Practical 04

By the end of this set of problems, you will be familiar with:

- Modules
- Derived types
- Interfaces

All `programs` and `modules` *must* contain `implicit none`. While not strictly necessary, it will eliminate an entire class of bugs.

It is a very good idea to always use the following compiler flags for all these problems: `-Wall -Wextra -fcheck=all -g`. Try to ensure you have no warnings. You may also want to use `-std=f2008` or `-std=f2018`, depending on the version of `gfortran` you are using. This will ensure you stick to standard Fortran.

You should use the lecture materials for help/inspiration, but please don't copy and paste! There is some value to be had in typing up the programs yourself.

There are very likely too many exercises here for the time you have available. It's ok if you don't get through them all!

There may be several ways to solve each problem. If you have time, you might like to try different approaches.

Use a separate file and program for each exercise.

Rational Numbers

1. Write a module that contains a type for rational numbers.
2. Implement functions that take two rational numbers and return the result of the basic arithmetic operations (addition, subtraction, multiplication and division).
3. Implement a function that returns true if two rational numbers are equal. Hint: you may want to implement a function that reduces a rational number.
4. Write a program that uses this module and checks that your functions give the correct answers.

Further

1. Write a `logical function` that takes three rational numbers and a `function` f and returns true if the result of applying f to the first two numbers is equal to the third number. Use this to implement your tests
2. We saw very briefly how to overload an operator, such as `+`, via an `interface` block. Use this syntax to overload `+`, `-`, `*`, `/`, and `==`.
3. Another way of overloading these operators is with the following syntax:

   ```
   type :: my_type
   contains
     procedure :: my_add
     generic :: operator(+) => my_add
   end type my_type
   ```

   Use this instead of an `interface` block.
4. Add `public`/`private` specifiers to your module and type so that only the type itself and the operators are visible. You may need to add "shims" (that just do the arithmetic operations) to your program so that your tests still work.

Integrate a function

Consider the following ODE:

$$f(y,t) = \frac{dy}{dt} = \frac{1}{t^2} - \frac{y}{t} - y^2$$
$$y(1) = -1$$
$$1 \leq t \leq 2$$

The exact solution is

$$y = -\frac{1}{t}$$

1. Take your modified Euler integration program from the previous exercises and move the relevant functions into a `module`.
2. Modify the integration function it so that you can pass in another function that takes the current values of y and t and returns the $\frac{dy}{dt}$.
3. Write a function that implements the ODE above ($f(y,t)$) and solve it using your integration routine. Note that you now need to be able to specify the initial time!
4. Compare the absolute error between the Euler integration and the exact solution as you vary the number of timesteps.

Further

1. Implement the second-order Runge-Kutta scheme:

$$k_1 = f(y^n, t)$$
$$k_2 = f(y^n + k_1 \Delta t, t + \Delta t)$$
$$y^{n+1} = y^n + \tfrac{1}{2}\Delta t(k_1 + k_2)$$

You may wish to do this in a separate `module` to the Euler scheme.