Object-oriented Programming: concepts and Fortran implementation

Nicolas Tancogne-Dejean
Octopus Advanced Courses
Concepts of object-oriented programming
Programming paradigms

A way to classify programming languages based on their features

- **Imperative**: instructs the machine how to change its states
  - Procedural
  - **Object-oriented**

- **Declarative**: programmer declares properties of the desired result, but not how to compute it
  - Functional; Logic; Mathematical; Reactive
Procedural programming

- Based on the concept of the procedure call
- Procedures (routines/subroutines) contain a series of computational steps to be carried out
- Procedures can be call at any point during the execution

Examples: Fortran, ALGOL, COBOL, BASIC, Pascal, C
Procedural programming

- Focus of procedural programming is to break down programming task into a collection of
  - Variables
  - Data structures
  - subroutines
Object-oriented programming (OOP)

- Based on the concept of objects and classes
- Objects contain
  - Data: fields, attributes, or properties
  - Code: procedures or methods

Objects interact with one another

Objects are instances of classes, with a determined type

Example: C++, Java, Python, C#, R, PHP, ....
Object-oriented programming (OOP)

Classes represent broad categories:
- Car
- Dog

Objects represented by the same class share attributes

Cars have a color, dogs have a name and an age, ...

Classes serve as a blueprints to create individual objects
Benefits of OOP

- Model complex things as reproducible simple structures → Abstraction
- Reusable
- Polymorphism
- Protect information though encapsulation
- Inheritance
Building blocks

• Classes → user-defined data types
  – A car

• Objects → instance of classes
  – Angel’s car

• Methods → represent a behavior/ perform actions
  – Change color, drive, stop

• Attributes → information stored
  – Angel’s car is blue
We want to reuse code from other classes

→ Supports re-usability

Child classes automatically gain access to functionalities of their parent class

Example: A *particle* class knows out to give its mass

Child class *charged_particle* directly knows it
Encapsulation

- Notion of public/privated variables
- Protects against common mistakes
- Hide complexity
- Other objects do not have access to the class/cannot make changes

Example: For a user, a car has a steering wheel, gas and brake. Complexity is hidden in the engine, and the car only exposes simple interfaces.
Abstraction

- Hide complexity
- Extension of encapsulation
- Certain classes are “abstract”. One can only instantiate a child class.

Example: An *animal* does not exist in nature. However, all animals have similar attribute/behavior.

*Cat* and *dog* are *animals*, and all have a *species* and an *age*. Species and age are common to all animals.
Polymorphism

• Same method to execute different behavior
• Overriding and overloading (compile time polymorphism)
• Objects of different types can be passed through the same interface
  → Avoids code duplication

Example: different time propagators behave differently. But they all do the same task: propagate from t to t+dt

\[ prop->\text{propagate}(dt) \]
Object-oriented programming in Fortran
Fortran vs Fortran 2003

- In “old-fashioned” Fortran: data types and modules
- Features of Fortran 2003:
  - Type-bound procedures
    - \( a = c\%\text{area}() \) instead of \( a = \text{circle\_area}(c) \)
  - Type extension
    - Allows for inheritance
  - Polymorphism
    - Procedure polymorphism
    - Data polymorphism
OOP in Fortran

- A type is a “class”. It can be abstract or extends from another type.
- One can ask the type of an object using select type.
- Public/private keywords → data hiding
- Unlimited polymorphic objects are possible
- Multiple inheritance is impossible
Some details about types in OOP

Fortran

- **Abstract** type:
  type that cannot be instantiated

- **Deferred** binding:
  not defined in the abstract type. Fixed interface for all the child classes → polymorphism

- **Non-overridable**:
  methods that cannot be overwritten by child types
Example: polymorphic linked list

Taken from src/basic/linked_list.F90

type :: linked_list_t
  private
    integer, public :: size = 0
    class(list_node_t), pointer :: first_node => null()
    class(list_node_t), pointer :: last_node => null()
  contains
    procedure :: add_node => linked_list_add_node
    procedure :: add_ptr  => linked_list_add_node_ptr
    procedure :: add_copy => linked_list_add_node_copy
    procedure :: delete => linked_list_delete_node
    procedure :: has => linked_list_has
    procedure :: copy => linked_list_copy
    generic :: assignment(=) => copy
    procedure :: empty => linked_list_empty
    final :: linked_list_finalize
end type linked_list_t
Example: polymorphic linked list

Taken from src/basic/linked_list.F90

type :: linked_list_t
    private
        integer, public :: size = 0
        class(list_node_t), pointer :: first_node => null()
        class(list_node_t), pointer :: last_node => null()
    contains
        procedure :: add_node => linked_list_add_node
        procedure :: add_ptr  => linked_list_add_node_ptr
        procedure :: add_copy => linked_list_add_node_copy
        procedure :: delete => linked_list_delete_node
        procedure :: has => linked_list_has
        procedure :: copy => linked_list_copy
        generic   :: assignment(=) => copy
        procedure :: empty => linked_list_empty
        final     :: linked_list_finalize
end type linked_list_t

The name of the class
Example: polymorphic linked list

Taken from src/basic/linked_list.F90

```fortran
type :: linked_list_t
  private
    integer, public :: size = 0
  class(list_node_t), pointer :: first_node => null()
  class(list_node_t), pointer :: last_node => null()
contains
  procedure :: add_node => linked_list_add_node
  procedure :: add_ptr  => linked_list_add_node_ptr
  procedure :: add_copy => linked_list_add_node_copy
  procedure :: delete => linked_list_delete_node
  procedure :: has => linked_list_has
  procedure :: copy => linked_list_copy
  generic :: assignment(=) => copy
  procedure :: empty => linked_list_empty
  final :: linked_list_finalize
end type linked_list_t
```

By default, we want all attributes to be private
Good practice in general
Example: polymorphic linked list

Taken from src/basic/linked_list.F90

```fortran
type :: linked_list_t
  private
  integer, public : size = 0
  class(list_node_t), pointer :: first_node => null()
  class(list_node_t), pointer :: last_node => null()
contains
  procedure :: add_node => linked_list_add_node
  procedure :: add_ptr  => linked_list_add_node_ptr
  procedure :: add_copy => linked_list_add_node_copy
  procedure :: delete => linked_list_delete_node
  procedure :: has => linked_list_has
  procedure :: copy => linked_list_copy
  generic :: assignment(=) => copy
  procedure :: empty => linked_list_empty
  final :: linked_list_finalize
end type linked_list_t
```

We want this attribute to be public
Example: polymorphic linked list

Taken from src/basic/linked_list.F90

type :: linked_list_t
  private
    integer, public :: size = 0
    class(list_node_t), pointer :: first_node => null()
    class(list_node_t), pointer :: last_node => null()
  contains
    procedure :: add_node => linked_list_add_node
    procedure :: add_ptr  => linked_list_add_node_ptr
    procedure :: add_copy => linked_list_add_node_copy
    procedure :: delete => linked_list_delete_node
    procedure :: has => linked_list_has
    procedure :: copy => linked_list_copy
    generic   :: assignment(=) => copy
    procedure :: empty => linked_list_empty
    final     :: linked_list_finalize
end type linked_list_t

Pointer to a “class” list_node_t
This can also be any child class
Example: polymorphic linked list

 Taken from src/basic/linked_list.F90

type :: linked_list_t
  private
    integer, public :: size = 0
  class(list_node_t), pointer :: first_node => null()
  class(list_node_t), pointer :: last_node => null()
contains
  procedure :: add_node => linked_list_add_node
  procedure :: add_ptr  => linked_list_add_node_ptr
  procedure :: add_copy => linked_list_add_node_copy
  procedure :: delete => linked_list_delete_node
  procedure :: has => linked_list_has
  procedure :: copy => linked_list_copy
  generic :: assignment(=) => copy
  procedure :: empty => linked_list_empty
  final    :: linked_list_finalize
end type linked_list_t
Example: polymorphic linked list

Taken from src/basic/linked_list.F90

type :: linked_list_t
  private
    integer, public :: size = 0
    class(list_node_t), pointer :: first_node => null()
    class(list_node_t), pointer :: last_node => null()
  contains
    procedure :: add_node  => linked_list_add_node
    procedure :: add_ptr   => linked_list_add_node_ptr
    procedure :: add_copy  => linked_list_add_node_copy
    procedure :: delete    => linked_list_delete_node
    procedure :: has       => linked_list_has
    procedure :: copy      => linked_list_copy
    generic :: assignment(=) => copy
    procedure :: empty     => linked_list_empty
    final    :: linked_list_finalize
end type linked_list_t
Example: polymorphic linked list

Taken from src/basic/linked_list.F90

type :: linked_list_t
  private
    integer, public :: size = 0
    class(list_node_t), pointer :: first_node => null()
    class(list_node_t), pointer :: last_node => null()
  contains
    procedure :: add_node => linked_list_add_node
    procedure :: add_ptr  => linked_list_add_node_ptr
    procedure :: add_copy => linked_list_add_node_copy
    procedure :: delete  => linked_list_delete_node
    procedure :: has     => linked_list_has
    procedure :: copy    => linked_list_copy
    generic    :: assignment(=) => copy
    procedure :: empty   => linked_list_empty
    final      :: linked_list_finalize
end type linked_list_t

Finalizer: a routine automatically called when the object is destroyed
Example: polymorphic linked list

Taken from src/basic/list_node.F90

```fortran
type :: list_node_t
  private
  logical :: clone
  class(*), pointer :: value => null()
  type(list_node_t), pointer :: next_node => null()
contains
  procedure :: get => list_node_get
  procedure :: next => list_node_next
  procedure :: set_next => list_node_set_next
  procedure :: is_equal => list_node_is_equal
  procedure :: copy => list_node_copy
  final :: list_node_finalize
end type list_node_t
```
Example: polymorphic linked list

Taken from src/basic/list_node.F90

```fortran
 type :: list_node_t
  private
  logical :: clone
  class(*),          pointer :: value => null()
  type(list_node_t), pointer :: next_node => null()
contains
  procedure :: get => list_node_get
  procedure :: next => list_node_next
  procedure :: set_next => list_node_set_next
  procedure :: is_equal => list_node_is_equal
  procedure :: copy => list_node_copy
  final :: list_node_finalize
end type list_node_t
```

Together, we have a linked list of `list_node`, which are pointing to any possible class (`class(*)`)
Example: polymorphic linked list

Creating a child class: a list of integers

type, extends(linked_list_t) :: integer_list_t
  private
contains
  procedure :: add => integer_list_add_node
end type integer_list_t
Example: polymorphic linked list

Creating a child class: a list of integers

The class `integer_list_t` is a child of `linked_list_t`

```plaintext
type, extends(linked_list_t) :: integer_list_t  
  private  
  contains  
    procedure :: add => integer_list_add_node  
end type integer_list_t
```
Example: polymorphic linked list

Creating a child class: a list of integers

type, extends(linked_list_t) :: integer_list_t
  private
  contains
    procedure :: add => integer_list_add_node
end type integer_list_t
Example: polymorphic linked list

Creating a child class: a list of integers

type, extends(linked_list_t) :: integer_list_t
  private
  contains
    procedure :: add => integer_list_add_node
  end type integer_list_t

subroutine integer_list_add_node(this, value)
  class(integer_list_t), intent(inout) :: this
  integer, target :: value

  call this%add_copy(value)
end subroutine integer_list_add_node
Creating a child class: a list of integers

type, extends(linked_list_t) :: integer_list_t
  private
contains
  procedure :: add => integer_list_add_node
end type integer_list_t

subroutine integer_list_add_node(this, value)
  class(integer_list_t), intent(inout) :: this
  integer, target :: value

  call this%add_copy(value)
end subroutine integer_list_add_node

Here we call the routine of the parent class, which takes a class(*) argument
→ code reused !
→ abstraction
→ encapsulation
Only integers can be added. Avoids misuses of the list of integers.

Behind this line: 35 lines of fully generic code.
From the linked list to a time propagator

In essence, a time propagator is nothing but an algorithm, a list of operators

In Octopus, \textit{propagor\_t} extends \textit{linked\_list\_t}

! function propagator\_verlet\_constructor(dt) result(this)
  FLOAT, intent(in) :: dt
  type(propagator\_verlet\_t), pointer :: this

PUSH\_SUB(propagator\_verlet\_constructor)

SAFE\_ALLOCATE(this)

this\%start\_step = OP\_VERLET\_START
this\%final\_step = OP\_VERLET\_FINISH

call this\%add\_operation(OP\_VERLET\_UPDATE\_POS)
call this\%add\_operation(OP\_UPDATE\_INTERACTIONS)
call this\%add\_operation(OP\_VERLET\_COMPUTE\_ACC)
call this\%add\_operation(OP\_VERLET\_COMPUTE\_VEL)
call this\%add\_operation(OP\_FINISHED)

! Verlet has only one algorithmic step
this\%algo\_steps = 1

this\%dt = dt

POP\_SUB(propagator\_verlet\_constructor)
end function propagator\_verlet\_constructor
From the linked list to a time propagator

In essence, a time propagator is nothing but an algorithm, a list of operators.

In Octopus, `propagor_t` extends `linked_list_t`
From the linked list to a time propagator

In essence, a time propagator is nothing but an algorithm, a list of operators.

In Octopus, `propagor_t` extends `linked_list_t`.

```fortran
! function propagator_verlet_constructor(dt) result(this)
FLOAT, intent(in) :: dt
 type(propagator_verlet_t), pointer :: this

PUSH_SUB(propagator_verlet_constructor)

SAFE_ALLOCATE(this)

this%start_step = OP_VERLET_START
this%final_step = OP_VERLET_FINISH

call this%add_operation(OP_VERLET.UPDATE_POS)
call this%add_operation(OP_UPDATE_INTERACTIONS)
call this%add_operation(OP_VERLET.COMPUTE_ACC)
call this%add_operation(OP_VERLET.COMPUTE_VEL)
call this%add_operation(OP_FINISHED)

! Verlet has only one algorithmic step
this%algo_steps = 1

this%dt = dt

POP_SUB(propagator_verlet_constructor)
```

Here we add elements to our list. These are “algorithmic steps”.
From the linked list to a time propagator

In essence, a time propagator is nothing but an algorithm, a list of operators

In Octopus, `propagor_t` extends `linked_list_t`

```fortran
! function propagator_verlet_constructor(dt) result(this)
  FLOAT, intent(in) :: dt
  type(propagator_verlet_t), pointer :: this

  PUSH_SUB(propagator_verlet_constructor)

  SAFE_ALLOCATE(this)

  this%start_step = OP_VERLET_START
  this%final_step = OP_VERLET_FINISH

  call this%add_operation(OP_VERLET_UPDATE_POS)
  call this%add_operation(OP_UPDATE_INTERACTIONS)
  call this%add_operation(OP_VERLET_COMPUTE_ACC)
  call this%add_operation(OP_VERLET_COMPUTE_VEL)
  call this%add_operation(OP_FINISHED)

  ! Verlet has only one algorithmic step
  this%algo_steps = 1

  this*dt = dt

  POP_SUB(propagator_verlet_constructor)
end function propagator_verlet_constructor
```

We can “read” the algorithm directly
Easier to debug!
From the linked list to a time propagator

In essence, a time propagator is nothing but an algorithm, a list of operations.

In Octopus, `propagor_t` extends `linked_list_t`.

```fortran
! function propagator_verlet_constructor(dt) result(this)
  FLOAT,       intent(in) :: dt
  type(propagator_verlet_t), pointer :: this

  PUSH_SUB(propagator_verlet_constructor)

  SAFE_ALLOCATE(this)

  this%start_step = OP_VERLET_START
  this%final_step = OP_VERLET_FINISH

  call this%add_operation(OP_VERLET_UPDATE_POS)
  call this%add_operation(OP_UPDATE_INTERACTIONS)
  call this%add_operation(OP_VERLET_COMPUTE_ACC)
  call this%add_operation(OP_VERLET_COMPUTE_VEL)
  call this%add_operation(OP_FINISHED)

  ! Verlet has only one algorithmic step
  this%algo_steps = 1

  this*dt = dt

  POP_SUB(propagator_verlet_constructor)
end function propagator_verlet_constructor
```

Some steps are generic.

Code reusability.
From the linked list to a time propagator
Knowing the type of an polymorphic object

Let’s come back to our list of integer:

```fortran
! function integer_iterator_get_next(this) result(value)
   class(integer_iterator_t), intent(inout) :: this
   integer                                  :: value

   select type (ptr => this%get_next_ptr())
   type is (integer)
      value = ptr
   class default
      ASSERT(.false.)
   end select

end function integer_iterator_get_next
```
Knowing the type of an polymorphic object

Let’s come back to our list of integer

```fortran
! function integer_iterator_get_next(this) result(value)
  class(integer_iterator_t), intent(inout) :: this
  integer                                  :: value

  select type (ptr => this%get_next_ptr())
    type is (integer)
      value = ptr
    class default
      ASSERT(.false.)
  end select

end function integer_iterator_get_next
```

get_next_ptr is a generic routine that returns a class(*) object...
Knowing the type of an polymorphic object

Let’s come back to our list of integer numbers.

```fortran
function integer_iterator_get_next(this) result(value)
  class(integer_iterator_t), intent(inout) :: this
  integer :: value
  select type (ptr => this%get_next_ptr())
  type is (integer)
    value = ptr
  class default
    ASSERT(.false.)
  end select
end function integer_iterator_get_next
```

Here we use `select type` to test the type of the object.
Knowing the type of an polymorphic object

Let’s come back to our list of integer

```fortran
! function integer_iterator_get_next(this) result(value)
 class(integer_iterator_t), intent(inout) :: this
 integer                                  :: value

 select type (ptr => this%get_next_ptr())
 type is (integer)
 value = ptr
 class default
 ASSERT(.false.)
 end select

 end function integer_iterator_get_next
```

Here we use `select type` to test the type of the object

Below this point, `ptr` is an integer