Maarten Pennings
maartenpennings@hotmail.com

Maarten Pennings

Promoting types to first-class citizens

The Delphi approach

1996 November 24, updated 2001 April 12, to pdf 2006 December 20

Promoting types to first-class citizens

The Delphi approach

1. Introduction

An important characteristic of programming languages is *orthogonality*. We say that a language is orthogonal if related concepts behave similarly. Pascal is an example of a fairly orthogonal language. For *any* type `T`, we can make a new type by applying the type constructor **array of** to it to form an array of `T`. Other orthogonal aspects of Pascal are that a parameter of a procedure or function can have *any* type and that the statement `a:=b` is legal for *any* two variables `a` and `b` having the same type. An example where Pascal is not orthogonal is the return type of a function: it can only be a built-in type like integer; not a self-made array. Likewise, the boolean expression `a=b` (where `a` and `b` share the type `T`) is not allowed for every type `T`.

Borlands Turbo Pascal 5.0 made Pascal even more orthogonal in one aspect: procedures were promoted to first-class citizens. What this means is that the type space was extended to procedures: variables of an appropriate type could be assigned a procedure! And, Pascal being an orthogonal language, procedural types can also be used as the type of parameters of procedures and functions; variables of the same procedural types can be assigned to each other and even tested for equality.

Borlands Delphi extends Pascals orthogonality even further: types themselves are now promoted to first-class citizens: the type space is extended with types! A variable with an appropriate type can now be assigned a type.

This paper explains the concept of types of types. First we skim types in general; then, we will refresh our knowledge on procedural types; and finally, we will delve into type types.

2. Types

Types can be considered as sets. The type `boolean` denotes the set {`true`, `false`} and the type `real` denotes the set of all real numbers (well, not all real numbers but a large subset thereof). For any type `T`, we can introduce a variable `x` by including the following declaration.

```
var x:T
```

This means that variable `x` can hold any member of the set denoted by `T`. We assign `x` such a member with an assignment where the right hand value `E` has type `T` too.

```
x:=E
```

Such an expression `E` of type `T` could be a compound expression with a top operator yielding a value of type `T` (e.g. with two integers `i` and `j` we can make a boolean expression by using the operator smaller-than: `i<j`). With a bit of luck, there is a constant `c` of type `T`. When `T` is one of the built-in types such as `integer`, `real` or `char`, constants are also built-in: `765`, `12.56E-3` respectively `'K'`. For user defined types, constants are defined using special syntax, as will we see later on.

Pascal allows one to define new types (compound types) from the built-in types, using so called type-constructors. The standard type constructors are **`record of`**, **`array of`**, **`set of`**, pointer to (for some strange reason denoted by `^`), and `file of`. Furthermore, for *some* of the built-in types there are type-constructors to make enumerations and subranges (indeed, this is not orthogonal).

Type constructors usually have well-known set-operations associated to them. For example, let S_1 and S_2 be sets denoted by the types `T1` and `T2`. The set associated with the type **`record`** `x1:T1; x2:T2` **`end`** is the Cartesian product $S_1 \times S_2$.

For compound types, Turbo Pascal allows you to define constants (members of the underlying set) too. In the following example, the constant `origin` is defined, so that `p` can be assigned a value.

```
type  point  = record x,y:real end;
const origin : point = (x:0.0 ; y:0.0);
var   p,q    : point;
begin
  p:=origin
end
```

An expression like `p=q` would not be legal in Turbo Pascal since equality test is not allowed on records (one could imagine that the compiler would simply use a block compare, however, variant records and world-alignment for record fields invalidates this approach).

3. Procedural types

In this section, we will see how functions and procedures are made first-class citizens. The purpose of this section is to get acquainted with the concepts needed to introduce first-class citizens so that the next section, on type types, is easier understood.

Turbo Pascal 5.0 introduced a new type-constructor to promote functions and procedures to first-class citizens. The reserved words **`function`** and **`procedure`** were overloaded to also be a type-constructor. The following are legal type declarations. By the way, note that we adopted the Delphi naming convention for types, that is, a `T` prefixes type names.

```
type
  TProc           = procedure;
  TIntInt2IntFunc = function (x,y:integer):integer;
```

As illustrated, the type declarations resemble a procedure / function header. Note however, that the function name is missing, and that the parameter *names* are purely decorative. The parameter types (and the return type), however, are important. They enable strong typing.

What is the set denoted by `TProc` (or `TIntInt2IntFunc` for that matter)? The set `TProc` contains *all procedures that have no parameters*. That is, `Exit`, `ClrScr`, and `Randomize` belong to `TProc`. However, the following useless procedure (and many, many more) also belongs to this set.[1]

```
procedure StrangeOne;
begin
  writeln('Foo');
  reset(input);
  x:=5 { assuming that x is a global integer }
end
```

The following functions are members of the set denoted by `TIntInt2IntFunc`.

```
function max(a,b:integer):integer;
begin
  if a>b then max:=a else max:=b
end;

function min(a,b:integer):integer;
begin
  if a<b then max:=a else max:=b
end;
```

Given procedural types, variables of such a type might be introduced: procedure and functions become first-class citizens!

```
var f,g:TIntInt2IntFunc
```

[1] Unfortunately, procedural types are not very orthogonal in many senses. The least restrictive requirement is that **`inline`**, **`interrupt`** or **`near`** procedures are never members of procedural types. However, *standard* and *nested* procedures are also excluded (which means that, in contrast to the statement in the main text, `Exit`, `ClrScr` and `Randomize` are not a member of `TProc`)!

Given this declaration, the following code is legal Turbo Pascal. It illustrates functional constants, functional assignment, applying functional variables and the functional equality test[2].

```
begin
  f:=max;                { functional constant          }
  g:=f;                  { plain assignment             }
  i:=f(3,5);             { applying functional variables }
  if g=min then {…}      { equality test                }
end
```

Note that there is no need to introduce special syntax for functional constants: user defined functions (`max` and `min`) can be used for that. Note also that, due to strong typing, the compiler "knows" that it makes sense to pass the two integers `3` and `5` to the function `f` (third example). The type `TIntInt2IntFunc` captures the common property of all members of the associated set, namely that it is a function with two input integers and an output integer.

Except for applying, assignment and equality test, Turbo Pascal offers no other operators on expressions of a procedural type. Of course, one might envision a concat operator (denoted by +) on `TProc`; the expression `StrangeOne+ClrScr` would then be allowed. However, this would obscure the language and the compiler too much (and is pretty hard to implement).

Procedural types are implemented as pointers; namely start addresses of functions and procedures. In other words, in the above example, `sizeof(f)=sizeof(pointer)`. The value of `f` is a start address. The compiler enforces that `f` only contains addresses of functions with two integer input parameters and an integer return type.

4. Type types

The core of the paper will now be presented: type types.

In analogy with a procedural type, where the elements of the associated set are procedures with a common property, the elements of a type type are types with a common property. A variable of a type type can be assigned a type. Using a variable of a type type boils down to dynamic creation of a variable of the stored type.

In this section, we will see how Delphi deals with these issues.

4.1 Example part I, or the introduction

The theory of type types will be enlightened with an example.

We will write an application dealing with fractions. Nothing complicated. A simple form will be displayed (see Figure 1). The left side shows a fraction, the right side a button. By pressing the button, the fraction is divided by two (the resulting form is displayed in Figure 2). Pressing the "Destroy Fraction" button closes the form.

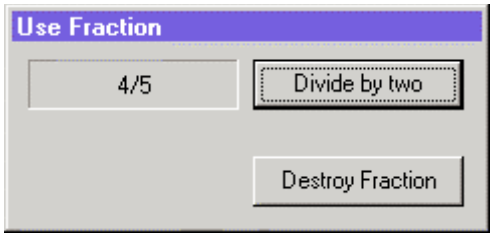

Figure 1 The application

Figure 2 After pressing "Divide by two"

What is so special? Before the "Use Fraction" form is displayed, the user must choose what kind of fraction he wants to use. In our example, there are two choices. A fraction can be implemented as two integers (numerator and denominator) or as a real. When the real implementation is chosen, the above session would look slightly differently; see Figure 3 and Figure 4.

[2] Equality test is quite tricky for functional variables. If `f` and `g` are functions of type `function:integer`, the expression `f=g` is ambiguous: do `f` and `g` denote functions, or the value obtained by calling them? Read the Borland manuals if you are interested.

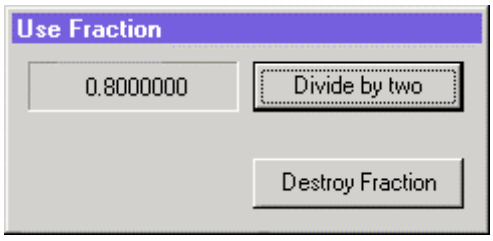

Figure 3 Using reals

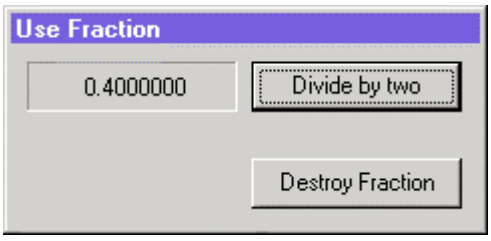

Figure 4 After pressing "Divide by two"

Figure 5 shows the main form at a point where the user has decided to start with a fraction of 4/5 and use the integer representation. By pressing “Create Fraction”, the window in Figure 1 will pop up. If, on the other hand, the user had selected the real representation (Figure 6) pressing “Create Fraction" would lead to Figure 3.

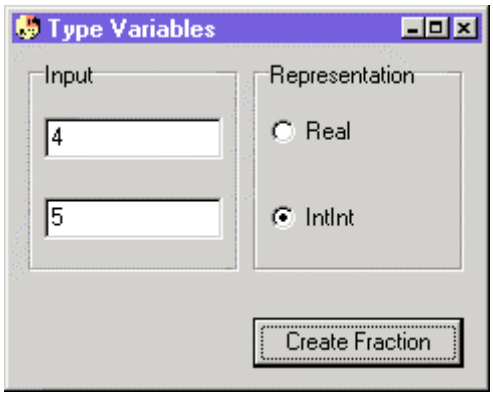

Figure 5 The main form

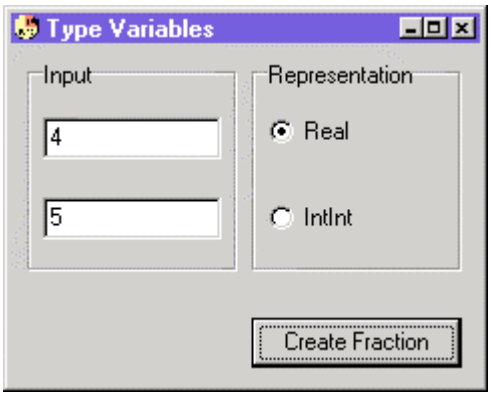

Figure 6 Using the real implementation

4.2 Syntax, or how to declare a type type

In analogy with procedural types, one could have expected that type types would overload the reserved word **`type`** as a type-constructor for type types. Syntax would then have read something like the following fragment.

```
type TType = type …
```

At the place of the ellipsis (…) we expect some syntax to further restrict the associated set.

Why?

Like with procedural types, we do not want that a single type **`procedure`** exists that denotes a set with every possible procedure, from `Randomize` and `Close` to `Line` and `InstallUserFont`. Rather, we want sets of procedures that have a common property like having a single integer parameter; a procedural type denoted by **`procedure`** `(x:int)`. In this way, we have strong typing, and it makes sense to call `p(3)`, if `p` has the just mentioned procedural type.

Similarly, we do not want a single type **`type`** that denotes the set of all types, from `boolean` and `real` to `array [1..10] of DateTime` and `TButton`. Rather we want sets of types that have a common property like that they have an addition operator (+). Such a type could have been denoted by the following definition.

```
type TAdditiveType = type with +
```

Since `integer`, `word`, `real` and `string` have an addition (well, a + operator) associated with them, we would expect these types to be a member of `TAdditiveType`.

This is indeed the right way to *think* about it, it is *not* the way that Delphi implements it.

The problem is that it is not very clear which operators are associated with which types. Especially not if we would want (and we would want that) to include user defined operators. For example, suppose I define a type `TBag` and an operator **`function`** `Add(a,b:TBag):TBag`. Does `TBag` now belong to the set associated with `TAdditiveType`? Ideally, yes. But how is the compiler to know that?

The solution to this problem is simple and straightforward. What is Delphi's way to associate operators with types? Classes!

For example, suppose we have the following type declaration.

```
type someclass = class
  private
    { … some attributes … }
  public
    procedure Inc
end
```

Then Delphi allows us to define the following type type. Note that the syntax for a type type uses the reserved words **`class of`**.

```
type TIncrementalType = class of someclass
```

The set associated with the `TIncrementalType` consists of all *types* derived (in the OO inheritance sense) from `someclass`. All members of this set are types that have an `Inc` operator associated with them!

4.3 Example part II, or setting up the types.

The main function of the "Use Fraction" form is the "Divide by two" button. The handler associated with the click event has the following form.

```
procedure TUseFrac.ButtonDivideByTwoClick(Sender: TObject);
begin
  Fraction.Half;
  PanelDisplay.Caption:=Fraction.Get
end;
```

This handler operates on a global variable `Fraction` that stores the fractional value. This code shows us the common properties we require in a fraction: a `Half` method (that divides the fraction by two) and a `Get` method that returns a string representation of the fraction. Next to these two methods, we need a constructor to initialize the fraction. We will use the Delphi convention, and call the constructor `Create`.

Every implementation of a fraction that has these three methods, will work with our example program. Let us formalize this type. The code below shows a unit that defines a class representing fractions.

```
unit FracAbstract;

interface
  type
    TFracAbstract = class
      public
        constructor Create(a,b:integer); virtual; abstract;
        procedure   Half;                virtual; abstract;
        function    Get:string;          virtual; abstract;
    end;

implementation
  // No implementation: abstract class

end.
```

The class `TFracAbstract` is an *abstract* class. Abstract classes are classes that have at least one abstract method. An abstract method is a method without a body; such a method is flagged with the reserved word **`abstract`**. When a method has no body, it can not be run. When a class has methods that can not be run, one should not create instances of that class (or at least not call any of the abstract methods, or an exception will be raised).

So, what is the use of an abstract class? The reason lies in implementation decoupling (using polymorphism). By defining an abstract fraction we can already write functions like the above `ButtonDivideByTwoClick` without having an implementation of a fraction. What's more, every implementation (subclass) of `TFracAbstract` will work with the function `ButtonDivideByTwoClick`. And that is precisely what we did. The global variable `Fraction` introduced at the beginning of this section has type `TFracAbstract`. Although we do not yet have a concrete descender class, we can already write the event handler that uses `Half` and `Get`.

Given the abstract class `TFracAbstract`, we can now declare a type type `TRepresentation`. The associated set contains all types that are subclasses of `TFracAbstract`. So every type in the set `TRepresentation` supports the operations `Half` and `Get`.Which types are part of `TRepresentation` is unknown at the moment.

```
type TRepresentation = class of TFracAbstract;
var  Representation  : TRepresentation;
```

4.4 Semantics, or how to use type types

We will no discuss how to use type variables (variables with a type type). From the procedural types we know that there are three topics: basic expressions, operators for compound expressions, and the application of type variables. We will discuss these topics using the following example declarations.

```
type
  TP     = class          {…} end;
  TC1    = class(TClassP) {…} end;
  TC2    = class(TClassP) {…} end;
  TPClass = class of TClassP;          // The type type
var
  pclass1,pclass2 : TPClass;
  p1,p2           : TP;
  c               : TC2;
  b               : Boolean
```

Observe that `TP` is a (parent) class with two child-classes: `TC1` and `TC2`. The type `TPClass` is a type type. Its elements are `TP` and its subclasses: `TC1` and `TC2` (in this example). In addition to these types a number of variables has been declared: `pclass1` and `pclass2` are type variables, `p1` and `p2` are objects of (descenders of) `TP`, `c` is an object of (descenders of) `TC2`, and `b` is a simple boolean.

4.4.1 Basic Expressions

The set denoted by `TPClass` contains all descenders of `TP`. In our example, `TPClass` thus has three elements: {`TP`,`TC1`,`TC2`}. This means that the following assignments are legal.

```
pclass1:=TP;
pclass1:=TC1;
pclass1:=TC2
```

Of course, type variables can be assigned to each other.

```
pclass2:=pclass1
```

4.4.2 Applying type variables

It does not make sense to just pass around types in an application. A stored type should be used now and then. So, how can we use a type? The answer is simple: by dynamically creating an instance of it.

One thing that has not yet been discussed so far, is Delphi's object model. Objects in Delphi are always stored on the heap. Thus, although `p1` behaves like an object of `TP`, it actually is a *pointer* to a `TP` object.[3]

But wait a minute, we never call `new` or `malloc` or whatever. When does Delphi allocate heap space? The secret is in constructors. Every class needs to have a constructor. A constructor is like an ordinary method, except that the keyword **`procedure`** is replaced by **`constructor`**. Delphi adds some special code to constructors. This code takes care of allocating a block on the heap of precisely the right size to store the instance. A pointer to the block is returned by the constructor.

Note also that every class you define has a parent class. If you do not explicitly specify a parent class yourself, Delphi uses the predefined class `TObject` as parent. This class has a constructor under the name `Create`. So, even if your class has no constructors, it inherits `Create` from `TObject`.

Let's go back to the example. Variable `p1` is of type `TP`. Before we can access any of the fields of `p1`, we must allocate space on the heap. You typically write `p1:=TP.Create`. If you forget to call an instance's constructor, Delphi raises an exception as soon as you access its fields.

[3] You can see the difference when printing `sizeof(p1)` and `sizeof(TP)`.

Did you observe the type reference in the constructor call: `p1:=TP.Create`? The type `TP` is a constant of the set `TPClass`. Instead of writing a constant, we can write a type expression there! In the following example we create an object of the type stored in variable `pclass1`.

```
p1:=pclass1.Create
```

If `pclass1` equals `TC1`, `p1` is now a `TC1`. If `pclass1` equals `TC2`, `p2` is a `TC2`, and if `pclass1` happens to be `TP`, `p1` is a `TP`. Note that all types that `pclass1` can hold are, by definition, derived classes of `TP`. Precisely those classes are legal instances for `p1`.

When you are done with an object, you should always free its block on the heap. We use the `Free` method any class inherits from `TObject`.

```
p1.Free
```

4.4.3 Operators

Types have been promoted to first-class citizens. We can have variables, like `pclass1` and `pclass2`, that can hold types. But, do we have operators these new type types?

Like with many types, equality test is allowed.

```
b:= pclass1=TP;      // Equality test with constant (parent)
b:= pclass1=TC1;     // Equality test with constant (child)
b:= pclass1=pclass2  // Equality test with other variable
```

However, Delphi provides two new operators, **`is`** and **`as`**. Both operators are binary infix operators. On the right-hand side, they require a type expression (e.g. a constant or type variable) and on the left-hand side, they require an object.

The **`is`** operator returns a boolean value. Iff the type of the object on the left equals the type on the right (or a descender thereof), the returned value is true.

```
b:= p1 is TP;       // Typetest object with a constant (parent)
b:= p1 is TC1;      // Typetest object with a constant (child)
b:= p1 is pclass1   // Typetest object with a type variable
```

The **`as`** operator returns an object of the type specified on the right, by casting the object on the left to that type. The advantage of `o as T` over a plain type cast `T(o)`, is that **`as`** raises an exception if `o is T` does not hold. Assuming that `p1` contains an object of `TC2`, the following **`as`**-operations will not raise an exception.

```
c:= p1 as TC2;      // Safe typecast with constant
c:= p1 as pclass2   // Safe typecast with a type variable
```

Suppose that `TC2` has a method `child` that `TP` has not. We can not write `p1.child`, since `child` is not a member of `TP`. However, when we use a typecast, we can call the method.

```
p1:=TC2.Create;
p1.child;           // Will not compile
(p1 as TC2).child   // Typecast to TC2, then select method
```

The fact that each object knows its type and its inheritance is based on techniques known as RTTI, which stands for *r*un *t*ime *t*ype *i*nformation. I *think* RTTI is implemented by having a record for each class that stores some information about that class. The information probably includes the class's size (needed for `Create`), its parent class (needed for inheritance hierarchy), and some tag (to identify the class). Probably this RTTI record is the same record as the VMT (*v*irtual *m*ethod *t*able) that stores for every class the pointers to the virtual methods. Every object must have a pointer to the RTTI record (like it must have to the VMT).

If RTTI is implemented like this, it is easy to implement type variables. A type variable is nothing else but the tag in the RTTI record, or even simpler, a pointer to the RTTI record.

4.5 Example part III, or defining the derived classes

The figures below show two concrete implementations (derived classes) of the abstract class `TFracAbstract`. Figure 7 shows the unit that implements a fraction as two integers, a numerator and denominator. Figure 8 shows a unit that uses a real instead.

The units belonging to the two forms, the main form and the use fraction form, are given in Figure 9 respectively Figure 10. I hope the annotation suffices to explain the last hairy details.

```
unit FracIntInt;

interface

  uses
    FracAbstract;

  type
    TFracIntInt = class(TFracAbstract)
      public
        constructor Create(a,b:integer); override;
        procedure Half; override;
        function Get:string; override;
      private
        t,n:integer
    end;

implementation

  constructor TFracIntInt.Create(a,b:integer);
  begin
    t:=a; n:=b
  end;

  procedure TFracIntInt.Half;
  begin
    if odd(t) then n:=n*2 else t:=t div 2
  end;

  function TfracIntInt.Get:string;
  var s1,s2:string;
  begin
    str(t,s1); str(n,s2); Result:=s1+'/'+s2
  end;

end.
```

Figure 7 Fractions implemented as two integers

```
unit FracReal;

interface

  uses
    FracAbstract;

  type
    TFracReal = class(TFracAbstract)
      public
        constructor Create(a,b:integer); override;
        procedure Half; override;
        function Get:string; override;
      private
        f:real
    end;

implementation

  constructor TFracReal.Create(a,b:integer);
  begin
    f:=a/b
  end;

  procedure TFracReal.Half;
  begin
    f:=f/2
  end;

  function TFracReal.Get:string;
  var s:string;
  begin
    str(f:10:7,s); Result:=s
  end;

end.
```

Figure 8 Fractions implemented as a real

```
unit FracTst1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, ExtCtrls, StdCtrls,
  FracTst2,     {This unit+form creates and uses fraction}
  FracAbstract, {Abstract class for fraction            }
  FracIntInt,   {Implementation of fraction with two ints}
  FracReal;     {Implementation of fraction with a real  }

type
  TTypeVars = class(TForm)
    Input: TgroupBox;
    Edit1: Tedit;
    Edit2: Tedit;
    RadioGroupRepresentation: TRadioGroup;
    ButtonCreateFraction: TButton;
    procedure RadioGroupRepresentationClick
              (Sender: TObject);
    procedure ButtonCreateFractionClick
              (Sender: TObject);
  end;

var
  TypeVars: TTypeVars;

implementation

{$R *.DFM}

procedure TTypeVars.RadioGroupRepresentationClick
          (Sender: TObject);
begin
  ButtonCreateFraction.Enabled:=True;
  if RadioGroupRepresentation.ItemIndex=0 {Real selected}
    then Representation:=TFracReal
    else Representation:=TFracIntInt
end;

procedure TTypeVars.ButtonCreateFractionClick
          (Sender: TObject);
begin
  UseFrac.ShowModal
end;

end.
```

The type type

Using the methods of the fraction.

The type variable `Representation` is set here.

The close button in the title bar is disabled.

Figure 9 The unit for the main form

```
unit FracTst2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ExtCtrls,
  FracAbstract, {abstract class for fraction            }
  FracIntInt,   {Implementation of fraction with two ints}
  FracReal;     {Implementation of fraction with a real  }

type
  TRepresentation = class of TFracAbstract;
  TUseFrac = class(TForm)
    PanelDisplay: TPanel;
    ButtonDivideByTwo: TButton;
    ButtonDestroyFraction: TButton;
    procedure ButtonDivideByTwoClick(Sender: TObject);
    procedure ButtonDestroyFractionClick(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure FormHide(Sender: TObject);
  end;

var
  UseFrac        : TUseFrac;
  Representation : TRepresentation;

implementation

{$R *.DFM}

var
  Fraction : TFracAbstract;

procedure TUseFrac.ButtonDivideByTwoClick(Sender:
TObject);
begin
  Fraction.Half;
  PanelDisplay.Caption:=Fraction.Get
end;

procedure TUseFrac.ButtonDestroyFractionClick
          (Sender: TObject);
begin
  Close
end;

procedure TUseFrac.FormShow(Sender: TObject);
var numerator,denominator:integer;
begin
  numerator  :=StrToInt(TypeVars.Edit1.Text);
  denominator:=StrToInt(TypeVars.Edit2.Text);
  MessageDlg('Creating fraction',mtConfirmation,[mbOk],0);
  Fraction:=Representation.Create(numerator,denominator);
  PanelDisplay.Caption:=Fraction.Get
end;

procedure TUseFrac.FormHide(Sender: TObject);
begin
  MessageDlg('Freeing fraction',mtConfirmation,[mbOk],0);
  Fraction.Free
end;

end.
```

The fraction is created using the chosen `Representation`.

The fraction's heap block is free-ed on form hide.

Figure 10 The unit for the "Use Fraction" form