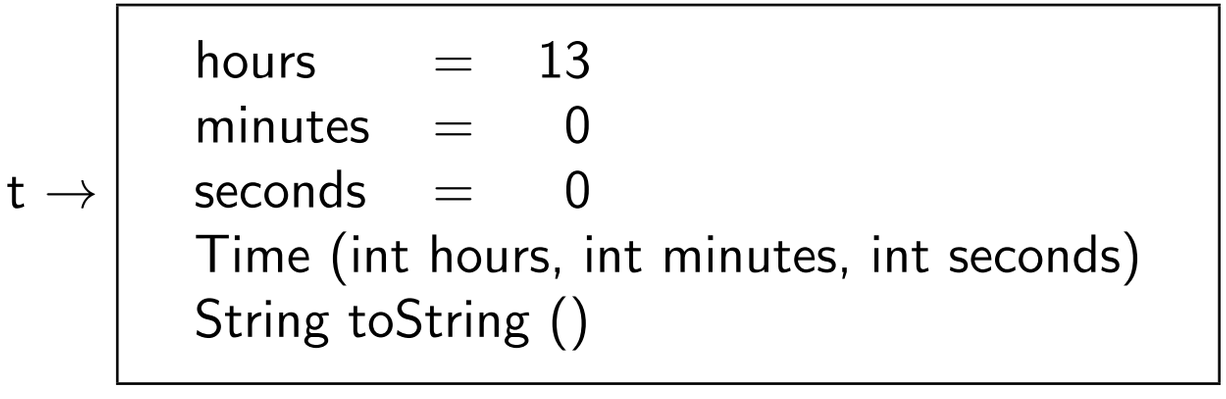
There are 2 ways to access class variables,
via the class:

```
System.out.print(Time.HOURS_PER_DAY);
=> 24
```

or via an instance:

```
Time t = new Time (13,0,0);
```

```
System.out.print(t.HOURS_PER_DAY);
=> 24
```

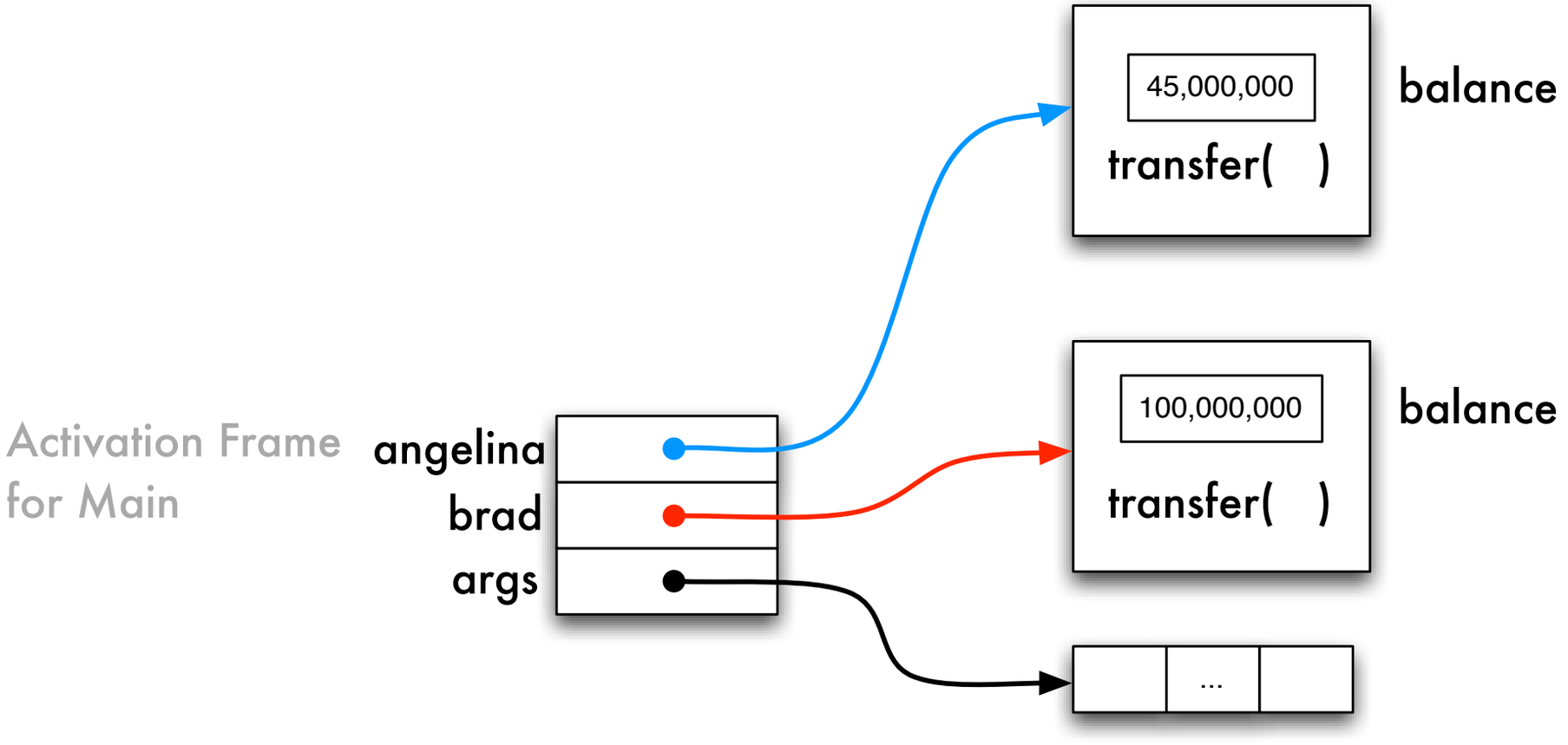


What's this?

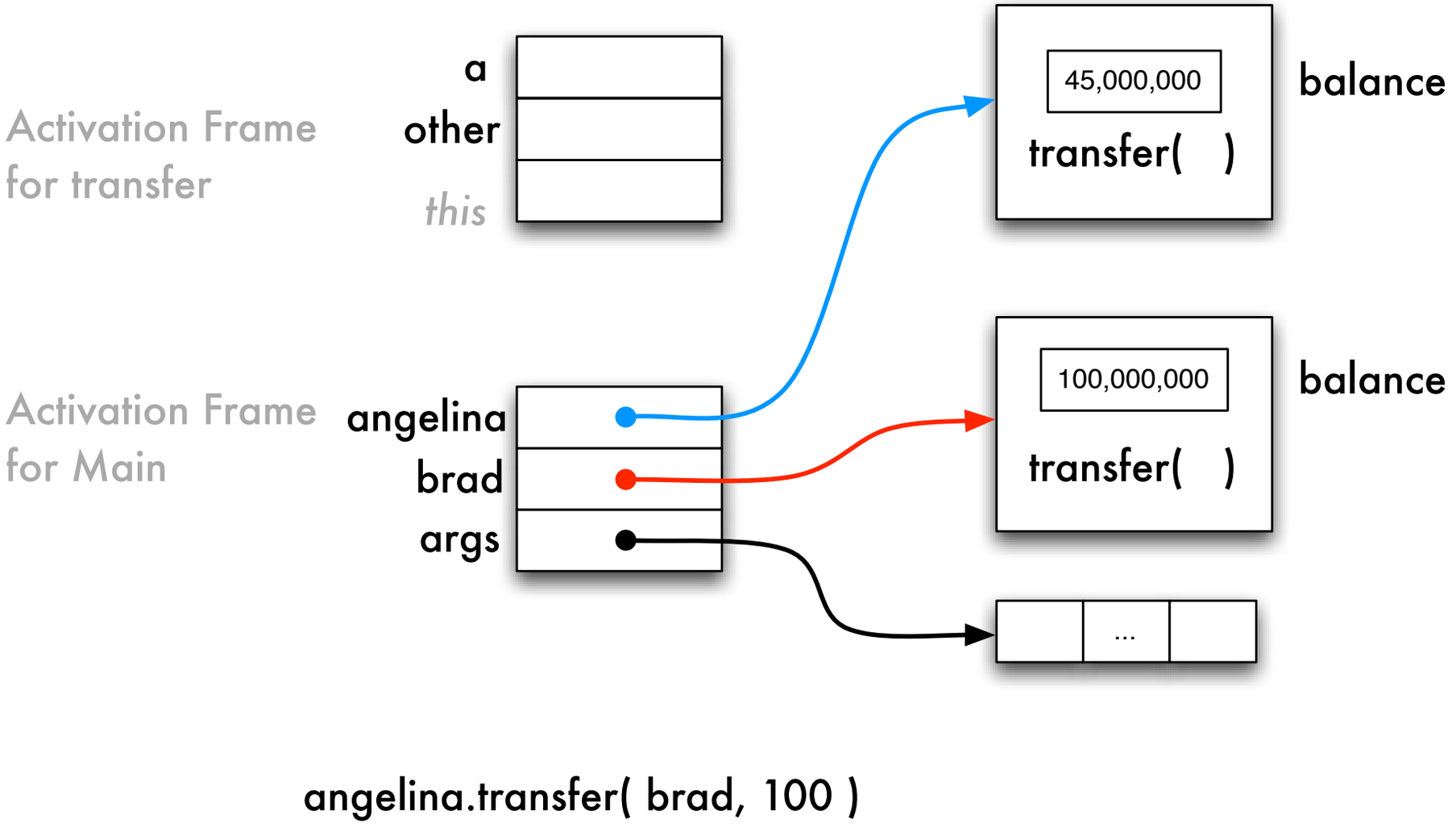
"this" is a self-reference.

BankAccount example

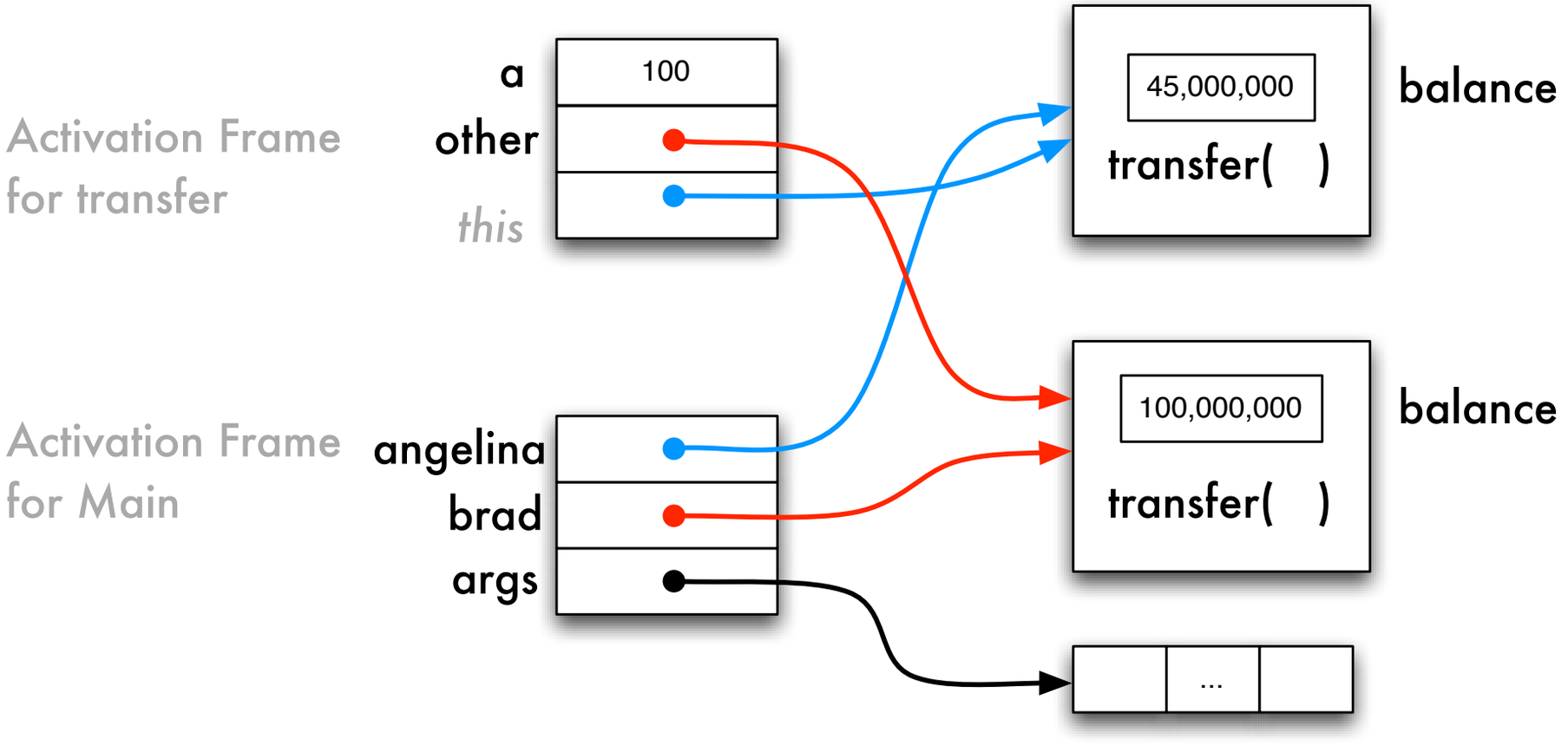
```
public class BankAccount {

  private double balance;

  // ...

  public boolean transfer(BankAccount other, double amount) {

    if (this == other)
        return false;

    ...
  }
}
```



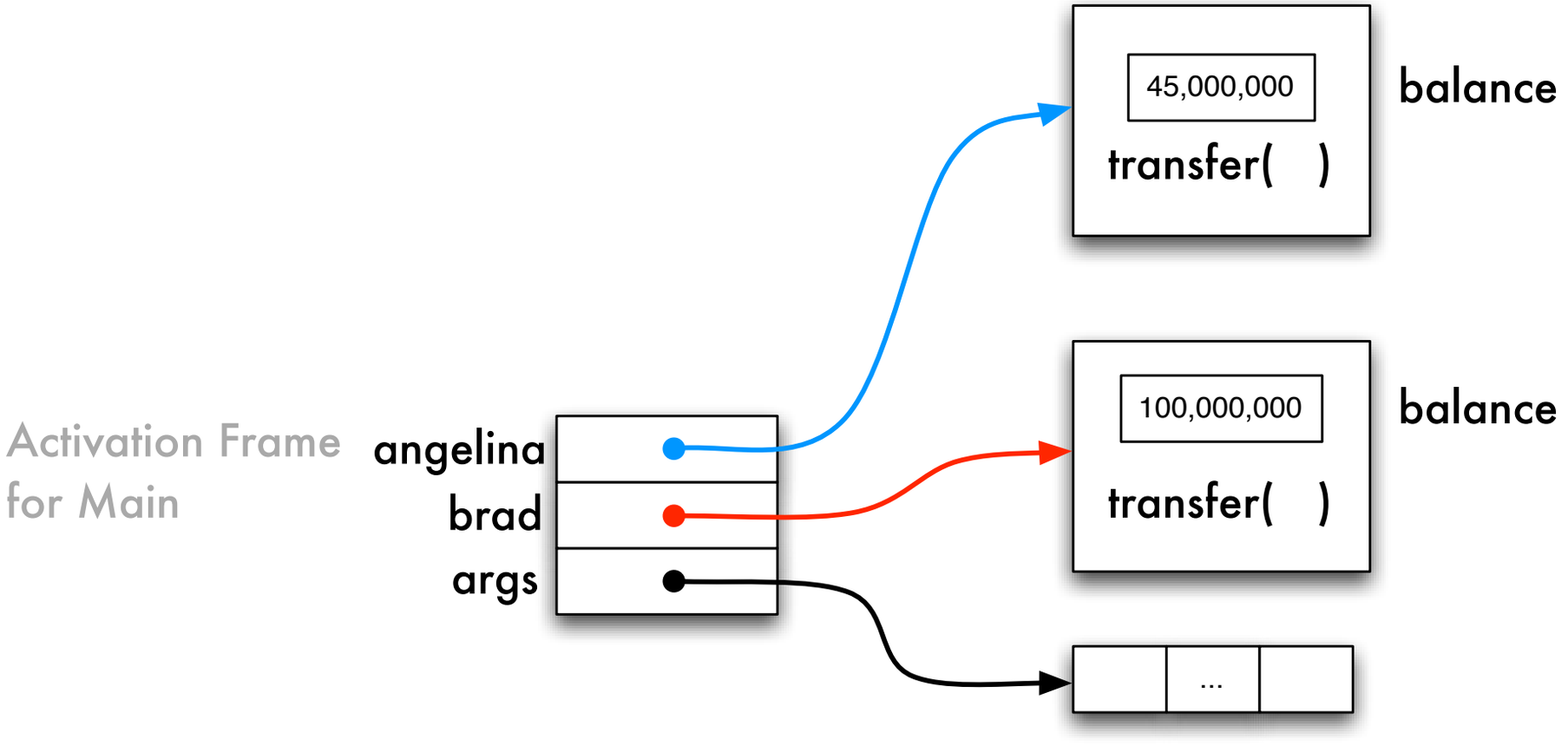
angelina.transfer(brad, 100)



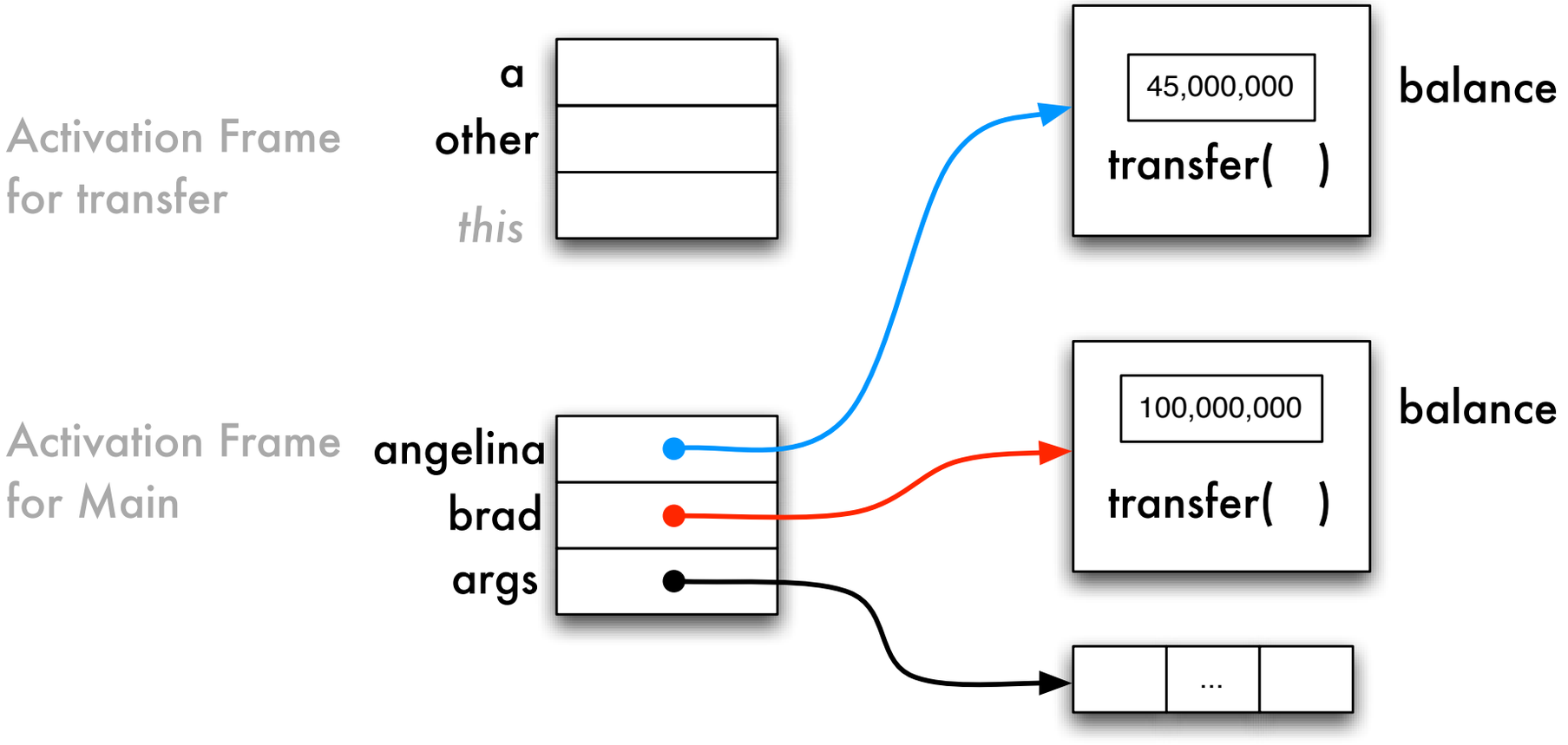
angelina.transfer(brad, 100)



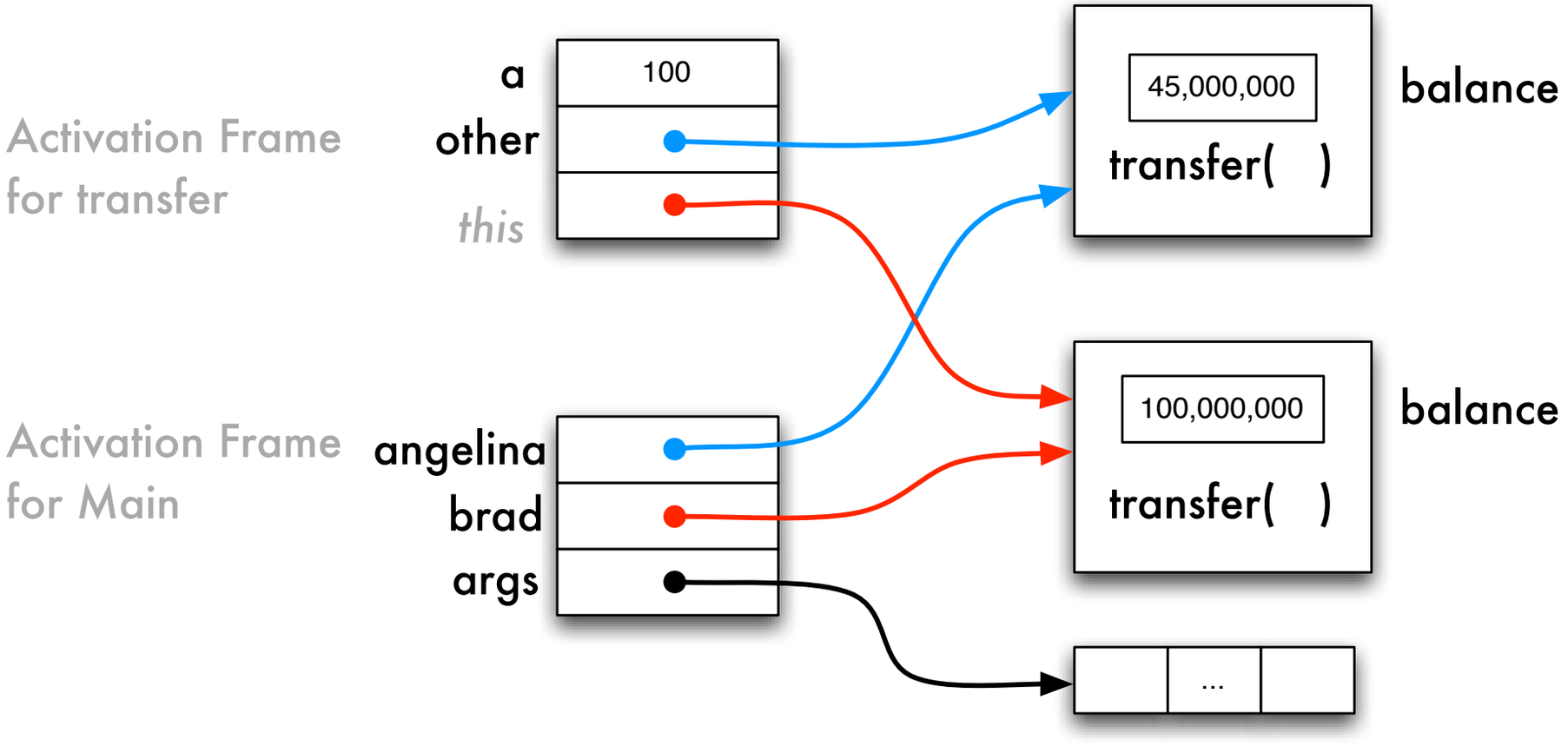
angelina.transfer(brad, 100)



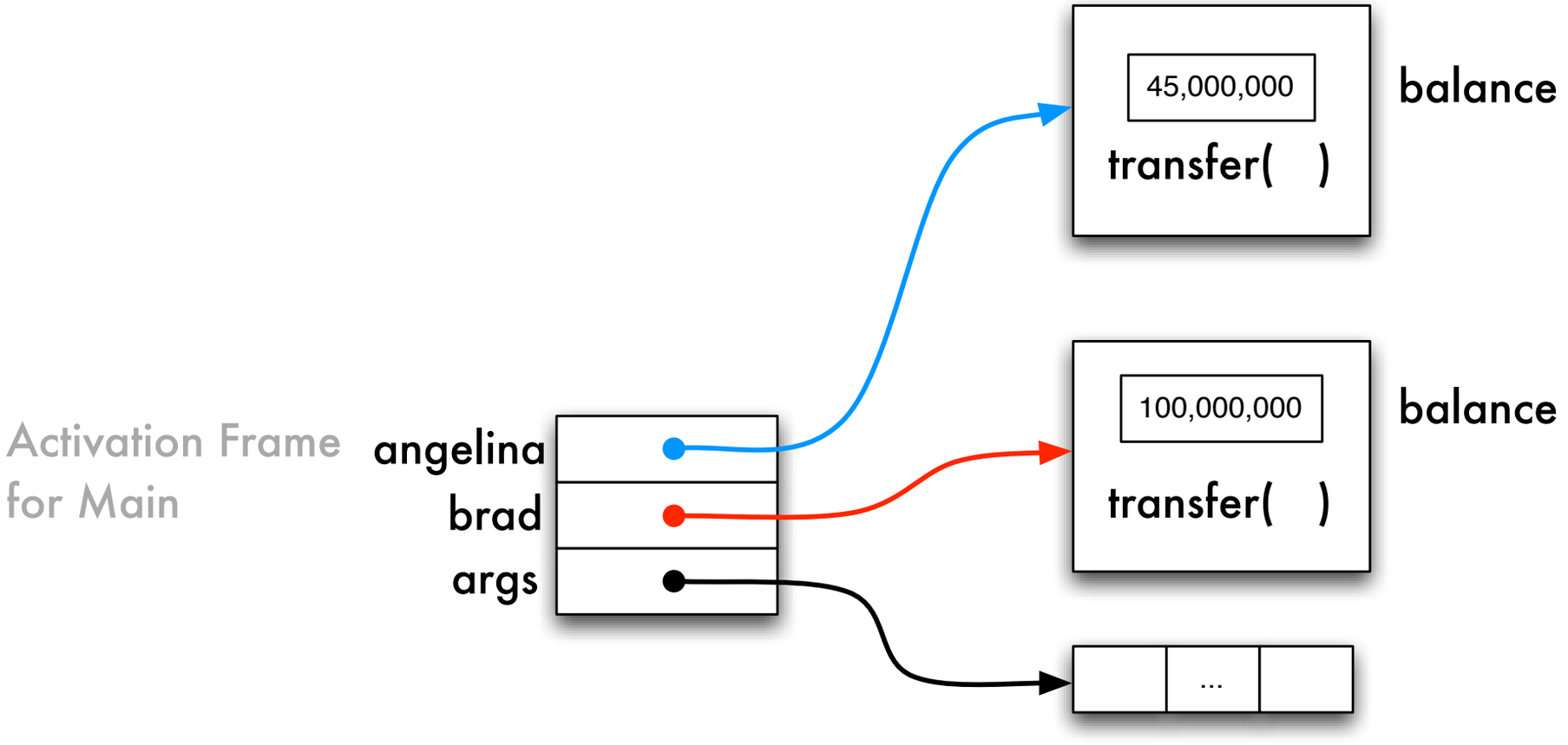
brad.transfer(angelina, 100)



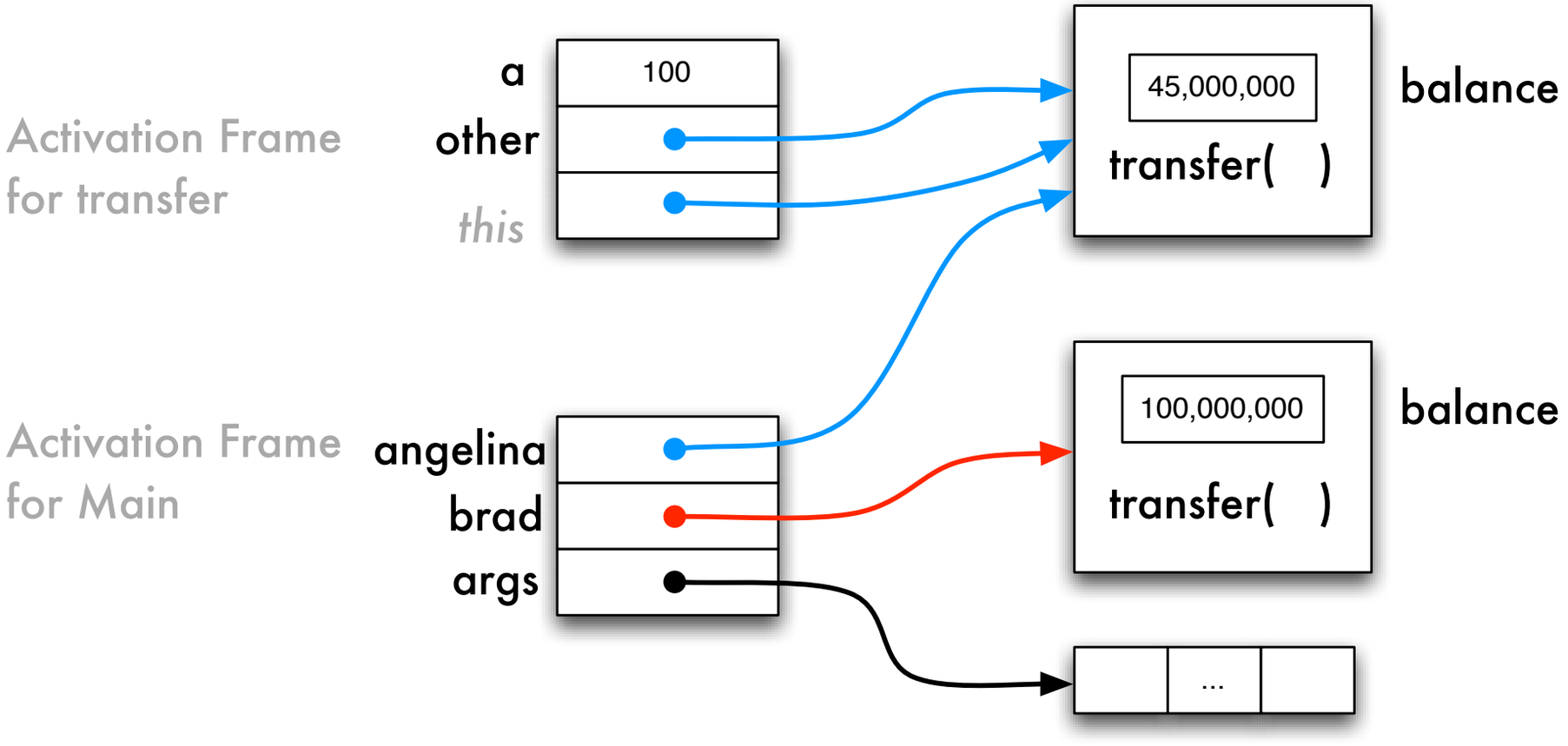
brad.transfer(angelina, 100)



brad.transfer(angelina, 100)



angelina.transfer(angelina, 100)



angelina.transfer(angelina, 100)

What's this?

```
public class Date {

  private int day;
  private int month;

  public Date( int day, int month ) {
     this.day = day;
     this.month = month;
  }

  // ...
}
```

What's this?

[New !!!!]

Don't use instance variables where local variables would do the job, why?

1. Wrong design - it shows that you don't understand the different kinds of variables well

2. Each object uses more memory than it needs too, and it will do so for all its life

3. It gets into the way of the garbage collector

Eg.

```
class A {

  int[] boxes;

  putObjectsIntoBoxes () {
```

```
        boxes = new int[ 1000 ];
    }

}
```

Resources

Java Language Specification
(syntax and semantics of the language)

java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html

JavaTM 2 Platform API Specification
Standard Edition, v 1.4.2
(libraries)

java.sun.com/j2se/1.4.2/docs/api