Object-Oriented Programming in Fortran 2003 Part 1: Code Reusability

Original articles by Mark Leair, PGI Compiler Engineer

This is Part 1 of a series of articles: * Part 1: Code Reusability * Part 2: Data Polymorphism

1. Introduction

Polymorphism is a term used in software development to describe a variety of techniques employed by programmers to create flexible and reusable software components. The term is Greek and it loosely translates to "many forms".

In programming languages, a polymorphic object is an entity, such as a variable or a procedure, that can hold or operate on values of differing types during the program's execution. Because a polymorphic object can operate on a variety of values and types, it can also be used in a variety of programs, sometimes with little or no change by the programmer. The idea of write once, run many, also known as code reusability, is an important characteristic to the programming paradigm known as Object-Oriented Programming (OOP).

OOP describes an approach to programming where a program is viewed as a collection of interacting, but mostly independent software components. These software components are known as objects in OOP and they are typically implemented in a programming language as an entity that encapsulates both data and procedures.

2. Objects in Fortran 90/95/2003

A Fortran 90/95 module can be viewed as an object because it can encapsulate both data and procedures. Fortran 2003 (F2003) added the ability for a derived type to encapsulate procedures in addition to data. So, by definition, a derived type can now be viewed as an object as well in F2003.

F2003 also introduced type extension to its derived types. This feature allows F2003 programmers to take advantage of one of the more powerful OOP features known as inheritance. Inheritance allows code reusability through an implied inheritance link in which leaf objects, known as children, reuse components from their parent and ancestor objects. For example,

```
type shape
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
end type shape

type, extends(shape) :: rectangle
    integer :: length
    integer :: width
```

```
end type rectangle

type, extends(rectangle) :: square
end type square
```

In the example above, we have a `square` type that inherits components from `rectangle` which inherits components from `shape`. The programmer indicates the inheritance relationship with the **EXTENDS** keyword followed by the name of the parent type in parentheses. A type that **EXTENDS** another type is known as a type extension (e.g., `rectangle` is a type extension of `shape`, `square` is a type extension of `rectangle` and `shape`). A type without any **EXTENDS** keyword is known as a base type (e.g., `shape` is a base type).

A type extension inherits all of the components of its parent (and ancestor) types. A type extension can also define additional components as well. For example, `rectangle` has a `length` and `width` component in addition to the `color`, `filled`, `x`, and `y` components that were inherited from `shape`. The `square` type, on the other hand, inherits all of the components from `rectangle` and `shape`, but does not define any components specific to `square` objects. Below is an example on how we may access the color component of `square`:

```
type(square) :: sq           ! declare sq as a square object

sq%color                     ! access color component for sq
sq%rectangle%color           ! access color component for sq
sq%reactangle%shape%color    ! access color component for sq
```

Note the three different ways for accessing the color component for `sq`. A type extension includes an implicit component with the same name and type as its parent type. This can come in handy when the programmer wants to operate on components specific to a parent type. It also helps illustrate an important relationship between the child and parent types.

We often say the child and parent types have a "is a" relationship. Using our `shape` example above, we can say "a `square` is a `rectangle`", "a `rectangle` is a `shape`", "a `square` is a `shape`", and "a `shape` is a base type". We can also apply this relationship to the type itself (e.g., "a `shape` is a `shape`", "a `rectangle` is a `rectangle`", and "a `square` is a `square`").

Note that the "is a" relationship does not imply the converse. A `rectangle` is a `shape`, but a `shape` is not a `rectangle` since there are components found in `rectangle` that are not found in `shape`. Furthermore, a `rectangle` is not a `square` because `square` has a component not found in `rectangle`; the implicit `rectangle` parent component.

3. Polymorphism in Fortran 2003

The "is a" relationship also helps us visualize how polymorphic variables interact with type extensions. The **CLASS** keyword allows F2003 programmers to create polymorphic variables. A polymorphic variable is a variable whose data type is dynamic at runtime. It must be a pointer

variable, allocatable variable, or a dummy argument. Below is an example:

```
class(shape), pointer :: sh
```

In the example above, the `sh` object can be a pointer to a `shape` or any of its type extensions. So, it can be a pointer to a `shape`, a `rectangle`, a `square`, or any future type extension of `shape`. As long as the type of the pointer target "is a" `shape`, `sh` can point to it.

There are two basic types of polymorphism: procedure polymorphism and data polymorphism. Procedure polymorphism deals with procedures that can operate on a variety of data types and values. Data polymorphism, a topic for part 2 of this article, deals with program variables that can store and operate on a variety of data types and values.

4. Procedure Polymorphism

Procedure polymorphism occurs when a procedure, such as a function or a subroutine, can take a variety of data types as arguments. This is accomplished in F2003 when a procedure has one or more dummy arguments declared with the **CLASS** keyword. For example,

```
subroutine setColor(sh, color)
    class(shape) :: sh
    integer :: color
    sh%color = color
end subroutine setColor
```

The `setColor` subroutine takes two arguments, `sh` and `color`. The `sh` dummy argument is polymorphic, based on the usage of `class(shape)`. The subroutine can operate on objects that satisfy the "is a" shape relationship. So, `setColor` can be called with a `shape`, `rectangle`, `square`, or any future type extension of shape. However, by default, only those components found in the declared type of an object are accessible. For example, `shape` is the declared type of `sh`. Therefore, you can only access the `shape` components, by default, for `sh` in `setColor` (i.e., `sh%color`, `sh%filled`, `sh%x`, `sh%y`). If the programmer needs to access the components of the dynamic type of an object (e.g., `sh%length` when `sh` is a `rectangle`), then they can use the F2003 **SELECT TYPE** construct. The following example illustrates how a **SELECT TYPE** construct can access the components of a dynamic type of an object:

```
subroutine initialize(sh, color, filled, x, y, length, width)
    ! initialize shape objects
    class(shape) :: sh
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
    integer, optional :: length
    integer, optional :: width
```

```
    sh%color = color
    sh%filled = filled
    sh%x = x
    sh%y = y

    select type (sh)
    type is (shape)
        ! no further initialization required
    class is (rectangle)
        ! rectangle or square specific initializations
        if (present(length))  then
           sh%length = length
        else
           sh%length = 0
        endif
        if (present(width)) then
            sh%width = width
        else
            sh%width = 0
        endif
    class default
      ! give error for unexpected/unsupported type
         stop 'initialize: unexpected type for sh object!'
    end select

end subroutine initialize
```

The above example illustrates an initialization procedure for our `shape` example. It takes one shape argument, `sh`, and a set of initial values for the components of `sh`. Two optional arguments, `length` and `width`, are specified when we want to initialize a rectangle or a square object. The **SELECT TYPE** construct allows us to perform a type check on an object. There are two styles of type checks that we can perform. The first type check is called **type is**. This type test is satisfied if the dynamic type of the object is the same as the type specified in parentheses following the **type is** keyword. The second type check is called **class is**. This type test is satisfied if the dynamic type of the object is the same or an extension of the specified type in parentheses following the **class is** keyword.

Returning to our example, we will initialize the length and width fields if the type of `sh` is `rectangle` or `square`. If the dynamic type of `sh` is not a `shape`, `rectangle`, or `square`, then we will execute the **class default** branch. This branch may get executed if we extended the `shape` type without updating the `initialize` subroutine. Because we added a **class default** branch, we also added the **type is (shape)** branch, even though it does not perform any additional assignments. Otherwise, we would incorrectly print our error message when `sh` is of type `shape`.

5. Procedure Polymorphism with Type-Bound Procedures

Section 2, "Objects in Fortran 90/95/2003", mentioned that derived types in F2003 are considered objects because they now can encapsulate data as well as procedures. Procedures encapsulated in a derived type are called type-bound procedures. Below illustrates how we may add a type-bound procedure to shape:

```
type shape
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
contains
    procedure :: initialize
end type shape
```

F2003 added a contains keyword to its derived types to separate a type's data definitions from its procedures. Anything that appears after the contains keyword in a derived type must be a type-bound procedure declaration. Below is the syntax of the type-bound procedure declaration:

```
PROCEDURE [(interface-name)] [[,binding-attr-list ]::] binding-name[=>
procedure-name]
```

Anything in brackets is considered optional in the type-bound procedure syntax above. At the minimum, a type-bound procedure is declared with the *PROCEDURE* keyword followed with a `binding-name`. The `binding-name` is the name of the type-bound procedure.

The first option is called `interface-name`. This option is a topic of discussion in part 2 of this article.

The `binding-attr-list` option is a list of `binding-attributes`. The `binding-attributes` that we will discuss in this article include **PASS**, **NOPASS**, **NON_OVERRIDABLE**, **PUBLIC**, and **PRIVATE**. There is one other `binding-attribute`, called **DEFERRED**, that is a topic of discussion in part 2 of this article.

The procedure-name option is the name of the underlying procedure that implements the type-bound procedure. This option is required if the name of the underlying procedure differs from the binding-name. The procedure-name can be either a module procedure or an external procedure with an explicit interface.

In our example above, we have a `binding-name` called `initialize`. Because `procedure-name` was not specified, an implicit procedure-name, called `initialize` is also declared. Another way to write our example above is `procedure :: initialize => initialize`.

Below is an example of a type-bound procedure that uses a module procedure:

```
module shape_mod

type shape
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
contains
    procedure :: initialize
```

```
end type shape

type, extends(shape) :: rectangle
        integer :: length
        integer :: width
end type rectangle

type, extends(rectangle) :: square
end type square

contains

subroutine initialize(sh, color, filled, x, y, length, width)
    ! initialize shape objects
    class(shape) :: sh
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
    integer, optional :: length
    integer, optional :: width

    sh%color = color
    sh%filled = filled
    sh%x = x
    sh%y = y

    select type (sh)
    type is (shape)
          ! no further initialization required
    class is (rectangle)
        ! rectangle or square specific initializations
        if (present(length))  then
           sh%length = length
        else
           sh%length = 0
        endif
        if (present(width)) then
            sh%width = width
        else
            sh%width = 0
        endif
    class default
      ! give error for unexpected/unsupported type
         stop 'initialize: unexpected type for sh object!'
    end select

end subroutine initialize

end module
```

Below is an example of a type-bound procedure that uses an external procedure with an explicit interface:

```
module shape_mod
```

```
type shape
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
contains
    procedure :: initialize
end type shape

type, extends(shape) :: rectangle
    integer :: length
    integer :: width
end type rectangle

type, extends(rectangle) :: square
end type square

interface
    subroutine initialize(sh, color, filled, x, y, length, width)
        import shape
        class(shape) :: sh
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
        integer, optional :: length
        integer, optional :: width
  end subroutine
end interface

end module
```

Using the examples above, we can invoke the type-bound procedure in the following manner:

```
use shape_mod
type(shape) :: shp                      ! declare an instance of shape
call shp%initialize(1, .true., 10, 20)   ! initialize shape
```

The syntax for invoking a type-bound procedure is very similar to accessing a data component in a derived type. The name of the component is preceded by the variable name separated by a percent (%) sign. In this case, the name of the component is initialize and the name of the variable is `shp`. So, we type `shp%initialize` to access the initialize type-bound procedure. The above example calls the `initialize` subroutine and passes in `1` for *color*, `.true.` for *filled*, `10` for *x*, and `20` for *y*.

You may notice that we have not yet mentioned anything about the first dummy argument, `sh`, in `initialize`. This dummy argument is known as the passed-object dummy argument. By default, the passed-object dummy is the first dummy argument in the type-bound procedure. It receives the object that invoked the type-bound procedure. In our example, `sh` is the passed-object dummy and the invoking object is `shp`. Therefore, the `shp` object gets assigned to `sh` when initialize is invoked.

The passed-object dummy argument must be declared **CLASS** and of the same type as the derived type that defined the type-bound procedure. For example, a type bound procedure declared in `shape` must have a passed-object dummy argument declared `class(shape)`.

We can also specify a different passed-object dummy argument using the **PASS** binding-attribute. For example, let's say that the `sh` dummy in our initialize subroutine did not appear as the first argument. Then we would need to specify a *PASS* attribute like in the following code:

```
type shape
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
contains
    procedure, pass(sh) :: initialize
end type shape
```

Sometimes we do not want to specify a passed-object dummy argument. We can choose to not specify one using the **NOPASS** binding-attribute:

```
type shape
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
contains
    procedure, nopass :: initialize
end type shape
```

If we specify **NOPASS** in our example, then we still invoke the type-bound procedure the same way. The only difference is that the invoking object is not automatically assigned to a passed-object dummy in the type-bound procedure. Therefore, if we were to specify **NOPASS** in our initialize type-bound procedure, we would invoke initialize in the following manner:

```
type(shape) :: shp                          ! declare an instance of shape
call shp%initialize(shp, 1, .true., 10, 20)   ! initialize shape
Note that we explicitly specify shp for the first argument of initialize because it
was declared NOPASS.
```

6. Inheritance and Type-Bound Procedures

Recall from section 2, "Objects in Fortran 90/95/2003", that a child type inherits or reuses components from their parent or ancestor types. This applies to both data and procedures when dealing with F2003 derived types. In the code below, rectangle and square will both inherit the initialize type-bound procedure from shape.

```
type shape
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
contains
    procedure :: initialize
end type shape

type, extends(shape) :: rectangle
    integer :: length
    integer :: width
end type rectangle

type, extends(rectangle) :: square
end type square
```

Using the example above, we can invoke initialize with a shape, rectangle, or square object:

```
type(shape) :: shp                                  ! declare an instance of shape
type(rectangle) :: rect                             ! declare an instance of
rectangle
type(square) :: sq                                  ! declare an instance of square
call shp%initialize(1, .true., 10, 20)              ! initialize shape
call rect%initialize(2, .false., 100, 200, 50, 25)  ! initialize rectangle
call sq%initialize(3, .false., 400, 500, 30, 20)    ! initialize rectangle
```

7. Procedure Overriding

Most OOP languages allow a child object to override a procedure inherited from its parent object. This is known as procedure overriding. In F2003, we can specify a type-bound procedure in a child type that has the same binding-name as a type-bound procedure in the parent type. When the child overrides a particular type-bound procedure, the version defined in its derived type will get invoked instead of the version defined in the parent. Below is an example where `rectangle` defines an `initialize` type-bound procedure that overrides `shape`'s initialize type-bound procedure:

```
module shape_mod
type shape
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
contains
    procedure :: initialize => initShape
end type shape

type, extends(shape) :: rectangle
    integer :: length
    integer :: width
contains
```

```
    procedure :: initialize => initRectangle
end type rectangle

type, extends(rectangle) :: square
end type square

contains

    subroutine initShape(this, color, filled, x, y, length, width)
        ! initialize shape objects
        class(shape) :: this
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
        integer, optional :: length  ! ingnored for shape
        integer, optional :: width   ! ignored for shape

        this%color = color
        this%filled = filled
        this%x = x
        this%y = y
    end subroutine initShape

    subroutine initRectangle(this, color, filled, x, y, length, width)
        ! initialize rectangle objects
        class(rectangle) :: this
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
        integer, optional :: length
        integer, optional :: width

        this%color = color
        this%filled = filled
        this%x = x
        this%y = y

        if (present(length)) then
           this%length = length
        else
           this%length = 0
        endif
        if (present(width)) then
            this%width = width
        else
             this%width = 0
        endif

    end subroutine initRectangle

end module
```

In the sample code above, we defined a type-bound procedure called `initialize` for both `shape` and `rectangle`. The only difference is that `shape`'s version of initialize will invoke a procedure called

`initShape` and `rectangle`'s version will invoke a procedure called `initRectangle`. Note that the passed-object dummy in `initShape` is declared `class(shape)` and the passed-object dummy in `initRectangle` is declared `class(rectangle)`. Recall that a type-bound procedure's passed-object dummy must match the type of the derived type that defined it. Other than differing passed-object dummy arguments, the interface for the child's overriding type-bound procedure is identical with the interface for the parent's type-bound procedure. That is because both type-bound procedures are invoked in the same manner:

```
type(shape) :: shp                                  ! declare an instance of shape
type(rectangle) :: rect                             ! declare an instance of
rectangle
type(square) :: sq                                  ! declare an instance of square
call shp%initialize(1, .true., 10, 20)              ! calls initShape
call rect%initialize(2, .false., 100, 200, 11, 22)  ! calls initRectangle
call sq%initialize(3, .false., 400, 500)            ! calls initRectangle
```

> Note that `sq` is declared `square` but its initialize type-bound procedure invokes `initRectangle` because `sq` inherits the `rectangle` version of `initialize`.

Although a type may override a type-bound procedure, it is still possible to invoke the version defined by a parent type. Recall in section 2, "Objects in Fortran 90/95/2003", that each type extension contains an implicit parent object of the same name and type as the parent. We can use this implicit parent object to access components specific to a parent, say, a parent's version of a type-bound procedure:

```
call rect%shape%initialize(2, .false., 100, 200)         ! calls initShape
call sq%rectangle%shape%initialize(3, .false., 400, 500) ! calls initShape
```

Sometimes we may not want a child to override a parent's type-bound procedure. We can use the **NON_OVERRIDABLE** binding-attribute to prevent any type extensions from overriding a particular type-bound procedure:

```
type shape
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
contains
    procedure, non_overridable :: initialize
end type shape
```

8. Functions as Type-Bound Procedures

Up to this point, subroutines have been used to implement type-bound procedures. We can also

implement type-bound procedures with functions as well. Below is an example with a function that queries the status of the filled component in `shape`.

```
module shape_mod

type shape
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
contains
    procedure :: isFilled
end type shape

contains

    logical function isFilled(this)
        class(shape) :: this
        isFilled = this%filled
    end function isFilled

end module
```

We can invoke the above function in the following manner:

```
use shape_mod
type(shape) :: shp        ! declare an instance of shape
logical filled
call shp%initialize(1, .true., 10, 20)
filled = shp%isFilled()
```

9. Information Hiding

In section 7, "Procedure Overriding", we showed how a child type can override a parent's type-bound procedure. This allows a user of our type to invoke, say, the `initialize` type-bound procedure, without any knowledge of the implementation details of initialize. This is an example of information hiding, another important feature of OOP.

Information hiding allows the programmer to view an object and its procedures as a "black box". That is, the programmer can use (or reuse) an object without any knowledge of the implementation details of the object.

Inquiry functions, like the `isFilled` function in section 8, "Functions as Type-Bound Procedures", are common with information hiding. The motivation for inquiry functions, rather than direct access to the underlying data, is that the object's implementer can change the underlying data without affecting the programs that use the object. Otherwise, each program that uses the object would need to be updated whenever the underlying data of the object changes.

To enable information hiding, F2003 provides a **PRIVATE** keyword (and binding-attribute). F2003 also provides a **PUBLIC** keyword (and binding-attribute) to disable information hiding. By default, all derived type components are declared **PUBLIC**. The **PRIVATE** keyword can be placed on derived type data and type-bound procedure components (and on module data and procedures). We illustrate **PUBLIC** and **PRIVATE** in the sample code below:

```
module shape_mod

private    ! hide the type-bound procedure implementation procedures
public :: shape, constructor   ! allow access to shape & constructor procedure

type shape
    private               ! hide the underlying details
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
contains
    private                          ! hide the type bound procedures by default
    procedure         :: initShape  ! private type-bound procedure
    procedure, public :: isFilled   ! allow access to isFilled type-bound procedure
    procedure, public :: print      ! allow access to print type-bound procedure
end type shape

contains

    logical function isFilled(this)
        class(shape) :: this
        isFilled = this%filled
    end function isFilled

    function constructor(color, filled, x, y)
        type(shape) :: constructor
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
        call constructor%initShape(color, filled, x, y)
    end function constructor

    subroutine initShape(this, color, filled, x, y)
        ! initialize shape objects
        class(shape) :: this
        integer :: color
        logical :: filled
        integer :: x
        integer :: y

        this%color = color
        this%filled = filled
        this%x = x
        this%y = y
    end subroutine initShape

    subroutine print(this)
```

```
        class(shape) :: this
        print *, this%color, this%filled, this%x, this%y
    end subroutine  print

end module
```

The example above uses information hiding in the host module as well as in the shape type. The private statement, located at the top of the module, enables information hiding on all module data and procedures. The `isFilled` module procedure (not to be confused with the `isFilled` type-bound procedure) is hidden as a result of the private statement at the top of the module. We added `public :: constructor` to allow the user to invoke the constructor module procedure. We added a **private** statement on the data components of `shape`. Now, the only way a user can query the filled component is through the `isFilled` type-bound procedure, which is declared public.

Note the **private** statement after the **contains** in type `shape`. The **private** that appears after type `shape` only affects the data components of `shape`. If you want your type-bound procedures to also be private, then a **private** statement must also be added after the **contains** keyword. Otherwise, type-bound procedures are **public** by default.

In our example, the `initShape` type-bound procedure is declared **private**. Therefore, only procedures local to the host module can invoke a **private** type-bound procedure. In our example above, the `constructor` module procedure invokes the `initShape` type-bound procedure. Below is how we may invoke our example from above:

```
program shape_prg
    use shape_mod
    type(shape) :: sh
    logical filled
    sh = constructor(5, .true., 100, 200)
    call sh%print()
end program shape_prg
```

Below is a sample compile and sample run of the above program (we assume that the shape_mod module is saved in a file called shape.f03 and that the main program is called main.f03):

```
% pgfortran -V ; pgfortran shape.f03 main.f03 -o shapeTest
pgfortran 11.2-1 64-bit target on x86-64 Linux -tp penryn
Copyright 1989-2000, The Portland Group, Inc.  All Rights Reserved.
Copyright 2000-2011, STMicroelectronics, Inc.  All Rights Reserved.
shape.f03:
main.f03:
% shapeTest
           5  T          100          200
```

10. Type Overloading

In our previous example, we created an instance of `shape` by invoking a function called `constructor`.

This allows us to hide the details for constructing a `shape` object, including the underlying type-bound procedure that performs the initialization. However, you may have noticed that the word `constructor` could very well be defined somewhere else in the host program. If that is the case, the program cannot use our module without renaming one of the constructor functions. But since OOP encourages information hiding and code reusability, it would make more sense to come up with a name that probably is not being defined in the host program. That name is the type name of the object we are constructing.

F2003 allows the programmer to overload a name of a derived type with a generic interface. The generic interface acts as a wrapper for our constructor function. The idea is that the user would then construct a shape in the following manner:

```
program shape_prg
    use shape_mod
    type(shape) :: sh
    logical filled
    sh = shape(5, .true., 100, 200)  ! invoke constructor through shape generic
interface
    call sh%print()
end program shape_prg
```

Below is the modified version of our example from section 9, "Information Hiding", that uses type overloading:

```
module shape_mod

private    ! hide the type-bound procedure implementation procedures
public :: shape ! allow access to shape

type shape
    private               ! hide the underlying details
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
contains
    private                          ! hide the type bound procedures by default
    procedure         :: initShape  ! private type-bound procedure
    procedure, public :: isFilled   ! allow access to isFilled type-bound procedure
end type shape

interface shape
    procedure constructor       ! add constructor to shape generic interface
end interface

contains
     :
     :
end module shape_mod
```

Our constructor function is now declared private and it is invoked through the `shape` public generic interface.

11. Conclusion

Code reusability, an important feature of Object-Oriented Programming (OOP), is enabled through inheritance, polymorphism, and information hiding. With inheritance, an object can be extended and code from the parent object can be reused or overloaded in the child object. Code reusability is also enabled through polymorphism. There are two types of polymorphism: procedure polymorphism and data polymorphism. Procedure polymorphism enables code reusability because a procedure can operate on a variety of data types and values. The programmer does not have to reinvent the wheel for every data type a program may encounter with a polymorphic procedure. Part 2 of this article will cover data polymorphism. Finally, we examined information hiding which allows programmers to use an object without having to understand its underlying implementation details. Fortran 2003 (F2003) supports inheritance, polymorphism, and information hiding through type extension, the **CLASS** keyword, and the **PUBLIC/PRIVATE** keywords/binding-attributes respectively.

In the next installment of this article we will continue our discussion of OOP with F2003 by examining the other form of polymorphism, data polymorphism, and see how it can be used to create flexible and reusable software components. We will also examine the following OOP features offered by F2003: unlimited polymorphic objects, typed allocation, sourced allocation, generic type-bound procedures, deferred bindings, and abstract types.