Smalltalk in a Nutshell

Objects & classes

Messages & methods

Inheritance & metaclasses

Smalltalk: Everything is an Object

- Application objects: customer, inventory
- GUI objects: button, text editor
- Foundation: string, set, numbers, booleans
- Tools: browser, debugger, compiler, GUI builder
- Language implementation: class, method, context,

Communicating with an Object

All computation is done by objects.

The only way to interact with an object is to "send a message" to it.

Smalltalk has three kinds of syntax for sending a message.

All messages have same implementation.

Three Kinds of Message Syntax

Unary messages

aSalaryChange date

Keyword messages

earned at: date add: money

Binary messages

(worked - 40) max: 0

Sending a Message

Object responds to message by looking in its *class* for a *method* with the same *selector*.

If it doesn't find the method, it looks in its *superclass*.

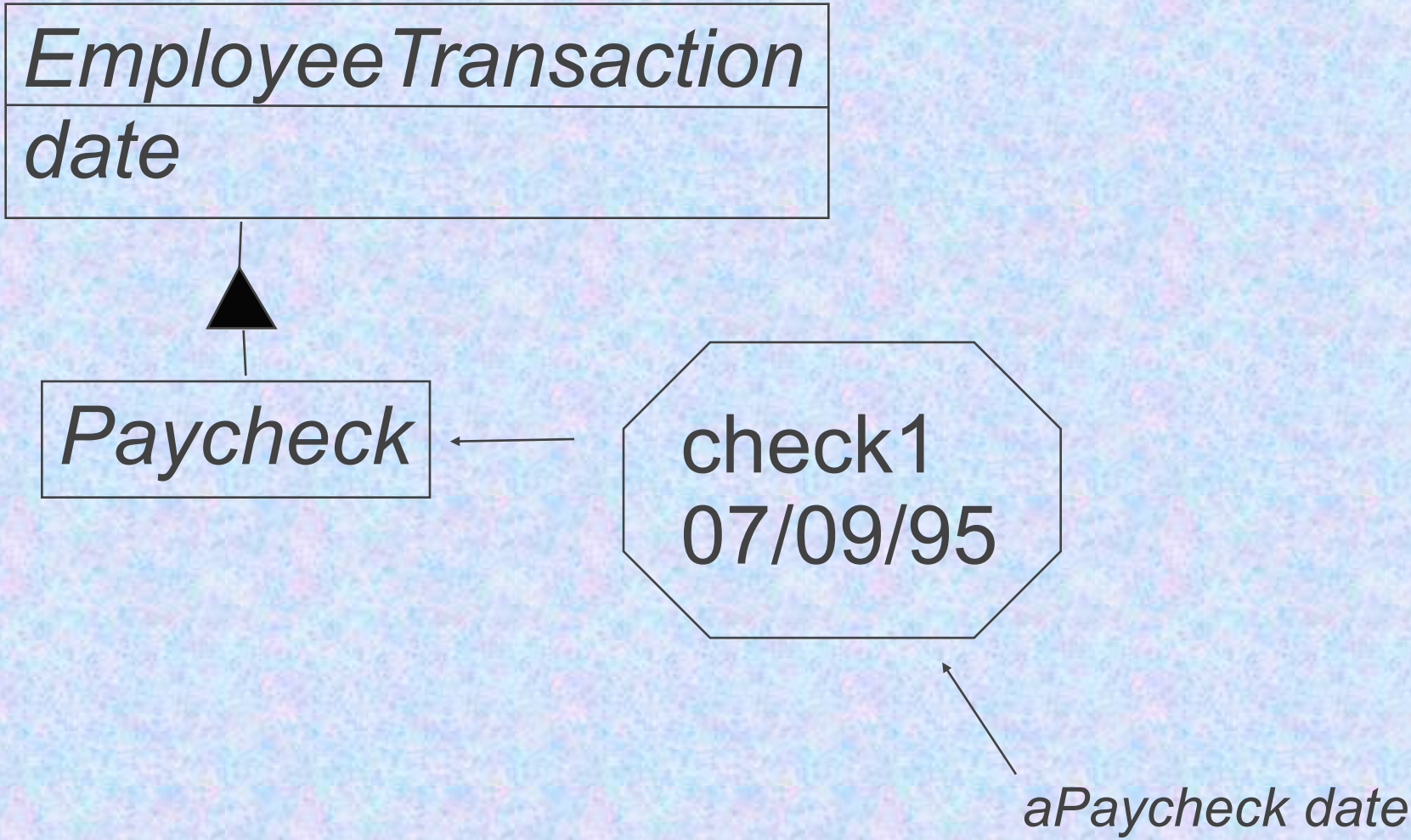
Repeat until it finds the method. If it never does, there is an error.

Message Lookup and Inheritance

EmployeeTransaction has subclasses Paycheck, SalaryChange, and Timecard.

EmployeeTransaction defines the instance variable *date* and the method:

```
date
    ^date
```


EmployeeTransaction
date
Paycheck
check1
07/09/95
aPaycheck date

Smalltalk Expression Syntax

Constants

3.675, 14, 'hello', #weight, $d,

#(#foo 'bar' 92)

Assignments and variables

v := v + 1

Messages

Smalltalk Expression Syntax

Sequences. Blocks.

```
x < y ifTrue: [z := x] ifFalse: [z := y].
paychecks do: [:each | each post]
```

Cascades

```
aSet add: #blue; add: #red
```

Smalltalk Method Syntax

Returns

```
^socialSecurity + federalTaxes +
  stateTaxes
```

Variables in Smalltalk

blockArguments and temporaries

methodArguments and temporaries

instanceVariables

Can be accessed only by the object's methods.

Variables in Smalltalk

ClassVariables

Shared by all objects in the class or in subclasses.

Globals

Shared by all objects.

Variables in Smalltalk

```
EmployeeTransaction subclass: #Paycheck
    instanceVariableNames: 'amountPaid
    taxes totalPaid totalTaxes '
    classVariableNames: 'AmountFormat
    DateFormat '
    poolDictionaries: ''
    category: 'Employee'
```

Using Variables

```
printOnCheckStream: aStream
    aStream cr; cr.
    aStream next: 40 put: (Character space).
    DateFormat print: date on: aStream.
    aStream cr.
    ...
```

More variables

```
test
    "PayrollSystem test"
    | payroll day1 ralph |
    day1 := Date newDay: 5 year: 1996.
    payroll := self new.
    ralph := Employee new name:
                        'Ralph Johnson'.
```

(continued)

```
ralph changeSalaryFor: day1 to: 20.
payroll addEmployee:  ralph.
self
    employee: ralph
    hours: self aLittleOvertime
    starting: day1.
^payroll
```

Initializing an Object

Every object needs to be initialized.
Uninitialized variables are nil.

The initialization method often defines the type of a variable.

Two methods: one class and one instance.

Class Methods

Class is an object. You can define methods for it, too.

For class Date class

day: dayInteger year: yearInteger

```
^self new day: dayInteger year:
  yearInteger
```

Instance initializing method

For class Date

day: dayInteger year: yearInteger

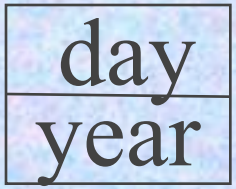
day := dayInteger.

year := yearInteger

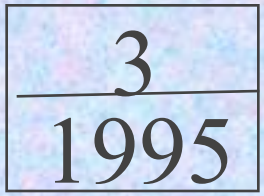
Creating a Date

Date day: 3 year: 1995

Date new day: 3 year: 1995



day: 3 year: 1995



Complete Smalltalk

Expressions (method definition)

Variables

Classes, inheritance

Class methods / instance methods

Pseudovariables

nil, true, false

self, super, thisContext

Smalltalk (the language) is trivial

Complexity is class library.

New language extensions fit in as well as numbers and control structures.

Language extension => core language is trivial

Implications

class library = language extension

=>

must know class library

must standardize class library

merging class libraries is like merging language extensions

hard to make class libraries

Uses of Inheritance

Provide a common implementation

Provide a default implementation

Provide a parameterized implementation

Magnitude

Superclass of classes like Number and Date
with <, >, <=, >=

<= aMagnitude

```
^(self > aMagnitude) not
```

> aMagnitude

```
self subclassResponsibility
```

Date

Instance variables: day, year

```
< aDate
    "Answer whether the argument, aDate, precedes
    the date of the receiver. "
    year = aDate year
        ifTrue: [^day < aDate day]
        ifFalse: [^year < aDate year]
```

Sorting

To sort a list, store it in a SortedCollection.

SortedCollection uses < to sort elements unless told otherwise.

```
SortedCollection new
        add: (Date newDay: 3 month: 'March' year:
          1955);
        add: (Date newDay: 4 month: 'November'
          year: 1960);
        add: (Date newDay: 22 month: 'June' year:
          1983);
        yourself
=>  SortedCollection (3 March 1955 4 November
    1960 22 June 1983 )
```

Abstract Classes

Abstract class

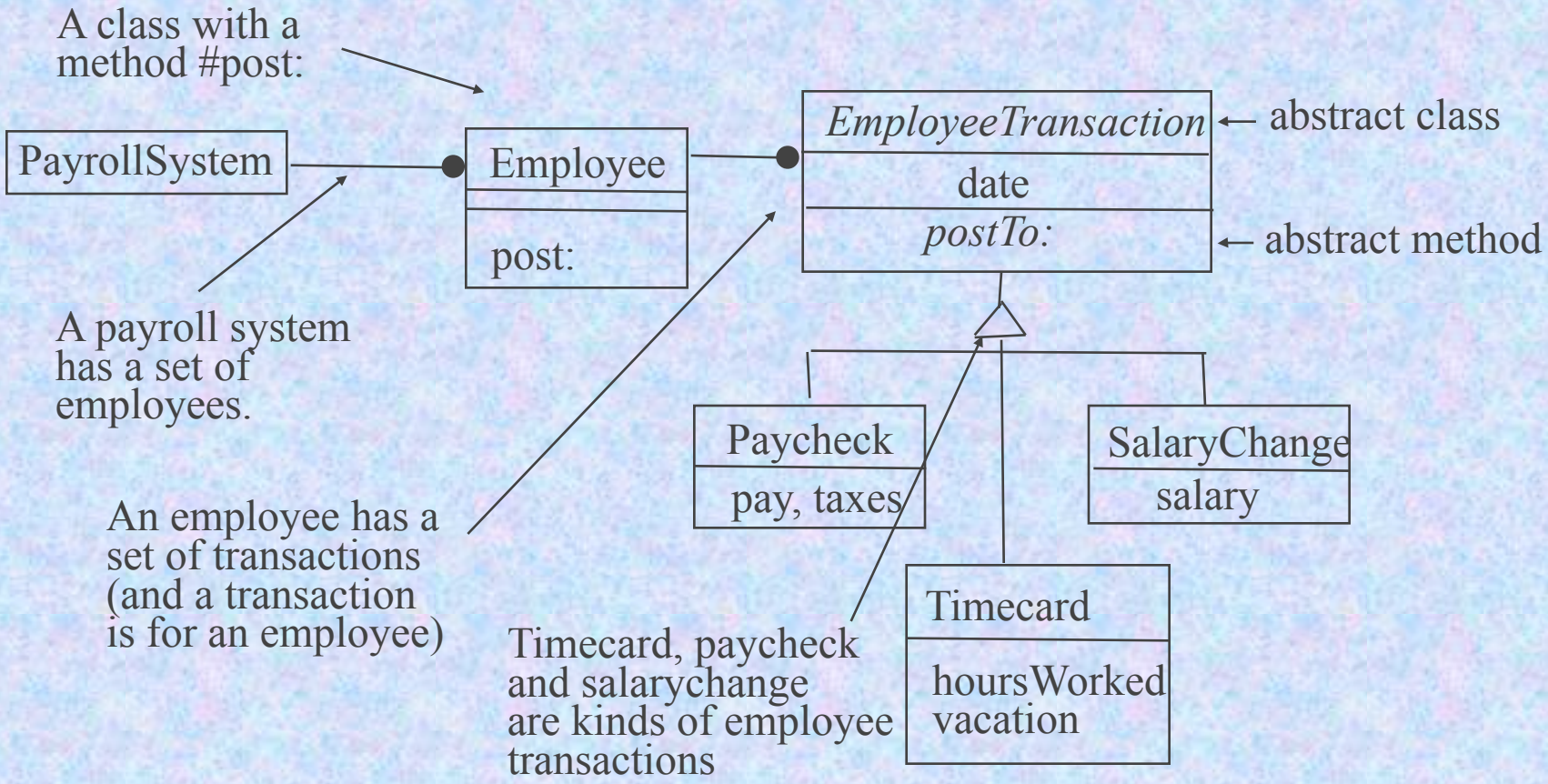
- class with no instances
- designed as a template for subclasses, not for instances.
- defines standard interface

A Payroll System

PayrollSystem keeps track of employees.

Employee knows salary, how much has been earned, how much has been payed, how much taxes have been withheld. Employee keeps an audit trail of all the transactions that have occured.

Object Model for Payroll



Employee

```
Object subclass: #Employee
    instanceVariableNames: 'name transactions
    salary earned paid withholding
    accruedVacation taxRule '
    classVariableNames: 'HeadTaxRate
    MarriedSeparateTaxRate MarriedTaxRate
    SingleTaxRate '
    poolDictionaries: ''
    category: 'Employee'
```

Employee Comment

I represent an employee in a payroll system. Thus, I keep track of the number of hours worked, the wages paid, the vacation accrued, the taxes withheld, and so on. My values change only when transactions are posted to me. Transactions are subclasses of EmployeeTransaction.

(continued)

Variables

name <String>

transactions <Collection of Transactions, sorted by date posted>

salary <Number>

...

taxRule <TaxRule>

Using Employee

```
| ralph |
ralph := Employee new name: 'Ralph Johnson'.
ralph changeSalaryFor: (Date newDay: 5 year:
    1996) to: 20.
ralph postTimeCardFor: (Date newDay: 10 year:
    1996) hoursWorked: 40
        vacation: 0.
```

Initialize

Change

(continued)

...

```
Transcript show: (ralph paidAt: (Date
   newDay: 11 year: 1996)) printString
```

Initializing an Employee

```
named: aString
    name := aString.
    transactions := OrderedCollection new.
    salary := 0.
    ...
    taxRule := SingleTaxRate
```

Complete Creation Method

Employee has the class method

```
named: aString
    ^self new named: aString
```

Timecard

Timecard is a subclass of
EmployeeTransaction

with instance variables 'hoursWorked' and
'hoursVacation '

Initializing a Timecard

```
date: aDate employee: anEmployee worked: hoursW vacation: hoursV
    date := aDate.
    employee := anEmployee.
    hoursWorked := hoursW.
    hoursVacation := hoursV
```

Complete Creation Method

Timecard has class method

date: aDate employee: anEmployee
worked: hoursW vacation: hoursV

^self new date: aDate employee: anEmployee worked: hoursW vacation: hoursV

Using a Timecard

Employee has method:

postTimeCardFor: aDate hoursWorked: hoursW vacation: hoursV

```
self postTransaction: (Timecard date: aDate
employee: self worked: hoursW vacation:
hoursV)
```

Initialization Patterns

There should be one or more methods that initialize object. They should be in protocol "initialize-release".

There should be one or more class methods that create initialized objects. They should be in protocol "instance creation".

Processing a Transaction

In Employee:

postTransaction: aTransaction

```
aTransaction postTo: self.
transactions add: aTransaction
```

Processing a Timecard

```
postTo: anEmployee
    | money  overtime nonovertime |
    overtime := (hoursWorked - 40) max: 0.
    nonovertime := hoursWorked min: 40.
    money := (overtime * 1.5 + nonovertime) *
    anEmployee salary.
    anEmployee incrementEarned:  money.
```

Processing a Timecard

salary

```
^salary
```

incrementEarned: anAmount

```
earned := earned + anAmount
```

Employee Action Protocol

All changes to an Employee should be recorded as transactions.

postTimeCardFor: aDate hoursWorked: hoursW vacation: hoursV

```
self postTransaction:
        (Timecard date: aDate employee: self
            worked: hoursW vacation: hoursV)
```