MI-GLSD-M1 -UEM213 :
Programming paradigms

Chapter V: Oriented Objects Paradigm

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of sciences & technology
Mathematics & computer sciences department

a.hariche@univ-dbkm.dz

This Course Inspiration

- Louv1.2x is the successor to Louv1.1x
  - We assume that you understand the concepts and notation of Louv1.1x (Oz language)

- Louv1.2x continues the story with three topics
  - Data abstraction and state
  - Concurrent programming
  - Programming paradigms

  } Essential concepts for programs in the real world

- Practical organization
  - 7 lessons, homework exercises, final exam
  - Exercises in mozart and graded by INGInious

Intelligent grader!
CorrectOz

HARICHE A. a.hariche@univ-dbkm.dz

Louv1.2x course organisation for 2018

- Certificate
  - Choose Verified Certificate (with donation) or Audit (no certificate)
- Lessons
  - Seven lessons (6 + 1 bonus); one lesson per week
  - First lesson Nov. 5, seventh (last) lesson Dec. 17
- Weekly exercises (50% of grade)
  - Conceptual exercises (multiple choice + fill in blanks)
  - Programming exercises (using Mozart and INGInious with CorrectOz)
  - One week deadline + two-week grace period
  - Infinite number of tries per exercise
- Final exam (50% of grade)
  - Starts Jan. 7, final due date Jan. 21
  - Two tries per exercise

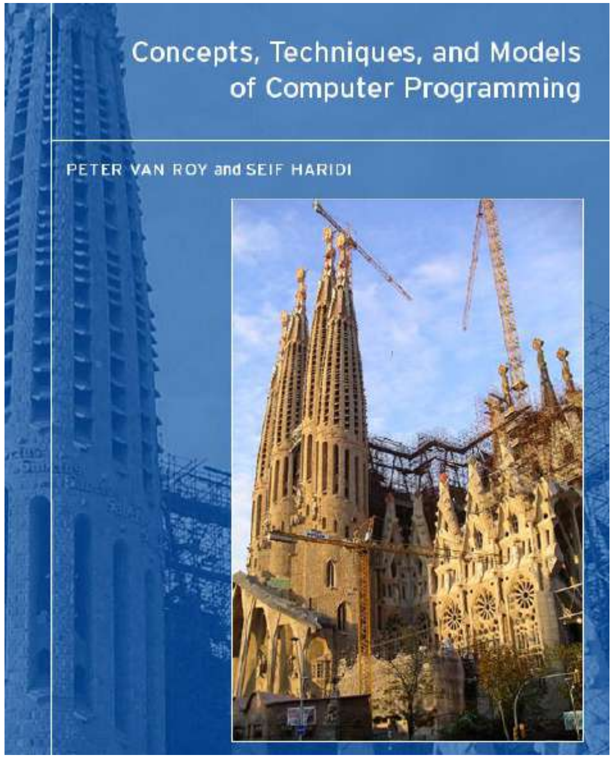
Schedule

- 1. Explicit state and data abstraction
- 2. Object-oriented programming
- 3. Java, multiple inheritance, and exceptions
- 4. Deterministic dataflow introduction
- 5. Deterministic dataflow techniques
- 6. Multiagent dataflow programming (bonus)
- 7. Paradigms of programming redux
- (two-week break for end-of-year festivities)
- Final exam
- End of course

Another Paradigm
Explore more

- Lesson 6 is a bonus lesson on multiagent dataflow programming (a.k.a. actor dataflow)
  - Multiagent dataflow extends deterministic dataflow with the ability to add nondeterminism exactly where needed
  - Multiagent dataflow is the best all round paradigm for concurrent programming that we know
- Lesson 6 will be given by Seif Haridi
  - Seif Haridi is professor at the Royal Institute of Technology in Stockholm and chief scientist at the Swedish Institute of Computer Science

Course textbook and handouts



- “Concepts, Techniques, and Models of Computer Programming” by Peter Van Roy and Seif Haridi, MIT Press
  - Same book for Louv1.1x and Louv1.2x
  - Each course sees 25% of the book
- MIT Press has made available part of the book for the course
  - Chapters 1-3
- This is complemented by slides and the last public draft of the book
  - Important for abstraction and concurrency

Abstraction and concurrency

- Essential concepts for building large programs that are part of the real world

- Data abstraction is the main organizing principle for building complex software systems
  - The real world is complex
- Explicit state allows to model change in a program
  - The real world has change
  - Explicit state supports data abstraction

- Concurrency is a property of systems that are made of activities that progress independently
  - The real world has independent activities
- Deterministic dataflow is a form of concurrency that always gives the same outputs for the same inputs

Paradigms of Louv1.1x and Louv1.2x

Louv1.1x

Functional programming
Functions and recursion
Higher-order programming
Single-assignment variables

+ state (cells)

+ concurrency (threads)

Lessons 1, 2, 3

Object-oriented programming
Data abstraction
Polymorphism
Inheritance

Lessons 4, 5

Deterministic dataflow
No race conditions
Concurrency transparency
Streams and agents

+ ports (named streams)

Lesson 6

Multi-agent dataflow
Extends deterministic dataflow
Nondeterminism where needed

Lesson 7

Active objects
Object-oriented programming
Multi-agent dataflow

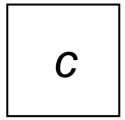
- We have seen one paradigm in Louv1.1x
- We will see four more in Louv1.2x
- Each paradigm has its own concepts and kernel language

Let's start

- This first lesson will introduce explicit state and data abstraction
  - You will understand exactly what explicit state adds to functional programming
  - You will see the four fundamentally different ways of building data abstractions
    - Objects as seen in Java or C++ are just one way
- This leads to our second paradigm, namely object-oriented programming

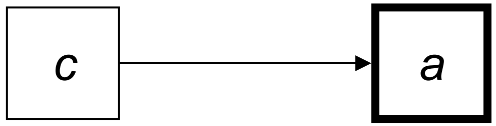
Adding explicit state to the language

- We can make the state explicit by extending the language
- With this extension a program can directly observe the sequence of values in time
  - This was not possible in the functional paradigm
- We call our extension a cell
  - The word “cell” is chosen to avoid confusion with related terms, such as the overused word “variable”
- A cell is a box with a content
  - The content can be changed but the box remains the same
  - The same cell can have different contents: we can observe change
  - The sequence of contents is a state



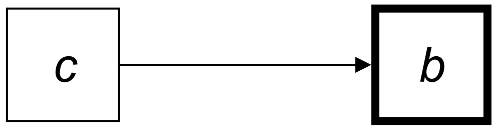
An unbound variable

Creating a cell with initial content *a* (=5)



cell

Replace the content by another variable *b* (=6)



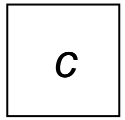
cell

HARICHE A. a.hariche@univ-dbkm.dz

A cell

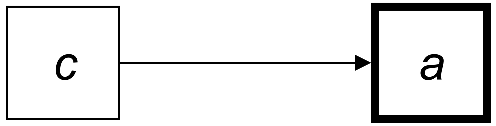
- A cell is a box with an identity and a content
  - The identity is a constant
    (the “name” or “address” of the cell)
  - The content is a variable
    (in the single-assignment store)
- The content can be replaced by another variable

```
A=5
B=6
C={NewCell A}  % Create a cell
{Browse @C}    % Display content
C:=B           % Change content
{Browse @C}    % Display content
```

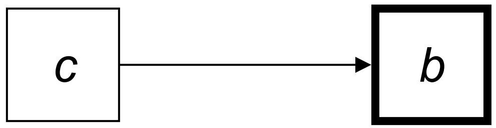


An unbound variable

Creating a cell with initial content *a* (=5)



Replace the content by another variable *b* (=6)



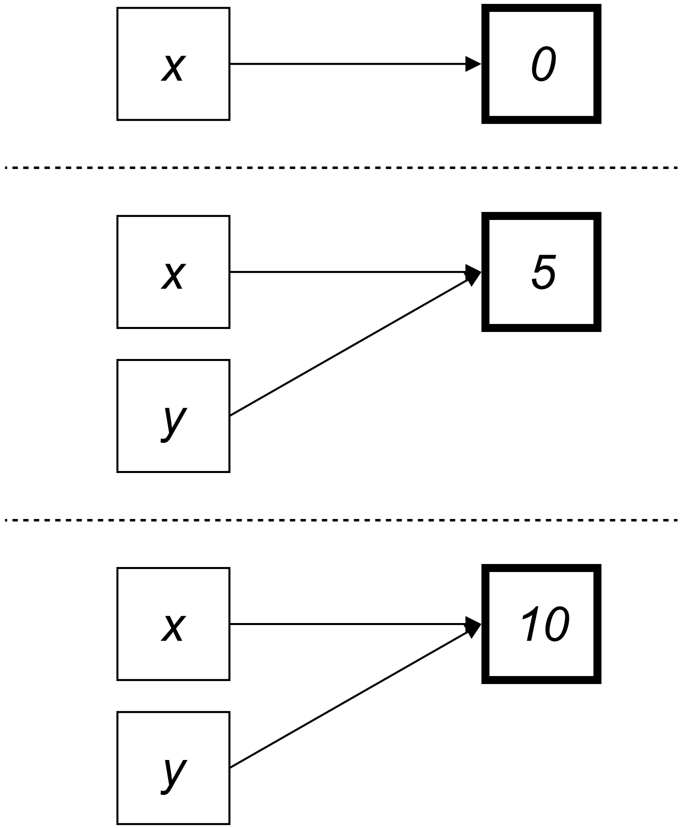
HARICHE A. a.hariche@univ-dbkm.dz

Adding cells to the kernel language

- We add cells and their operations
  - Cells have three operations
- C={NewCell A}
  - Create a new cell with initial content A
  - Bind C to the cell's identity
- C:=B
  - Check that C is bound to a cell's identity
  - Replace the cell's content by B
- Z=@C
  - Check that C is bound to a cell's identity
  - Bind Z to the cell's content

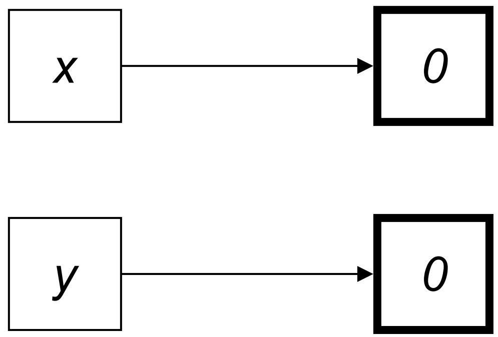
Some examples (1)

- X={NewCell 0}
- X:=5
- Y=X
- Y:=10
- @X==10 % true
- X==Y % true



Some examples (2)

- X={NewCell 0}
- Y={NewCell 0}



- X==Y % false
- Because X and Y refer to different cells, with different identities

- @X==@Y % true
- Because the contents of X and Y are the same value

Time and change

- In the functional paradigm, there is no notion of time
  - All functions are mathematical functions; once defined they never change
  - Programs do execute on a real machine, but a program cannot observe the execution of another program or of part of itself
    - It can only see the results of a function call, not the execution itself
    - Observing an execution of a program can only be done outside of the program's implementation
- In the real world, there is time and change
  - Organisms change their behavior over time, they grow and learn
  - How can we model this in a program?
- We need to add time to a program
  - Time is a complicated concept! Let us start with a simplified version of time, an abstract time, that keeps the essential property that we need: modeling change.

State as an abstract time (1)

- Here's one solution: We define the abstract time as a sequence of values and we call it a state
- A state is a sequence of values calculated progressively, which contains the intermediate results of a computation
- The functional paradigm can use state according to this definition!
- The definition of Sum given here has a state

```
fun {Sum Xs A}
   case Xs
   of nil then A
   [] X|Xr then
        {Sum Xr A+X}
   end
end

{Browse {Sum [1 2 3 4] 0}}
```

State as an abstract time (2)

- The two arguments Xs and A give us an implicit state

Xs	A
[1 2 3 4]	0
[2 3 4]	1
[3 4]	3
[4]	6
nil	10

- It is implicit because the language has not changed
  - It is purely in the programmer's head: the programmer observes the changes in the program
- In most cases this is not good enough: we want the program itself to observe the changes
  - We need a language extension!
  - *We leave the functional paradigm and enter another paradigm*

```
fun {Sum Xs A}
   case Xs
   of nil then A
   [] X|Xr then
         {Sum Xr A+X}
   end
end

{Browse {Sum [1 2 3 4] 0}}
```

Structure equality and token equality

- Two lists are equal if their values are equal (structure equality)
  - Two structures with same values created separately are equal
  - A=[1 2]
    B=[1 2]
    {Browse A==B} % true
- Two cells are equal if they are the same cell (token equality)
  - Two cells created separately are always different
  - C={NewCell [1 2]}
    D={NewCell [1 2]}
    {Browse C==D} % false
    {Browse @C==@D} % true (Since the contents are lists, they are compared with structure equality)

Semantics of cells (1)

- We have extended the kernel language with cells
  - Let us now extend the abstract machine to explain how cells execute
- There are now two stores in the abstract machine:
  - Single-assignment store (contains variables: immutable store)
  - Multiple-assignment store (contains cells: mutable store)
- A cell is a pair of two variables
  - The first variable is bound to the name of the cell (a constant)
  - The second variable is the cell's content
- Assigning a cell to a new content
  - The pair is changed: the second variable in the pair is replaced by another variable (the first variable stays the same)

Warning: The variables do *not* change! The single-assignment store is unchanged when a cell is assigned.

Semantics of cells (2)

- The full store $\sigma = \sigma_1 \cup \sigma_2$ has two parts:
  - Single-assignment store (contains variables)
    $\sigma_1 = \{t, u, v, x=\xi, y=\zeta, z=10, w=5\}$
  - Multiple-assignment store (contains pairs)
    $\sigma_2 = \{x{:}t, y{:}w\}$

- In σ_2 there are two cells, *x* and *y*
  - The name of *x* is the constant ξ, the name of *y* is ζ
  - The operation X:=Z changes *x*:*t* into *x*:*z*
  - The operation @Y returns the variable *w*
    (assuming the environment $\{X \rightarrow x, Y \rightarrow y, Z \rightarrow z, W \rightarrow w\}$)

Imperative paradigm

- By adding cells, we have left the functional paradigm and entered the imperative paradigm
  - Imperative paradigm = functional paradigm + cells
- The imperative paradigm allows programs to express and observe growth and change
  - This gives new ways of thinking that were not possible in the functional paradigm
- The imperative paradigm is the foundation of object-oriented programming (OOP)
  - OOP has new ways of structuring programs that are essential for building large systems

Kernel language of the imperative paradigm

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {NewCell <y> <x>}
  | <x>:=<y>
  | <y>=@<x>
- <v> ::= <number> | <procedure> | <record>
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <record>, <p> ::= <lit> | <lit>(<f>$_1$:<x>$_1$ … <f>$_n$:<x>$_n$)

Kernel language of the imperative paradigm

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {NewCell <y> <x>}
  | {Exchange <x> <y> <z>}
- <v> ::= <number> | <procedure> | <record>
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <record>, <p> ::= <lit> | <lit>(<f>$_1$:<x>$_1$ … <f>$_n$:<x>$_n$)

Second version
Both versions are equally expressive (since Exchange can be expressed with @ and := and vice versa), but the second version is more convenient for concurrent programming

<y>=@<x> and <x>:=<z>
(*atomically* : as one operation)

Explicit state is useful for modularity

- Before looking at data abstraction and object-oriented programming, let's take a closer look at what explicit state is good for
- We say that a program (or system) is modular with respect to a given part if that part can be changed without changing the rest of the program
  - "part" = function, procedure, component, module, class, library, package, file, …
- We will show by means of an example that the use of explicit state allows us to make a program modular
  - This is not possible in the functional paradigm

A scenario (1)

- Once upon a time there were three developers, P, U1, and U2
- P has developed module M that implements two functions F and G
- U1 and U2 are both happy users of module M

```
fun {MF}  % Module definition
   fun {F ...}
      ⟨Definition of F⟩
   end
   fun {G ...}
      ⟨Definition of G⟩
   end
in 'export'(f:F g:G)
end
M = {MF} % Module instantiation
```

A scenario (2)

- One day, developer U2 writes an application that runs slowly because it does too much computation
- U2 would like to extend M to count the number of times F is called by the application
- U2 asks P to make this extension, but to keep it modular so that no programs have to be changed to use it

```
fun {MF}
   fun {F ...}
      ⟨Definition of F⟩
   end
   fun {G ...}
      ⟨Definition of G⟩
   end
in 'export'(f:F  g:G)
end
M = {MF}
```

Oops!

- This is impossible in the functional paradigm, because F does not remember what happened in previous calls: it cannot count its calls
  - The only solution is to change the interface of F by adding two arguments, F_{in} and F_{out}:
    fun {F … F_{in} F_{out}} F_{out}=F_{in}+1 … **end**
  - The rest of the program has to make sure that the F_{out} of each call to F is passed as F_{in} to the next call of F
- This means that M's interface has changed
- All M's users, even U1, have to change their programs
  - U1 is especially unhappy, since it makes a lot of extra work for nothing

Solution using a cell

- Create a cell when MF is called and increment it inside F
  - Because of static scope, the cell is hidden from the rest of the program: it is only visible inside M
- M's interface is extended without changing existing calls
  - M.f stays the same
  - A new function M.c appears that can safely be ignored
- P, U1, and U2 live happily ever after

```
fun {MF}
   X = {NewCell 0}
   fun {F ...}
      X:=@X+1
      ⟨Definition of F⟩
   end
   fun {G ...}
      ⟨Definition of G⟩
   end
   fun {Count} @X end
in 'export'(f:F g:G c:Count)
end
M = {MF}
```

Comparison

- Functional paradigm:
  - \+ A component never changes its behavior (if it is correct, it stays correct)
  - – Updating a component often means that its interface changes and therefore many other components must be updated
- Imperative paradigm:
  - \+ A component can be updated without changing its interface and so without changing the rest of the program (modularity)
  - – A component can change its behavior because of past calls (for example, it might break)

- Sometimes it is possible to combine both advantages
  - Use explicit state to manage updates, but make sure that the behavior of components does not change

Data abstraction

- Data abstraction is the main organizing principle for building complex software systems
  - Without data abstraction, computing technology would stop dead in its tracks
- We will study what data abstraction is and how it is supported by the programming language
  - The first step toward data abstraction is called encapsulation
  - Data abstraction is supported by language concepts such as higher-order programming, static scoping, and explicit state

Encapsulation

- The first step toward data abstraction, which is the basic organizing principle for large programs, is encapsulation
- Assume your television set is not enclosed in a box
  - All the interior circuitry is exposed to the outside
  - It's lighter and takes up less space, so it's good, right? NO!
- It's dangerous for you: if you touch the circuitry, you can get an electric shock
- It's bad for the television set: if you spill a cup of coffee inside it, you can provoke a short-circuit
  - If you like electronics, you may be tempted to tweak the insides, to "improve" the television's performance
- So it can be a good idea to put the television in an enclosing box
  - A box that protects the television against damage and that only authorizes proper interaction (on/off, channel selection, volume)

Encapsulation in a program

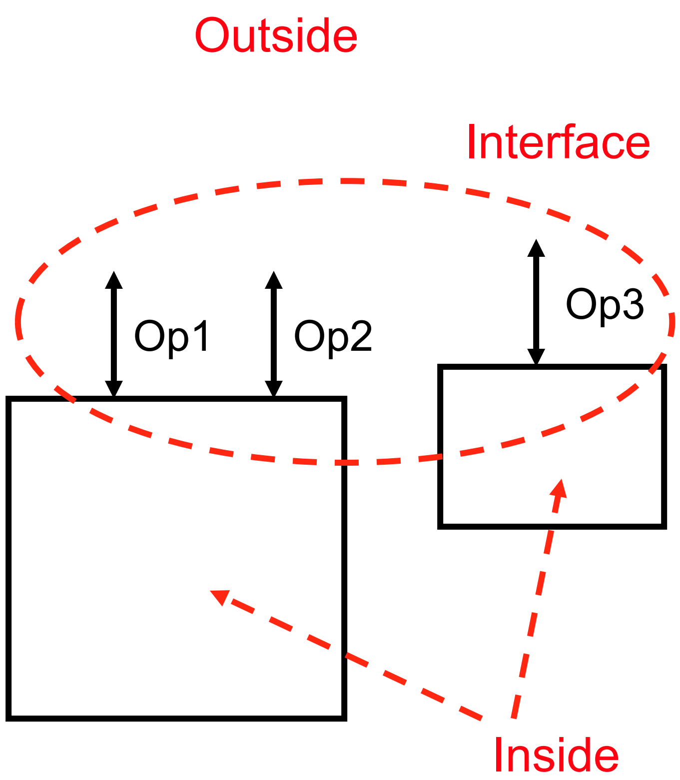
- Assume your program uses a stack with the following implementation:

```
fun {NewStack} nil end
fun {Push S X} X|S end
fun {Pop S X} X=S.1 S.2 end
fun {IsEmpty S} S==nil end
```

- This implementation is not encapsulated!
  - It has the same problems as a television set without enclosure
  - It is implemented using lists that are not protected
    - A user can read stack values without the implementation knowing
    - A user can create stack values outside of the implementation
- There is no way to guarantee that an unencapsulated stack will work correctly
  - The stack must be encapsulated → data abstraction

Definition of data abstraction

- A data abstraction is a part of a program that has an inside, an outside, and an interface in between
- The inside is hidden from the outside
  - All operations on the inside must pass through the interface, i.e., the data abstraction must use encapsulation
- The interface is a set of operations that can be used according to certain rules
  - Correct use of the rules guarantees that the results are correct
- The encapsulation must be supported by the programming language
  - We will see how the language can support encapsulation, that is, how it can enforce the separation between inside and outside



Advantages of data abstraction

- A guarantee that the abstraction will work correctly
  - The interface only allows well-defined interaction with the inside
- A reduction of complexity
  - The user does not have to know the implementation, but only the interface, which is generally much simpler
  - A program can be partitioned into many independent abstractions, which greatly simplifies use
- The development of large programs becomes possible
  - Each abstraction has a responsible developer: the person who implements it, maintains it, and guarantees its behavior
  - Each responsible developer only has to know the interfaces of the abstractions used by the abstraction
  - It’s possible for teams of developers to develop large programs

The two main kinds of data abstraction

- There are two main kinds of data abstraction, namely objects and abstract data types
  - An object groups together value and operations in a single entity
  - An abstract data type keeps values and operations separate
- Some real world examples
  - A television set is an object: it can be used directly through its interface (on/off, channel selection, volume control)
  - Coin-operated vending machines are abstract data types: the coins and products are the values and the operations are the vending machines
- We will look at both objects and ADTs
  - Each has its own advantages and disadvantages

Abstract data types

- An ADT consists of a set of values and a set of operations
- A common example: integers
  - Values: 1, 2, 3, …
  - Operations: +, -, *, div, …
- In most of the popular uses of ADTs, the values and operations have no state
  - The values are constants
  - The operations have no internal memory (they don't remember anything in between calls)

A stack ADT

- We can implement a stack as an ADT:
  - Values: all possible stacks and elements
  - Operations: NewStack, Push, Pop, IsEmpty
- The operations take (zero or more) stacks and elements as input and return (zero or more) stacks and elements as output
  - S={NewStack}
  - S2={Push S X}
  - S2={Pop S X}
  - {IsEmpty S}
- For example:
  - S={Push {Push {NewStack} a} b} returns the stack S=[b a]
  - S2={Pop S X} returns the stack S2=[a] and the top X=b

Unencapsulated implementation

- The stack we saw before is almost an ADT:

  - **fun** {NewStack} nil **end**
  - **fun** {Push S X} X|S **end**
  - **fun** {Pop S X} X=S.1 S.2 **end**
  - **fun** {IsEmpty S} S==nil **end**

- Here the stack is represented by a list
- But this is not a data abstraction, since the list is not protected

- How can we protect the list, and make this a true ADT?
  - How can we build an abstract data type with encapsulation?
  - We need a way to protect values

Encapsulation using a secure wrapper

- To protect the values, we will use a secure wrapper:
  - The two functions Wrap and Unwrap will “wrap” and “unwrap” a value
  - W={Wrap X}   % Given X, returns a protected version W
  - X={Unwrap W}   % Given W, returns the original value X
- The simplest way to understand this is to consider that Wrap and Unwrap do encryption and decryption using a shared key that is only known by them
- We need a new Wrap/Unwrap pair for each ADT that we want to protect, so we use a procedure that creates them:
  - {NewWrapper Wrap Unwrap} creates the functions Wrap and Unwrap
  - Each call to NewWrapper creates a pair with a new shared key
- We will not explain here how to implement NewWrapper, but if you are curious you can look in the book (Section 3.7.5)

Implementing the stack ADT

- Now we can implement a true stack ADT:

```
local Wrap Unwrap in
      {NewWrapper Wrap Unwrap}

      fun {NewStack} {Wrap nil} end
      fun {Push W X} {Wrap X|{Unwrap W}} end
      fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
      fun {IsEmpty W} {Unwrap W}==nil end
end
```

- How does this work? Look at the Push function: it first calls {Unwrap W}, which returns a stack value S, then it builds X|S, and finally it calls {Wrap X|S} to return a protected result
- Wrap and Unwrap are hidden from the rest of the program (static scoping)

Final remarks on ADTs

- ADT languages have a long history
  - The language CLU, developed by Barbara Liskov and her students in 1974, is the first
  - This is only a little bit later than the first object-oriented language Simula 67 in 1967
  - Both CLU and Simula 67 strongly influenced later object-oriented languages up to the present day
- ADT languages support a protection concept similar to Wrap/Unwrap
  - CLU has syntactic support that makes the creation of ADTs very easy
- Many object-oriented languages also support ADTs
  - For example, we will see that Java objects are also ADTs

Objects

- A single object represents both a value and a set of operations
- Example interface of a stack object:

```
S={NewStack}
{S push(X)}
{S pop(X)}
{S isEmpty(B)}
```

- The stack value is stored inside the object S
- Example use of a stack object:

```
S={NewStack}
{S push(a)}
{S push(b)}
local X in {S pop(X)} {Browse X} end
```

Implementing the stack object

- Implementation of the stack object:

```
fun {NewStack}
        C={NewCell nil}
        proc {Push X} C:=X|@C end
        proc {Pop X} S=@C in C:=S.2 X=S.1 end
        proc {IsEmpty B} B=(@C==nil) end
in
        proc {$ M}
            case M of push(X) then {Push X}
            [] pop(X) then {Pop X}
            [] isEmpty(B) then {IsEmpty B} end
        end
end
```

- Each call to NewStack creates a new stack object
- The object is represented by a one-argument procedure that does procedure dispatching: a case statement chooses the operation to execute
- Encapsulation is enforced by hiding the cell with static scoping

Stack as ADT and stack as object

- Here is the stack as ADT:

```
local Wrap Unwrap in
        {NewWrapper Wrap Unwrap}
        fun {NewStack} {Wrap nil} end
        fun {Push W X} {Wrap X|{Unwrap W}} end
        fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
        fun {IsEmpty W} {Unwrap W}==nil end
end
```

- Here is the stack as object: (represented by a record)

```
fun {NewStack}
        C={NewCell nil}
        proc {Push X} C:=X|@C end
        proc {Pop X} S=@C in X=S.1 C:=S.2 end
        fun {IsEmpty} @C==nil end
in
        stack(push:Push pop:Pop isEmpty:IsEmpty)
end
```

- Any data abstraction can be implemented as an ADT or as an object

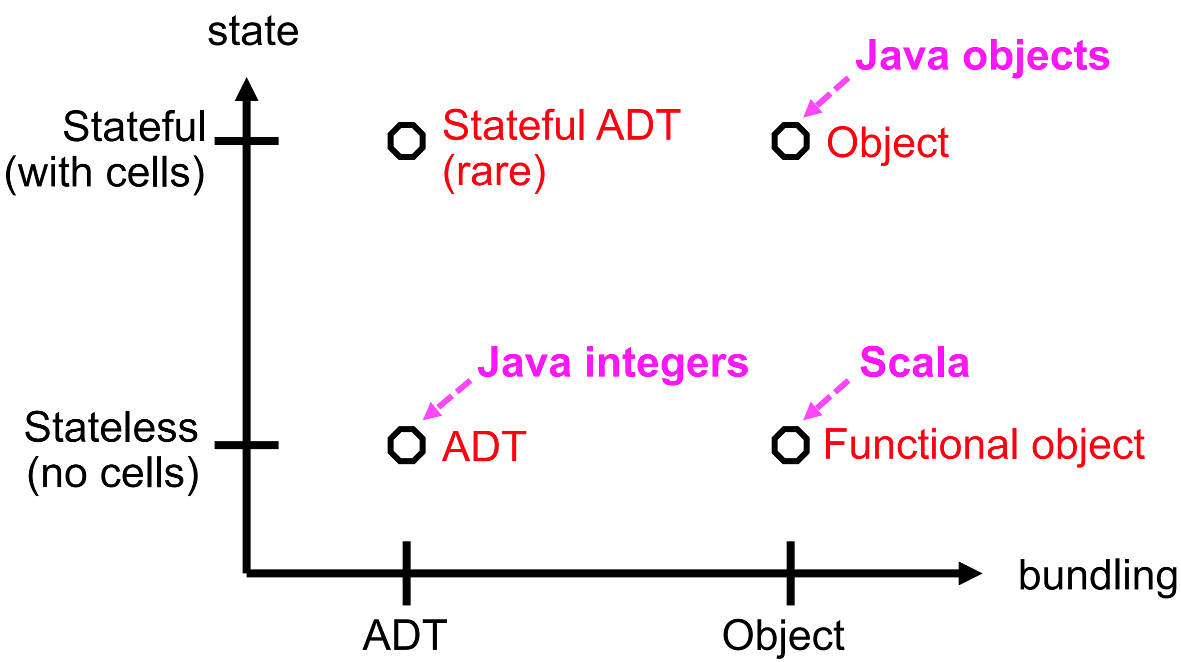
Final remarks on objects

- Objects are omnipresent in computing today
- The first major object-oriented language was Simula-67, introduced in 1967
  - It directly influenced Smalltalk (starting in 1971) and C++ (starting in 1979), and through them, most modern object-oriented languages (Java, C#, Python, Ruby, and so forth)
- Most modern OO languages are in fact data abstraction languages: they incorporate both objects and ADTs
  - And other data abstraction concepts as well, such as components and modules
- The next lesson will be completely focused on object-oriented programming

Four ways to do data abstraction

- We have seen two ways to make data abstractions:
  - Abstract data types (without state)
  - Objects (with state)
- There are two more ways to build data abstractions
  - Abstract data types with state (stateful ADTs)
  - Objects without state (functional objects)
- This gives four ways in all
  - Let's take a look at the two additional ways
  - And then we'll conclude this lesson on data abstraction

Four ways to do data abstraction



- Objects (with state) and ADTs (stateless) are popular
- Functional objects are less popular (except in Scala)
- Stateful ADTs are rarely used

The two less-used data abstractions

- A functional object is possible
  - Functional objects are immutable; invoking an object returns another object with a new value
  - Functional objects are becoming more popular because of Scala
- A stateful ADT is possible
  - Stateful ADTs were much used in the C language (although without enforced encapsulation, since it is impossible in C)
  - They are also used in other languages (e.g., classes with static attributes in Java)
- Let's take a closer look at how to build them

A functional object

- We can implement the stack as a functional object:

```
local
   fun {StackObject S}
      fun {Push E} {StackObject E|S} end
      fun {Pop S1}
         case S of X|T then S1={StackObject T} X end end
      fun {IsEmpty} S==nil end
   in stack(push:Push pop:Pop isEmpty:IsEmpty) end
in
   fun {NewStack} {StackObject nil} end
end
```

- This uses no cells and no secure wrappers. It's the simplest of all our data abstractions since it only needs higher-order programming.

Functional objects in Scala

- Scala is a hybrid functional-object language: it supports both the functional and object-oriented paradigms
- In Scala we can define an immutable object that returns another immutable object
  - For example, a RationalNumber class whose instances are rational numbers (and therefore immutable)
  - Adding two rational numbers returns another rational number
- Immutable objects are functional objects
  - The advantage is that they cannot be changed (the same advantage of any functional data structure)

A stateful ADT

- Finally, let us implement our trusty stack as a stateful ADT:

```
local Wrap Unwrap
   {NewWrapper Wrap Unwrap}
   fun {NewStack} {Wrap {NewCell nil}} end
   proc {Push S E} C={Unwrap S} in C:=E|@C end
   fun {Pop S} C={Unwrap S} in
      case @C of X|S1 then C:=S1 X end
   end
   fun {IsEmpty S} @{Unwrap S}==nil end
in
   Stack=stack(new:NewStack push:Push pop:Pop isEmpty:IsEmpty)
end
```

- This uses both a cell and a secure wrapper. Note that Push, Pop, and IsEmpty do not need Wrap! They modify the stack state by updating the cell *inside* the secure wrapper.

Conclusion

- Data abstractions are a key concept needed for building large programs with confidence
  - Data abstractions are built on top of higher-order programming, static scoping, explicit state, records, and secret keys
  - Data abstractions are defined precisely in terms of these concepts; our definitions give the semantics of data abstractions
- There are four kinds of data abstraction, along two axes: objects versus ADTs on one axis and stateful versus stateless on the other
  - Two kinds are more visible than the others, but the others also have their uses (for example, functional objects are used in Scala)
- Modern programming languages strongly support data abstractions
  - They support much more than just objects; it is more correct to consider them data abstraction languages and not just object-oriented languages

Object-oriented programming

- The concept of object is omnipresent in programming languages today
  - A simple idea: a data abstraction that contains both value and operations
  - First major system was Simula 67, widely disseminated via Smalltalk et C++
- Caveat: object-oriented programming has become a buzzword
  - There are many variations, but not always correct (e.g., many so-called OO languages do not provide proper encapsulation, like C++ and Javascript, or do not properly support inheritance with the substitution principle, like C++)
  - It is not always the right paradigm, e.g., Erlang is better for fault tolerance
  - We will try to be as rigorous as possible, and focus on the main principles
- OOP provides three main principles for structuring programs:
  - Data abstraction: provide guarantees and reduce complexity
  - Polymorphism: compartmentalize responsibility
  - Inheritance: avoid redundancy and encourage incremental development
- Abstract data types are just as omnipresent!
  - It is important to understand both objects and ADTs, because languages mix the two. For example, a Java object is a mix of pure objects and pure ADTs.
  - Advanced object-oriented languages are actually data abstraction languages

An object

```
declare
local
   A1={NewCell I1}
   …
   An={NewCell In}
in
   proc {M1 …} … end
   …
   proc {Mm …} … end
end
```

This code gives the structure of an object abstraction.

An object is a combination of local cells A1, …, An and global procedures M1, …, Mm.

We call A1, …, An the "attributes" and M1, …, Mm the "methods".

Attributes A1, ..., An are *hidden* from the outside and methods M1, ..., Mm are *visible* from the outside (interface!).

A counter object

```
declare
local
   A1={NewCell 0}
in
   proc {Inc} A1:=@A1+1 end
   proc {Get X} X=@A1 end
end

{Inc}
local X in {Get X} {Browse X} end
```

This code creates one object that implements a counter.

The object has two methods, Inc and Get, and is initialized to 0.

Since the cell can only be accessed by the methods, the behavior is guaranteed correct: {Get X} binds X to an integer that gives the number of calls {Inc} done before.

Adding abilities to objects in four steps

- Objects in OOP are much more than simple data abstractions: they add important abilities needed for practical programming

- Let us start with an object abstraction and extend it in four steps:
  - First step: a single object (data abstraction)
  - Second step: a single entry point (dispatch)
  - Third step: creating multiple objects (instantiation)
  - Fourth step: specialized syntax (classes)

First step: An object

```
declare
local
   A1={NewCell I1}
   …
   An={NewCell In}
in
   proc {M1 …} … end
   …
   proc {Mm …} … end
end
```

This code gives the structure of an object abstraction.

An object is a combination of local cells A1, …, An and global procedures M1, …, Mm.

We call A1, …, An the "attributes" and M1, …, Mm the "methods".

Attributes A1, ..., An are *hidden* from the outside and methods M1, ..., Mm are *visible* from the outside (interface!).

First step: An object

```
declare
local
   A1={NewCell 0}
in
   proc {Inc} A1:=@A1+1 end
   proc {Get X} X=@A1 end
end

{Inc}
local X in {Get X} {Browse X} end
```

This code creates one object that implements a counter.

The object has two methods, Inc and Get, and is initialized to 0.

Since the cell can only be accessed with the methods, the behavior is guaranteed correct: {Get X} binds X to an integer that gives the number of calls {Inc} done before.

Second step: Single entry point

```
declare
local
   A1={NewCell 0}
   proc {Inc} A1:=@A1+1 end
   proc {Get X} X=@A1 end
in
   proc {Counter M}
         case M of inc then {Inc}
         [] get(X) then {Get X}
         end
   end
end
```

This extends the counter object to invoke all methods from a single entry point: the procedure Counter.

{Counter inc}

{Counter inc}

{Counter get(X)}

In this example, this is called procedure dispatch, since the entry point is a procedure. The argument M is usually called a message.

Third step:
Creating multiple objects

```
declare
fun {NewCounter}
    A1={NewCell 0}
    proc {Inc} A1:=@A1+1 end
    proc {Get X} X=@A1 end
in
    proc {$ M}
        case M of inc then {Inc}
        [] get(X) then {Get X}
        end
    end
end
```

We add the ability to create many counter objects with the same methods but different states.

The function NewCounter creates a new counter object each time it is called. This is an example of instantiation (higher-order programming).

The call C={NewCounter} creates a new cell in A1 and returns an object with methods Inc and Get, that both access the new cell.

Each new object is completely independent of the others.

Using NewCounter

```
C1={NewCounter} % First object
C2={NewCounter} % Second object

{C1 inc} % Increment first object twice
{C1 inc}

local X in {C1 get(X)} {Browse X} end % Shows 2
local X in {C2 get(X)} {Browse X} end % Shows 0
```

Fourth step: Specialized syntax

```
class Counter
   attr a1
   meth init a1:=0 end
   meth inc a1:=@a1+1 end
   meth get(X) X=@a1 end
end

C1={New Counter init}
{C1 inc}
local X in
   {C1 get(X)} {Browse X}
end
```

We introduce a new syntax for defining objects, in which we define attributes and methods.

We call this definition a class, since we can use it to define many objects with the same behavior (they are of the same class).
We separate the object definition (the class) from the object creation (the function New).

The new syntax guarantees that the object is constructed without error.
It also improves readability and lets the system improve performance.

What is a class? (1)

- The class Counter that we defined is an argument to the function New:
  - C={New Counter Init}
- This means that Counter is a value
  - Class definition and object creation are separated
  - The class is an abstract data type with two basic operations: class definition and object definition
- In our earlier example, the function NewCounter combined both operations: defining the object behavior and creating the object
  - Most object-oriented languages separate the two operations to improve flexibility

What is a class? (2)

- How do we represent a class as a value? A class is a record that groups the attributes and method definitions:

```
Counter=c(attrs:[a1] methods:m(init:Init inc:Inc get:Get))
```

- The function New takes the record, creates the attributes (cells), and creates the object (a procedure that calls the methods with the attributes):

```
fun {New Class Init}
   S=(...)   % S is the state (record containing attributes)
   proc {Obj M}   % Obj is a one-argument procedure
      {Class.methods.{Label M} M S}
   end
in
   {Obj Init}   % Obj is initialized before it is returned
   Obj
end
```

- As an exercise, read and understand Section 7.2.2 in the book, which gives the full definition of New and shows how to create a class record.

Polymorphism

- In everyday language, an entity is polymorphic if it can assume different forms
  - The Greek god Proteus is polymorphic; he is a shape-shifter able to assume many forms
- In computing, an operation is polymorphic if it works correctly for arguments of different types
  - For example, an object message is polymorphic if many different objects will accept it
- This ability is needed in order to properly apportion responsibility over different parts of a program
  - A single responsibility should not be spread out; it should rather be concentrated in one place if possible

The responsibility principle

- Polymorphism allows to isolate responsibilities to the parts of the program that are concerned with them
  - A responsibility should be concentrated in one part of the program
- Example: a patient goes to see a medical doctor
  - The patient does not have to be a doctor!
  - The patient tells the doctor: "cure me"
  - The doctor understands this message and does the right thing (either cures the patient, or sends the patient to another doctor; we assume that eventually the right doctor is found!)
- The message "cure me" is polymorphic: it works with all medical specialties
  - All doctors understand the message "cure me"
  - The ability to cure a specific illness is concentrated in the doctor whose specialty covers that illness; we assume there is a mechanism to find the right doctor (for example, the generalist directs you to a specialist)

Implementing polymorphism

- All data abstractions we have seen can support polymorphism
  - Both objects and ADTs support it
  - But it is especially simple for objects
    - This is one reason for objects' enormous success
  - In this course, we will only talk about object polymorphism
    - The book also explains ADT polymorphism, if you are curious

- The idea is simple: we define the interface that the program needs
  - Then the program can accept all abstractions with that interface

Example: drawing of geometric figures

```
class Figure
    …
end
class Circle
    attr x y r
    meth draw … end
    …
end
class Line
    attr x1 y1 x2 y2
    meth draw … end
    …
end
```

```
class CompoundFigure
    attr figlist
    meth draw
        for F in @figlist do
            {F draw}
        end
    end
    …
end
```

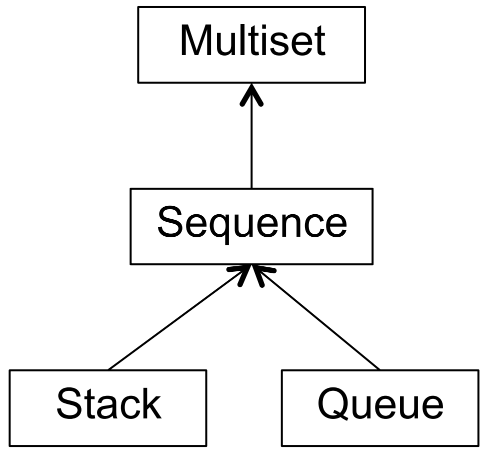
This definition of draw in CompoundFigure works for all possible figures: circles, lines, and other CompoundFigures!

Correctness of a polymorphic program

- When is a polymorphic program correct?
  - To be correct, each abstraction that the program accepts needs to satisfy certain properties (namely, those needed by the program)
  - For each abstraction, we need to verify that its specification has those properties
- For the figure drawing example, each draw method must correctly draw the object's figure
- For the doctor example, all doctors must cure the patient for their specialty
  - And for patients with another illness, the doctor must send the patient to a doctor better able to cure the illness (no cycles to avoid infinite loops!)

Similar data abstractions

- Data abstractions are often very similar
  - Especially if the entities they represent are similar (such as “person” versus “employee”, “car part” versus “airplane part”, and so forth)
- A simple example is the concept “collection of elements”
  - Multiset: a collection with no defined order
  - Sequence: a multiset with a total order
    - Sequence = multiset + total order
  - Stack: a sequence where adding and removing are done on the same side
    - Stack = sequence + add/remove constraint
  - Queue: a sequence where adding is done on one side and removing on the other side
    - Queue = sequence + add/remove constraint



- Language support for similar data abstractions is important

Incremental definition with inheritance

- It is important to avoid duplicated code in a program
  - Duplicated code is problematic at two levels
    - Different copies tend to diverge slightly with time (low-level bugs)
    - The same idea is expressed twice (high-level bugs)
  - It is much better, for program structure and maintenance, to express the same idea exactly once
- Inheritance achieves this for similar data abstractions
  - Definition A can “inherit” from definition B
  - This means that A uses B as a base, possibly with modifications and extensions
- The incremental definition A is also called a class
  - A class can either be a complete or incremental definition
  - The resulting definition (A + the classes it inherits from directly or indirectly) is always complete

Dangers of inheritance

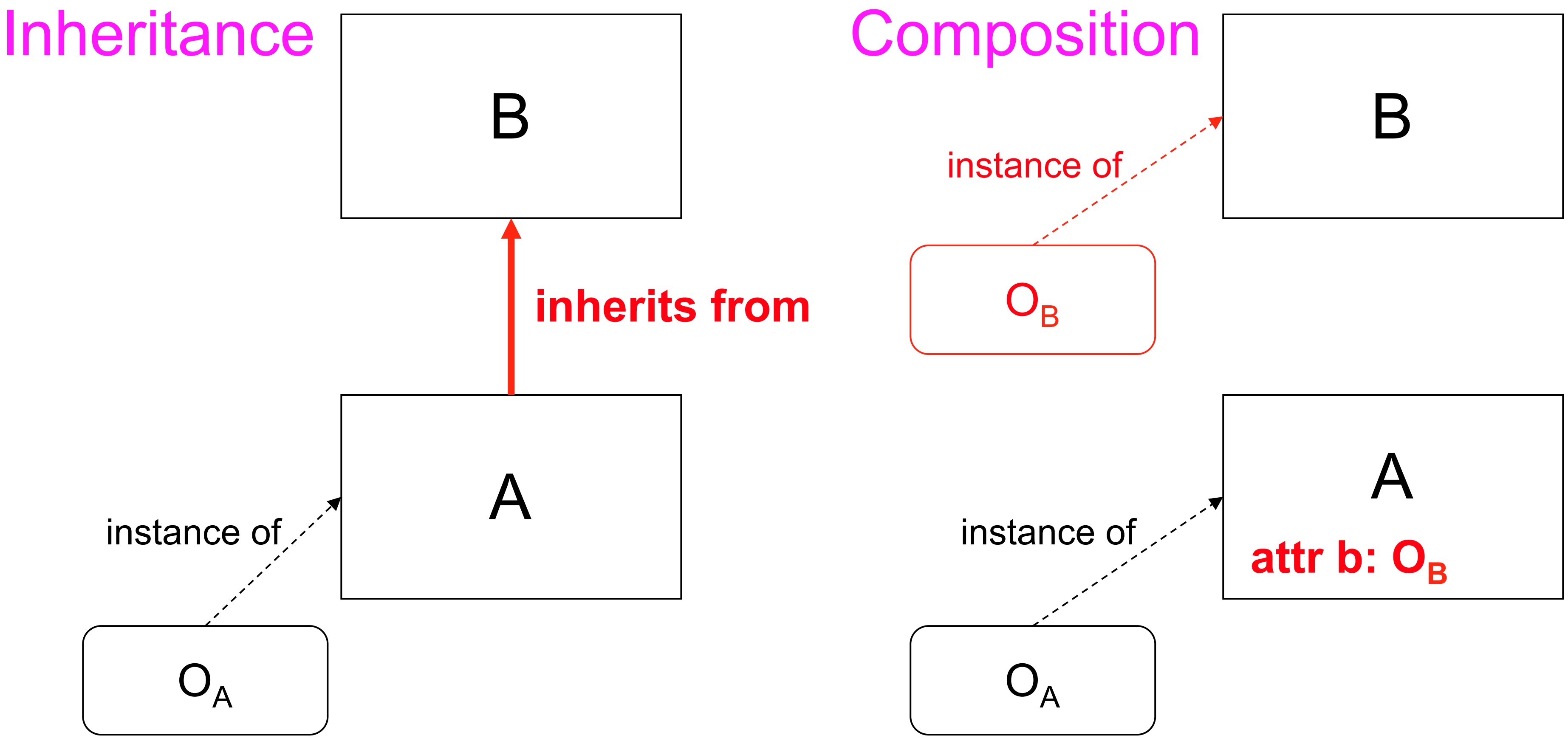
- Inheritance can be very useful, but its use is fraught with dangers
- The ability to extend A with inheritance is another interface to A
  - An additional interface to A's usual interface
  - This interface is extremely difficult to make correct and maintain correct throughout the lifetime of the abstraction
- So we must be very careful when using inheritance – two general rules:
  1. Prefer composition over inheritance
  2. When using inheritance, always follow the substitution principle

(1) Prefer composition over inheritance

- It is important to use inheritance as little as possible
  - Only use it in well-defined ways, for example in well-established "programming patterns"
  - When defining a class, it should be declared "final" (not extensible by inheritance) by default

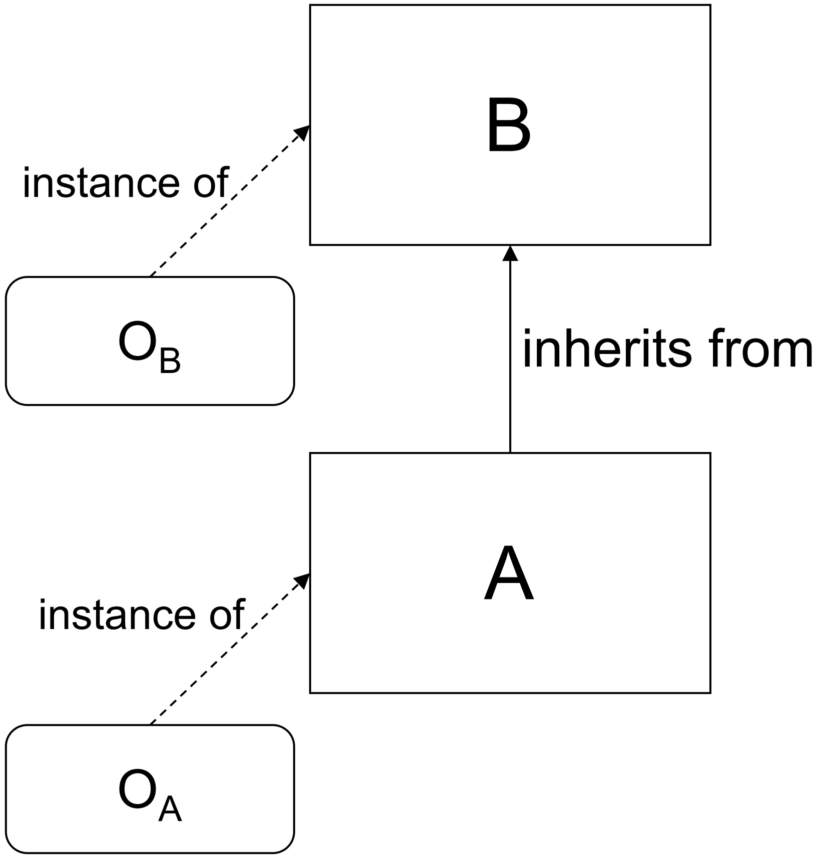
- Composition is much easier to use than inheritance and is often sufficient
  - Composition = an object refers to another object in one of its attributes (such as attribute figlist in CompoundFigure)
  - Composition does not add another interface: the object referred to is always accessed through its usual interface

Inheritance versus composition



(2) Always follow the substitution principle

- The use of inheritance is much easier if the substitution principle is followed
- Suppose that A inherits from B with objects O_A et O_B
  - Substitution principle: Every procedure that accepts O_B must accept O_A
  - If this principle is followed, then inheritance does not break anything! We say that A is a conservative extension of B.
- This is also called LSP (Liskov Substitution Principle)



HARICHE A. a.hariche@univ-dbkm.dz

Example:
class Account

```
class Account
   attr balance:0
   meth transfer(Amount)
         balance := @balance+Amount
   end
   meth getBal(B)
         B=@balance
   end
end
A={New Account transfer(100)}
```

Conservative extension

VerboseAccount:
An account that displays all transactions

```
class VerboseAccount
   from Account
   meth verboseTransfer(Amount)
        …
   end
end
```

The class VerboseAccount has methods transfer, getBal and the new method verboseTransfer.

Nonconservative extension

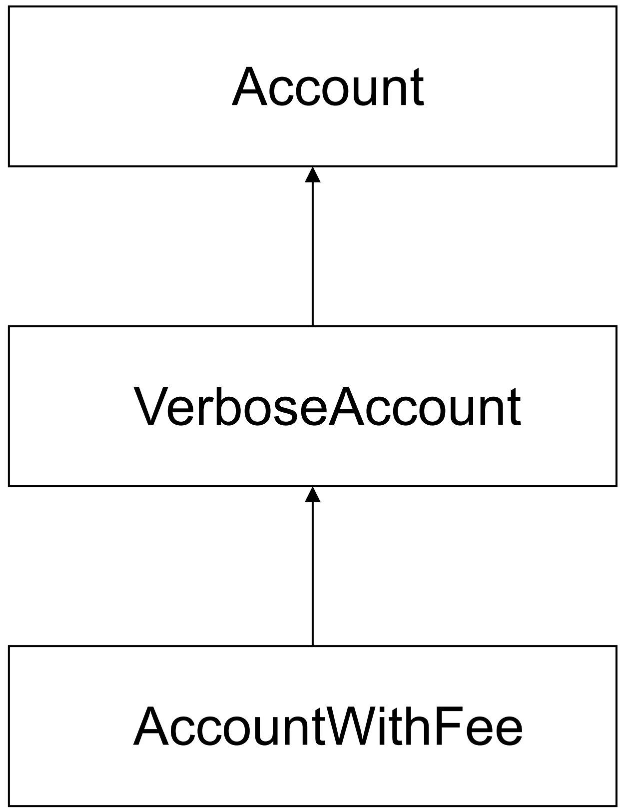
AccountWithFee:
An account with a fee

```
class AccountWithFee
    from VerboseAccount
    attr fee:5
    meth transfer(Amount)
         …
    end
end
```

The class AccountWithFee has methods transfer, getBal and verboseTransfer. The transfer method has been overridden.

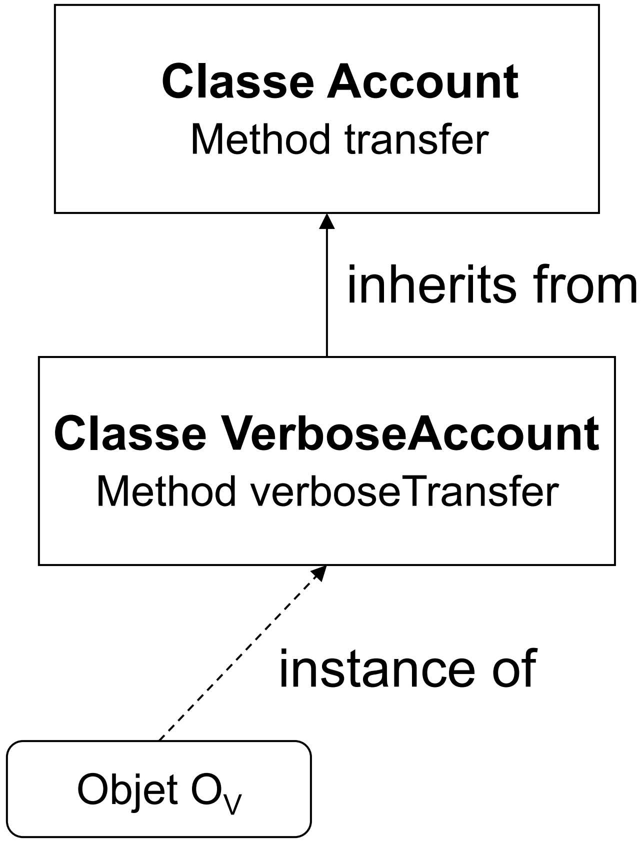
Class hierarchy

```
class VerboseAccount
    from Account
    meth verboseTransfer(Amount)
        …
    end
end

class AccountWithFee
    from VerboseAccount
    attr fee:5
    meth transfer(Amount)
        …
    end
end
```



Dynamic link

- Let us define the new method verboseTransfer
- In the definition of verboseTransfer, we need to call transfer
- Syntax: {**self** transfer(A)}
  - The transfer method is chosen in the class of the calling object O_V
  - **self** = the calling object, instance of VerboseAccount



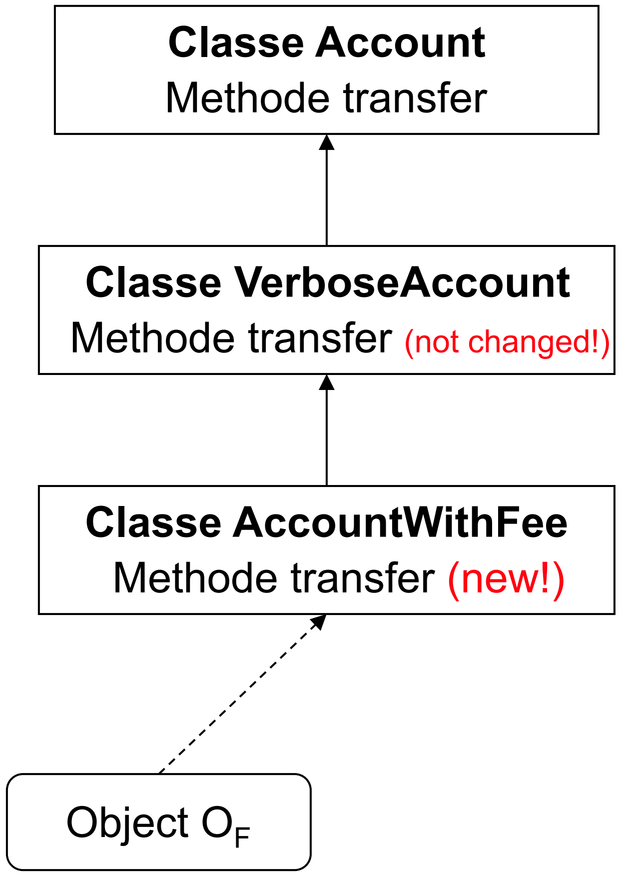
Definition of VerboseAccount

```
class VerboseAccount
    from Account
    meth verboseTransfer(Amount)
        {self transfer(Amount)}
        {Browse @balance}
    end
end
```

The class VerboseAccount has methods transfer, getBal and verboseTransfer.

Static link

- Let us override the old transfer method in AccountWithFee
- In the new transfer method, we need to call the old method!
- Syntax: VerboseAccount,transfer(A)
  - The class containing the old definition has to be named!
  - The transfer method is taken from the class VerboseAccount



HARICHE A. a.hariche@univ-dbkm.dz

Definition of AccountWithFee

```
class AccountWithFee
    from VerboseAccount
    attr fee:5
    meth transfer(Amt)
        VerboseAccount,transfer(Amt-@fee)
    end
end
```

The class AccountWithFee has methods transfer, getBal and verboseTransfer. The transfer method has been overridden.

The magic of dynamic links

- Look at the following fragment:
  A={New AccountWithFee transfer(100)}
  {A verboseTransfer(200)}
- What does it do?
  - Which transfer method is called by verboseTransfer?
    - The old one or the new one?
  - Observe: when VerboseAccount was defined, the class AccountWithFee did not exist yet
- Answer: !!

Example of a dynamic link

```
meth verboseTransfer(Amount)
    {self transfer(Amount)}
    {Browse @balance}
end
```

Call 1:

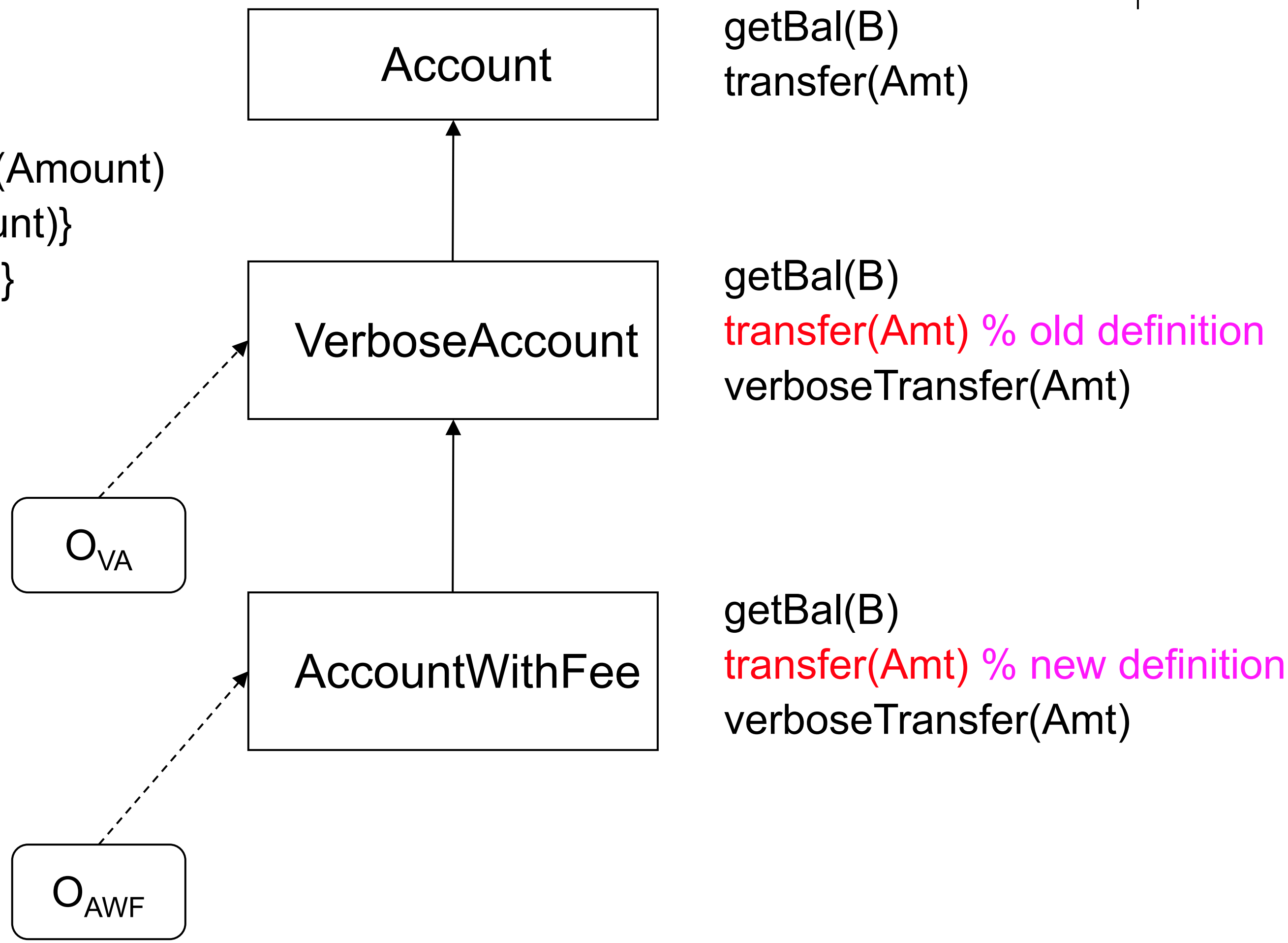
{O_{VA} verboseTransfer(200)}

Which transfer method?

Call 2:

{O_{AWF} verboseTransfer(200)}

Which transfer method?



HARICHE A. a.hariche@univ-dbkm.dz

Nonconservative extension


```
class AccountWithFee
    from VerboseAccount
    attr fee:5
    meth transfer(Amt)
        VerboseAccount,transfer(Amt-@fee)
    end
end
```

Invariant:

{A getBal(B)}

{A transfer(S)}

{A getBal(B1)}

% Is B1=B+S ?

% **No! It's broken!**

Summary of static and dynamic links

- The goal of static and dynamic links is to choose which method to execute
- Dynamic link: {**self** M}
  - The method is chosen in the class of the object
  - This class is only known during execution, this is why it is called a *dynamic* link
  - It should *always* be used by default
- Static link: SuperClass,M
  - The method is chosen in SuperClass
  - This class is known during compilation (it is SuperClass), this is why it is called a *static* link
  - It is *only* needed for overriding an existing method
  - When a method is overridden, the new definition often has to access the old one, and it uses a static link to do this

Whither object-oriented programming?

- Data abstraction languages have an enormous literature
  - These two lessons have barely introduced the three main principles of data abstraction (objects and ADTs), polymorphism, and inheritance
- The principles of data abstraction are now well-established
  - OOP has traditionally focused on sequential centralized programs. It is now being extended to long-lived distributed systems (« services »), with concurrency abstractions, fault tolerance, security, resource management, and configuration management (component-oriented programming).
  - Influential language developments are Scala (functional/object paradigms, message-passing concurrency) and Erlang (message passing with support for high availability), together with interesting experiments too numerous to mention
  - Large-scale distributed programming, including cloud-based big data and peer-to-peer computation, is pushing the limits of current data abstraction languages
- To meet these challenges, the structure of data abstraction languages will change significantly in the next two decades
  - Loose coupling, interoperability, distribution, and security will enter the language

Static typing versus dynamic typing

- A major property of a language is whether it is statically or dynamically typed

- Static typing: Variable types are known at compile time
  - Java, Scala, Haskell
- Dynamic typing: Variable types are not known at compile time but only at run time
  - Ruby, Python, Erlang, Scheme, Oz (language of this course)

- Static typing versus dynamic typing?
  - This question evokes intense debate between language designers
  - The main issues are guarantees and flexibility
  - Java augments static typing with concepts to increase flexibility
    - An Object class that is the root of the class hierarchy
    - The ability to define class code at run time with a class loader

Types in Java

- Two kinds of types: primitive types and reference types
  - User-defined types (e.g., classes) are reference types

- Primitive type: boolean (1 bit), character (16 bits), byte (8 bit integer, -128..127), short (16), int (32), long (64), float (32), double (64)
  - Characters: Unicode standard (all written languages)
  - Integers: representation in 2’s complement
  - Floating point: IEEE754 standard

- Reference type: class, interface, or array
  - A value is either “`null`” or a reference to an object or an array
  - An array type has the form `t[]` where `t` can be any type

Object-oriented programming in Java

- Data abstraction in Java
  - Primitive types are ADTs, user-defined types are objects
  - Rules of visibility
    - Private, package, protected, public
  - Objects of the same class can see inside each other (ADT property)

- Polymorphism in Java
  - Static polymorphism: Methods in the same class with the same name but different argument types (a.k.a. method overloading)
  - Dynamic polymorphism: Methods with the same name in different classes

- Inheritance in Java
  - Support for the substitution principle: an argument of a given class type will accept objects of any subclass
  - Support for multiple inheritance using a new concept called interface (a specific form of a general data abstraction interface)

Functional programming in Java

- Not much support for functional paradigm
  - More support is being added as Java evolves (lambda expressions in Java 8, which are procedure values)
    - Problem of legacy code!
  - Scala has full support for functional paradigm

- Final attributes and variables: can only be assigned once
  - Objects can be immutable, but are not functional objects

- Final classes: cannot be extended with inheritance

- "inner classes": a class defined inside another class
  - An instance of an inner class is almost (but not completely) a procedure value

Java, multiple inheritance, and exceptions

- This lesson completes the discussion of data abstraction and object-oriented programming with presentations of Java, multiple inheritance, and exceptions
- Java is a popular object-oriented language that has much support for practical programmers
- Multiple inheritance is when a class inherits from more than one class
- Exceptions are an important concept in imperative languages for handling error conditions (both program errors and environment errors)

Introduction to Java

- Java is the most-used language in the world today
  - Supported by libraries, tools, a high-quality implementation (the JVM) and a large developer community
  - But Java is >20 years old: there are many competitors, of which C++, Scala, and Erlang exemplify other parts of the language space
    - C++: closer to the processor architecture; older than Java
    - Scala: a more modern functional/object language built on the JVM
    - Erlang: a multi-agent language for highly available applications
- It is important to understand the execution of Java
  - Examples of Java semantics with the abstract machine
  - Java’s support for object-oriented programming
  - Limitations of Java

Two philosophies: Java versus C++

- Both Java and C++ implement an imperative paradigm supplemented with concurrency
  - (We will discuss concurrency in the next lesson)
  - Structured programming: a program is a set of nested blocks where each block has an entry and exit; there is no “goto” instruction in Java (but there is in C++)
  - Imperative control: if, switch, while, for, break, return, etc.

- Basic difference in design philosophy
  - C++ allows access to internal representation of data structures; memory management is manual
  - Java hides the internal representation; memory management is automatic (“garbage collection”)

Example program in Java

```
class Fibonacci {
   public static void main(String [] args) {
      int lo=1;
      int hi=1;
      System.out.println(lo);
      while (hi<50) {
         System.out.println(hi);
         hi=lo+hi;
         lo=hi-lo;
      }
   }
}
```

- All programs have a method `main` annotated `public static void`, executed when the program starts
- A Java variable (argument or local variable) is a cell
- Local variables must be initialized before use
- Integers are not objects but ADTs

- The method `println` is overloaded – there exist many methods with that name and the implementation chooses the right method according to the argument type (this is also called static polymorphism)

public static void main(...)

- All methods can be given modifiers

- The `main` method has the following modifiers:
  - `public`: visible in the whole program (no restrictions)
  - `static`: there is one per class (not one per object)
  - `void`: the method returns no result
    (so it is a procedure, not a function)
- The `main` method has one argument
  - `String[]`: the argument's type, an array that contains
    String objects

Java semantics with the abstract machine

- As for any language, it is important to understand precisely what the Java language does
  - We can define Java semantics with the abstract machine
  - Most (but not all) of the semantics is straightforward
- We give two examples to show how to give the semantics of Java concepts
  - Parameter passing
  - Static attributes in classes
- For a complete semantics of Java we recommend the book
  - *Java Precisely* by Peter Sestoft, MIT Press, 2005

Parameter passing in Java

```
class ByValueExample {
   public static void main(String[] args) {
      double one=1.0;
      System.out.println(“before: one = “ + one);
      halveIt(one);
      System.out.println(“after:  one = “ + one);
   }
   public static void halveIt(double arg) {
      arg /= 2.0;
   }
}
```

- Parameter passing is an important part of a language that needs to be understood precisely
- This program calls `halveIt` with argument `one`: what does it print?

Semantics of halveIt

```
public static void halveIt(double arg) {
   arg = arg/2.0;
}
```

```
proc {HalveIt X}
   Arg={NewCell X}
in
   Arg := @Arg / 2.0
end
```

- Here is how to write `halveIt` in Oz
  - This definition gives its semantics
  - This defines only the execution behavior, not the type checking
- The argument Arg is a local cell
  - The number is passed into the local cell
  - Assignments to Arg affect only the local cell, not the cell in the method main
- The number is passed by value

HARICHE A. a.hariche@univ-dbkm.dz

Passing an object parameter

```
class Body {
   public long idNum;
   public String name = “<unnamed>”;
   public Body orbits = null;
   private static long nextID = 0;

   Body(String bName, Body orbArd) {
      idNum = nextID++;
      name = bName;
      orbits = orbArd;
   }
}
```

```
class ByValueRef {
   public static void main(String [] args) {
     Body sirius = new Body(“Sirius”, null);
     System.out.println(“bef:“+sirius.name);
     commonName(sirius);
     System.out.println(“aft:“+sirius.name);
   }
   public static void commonName(Body bRef) {
     bRef.name = “Dog Star”;
     bRef = null;
   }
}
```

- The class `Body` has a constructor (the method `Body`) and a static attribute (the integer `nextID`)
- The program calls `commonName` with the object `sirius`
- The content of `sirius` is modified by `commonName`, but assigning `bRef` to `null` has no effect on `sirius`!

HARICHE A. a.hariche@univ-dbkm.dz

Semantics of commonName

```
public static void commonName(Body bRef)
{
    bRef.name = "Dog Star";
    bRef = null;
}
```

```
proc {CommonName X}
   BRef={NewCell X}
in
   {@BRef setName("Dog Star")}
  BRef:=null
end
```

- Here is how to write `commonName` in Oz
- BRef is a local cell whose content is an object reference
- When CommonName is called, then BRef is initialized with a reference to the object Sirius
- The object reference is passed by value
  - Changes to the content of BRef do not affect the object Sirius

The class Body and its static attribute

```
declare
local NextID Body in
   NextID={NewCell 0}
   class Body
      attr idNum
         name:"<unnamed>"
         orbits:null
      meth initBody(BName OrbArd)
         idNum:=@NextID
         NextID:=@NextID+1
         name:=BName
         orbits:=OrbArd
      end
   end
end
```

```
class Body {
   public long idNum;
   public String name = "<unnamed>";
   public Body orbits = null;
   private static long nextID = 0;

   Body(String bName, Body orbArd) {
      idNum = nextID++;
      name = bName;
      orbits = orbArd;
   }
}
```

- The definition of class `Body` in Oz gives its semantics
- NextID is a static attribute: a cell defined outside the class, at the same time as the class
  - Not like other attributes which are defined per object
- The constructor `Body` corresponds to method initBody

Classes in Java

- A Java class has fields (attributes or methods), and members (other classes or interfaces)
- Java has syntax for static and dynamic links
  - The keyword "`super`" gives a static link to the class one level up (as we saw, it should be rarely used!)
  - The keyword "`this`" is used to mean "**self**"
- Java allows single inheritance of classes
  - A class can inherit from exactly one other class

Inheritance example

```
class Point {
  public double x, y;

  public void clear() {
     x=0.0;
     y=0.0;
  }
}
```

```
class Pixel extends Point {
  Color color;

  public void clear() {
     super.clear();
     color=null;
  }
}
```

The class Object

- The class Object is the root of the hierarchy
  - All classes inherit from Object

```
Object oref = new Pixel();
oref = “Some String”;
oref = “Another String”;
```

- The reference `oref` can refer to any object
  - We regain some of the flexibility of dynamic typing
  - (String objects are immutable)

Abstract classes and concrete classes

- An abstract class is a class that does not implement all its methods (bodies are missing)
  - An abstract class cannot be instantiated
- A concrete class implements all its methods
  - A concrete class can inherit from an abstract class
  - A concrete class can be instantiated
- With abstract classes, we can write generic programs
  - We define the missing methods using inheritance, to get a concrete class that we can instantiate and execute

Example of an abstract class

```
abstract class Benchmark {
    abstract void benchmark();

    public long repeat(int count) {
        long start=System.currentTimeMillis();
        for (int i=0; i<count; i++)
            benchmark();
        return (System.currentTimeMillis()-start);
    }
}
```

Doing the same with a higher-order function

- We can achieve the same effect using a higher-order function:

```
fun {Repeat Count Benchmark}
   Start={OS.time}
in
   for I in 1..Count do {Benchmark} end
   {OS.time}-Start
end
```

- Function Repeat corresponds to method `repeat`
- Procedure argument Benchmark corresponds to method `benchmark`
- With abstract classes, we can achieve the same effect as passing a procedure as argument
  - We use inheritance to simulate a procedure argument

Final classes

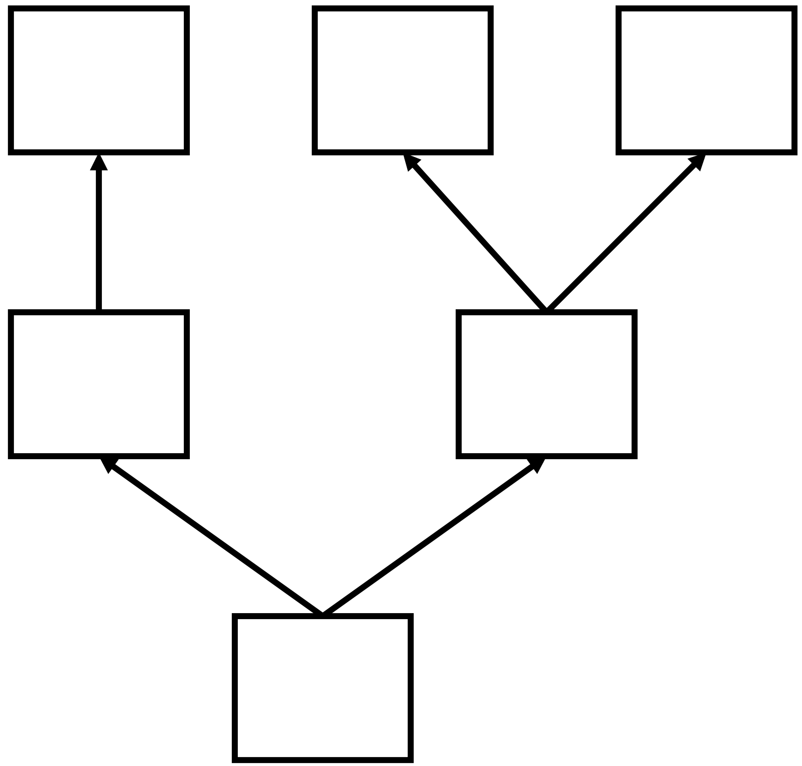
- A final class cannot be extended with inheritance

```
final class NotExtendable {
    ...
}
```

- A final method cannot be redefined with inheritance
- It is good practice to define all classes as final classes, except those we wish to be extensible
  - Is it a good idea to define an abstract class as final?

The inheritance hierarchy

- We add an edge between each class and its direct superclasses
  - This gives a directed acyclic graph called the inheritance hierarchy
- We know how to define a class that inherits from one class (single inheritance), but how can a class inherit from more than one (multiple inheritance)?
  - Multiple inheritance is complicated but it can be a powerful tool
- We give a simple example; for much more see the book
  - *Object-oriented Software Construction* by Bertrand Meyer, Prentice-Hall, 1997



Example of multiple inheritance

- Geometric figures

```
class Figure
  meth draw ... end
  ...
end
class Line from Figure
  meth draw ... end
  ...
End
```

- Linked lists

```
class LinkedList
  meth forall(M)
    ... % invoke M on all elements
  end
  ...
end
```

- Compound figures

```
class CompoundFigure from
  Figure LinkedList
  meth draw
    {self forall(draw)}
  end
  ...
end
```

- A compound figure is *both* a figure and a linked list
- Multiple inheritance works in this case because the two superclasses are *independent*

HARICHE A. a.hariche@univ-dbkm.dz

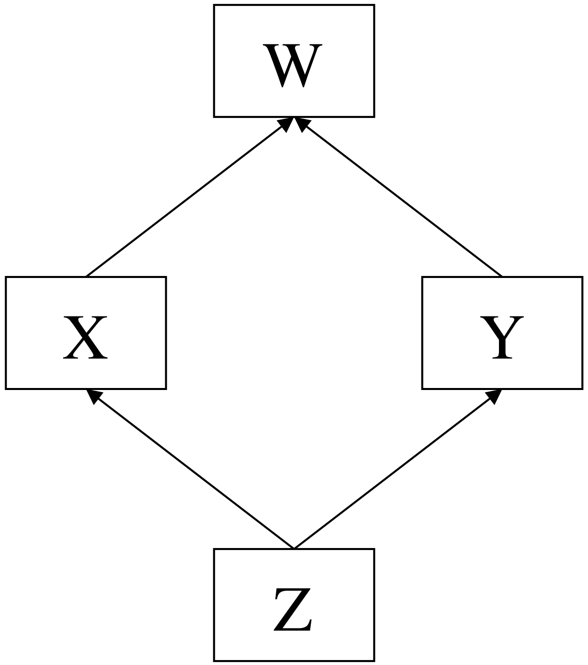
Java interfaces and multiple inheritance

- Java only allows single inheritance for classes
  - Multiple inheritance is forbidden, but to keep some of its expressiveness, Java introduces the concept of interface
- An interface is similar to an abstract class with no method implementations
  - The interface gives the method names and their argument types, without the implementation
- Java allows multiple inheritance for interfaces

Example of a Java interface

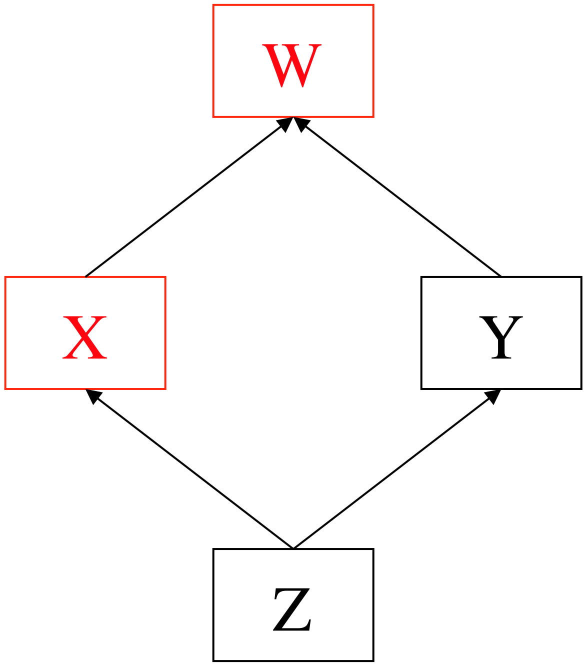
```
interface Lookup {
   Object find(String name);
}

class SimpleLookup implements Lookup {
   private String[] Names;
   private Object[] Values;
   public Object find(String name) {
      for (int i=0; i<Names.length; i++) {
         if (Names[i].equals(name))
            return Values[i];
      }
      return null;
   }
}
```

The diamond problem



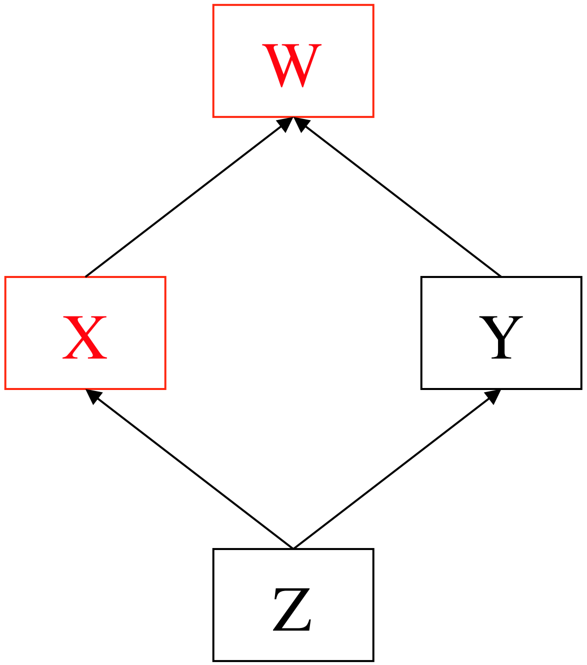
- The diamond problem is a classic problem with multiple inheritance
- When class W has state (attributes), who will initialise W? X or Y or both?
  - There is no simple solution
  - This is one reason why multiple inheritance is not allowed in Java
- Interfaces give a partial solution to this problem

A solution with interfaces



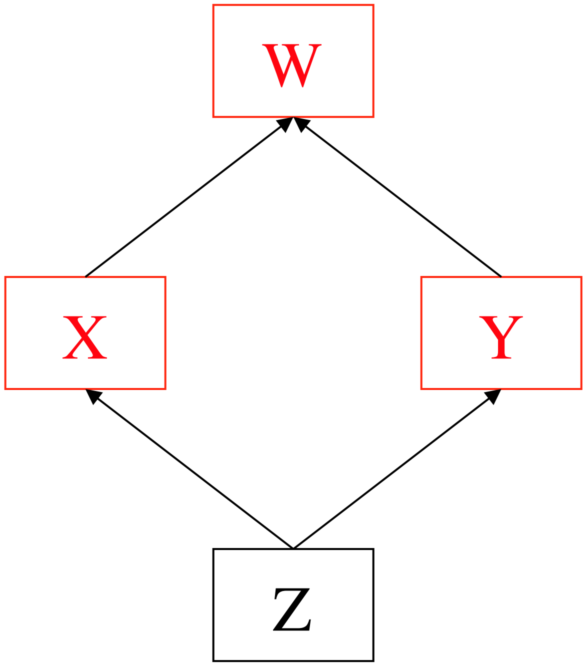
- Interfaces are given in red
- There is no more diamond inheritance: class Z only inherits from class Y
- For an interface, inheritance is just a constraint on the method headers (names and arguments) in the classes
  - Multiple inheritance means more constraints on the method headers
  - An interface contains no code; no code means no diamond problem

Java syntax for the diamond example




```
interface W { }
interface X extends W { }
class Y implements W { }
class Z extends Y
        implements X { }
```

Another solution for the same example



- In this solution, Z is the only class in the hierarchy
- It has the following syntax:

```
interface W { }
interface X extends W { }
interface Y extends W { }
class Z implements X, Y { }
```

- Are there any other solutions for this example?

HARICHE A. a.hariche@univ-dbkm.dz

Example
using exceptions

```
fun {Eval E}
   if {IsNumber E} then E
   else
      case E
      of plus(X Y) then {Eval X}+{Eval Y}
      [] times(X Y) then {Eval X}*{Eval Y}
      else raise badExpression(E) end
      end
   end
end

try
   {Browse {Eval plus(23 times(5 5))}}
   {Browse {Eval plus(23 minus(4 3))}}
catch X then {Browse X} end
```

- The error handling code does not clutter up the program

If we did not have exceptions…

```
fun {Eval E}
   if {IsNumber E} then E
   else
      case E
      of plus(X Y) then R={Eval X} in
         case R of badExpression(RE) then badExpression(RE)
         else R2={Eval Y} in
            case R2 of badExpression(RE) then badExpression(RE)
            else R+R2
            end
         end
      [] times(X Y) then
         % … Same code as plus
      else badExpression(E)
      end
   end
end
```

- Much more code!
  - In this example, 22 lines instead of 10 (more than double)
- The code is much more complicated because of all the **case** statements handling badExpression

The “finally” clause

- The **try** has an additional **finally** clause, for an operation that must always be executed (in both the correct and error cases):

```
FH={OpenFile “foobar”}
try
     {ProcessFile FH}
catch X then
     {Show “*** Exception during execution ***”}
finally {CloseFile FH} end % Always close the file
```

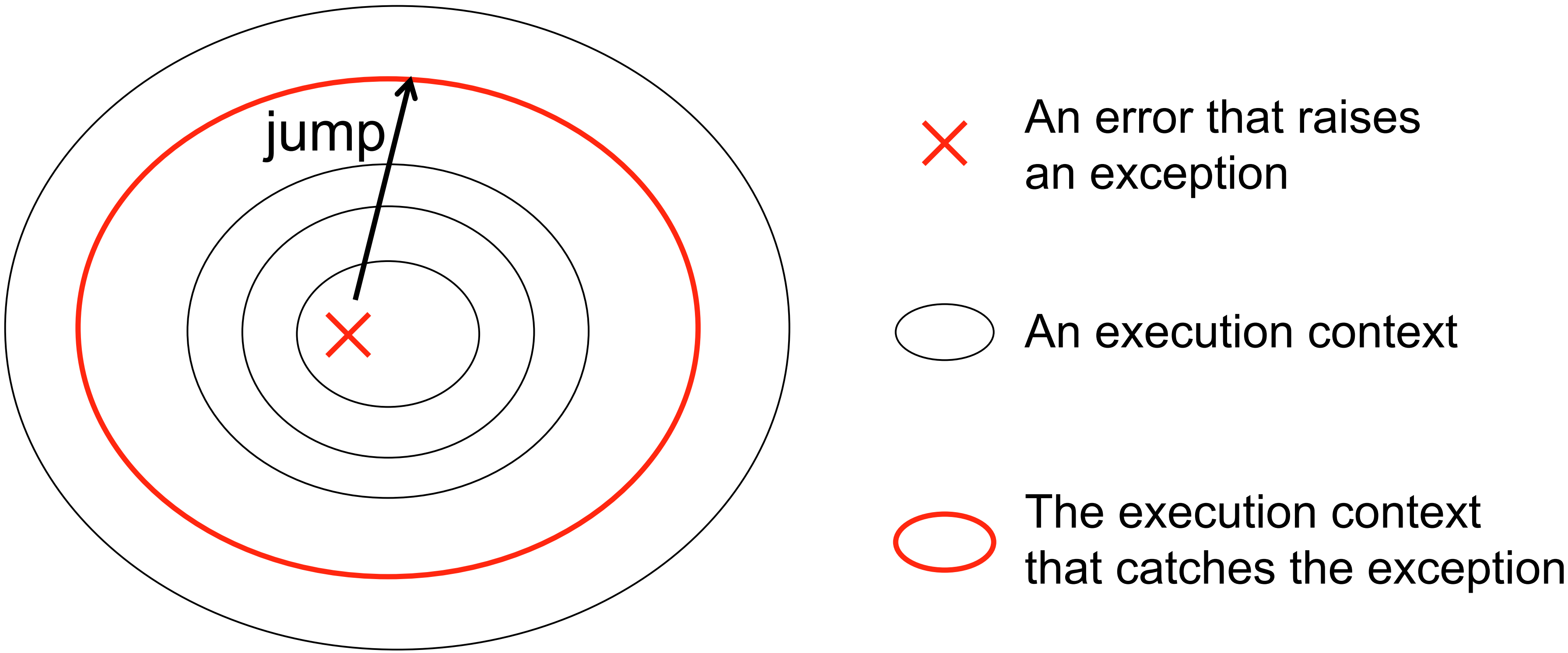
How to handle exceptional situations

- How can we handle exceptional situations in a program?
  - Such as: division by 0, opening a nonexistent file, and so forth
  - Program errors but also errors from outside the program
  - Things that happen rarely but that must be taken care of
- We add a new programming concept called exceptions
  - We define exceptions and show how they are used
  - We give the semantics of exceptions in the abstract machine
- With exceptions, we can handle exceptional situations without cluttering up the program with rarely used error checking code

The containment principle

- When an error occurs, we would like to be able to recover from the error
- Furthermore, we would like the error to affect as little as possible of the program
- We propose the containment principle:
  - A program is a set of nested execution contexts
  - An error will occur inside an execution context
  - A recovery routine (exception handler) exists at the boundary of an execution context, to make sure the error does not propagate to higher execution contexts

Handling an exception



- An executing program that encounters an error must jump to another part (the exception handler) and give it a reference (the exception) that describes the error

The try and raise instructions

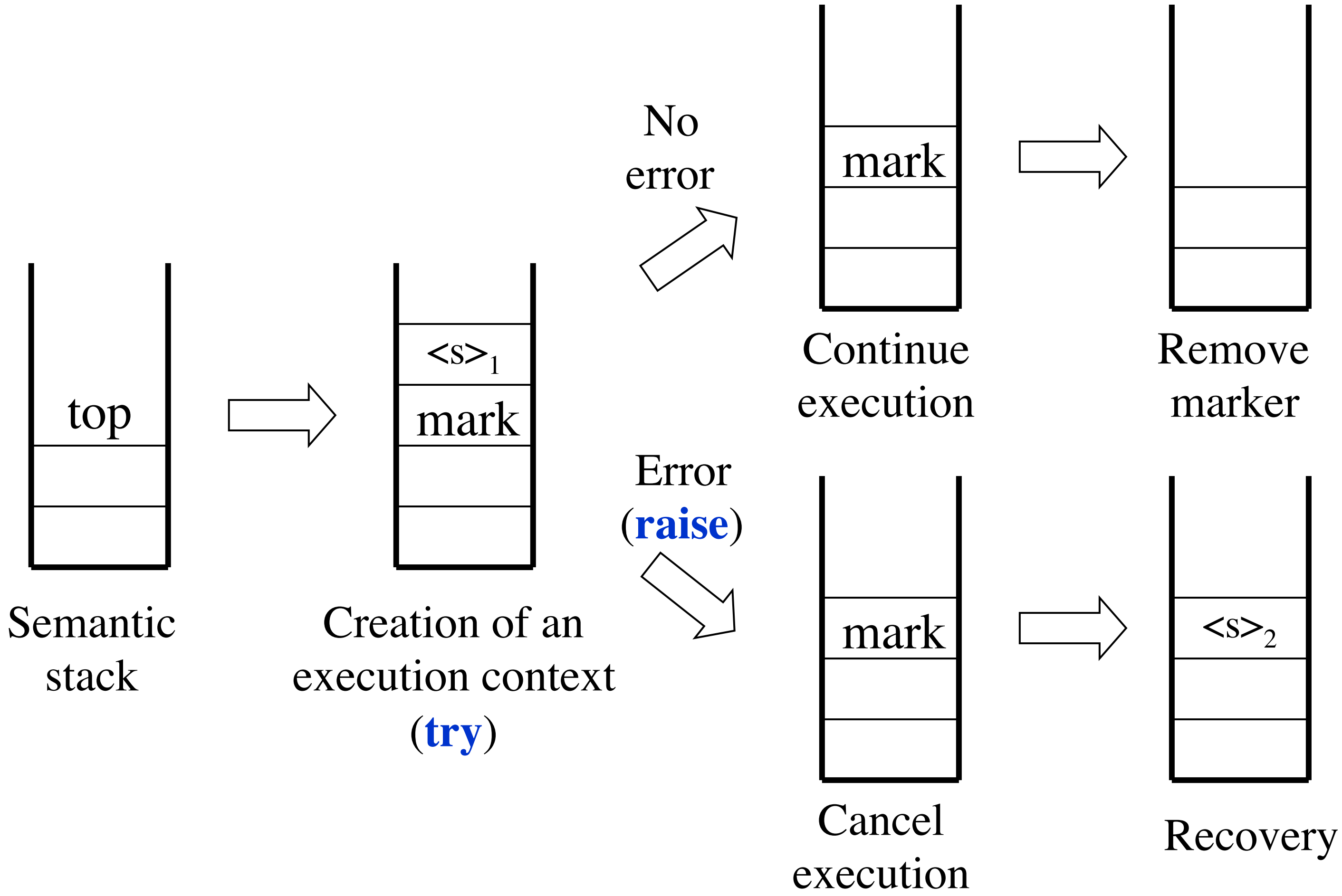
- We introduce two new instructions for handling exceptions:

try <s>$_1$ **catch** <y> **then** <s>$_2$ **end** % Create an execution context
raise <x> **end** % Raise an exception

- With the following behavior:
  - **try** puts a “marker” on the stack and starts executing <s>$_1$
  - If there is no error, <s>$_1$ executes normally and removes the marker when it terminates
  - **raise** is executed when there is an error, which empties the stack up to the marker (the rest of <s>$_1$ is therefore canceled)
    - Then <s>$_2$ is executed
    - <y> refers to the same variable as <x>
    - The scope of <y> exactly covers <s>$_2$

HARICHE A. a.hariche@univ-dbkm.dz

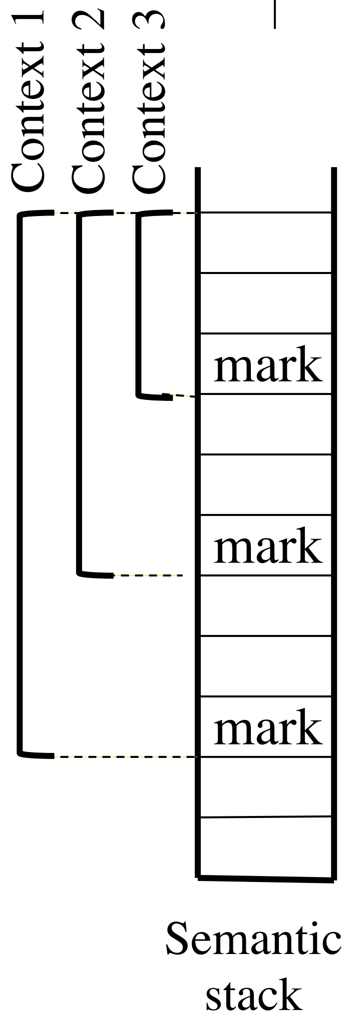
Semantics of exceptions



An execution context

- An execution context is the part of the semantic stack that starts with a marker and continues to the stack top:

```
try ... % Context 1
  try ... % Context 2
    try ... % Context 3
    catch <x> then <s>3 end
    ...
  catch <x> then <s>2 end
  ...
catch <x> then <s>1 end
```



HARICHE A. a.hariche@univ-dbkm.dz

Exceptions in Java

- An exception is an object that inherits from the class Exception (which is a subclass of Throwable)
- There are two kinds of exceptions
  - Checked exceptions: The compiler verifies that all methods only throw the exceptions declared for the class
  - Unchecked exceptions: Some exceptions can arrive without the compiler being able to verify them. They inherit from RuntimeException and Error.
- For exceptions that the program itself defines, you should always use checked exceptions, since they are declared and therefore part of the program's interface

Java exception syntax

```
throw new NoSuchAttributeException(name);

try {
   <stmt>
} catch (exctype1 id1) {
   <stmt>
} catch (exctype2 id2) {
   …
} finally {
   <stmt>
}
```

Good style

- We read a file and perform an action for each item in the file:

```
try
   while (!stream.eof())
      process(stream.nextToken());
finally
   stream.close();
```

Bad style

- We can use the exception handler to change the execution order during normal execution:

```
try {
   for (;;)
      process (stream.next());
} catch (StreamEndException e) {
   stream.close();
}
```

- Reaching the end of a stream is completely normal, it is not an error. What happens if a real error happens and is mixed in with the normal operation? You don't want to handle this. Normal operation should be kept separate from errors!

Final remarks

- This completes the part of the course related to data abstraction
  - Explicit state and object-oriented programming
  - Java, multiple inheritance, and exceptions
- We have covered three of the four themes
  - Functional programming (including recursion, invariant programming, and higher-order programming)
  - Language semantics (a complete operational semantics)
  - Data abstraction (including explicit state and object-oriented programming)
- We end this theme with a reflection on language design and an introduction to concurrent programming

Java, Scala, and language design

- We have discussed some of the principles that were used to design Java (1990s)
  - True data abstraction (encapsulation, GC)
  - Almost all entities are objects
  - Support for object-oriented design
- Scala has added two principles to this (2000s)
  - Strict separation between mutable/immutable
  - Everything is an object (including functions)
- These principles considerably increase Scala's expressive power compared to Java
  - We consider that Scala is a worthy successor to Java

Final theme: concurrency

- The final theme of the course will be concurrency
  - Multiple activities that evolve independently and collaborate
  - There are three fundamental forms of concurrent programming: deterministic dataflow, message passing, and shared state
  - All three were invented (or discovered?) in the early 1970s!
- We will present deterministic dataflow in depth
  - It is an extremely powerful yet easy to use model that deserves to be more widely known
  - All the techniques of functional programming generalize for deterministic dataflow

Many important ideas

Louv1.1x

- Identifiers and environments
- Functional programming
- Recursion
- Invariant programming
- Lists, trees, and records
- Symbolic programming
- Instantiation
- Genericity
- Higher-order programming
- Complexity and Big-O notation
- Moore's Law
- NP and NP-complete problems
- Kernel languages
- Abstract machines
- Mathematical semantics

Louv1.2x

- Explicit state
- Data abstraction
- Abstract data types and objects
- Polymorphism
- Inheritance
- Multiple inheritance
- Object-oriented programming
- Exception handling
- Concurrency
- Nondeterminism
- Scheduling and fairness
- Dataflow synchronization
- Deterministic dataflow
- Agents and streams
- Multi-agent programming

PLP_Drive space

https://drive.google.com/drive/folders/1YBCIZzAldeiT19DIfDiREQwP-NAQ1qMN