Lecture 29-30
Advanced Concepts and Structures

prof. Marco Bellia, Dip. Informatica, Università di Pisa

May 13-16, 2014

Advanced Concepts and Structures

- OO Fundamentals: Constructs in FJ
- Syntactic Domains for Classes and Objects
- Semantic Domains: Environments as First Class Values
- Semantic Functions: Classes, Objects, Fields, Methods, Constructors
- Implementation Aspects
- Additional Mechanisms: Generic and of Sub-type Polymorphism
- Additional Mechanisms: Interfaces and Type Constraints
- Additional Mechanisms: Anonymous Classes and Higher Order
- Additional Mechanisms: Sub-classes (inner)
- Methodologies: Inheritance and Super-Classes
- Methodologies: Classes and Abstract Data Types
- Methodologies: Code Extension and Code Reuse

OO Fundamentals: Constructs in FJ/1

FJ is a didactic language which extends the procedural paradigm with the OO paradigm (in the form of a, possibly orthogonal, kernel of Java).

- Declarations, **D**, and Commands, **C**, may include typical procedural constructs.
- A Class Collection is a Package: **classes** $\overline{\texttt{Z}}$ **end**
- The Package furnishes a very good way to integrate OO with other programming paradigms (such as those supported by the constructs considered until now, included HOP)
- The Package may be in the scope of the bindings that are introduced by blocks: The Semantics may either integrate or make separated, the paradigms.

Syntax		
D	::=	... | **classes** $\overline{\texttt{Z}}$ **end** | ...
C	::=	... | I := E | Call I () | ...
E	::=	... | **new** I($\overline{\texttt{E}}$) | **this** | E.**super** | E.G[($\overline{\texttt{E}}$)] | ...
Z	::=	**class** I [**extends** N] {$\overline{\texttt{F}}$ K $\overline{\texttt{M}}$}
N	::=	I | **Object**
G	::=	I
F	::=	I = E
K	::=	**cstr** = ($\overline{\texttt{P}}$) C
M	::=	I ($\overline{\texttt{P}}$) C

Legenda
[e] is a meta for an optional occurrence of e

Example

Exercise1. Underline the (syntactic) constructs that belong to the kernel of Java;
Exercise2. Add the category P for programs that integrate procedural and OO paradigms in the way discussed in the last sentence, above.

OO Fundamentals: Constructs in FJ/2

- The option [**extends** N] is considered in the concrete syntax: The abstract syntax may always include it
- In Java, for instance, the omission of any specific super Class, in the concrete code, leads to an explicit use of **extends Object** in the abstract code.
- The Class contains a, possibly empty, sequence of Fields, $\overline{\texttt{F}}$ and of Methods, $\overline{\texttt{M}}$, and one anonymous Constructor, K.

Syntax		
D	::=	... | **classes** $\overline{\texttt{Z}}$ **end** | ...
C	::=	... | I := E | Call I () | ...
E	::=	... | **new** I($\overline{\texttt{E}}$) | **this** | E.**super** | E.G[($\overline{\texttt{E}}$)] | ...
Z	::=	**class** I [**extends** N] {$\overline{\texttt{F}}$ K $\overline{\texttt{M}}$}
N	::=	I | **Object**
G	::=	I
F	::=	I = E
K	::=	**cstr** = ($\overline{\texttt{P}}$) C
M	::=	I ($\overline{\texttt{P}}$) C

OO Fundamentals: Constructs in FJ/3

- **super** cannot be emulated and it is essential for inheritance and code reuse.

Syntax		
D	::=	... | **classes** $\overline{\texttt{Z}}$ **end** | ...
C	::=	... | `I := E` | `Call I ()` | ...
E	::=	... | **new** `I`($\overline{\texttt{E}}$) | **this** | `E.`**super** | `E.G`[($\overline{\texttt{E}}$)] | ...
Z	::=	**class** `I` [**extends** `N`] {$\overline{\texttt{F}}$ `K` $\overline{\texttt{M}}$}
N	::=	`I` | **Object**
G	::=	`I`
F	::=	`I` = `E`
K	::=	**cstr** = ($\overline{\texttt{P}}$) `C`
M	::=	`I` ($\overline{\texttt{P}}$) `C`

- In the example below, the Method `m` of class `A` is defined in terms of the Method `m` of the superclass `B`

Example

```
class A extends B {
   ...
   m(..){... this.super.m(...) ...}
   ...
}
```

Syntactic Domains for Classes, Objects

Table 16 – Syntactic Domains

Syntactic Domains

$$\begin{array}{lcl} \texttt{D} & ::= & \ldots \mid \textbf{classes } \overline{\textbf{class } \texttt{I } \textbf{extends } \texttt{N } \{\overline{\texttt{F}}\ \texttt{K}\ \overline{\texttt{M}}\}} \textbf{ end} \mid \ldots \\ \texttt{C} & ::= & \ldots \mid \texttt{I := E} \mid \texttt{Call I ()} \mid \ldots \\ \texttt{E} & ::= & \ldots \mid \textbf{new } \texttt{I}(\overline{\texttt{E}}) \mid \textbf{this} \mid \texttt{E.}\textbf{super} \mid \texttt{E.G}(\overline{\texttt{E}}) \mid \texttt{E.G} \mid \ldots \end{array}$$

Auxiliary Syntactic Domains

$$\begin{array}{lcll} \texttt{N} & ::= & \texttt{I} \mid \textbf{Object} & (\textit{Class names}) \\ \texttt{G} & ::= & \texttt{I} & (\textit{Method and Field names}) \end{array}$$

Semantic Domains: Environments as Values

Table 16.1 – Semantic Domains

Semantic Domains	
$\mathtt{Env}, \rho, \delta \equiv \mathtt{I} \longrightarrow \mathtt{Den}$	(*Environment*)
$\mathtt{Store}, \mathtt{s} \equiv \mathtt{Loc} \longrightarrow \mathtt{Mem}$	(*Store*)
Auxiliary Semantic Domains	
$\mathtt{Val}, \mathtt{v} ::= \ldots + \mathtt{objectV} + \ldots$	(*Computable V.*)
$\mathtt{Den}, \mathtt{d} ::= \ldots + \mathtt{objectD} + \ldots$	(*Denotable V.*)
$\mathtt{Mem}, \mathtt{d} ::= \ldots + \mathtt{objectM} + \ldots$	(*Storable V.*)
Auxiliary Functions	
$\mathtt{O_v} : \mathtt{Env} \longrightarrow \mathtt{Val}$	
$\mathtt{O_m} : \mathtt{Env} \longrightarrow \mathtt{Mem}$	
$\mathtt{O_d} : \mathtt{Env} \longrightarrow \mathtt{Den}$	
$\mathtt{A} : (\mathtt{Val}^{\mathtt{n}} \longrightarrow \mathtt{Store} \longrightarrow (\mathtt{O}_v \times \mathtt{Store})) \longrightarrow \mathtt{Den}$	

- The Objects are environments (obviously, not the same ones used in the semantic functions, but they are very similar to those for ADTs)
- The Objects are Computable, Storable, Denotable Values ($\mathtt{O_v}, \mathtt{O_m}, \mathtt{O_d}$)
- Classes may only introduce Objects: Hence, the semantics of a class is a Denotable value, $\mathtt{A(d)}$, for an Object Construction function, $\mathtt{d}$, with some features.

Semantics: Package

- A Package encloses a collection of (mutually visible) Classes .
- Mutually Visible: Each Class has visibility of itself and of each other Class of the Package, in which it is defined: Hence the need of the FixedPoint, below, in the Table.

Table16.2 – Package

Semantic Functions

$$\mathcal{D}_E[\![\mathtt{D}]\!] : (\mathtt{Env} \times \mathtt{Store}) \to \mathtt{Env}_\perp$$

$$\mathcal{D}_E[\![\mathtt{classes\ C}_1; ...; \mathtt{C}_n\ \mathtt{end}]\!]_\rho =$$

$$\mathtt{Y}\delta\,.\,\mathcal{D}_E[\![\mathtt{C}_1]\!]_\delta \circ ... \circ \mathcal{D}_E[\![\mathtt{C}_n]\!]_\delta \circ \rho$$

- A Method may contain expressions for:
  - The creation,
  - The access and the modification of the Fields,
  - The invocations of Methods

  of Objects of each class in the Package.

Semantics: Class/1

- A Class defines one function for the creation of the Objects of the Class: Function d, and its corresponding, denotable value, A(d).

Table16.3.1 – Class

Semantic Functions

$\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \rightarrow \mathtt{Env}$

$\mathcal{D}_E[\![\mathtt{class\ I\ extends\ I_u}\ \{\overline{\mathtt{F}}\ \mathtt{K}\ \overline{\mathtt{M}}\}]\!]_\rho =$

$\mathtt{Let}\{\mathtt{d} = \lambda\overline{\mathtt{v}}_\mathtt{K}.\lambda\mathtt{s}.(\mathtt{O}_\mathtt{v}(\rho_\mathtt{o}), \mathtt{s}_\mathtt{o})\}$

$\mathtt{bind}(\mathtt{I}, \mathtt{A}(\mathtt{d}), \rho)$

- The creation of an Object
  - **may** be parametric (as in Java) and receive a list of parameters $\overline{\mathtt{v}}_\mathtt{K}$.
  - **always** modifies the store, $\mathtt{s}_\mathtt{o}$, where the Objects are allocated
  - **always** returns a value, $\mathtt{O}_\mathtt{v}(\rho_\mathtt{o})$, which is essentially, the environment $\rho_\mathtt{o}$ of the bindings for the Fields and the Methods of the created Object.
- The definition of $(\mathtt{O}_\mathtt{v}(\rho_\mathtt{o}),\mathtt{s}_\mathtt{o})$ depends on the specific OO Language.

Semantics: Class/2

- $(\mathtt{O_v}(\rho_o),\mathtt{s}_o)$ depends on the specific OO Language and may contain small technical complications.
- Here, in FJ, it consists of 3 steps:
  - Creation of an Object the Class super, $(\mathtt{o_u},\mathtt{s_u})$, i.e. of the Class that has been declared be the super of the Class of Object to be created;
  - Extension of $(\mathtt{o_u},\mathtt{s_u})$ with the Fields and with the Methods of the Class;
  - Evaluation of the Initialization Code, i.e. The code that has been specified for this aim (in a block or in the Constructor, as in Java).

Table16.3.2 – Class

Funzioni Semantiche

$$\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \longrightarrow \mathtt{Env}_\perp$$

$$\mathcal{D}_E[\![\mathtt{class\ I\ extends\ I_u\ \{\overline{F}\ K\ \overline{M}\}}]\!]\rho =$$

$$\mathtt{Let}\{\mathtt{d} = \lambda\overline{\mathtt{v}}_\mathtt{K}.\lambda\mathtt{s}.$$

$$\mathtt{Let}\ \{**\textit{ Creation Object super }**\}$$

$$\{**\textit{ Extension of the Object }**\}$$

$$\{**\textit{ Evaluation of the Constructor }**\}$$

$$(\mathtt{O_v}(\rho_\mathtt{o}), \mathtt{s_o})\}$$

$$\mathtt{bind}(\mathtt{I}, \mathtt{A}(\mathtt{d}), \rho)$$

Semantics: Class/3

- Creation of an Object of the Class super, (o_u, s_u)

Table16.3.3 – Class

Semantic Functions

$$\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \to \mathtt{Env}_\perp$$

$$\mathcal{D}_E[\![\mathtt{class\ I\ extends\ I_u\ \{\overline{F}\ K\ \overline{M}\}}]\!]_\rho =$$

$$\mathtt{Let}\{\mathtt{d} = \lambda \overline{v}_K . \lambda s .$$

$$\mathtt{Let}\{\mathtt{A}(\mathtt{d_u}) = \rho(\mathtt{I_u})\}$$

$$\{(\mathtt{O_v}(\rho_u), \mathtt{s_u}) = \mathtt{d_u}()(\mathtt{s})\}$$

$$\{**\ \textit{Extension of the Object}\ **\}$$

$$\{**\ \textit{Evaluation of the Constructor}\ **\}$$

$$(\mathtt{O_v}(\rho_o), \mathtt{s_o})\}$$

$$\mathtt{bind}(\mathtt{I}, \mathtt{A}(\mathtt{d}), \rho)$$

- Note the use of $\mathtt{d_u}()(\mathtt{s})$: It requires that the function $\mathtt{d_u}$ is assumed be a parameterless function (see discussion on slide "Semantics: Class/5", later on)
- The Object (o_u, s_u), in Java, is a value that is accessible through the use of "function" `super`
- Here, we omit these aspects and, consequently, the ability to access such an Object: However, simple changes to the semantics (that has been given in these slides) allow to consider them and to include the "function" `super` (This is left for an exercise, later on)

Semantics: Class/4

- Extension of the Object (o_u,s_u) with the Fields, the Constructor and the Methods of the Class

Table16.3.4 – Class

Semantic Functions

$$\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \longrightarrow \mathtt{Env}_\perp$$

$$\mathcal{D}_E[\![\mathtt{class\ I\ extends\ I_u\ \{\overline{F}\ K\ \overline{M}\}}]\!]_\rho =$$
$$\mathtt{Let}\{\mathtt{d} = \lambda\overline{v}_K.\lambda s.$$
$$\mathtt{Let}\{** \textit{Creation Object super} : (\mathtt{O_v}(\rho_u), s_u) **\}$$
$$\{\mathtt{Let}\{\|A\| \equiv \lambda(\rho\prime, s\prime).\mathcal{D}[\![\mathtt{A}]\!]_{\rho\prime\circ\rho_u}(s_u \circ s\prime)\}$$
$$(\rho_o, s_{\overline{F}}) = \mathtt{Y}(\rho\prime, s\prime).(\|\overline{F}\| \circ \|K\| \circ \|\overline{M}\|)(\rho\prime, s\prime)\}$$
$$\{** \textit{Evaluation of the Constructor} **\}$$
$$(\mathtt{O_v}(\rho_o), s_o)\}$$
$$\mathtt{bind}(I, A(d), \rho)$$

- Note the use of a functional $\|A\|$: It applies to a pair of functions (ρ, s) and returns the new pair that results from $\mathcal{D}[\![\mathtt{A}]\!]$.
- Note the use of the FixedPoint on the composition of all the transformations: It is in order to guarantee that all the bindings (and all the locations that are allocated in the store) be visible and accessible in all the definitions of the Fields, of the Constructor and of the Methods.
- Note that, in addition to Fields and Methods, ρ_o contains also one Constructor.

- Remarks:
  - The semantics $\mathcal{D}[\![]\!]$ on the Class Fields could be the same of the variable declaration in procedural languages.
    – obviously, with the initialization expressions that are extended with the new expressions for Objects and OO languages.
  - The semantics of the Methods could be the same of the procedural declaration in procedural languages
    – but using a different invocation mechanism, as we will see later on.

Semantics: Class/4: Exercise

Table16.3.4 — Class

Semantic Functions

$$\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \longrightarrow \mathtt{Env}_\perp$$

$$\mathcal{D}_E[\![\mathtt{class\ I\ extends\ I_u\ \{\overline{F}\ K\ \overline{M}\}}]\!]_\rho =$$

$$\mathtt{Let}\{\mathtt{d} = \lambda \overline{\mathtt{v}}_\mathtt{K}.\lambda \mathtt{s}.$$

$$\mathtt{Let}\ \{** \textit{Creation Object super} : (\mathtt{O_v}(\rho_\mathtt{u}), \mathtt{s_u}) **\}$$

$$\{\mathtt{Let}\{\|\mathtt{A}\| \equiv \lambda(\rho\prime, \mathtt{s}\prime).\mathcal{D}[\![\mathtt{A}]\!]_{\rho\prime \circ \rho_\mathtt{u}}(\mathtt{s_u} \circ \mathtt{s}\prime)\}$$

$$(\rho_o, \mathtt{s}_{\overline{\mathtt{F}}}) = \mathtt{Y}(\rho\prime, \mathtt{s}\prime).(\|\overline{\mathtt{F}}\| \circ \|\mathtt{K}\| \circ \|\overline{\mathtt{M}}\|)(\rho\prime, \mathtt{s}\prime)\}$$

$$\{** \textit{Evaluation of the Constructor} **\}$$

$$(\mathtt{O_v}(\rho_\mathtt{o}), \mathtt{s_o})\}$$

$$\mathtt{bind}(\mathtt{I}, \mathtt{A}(\mathtt{d}), \rho)$$

Example

Exercise. Apply the semantics of the Table, above, to obtain $(\rho_o, \mathtt{s}_{\overline{\mathtt{F}}})$ when the body of the class has: $\overline{\mathtt{M}} \equiv \mathtt{A()\{Call\ B()\}; B()\{Call\ A()\}}$, and, for simplicity, K and $\overline{\mathtt{F}}$ are both, omitted.

Hence, complete:

$$(\rho_o, \mathtt{s}_{\overline{\mathtt{F}}}) = \mathtt{Y}(\rho\prime, \mathtt{s}\prime).(\|\overline{\mathtt{F}}\| \circ \|\mathtt{K}\| \circ \|\overline{\mathtt{M}}\|)(\rho\prime, \mathtt{s}\prime) = \mathtt{Y}(\rho\prime, \mathtt{s}\prime).\|\mathtt{A()\{Call\ B()\}; B()\{Call\ A()\}}\|(\rho\prime, \mathtt{s}\prime)$$

$$= \mathtt{Y}(\rho\prime, \mathtt{s}\prime).\mathcal{D}[\![\mathtt{A()\{Call\ B()\}; B()\{Call\ A()\}}]\!]_{\rho\prime \circ \rho_\mathtt{u}}(\mathtt{s_u} \circ \mathtt{s}\prime)$$

where, by using the semantics of procedures (see the lecture7-8):

$$= (\mathtt{Y}\rho\prime.\mathcal{D}_E[\![\mathtt{A()\{Call\ B()\}; B()\{Call\ A()\}}]\!]_{\rho\prime \circ \rho_\mathtt{u}}, \mathtt{s_u})$$

since the declaration of procedure leaves unchanged the store $\mathtt{s_u}$. Hence, the calculus reduces to the following:

$$\mathtt{Y}\rho\prime.\mathcal{D}_E[\![\mathtt{A()\{Call\ B()\}; B()\{Call\ A()\}}]\!]_{\rho\prime \circ \rho_\mathtt{u}}$$

Hence (see Lecture 13-14), let

$$\mathtt{H} \equiv \lambda\rho\prime.\mathtt{bind}(\mathtt{A}, \mathtt{F}(\mathcal{M}[\![\mathtt{Call\ B()}]\!]_{\rho\prime}), \mathtt{bind}(\mathtt{B}, \mathtt{F}(\mathcal{M}[\![\mathtt{Call\ A()}]\!]_{\rho\prime}), \rho_\mathtt{u}))$$

$$\mathtt{H} \equiv \lambda\rho\prime.\lambda\mathtt{x}.\mathtt{if}(\mathtt{x} = \mathtt{A})\ \mathtt{then}\ \mathtt{F}(\mathcal{M}[\![\mathtt{Call\ B()}]\!]_{\rho\prime})\ \mathtt{else}\ (\mathtt{if}(\mathtt{x} = \mathtt{B})\ \mathtt{then}\ \mathtt{F}(\mathcal{M}[\![\mathtt{Call\ A()}]\!]_{\rho\prime})\ \mathtt{else}\ \rho_\mathtt{u}(\mathtt{x}))$$

$$\mathtt{H}^0_\perp \equiv \perp; \qquad \mathtt{H}^{k+1}_\perp \equiv \mathtt{H}[\rho\prime \leftarrow \mathtt{H}^k]$$

... (3 steps) ...

Semantics: Class/4: Exercise-bis

- Extension of the Object (o_u,s_u) with the Fields, the Constructor and the Methods of the Class

Table16.3.4 – Class

Semantic Functions

$\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \longrightarrow \mathtt{Env}_\perp$

$\mathcal{D}_E[\![\mathtt{class\ I\ extends\ I_u\ \{\overline{F}\ K\ \overline{M}\}}]\!]_\rho =$

$\mathtt{Let}\{\mathtt{d} = \lambda \overline{\mathtt{v}}_\mathtt{K}.\lambda \mathtt{s}.$

$\mathtt{Let}\ \{** \textit{Creation Object super} : (\mathtt{O_v}(\rho_u), \mathtt{s_u}) **\}$

$\{\mathtt{Let}\{\|\mathtt{A}\| \equiv \lambda(\rho\prime, \mathtt{s}\prime).\mathcal{D}[\![\mathtt{A}]\!]_{\rho\prime \circ \rho_u}(\mathtt{s_u} \circ \mathtt{s}\prime)\}$

$(\rho_o, \mathtt{s}_{\overline{F}}) = \mathtt{Y}(\rho\prime, \mathtt{s}\prime).(\|\overline{\mathtt{F}}\| \circ \|\mathtt{K}\| \circ \|\overline{\mathtt{M}}\|)(\rho\prime, \mathtt{s}\prime)\}$

$\{** \textit{Evaluation of the Constructor} **\}$

$(\mathtt{O_v}(\rho_o), \mathtt{s_o})\}$

$\mathtt{bind}(\mathtt{I}, \mathtt{A}(\mathtt{d}), \rho)$

Example

Exercise1. Apply the semantics of the Table, above, to obtain $(\rho_o, \mathtt{s}_{\overline{F}})$ when the body of the class has: $\overline{\mathtt{M}} \equiv \mathtt{A()\{Call\ B()} + 4\}; \mathtt{B()\{3\}}$ and, for simplicity, $\mathtt{K}$ and $\overline{\mathtt{F}}$ are both, omitted.

Exercise2 Apply the semantics of the Table, above, to obtain $(\rho_o, \mathtt{s}_{\overline{F}})$ when the body of the class has: $\overline{\mathtt{F}} \equiv \mathtt{x} = \mathtt{y} + 3; \mathtt{y} = 4$, and, for simplicity, $\mathtt{K}$ and $\overline{\mathtt{M}}$ are both, omitted.

Hence, complete:

$(\rho_o, \mathtt{s}_{\overline{F}}) = \mathtt{Y}(\rho\prime, \mathtt{s}\prime).(\|\overline{\mathtt{F}}\| \circ \|\mathtt{K}\| \circ \|\overline{\mathtt{M}}\|)(\rho\prime, \mathtt{s}\prime)$

$= \mathtt{Y}(\rho\prime, \mathtt{s}\prime).\|\mathtt{x} = \mathtt{y} + 3; \mathtt{y} = 4\|(\rho\prime, \mathtt{s}\prime) = \mathtt{Y}(\rho\prime, \mathtt{s}\prime).\lambda(\rho\prime, \mathtt{s}\prime).(\mathcal{D}[\![\mathtt{x} = \mathtt{y} + 3; \mathtt{y} = 4]\!]_{\rho\prime \circ \rho_u}(\mathtt{s_u} \circ \mathtt{s}\prime))(\rho\prime, \mathtt{s}\prime)$

$= (\mathtt{Y}\rho\prime.\mathcal{D}_E[\![\mathtt{x} = \mathtt{y} + 3; \mathtt{y} = 4]\!]_{\rho\prime \circ \rho_u}, \mathtt{s_u} \circ \mathtt{s}\prime)$

let $\mathtt{H} \equiv$

Semantics: Class/5

- Evaluation of the Initialization (specified in a block or in a Constructor)

Table16.3.5 – Class

Semantic Functions

$$\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \longrightarrow \mathtt{Env}_\perp$$

$$\mathcal{D}_E[\![\mathtt{class\ I\ extends\ I_u\ \{\overline{F}\ K\ \overline{M}\}}]\!]\rho =$$

$$\mathtt{Let}\{\mathtt{d} = \lambda\overline{\mathtt{v}}_\mathtt{K}.\lambda\mathtt{s}.$$

$$\mathtt{Let}\ \{** \textit{Creation Object super} : (\mathtt{O_v}(\rho_\mathtt{u}), \mathtt{s_u}) **\}$$

$$\{** \textit{Extension of the Object} : (\rho_o, \mathtt{s}_{\overline{\mathtt{F}}}) **\}$$

$$\{\mathtt{s}_o = \rho_o(\mathtt{cstr})(\overline{\mathtt{v}}_\mathtt{K})(\mathtt{s}_{\overline{\mathtt{F}}})\}$$

$$(\mathtt{O_v}(\rho_\mathtt{o}), \mathtt{s_o})\}$$

$$\mathtt{bind}(\mathtt{I}, \mathtt{A}(\mathtt{d}), \rho)$$

- In the semantics of **Table 16.3.5**, for simplicity sake, the Constructors behave like parameterless procedures: Hence $(\overline{\mathtt{v}}_\mathtt{K}) = ()$ is an empty list.
- However, when parameters are considered, $\overline{\mathtt{v}}_\mathtt{K}$ must be split in two sub-lists of parameters, $\overline{\mathtt{v}}^{\mathtt{u}}_\mathtt{K}$ and $\overline{\mathtt{v}}^{\mathtt{o}}_\mathtt{K}$: The first one contains the parameters of Constructor of the Class super and the other, that of the class.
  – Then, the first sub-list must be passed to the Constructor of the Class $\mathtt{I_u}$.
  – Moreover, the size of the sub-lists depends from the arity (number of the parameters) that has been declared for each Constructor.

Example

Exercise 1 Java has Object Constructors with parameters: Discuss how Java programs should be re-phrased if only parameterless Constructors are allowed.

Exercise 2 Discuss how the list of parameters should be split, when the Constructors have parameters. Then modify the semantics of the Class in order to provide for the use of Constructors with parameters.

Semantics: Class/6

- The Complete Definition

Table16.3 – Class

Semantic Functions

$$\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \to \mathtt{Env}_\perp$$

$$\mathcal{D}_E[\![\mathtt{class\ I\ extends\ I_u\ \{\overline{F}\ K\ \overline{M}\}}]\!]\rho =$$

$$\mathtt{Let}\{\mathtt{d} = \lambda \overline{\mathtt{v}}^{\mathtt{u}}_{\mathtt{k}}\, \overline{\mathtt{v}}^{\mathtt{o}}_{\mathtt{k}}.\lambda \mathtt{s}.$$

$$\mathtt{Let}\{\mathtt{A}(\mathtt{d_u}) = \rho(\mathtt{I_u})\}$$

$$\{(\mathtt{0_v}(\rho_\mathtt{u}), \mathtt{s_u}) = \mathtt{d_u}(\overline{\mathtt{v}}^{\mathtt{u}}_{\mathtt{k}})(\mathtt{s})\}$$

$$\{\mathtt{Let}\{\|\mathtt{A}\| \equiv \lambda(\rho\prime, \mathtt{s}\prime).\mathcal{D}[\![\mathtt{A}]\!]_{\rho\prime \circ \rho_\mathtt{u}}(\mathtt{s_u} \circ \mathtt{s}\prime);$$

$$(\rho_o, \mathtt{s}_{\overline{\mathtt{F}}}) = \mathtt{Y}(\rho\prime, \mathtt{s}\prime).(\|\overline{\mathtt{F}}\| \circ \|\mathtt{K}\| \circ \|\overline{\mathtt{M}}\|)(\rho\prime, \mathtt{s}\prime)\}$$

$$\{\mathtt{s}_o = \rho_o(\mathtt{cstr})(\overline{\mathtt{v}}^{\mathtt{o}}_{\mathtt{k}})(\mathtt{s}_{\overline{\mathtt{F}}})\}$$

$$(\mathtt{0_v}(\rho_\mathtt{o}), \mathtt{s_o})\}$$

$$\mathtt{bind}(\mathtt{I}, \mathtt{A}(\mathtt{d}), \rho)$$

- In this semantics, the self-reference **this** has been omitted:
  - What is it?
  - Can it be avoided?: How? Where?
  - How can it be added to the Language ?

Semantics: Class/6bis

- In this semantics, the self-reference **this**, as well as the reference to the Object **super**, has been omitted:
  - What is it?
  - Can it be avoided?: How? Where?
  - How can it be added to the Language ?
- In Java, the self-reference **this** supports recursively defined, Non-Static, Methods

Example

Table16.3 – Class with Self – Reference

Semantic Functions

$$\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \longrightarrow \mathtt{Env}_\perp$$

$$\mathcal{D}_E[\![\mathtt{class\ I\ extends\ I_u\ \{\overline{F}\ K\ \overline{M}\}}]\!]\rho =$$
$$\mathtt{Let}\{\mathtt{d} = \lambda\overline{\mathtt{v}}^{\mathtt{u}}_{\mathtt{k}}\,\overline{\mathtt{v}}^{\mathtt{o}}_{\mathtt{k}}.\lambda\mathtt{s}.$$
$$\mathtt{Let}\{\mathtt{A}(\mathtt{d_u}) = \rho(\mathtt{I_u})\}$$
$$\{(\mathtt{O_v}(\rho_\mathtt{u}), \mathtt{s_u}) = \mathtt{d_u}(\overline{\mathtt{v}}^{\mathtt{u}}_{\mathtt{k}})(\mathtt{s})\}$$
$$\{\mathtt{Let}\{\|\mathtt{A}\| \equiv \lambda(\rho\prime, \mathtt{s}\prime).\mathcal{D}[\![\mathtt{A}]\!]_{\rho\prime\circ\rho_\mathtt{u}}(\mathtt{s_u} \circ \mathtt{s}\prime);$$
$$\mathtt{self} \equiv \lambda(\rho\prime, \mathtt{s}\prime).(\mathtt{bind}(\mathtt{this}, \mathtt{O_d}(\rho\prime), \rho\prime), \mathtt{s}\prime)\}$$
$$(\rho_o, \mathtt{s}_{\overline{\mathtt{F}}}) = \mathtt{Y}(\rho\prime, \mathtt{s}\prime).(\|\overline{\mathtt{F}}\| \circ \|\mathtt{K}\| \circ \|\overline{\mathtt{M}}\| \circ \mathtt{self})(\rho\prime, \mathtt{s}\prime)\}$$
$$\{\mathtt{s}_o = \rho_o(\mathtt{cstr})(\overline{\mathtt{v}}^{\mathtt{o}}_{\mathtt{k}})(\mathtt{s}_{\overline{\mathtt{F}}})\}$$
$$(\mathtt{O_v}(\rho_\mathtt{o}), \mathtt{s_o})\}$$
$$\mathtt{bind}(\mathtt{I}, \mathtt{A}(\mathtt{d}), \rho)$$

Semantics: The construct Class with **this** and **super**

- In the semantics above, the reference to the Object **super** is again omitted: What is it? Can it be avoided?: How? Where? How can it be added to the Language ?
- In Java, the reference **super** supports code extension and reuse

Example

Table16.3 — Semantics of Class with this and super

Funzioni Semantiche

$$
\begin{array}{l}
\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \longrightarrow \mathtt{Env}_\perp \\
\quad \mathcal{D}_E[\![\mathtt{class\ I\ extends\ I_u\ \{\overline{F}\ K\ \overline{M}\}}]\!]\rho = \\
\quad\quad \mathtt{Let}\{\mathtt{d} = \lambda \overline{\mathtt{v}}^{\mathtt{u}}_{\mathtt{k}}\, \overline{\mathtt{v}}^{\mathtt{o}}_{\mathtt{k}}.\lambda \mathtt{s}. \\
\quad\quad\quad\quad \mathtt{Let}\{\mathtt{A}(\mathtt{d_u}) = \rho(\mathtt{I_u})\} \\
\quad\quad\quad\quad\quad\quad \{(\mathtt{O_v}(\rho_{\mathtt{u}\prime}), \mathtt{s_u}) = \mathtt{d_u}(\overline{\mathtt{v}}^{\mathtt{u}}_{\mathtt{k}})(\mathtt{s})\} \\
\quad\quad\quad\quad\quad\quad \{\rho_{\mathtt{u}} = \mathtt{bind}(\mathtt{super}, \mathtt{O_d}(\rho_{\mathtt{u}\prime}), \rho_{\mathtt{u}\prime})\} \\
\quad\quad\quad\quad\quad\quad \{\mathtt{Let}\{\|\mathtt{A}\| \equiv \lambda(\rho\prime, \mathtt{s}\prime).\mathcal{D}[\![\mathtt{A}]\!]_{\rho\prime \circ \rho_{\mathtt{u}}}(\mathtt{s_u} \circ \mathtt{s}\prime); \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathtt{self} \equiv \lambda(\rho\prime, \mathtt{s}\prime).(\mathtt{bind}(\mathtt{this}, \mathtt{O_d}(\rho\prime), \rho\prime), \mathtt{s}\prime)\} \\
\quad\quad\quad\quad\quad\quad\quad\quad (\rho_o, \mathtt{s}_{\overline{\mathtt{F}}}) = \mathtt{Y}(\rho, \mathtt{s}).(\|\overline{\mathtt{F}}\| \circ \|\mathtt{K}\| \circ \|\overline{\mathtt{M}}\| \circ \mathtt{self})(\rho\prime, \mathtt{s}\prime)\} \\
\quad\quad\quad\quad\quad\quad \{\mathtt{s}_o = \rho_o(\mathtt{cstr})(\overline{\mathtt{v}}^{\mathtt{o}}_{\mathtt{k}})(\mathtt{s}_{\overline{F}})\} \\
\quad\quad\quad\quad\quad\quad (\mathtt{O_v}(\rho_{\mathtt{o}}), \mathtt{s_o})\} \\
\quad\quad\quad\quad \mathtt{bind}(\mathtt{I}, \mathtt{A}(\mathtt{d}), \rho)
\end{array}
$$

Semantics: The Class Object

- The class Object is the top of each Hierarchy: Hence, it requires a special definition
- In effect the Class Object is the only Class that has not super Class
- Here, for simplicity sake, we assume that class Object has not bindings (in Java, it contains the bindings for all primitive fields and methods)

Table16.4 – Class Object

Semantic Functions

$$\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \rightarrow \mathtt{Env}_\perp$$

$$\mathcal{D}_E[\![\mathtt{class\ Object\ \{\}}]\!]_\rho =$$
$$\mathtt{Let}\{\mathtt{d} = \lambda().\lambda\mathtt{s}.(\mathtt{O}_\mathtt{v}(\lambda\mathtt{x}.\mathtt{x}), \mathtt{s})\}$$
$$\mathtt{bind}(\mathtt{I}, \mathtt{A}(\mathtt{d}), \rho)$$

- The semantics uses the empty environment $\lambda\mathtt{x}.\mathtt{x}$ (see Table2 in Lecture 7-8)

Implementation Notes: Class/1

- How is the Class Implementation?

Table16.3 – Semantics of Class

Semantic Functions

$$\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \to \mathtt{Env}_\perp$$

$$\mathcal{D}_E[\![\mathtt{class\ I\ extends\ I_u\ \{\overline{F}\ K\ \overline{M}\}}]\!]_\rho =$$
$$\mathtt{Let}\{\mathtt{d} = \lambda \overline{\mathtt{v}}^{\mathtt{u}}_{\mathtt{k}}\, \overline{\mathtt{v}}^{\mathtt{o}}_{\mathtt{k}}.\lambda \mathtt{s}.$$
$$\mathtt{Let}\{\mathtt{A}(\mathtt{d_u}) = \rho(\mathtt{I_u})\}$$
$$\{(\mathtt{O_v}(\rho_\mathtt{u}), \mathtt{s_u}) = \mathtt{d_u}(\overline{\mathtt{v}}^{\mathtt{u}}_{\mathtt{k}})(\mathtt{s})\}$$
$$\{\mathtt{Let}\{\|\mathtt{A}\| \equiv \lambda(\rho\prime, \mathtt{s}\prime).\mathcal{D}[\![\mathtt{A}]\!]_{\rho\prime \circ \rho_\mathtt{u}}(\mathtt{s_u} \circ \mathtt{s}\prime);$$
$$(\rho_o, \mathtt{s}_{\overline{\mathtt{F}}}) = \mathtt{Y}(\rho\prime, \mathtt{s}\prime).(\|\overline{\mathtt{F}}\| \circ \|\mathtt{K}\| \circ \|\overline{\mathtt{M}}\|)(\rho\prime, \mathtt{s}\prime)\}$$
$$\{\mathtt{s}_o = \rho_o(\mathtt{cstr})(\overline{\mathtt{v}}^{\mathtt{o}}_{\mathtt{k}})(\mathtt{s}_{\overline{\mathtt{F}}})\}$$
$$(\mathtt{O_v}(\rho_\mathtt{o}), \mathtt{s_o})\}$$
$$\mathtt{bind}(\mathtt{I}, \mathtt{A}(\mathtt{d}), \rho)$$

- The implementation depends on the structure of the available machine: **JVM** (Java Virtual Machine) or **CLR** (Common Language Runtime of .NET), or **P-Machine** (Pascal) or **3AC** machine (3 address code)

Implementation Notes: Class/2 - NO

- Depends on the available machine: P-Machine (Pascal)
  - Control is supported be the Activation Records
  - The core structure is: AR, Static Memory, Dynamic Memory for the Control Stack (for method application, see later on) and for the Heap (of the creates Objects, see later on)
  - The frame of an AR contains the binding name - denotation of each Class of the Package that has in its scope, the running code.

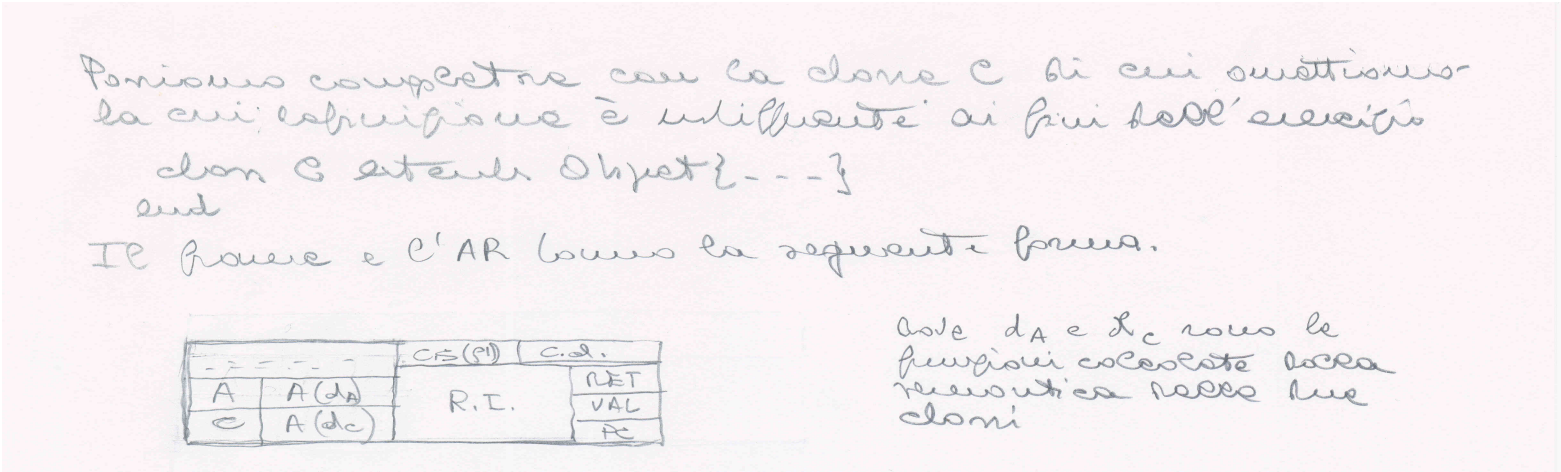
Example

Suitably complete the Package below:

```
classes class A extends Object{
        int x = this.y.x + 2;
        C y = new C();
        cstr(C u) {y=u;};
        int add(int u){this.x=2*this.x+u;};
        }
A z = new A(new C());
```

Then, show a graphical picture of the structure of the ARs before the introduction of the variable z.

- Solution: The structure contains only one AR



Semantics: Object Creation (Method invocation, Object Selection)

- Object Creation is the invocation of the function defined by the Class Semantics
- Hence, it is similar to procedure/function invocation (Lecture 13-14):

Table16.5.1 – Object Creation

Semantic Function

$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{Store} \rightarrow (\mathtt{Val} \times \mathtt{Store})_\perp$

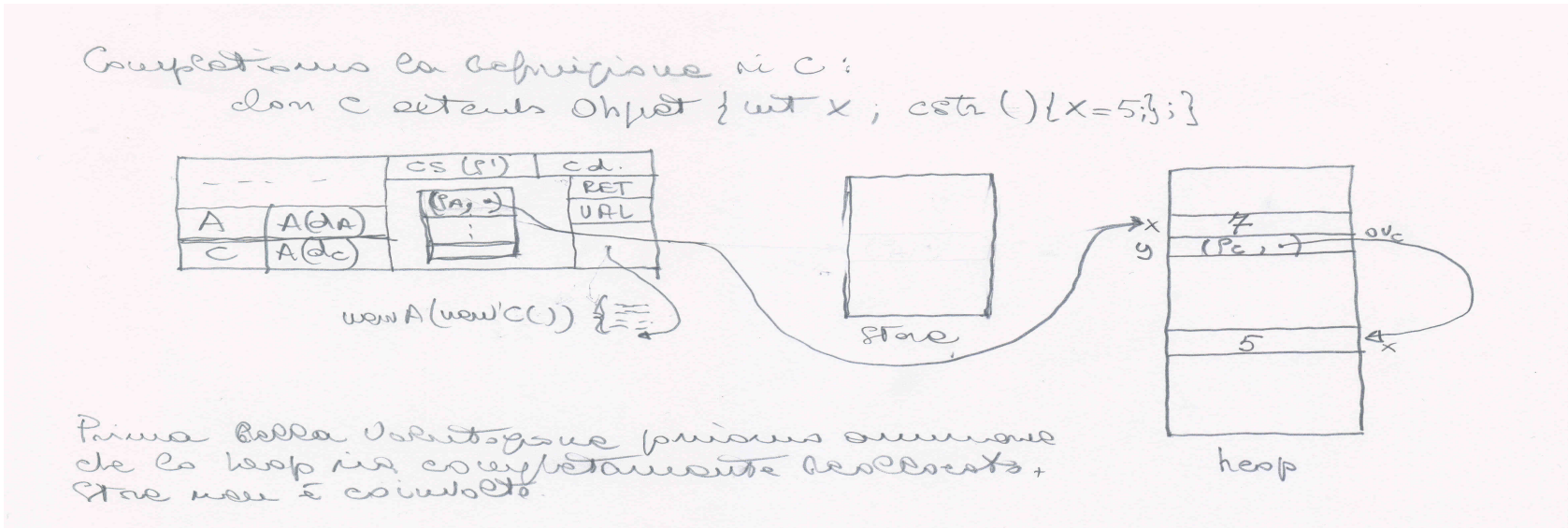
$\mathcal{E}[\![\mathtt{new\ I(A_1 \ldots A_n)}]\!]_\rho(\mathtt{s}) =$

$\mathtt{Let\{A(d)} = \rho(\mathtt{I}), ((v_1 \ldots v_n), \mathtt{s_n}) = \mathcal{T}[\![(\mathtt{A_1 \ldots A_n})]\!]_\rho(\mathtt{s})\}$

$\mathtt{d}(v_1 \ldots v_n)(\mathtt{s_n})$

Example

Show a graphical picture of the store before and after the evaluation of **new A(new C())** (use the AR of the previous example)



Semantics: Method invocation (Selection)

- The invocation of a method on an Object, is similar to procedure/function invocation but the binding of the method is looked up in the environment denoted by the Object.

Table16.5.2 – Invocation of Method

Funzioni Semantiche

$$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{Store} \to (\mathtt{Val} \times \mathtt{Store})_\perp$$

$$\mathcal{E}[\![\mathtt{E.I}(\mathtt{A}_1 \ldots \mathtt{A}_n)]\!]_\rho(\mathtt{s}) =$$
$$\mathtt{Let}\{(\mathtt{O}_v(\rho_0), \mathtt{s}_0) = \mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s})\}$$
$$\{((v_1 \ldots v_n), \mathtt{s}_n) = \mathcal{T}[\![(\mathtt{A}_1 \ldots \mathtt{A}_n)]\!]_\rho(\mathtt{s}_0),\ \mathtt{F}(\mathtt{f}) = \rho_0(\mathtt{I})\}$$
$$\mathtt{f}(v_1 \ldots v_n)(\mathtt{s}_n)$$

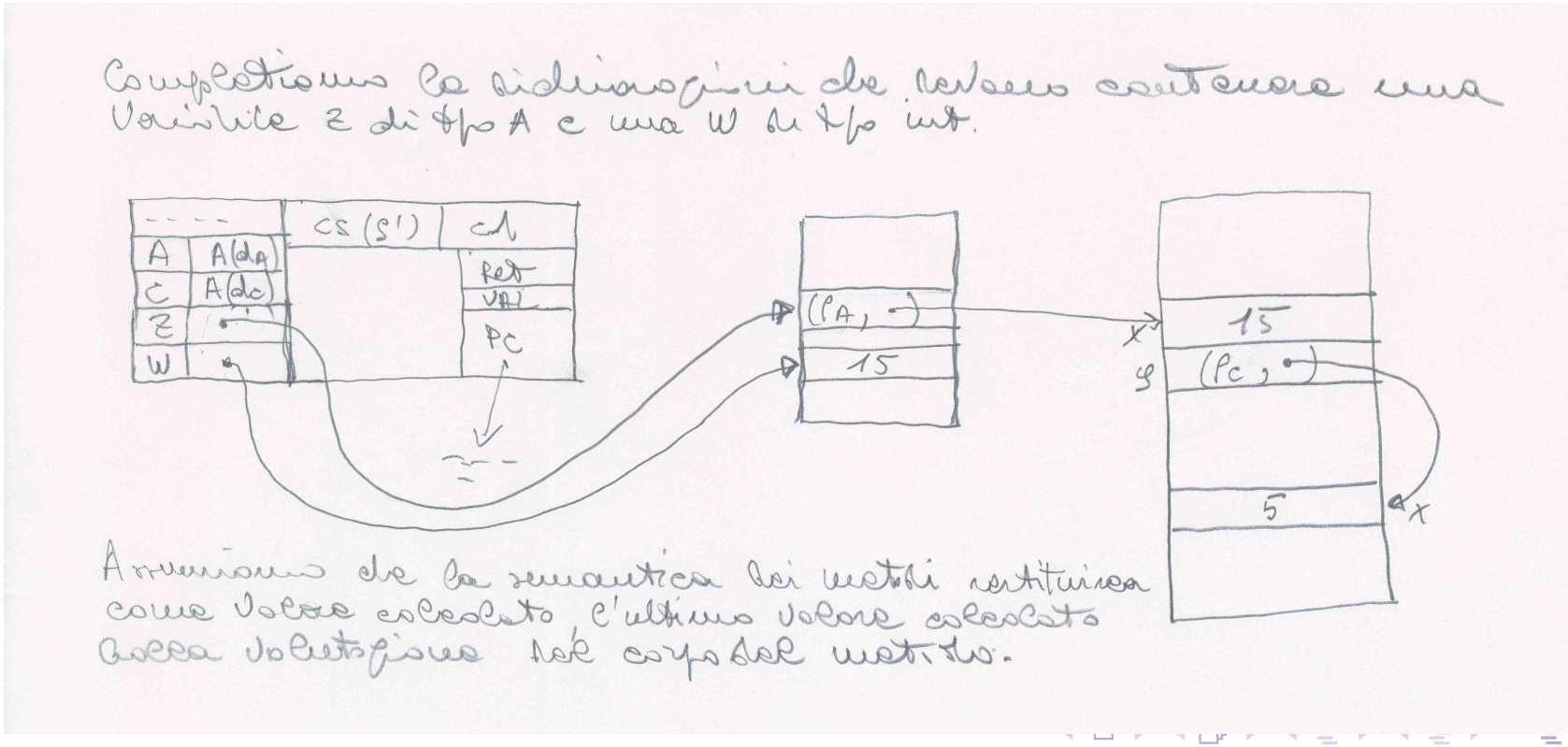
Example

```
classes class A extends Object{
        int x = this.y.x + 2;  C y = new C();
        cstr(C u) {y=u;};
        int add(int u){this.x=2*this.x+u;};
        }
        class C extends Object{int x; cstr(){x=5;}}
        A z = new A(new C());
        int w = z.add(1);
```

Implementation: Method invocation (Selection)/1 - NO

Example

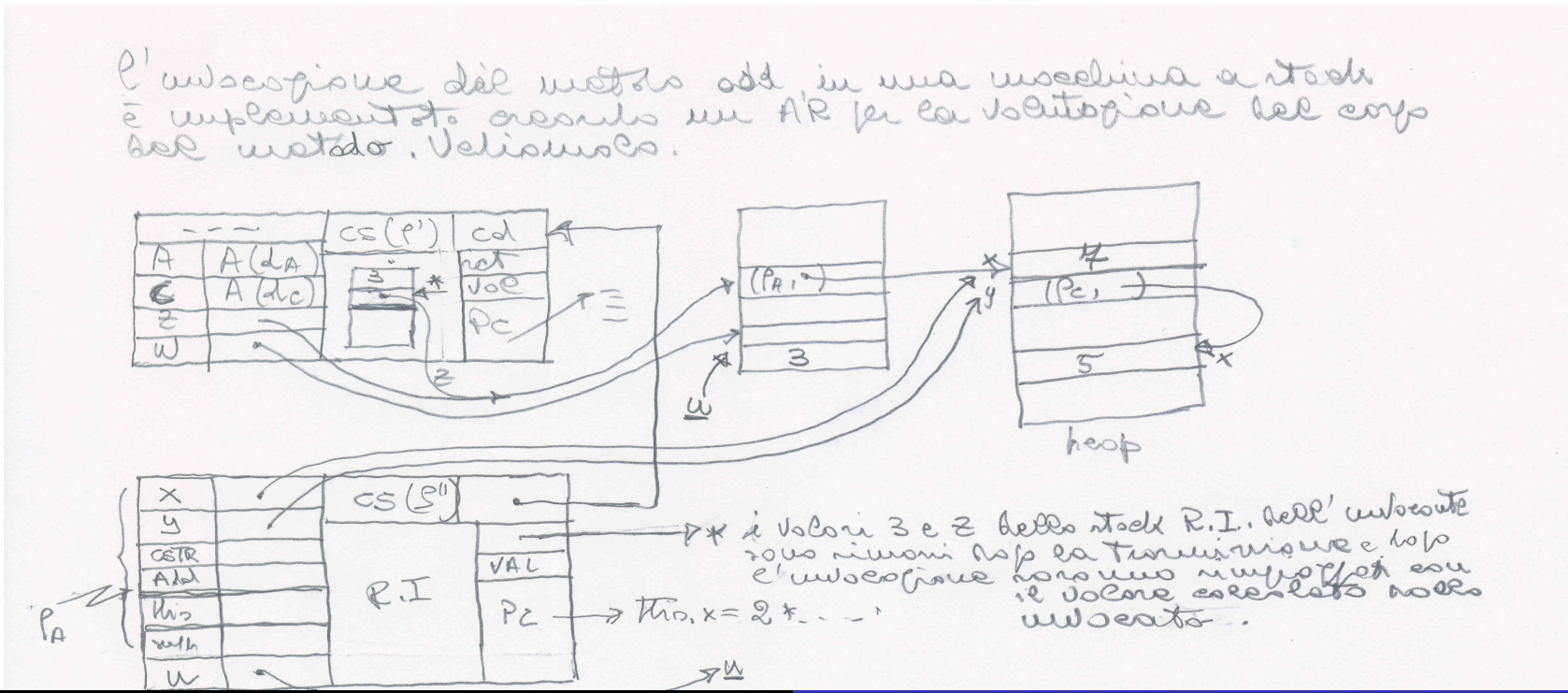
```
classes class A extends Object{
        int x = this.y.x + 2;  C y = new C();
        cstr(C u) {y=u;};
        int add(int u){this.x=2*this.x+u;};
        }
        class C extends Object{int x; cstr(){x=5;}}
        A z = new A(new C());
        int w = z.add(1);
```



Implementation: Activation Record of an invoked Method - NO

Example

```
classes class A extends Object{
        int x = this.y.x + 2;  C y = new C();
        cstr(C u) {y=u;};
        int add(int u){this.x=2*this.x+u;};
        }
        class C extends Object{int x; cstr(){x=5;}}
        A z = new A(new C());
        int w = z.add(1);
```



Semantics: Object Selection

- Object Selection is an access to values that are accessible only through the bindings of the environment denoted by the Object

Table16.5.3 – Selection of super (this) and of the fields

Semantic Functions

$\mathcal{E}[\![\mathtt{E}]\!]_\rho$: $\mathtt{Store} \longrightarrow (\mathtt{Val} \times \mathtt{Store})_\perp$

$$\mathcal{E}[\![\mathtt{E.super}]\!]_\rho(\mathtt{s}) = \mathtt{Let}\{(\mathtt{0_v}(\rho_0), \mathtt{s_0}) = \mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s})\} \; (\rho_0(\mathtt{super}), \mathtt{s_0})$$

$$\mathcal{E}[\![\mathtt{E.I}]\!]_\rho(\mathtt{s}) = \mathtt{Let}\{(\mathtt{0_v}(\rho_0), \mathtt{s_0}) = \mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s})\} \; (\mathtt{MV}(\mathtt{look}(\rho_0(\mathtt{I}), \mathtt{s_0})), \mathtt{s_0})$$

Example (Object Creation and Selection, and Method Invocation)

Exsercise 1. Show the graphical picture of the structure of the store before and after the evaluation of...

```
classes class A extends Object{
            int x = this.y.x + 2;
            C y = new C();
            cstr(C u) {y=u;};
            int add(int u){this.x=2*this.x+u;};}
        class B extends A{
            int x = 0;
            cstr() {x=10;};
            int add(int u){this.x=u;};}
        class C extends Object{int x = 7; cstr(){};}
end
A z = new B(new C());
int w = z.super.x;
```

Exsercise 2. Show how the program above is rephrased in Java

Additional Mechanisms: Generic and Sub-type Polymorphism

- Sub-type Polymorphism. It is explicitly expressed in Java through **extends**
- In the example below, `A` is sub-type of `B`: The Objects of `A` are also values of the type `B`

Example

Complete the Java Package below:

```
class A<T extends B> extends B{
        T x = this.y.x.m1(2);
        C<T> y = new C<T>;
        A(C<T> u) {y=u;};
        int add(int u){this.x.m2(2*this.x.m3(u));};
        }
```

by providing for a suitable definition of methods `m1, m2, m3`

- Note the use of Generic Polymorphism: The Type variable `T` is universally quantified (on all the types that precede `B`)

Additionals: Interfaces and Anonymous Classes/1

- Interfaces may have many, different uses:
  - Type Constraints
  - Signature, i.e. API, for Abstract Data Types Data
  - In Java, Interfaces are combined with Anonymous Classes for dynamic generation of Object of new Classes (types)

Example (Type constrains in Divide and Conquer Methodology, see Lecture15-17 and Lecture20)

```
interface Collection<T>{//For immutable values
   Collection<T> addE(T u);
   Collection<T> appendE(Collection<T> c2);
   Collection<T> filterE(Predicatef<T> r, T x); //Additional argument x since Functions as
   T selE();                                    //values are lacking
   int sizeE();
}
interface Ordered<T>{
   Predicatef<T> greater();
   Predicatef<T> equiv();
   Predicatef<T> less();
}
class SortableCollection<T extends Ordered<T>> extends Collection<T>{
   ...
   public Collection<T> QuickS (){
      if (sizeE()<2) return this;
      T u = selE();
      Collection<T> gt = filterE(u.greater());
      Collection<T> lt = filterE(lu.less());
      return lt.append(gt.addE(u));
  ... }
```

Additionals: Interfaces and Anonymous Classes/2

- Interfaces may have many, different uses:
  - Type Constraints
  - Signature, i.e. API, for Abstract Data Types Data
  - In Java, Interfaces are combined with Anonymous Classes for dynamic generation of Object of new Classes (types)

Example (Interface used as a Java Abstract Class)

```
interface List <T>;{
   T hd()throws NotApplicable;;
   List<T> tl()throws NotApplicable;;
}
class empty<T> implements List<T>{
   public empty(){};
   public List<T> cons(T e) {return new nonempty<T>(e,this);}
   public T hd() throws NotApplicable {throw new NotApplicable("hd");}
   public List<T> tl() throws NotApplicable {throw new NotApplicable("tl");}
}
class nonempty<T> implements List<T>{
   private T elem;
   private List<T> rest;
   nonempty(T e, List<T> r){elem=e; rest=r;}
   public List<T> cons(T e) {return new nonempty<T>(elem,this);}
   public T hd(){return elem;}
   public List<T> tl() {return rest;}
}
```

Additionals: Interfaces and Anonymous Classes/3

- Interfaces may have many, different uses:
  - Type Constraints
  - Signature, i.e. API, for Abstract Data Types Data
  - In Java, Interfaces are combined with Anonymous Classes for dynamic generation of Object of new Classes (types)

Example (Interface combined with Anonymous Classes to mimic a function calculus)

```
interface Fun<T1, T2> {T1 apply(T2 x)}
        ...
Fun<int, int>sqr= new Fun<int, int>(){
        apply(int x){return x*x;}};
Fun<int, int>fact= new Fun<int, int>(){
        apply(int x){return (x==0)?1:x*apply(x-1);}};
        ...
... sqr.apply(5)...
```

Exercise. Complete the Java program that introduced the class SortableCollection. Then run it for sorting the sequence (7,4),(12,1),(3,11) by using the following ordering relation: `(x1,y1)<(x2,y2) if x1*y2<x2*y2`. Eventually, show how the classes of the program can be suitably extended in order to include an operation for printing the value of a collection. Then, apply it to the case of the collection (7,4),(12,1),(3,11).

Additionals: Sub-Class and Class Hierarchy

- Many OO languages provide for a notion of Sub-Class and of Class Hierarchy: But such notions could be absent
- When the language lacks of such notions, the OO programming lacks of some relevant features, including:
  - Sub-type Polymorphism
  - Inheritance
  - Code Reuse

Methodologies: Inheritance and Super-Classes

- A class TwoPoint has been initially, defined in order to introduce points of a cartesian plan in some calculus.
- Next, the calculus has been extended to a calculus in a three dimensional space.
- The extension has been obtained by:
  - preserving the old code
  - defining the new code by re-using the old one

Example

```
public class TwoPoint{
  double x; double y;
  public Point (double a, double b){x=a; y=b};
  public double projectionX(){return x}
  ...
  public double distance (TwoPoint q){
    double xx = x-q.x; double yy = y-q.y;
    return Math.sqrt(xx*xx + yy*yy);
  }
}
public class ThreePoint extends TwoPoint{
  double z;  // ** but, the first and the second components are inherited
  public Point (double a, double b, double c){
      // ** to be re-defined (Meth. Overriding)**
  }
      // ** but, projectionX is inherited and works well
  public double distance(ThreePoint q){
      // ** to be Overridden **
  }
}
```

- What are the constructors of the two Classes?

Methodologies: Inheritance and Super-Classes/2

- The extension has been obtained by:
  - preserving the old code
  - defining the new code by re-using the old one

Example

```
public class TwoPoint{
  double x; double y;
  public Point (double a, double b){x=a; y=b};
  public double projectionX(){return x}
  ...
  public double distance (TwoPoint q){
    double xx = x-q.x; double yy = y-q.y;
    return Math.sqrt(xx*xx + yy*yy);
  }
}
public class ThreePoint extends TwoPoint{
  double z;  // ** but, the first and the second components are inherited
  public Point (double a, double b, double c){
      super.Point(a,b);
      z = c;
  }
      // ** but, projectionX is inherited and works well
  public double distance(ThreePoint q){
      double d = super.distance((TwoPoint)q);
      double d2 = d*d;
      double zz = z-q.z;
      return Math.sqrt(zz*zz + d2);
  }
}
```

- However, TwoPoint and ThreePoint are not ADT since they both, show the implementation.

Methodologies: Abstract Data Types

- An API for the type Relazione

Example

```
 public abstract class Relazione <A, B>{//A Java API for Relazione
        public abstract RelazioneC ();
        public abstract void add (A x, B y);
        public abstract void remove (A x, B y);
        public abstract LinkedList <A> getUno(B y);
 }

module type RELAZIONE =
 sig type ('a,'b) relazione
  val relazioneC:unit -> ('a,'b) relazione
  val add:  ('a,'b) relazione -> 'a -> 'b -> ('a,'b) relazione
  val remove:  ('a,'b) relazione -> 'a -> 'b -> ('a,'b) relazione
  val getUno:('a,'b) relazione -> 'b -> 'a list
 end;;
```

- The type Relazione in Java and in Ocaml differ one another in several ways:
  - One is for mutable values whilst the other one is for immutable values
  - Operations are curried in one type whilst are un-curried in the other one

Methodologies: Abstract Data Types/2

- Java allows API for mutable as well as immutable values
- But Operations are always un-curried

Example

```
 public abstract class Relazione <A, B>{//One Java API for the type Relazione
        public abstract Relazione<A, B> RelazioneC();
        public abstract Relazione<A, B> add (A x, B y);
        public abstract Relazione<A, B> remove (A x, B y);
        public abstract LinkedList<A> getUno(B y);
 }

module type RELAZIONE =
 sig type ('a,'b) relazione
  val relazioneC:unit -> ('a,'b) relazione
  val add:  ('a,'b) relazione -> 'a -> 'b -> ('a,'b) relazione
  val remove:  ('a,'b) relazione -> 'a -> 'b -> ('a,'b) relazione
  val getUno:('a,'b) relazione -> 'b -> 'a list
 end;;
```

Methodologies: Abstract Data Types/3 - A Wrong ADT

- An ADT is a sub-Class that implements all the public operations and hides all the data about the structure of the implementation.

Example (A Wrong Definition for an ADT: Where? Why)

```
public abstract class Relazione <A, B>{//One Java API for the type Relazione
         public abstract Relazione<A, B> RelazioneC();
         public abstract Relazione<A, B> add (A x, B y);
         public abstract Relazione<A, B> remove (A x, B y);
         public abstract LinkedList<A> getUno(B y);
  }
public class RelazioneADT1<A,B> extends Relazione<A,B>{//One Java ADT for the API Relazione
      private Vector<A> Left;
      private Vector<B> Right;
      public Relazione<A,B> RelazioneC(){
         Left = new Vector<A>(); Right = new Vector<B>();
         return this;
      }
      public Relazione<A,B> add(A x, B y){
         Left.add(x); Right.add(y);
         return this;
      }
      public Relazione<A,B> remove(A x, B y){
         for(int i=0;left.size();i++){
                  if((Left.get(i)==x) && (Right.get(i)==y)){
                           Left.remove(i); Right.remove(i);
                           return this;}
         return this;
      }
      public LinkedList<A> getUno(B y){...
```

Methodologies: Abstract Data Types/4 - A Wrong ADT

- An ADT is a sub-Class that implements all the public operations and data all the things about the structure of the implementation.

Example (A Different, but equally, wrong ADT)

```
public abstract class Relazione <A, B>{//One Java API for the type Relazione
        public abstract Relazione<A, B> RelazioneC();
        public abstract Relazione<A, B> add (A x, B y);
        public abstract Relazione<A, B> remove (A x, B y);
        public abstract LinkedList<A> getUno(B y);
        }
public class RelazioneADT2<A,B> extends Relazione<A,B>{//One Java ADT for the API Relazione
    private Vector<A> Left;
    private Vector<B> Right;
    public Relazione<A,B> RelazioneC(){
       Relazione<A,B> temp = new Relazione<A,B>();
       temp.Left = new Vector<A>(); temp.Right = new Vector<B>();
       return temp;
    }
    public Relazione<A,B> add(A x, B y){
       LinkedList ListOfx = getDue(x);
       if (ListOfx.contains(y)) return this;
       Left.add(x); Right.add(y);
       return this;
    }
    public Relazione<A,B> remove(A x, B y){...
    public LinkedList<A> getUno(B y){...
```

Methodologies: Code Extension and Reuse

- Many other implementations can be given (for instance, one more that uses one Vector of pairs...)
- Assume we have one ADT for "rel". The ADT is supplied to us in a package of which we use the object code. Hence all we know about it is what is written in the API.
- In this scenario, provide an extension of the ADT for relations that cannot have more than k distinct pairs (see slide 44)
- For this aims, the operator RelazioneC is extended to include the parameter k.

Example (Code Extension and Reuse with ADTs)

```
   public class RelazioneK<A,B> extends RelazioneADT1<A,B>{//Code Extension and Reuse
      int limit, size;
      public Relazione<A,B> RelazioneC(int k){
         super.RelazioneC();
         limit = k; size=0;
      }
** to be completed **
```

Exercise1 Define an ADT, RelazioneADT3<A,B>, that implements relations by using a vector of pair. With this aim, show how the class pair can be defined as a nested class of the ADT.

Abstract Data Types: IMMUTABLE VALUES/3 correct

- However, the ADT RelazioneADT1<A,B> of slide "Abstract Data Types/3" is very strange since it has operations that are producers and modifiers at the same time. The consequence is a lost of values.

Example (API-ADT1 for IMMUTABLE VALUES)

```
public abstract class Relazione <A, B>{//One Java API for the type Relazione
         public abstract Relazione<A, B> RelazioneC();
         public abstract Relazione<A, B> add (A x, B y);
         public abstract Relazione<A, B> remove (A x, B y);
         public abstract LinkedList<A> getUno(B y);
  }
public class RelazioneADT1<A,B> extends Relazione<A,B> {
                          //IMMUTABLE ABSTRACT VALUES: Functional Use of the ADT
      private Vector<A> Left;
      private Vector<B> Right;
      public Relazione<A,B> RelazioneC(){
         RelazioneADT1<A,B> temp = new RelazioneADT1<A,B>();
         temp.Left = new Vector<A>(); temp.Right = new Vector<B>();
         return temp;
      }
      public Relazione<A,B> add(A x, B y){
         RelazioneADT1 <A,B> temp = clone();
         temp.Left.add(x); temp.Right.add(y);
         return temp;
      }
      public Relazione<A,B> remove(A x, B y){...}
      ...
      private RelazioneADT1<A,B> clone(){
         RelazioneADT1<A,B> C = new RelazioneADT1<A,B>();
         C.Left = new Vector<A>();
         ...
```

Abstract Data Types: IMMUTABLE VALUES/4 correct

- Again, the ADT RelazioneADT2<A,B> of slide "Abstract Data Types/4" is wrong since it has operations that are producers and modifiers at the same time. The consequence is a lost of values.

Example (API-ADT2 for IMMUTABLE VALUES - part1)

```
public abstract class Relazione <A, B>{//One Java API for the type Relazione
       public abstract Relazione<A, B> RelazioneC();
       public abstract Relazione<A, B> add (A x, B y);
       public abstract Relazione<A, B> remove (A x, B y);
       public abstract LinkedList<A> getUno(B y);
       }
 public class RelazioneADT2<A,B> extends Relazione<A,B> {
                        //IMMUTABLE ABSTRACT VALUES: Functional Use of the ADT
    private Vector<A> Left;
    private Vector<B> Right;
    public Relazione<A,B> RelazioneC(){
       RelazioneADT1<A,B> temp = new RelazioneADT1<A,B>();
       temp.Left = new Vector<A>(); temp.Right = new Vector<B>();
       return temp;
    }
    public Relazione<A,B> add(A x, B y){
       LinkedList ListOfx = getuno(y);
       if (ListOfx.contains(x)) return this;
       RelazioneADT2 <A,B> temp = clone();
       temp.Left.add(x); temp.Right.add(y);
       return temp;
    }
    public Relazione<A,B> remove(A x, B y){...}
    public LinkedList<A> getUno(B y){...}
    private RelazioneADT2<A,B> clone(){...}
       ...
```

Abstract Data Types: IMMUTABLE VALUES/5 - continued

- Does ADT RelazioneADT2<A,B> need to implement the interface cloneable? Complete.

Example (API-ADT2 for IMMUTABLE VALUES - part2)

```
public class RelazioneADT2<A,B> extends Relazione<A,B> {
                                  //IMMUTABLE ABSTRACT VALUES: Functional Use of the ADT
   private Vector<A> Left; private Vector<B> Right;
   public Relazione<A,B> RelazioneC(){...}
   public Relazione<A,B> add(A x, B y){...}
   public Relazione<A,B> remove(A x, B y){
      RelazioneADT2 <A,B> temp = clone();
      for(int i=0;temp.Left.size();i++){
            if(temp.Left.get(i).equals(x) && temp.Right.get(i).equals(y)){
                  temp.Left.remove(i); temp.Right.remove(i);}
      }
      return temp;
   }
   public LinkedList<A> getUno(B y){
      LinkedList <A> temp = new LinkedList<A>();
      for(int i=0;Right.size();i++){if(Right.get(i).equals(y)) temp.addLast(Left.get(i));}
      return temp;
   }
   private RelazioneADT2<A,B> clone(){
      RelazioneADT2<A,B> C = new RelazioneADT2<A,B>();
      C.Left = new Vector<A>(); C.Right = new Vector<B>();
      for(int i=0;Right.size();i++){
            C.Left(i).addLast(Left.get(i)); C.Right(i).addLast(Right.get(i));
      };
      return C;
   }
```

Abstract Data Types: MUTABLE VALUES

- Java allows API for mutable as well as immutable values
- But Operations are always un-curried
- The API for Mutable Values is below (compare it with the Ocaml API for immutable values)

Example

```
public abstract class Relazione <A,B>{//One Java API for the type Relazione
   public abstract void RelazioneC();
   public abstract void add (A x, B y);
   public abstract void remove (A x, B y);
   public abstract LinkedList<A> getUno(B y);
   }

module type RELAZIONE =      (* Ocaml API for immutable values *)
 sig type ('a,'b) relazione
  val relazioneC:unit -> ('a,'b) relazione
  val add:  ('a,'b) relazione -> 'a -> 'b -> ('a,'b) relazione
  val remove:  ('a,'b) relazione -> 'a -> 'b -> ('a,'b) relazione
  val getUno:('a,'b) relazione -> 'b -> 'a list
 end;;
```

Abstract Data Types: MUTABLE VALUES/2

Example (API-ADT for MUTABLE VALUES)

```
public abstract class Relazione <A,B>{//One Java API for the type Relazione
   public abstract void RelazioneC();
   public abstract void add (A x, B y);
   public abstract void remove (A x, B y);
   public abstract LinkedList<A> getUno(B y);
   }
public class RelazioneADT<A,B> extends Relazione<A,B> {
                          //MUTABLE ABSTRACT VALUES: Procedural Use of the ADT
      private Vector<A> Left;
      private Vector<B> Right;
      public void RelazioneC(){
         Left = new Vector<A>(); Right = new Vector<B>();
      }
      public void add(A x, B y){//why equals instead of ==
         int size = left.size();
         for(int i=0;i<size;i++) if(Left.get(i).equals(x)&&Right.get(i).equals(y)) return;
         Left.add(x); Right.add(y);
      }
      public void remove(A x, B y){
         int size = Left.size();
         for(int i=0;i<size;i++) if(Left.get(i).equals(x)&&(Right.get(i).equals(y)) {
                  Left.remove(i); Right.remove(i);
                  return;}
      }
      public LinkedList <A> getUno(B y){
         LinkedList <A> LeftY=new LinkedList<A>(); int size = Left.size();
         for(int i=0;i<size;i++) if(Right.get(i).equals(y)) LeftY.add(Left.get(i));
         return LeftY;
      }
}
```

Abstract Data Types: MUTABLE VALUES/3

Example (ADT for MUTABLE VALUES - Use of Nested Classes)

```
public class RelazioneADT<A,B> extends Relazione<A,B> {
                          //MUTABLE ABSTRACT VALUES: Procedural Use of the ADT
      private class pair<A,B>{//Fully Abstract Abstractions:  Comment
         A x; B y;
         pair(A x, B y){this.x=x; this.y=y;}
      }
      private Vector<pair<A,B>> rep;
      public void RelazioneC(){
         rep = new Vector<pair<A,B>>();
      }
      public void add(A x, B y){
         int size = rep.size();
         for(int i=0;i<size;i++) if(rep.get(i).x.equals(x)&&rep.get(i).y.equals(y)) return;
         rep.add(new pair<A,B>(x,y));
      }
      public void remove(A x, B y){
         int size = rep.size();
         for(int i=0;i<size;i++) if(rep.get(i).x.equals(x)&&rep.get(i).y.equals(y)) {
                  rep.remove(i); return;}
      }
      public LinkedList <A> getUno(B y){
         LinkedList <A> LeftY=new LinkedList<A>(); int size = rep.size();
         for(int i=0;i<size;i++) if(rep.get(i).y.equals(y)) LeftY.add(rep.get(i).x);
         return LeftY;
      }
}
```

Methodologies: Code Extension and Reuse /2

- Exercise1. Extend the ADT for RelazioneADT<A,B> into an ADT RelazioneOrdered for relations on ordered pairs such that the first member is always not greater than the second one.

Example (ADT for MUTABLE VALUES - Class Extension - Exercise1)

```
public class RelazioneOrdered<C extends Comporable<C>> extends RelazioneADT<C,C> {
                          //MUTABLE ABSTRACT VALUES: Procedural Use of the ADT
      public void RelazioneC(){  //This definition should be omitted since it exactly, computes
         super.RelazioneC();     //as its overriding method
      public void add(C x, C y){
         if(x.compareTo(y)>0) return;
         super.add(x,y);
      }
      //public void remove(C x, C y) is inherited
      //public LinkedList<C> getUno(C y) is inherited
}
```

- Exercise2. Extend the ADT for RelazioneADT<A,B> into an ADT RelazioneK for relations that never contain more than K different pairs: The value of K is stated at the moment of the definition of the relation.
- Exercise3. Extend the ADT for RelazioneADT<A,B> into an ADT MonoFun<A,B> for unary functions
- Exercise4. Extend the ADT MonoFun<A,B> into an ADT MonoInvFun for unary, invertible functions

Methodologies: Code Extension and Reuse /3

Example (ADT for MUTABLE VALUES - Class Extension - Exercise2)

```
public class RelazioneK<A,B> extends RelazioneADT<A,B> {
                          //MUTABLE ABSTRACT VALUES: Procedural Use of the ADT
      private class ThePair<A,B>{//A different class for pairs
         A a; B b;
         ThePair(A x, B y){a=x; b=y;}
      }
      private int limit; int size; Vector<ThePair<A,B>> repK;
      public void RelazioneC(int k){
         super.RelazioneC();
         limit=k; size=0;
         repK = new Vector<ThePair<A,B>>();
      }
      public void add(A x, B y){
         for(int i=0;i<size-1;i++) if(repK.get(i).a.equals(x)&&repK.get(i).b.equals(y)) return;
         if(size=limit)return;
         size++;
         super.add(x,y);
         repK.add(new pair<A,B>(x,y));
      }
      public void remove(A x, B y){
         super.remove(x,y);
         for(int i=0;i<size;i++) if(rep.get(i).a.equals(x)&&rep.get(i).b.equals(y)) {
                  repK.remove(i); size--; return;}
      }
      public LinkedList<A> getUno(B y){//it can be omitted since inheritance and scope rule
         return super.getUno(y)
      }
}
```

Methodologies: Code Extension and Reuse /4

- A different, more compact, less time-space expensive, solution is based on the analysis of the properties of the methods that the super-class makes public and in the definition of a private predicate method isIn.

Example (ADT for MUTABLE VALUES - Class Extension - Exercise2: A different solution)

```
public class RelazioneK<A,B> extends RelazioneADT<A,B> {
                          //MUTABLE ABSTRACT VALUES: Procedural Use of the ADT
      private int limit; int size;
      public void RelazioneC(int k){
         limit=k; size=0;
         super.RelazioneC();
      }
      public void add(A x, B y){
         if ((!isIn(x,y))||size==limit) return;
         size++;
         super.add(x,y);
      }
      public void remove(A x, B y){
         if (isIn(x,y)) return
         size--;
         super.remove(x,y);
      }
      private boolean isIn(A x, B y){//it can be implemented by a suitable use of method getUno
         return getUno(y).contains(x);
      }
}
```

Methodologies: Code Extension and Reuse /5

- The solution is based on the analysis of the properties of the methods that the super-class makes public and in the store of the function definition domain.

Example (ADT for MUTABLE VALUES - Class Extension - Exercise3)

```
public class MonoFun<A,B> extends RelazioneADT<A,B> {
                          //MUTABLE ABSTRACT VALUES: Procedural Use of the ADT
      private LinkedList<A> Dom;
      public void RelazioneC(){
         Dom = new LinkedList<A>();
         super.RelazioneC();
      }
      public void add(A x, B y){
         if (isIn(x,y)||isInDom(x)) return;
         Dom.add(x);
         super.add(x,y);
      }
      public void remove(A x, B y){
         if (!isIn(x,y)) return
         Dom.remove(x);
         super.remove(x,y);
      }
      private boolean isIn(A x, B y){//implemented by using method getUno
         return getUno(y).contains(x);
      }
      private boolean isInDom(A x){
         return Dom.contains(x);
      }
}
```

Methodologies: Code Extension and Reuse /6

- The solution can't access the private structure of the super MonoFun.
- The solution makes use of the analysis of the properties of the super MonoFun: It checks the size of getUno(y) before adding any pair (x,y).

Example (ADT for MUTABLE VALUES - Class Extension - Exercise4)

```
public class MonoInvFun<A,B> extends MonoFun<A,B> {
                          //MUTABLE ABSTRACT VALUES: Procedural Use of the ADT
      public void add(A x, B y){
         if (isInImg(y)) return;
         super.add(x,y);
      }
      private boolean isInImg(B y){
         return getUno(y).size()!=0;
      }
}
```