Chapter 28
Introduction to Object Oriented Programming

For Madmen only
Hermann Hesse, Steppenwolf

Aims
The aims of this chapter are to look at object oriented programming in Fortran.

28.1 Introduction

This chapter looks at object oriented programming in Fortran. The chapter on programming languages covers the topic in a broader context.

28.2 Brief Review of the History of Object Oriented Programming

Object oriented programming is not new. One of the first languages to offer support was Simula 67, a language designed for discrete event simulation by Ole Johan Dahl, Bjorn Myhrhaug and Kristen Nygaard whilst working at the Norwegian Computing Centre in Oslo in the 1960's.

One of the next major developments was in the 1970's at the Xerox Palo Alto Research Centre Learning Research Group who began working on a vision of the ways different people might effectively use computing power. One of the outcomes of their work was the Smalltalk 80 system. Objects are at the core of the Smalltalk 80 system.

© Springer International Publishing AG, part of Springer Nature 2018

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_28

The 1980's and 1990's saw a number of object oriented programming languages emerge. They include

- Eiffel. Bertrand Meyer, Eiffel Software.
- C++ from C with classes. Bjarne Stroustrup at Bell Labs.
- Oberon 2. Niklaus Wirth at ETH in Zurich.
- Java. James Gosling, originally Sun, now Oracle.
- C# is a recent Microsoft addition to the list.

Object-oriented programming is effectively a programming methodology or paradigm using objects (data structures made up of data and methods). We will use the concept of a shape class in our explanation and examples. The Simula Begin book starts with shapes, and it is often used in introductions to object oriented programming in other languages.

Some of the key concepts are

- encapsulation or information hiding - the implementation of the data is hidden inside an object and clients or users of the data only have access to an abstract view of it. Methods are used to access and manipulate the data. For example a shape class may have an x and y position, and methods exist to get and set the positions and draw and move the shape.
- data abstraction - if we have an abstract shape data type we can create multiple variables of that type.
- inheritance - an existing abstract data type can be extended. It will inherit the data and methods from the base type and add additional data and methods. A key to inheritance is that the extended type is compatible with the base type. Anything that works with objects or variables of the base type also works with objects of the extended type. A circle would have a radius in addition to an x and y position, a rectangle would have a width and height.
- dynamic binding - if we have a base shape class and derive circles and rectangles from it dynamic binding ensures that the correct method to calculate the area is called at run time.
- polymorphism - variables can therefore be polymorphic. Using the shape example we can therefore create an array of shapes, one may be a shape, one may be a circle and another may be a rectangle.

Extensible abstract data types with dynamically bound methods are often called classes. This is the terminology we will use in what follows.

28.3 Background Technical Material

We need to look more formally at a number of concepts so that we can actually do object oriented programming in Fortran. The following sections cover some of the introductory material we need, and are taken from the standard.

28.3.1 The Concept of Type

Fortran provides an abstract means whereby data can be categorized without relying on a particular physical representation. This abstract means is the concept of type. A type has a name, a set of valid values, a means to denote such values (constants), and a set of operations to manipulate the values.

28.3.2 Type Classification

A type is either an intrinsic type or a derived type. This document defines five intrinsic types: integer, real, complex, character, and logical. A derived type is one that is defined by a derived-type definition (7.5.2) or by an intrinsic module. It shall be used only where it is accessible (7.5.2.2). An intrinsic type is always accessible.

28.3.3 Set of Values

For each type, there is a set of valid values. The sets of valid values for integer, character, and real are processor dependent. The set of valid values for complex consists of the set of all the combinations of the values of the real and imaginary parts. The set of valid values for a derived type is as defined in 7.5.8.

28.3.4 Type

A `type` type specifier is used to declare entities that are assumed-type, or of an intrinsic or derived type.

An entity that is declared using the TYPE(*) type specifier is assumed-type and is an unlimited polymorphic entity. It is not declared to have a type, and is not considered to have the same declared type as any other entity, including another unlimited polymorphic entity. Its dynamic type and type parameters are assumed from its effective argument.

28.3.5 Class

The CLASS type specifier is used to declare polymorphic entities. A polymorphic entity is a data entity that is able to be of differing dynamic types during program execution.

The declared type of a polymorphic entity is the specified type if the CLASS type specifier contains a type name.

An entity declared with the CLASS(*) specifier is an unlimited polymorphic entity. It is not declared to have a type, and is not considered to have the same declared type as any other entity, including another unlimited polymorphic entity.

28.3.6 Attributes

The additional attributes that may appear in the attribute specification of a type declaration statement further specify the nature of the entities being declared or specify restrictions on their use in the program.

28.3.6.1 Accessibility Attribute

The accessibility attribute specifies the accessibility of an entity via a particular identifier. The following is taken from Sect. 8.5.2 of the Fortran 2018 standard.

- access-spec is `public` or `private`
- An access-spec shall appear only in the specification-part of a module.

Identifiers that are specified in a module or accessible in that module by use association have either the `public` or `private` attribute. Identifiers for which an access-spec is not explicitly specified in that module have the default accessibility attribute for that module. The default accessibility attribute for a module is `public` unless it has been changed by a `private` statement. Only identifiers that have the `public` attribute in that module are available to be accessed from that module by use association.

28.3.7 Passed Object Dummy Arguments

Section 3.107 of the Fortran 2018 standard introduces the concept of passed object dummy argument. Here is an extract from the standard:

- A passed-object dummy argument is a distinguished dummy argument of a procedure pointer component or type-bound procedure (7.5.5). It affects procedure overriding (7.5.7.3) and argument association (15.5.2.2).
- If NOPASS is specified, the procedure pointer component or type-bound procedure has no passed-object dummy argument.
- If neither PASS nor NOPASS is specified or PASS is specified without arg-name, the first dummy argument of a procedure pointer component or type-bound procedure is its passed-object dummy argument.

- If PASS (arg-name) is specified, the dummy argument named arg-name is the passed-object dummy argument of the procedure pointer component or named type-bound procedure.
- Constraint C761 The passed-object dummy argument shall be a scalar, nonpointer, nonallocatable dummy data object with the same declared type as the type being defined; all of its length type parameters shall be assumed; it shall be polymorphic (7.3.2.3) if and only if the type being defined is extensible (7.5.7). It shall not have the VALUE attribute.

The key here is that we are going to use the `pass` and `nopass` attributes with type bound procedures - a component of object oriented programming in Fortran.

28.3.8 Derived Types and Structure Constructors

A derived type is a type that is not defined by the language but requires a type definition to declare its components. A scalar object of such a derived type is called a structure. Assignment of structures is defined intrinsically, but there are no intrinsic operations for structures. For each derived type, a structure constructor is available to provide values.

A derived-type definition implicitly defines a corresponding structure constructor that allows construction of values of that derived type.

28.3.9 Structure Constructors and Generic Names

A generic name may be the same as a type name. This can be used to emulate user-defined structure constructors for that type, even if the type has private components. The following example is taken from the standard to illustrate this.

```
module mytype_module
type mytype
  private
    complex value
    logical exact
end type
interface mytype
  module procedure int_to_mytype
end interface
! Operator definitions etc.
...
contains
  type(mytype) function int_to_mytype(i)
    integer,intent(in) :: i
    int_to_mytype%value = i
```

```
      int_to_mytype%exact = .true.
    end function
  ! Procedures to support operators etc.
  ...
  end
```

28.3.10 Assignment

Execution of an assignment statement causes a variable to become defined or redefined. Simplistically

```
variable = expression
```

28.3.11 Intrinsic Assignment Statement

An intrinsic assignment statement is an assignment statement that is not a defined assignment statement (10.2.1.4). In an intrinsic assignment statement,

- if the variable is polymorphic it shall be allocatable and not a coarray,
- if expr is an array then the variable shall also be an array,
- the variable and expr shall be conformable unless the variable is an allocatable array that has the same rank as expr and is not a coarray,
- if the variable is polymorphic it shall be type compatible with expr; otherwise the declared types of the variable and expr shall conform as specified in Table 10.8 of the standard,
- if the variable is of type character and of ISO 10646, ASCII, or default character kind, expr shall be of ISO 10646, ASCII, or default character kind,
- otherwise if the variable is of type character expr shall have the same kind type parameter,
- if the variable is of derived type each kind type parameter of the variable shall have the same value as the corresponding kind type parameter of expr, and
- if the variable is of derived type each length type parameter of the variable shall have the same value as the corresponding type parameter of expr unless the variable is allocatable, is not a coarray, and its corresponding type parameter is deferred.

28.3.12 Defined Assignment Statement

A defined assignment statement is an assignment statement that is defined by a subroutine and a generic interface that specifies ASSIGNMENT (=).

28.3.13 Polymorphic Variables

Here are some of the technical definitions regarding polymorphic taken from the standard.

- polymorphic - polymorphic data entity able to be of differing dynamic types during program execution (7.3.2.3)
- unlimited polymorphic - able to have any dynamic type during program execution (7.3.2.3)

A polymorphic variable must be a pointer or allocatable variable. We will use allocatable variables to achieve polymorphism in our examples.

28.3.14 Executable Constructs Containing Blocks

The following are executable constructs that contain blocks:

- `associate` construct
- `case` construct
- `do` construct
- `if` construct
- `select type` construct

We will look at the `associate` construct and `select type` construct next.

28.3.15 The `associate` Construct

The `associate` construct associates named entities with expressions or variables during the execution of its block. These named construct entities are associating entities. The names are associate names.

The following example illustrates an association with a derived-type variable.

```
associate ( xc => ax%b(i,i)%c )
xc%dv = xc%dv + product(xc%ev(1:n))
end associate
```

28.3.16 The `select type` Construct

The `select type` construct selects for execution at most one of its constituent blocks. The selection is based on the dynamic type of an expression. A name is associated with the expression, in the same way as for the `associate` construct.

Quite a lot to take in! Let's illustrate the use of the above in some actual examples.

28.4 Example 1: The Basic Shape Class

The code for the base shape class is given below.

- shape class data: integer variables `x` and `y` for the position.
- shape class methods: `get` and `set` for the `x` and `y` values, and `moveto` and `draw`.

We have used an include statement in the examples that follow to reduce code duplication. In this example we have used the default accessibility for the data and methods in the `shape_module`.

```
module shape_module

  type shape_type

    integer :: x_ = 0
    integer :: y_ = 0

  contains

    procedure, pass (this) :: get_x
    procedure, pass (this) :: get_y
    procedure, pass (this) :: set_x
    procedure, pass (this) :: set_y
    procedure, pass (this) :: moveto
    procedure, pass (this) :: draw

  end type shape_type

contains

  include 'shape_module_include_code.f90'

end module shape_module
```

Here is the code in the include file.

```
!start shape_module_common_code
integer function get_x(this)
  implicit none
  class (shape_type), intent (in) :: this

  get_x = this%x_
end function get_x
```

```
integer function get_y(this)
  implicit none
  class (shape_type), intent (in) :: this

  get_y = this%y_
end function get_y

subroutine set_x(this, x)
  implicit none
  class (shape_type), intent (inout) :: this
  integer, intent (in) :: x

  this%x_ = x
end subroutine set_x

subroutine set_y(this, y)
  implicit none
  class (shape_type), intent (inout) :: this
  integer, intent (in) :: y

  this%y_ = y
end subroutine set_y

subroutine moveto(this, newx, newy)
  implicit none
  class (shape_type), intent (inout) :: this
  integer, intent (in) :: newx
  integer, intent (in) :: newy

  this%x_ = newx
  this%y_ = newy
end subroutine moveto

subroutine draw(this)
  implicit none
  class (shape_type), intent (in) :: this

  print *, ' x = ', this%x_
  print *, ' y = ', this%y_
end subroutine draw
!end shape_module_common_code
```

28.4.1 Key Points

Some of the key concepts are:

- We use a module as the organisational unit for the class.
- We use `type` and `end type` to contain the data and the procedures - called type bound procedures in Fortran terminology.
- The data in the base class is an x and y position.
- The type bound methods within the class are
  - `get_x` and `set_x`
  - `get_y` and `set_y`
  - `draw`
  - `moveto`
- We have used the default accessibility for the data and methods in the type.

Let us look at the code in stages.

```
module shape_module
```

The module is called `shape_module`

```
  type shape_type
```

The type is called `shape_type`

```
    integer :: x_ = 0
    integer :: y_ = 0
```

The data associated with the shape type are integer variables that are the x and y coordinates of the shape. We initialise to zero.

```
  contains
```

The type also contains procedures or methods.

```
    procedure, pass(this) :: get_x
    procedure, pass(this) :: get_y
    procedure, pass(this) :: set_x
    procedure, pass(this) :: set_y
    procedure, pass(this) :: moveto
    procedure, pass(this) :: draw
```

These are called type bound procedures in Fortran terminology. It is common in object oriented programming to have get and set methods for each of the data components of the type or object. We also have a `moveto` and `draw` method.

Each of these methods has the `pass` attribute. When a type bound procedure is called or invoked the object through which is invoked is normally passed as a hidden parameter. We have used the `pass` attribute to explicitly confirm the default behaviour of passing the invoking object as the first parameter. We have also followed the convention in object oriented programming of using the word `this` to refer to the current object.

```
  end type shape_type
```

This is the end of the type definition.

```
contains
```

The module then contains the actual implementation of the type bound procedures. We will look at a couple of these.

```
  integer function get_x(this)
    implicit none
    class (shape_type), intent (in) :: this
      get_x = this%x_
  end function get_x
```

As we stated earlier it is common in object oriented programming to have get and set methods for each data item in an object. This function implements the `get_x` method. The first argument is the current object, referred to as this. We then have the type declaration for this parameter. We declare the variable using class rather than type as we want the variable to be polymorphic. The rest of the function is self explanatory.

```
  subroutine set_x(this,x)
    implicit none
    class (shape_type), intent (inout) :: this
    integer, intent (in) :: x
      this%x_ = x
  end subroutine set_x
```

The `set_x` procedure is a subroutine. It takes two parameters, the current object and the new x value. Again we use the class declaration mechanism as we want the variable to be polymorphic.

Here is a program to test the above shape module out.

```
include 'ch2801_shape_module.f90'

program ch2801
  use shape_module
  implicit none
  type (shape_type) :: s1 = shape_type(10, 20)
  integer :: x1 = 100
  integer :: y1 = 200

  print *, ' get '
  print *, s1%get_x(), ' ', s1%get_y()
  print *, ' draw '
  call s1%draw()
  print *, ' moveto '
  call s1%moveto(x1, y1)
  print *, ' draw '
  call s1%draw()
  print *, ' set '
  call s1%set_x(99)
  call s1%set_y(99)
  print *, ' draw'
  call s1%draw()
end program ch2801
```

The first statement of interest is the `use` statement, where we make available the `shape_module` to the test program. The next statement of interest is

```
type (shape_type) :: s1 = shape_type(10,20)
```

We then have a type declaration for the variable `s1`. We also have the use of what Fortran calls a structure constructor `shape_type` to provide initial values to the x and y positions. The term constructor is used in other object oriented programming languages, e.g. C++, Java, C#. It has the same name as the type or class and is created automatically for us by the compiler in this example.

The

```
print *, s1%get_x(), ' ', s1%get_y()
```

statement prints out the `x` and`y` values for the object `s1`. We use the standard % notation that we used in derived types, to separate the components of the derived types. If one looks at the implementation of the `get_x` function and examines the first line, repeated below

```
integer function get_x(this)
```

how we refer to the current object, `s1`, through the syntax `s1%get_x()`. The following call:

```
call s1%draw()
```

shows how to invoke the draw method for the `s1` object, using the `s1%draw()` syntax. The first line of the draw subroutine

```
subroutine draw(this)
```

shows how the current object is passed as the first argument.

28.4.2 Notes

In this example we have accepted the default Fortran accessibility behaviour. This means that we can use the compiler provided structure constructor `shape_type()`

```
type (shape_type) :: s1 = shape_type(10,20)
```

in the type declaration to provide initial values, as they are public by default. Direct access to the data is often not a good idea, as it is possible to makes changes to the data anywhere in the program. The next example makes the data private.

28.5 Example 2: Base Class with Private Data

Here is the modified base class.

```
module shape_module

  type shape_type

    integer, private :: x_ = 0
    integer, private :: y_ = 0

  contains

    procedure, pass (this) :: get_x
    procedure, pass (this) :: get_y
    procedure, pass (this) :: set_x
    procedure, pass (this) :: set_y
    procedure, pass (this) :: moveto
```

```
      procedure, pass (this) :: draw

    end type shape_type

  contains

    include 'shape_module_include_code.f90'

  end module shape_module
```

Here is the diff output between the two shape modules.

```
5,6c5,6
<     integer :: x_ = 0
<     integer :: y_ = 0
---
>     integer, private :: x_ = 0
>     integer, private :: y_ = 0
```

This example will now not compile as the default compiler provided structure constructor does not have access to the private data.

The test program is the same as in the first example.

Here is the output from trying to compile this example.

```
Error: ch2802.f90, line 4:
Constructor for type SHAPE_TYPE has value
for PRIVATE component X_
Errors in declarations,
no further processing for CH2802
[NAG Fortran Compiler error termination, 1 error]
```

Not all compilers diagnose this problem. Test yours to see if you get an error message!

An earlier solution to this type of problem can be found in the date class in Chap. 22, where we provide our own structure constructor `date_()`. Most object oriented programming languages provide the ability to use the same name as a class as a constructor name even if the data is private. Modern Fortran provides another solution to this problem. In the example below we will provide our own structure constructor inside an interface.

28.6 Example 3: Using an Interface to Use the Class Name for the Structure Constructor

Here is the modified base class.

```
module shape_module

  type shape_type

    integer, private :: x_ = 0
    integer, private :: y_ = 0

  contains

    procedure, pass (this) :: get_x
    procedure, pass (this) :: get_y
    procedure, pass (this) :: set_x
    procedure, pass (this) :: set_y
    procedure, pass (this) :: moveto
    procedure, pass (this) :: draw

  end type shape_type

  interface shape_type
    module procedure shape_type_constructor
  end interface shape_type

contains

  type (shape_type) function &
    shape_type_constructor(x, y)
    implicit none
    integer, intent (in) :: x
    integer, intent (in) :: y

    shape_type_constructor%x_ = x
    shape_type_constructor%y_ = y
  end function shape_type_constructor

  include 'shape_module_include_code.f90'

end module shape_module
```

Here is the diff output between the second and third shape modules.

```
18a19,22
>   interface shape_type
>     module procedure shape_type_constructor
>   end interface shape_type
>
19a24,33
>
>   type (shape_type) function &
>     shape_type_constructor(x, y)
>     implicit none
>     integer, intent (in) :: x
>     integer, intent (in) :: y
>
>     shape_type_constructor%x_ = x
>     shape_type_constructor%y_ = y
>   end function shape_type_constructor
```

The key statements are

```
interface shape_type
  module procedure shape_type_constructor
end interface
```

which enables us to map a call or reference to `shape_type` (our structure constructor name) to our implementation of `shape_type_constructor`. Here is the implementation of this structure constructor.

```
type (shape_type) function &
  shape_type_constructor(x,y)
  implicit none
  integer, intent (in) :: x
  integer, intent (in) :: y
  shape_type_constructor%x_ = x
  shape_type_constructor%y_ = y
end function shape_type_constructor
```

The function is called `shape_type_constructor` hence we use this name to initialise the components of the type, and the function returns a value of type `shape_type`.

Here is the program to test the above out.

```
include 'ch2803_shape_module.f90'

program ch2803
```

```
    use shape_module
    implicit none
    type (shape_type) :: s1
    integer :: x1 = 100
    integer :: y1 = 200

    s1 = shape_type(10, 20)
    print *, ' get '
    print *, s1%get_x(), ' ', s1%get_y()
    print *, ' draw '
    call s1%draw()
    print *, ' moveto '
    call s1%moveto(x1, y1)
    print *, ' draw '
    call s1%draw()
    print *, ' set '
    call s1%set_x(99)
    call s1%set_y(99)
    print *, ' draw'
    call s1%draw()
  end program ch2803
```

Note that in this example we cannot initialise `s1` at definition time using our own (user defined) structure constructor. This must now be done within the execution part of the program. This is a Fortran restriction, and makes it consistent with the rest of the language.

These examples illustrate some of the basics of object oriented programming in Fortran. To summarise

- the data in our class is private;
- access to the data is via get and set methods;
- the data and methods are within the derived type definition - the methods are called type bound procedures in Fortran terminology;
- we can use interfaces to provide user defined structure constructors, which have the same name as the class - this is a common practice in object oriented programming;
- we have used class to declare the variables within the type bound methods. We need to use class when we want to use polymorphic variables in Fortran.

28.6.1 Public and Private Accessibility

We have only made the internal data in the class private in the above example. There will be cases where some of the methods are only used within the class, in which case they can be made private.

28.7 Example 4: Simple Inheritance

In this example we look at inheritance. We use the same base shape class and derive two classes from it - circle and rectangle.

A circle has a radius. This is the additional data component of the derived class. We also have get and set methods.

A rectangle has a width and height. These are the additional data components of the derived rectangle class. We also have get and set methods.

28.7.1 Base Shape Class

The base shape class is as in the previous example.

28.7.2 Circle - Derived Type 1

Here is the code.

```
module circle_module

  use shape_module

  type, extends (shape_type) :: circle_type

    integer, private :: radius_

  contains

    procedure, pass (this) :: get_radius
    procedure, pass (this) :: set_radius
    procedure, pass (this) :: draw => &
      draw_circle

  end type circle_type

  interface circle_type
    module procedure circle_type_constructor
  end interface circle_type

contains
```

```
  type (circle_type) function &
    circle_type_constructor(x, y, radius)
    implicit none
    integer, intent (in) :: x
    integer, intent (in) :: y
    integer, intent (in) :: radius

    call circle_type_constructor%set_x(x)
    call circle_type_constructor%set_y(y)
    circle_type_constructor%radius_ = radius
  end function circle_type_constructor

  integer function get_radius(this)
    implicit none
    class (circle_type), intent (in) :: this

    get_radius = this%radius_
  end function get_radius

  subroutine set_radius(this, radius)
    implicit none
    class (circle_type), intent (inout) :: this
    integer, intent (in) :: radius

    this%radius_ = radius
  end subroutine set_radius

  subroutine draw_circle(this)
    implicit none
    class (circle_type), intent (in) :: this

    print *, ' x = ', this%get_x()
    print *, ' y = ', this%get_y()
    print *, ' radius = ', this%radius_
  end subroutine draw_circle

end module circle_module
```

Let us look more closely at the statements within this class. Firstly we have

```
module circle_module
```

which introduces our circle module. We then

```
use shape_module
```

within this module to make available the shape class. The next statement

```
type , extends(shape_type) :: circle_type
```

is the key statement in inheritance. What this statement says is base our new `circle_type` on the base `shape_type`. It is an extension of the `shape_type`. We then have the additional data in our `circle_type`

```
integer , private :: radius_
```

and the following additional type bound procedures.

```
procedure , pass(this) :: get_radius
procedure , pass(this) :: set_radius
procedure , pass(this) :: draw => draw_circle
```

and we have the simple get and set methods for the radius, and a type specific draw method for our `circle_type`. It is this method that will be called when drawing with a circle, rather than the `draw` method in the base `shape_type`.

We then have an interface to provide us with our own user defined structure constructor for our `circle_type`.

```
interface circle_type
  module procedure circle_type_constructor
end interface
```

As has been stated earlier it is common practice in object oriented programming to use the same name as the type for constructors.

We then have the implementation of the constructor.

```
type (circle_type) function &
  circle_type_constructor(x,y,radius)
  implicit none
  integer, intent (in) :: x
  integer, intent (in) :: y
  integer, intent (in) :: radius
  call circle_type_constructor%set_x(x)
  call circle_type_constructor%set_y(y)
  circle_type_constructor%radius_=radius
end function circle_type_constructor
```

Note that we use the `set_x` and `set_y` methods to provide initial values to the `x` and `y` values. They are private in the base class so we need to use these methods.

We can directly initialise the radius as this is a data component of this class, and we have access to it.

We next have the `get` and `set` methods for the radius.

Finally we have the implementation for the draw circle method.

```
subroutine draw_circle(this)
implicit none
  class (circle_type), intent(in) :: this
  print *,' x = ' , this%get_x()
  print *,' y = ' , this%get_y()
  print *,' radius = ' , this%radius_
end subroutine draw_circle
```

Notice again that we use the `get_x` and `get_y` methods to access the `x` and`y` private data from the base shape class.

28.7.3 Rectangle - Derived Type 2

Here is the code for the second derived type.

```
module rectangle_module

  use shape_module

  type, extends (shape_type) :: rectangle_type

    integer, private :: width_
    integer, private :: height_

  contains

    procedure, pass (this) :: get_width
    procedure, pass (this) :: set_width
    procedure, pass (this) :: get_height
    procedure, pass (this) :: set_height
    procedure, pass (this) :: draw => &
      draw_rectangle

  end type rectangle_type

  interface rectangle_type
    module procedure rectangle_type_constructor
```

```
  end interface rectangle_type

contains

  type (rectangle_type) function &
    rectangle_type_constructor(x, y, width, &
    height)
    implicit none
    integer, intent (in) :: x
    integer, intent (in) :: y
    integer, intent (in) :: width
    integer, intent (in) :: height

    call rectangle_type_constructor%set_x(x)
    call rectangle_type_constructor%set_y(y)
    rectangle_type_constructor%width_ = width
    rectangle_type_constructor%height_ = height
  end function rectangle_type_constructor

  integer function get_width(this)
    implicit none
    class (rectangle_type), intent (in) :: this

    get_width = this%width_
  end function get_width

  subroutine set_width(this, width)
    implicit none
    class (rectangle_type), intent (inout) :: &
      this
    integer, intent (in) :: width

    this%width_ = width
  end subroutine set_width

  integer function get_height(this)
    implicit none
    class (rectangle_type), intent (in) :: this

    get_height = this%height_
  end function get_height

  subroutine set_height(this, height)
    implicit none
    class (rectangle_type), intent (inout) :: &
```

```
      this
    integer, intent (in) :: height

    this%height_ = height
  end subroutine set_height

  subroutine draw_rectangle(this)
    implicit none
    class (rectangle_type), intent (in) :: this

    print *, ' x = ', this%get_x()
    print *, ' y = ', this%get_y()
    print *, ' width = ', this%width_
    print *, ' height = ', this%height_
  end subroutine draw_rectangle

end module rectangle_module
```

The code is obviously very similar to that of the first derived type.

28.7.4 Simple Inheritance Test Program

Here is a test program that illustrates the use of the shape type, circle type and rectangle type.

```
include 'ch2803_shape_module.f90'
include 'ch2804_circle_module.f90'
include 'ch2804_rectangle_module.f90'

program ch2804

  use shape_module
  use circle_module
  use rectangle_module

  implicit none

  type (shape_type) :: vs
  type (circle_type) :: vc
  type (rectangle_type) :: vr

  vs = shape_type(10, 20)
  vc = circle_type(100, 200, 300)
```

```
  vr = rectangle_type(1000, 2000, 3000, 4000)
  print *, ' get '
  print *, ' shape     ', vs%get_x(), ' ', &
    vs%get_y()
  print *, ' circle    ', vc%get_x(), ' ', &
    vc%get_y(), 'radius = ', vc%get_radius()
  print *, ' rectangle ', vr%get_x(), ' ', &
    vr%get_y(), 'width = ', vr%get_width(), &
    'height ', vr%get_height()
  print *, ' draw '
  call vs%draw()
  call vc%draw()
  call vr%draw()
  print *, ' set '
  call vs%set_x(19)
  call vs%set_y(19)
  call vc%set_x(199)
  call vc%set_y(199)
  call vc%set_radius(199)
  call vr%set_x(1999)
  call vr%set_y(1999)
  call vr%set_width(1999)
  call vr%set_height(1999)
  print *, ' draw '
  call vs%draw()
  call vc%draw()
  call vr%draw()
end program ch2804
```

The first statements of note are

```
use shape_module
use circle_module
use rectangle_module
```

which make available the shape, circle and rectangle types within the program. The following statements

```
type (shape_type)     :: vs
type (circle_type)    :: vc
type (rectangle_type) :: vr
```

declare `vs`, `vc` and `vr` to be of type shape, circle and rectangle respectively. The following three statements

```
vs = shape_type(10,20)
vc = circle_type(100,200,300)
vr = rectangle_type(1000,2000,3000,4000)
```

call the three user defined structure constructor functions.

We then use the `get` functions to print out the values of the private data in each object.

```
print *,' shape     ', vs%get_x(),&
  ' ',vs%get_y()
print *,' circle    ', vc%get_x(),&
  ' ',vc%get_y(),' radius = ',vc%get_radius()
print *,' rectangle ', vr%get_x(),&
  ' ',vr%get_y(),' width  = ',vr%get_width(),'
height ',vr%get_height()
```

We then call the `draw` method for each type.

```
call vs%draw()
call vc%draw()
call vr%draw()
```

and the appropriate draw method is called for each type. We finally call the `set` functions for each variable and repeat the calls to the draw methods.

The draw methods in the derived types override the draw method in the base shape class.

28.8 Example 5: Polymorphism and Dynamic Binding

An inheritance hierarchy can provide considerable flexibility in our ability to manipulate objects, whilst still taking advantage of static or compile time type checking. If we combine inheritance with polymorphism and dynamic binding we have a very powerful programming tool. We will illustrate this with a concrete example.

28.8.1 Base Shape Class

This is our base class. A polymorphic variable is a variable whose data type may vary at run time. It must be a pointer or allocatable variable, and it must be declared using the `class` keyword. Our original base class declared variables using the `class` keyword from the beginning as we always intended to design a class that could be polymorphic.

We have had to make one change to the previous one. To make the polymorphism work we have had to provide our own assignment operator. So we have

```
interface assignment (=)
  module procedure generic_shape_assign
end interface
```

which means that our implementation of `generic_shape_assign` will replace the intrinsic assignment. Here is the actual implementation.

```
subroutine generic_shape_assign(lhs,rhs)
  implicit none
  class (shape_type), intent (out), &
    allocatable :: lhs
  class (shape_type), intent (in) :: rhs
  allocate (lhs,source=rhs)
end subroutine generic_shape_assign
```

In an assignment we obviously have

```
  left_hand_side = right_hand_side
```

and in our code we have variables `lhs` and `rhs` to clarify what is happening. We also have an enhanced form of allocation statement:

```
  allocate (lhs,source=rhs)
```

and the key is that the left hand side variable is allocated with the values and type of the right hand side variable. Here is the complete code.

```
module shape_module

  type shape_type

    integer, private :: x_ = 0
    integer, private :: y_ = 0

  contains

    procedure, pass (this) :: get_x
    procedure, pass (this) :: get_y
    procedure, pass (this) :: set_x
    procedure, pass (this) :: set_y
    procedure, pass (this) :: moveto
```

```
      procedure, pass (this) :: draw

    end type shape_type

    interface shape_type
      module procedure shape_type_constructor
    end interface shape_type

    interface assignment (=)
      module procedure generic_shape_assign
    end interface assignment (=)

  contains

    type (shape_type) function &
      shape_type_constructor(x, y)
      implicit none
      integer, intent (in) :: x
      integer, intent (in) :: y

      shape_type_constructor%x_ = x
      shape_type_constructor%y_ = y
    end function shape_type_constructor

    include 'shape_module_include_code.f90'

    subroutine generic_shape_assign(lhs, rhs)
      implicit none
      class (shape_type), intent (out), &
        allocatable :: lhs
      class (shape_type), intent (in) :: rhs

      allocate (lhs, source=rhs)
    end subroutine generic_shape_assign

  end module shape_module
```

28.8.2 Circle - Derived Type 1

The circle code is the same as before.

28.8.3 Rectangle - Derived Type 2

The rectangle code is as before.

28.8.4 Shape Wrapper Module

As was stated earlier a polymorphic variable must be a pointer or allocatable variable. We have chosen to go the allocatable route. The following is a wrapper routine to allow us to have a derived type whose types can be polymorphic.

```
module shape_wrapper_module
  use shape_module
  use circle_module
  use rectangle_module
  type shape_wrapper

    class (shape_type), allocatable :: x
  end type shape_wrapper
end module shape_wrapper_module
```

So now `x` can be of `shape_type` or of any type derived from `shape_type`. Don't panic if this isn't clear at the moment, the complete program should help out!

28.8.5 Display Subroutine

This is the key subroutine in this example. We can pass into this routine an array of type `shape_wrapper`. In the code so far we have variables of type

- `shape_type`
- `circle_type`
- `rectangle_type`

and we are passing in an array of elements and each element can be of any of these types, i.e. the `shape_array` is polymorphic.

The next statement of interest is

```
call shape_array(i)%x%draw()
```

and at run time the correct `draw` method will be called. This is called dynamic binding. Here is the complete code.

```
module display_module

contains

  subroutine display(n_shapes, shape_array)
    use shape_wrapper_module
    implicit none
    integer, intent (in) :: n_shapes
    type (shape_wrapper), dimension (n_shapes) &
      :: shape_array
    integer :: i

    do i = 1, n_shapes
      call shape_array(i)%x%draw()
    end do
  end subroutine display

end module display_module
```

28.8.6 Test Program for Polymorphism and Dynamic Binding

We now have the complete program that illustrates polymorphism and dynamic binding in action.

```
include 'ch2805_shape_module.f90'
include 'ch2804_circle_module.f90'
include 'ch2804_rectangle_module.f90'
include 'ch2805_shape_wrapper_module.f90'
include 'ch2805_display_module.f90'

program ch2805
  use shape_module
  use circle_module
  use rectangle_module
  use shape_wrapper_module
  use display_module
  implicit none
  integer, parameter :: n = 6
  integer :: i
  type (shape_wrapper), dimension (n) :: s

  s(1)%x = shape_type(10, 20)
```

```
s(2)%x = circle_type(100, 200, 300)
s(3)%x = rectangle_type(1000, 2000, 3000, &
  4000)
s(4)%x = s(1)%x
s(5)%x = s(2)%x
s(6)%x = s(3)%x
print *, ' calling display subroutine'
call display(n, s)
print *, ' select type with get methods'
do i = 1, n
  select type (t=>s(i)%x)
  class is (shape_type)
    print *, ' x = ', t%get_x(), ' y = ', &
      t%get_y()
  class is (circle_type)
    print *, ' x = ', t%get_x(), ' y = ', &
      t%get_y()
    print *, ' radius = ', t%get_radius()
  class is (rectangle_type)
    print *, ' x = ', t%get_x(), ' y = ', &
      t%get_y()
    print *, ' height = ', t%get_height()
    print *, ' width = ', t%get_width()
  class default
    print *, ' do nothing'
  end select
end do
print *, ' select type with set methods'
do i = 1, n
  select type (t=>s(i)%x)
  class is (shape_type)
    call t%set_x(19)
    call t%set_y(19)
  class is (circle_type)
    call t%set_x(199)
    call t%set_y(199)
    call t%set_radius(199)
  class is (rectangle_type)
    call t%set_x(1999)
    call t%set_y(1999)
    call t%set_height(1999)
    call t%set_width(1999)
  class default
    print *, ' do nothing'
  end select
```

```
    end do
    print *, ' calling display subroutine'
    call display(n, s)
  end program ch2805
```

Let us look at the key statements in more detail.

```
    type (shape_wrapper), dimension (n) :: s
```

This is the key declaration statement. `s` will be our polymorphic array. The following six assignment statements

```
    s(1) %x = shape_type(10,20)
    s(2) %x = circle_type(100,200,300)
    s(3) %x = rectangle_type(1000,2000,3000,4000)
    s(4) %x = s(1)%x
    s(5) %x = s(2)%x
    s(6) %x = s(3)%x
```

will call our own assignment subroutine to do the assignment. The allocation is hidden in the implementation. We then have

```
    call display(n,s)
```

which calls the display subroutine. The compiler at run time works out which draw method to call depending of the type of the elements in the shape_wrapper array.

Imagine now adding another shape type, let us say a triangle. We need to do the following

- inherit from the base shape type
- add the additional data to define a triangle
- add the appropriate get and set methods
- add a draw triangle method
- add a use statement to the shape_wrapper_module
- add a use statement to the main program

and we now can work with the new triangle shape type. The display subroutine is unchanged! We can repeat the above steps for any additional shape type we want. Polymorphism and dynamic binding thus shorten our development and maintenance time, as it reduces the amount of code we need to write and test.

We then have an example of the use of the `select type` statement. The compiler determines the type of the elements in the array and then executes the matching block.

```
do i=1,n
  select type ( t=>s(i) %x )
    class is (shape_type)
      print *,' x =        ',   t%get_x(),' y = ',t%get_y()
    class is (circle_type)
      print *,' x =        ',   t%get_x(),' y = ',t%get_y()
      print *,' radius =   ',   t%get_radius()
    class is (rectangle_type)
      print *,' x =        ',   t%get_x(),' y = ',t%get_y()
      print *,' height =   ',   t%get_height()
      print *,' width =    ',   t%get_width()
    class default
      print *,' do nothing'
  end select
end do
```

Now imagine adding support for the new triangle type. Anywhere we have select type constructs we have to add support for our new triangle shape. There is obviously more work involved when we use the `select type` construct in our polymorphic code. However some problems will be amenable to polymorphism and dynamic binding, others will require the explicit use of `select type` statements. This example illustrates the use of both.

28.9 Fortran 2008 and Polymorphic Intrinsic Assignment

The previous example works with Fortran 2003 conformant compilers. This example illustrates a simple variant that will work if your compiler supports a feature from the 2008 standard - polymorphic intrinsic assignment. In this case we do not need to provide a user defined assignment subroutine.

Here is the modified shape module.

```
module shape_module

  type shape_type

    integer, private :: x_ = 0
    integer, private :: y_ = 0

  contains

    procedure, pass (this) :: get_x
    procedure, pass (this) :: get_y
    procedure, pass (this) :: set_x
```

```
    procedure, pass (this) :: set_y
    procedure, pass (this) :: moveto
    procedure, pass (this) :: draw

  end type shape_type

  interface shape_type
    module procedure shape_type_constructor
  end interface shape_type

contains

  type (shape_type) function &
    shape_type_constructor(x, y)
    implicit none
    integer, intent (in) :: x
    integer, intent (in) :: y

    shape_type_constructor%x_ = x
    shape_type_constructor%y_ = y
  end function shape_type_constructor

  include 'shape_module_include_code.f90'

end module shape_module
```

The rest of the code is the same as in the previous example.

Compiling with gfortran 6.4 will generate the following error message.

Error: Assignment to an allocatable polymorphic variable at (1) is not yet supported

We maintain compiler standard conformance tables that document what features from the 2003, 2008 and 2018 standards are supported by current compilers.

Visit

```
https://www.fortranplus.co.uk/fortran-information/
```

to get up to date information. At the time of writing Table 28.1 was correct for compilers we have used in this edition.

Table 28.1 Polymorphic intrinsic assignment support

Compiler	Version	Assignment support
Cray	7.4	Yes
gfortran	4.x	No
	5.x	No
	6.x	No
	7.1	Yes
Intel	17.x	No
	18.x	Yes
Nag	6.0	Yes
Oracle	12.6	No
Pathscale	6.0.1148	No
PGI	17.4.0	No

28.10 Summary

This chapter has introduced some of the essentials of object oriented programming. The first example looked at object oriented programming as an extension of basic data structuring. We used type bound procedures to implement our shape class. We used methods to access the internal data of the shape object.

The second example looked at simple inheritance. We saw in this example how we could reuse the methods from the base class and also add new data and methods specific to the new shapes - circles and rectangles.

The third example then looked at how to achieve polymorphism in Fortran. We could then create arrays of our base type and dynamically bind the appropriate methods at run rime. Dynamic binding is needed when multiple classes contain different implementations of the same method, i.e. to ensure in the following code

```
call shape_array(i) %x%draw()
```

that the correct draw method is invoked on the shape object.

28.11 Problems

28.1 Compile and run all of the examples in this chapter with your compiler.

28.2 Add a triangle type to the simple inheritance example.

28.3 Add a triangle type to the polymorphic example.

28.12 Further Reading

The following book

ISO/IEC DIS 1539-1 Information technology–Programming languages–Fortran–Part 1: Base language

- Fortran 2018 draft standard.

```
https://www.iso.org/standard/72320.html
```

- Rouson D., Xia J., Xu X., Scientific Software Design: The Object Oriented Way, Cambridge University Press, 2011.

uses Fortran throughout and is a very good coverage of what is possible in modern Fortran. Well worth a read.

The second edition of the following book

- Meyer Bertrand, Object Oriented Software Construction, Prentice Hall, 1997.

provides a very good coverage and uses Eiffel throughout - he did design the language!