Programming Language Concepts
July 2, 2003

Michael Haupt

Darmstadt University of Technology
Software Technology Group
haupt@informatik.tu-darmstadt.de

Agenda

- Smalltalk
  - history outline, introduction to the language
  - error handling, run-time library, dependency
  - programming environment
  - the meta level
- Self
  - history outline
  - brief introduction to the language
  - world structure and meta-level

"Real" Object Orientation: Smalltalk

"When I invented the term 'object-oriented'
I did not have C++ in mind."

-- Alan Kay

Smalltalk's History

- 1968: SIMULA
  - first "object-oriented" language
- 1971: Dynabook (Alan Kay)
  - easy-to-use computer for anyone
  - bitmapped display, pointing device, external storage
  - programming ~ creating simulations
  - Smalltalk-71
- 1973: Xerox Alto computer
  - running Smalltalk (implemented in BASIC)
  - iconic characters, turtle graphics (LOGO)
  - classes (no hierarchies), instances, **`self`**

Smalltalk's History

- Smalltalk-80
  - 2 goals: hardware and community
  - language standard
  - massively adopts MVC
  - forerunner of today's GUIs
- Meta-circular implementation
  - compiler implemented in Smalltalk
  - source code is always there
  - virtual machine vs. virtual image

Smalltalk's History

- 1974
  - performance improvements
  - first metaobjects
  - Alto: virtual memory (96 kB RAM, 2.4 MB hard disk)
- 1976
  - Smalltalk bytecode interpreter in Alto microcode
  - everything is an object
  - class hierarchy, **`super`**
  - code browser, inspector, debugger
- 1978: NoteTaker
  - step towards Dynabook: portable computer
  - cancelled: "No one wants portability."

PL SS03 – Klaus Ostermann / Michael Haupt, {ostermann,haupt}@informatik.tu-darmstadt.de

Smalltalk – the Language

- "Pure" object-oriented language
  - absolutely everything is an object
- Features
  - single inheritance
  - untyped
  - powerful library and meta-level
  - strict class hierarchy (superclass: **Object**)

Smalltalk – the Language

- Lexical details
  - comments: double quotes
  - characters: **$<char>**
  - strings: single quotes
  - symbols: **#<sym>**
  - arrays: **#(<elem> <elem> ...)**
- Operators
  - assignment **:=**
  - equality **=**
  - identity **==**

```
"this is a comment"
$x
'this is a string'
#thisIsASymbol
#('this' 'is' 'an' 'array' 'with' 7 'elements')
```

Keywords and Messages

- Keywords are pervasive

C++ / Java:

```
Transformation t;
float a;
Vector v;
t->rotate(a,v); // for C++
t.rotate(a,v); // for Java
```

Smalltalk:

```
| t a v |
| aTransformation angle aVector |
```

~~t . rotate (a , v) ;~~

~~t rotate by: a around: v;~~

`t rotateBy: a around: v` This is Smalltalk!

Keywords and Messages

- Rotation method source code

```
rotateBy: angle around: aVector
  | result |
  result := do some computations.
  ^result
```

- Alternatives:
  - **`rotateAround: aVector by: angle`**
  - **`rotate: angle and: aVector`** (bad style)
  - keywords tell the order of parameters

Keywords and Messages

- Methods (messages)
  - general structure

```
keyword1: param1 keyword2: param2 ...
  | local variables |
  instructions
```

  - instructions end with `.` (except for last)
  - immediate return with **^<value>**
  - message's selector is **keyword1:keyword2:...**

Keywords and Messages

- Unary messages
  - no parameters
  - for example: **someObject clone**
  - definition:

```
messageName
  | local variables |
  instructions
```

- Binary operators
  - there are no unary operators!
  - keywords: special characters like **+ - * / ,**
  - exceptions: e.g., **. ;**

Keywords and Messages

- Message sending semantics
  - messages are sent to objects
  - no implicit sending to **`self`**
  - first element in an expression is always an object
  - result of an expression is an object, too
  - evaluation: *left to right*
- Arithmetics
  - numbers are objects, even literals!
  - expression **`4 sqrt`** results in **`2.0`**
  - expression **`1 + 2 * 3`** results in **`9`** (yes, indeed)
  - expression **`1 + (2 * 3)`** results in **`7`**

Keywords and Messages

- Cascading
  - send multiple messages to a single object
  - instead of

```
| p |
p := Client new.
p name: 'Jack'.
p age: 32.
p address: 'Earth'
```

  - use

```
| p |
p := Client new.
p name: 'Jack'; age: 32; address: 'Earth'
```

  - or even

```
| p |
p := (Client new name: 'Jack'; age: '32'; address: 'Earth')
```

PL SS03 – Klaus Ostermann / Michael Haupt, {ostermann,haupt}@informatik.tu-darmstadt.de

Conditions

- Conditions
  - there are actually no control structures
  - all "control structures" are messages

```
if(a < 0 || (b >= c && b <= d))
  a = -c;
```

```
a negative | (b between: c and: d) ifTrue: [ a := c negated ]
```

- Messages that can be sent to Boolean objects
  - **`ifTrue:`**, **`ifFalse:`**
  - **`ifTrue:ifFalse:`**, **`ifFalse:ifTrue:`**
  - conditional functionality encapsulated in *blocks*

Conditions

- How are conditionals implemented?
  - class **`Boolean`** is abstract
  - classes **`True`** and **`False`** inherit from **`Boolean`**
  - each has one instance: **`true`** resp. **`false`**
  - implementation of **`ifTrue:`** in class **`True`**

```
ifTrue: alternativeBlock
  ^alternativeBlock value
```

  - implementation of **`ifFalse:`** in class **`True`**

```
ifFalse: alternativeBlock
  ^nil
```

Blocks

- Blocks
  - nameless functions (closures, lambda expressions)

    `a = int f(x) { return x + 1; }`   `a := [ :x | x + 1 ]`
  - evaluate blocks by sending them **`value`** messages

    `b := a value: 41`   b will contain 42 after that
  - possible: **`value`**, **`value:`**, **`value:value:`**, …
  - return value is value of last expression in block
- Scope and returning
  - in a block, **`self`** is bound to the surrounding object
  - ^ returns from the defining method

Blocks

- A closer look at returning
  - ^ returns from the *defining method* (rather: *context*)
  - the following will not work

```
getBlock
  ^ [ ^42 ]
```

```
useBlock
  self getBlock value
```

- Does it make sense?
  - blocks are frequently passed to methods
  - returns should not terminate the target

```
divide: a by: b
  b = 0 ifTrue: [ ^'error' ] ifFalse: [ ^a/b ]
```

  - not **`ifXXX:`** should be terminated, but **`divide:by:`**

Loops

- Counting loops
  - do something 100 times

```
100 timesRepeat: [ Transcript show: 'Hello world.'; cr ]
```

  - advanced "for" loops

```
1 to: 100 do: [ :i | Transcript show: i; cr ]
```

```
1 to: 100 by: 10 do: [ :i | Transcript show: i; cr ]
```

```
100 to: 1 by: -1 do: [ ... ]
```

```
0.5 to: 7.3 by: 1.1 do: [ ... ]
```

- While loops
  - blocks can return **`Boolean`** values
  - messages **`whileTrue:`** and **`whileFalse:`**

```
| a |
a := 100 atRandom.
[ a = 42 ] whileFalse: [ a := 100 atRandom ]
```


Classes and Objects

- Classes and objects
  - general superclass: **`Object`**
  - there are class and instance variables and methods
- Instantiation
  - "constructors" are class methods, e.g., **`new`**
  - expressive "constructor" names
  - possible instantiation method for a **`Person`** class

```
withName: name age: age address: address
  ^(self new name: name; age: age; address: address)
```

  - usage:

```
| p |
p := Person withName: 'John' age: 32 address: 'Earth'
```

- Visibility
  - all instance variables are private
  - all messages are public
  - instances of the same class cannot access members
- Instance variable access
  - convention: define two messages of the same name
  - instance variable **age**
  - getter message **age**
  - setter message **age:**

```
age
  ^age
```

```
age: newAge
  age := newAge
```


Classes and Objects

- Abstract classes
  - no syntactic representation of abstractness
  - simply call self subclassResponsibility

```
abstractMethod: someParameter
  self subclassResponsibility
```

  - default behaviour: invoke debugger

PL SS03 – Klaus Ostermann / Michael Haupt, {ostermann,haupt}@informatik.tu-darmstadt.de

Error Handling

- Dynamic typing caveat: unknown messages
  - in **`Object`**: message **`doesNotUnderstand:`**
  - invoked implicitly upon illegal message sends
  - override to deal with it

```
doesNotUnderstand: aMessage
  Transcript show: 'Do not send me '; show: aMessage selector
```

  - standard behaviour: invoke debugger

Error Handling

- Exceptions

```
[ x / y ] on: ZeroDivide do: [ :ex | ex resume: 1 ]
```

  - upon a division by zero, return 1
- Signal exceptions
  - superclass of all exceptions: **`Exception`**
  - program errors: **`Error`**
  - notifications that can be ignored: **`Notification`**
  - warnings the user should be aware of: **`Warning`**
  - signal e.g., like this

```
Error signal: 'Something really bad happened.'
```

PL SS03 – Klaus Ostermann / Michael Haupt, {ostermann,haupt}@informatik.tu-darmstadt.de

Error Handling

- Return values

```
[ Error signal ] on: Error do: [ : ex | someOp. 42 ]
```

  - this returns 42 if an error occurs
  - return value of **`on:do:`**
- Resuming
  - resumable exceptions (**`Notification`**, **`Warning`**)

```
[ Warning signal ] on: Warning do: [ :ex | ex resume: 42 ]
```

  - returns 42, but where the **`Warning`** was signalled
  - react to the type of exception

```
[ someException signal ] on: Exception do:
  [ :ex | ex isResumable ifTrue: [ ex resume: 42 ]
                         ifFalse: [ 23 ]]
```


Error Handling

- Exiting handler blocks
  - explicit exit: two semantically identical ways

```
[ Error signal ] on: Error do: [ :ex | 23 ]
[ Error signal ] on: Error do: [ :ex | ex exit: 23 ]
```

- Retrying
  - adjust erroneous values

```
[ ^ x / y ] on: ZeroDivide do: [ :ex | y := 0.000001. ex retry ]
```

  - use a fallback

```
[ doSomethingUnsafely ] on: Error do:
  [ :ex | ex retryUsing: [ doItSafely ]]
```

Error Handling

- Cleaning up
  - place a lock on an external file
  - locks must be released even if exceptions occur

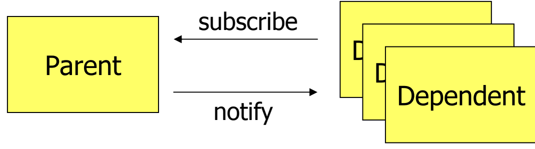
```
doLocked: aBlock
  self lock.
  ^aBlock ensure: [ self unlock ]
```

- Conditionally cleaning up
  - perform cleaning up only if an exception occurred

```
someBlock ifCurtailed: [ cleanup ]
```


Dependency

- Observer design pattern
  - clients subscribe for notifications on subject updates
  - Smalltalk: clients = dependents, subject = parent



- Dependency support
  - standard, implemented in `Object`
  - subjects (i.e., all objects) implement notification

Dependency

- Notification messages
  - notify unspecified: **`changed`**
  - notify of specific update: **`changed:`**
- Notification for **`age`** instance variable
  - method **`age:`** in class **`Person`**

```
age: newAge
  | old |
  old := self age.
  age := newAge.
  self changed:
    (Dictionary new
      at: #parent put: self;
      at: #member put: #age;
      at: #oldValue put: old;
      yourself)
```


Dependency

- Becoming a dependent
  - send **`addDependent:`** to the parent

```
somePerson addDependent: self
```

  - implement **`update`** or **`update:`**

```
update: param
  Transcript show: 'Person was updated.'; show param; cr
```

- Revoking dependency
  - send **`removeDependent:`** to the parent

```
somePerson removeDependent: self
```

Multitasking

- Create a process by sending **`fork`** to a block

```
[ 10 timesRepeat:
  [ Transcript show: '1'. (Delay forMilliseconds: 1) wait ]
] fork.
[ 10 timesRepeat:
  [ Transcript show: '2'. (Delay forMilliseconds: 1) wait ]
] fork
```

  – outputs **`12121212121212121212`**
  – alternative: tell the scheduler to swap explicitly

```
Processor yield
```

- Priorities
  – range from 1 (high) to 100 (low); 50 is default
  – assign priority by sending **`forkAt:`**

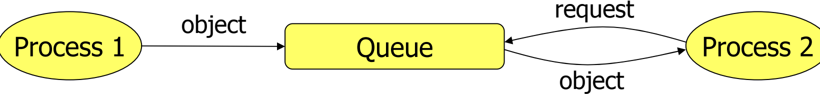
Multitasking

- Suspending and resuming processes
  – suspending: send **`suspend`**
  – send **`resume`** to sleeping processes for wakeup

```
| p |
p := [ [ Transcript show: 'Hello world! ' ] repeat ] newProcess.
p resume
```

```
| p |
p := [ 1 to: 10 do:
        [ :i | Transcript cr; show: Time now.
                i = 5 ifTrue: [ Processor activeProcess suspend ]
        ] ] fork.
(Delay forSeconds: 5) wait.
p resume
```

- Process communication: queues



- Standard FIFO queues: **`SharedQueue`**
  - insert an object with **`nextPut:`**
  - request the next object with **`next`**

```
| q read |
q := SharedQueue new.
read := [ [ Transcript show: ' R', q next printString ] repeat ] fork.
[ 1 to: 5 do:
  [ :i | Transcript show: ' W', i printString. q nextPut: i. Processor yield ]
] fork.
(Delay forSeconds: 5) wait.
read terminate
```


- Thread safety with shared data
  - use **`Semaphore`** class for critical sections

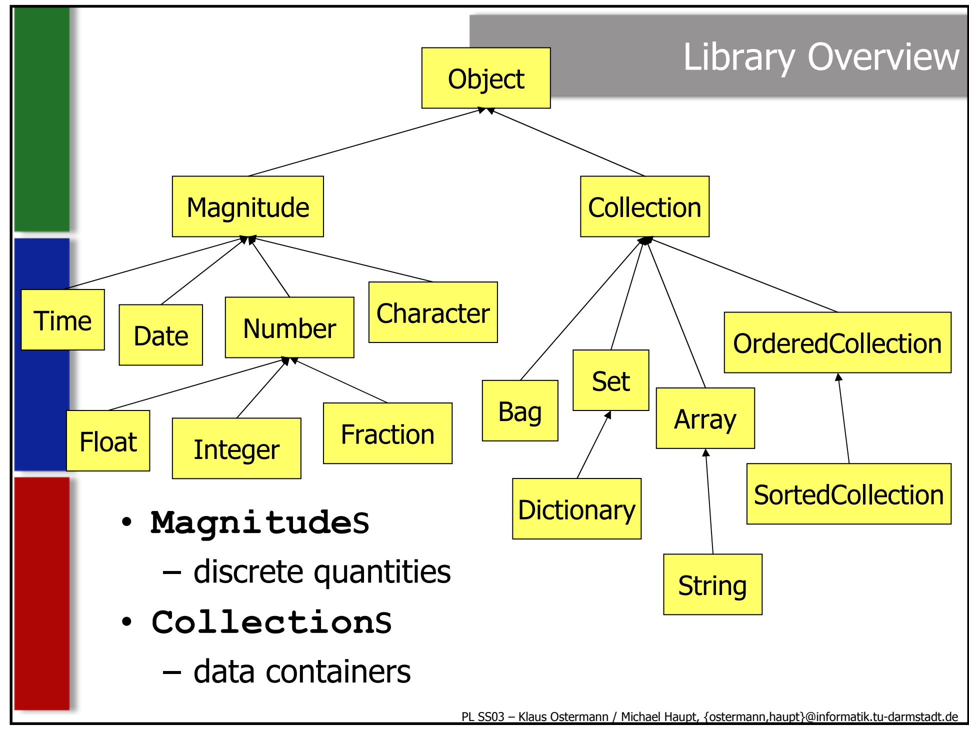
```
| sem |
sem := Semaphore forMutualExclusion.
sem critical: [ criticalFunctionality ]
```

  - other messages: **`wait`**, **`signal`** (as expected)

```
| sem |
sem := Semaphore forMutualExclusion.
sem wait.
criticalFunctionality.
sem signal
```

Library Overview

- Protocols define standard library functionality
  - sets of related messages an object can support
  - no formal language part, embedded specification
- Protocol
  - named specification for a group of messages
  - for each selector, functionality is specified precisely
- Example: `<Object>` protocol
  - specifies messages all Smalltalk objects support
  - mostly comparing and testing
  - some for message sends and string representations



Numeric Protocols

- Protocol `<Number>`
  - arithmetic

    `+ - * / abs ceiling floor sqrt ...`
  - conversion

    `asInteger asFloat asFraction ...`
  - iteration

    `to:do: to:by:do: ...`
- Protocol `<Float>`

  `cos sin tan fractionPart integerPart ln log: ...`
- Protocol `<Integer>`

  `bitAnd: bitOr: bitAt: clearBit: even factorial ...`

Collections

- Collections
  - powerful data structures for many uses
- Unordered collections
  - **Bag**: variable size, may have dups, access by value
  - **Set**: variable size, no dups, access by value
  - **Dictionary**: variable size, key-value storage
- Ordered collections
  - access by numeric indexes
  - **OrderedCollection**: variable size
  - **Array**: fixed size
  - **SortedCollection**: variable size
  - **String**: fixed size, character elements

- Collections example

```
| a b |
a := OrderedCollection new
  add: #green;
  add: 'red';
  yourself.
b := Dictionary new
  at: #red put: #rouge;
  at: 'green' put: #($v $e $r $t);
  yourself
```

  - both **`add:`** and **`at:put:`** return last parameter
  - send **`yourself`** to an **`Object`** to get it itself
  - contained data can be of arbitrary type

- Protocol **`<Collection>`**
  - defines various powerful standard mechanisms
  - cross-conversion

```
asArray asBag asSet asSortedCollection ...
```

  - testing

```
isEmpty size includes: ...
```

  - iteration
- Accessing
  - different protocols define access strategies
  - e.g., **`at:put:`** makes no sense with **`Sets`**

- Basic iteration: **`do:`**
  - perform functionality for each element

```
aCollection do: [ :item | Transcript show: item ]
```

  - outputs all elements in a collection
- Selection: **`select:`**, **`reject:`**
  - return a collection of matching elements

```
underage := persons select: [ :p | p age < 21 ]
```

  - get all persons younger than 21 years
  - with **`reject:`**, the block must evaluate to **`false`**

- Collection: **`collect:`**
  - return a collection of computation results

```
names := persons collect: [ :p | p name ]
```

  - get all persons' names
- Accumulate: **`inject:into:`**
  - starting with a value, iterate over a collection

```
avgAge := persons inject: 0 into:
  [ :subtotal :p | subtotal := subtotal + p age ] / persons size
```

  - compute the average age

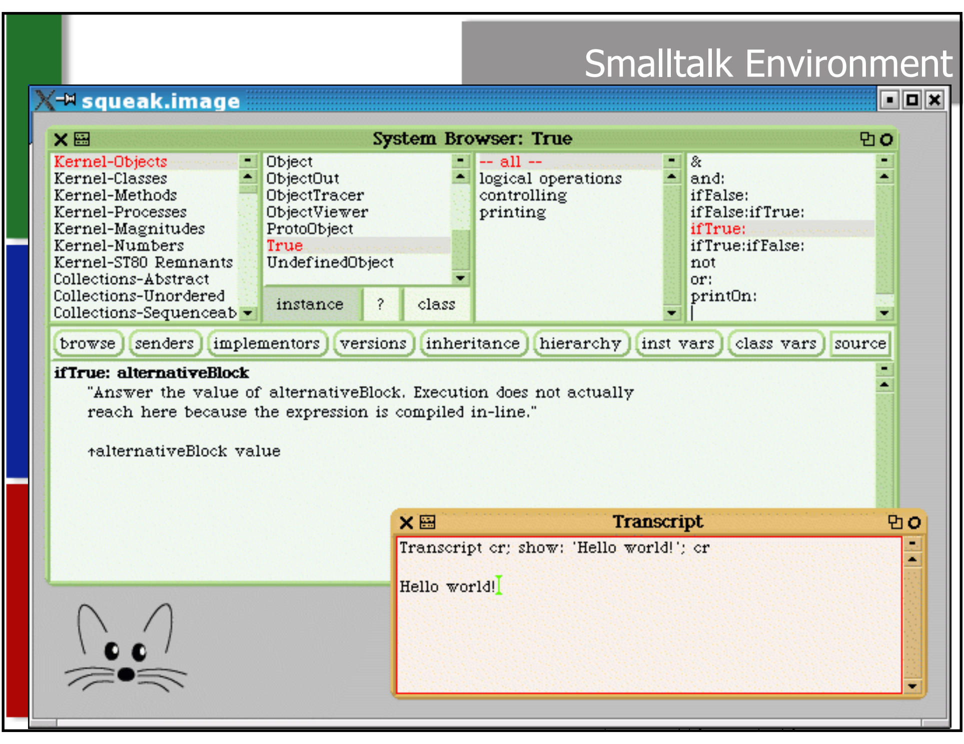
PL SS03 – Klaus Ostermann / Michael Haupt, {ostermann,haupt}@informatik.tu-darmstadt.de

Smalltalk Environment

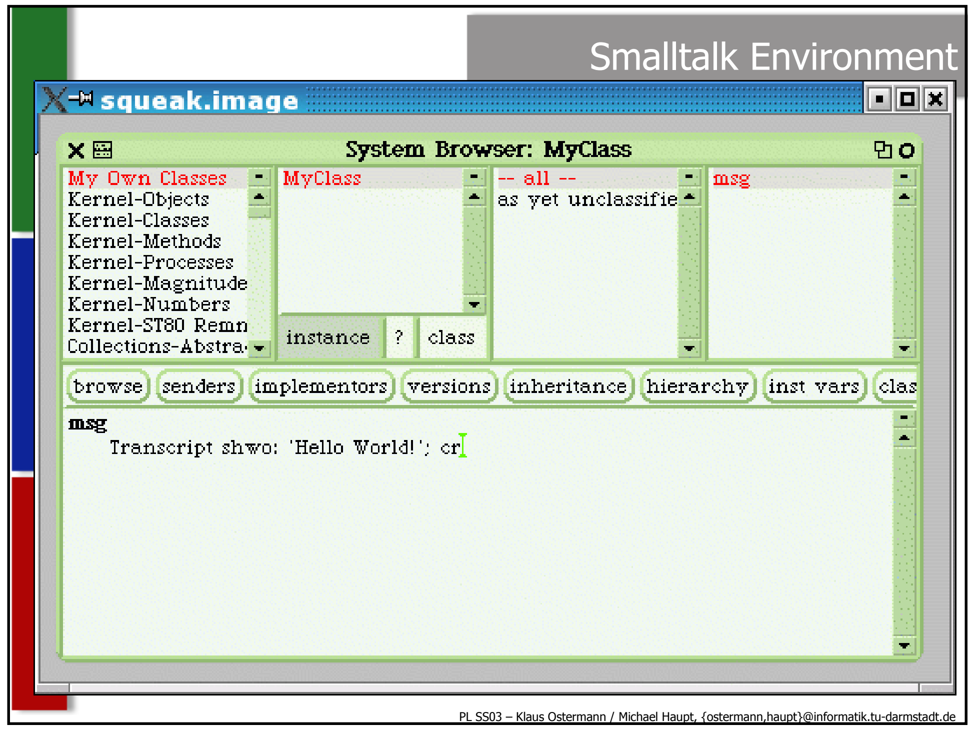
- Smalltalk implementations (free)
  - Squeak (used here, open source)
    http://www.squeak.org/
  - Smalltalk/X: free for non-commercial use
    http://www.exept.de/
  - Cincom Smalltalk: free for non-commercial use
    http://www.parcplace.com/
  - Strongtalk (*typed* Smalltalk)
    http://www.cs.ucsb.edu/projects/strongtalk/
    pages/index.html

Smalltalk Environment

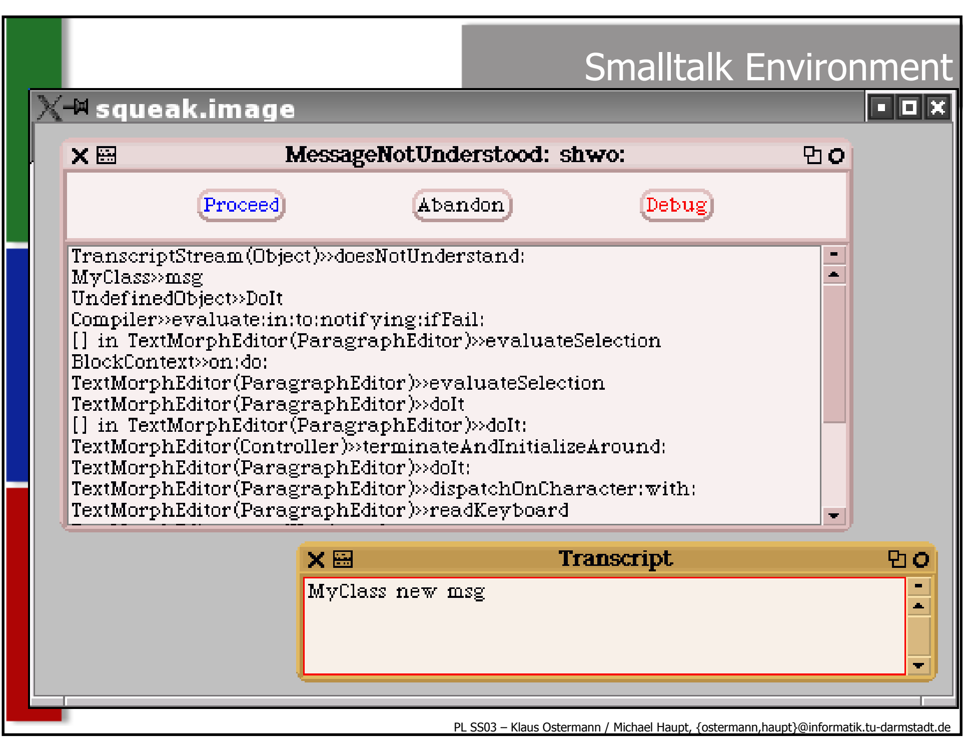
- Environment
  - virtual machine (usually with JIT)
  - image = snapshot
  - export applications as image, deliver with VM binary
- Benefits
  - environment is IDE and run-time all the same
  - powerful debugging facilities
  - on-the-fly implementation modification

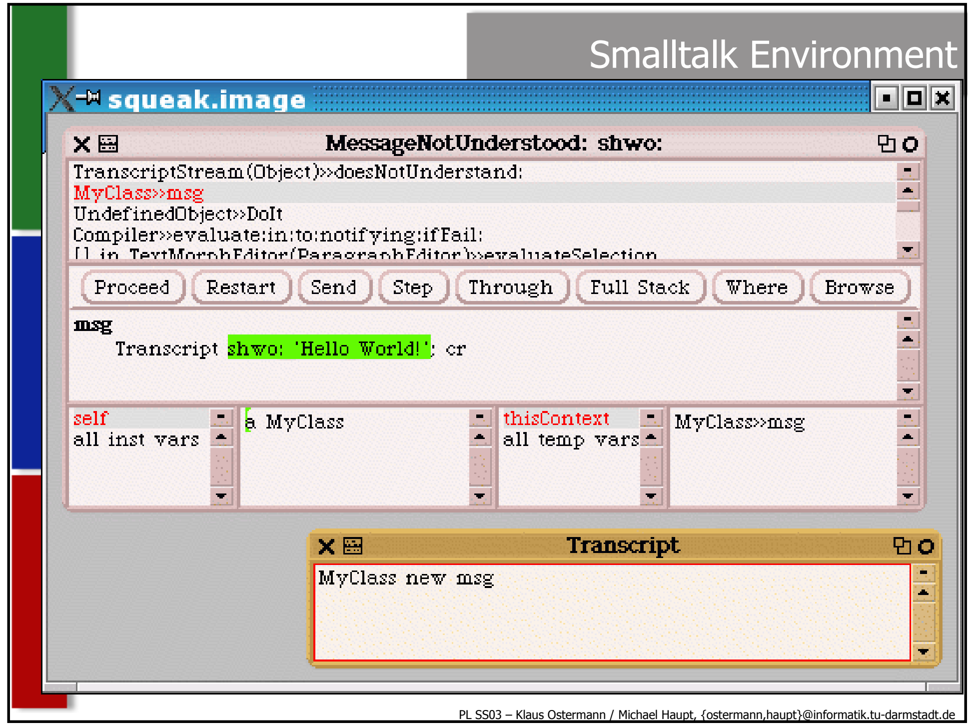


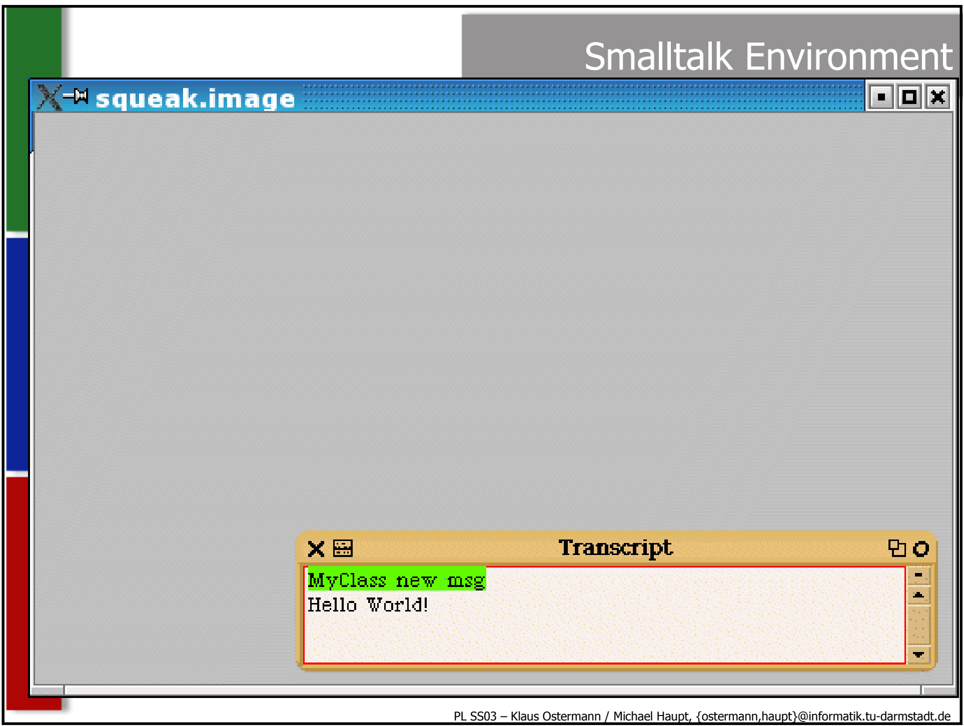
Smalltalk Environment
squeak.image
System Browser: True
Kernel-Objects
Kernel-Classes
Kernel-Methods
Kernel-Processes
Kernel-Magnitudes
Kernel-Numbers
Kernel-ST80 Remnants
Collections-Abstract
Collections-Unordered
Collections-Sequenceab
Object
ObjectOut
ObjectTracer
ObjectViewer
ProtoObject
True
UndefinedObject
instance
?
class
-- all --
logical operations
controlling
printing
&
and:
ifFalse:
ifFalse:ifTrue:
ifTrue:
ifTrue:ifFalse:
not
or:
printOn:
browse
senders
implementors
versions
inheritance
hierarchy
inst vars
class vars
source
ifTrue: alternativeBlock
"Answer the value of alternativeBlock. Execution does not actually
reach here because the expression is compiled in-line."
↑alternativeBlock value
Transcript
Transcript cr; show: 'Hello world!'; cr
Hello world!



Smalltalk Environment
squeak.image
System Browser: MyClass
My Own Classes
Kernel-Objects
Kernel-Classes
Kernel-Methods
Kernel-Processes
Kernel-Magnitude
Kernel-Numbers
Kernel-ST80 Remn
Collections-Abstra
MyClass
instance
?
class
-- all --
as yet unclassifie
msg
browse
senders
implementors
versions
inheritance
hierarchy
inst vars
clas
msg
Transcript shwo: 'Hello World!'; cr

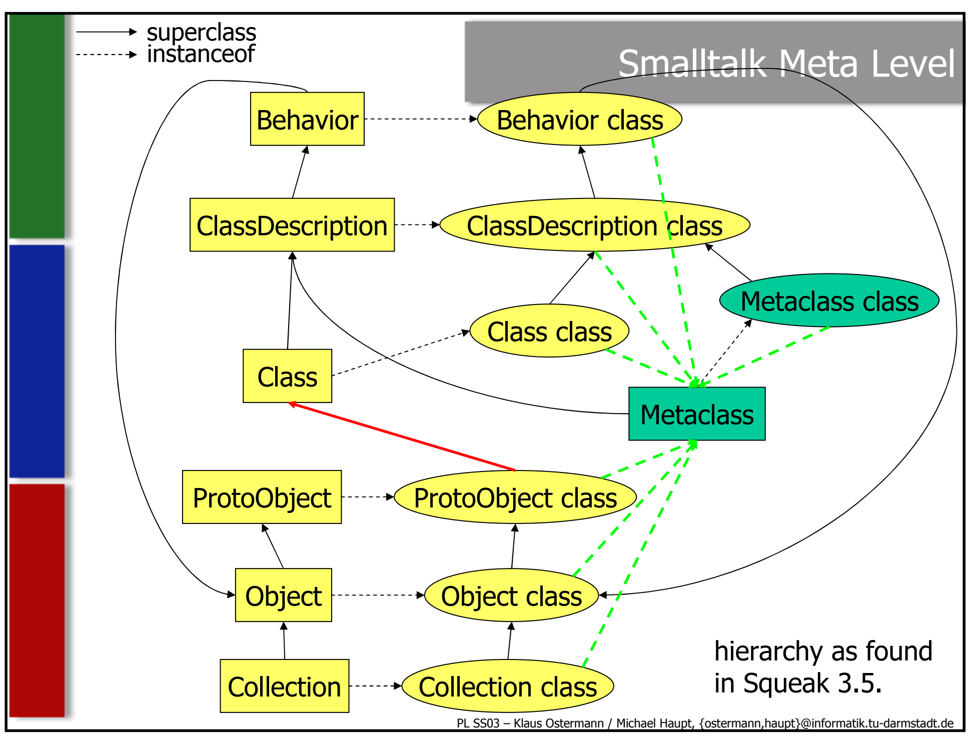

Smalltalk Environment
squeak.image
MessageNotUnderstood: shwo:
Proceed
Abandon
Debug
TranscriptStream(Object)>>doesNotUnderstand:
MyClass>>msg
UndefinedObject>>DoIt
Compiler>>evaluate:in:to:notifying:ifFail:
[] in TextMorphEditor(ParagraphEditor)>>evaluateSelection
BlockContext>>on:do:
TextMorphEditor(ParagraphEditor)>>evaluateSelection
TextMorphEditor(ParagraphEditor)>>doIt
[] in TextMorphEditor(ParagraphEditor)>>doIt:
TextMorphEditor(Controller)>>terminateAndInitializeAround:
TextMorphEditor(ParagraphEditor)>>doIt:
TextMorphEditor(ParagraphEditor)>>dispatchOnCharacter:with:
TextMorphEditor(ParagraphEditor)>>readKeyboard
Transcript
MyClass new msg
PL SS03 – Klaus Ostermann / Michael Haupt, {ostermann,haupt}@informatik.tu-darmstadt.de


Smalltalk Environment
squeak.image
MessageNotUnderstood: shwo:
TranscriptStream(Object)>>doesNotUnderstand:
MyClass>>msg
UndefinedObject>>DoIt
Compiler>>evaluate:in:to:notifying:ifFail:
Proceed
Restart
Send
Step
Through
Full Stack
Where
Browse
msg
Transcript shwo: 'Hello World!'; cr
self
all inst vars
a MyClass
thisContext
all temp vars
MyClass>>msg
Transcript
MyClass new msg
PL SS03 – Klaus Ostermann / Michael Haupt, {ostermann,haupt}@informatik.tu-darmstadt.de



Smalltalk Meta Level

- "Everything is an object."
  - consequently, classes are objects, too
  - classes are singleton instances of metaclasses
  - metaclasses are instances of **`Metaclass`**
- Subsequent examples
  - class **`MyClass`**
  - instance method **`insmsg`**, class method **`clsmsg`**
  - instance variable **`insvar`**, class variable **`Clsvar`**
- Hold tight...



Smalltalk Meta Level

- Classes
  - methods in **`methodDict`** (**`Dictionary`**)
  - inst. var. names in **`instanceVariables`** (**`String`**)
  - class variables in **`classPool`** (**`Dictionary`**)
  - subclass names in **`subclass`** (**`Array`**)
  - superclass in **`superclass`**
  - class methods are instance methods here!

Smalltalk Meta Level

- Example **`MyClass`**
- Method dictionary
  - one entry, **`#insmsg`** -> a **`CompiledMethod`**
- Instance variables: **`'insvar'`**
- Class variables
  - one entry, **`#Clsvar`** -> **`nil`**
- Subclass names: **`nil`**
- Superclass: **`Object`**
- Methods
  - apart from **`Metaclass`** instance methods
  - class method of **`MyClass`**: **`clsmsg`**

Smalltalk Meta Level

- Example **`MyClass class`**
- Method dictionary
  - one entry, **`#clsmsg`** -> a **`CompiledMethod`**
- Instance variables: **`nil`**
- Class variables: none
- Subclass names: none
- Superclass: **`Object class`**
- Methods: none special

Smalltalk Meta Level

- Methods
  - methods are objects, too
  - in metaclasses: instances of **`CompiledMethod`**
  - show the source code of a method

    (MyClass methodDict at: #insmsg) decompileString
- Reflection
  - methods can be replaced with new implementations
  - all classes are modifiable
  - full access to compiler

"Hard Core" Object Orientation: Self

Self's History

- Another Xerox PARC outcome
  - 1987: first design and implementation
  - further development at Stanford University
  - moved to Sun as research project
  - wealth of research results for JIT technology
- Still "alive"
  - latest version: Self 4.2 (June 2003)
    http://research.sun.com/self/
    release for MacOS and Solaris (PowerPC)
  - Self4Linux (and CygWin)
    http://www.gliebe.de/self/

Self – the Language

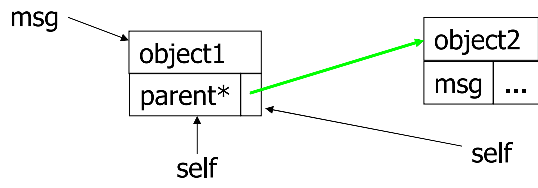
- Smalltalk inspiration
  - syntax is very similar
  - virtual machine and image
- Features
  - everything is an object
  - *absolutely no classes at all*
  - prototype-based language
  - delegation as inherent language mechanism
  - multiple "inheritance"

Self – the Language

- Objects
  - consist of slots
  - slots contain either data or behaviour
  - of course, slots are themselves objects
- Messages
  - sending messages = sending slot names
- Data slots
  - store a value named "age"
  - pair of slots; reading: **`age`**, writing: **`age:`**
- Method slots
  - contain code

Self – the Language

- Inheritance
  - parent slots: slot name followed by **`*`**
  - point to parent objects
  - more than one parent possible
- Delegation
  - messages not understood are *delegated* to parents
  - **`self`** remains bound to original target



Self – the Language

- Sending messages
  - like in Smalltalk: `<object> <message>`
  - implicit send to **`self`**
- Explicit resends
  - like **`super`** calls
  - undirected resend: **`resend.msg`**
  - directed resend through parent slot: **`parent.msg`**
  - used to resolve name conflicts

Self – the Language

- Unary and binary messages
  - unary: like in Smalltalk
  - binary: sequence of special characters
  - e.g., `<-> @ + - * /` ...
- Keyword messages
  - first keyword lower case, subsequent upper case

    to:Do:By: min:Max: ...
  - primitives: begin with underscore and capital letter

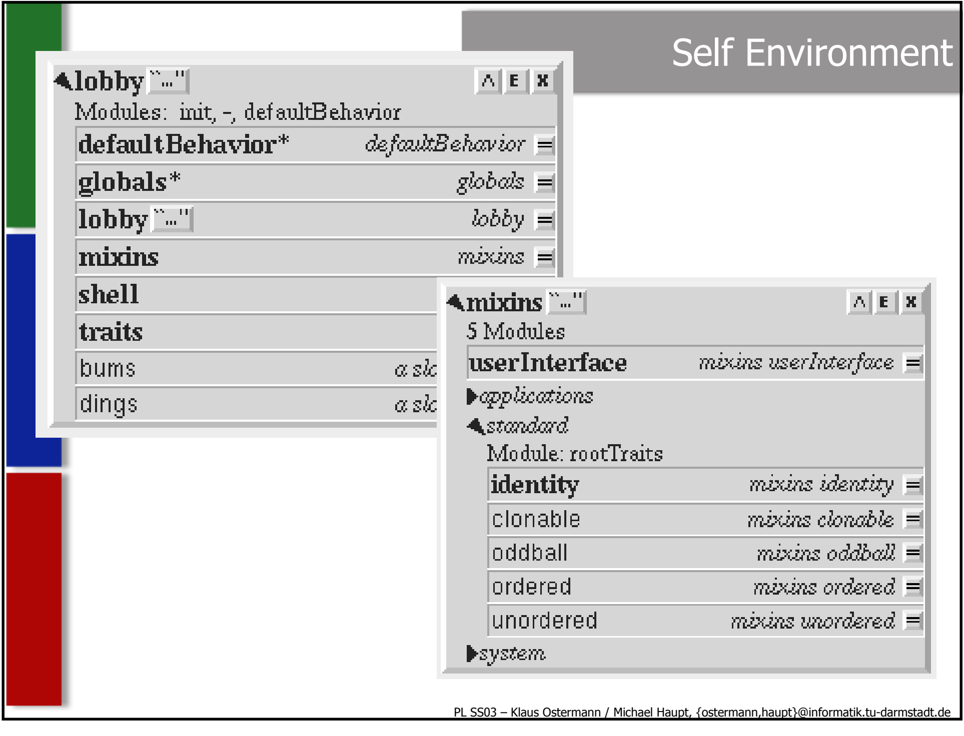
    _AddSlotsIfAbsent: _Define: ...

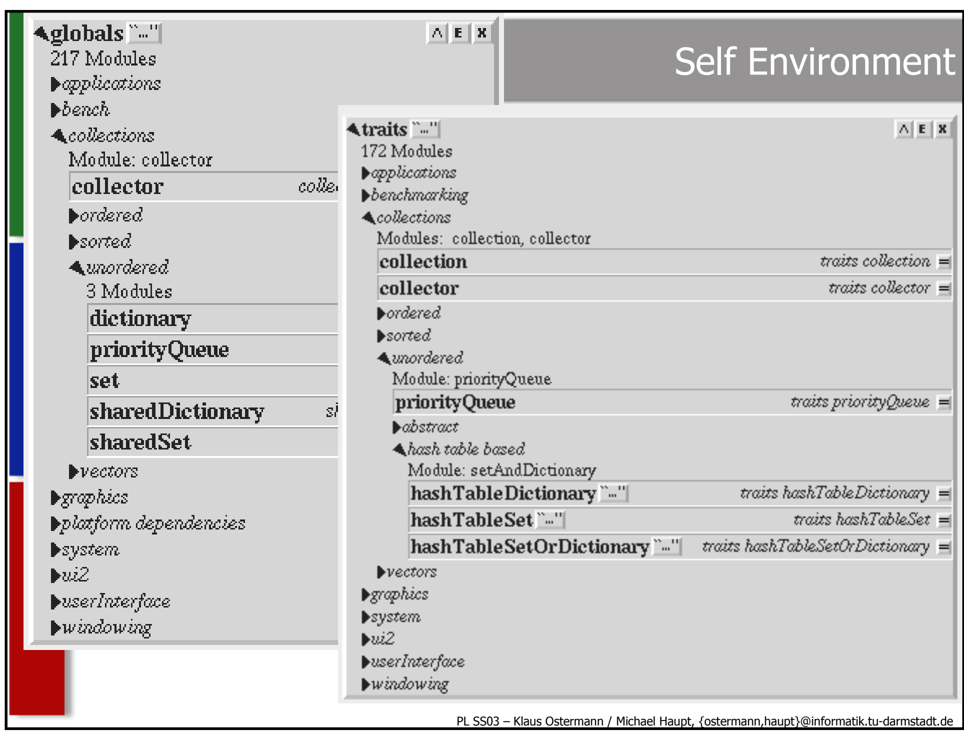
PL SS03 – Klaus Ostermann / Michael Haupt, {ostermann,haupt}@informatik.tu-darmstadt.de

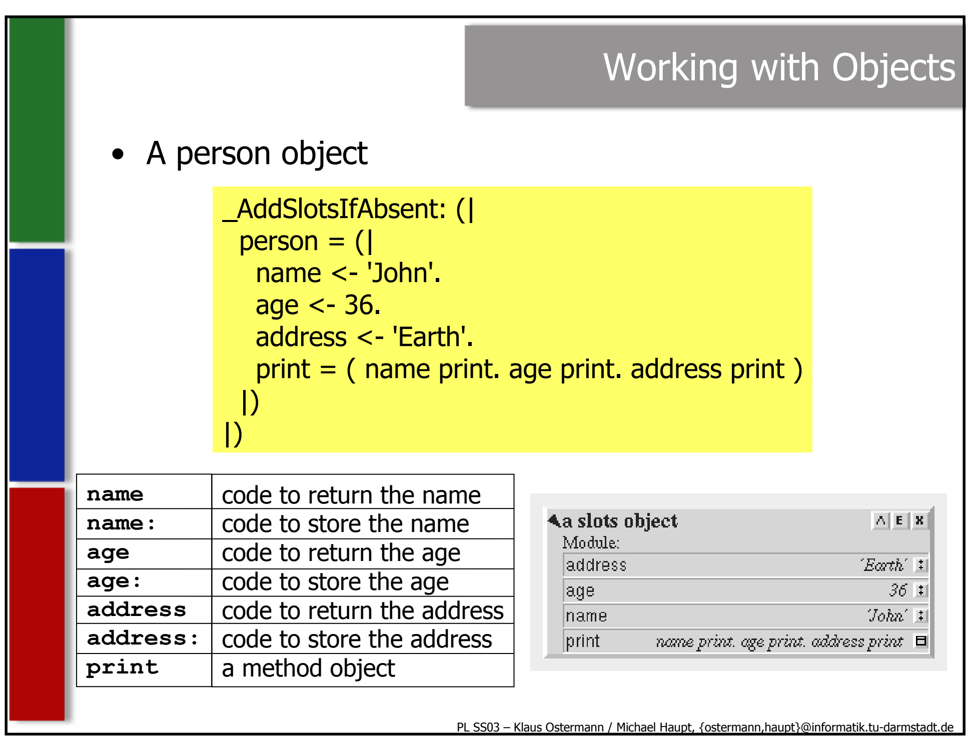
Self Environment

- The `lobby`
  - where Self objects enter the world
  - contains root slots of the global namespace
- Behaviour: `traits`
  - each prototype has an associated traits object
  - traits describe shared behaviour of object kinds
- Prototypes: `globals`
  - blueprints go here
  - also, one-of-a-kind objects ("oddballs")
- More behaviour: `mixins`
  - small parentless bundles of behaviour

Self Environment




Self Environment
globals
217 Modules
applications
bench
collections
Module: collector
collector
ordered
sorted
unordered
3 Modules
dictionary
priorityQueue
set
sharedDictionary
sharedSet
vectors
graphics
platform dependencies
system
ui2
userInterface
windowing
traits
172 Modules
applications
benchmarking
collections
Modules: collection, collector
collection
traits collection
collector
traits collector
ordered
sorted
unordered
Module: priorityQueue
priorityQueue
traits priorityQueue
abstract
hash table based
Module: setAndDictionary
hashTableDictionary
traits hashTableDictionary
hashTableSet
traits hashTableSet
hashTableSetOrDictionary
traits hashTableSetOrDictionary
vectors
graphics
system
ui2
userInterface
windowing
PL SS03 – Klaus Ostermann / Michael Haupt, {ostermann,haupt}@informatik.tu-darmstadt.de


Working with Objects
A person object
_AddSlotsIfAbsent: (|
person = (|
name <- 'John'.
age <- 36.
address <- 'Earth'.
print = (name print. age print. address print)
|)
|)
name
code to return the name
name:
code to store the name
age
code to return the age
age:
code to store the age
address
code to return the address
address:
code to store the address
print
a method object
a slots object
Module:
address
'Earth'
age
36
name
'John'
print
name print. age print. address print
PL SS03 – Klaus Ostermann / Michael Haupt, {ostermann,haupt}@informatik.tu-darmstadt.de

Working with Objects

- There may be more than one person...
  - OO way: create a Person class
  - Self way: create a Person traits and prototype
- Traits
  - define functionality all persons share
  - here: print functionality
- Prototype
  - a blueprint for a person (data)
  - slots for name, age, and address

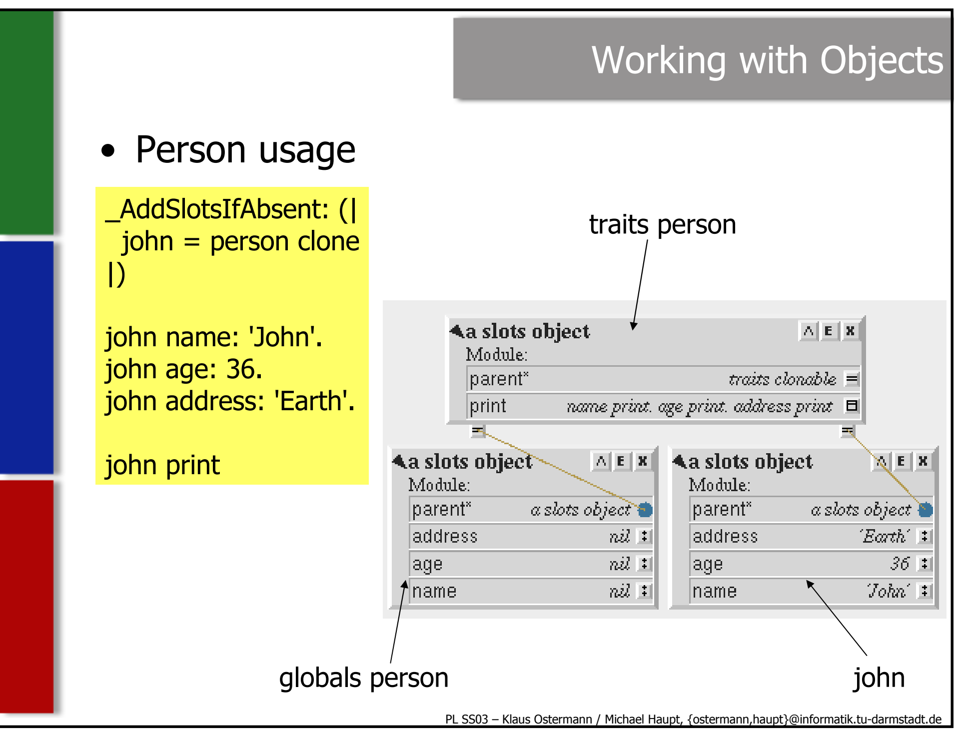
Working with Objects

- Person traits

```
traits _AddSlotsIfAbsent: (|
  person = (|
    parent* = traits clonable.
    print = ( name print. age print. address print )
  |)
|)
```

- Person prototype

```
globals _AddSlotsIfAbsent: (|
  person = (|
    parent* = traits person.
    name <- nil. age <- nil. address <- nil
  |)
|)
```



Conditionals and Blocks

- Conditions look like in Smalltalk
  - yet small syntactical differences
  - send **`ifTrue:`**, **`ifFalse:`**
  - send **`ifTrue:False:`**, **`ifFalse:True:`**
- Self blocks
  - quite like in Smalltalk, small difference in syntax

```
[ | :arg1. :arg2 | arg1 + arg2 ] value: 42 With: 23
```

Loops

- Loops
  - send according messages to blocks
  - endless loop

```
[ 'Hello world! ' print ] loop
```

  - while loops with **`whileTrue:`**, **`whileFalse:`**
  - until loops with **`untilTrue:`**, **`untilFalse:`**
- Exiting loops and blocks

```
[ | :exit | ... cond ifTrue: exit ... ] loopExit
[ | :exit | ... cond ifTrue: exit ... ] exit
[ | :exit | ... cond ifTrue: [ exit value: 42 ] ... ] exitValue
```


Loops and More

- More loops
  - common behaviour to integers in **`traits integer`**
  - e.g., **`to:Do:`**, **`to:By:Do:`**, **`downTo:Do:`**, ...

```
10 downTo: 0 Do: [ | :i | i print ]
```

- Library
  - extensive, like in Smalltalk
  - collections, numbers, processes, GUI, ...
- Meta-level
  - full access to all objects (including system objects)
  - 100 % reflective environment