

**A BEGINNER'S GUIDE TO
OBJECT ORIENTED PROGRAMMING
IN EUPHORIA
USING DIAMOND LITE**

Alexander CARACATSANIS
sunpsych@ncable.com.au

September 2003

CONTENTS

HEADING	PAGE
PREFACE AND ACKNOWLEDGEMENTS	3
AN ORIENTATION TO OOP	3
AN ORIENTATION TO DIAMOND LITE	4
FORMATTING AND CONVENTIONS USED HERE	5
INTRODUCTORY CONCEPTS – THE VERY BEGINNING	5
STEP 1: NO CLASS AT ALL – JUST AN INCLUDE FILE!	5
STEP 2: PREDEFINED CLASSES AND AN INSTANCE	6
STEP 3: WE CREATE OUR FIRST INSTANCE	7
STEP 4: A FIRST LOOK INSIDE AN INSTANCE – THE METHOD new()	9
STEP 5: A SECOND LOOK INSIDE AN INSTANCE – THE METHOD clone()	10
PAUSE: PROGRAM CONTEXT	11
STEP 6: A THIRD LOOK INSIDE AN INSTANCE – THE METHOD delete()	12
A RECAP...	15
CREATING OUR FIRST CLASS	
STEP 7: AN INERT CLASS	15
STEP 8: AN INERT CLASS WITH A DEFAULT CONSTRUCTOR	18
PAUSE: OVERRIDING METHODS	18
BACK TO THE CONSTRUCTOR...	19
STEP 9: AN INERT CLASS WITH A DEFAULT DESTRUCTOR	21
INTRODUCING PROPERTIES	
STEP 10: A SIMPLE CLASS WITH A PROPERTY – AND HOW TO GET IT	24
STEP 11: A SIMPLE CLASS WITH A PROPERTY – AND HOW TO SET IT	27
THE INEVITABLE "HELLO WORLD" PROGRAM!	
STEP 12: A GREETING CLASS WITH PARAMETERISED CONSTRUCTOR	31
STEP 13: A GREETING CLASS WITH PARAMETERISED DESTRUCTOR	36
STEP 14: A GREETING CLASS WITH A SETTER METHOD	38
STEP 15: THE GREETING CLASS STRIPPED DOWN	40
INTERACTING WITH THE USER	
STEP 16: A PRODUCT CLASS WITH USER INPUT	43
MAKING COPIES OF OBJECTS	
STEP 17: COPYING OBJECTS BY ASSIGNING A REFERENCE	46
A DETOUR: DL's HANDLES	51
BACK TO MAKING COPIES OF OBJECTS...	
STEP 17a: COPYING OBJECTS BY DECLARING NEW CLASS ENTITIES	53
STEP 17b: COPYING OBJECTS USING THE METHOD clone()	55
A RECAP AND A LOOK AHEAD...	58
A QUICK LOOK AT COMPOSITION	60
DEEP AND SHALLOW CLONING	62
STEP 18: CLONING A WINDOW	62
STEP 19: A WINDOW WITH A BUTTON	65
STEP 20: TWO WINDOWS, EACH WITH AN IDENTICAL NEW BUTTON	68

STEP 20a:	TWO WINDOWS, EACH WITH A SHALLOW CLONED BUTTON	69
STEP 20b:	TWO WINDOWS, EACH WITH A DEEP CLONED BUTTON	70
AN INTRODUCTION TO METHOD OVERLOADING		
STEP 20c:	DEEP AND SHALLOW CLONING TOGETHER	72
SOME EXTENSION EXERCISES		
STEP 21:	A WINDOW WITH TWO IDENTICAL BUTTONS	76
STEP 21a:	START WITH A NEW BUTTON, AND CLONE IT TWICE	78
STEP 21b:	TAKE A NEW BUTTON, CLONE IT ONCE, ASSIGN IT TWICE	79
EXCEPTION HANDLING		
STEP 22:	DO NOTHING - LET THE LANGUAGE DEAL WITH IT!	81
STEP 22a:	RETURN AN ERROR CODE	82
STEP 22b:	COMBINE ERROR-HANDLING CODE WITH NORMAL CASE	82
STEP 22c:	BUNDLE ERROR-HANDLING CODE INTO ROUTINES	83
A DETOUR: AN INTRODUCTION TO INHERITANCE		85
DL's EXCEPTION HANDLING SYSTEM		87
STEP 23:	HANDLING FATAL ERRORS	90
STEP 23a:	ONLY TESTING FOR A PENDING EXCEPTION	92
STEP 23b:	CLEARING WHICHEVER EXCEPTION WE HAPPEN TO CATCH	95
STEP 23c:	CLEARING EACH EXCEPTION IN ITS PARTICULAR WAY	97
STEP 23d:	AN EXTENSION EXERCISE	101
STEP 23e:	RETHROWING EXCEPTIONS	103
STEP 24:	A FULL CLASS DEFINITION, COMPLETE WITH EXCEPTIONS	107
INHERITANCE		111
STEP 25:	A CHILD CLASS INHERITING FROM ITS PARENT CLASS	112
STEP 25a:	A CHILD CLASS INHERITING ITS PARENT'S PROPERTY	114
STEP 25b:	INHERITANCE AND METHOD OVERRIDING	115
STEP 25c:	INHERITANCE AND PARAMETERISED CONSTRUCTORS	117
STEP 25d:	COMPLETING THE CLASSES – AN EXTENSION EXERCISE	121
POLYMORPHISM		122
CLASS HIERARCHIES		
STEP 26:	DEFINE THE CLASS HIERARCHY	123
STEP 26a:	AN OVERVIEW OF THE CLASS DEFINITIONS	124
STEP 26b:	DEFINE THE BASE CLASS – SHAPE	125
STEP 26c:	DEFINE THE SUBCLASSES – eg RECTANGLE	128
STEP 26d:	WRITE THE APPLICATION FILE	130
A FINAL CLASS JUST FOR FUN – SELF-AWARE CLASS		130
APPENDICES		
A:	DL's CLASS SYSTEM	132
B:	DL's ROUTINES BY PROGRAM CONTEXT	133
C:	DL's CONSTANTS	134
D:	DL's VARIABLE	134
E:	DL ROUTINES THAT TEST FOR TYPES	135
F:	DL ROUTINES THAT RELATE TO CLASSES	135
G:	DL ROUTINES THAT RELATE TO PROPERTIES	135
H:	DL ROUTINES THAT RELATE TO METHODS	135
I:	DL ROUTINES THAT RELATE TO ERROR HANDLING	135
J:	DL's FATAL ERROR MESSAGES	136

A BEGINNER'S GUIDE TO OBJECT ORIENTED PROGRAMMING IN EUPHORIA USING DIAMOND LITE

PREFACE AND ACKNOWLEDGEMENTS

From time to time Euphoria (Eu) programmers show an interest in developing applications using the principles and practices of Object Oriented Programming (OOP). This "Guide" is an attempt to foster this interest by presenting a series of steps that a beginner might take to learn how to write Eu programs in an object-oriented (OO) way, using the library of routines in Michael Nelson's Diamond Lite (DL).

I will assume that the reader is familiar with simple Eu programming, and the basic OOP concepts. Since I aim to address the needs of the raw beginner I will provide simple, detailed, incremental examples and explanations – those with more experience will know what material to gloss over. At the end I have included several appendices as tables summarising various aspects of DL – its system of classes; its predefined constants and variables; its routines and their allowable program contexts; and its fatal error messages. The material reflects some of my own learning steps, and my assumptions about what a beginner might need to know. I welcome feedback from other Euphorians – beginners and seasoned programmers – regarding the content, style, order, utility, and accuracy of the material, as well as comments on the suitability of the examples and suggestions for improvements in the presentation.

I chose DL because it was promoted as suitable for beginners. I have no other motive for using DL – I am not trying implicitly to promote it, and am not receiving any payment for it. Michael Nelson has read my drafts, to correct my code and errors of fact, but the writing and presentation are my own (as are any mistakes you find). We corresponded closely for more than six months, and I benefited from his corrections, comments and code. Wherever I have used his code extensively, I have acknowledged doing so in the appropriate place in the text. I have also modified his library **diamondlite.e** by adding comments to demonstrate the call chain of the routines (I have not modified the statements themselves). I have called the modified file **DL.e**. When we come to it in the text, I discuss how to benefit from its use.

AN ORIENTATION TO OOP

The Eu Reference Manual says that although Eu is not an OO language, it achieves many of the benefits of OO languages in a much simpler way. This is primarily because of its support for sequences, which allow us to create arbitrarily complex data structures. Furthermore since DL itself is written in Eu, there is nothing you can do with it that ultimately could not have been done in pure Eu. So why use OOP?

Because as programs become larger and more complex, the procedural approach to programming becomes increasingly challenged – and programs become harder to plan, code, and maintain. This is partly because in procedural programming we think in terms of operating on data – how to capture data, read it, change it, file it, display it, and so on – and eventually struggle with the complexity of achieving these tasks.

In OOP we become interested in the data itself: what it is, and what it can do. We think about our programming in ways that simulate (model) our thinking about ordinary "things" (*objects*) – their characteristics (*properties*); what they can do (*methods*); the category (*class*) they belong to; what they have *inherited* from other objects, and where they stand in a *hierarchy* of related objects; what they are *composed of*; how they are an "entity" or unit (*encapsulated*); how we never know what's inside them (their data is *hidden*) until they interact (interface) with us in predefined ways; and how that same interaction with others, in different contexts,

can lead to very different expressions of that object's properties and capabilities (*polymorphism*). Modelling real-world objects, and reusing existing code, are important aspects of OOP.

In OOP, a **class** is a design or a "blueprint" for objects belonging to the same category by virtue of sharing the same characteristics. A class models in the domain of computers, something (an *entity*) that exists in the world around us. It is an abstraction – a framework – that defines the relationship between data, the things the data can do, and the things that can be done to or with that data. Because it provides this "framework", we can think of a class as a *data structure*. And since it defines the format of an object, it only has the potential to become an object.

In the domain of computers, an **object** is a manifestation, instance, or realisation of a class during program execution – for a while, at run-time, it exists in the computer's memory.

AN ORIENTATION TO DIAMOND LITE

DL provides a predefined, consistent, preformulated way of achieving these OO capabilities – particularly encapsulation, data hiding, polymorphism, inheritance, and pass-by-reference.

In DL both a class and an object are referred to, and implemented as, an **entity**. Where the distinction is important, a class can be referred to as a *class entity*; and an object can be referred to as an *instance entity* (or, more simply, as an *instance*).

A **class** may contain *instance properties* or *instance methods* (that can be incorporated into instances that you create), and *class properties* or *class methods* (that pertain only to the class itself). An **object** (*instance entity*) may contain only *instance properties* and *instance methods*.

An **entity** consists of two parts – a *handle* (which is like a "tag" with which to refer to the entity), and a *value* (which is like a "composite" of all the entity's components). DL finds and works with an entity via its handle, which is a sequence of three integers – the first represents the class; the second represents the instance; and the third is Eu's largest negative integer.

DL provides a *base class* called **Entity**, which automatically passes three capabilities (methods) to each class that you design:

- ❖ **new()**, which makes it possible for a class to create a new instance, with properties set to their default values and methods ready and available for use when called
- ❖ **clone()**, which makes it possible for an instance to produce a copy of itself, with its properties set to the values they had at the moment of copying
- ❖ **delete()**, which makes it possible for an instance to be decommissioned

DL also has another class – a *special class* – called **Exception**, which has no properties or methods, but from which you can create new classes of your own to handle recoverable errors that might occur while your program is running. (Note, however, that you may not create instances of **Exception** or of your own exception classes.)

And whenever you design a class, DL will automatically create a subclass called **Null_Class**, which contains no properties or methods, and which is used for error reporting. DL also automatically creates **Null_Instance**, a single instance of **Null_Class**, which can only contain a reference to data that is used in reporting errors. (Note that you may not create subclasses of **Null_Class**.)

If you want your classes to have more functionality than this, you'll have to design them yourself, using the tools provided by DL and Eu. In the meantime you might like to have a look at **APPENDIX A** for a sneak preview of DL's class system – we'll discuss it fully later.

FORMATTING AND CONVENTIONS USED HERE

In the little projects I discuss here, I'll be adopting the convention of creating two source files – the first for the definition of the class (eg **ClassFile.e**); the second for the instantiation and application of the class (eg **ClassDemo.ex**). On each occasion:

ClassFile.e will begin with the statement: **include diamondlite.e**

ClassDemo.ex will begin with the statement: **include ClassFile.e**

I'll also use the following colour scheme:

- ❖ **FileName.ext** filenames
- ❖ **keyword** eg: **include**
- ❖ **datatype** eg: **atom, entity, sequence, object**
- ❖ **CONSTANT** a DL constant, eg: **NONE, NIL, CLASS, INSTANCE**
- ❖ **routine()** any routine, function, procedure, or method
- ❖ **{sequence}** an Eu sequence
- ❖ **sequence[i]** brackets
- ❖ **-- comments** eg: **-- this is a comment!**
- ❖ **greyed-out** old code we have met before, to contrast it against our new code

The application file will generally look like this:

```
procedure main()
    -- executable code here
end procedure

main()
if getc(0) then end if -- in case you need to prevent the console from disappearing
```

I've taken the liberty of adding comments to **diamondlite.e** to help you see DL in action. It will display messages on your screen – eg: **DL: call_method() calls..** – to help you "trace" the execution of your program and show you how DL interacts with your application. It's not as detailed as Eu's trace facility, but it should be sufficient for our purposes. I've called this modified file **DL.e**. I encourage you to use it whenever you want to study what DL is doing behind the scenes. (Make sure you save it in the same place as your other include files.)

INTRODUCTORY CONCEPTS – THE VERY BEGINNING

In the next six steps I want to introduce you to some concepts at the heart of DL, and the syntax for one of its very important routines. In order not to distract you with lots of code, I'll be using some very rudimentary examples. Once you've understood the concepts you won't code the specific items we use in these six steps – they are there only for introductory teaching purposes, and as a reference for revision. We'll discuss these concepts again, in more detail, when we apply them to code that will implement functionality relevant to "real world" projects.

STEP 1: NO CLASS AT ALL – JUST AN INCLUDE FILE!

Let's begin with the statement that will appear somewhere in every DL-style OOP program that we write: **include diamondlite.e** (or, for teaching purposes, **DL.e**). Let's see what happens when an application executes that statement. Consider the following file **IncludeDL.ex**:

```
-- IncludeDL.ex v1.0

include DL.e

procedure main()
```

```
end procedure
```

```
main()
```

We haven't done any OO programming of our own yet – we're just peering into DL. We're asking Eu to *include* (in **IncludeDL.ex**) whatever is in **DL.e**, and carry out any executable statements therein. For our purposes it will be enough to say that at this point Eu uses DL to do some initialisations, set some internal values, create the base class (**Entity**) and special classes (**Exception**, **Null_Class**), and create a predefined instance (**Null_Instance**). The application is now ready to locate and use any part of DL which may be called at run-time. Run this application and note the output:

```
DL: method()
DL: method()
DL: method()
```

These comments emanate from **DL.e**. They demonstrate that a routine called **method()** is executed three times. As a result of this execution, three methods will have been declared: **new()**, **clone()**, and **delete()**. They belong to the base class **Entity**, and will be inherited automatically by every normal class that we will code later on.

STEP 2: PREDEFINED CLASSES AND AN INSTANCE

We can learn more about what's happened so far, by adding comments to our file:

```
-- IncludeDL.ex v1.1

include DL.e

procedure main()
    puts(1, "\n\nEX: In main()..")
    puts(1, "\nEX: Entity      = ")   print(1, Entity)
    puts(1, "\nEX: Exception    = ")   print(1, Exception)
    puts(1, "\nEX: Null_Class   = ")   print(1, Null_Class)
    puts(1, "\nEX: Null_Instance = ")   print(1, Null_Instance)
end procedure

main()
```

Run this application and note the screen display:

```
DL: method()
DL: method()
DL: method()

EX: In main()..
EX: Entity      = {1,0,-1073741824}
EX: Exception    = {2,0,-1073741824}
EX: Null_Class   = {3,0,-1073741824}
EX: Null_Instance = {3,1,-1073741824}
```

It tells us that DL executed three calls to **method()** (to create the **new()**, **clone()**, and **delete()** methods of **Entity**). We then went to our application's procedure **main()**, which displayed four different sequences, each of three elements. Each sequence is a *handle* – a *reference* or a "tag" – that is automatically associated with one of these predefined entities.

Every time our application runs, these predefined entities will be given the same handles – ie they are *constants*. Our application will use these methods via their handles.

The first element of the handle represents the number of a class, in the order that it was created – **Entity** was created first ({1..}); **Exception** was created next ({2..}); and **Null_Class** came third ({3..}).

(Qu: What will be the value of the first element of the handle of the very next class that DL creates? **Ans: 4**)

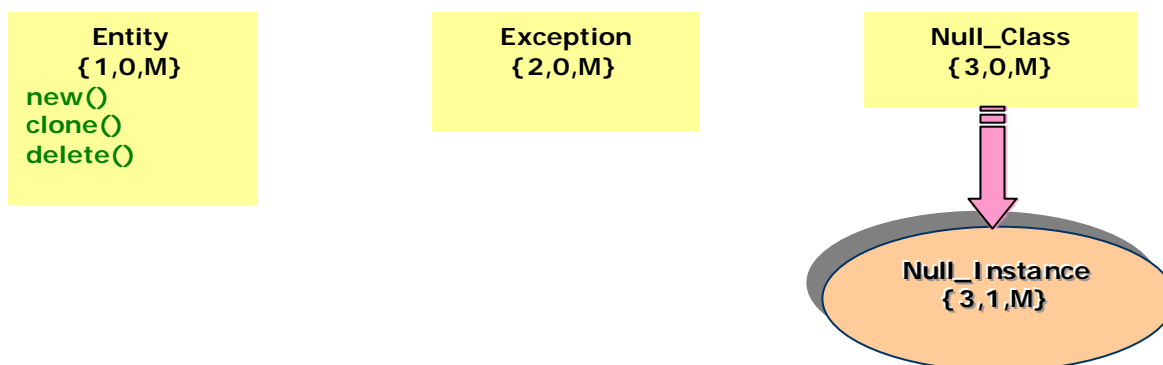
The second element of the handle represents the number of each instance, in the order of its creation. **Null_Instance** is the very first instance created, so its handle gets the integer **1** as the second element. And since **Null_Instance** is an instance of **Null_Class** (whose class number is **3**), you can see why its handle begins with **{3,1,..}**.

(Qu: What will be the value of the second element of the handle of the very next instance that DL creates? **Ans: 2**)

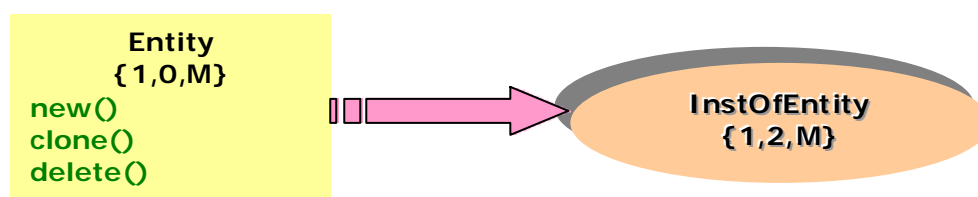
And now for the third element. It is always the number **-1073741824**, Eu's largest negative integer. DL places it there to help create a sequence that would be extremely unlikely to be one of your own application's data. It's a constant called **MARKER** in DL – I'll refer to it as **M**.

STEP 3: WE CREATE OUR FIRST INSTANCE

So far we've watched **IncludeDL.ex** execute some internal code in the file **DL.e**. We've seen it create three predefined classes, and give each of them a predefined, unique, and constant handle. And we've seen it create a predefined instance (and give it a unique handle). We can visualise the situation at this point like this:



What's the very next thing we can do? Well, although **Null_Class** and **Null_Instance** have already been created, we can't do much with them – they have no data or actions associated with them. We might be able to do something with the class **Exception**, but since it's there to help handle run-time errors (and since we haven't made any mistakes yet!), it might be best to leave it alone for the time being. **Entity** looks interesting – it's got no data, but it does have methods. We can use **Entity** as a class from which to create an *instance* (*entity*). We can represent what we need to achieve as follows:



This isn't a very useful thing to do in its own right, but it will help us explore some basic concepts of DL, and give us practice in using some of DL's syntax – particularly the routine `call_method()`, which we'll be using very often.

Our class (**Entity**) is already defined, so we don't have to write any code for it. But what we can do is to write an application to create an *instance* of **Entity**:

```
-- InstanceOfEntity.ex v1.0

include DL.e

procedure main()
    entity InstOfEntity

    puts(1, "\nEX: In main()..")
    InstOfEntity = call_method(Entity, "new", NONE)
end procedure

main()
```

This program does the following:

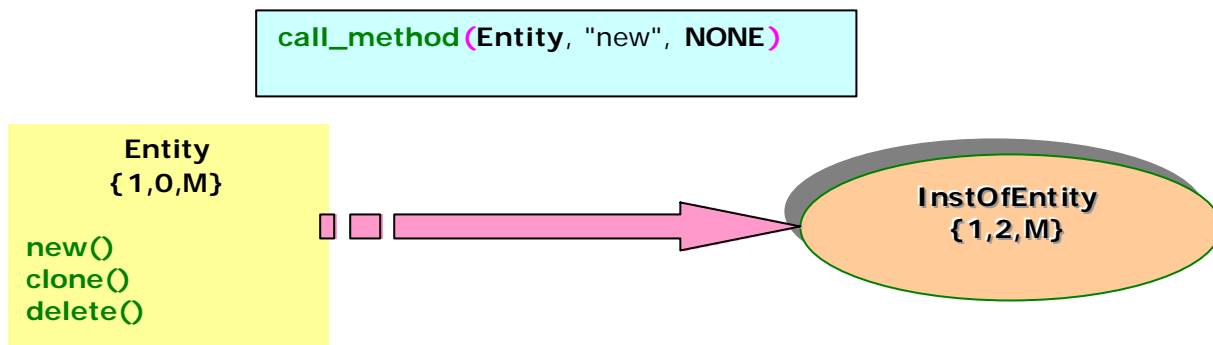
1. it includes `DL.e` and precreates the classes, instance, and other internal values we discussed above
2. it goes to `main()`
3. it declares a new variable called `InstOfEntity` of type **entity** – a sequence of three integers, being a reference to an instance (as described before)
4. it displays the message: `EX: In main()..`
5. it calls the DL routine `call_method()`, passing three arguments
6. it returns a result, a three-integer-sequence reference, which it assigns to `InstOfEntity`

The routine `call_method()` says this: "Call the method named `new()`, on the target entity named **Entity**, without passing any arguments to it". We can think of `call_method()` as DL's way of implementing **Entity.new()**.

Run the application and note the screen display:

```
DL: method()
DL: method()
DL: method()
EX: In main()..
DL: call_method() calls..
DL: Entity_new(),          which returns ref to new entity: { 1,2,M}
```

We see three calls to `method()` as part of the initial processing of the included file `DL.e`. Then we go to the file `InstanceOfEntity.ex`, which displays `EX: In main()..` and calls the DL routine `call_method()`. This then calls the DL routine `Entity_new()`, which creates the new entity (**InstOfEntity**), and returns a reference (`{1,2,M}`) to it. We can picture the situation as on the next page:



Notice that the references of both the class and its instance have the same first element – **{1,..}** – as they should, since they're from the same class. Notice that whereas **Entity**'s second element is **0** (because **Entity** is a class), the second element of **InstOfEntity** is **2** (signifying the second *instance entity* that our application has created). The third element, **MARKER (M)**, is a constant – it's the same in all references.

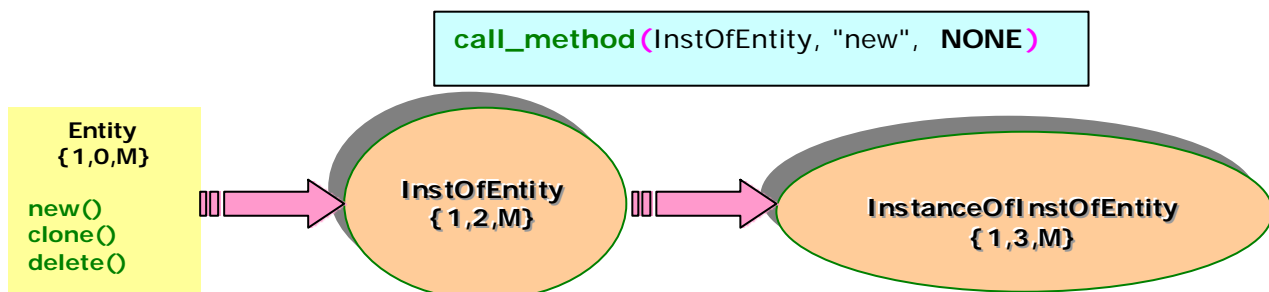
(Qu: What was the first instance entity that our program created?

Ans: the precreated **Null_Instance**, whose reference is always **{3,1,M}**)

STEP 4: A FIRST LOOK INSIDE AN INSTANCE – THE METHOD **new()**

We've created an *instance* of the predefined class **Entity** – an object that "comes alive" during the execution of our program. What can our instance do? Recalling that our instance came from **Entity**, which doesn't have any data (properties), we can assume that **InstOfEntity** doesn't have any data either, but that it must have inherited some methods.

Using the syntax with which we previously created **InstOfEntity** from **Entity**, we could try to create a new instance of this instance like this:



This syntax would mean "Call the method named **new()** on the target entity named **InstOfEntity** without passing any arguments to it" – DL's implementation of **InstOfEntity.new()**. Let's use this in a modified application – **InstanceOfInstance.ex**:

```
-- InstanceOfInstance.ex v1.0

include diamondlite.e    -- or DL.e

procedure main()
  entity InstOfEntity, InstOfInstEnt

  puts(1, "\nEX: In main()..")
  InstOfEntity = call_method(Entity, "new", NONE)
  InstOfInstEnt = call_method(InstOfEntity, "new", NONE)
end procedure
```

```
main()
```

When you run this application your screen should display something like this:

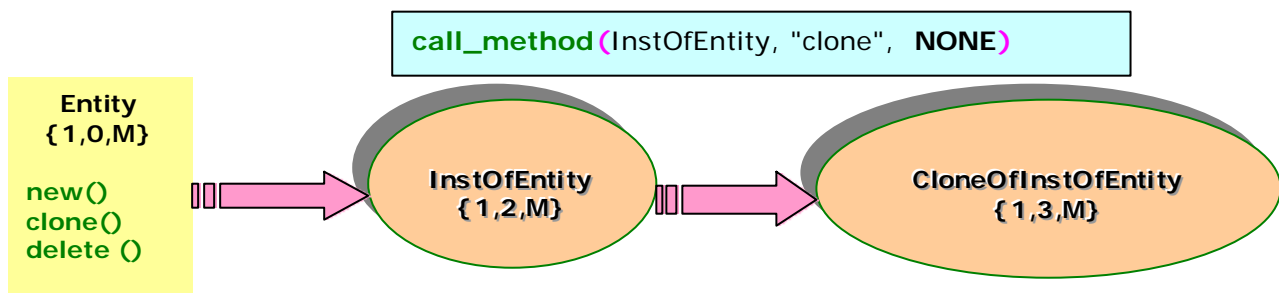
```
In call_method().
Class Entity does not define instance method new#0.
In main program.
```

We get an error message! What we've stumbled across is this: **InstOfEntity** has not inherited the method **new()** from **Entity**, and so it can't create a new instance of itself – another brand new "itself" – from scratch. When you think about it, it makes sense – "I" can't bring forth "me", "myself", anew, fresh, from zero. At best, I can only create an offspring of myself – a kind of "copy" of me. If I were a bacterium, I might even be able to split myself into two identical bacteria – "clone" copies of me – but I still couldn't produce another new "me" from scratch, with all the properties and behaviours I had at the very start of my existence (before I was even old enough to reproduce!).

So we realise that an instance doesn't inherit the method **new()** from **Entity**. Accordingly, we call **new()** a *class method* – it's available to the class, but not to an instance of the class.

STEP 5: A SECOND LOOK INSIDE AN INSTANCE – THE METHOD **clone()**

Let's turn our attention to the next method available in Entity (**clone()**), to see whether it is contained in **InstOfEntity** – because if it is, then we will be able to create an instance that is a copy of **InstOfEntity**, with all the qualities it possessed at the very moment of being copied. We can visualise our task like this:



This syntax would mean "Call the method named **clone()** on the target entity named **InstOfEntity** without passing any arguments to it" – DL's implementation of **InstOfEntity.clone()**. Let's use this in a modified application – **CloneOfInstance.ex**:

```
-- CloneOfInstance.ex v1.0

include DL.e

procedure main()
    entity InstOfEntity, CloneOfInstEnt

    puts(1, "\nEX: In main()..")
    InstOfEntity = call_method(Entity, "new", NONE)

    -- delete this – it doesn't work!
    -- InstOfInstEnt = call_method(InstOfEntity, "new", NONE)

    CloneOfInstEnt = call_method(InstOfEntity, "clone", NONE)
```

```
end procedure
```

```
main()
```

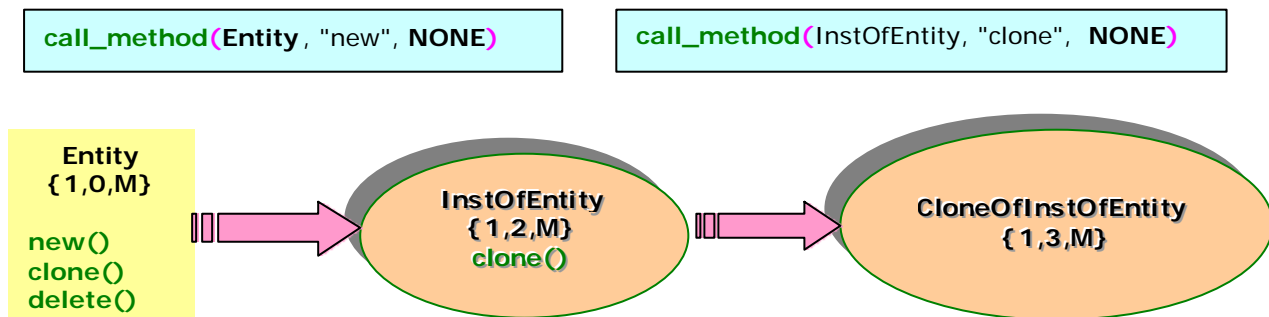
When you run this application you should see the following on your screen:

```
DL: method()
DL: method()
DL: method()
EX: In main()..
DL: call_method() calls..
DL: Entity_new(),      which returns ref to new entity: {1,2,M}
DL: call_method() calls..
DL: Entity_clone(),    which returns ref to cloned entity: {1,3,M}
```

As before we have the three calls to `method()`, to create `Entity`'s methods (`new()`, `clone()`, and `delete()`). We then go to our main program, which calls DL's `call_method()` routine; this then calls DL's `Entity_new()` routine, which creates `InstOfEntity` and returns its reference. Execution then continues at the application's next call to DL's `call_method()` routine; this then calls DL's `Entity_clone()` routine, which creates `CloneOfInstEnt` and returns its reference. Incidentally, notice that both our instances have references whose first element is 1 – because they both come from the class `Entity` (whose reference is `{1,0,M}`) – and whose second elements are 2 and 3 consecutively (since 1 has already been used as part of the reference to another instance).

(Qu: Which one? Ans: `Null_Instance`, whose reference is `{3,1,M}`)

So we've discovered that the method `clone()` is inherited by an instance – and accordingly it's called an *instance method*. (Remember that the method `new()` is a *class method* – it isn't inherited by an instance.) We can visualise the situation schematically like this:



There's another point we can make at this stage. When we represent our classes and instances pictorially, we show them as "containing" methods inside them. But when we look at the output on the screen after our application has executed with `DL.e`, we get a more dynamic picture – we see our application interacting with DL in such a way that there is a to-and-fro movement between Eu, our application, DL, and back again. Instead of thinking that classes and instances "contain" things, we can think of them as "having access to" program elements – eg: `InstOfEntity` has access to a `clone()` method, but not to a `new()` method.

A PAUSE: PROGRAM CONTEXT

DL uses the term *program context* to describe this dynamic situation. For example when the method `clone()` is executing we would recognise that program context as an *instance method*; when the method `new()` is executing we would recognise that program context as a *class method*. We haven't written any code to define a class yet, but if we had, and if we were registering a property, declaring a method, or ending the class definition, then we would

recognise that program context as the *class definition*. And when our application executes something that doesn't involve any of the above – eg displaying "**EX: In main()..**" – we would recognise that program context as the *main program*. So program context refers to *the aspect of the program that is executing at any time* – not to the physical layout or order of the code written by the programmer.

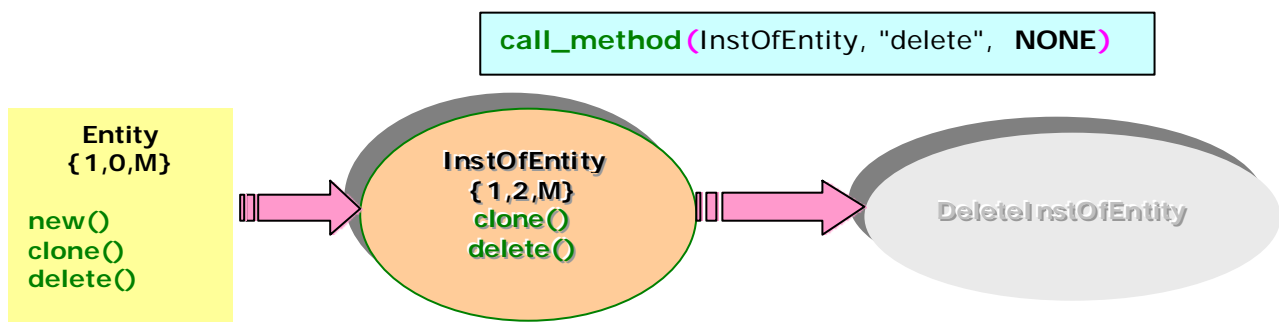
To summarise... When execution starts, the program context is main program. We then go to class definition context, which executes over the time that a class is created, properties are registered, methods are declared, and the class definition is formally closed. Then, executing **call_method()** from the main program shifts the context to instance method or class method as appropriate. When the method returns, program context returns to main program. It's helpful to be aware of the dynamic changes in the program context, because it helps you plan and write your code – some DL routines can only be used in certain program contexts, but not in others. (See **APPENDIX B** for a summary of what's allowed where.)

Finally, notice how heavily DL relies on handles. We'll have more to say about them later, but for now just notice that DL doesn't manipulate large chunks of data structures in the way we've represented them as rectangles or ellipses – it uses references to help it locate what it needs to use when it is called for.

BACK TO THE BASIC STEPS

STEP 6: A THIRD LOOK INSIDE AN INSTANCE – THE METHOD delete()

It's time for us to see whether our instance **InstOfEntity** has inherited **delete()**, the remaining method predefined by **Entity**, and to see how we might use it. We visualise our task as follows:



This syntax would mean "Call the method named **delete()** on the target entity named **InstOfEntity** without passing any arguments to it". It is DL's way of implementing **InstOfEntity.delete()**. Let's use it in a modified application – **DeletedInstance.ex**:

```
-- DeletedInstance.ex v1.0

include DL.e

procedure main()
  entity InstOfEntity, CloneOfInstEnt

  puts(1, "\nEX: In main()..")
  InstOfEntity = call_method(Entity, "new", NONE)
  CloneOfInstEnt = call_method(InstOfEntity, "clone", NONE)
  VOID = call_method(InstOfEntity, "delete", NONE)
end procedure

main()
```

Most of the syntax should be looking familiar by now. You might have been surprised by the value **VOID**. This is a DL **variable**; it means "no meaningful return value". This makes sense – we're decommissioning our instance (an act that will result in an absence of an instance), so there shouldn't be anything left afterwards to which to give a meaningful value!

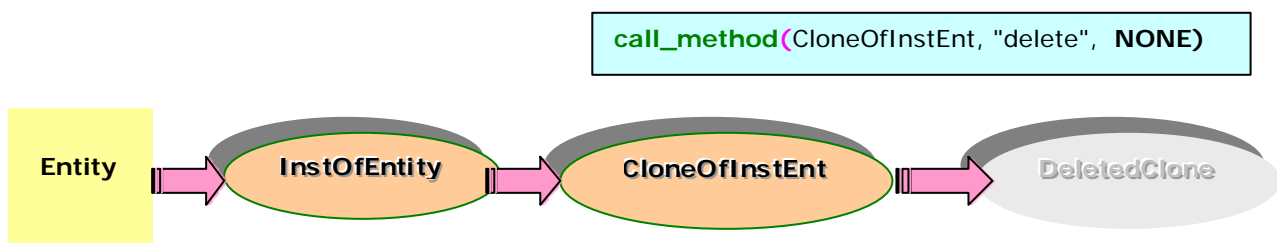
DL also has a number of **constants**. We've already met **NONE**, which means "an empty sequence" (`{ }`). There's also **NIL**, which means "no meaningful numeric value – 0". **NONE** and **NIL** can be used as default *place-holders* until a specific sequence or a specific number comes along to take their place. (Have a look at **APPENDIX C** and **D** for a summary of all these values.) Very soon we'll be finding such a use for another DL constant. But first, let's run the application and note the screen display:

```
DL: method()
DL: method()
DL: method()
EX: In main()..
DL: call_method() calls..
DL: Entity_new(),      which returns ref to new entity: { 1,2,M}
DL: call_method() calls..
DL: Entity_clone(),    which returns ref to cloned entity: { 1,3,M}
DL: call_method() calls..
DL: Entity_delete(),   which returns ref to Null_Instance: { 3,1,M}
```

Most of it should be familiar by now – creating **Entity**'s methods; going to the application; creating a new instance of **Entity** and returning its reference; creating an instance (**CloneOfInstEnt**) that is a copy of **InstOfEntity**, and returning its reference; and calling the DL routine **Entity_delete()** to decommission **InstOfEntity**. But why do we get back to **Null_Instance**? And what's happened to **CloneOfInstEnt**?

When you think about it, it should make sense. When we decommission an instance, we destroy it – we send it to the "Instance Graveyard", where it has no further existence. It has become a "non-instance" – a **Null_Instance**. All decommissioned instances end up here, irrespective of their origins. What's more, **Null_Instance** itself can be used as a default place-holder, meaning "no meaningful instance". Later on we'll see how we can replace it with more meaningful instance values.

And now to the question we asked before: what's happened to the instance **CloneOfInstEnt**? Well, just before our application ended, DL saw to it that the instance was automatically decommissioned, so we didn't see it happening. But it is possible to delete it explicitly. We can picture our task like this:



To achieve this, we will modify our application **DeletedInstance.ex** like this:

```
-- DeletedInstance.ex v1.1
include DL.e
```

```

procedure main()
    entity InstOfEntity, CloneOfInstEnt

    puts(1, "\nEX: In main()..")
    InstOfEntity = call_method(Entity, "new", NONE)
    CloneOfInstEnt = call_method(InstOfEntity, "clone", NONE)
    VOID = call_method(InstOfEntity, "delete", NONE)
    VOID = call_method(CloneOfInstEnt, "delete", NONE)
end procedure

main()

```

Run the application and notice the screen display:

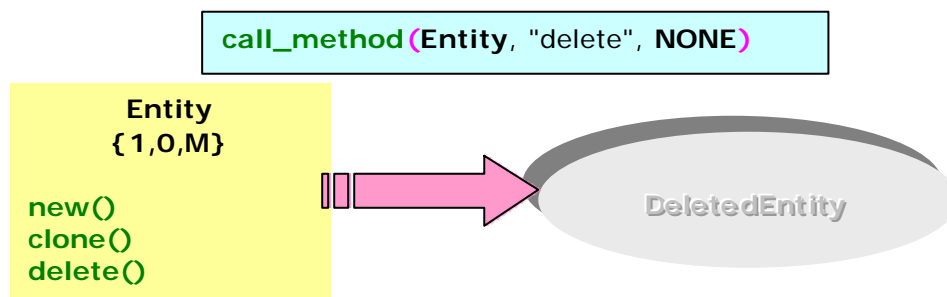
```

DL: method()
DL: method()
DL: method()
EX: In main()..
DL: call_method() calls..
DL: Entity_new(),          which returns ref to new entity: { 1,2,M}
DL: call_method() calls..
DL: Entity_clone(),       which returns ref to cloned entity: { 1,3,M}
DL: call_method() calls..
DL: Entity_delete(),      which returns ref to Null_Instance: { 3,1,M}
DL: call_method() calls..
DL: Entity_delete(),      which returns ref to Null_Instance: { 3,1,M}

```

Notice the added reference to **Null_Instance**, signifying that **CloneOfInstEnt** has been explicitly decommissioned. All the other steps have remained the same.

We can now ask another question: having successfully deleted our own instances, can we go one step further and delete the class (**Entity**) from which they were created? In other words, can we do this:



The following code in **DeletedEntity.ex** should look familiar to you by now:

```

-- DeletedEntity.ex v1.0

include DL.e

procedure main()
    VOID = call_method(Entity, "delete", NONE)
end procedure

main()

```

Run the application and note the screen display:

FATAL ERROR:
In call_method().
Class Entity does not define class method delete#0.
In main program.

We can't do that! We can't decommission a class. The error message tells us that the class **Entity** does not "contain" (or have access to) a **delete()** method capable of decommissioning itself, only *instances* of the class. This makes sense – we don't want to expose our class to the risk of death, whether by design or accident. And we don't want to destroy a perfectly good blueprint! (By the way, now might be a good time to take a quick look at **APPENDIX J**, a table summarising the meaning of all DL's error messages – then use it as a quick reference.)

PAUSE: A RECAP

Before we write more code of our own, let's summarise the main points we've made so far:

1. we've seen that our application interacts with **diamondlite.e** through some code within it, and through code that we write
2. we've seen some of **diamondlite.e**'s initialisations – particularly in creating its predefined classes and an instance, as well as declaring predefined methods for our normal classes to inherit
3. we've learnt to use the routine **call_method()** to invoke certain DL methods on their target entity, and we've had a glimpse of what happens when certain methods are called
4. we've distinguished between *class methods* and *instance methods*
5. we've tried our hand at creating a new *instance entity* (from a class), and a copy of that entity; and we've learnt how to decommission our instances (and that we cannot delete a class!)
6. we've introduced DL's system for providing *references* for our classes and instances
7. we've learnt something about the various *program contexts* that apply as our application executes
8. we've introduced some *constants* that are defined in DL, and which can act as default place-holders
9. we've learnt that some program elements may be *inherited* from classes – and that some may not!

From now on we will be writing code for our own classes, instances, and other functionality. As we proceed one step at a time, resist the temptation of thinking that you could achieve the same results using fewer lines of code in non-OO Eu. Remember that the benefits of OOP are realised in the way it helps us think about our programming tasks; in the way it helps us organise large, complex programs; in the way it helps us reuse our code; and in the way it helps us model objects in the world beyond computers.

CREATING OUR FIRST CLASS

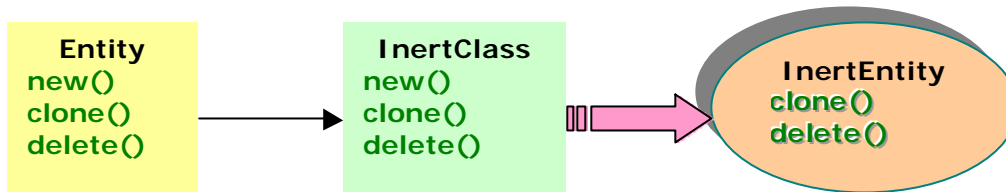
STEP 7: AN INERT CLASS

Let's suppose our task is to create an application to model a fail-safe object that could be relied upon to be impassive, remain inert, do nothing under any and all circumstances – an **InertEntity**.

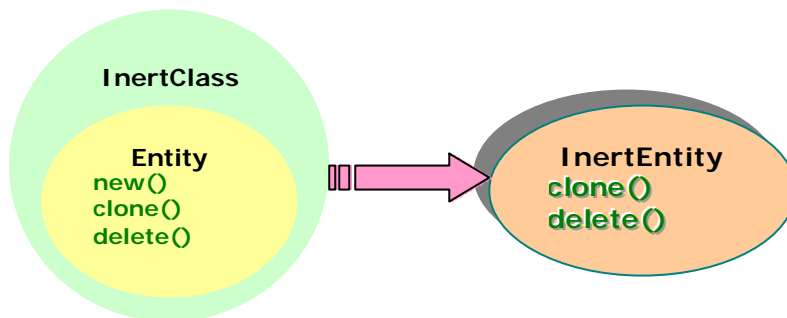
This object's blueprint would define the qualities and capabilities of all such objects – that they have no data, and that they cannot perform any actions (other than coming into existence, making copies of themselves, and being decommissioned). This class, which we can call **InertClass**, will need to ensure that any object created from it will contain nothing and do

nothing, and that it will be brought to life and eventually extinguished automatically with no ability to interact with the world. It would inherit its most basic qualities from DL's base class - **Entity** – the methods **new()**, **clone()**, and **delete()**.

We can picture the situation like this:



Or even....



These drawings represent an entity **InertEntity** that has access to methods called **clone()** and **delete()**. The object and its contents are detailed in the class definition **InertClass**, which in turn has inherited from **Entity** these methods as well as the method **new()** – to which it has access.

Our next task is to write the code in the application file to create **InertEntity**. We've already met this – it's:

```
InertEntity = call_method(InertClass, "new", NONE)
```

It says: "Call the method **new()** on **InertClass**, passing no arguments, and return a handle to assign to the entity **InertEntity**." It implements **InertClass.new()** We now have to code the class **InertClass** in the class definition file, as follows:

```
-- InertClass.e v1.0

include DL.e

global constant InertClass = class("InertClass", Entity)
end_class()
```

The first line includes the file **DL.e**. The second line calls the routine **class()**, passing as arguments the name of the class (**InertClass**) and the identifier for its superclass (**Entity**); it will return a reference (**InertClass**) to the class. From here on, any access to the class will occur only through this reference – hence the importance of making it a **global constant**. The last line calls the routine **end_class()** to terminate the class definition.

To see how the class behaves, we'll create an application file **InertDemo.ex**:

```
-- InertDemo.ex v1.0
```

```

include InertClass.e

procedure main()
    entity InertEntity
        InertEntity = call_method(InertClass, "new", NONE)
    end procedure

main()

```

The first line includes the definition of the class in **InertClass.e**. The second line declares an object called **InertEntity**, of type **entity**. The third line invokes the routine **call_method()**, which takes as arguments:

- ❖ the *reference* to the class itself (**InertClass**)
- ❖ the method **new()**, which is inherited from the base class **Entity**
- ❖ a sequence of the arguments to be passed to the method. **NONE** is a DL constant, whose value is the empty sequence – so we could code: **call_method(InertClass, "new", {})**

When you run the application file **InertDemo.ex** you should see:

```

DL: method()
DL: method()
DL: method()
DL: class() calls new_class(): returns ref to new class {4,0,M}
DL: end_class()
DL: call_method() calls..
DL: Entity_new(), which returns ref to new entity {4,2,M}

```

We see the three calls to **method()** followed by a call to DL's routine **class()** (made from **InertClass.e**), which in turn calls DL's routine **new_class()**, which returns a reference to the class **InertClass** (**{4,0,M}**), before closing the class definition with **end_class()**. The application then calls **call_method()**, which in turn calls the DL routine **Entity_new()**; this returns a reference (**{4,2,M}**) which is assigned to **InertEntity**. Had we used **diamondlite.e**, we wouldn't have seen any of this – only a blank screen, because this entity is unable to interact with us in any way.

At this point we can emphasise a few things:

- ❖ Note the order of execution of the application:
 1. some initialisations from the included file **DL.e** (lines 1, 2, 3)
 2. some processing pertaining to the included class definition file **InertClass.e** (lines 4, 5)
 3. an interaction between executable statements in the application file **InertDemo.ex**, and **DL.e** (lines 6, 7)
- ❖ Note that the first element of the handle of the class is the same as the first element of the handle of its entity (ie **4**) – as it should be, since **InertEntity** is an instance of **InertClass**.
- ❖ Note also that the second element of the handle of the entity is the integer **2** – as it should be, since this is the second entity that our application has created (**Null_Instance** was the first; its handle is **{3,1,M}**).
- ❖ Be aware that even though we can draw entities "containing" things inside them, it is more accurate to think of classes and objects "having access to" certain program elements and constructs.
- ❖ Finally, note that something else has happened automatically, behind the scenes – **InertEntity** has been decommissioned. We'll look at that process in more detail later.

Now would be a good time to read through **STEPS 1** to **6** to revise these concepts and see how they apply to our present tasks.

STEP 8: AN INERT CLASS WITH A DEFAULT CONSTRUCTOR

So far we've designed a class (**InertClass**) and have given it brief existence as an object (**InertEntity**), but we've had no say in how this object came to life, the state it was in at the moment of its creation, or how it expired. These matters were taken care of by DL, by what we could call the *inherited* "automatic constructor" and "automatic destructor".

But sometimes we do need to have a say in the state of an object at the moment of its creation. For instance, over the course of its existence an object might need to keep track of something about itself (eg how many instances it has created); that "something" will have to be set to some initial value when the object is created – we won't be able to leave it up to the default constructor to do it.

To enable us to do such things, we will have to incorporate in the design of the class a capability (a method) called a *default constructor*, to supplant (*override*) DL's "automatic constructor" (which will immediately become unavailable to any new entity of this class). This default constructor will not be able to receive data from outside the object, and it will only be able to create an object in the manner specified in the definition of the method.

Now we don't actually need a default constructor for our present purposes, but we can use it to introduce us to DL's consistent syntax for writing methods within a class. We'll use this syntax repeatedly, so I introduce it now:

```
function foo(parameter_list)
  -- statements
  return some_value
end function
method(a, b, c, routine_id("foo"))
```

This code has two parts:

- ❖ the method is defined as a **function**, taking a certain number of **parameters**, executing its **statements**, and **returning** its appropriate **value**
- ❖ the method is then *registered* with DL (declared) using the routine **method()**.

method()'s parameters are as follows:

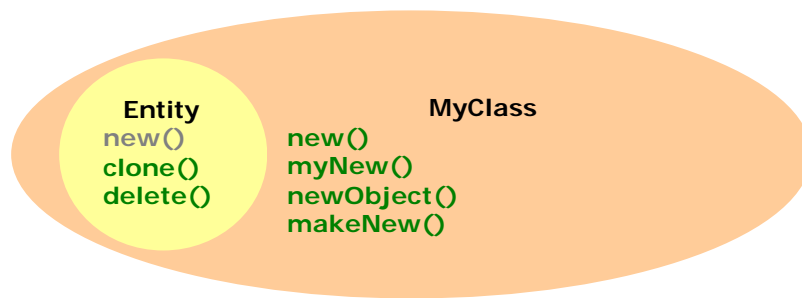
- ❖ **a** is the name we've chosen to give to this method – eg: **"myFooMethod"**
- ❖ **b** is the number of parameters in foo's parameter_list – eg: 0 or 1 etc...
- ❖ **c** is a named integer constant defined in DL, and identified by the word **INSTANCE** or **CLASS**, depending on whether this method is being called on an object, or on a class (respectively)
- ❖ **routine_id()** will return a *reference* to the method named **foo**; this reference will be used to invoke foo whenever you subsequently call **myFooMethod**

You can substitute anything you want for **foo**. I'll use the following self-documenting syntax:
ClassName_methodName_N (where **N** = number of parameters)

PAUSE: OVERRIDING METHODS

We are free to give our constructor any name at all – eg **"makeObject"** for **a** above – and it will work just fine. But we will call our default constructor **"new"**, because we will want it to be used in place of the automatic constructor (**new()**) that our class inherited from its superclass (**Entity**). When we do this, we are using our own method to *override* – to supplant; to be used instead of – the method that our class already inherited. If we don't do this, the automatic constructor (**Entity.new()**) will remain available, and someone could then use it to make a new entity of this class, bypassing our own carefully coded (and presumably important and necessary) constructor!

To understand overriding better, consider the following diagram:



It depicts a class, **MyClass**, that has inherited **Entity**'s constructor **new()** (and **clone()** and **delete()**) – as described in previous paragraphs – and that contains four (programmer-defined) constructors (**new()**, **myNew()**, **newObject()**, and **makeNew()**). Each of these will be capable of constructing some kind of new entity (according to the code in the function's body), but only **MyClass**'s **new()** will be able to *override* **Entity**'s **new()** and "disable" it so that an entity can no longer be created from it. Remember that what we have drawn here is a static diagram to illustrate an idea. If we think about it in *dynamic* terms, we can say that **MyClass**'s user-defined default constructor **new()** is there to deny access to **Entity**'s **new()**.

The general principle is that we can write methods to override (or to be used in place of) methods that a class has already inherited – provided that the overriding methods have the same name and same parameters as the overridden methods. We won't appreciate the full significance of this until we discuss *inheritance* much later on...

BACK TO THE CONSTRUCTOR...

We now have all we need in order to code explicitly how our class will construct an entity of its type. Go back to **InertClass.e** and add the following:

```
-- InertClass.e v1.1

include DL.e

global constant InertClass = class("InertClass", Entity)
    function InertClass_new_0()
        entity newInert

        newInert = call_method(super(), "new", NONE)

        return newInert
    end function
    method("new", 0, CLASS, routine_id("InertClass_new_0"))
end_class()
```

What have we done?

- ❖ We've included the library of routines in **diamondlite.e**
- ❖ We've then called **class()**, passing as arguments the name of our own class (**InertClass**) and its superclass (**Entity**)
- ❖ This function has returned a *reference* to our own class – a reference that we've assigned to a global constant with the same name as our class. (We could've given it any other name, but this convention is self-documenting – we'll always know which class we're referring to.)

- ❖ We've then defined our *default constructor* (method) as a function, and registered it with DL using the routine **method()**
- ❖ We've called our function `InertClass_new_0` because it will call the **new()** method that **InertClass** inherited from the base class **Entity**. And we've added the zero because there will be no parameters.
- ❖ We've then declared a variable of the predefined type **entity**; we've called it **newInert**. (We could've called it anything – even "new". But be careful – **Entity** [the base class] is not the same as **entity** [the predefined type]; and **new** [if used as an identifier for a variable of type **entity**] is not the same as **"new"** [the name of the **new()** method].)
- ❖ We've then invoked **call_method()**, passing it three arguments. This will call the method **new()** (to which it passes an empty sequence [**NONE**] of arguments) of the superclass of **InertClass**. (Think of **call_method()** as DL's way of implementing **SuperClass.new()**.)
- ❖ **call_method()** has returned a reference to the newly created object
- ❖ We've assigned this to the variable **newInert**, which is returned by the function
- ❖ We've then registered the method by calling the procedure **method()**, to which we've passed as arguments:
 - the name we've given to our method – **"new"**, to override **Entity's new()**
 - the number of arguments we are passing – in this case **0**
 - the named constant signifying that we're using a class method – **CLASS**
 - and a reference to our default constructor method – **"InertClass_new_0"**
- ❖ Finally, we've ended the class definition by calling the procedure **end_class()**

To help us demonstrate that our default constructor method really does something, I'll add one more statement:

```
-- InertClass.e v1.2

include DL.e

global constant InertClass = class("InertClass", Entity)
  function InertClass_new_0()
    entity newInert

    newInert = call_method(super(), "new", NONE)
    puts(1, "\nDL: About to leave default constructor.. ")

    return newInert
  end function
  method("new", 0, CLASS, routine_id("InertClass_new_0"))
end_class()
```

To see how the class behaves now, we'll change **InertDemo.ex** to this:

```
-- InertDemo.ex v1.1

include InertClass.e

procedure main()
  entity InertEntity

  puts(1, "\nEX: Before construction of the object.. ")
  InertEntity = call_method(InertClass, "new", NONE)
  puts(1, "\nEX: After construction of the object")
end procedure
main()
```

This file contains an application of the class. As before, it includes our class definition in file **InertClass.e**. As before, it declares an object (**InertEntity**) of type **entity**. Then it displays a short message informing us that we haven't yet created our object. We then invoke **call_method()** which calls the method **new()** (with no arguments), which we utilised when we defined and registered our default constructor. (Think of **call_method()** as implementing **InertClass.new()**.) We are now taken to our own constructor, which displays a short message telling us that we're about to leave it, before going back to the main program (where we're told that we've finished instantiating our object).

When you run **InertDemo.ex** you'll see the following screen display:

```
DL: method()
DL: method()
DL: method()
DL: class() calls new_class(): returns ref to new class {4,0,M}
DL: method()
DL: end_class()
EX: Before construction of the object..
DL: call_method() calls..
DL: super(), which calls the method of this target's direct superclass
DL: call_method() calls..
DL: Entity_new(), which returns ref to new entity {4,2,M}
DL: About to leave default constructor..
EX: After construction of the object
```

Notice the three calls to **method()** (part of the initial processing of **DL.e**) followed by a call to **class()**. This calls **new_class()**, which returns a reference (**{4,0,M}**) to **InertClass**, which calls **method()** (to register our default constructor) before ending the class definition with **end_class()**. Execution continues in the application file

(**EX: Before construction of the object..**),

which calls the **new()** method of its target (**InertClass**). We are now in our default constructor, which contains explicit instructions for creating an object of the class. It declares a variable (**newInert**) of type **entity**, then calls the DL routine **super()**, which in effect gains access to the overridden method – **Entity's new()** – and returns a reference (**{4,2,M}**) to the new entity (**newInert**). After a short message to confirm that we've been in the default constructor

(**DL: About to leave default constructor..**)

execution continues in the application file

(**EX: After construction of the object**).

Had we run our application using **diamondlite.e**, we would have seen the following:

```
EX: Before construction of the object..
DL: About to leave default constructor..
EX: After construction of the object
```

Again, **InertEntity** was decommissioned silently and automatically by DL before our application ended. The next step is to learn how we can control this process.

STEP 9: AN INERT CLASS WITH A DEFAULT DESTRUCTOR

It's now time to be more explicit about the destruction of our entity. We introduced this topic in **STEP 6**, where we found that DL provides us with a destructor (**delete()**) that's automatically inherited (from the base class **Entity**) by every class we write. We've been relying on it to decommission our object automatically at the end of our application.

But our object may have special requirements of its own that need to be taken into account. It may need to release certain resources that it's still holding onto. It may need to divest itself of things (even other entities) it has accumulated. It may even need to keep track of the process of its own demise.

To fulfill these purposes, the programmer has to design a *default destructor*. Reread **STEP 6** and **STEP 8**, with the default destructor in mind. The concepts, and most of the syntax, will apply here too. All we need do is to note the differences.

Let's go back to our last version of **InertClass.e** and add the code for our default destructor:

```
-- InertClass.e v1.3

include DL.e

global constant InertClass = class("InertClass", Entity)
  function InertClass_new_0()
    entity newInert

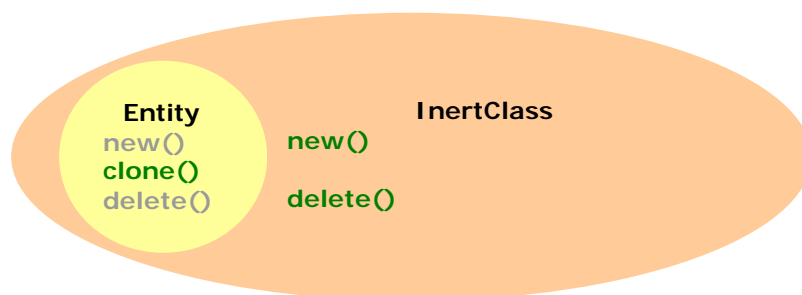
    newInert = call_method(super(), "new", NONE)
    puts(1, "\nDL: About to leave default constructor.. ")

    return newInert
  end function
  method("new", 0, CLASS, routine_id("InertClass_new_0"))

  function InertClass_delete_0()
    puts(1, "\nDL: In default destructor, destructing the object..")

    return call_method(super(), "delete", NONE)
  end function
  method("delete", 0, INSTANCE, routine_id("InertClass_delete_0"))
end_class()
```

What have we done? We've overridden the method **delete()** that our class inherited from **Entity**. We can diagram the situation as below:



And we've passed to **method()** a named constant called **INSTANCE**, to signify that the method is to apply to an object of the class, rather than to the class itself. To see how the class behaves now, let's go back to the file **InertDemo.ex** and add two more statements:

```
-- InertDemo.ex v1.2

include InertClass.e

procedure main()
```



```

entity InertEntity

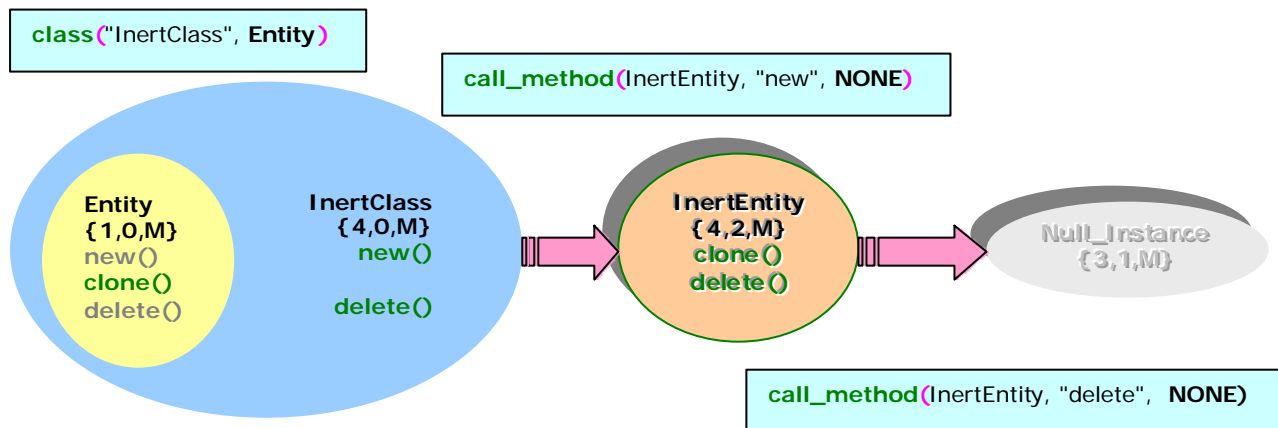
puts(1, "\nEX: Before construction of the object.. ")
InertEntity = call_method(InertClass, "new", NONE)
puts(1, "\nEX: After construction of the object")

VOID = call_method(InertEntity, "delete", NONE)
puts(1, "\nEX: After destruction of the object")
end procedure

main()

```

This file contains an application of the class. Everything is as it was before, apart from the call to the function `call_method()`, which in effect calls the method `delete()` (with an empty sequence of arguments) upon our entity (`InertEntity`), and assigns the returned value to `VOID` (a variable used to hold discarded values). (Think of `call_method()` as implementing the syntax: `InertEntity.delete()`.) We're taken to the default destructor we designed, and are rewarded with a short message telling us the deed is being done. We're then returned to the main application, to see a message telling us it's all over. We can picture what we've done, like this:



We'll run `InertDemo.ex` with `DL.e` to study the execution of the program:

```

DL: method()
DL: method()
DL: method()
DL: class() calls new_class(): returns ref to new class {4,0,M}
DL: method()
DL: method()
DL: end_class()
EX: Before construction of the object..
DL: call_method() calls..
DL: super(), which calls the method of this target's direct superclass
DL: call_method() calls..
DL: Entity_new(), which returns ref to new entity {4,2,M}
DL: About to leave default constructor..
EX: After construction of the object
DL: call_method() calls..
DL: In default destructor, destructing the object..
DL: super(), which calls the method of this target's direct superclass
DL: call_method() calls..
DL: Entity_delete(), which returns ref to Null_Instance: {3,1,M}

```

EX: After destruction of the object

The greyed-out portions of the display represent output that is exactly the same as that of **STEP 8**. Notice the extra **method()** during the class definition program context – that is the result of declaring an extra method (the default destructor that we coded, which wasn't there before). We pick up the story from the point after which we have left the default constructor and are back in **main()**. We are then taken to our default destructor, which overrides the automatic destructor to return a reference to an entity that represents the final resting place of all destructed entities – **Null_Instance**, {3,1,M}.

Had we run **InertDemo.ex** with **diamondlite.e** we would have seen the following output:

```
EX: Before construction of the object..  
DL: About to leave default constructor..  
EX: After construction of the object  
DL: In default destructor, destructing the object..  
EX: After destruction of the object
```

INTRODUCING PROPERTIES

STEP 10: A SIMPLE CLASS WITH A PROPERTY – AND HOW TO GET IT

So far our entities have merely come and gone. The programmer has displayed signs of their existence ("We are here, constructing..."; "..., destructing..."; etc), but we have yet to hear from the entities themselves, because they had nothing within them to say, and nothing to say it with anyway. We need to give our entities some data (*properties*).

We incorporate a property into our class definition by *registering* it with DL using the procedure **property(a, b, c)**, which takes three arguments:

- ❖ **a** is the name we choose to give to this property – eg **"myProperty"**
- ❖ **b** is a named constant, either **INSTANCE** or **CLASS**, depending on whether we're dealing with what's inside an object of the class, or with the class as a whole
- ❖ **c** is the value of the property – eg **"priceless"**, or **1000000** etc

To see how this might work, let's build up a new class that will contain some data. We will create a file **SimpleClass.e** (To keep things uncluttered, we'll use the automatic constructor and destructor.)

```
-- SimpleClass.e v1.0  
  
include DL.e  
  
global constant SimpleClass = class("SimpleClass", Entity)  
    property("myName", INSTANCE, "Alex")  
end_class()
```

We've defined a class (**SimpleClass**) that closely resembles the definition of **InertClass** (see **STEP 7**), except that now it has something in it – a property. We've declared, initialised, and registered with DL, this property (a variable) that will be a component of each object (**INSTANCE**) of the class; we've called this property **myName**, and we've given it an initial value "Alex".

To see how this class behaves, we'll write a file **SimpleDemo.ex**, in which we'll create an entity of the class; and we'll design code to access the property within the class. Look at this:

```
-- SimpleDemo.ex v1.0
```

```

include SimpleClass.e

procedure main()
    entity MySimpleObject

        MySimpleObject = call_method(SimpleClass, "new", NONE)
        puts(1, myName)
    end procedure

main()

```

The file is almost exactly like that of **InertDemo.ex**, but it adds a call to Eu's routine **puts()**, to display a human-readable string ("Alex") which is the value of the variable (**myName**), which is the sole piece of data (property) available to **MySimpleObject** from its class **SimpleClass**. Run **SimpleDemo.ex**

You should see an error message – **myName has not been declared**. This may have surprised you, because we have already declared **myName** – in **property()**. What happened? We've tried to use **puts()** to interrogate **MySimpleObject** and get the value of **myName** directly. We need to remember that in OOP the data (*properties*) and the operations (*methods*) allowable upon it, are bound together (*encapsulated*) in the class, thereby hiding the details from our applications, except through strictly defined access points (*interfaces*).

DL provides a routine **get_property()**, which takes two arguments – the name of the *entity* (eg **MySimpleObject**), and the name of the *property* (eg **myName**) – and returns the *value* of the property (eg: "Alex"). Let's modify **SimpleDemo.ex**, using this routine to give us access to the property (which is turning out to be a very private individual).

```

-- SimpleDemo.ex v1.1

include SimpleClass.e

procedure main()
    entity MySimpleObject
        MySimpleObject = call_method(SimpleClass, "new", NONE)

        -- remove this – it doesn't work!
        -- puts(1, myName)

        puts(1, get_property(MySimpleObject, "myName"))
    end procedure

main()

```

We've now asked **puts()** to ask **get_property()** to return the value ("Alex") of this object's (**MySimpleObject**) property (**myName**). Run **SimpleDemo.ex** to see....
an error message! We're told:

```

In get_property()
Access to SimpleClass instance property myName denied.
In main program.

```

We're denied access to the property directly from our application – the property is a private member of the class. (All properties are *private* in DL!) We need an access point (or "bridge", or interface) between our object and the application that's trying to get at its data. The routine

`call_method()` is available for this task. It will require a suitable method to call. The following changes to `SimpleClass.e` define and register a method (`getName`) implemented as the `function` `SimpleClass_getName_0()`:

```
-- SimpleClass.e v1.1

include diamondlite.e    -- or DL.e

global constant SimpleClass = class("SimpleClass", Entity)
    property("myName", INSTANCE, "Alex")

    function SimpleClass_getName_0()
        puts(1, "\nDL: getName(), whose target is..")

        return get_property(this(), "myName")
    end function
    method("getName", 0, INSTANCE, routine_id("SimpleClass_getName_0"))
end_class()
```

What have we done? Having generated a reference (`SimpleClass`) for our class (`"SimpleClass"`) whose superclass is `Entity`, we called the procedure `property()` to register an instance variable (`INSTANCE`), called `MyName`, whose value is `"Alex"`. We then designed a programmer-defined function (`SimpleClass_getName_0`), to invoke the function `get_property()` which will identify the variable called `"myName"` as a property of this class. We then called the procedure `method()` to register the details with DL. Finally, we ended the class definition with `end_class()`. Let's now modify `SimpleDemo.ex` accordingly:

```
-- SimpleDemo.ex v1.2

include SimpleClass.e

procedure main()
    entity MySimpleObject
    sequence itsValue

    MySimpleObject = call_method(SimpleClass, "new", NONE)

    -- remove these – they don't work!
    -- puts(1, myName)
    -- puts(1, get_property(MySimpleObject, "myName"))

    itsValue = call_method(MySimpleObject, "getName", NONE)
    puts(1, "\nEX: MySimpleObject.myName = " & itsValue)
end procedure

main()
```

Now, `puts()` will work once `call_method()` invokes the `getName` method on `MySimpleObject` and returns the property's value (assigned to `itsValue`). Run the application with `DL.e` and note the output:

```
DL: method()
DL: method()
DL: method()
DL: class() calls new_class(): returns ref to new class {4,0,M}
DL: property()
```

```
DL: method()
DL: end_class()
DL: call_method() calls..
DL: Entity_new(), which returns ref to new entity {4,2,M}
DL: call_method() calls..
DL: getName(), whose target is..
DL: this(), which returns ref to target entity: {4,2,M}
DL: get_property() returns value of instance property
EX: MySimpleObject.myName = Alex
```

Much of the display should look familiar by now...

- ❖ the three calls to **method()**
- ❖ the class definition,
 - getting a reference (**{4,0,M}**) to the class **SimpleClass**,
 - registering a **property()** and a **method()**,
 - and ending with **end_class()**
- ❖ using the inherited default constructor to return a reference (**{4,2,M}**) to an instance (**MySimpleObject**) of the class

Using **call_method()**, we then invoke the method **getName()** on **MySimpleObject**. This gives us access to our function **SimpleClass_getName_0()**. Notice the call to the DL routine **this()** – we'll see it often. It returns the reference to the entity that is the target of our current method – here, it happens to be **MySimpleObject, {4,2,M}** – and uses it as the target of the next call (in this case, a call to the routine **get_property()**). (We can imagine a dialogue: "Call the entity's method. Which entity's method? This entity's method.") The result is that we gain access to the value of the (otherwise private) property (**myName**).

STEP 11: A SIMPLE CLASS WITH A PROPERTY – AND HOW TO SET IT

We've had some interaction with the object, in the sense that we've found a way to access the value of its (private) property, using a consistent means provided by the function **call_method()**, which has allowed us to implement a public accessor method (in this case, a getter - **getName**); we were then able to call the Eu routine **puts()**, to display the value.

Now we need to know how to access a private property and have our application change the property's value – by order of the programmer, or by input from the user. Thinking back to the discussion in **STEP 10**, we can make the following predictions:

- ❖ that our application won't be able to change the property directly – by assigning a new value to it – ie **myName = "Christopher"** won't work
- ❖ that in our class definition we will have to define another public accessor method – this time a setter – which we might call **"setName"**
- ❖ that this method will need to be implemented as a function whose signature is likely to be **MySimpleClass_setName_1(parameter)**, and which will likely be registered using the procedure **method()**
- ❖ that our application won't be able to call the setter directly, but that it may invoke it via a call to **call_method()**

Let's convince ourselves about the first point. We won't change **SimpleClass.e**, but we'll change **SimpleDemo.ex** as follows:

```
-- SimpleDemo.ex v1.3

include SimpleClass.e

procedure main()
  entity MySimpleObject
  sequence itsValue
```

```

MySimpleObject = call_method(SimpleClass, "new", NONE)

myName = "Christopher"
itsValue = call_method(MySimpleObject, "getName", NONE)
puts(1, "\nMySimpleObject.myName = " & itsValue)
end procedure

main()

```

This application appears to be doing the following things:

- ❖ creating a new object (**MySimpleObject**) of type entity, as an instance of **SimpleClass**
- ❖ assigning the name "**Christopher**" to the object's private property **myName**
- ❖ declaring a variable **itsValue** of type **sequence**
- ❖ accessing the public method **getName**
- ❖ displaying the value of **myName**

Run the application, and confirm that you get....

an error message: **myName has not been declared**

You probably realise already that the application doesn't recognise the object's property (**myName**). Let's go back to the file **SimpleClass.e**, and define a public method analogous to **getName()**, to give us a way of accessing the property:

```

-- SimpleClass.e v1.2

include DL.e

global constant SimpleClass = class("SimpleClass", Entity)
  property("myName", INSTANCE, "Alex")

  function SimpleClass_getName_0()
    puts(1, "\nDL: getName(), whose target is..")

    return get_property(this(), "myName")
  end function
  method("getName", 0, INSTANCE, routine_id("SimpleClass_getName_0"))

  function SimpleClass_setName_1(sequence anyName)
    puts(1, "\nDL: setName(), whose target is..")

    set_property(this(), "myName", anyName)

    return NIL
  end function
  method("setName", 1, INSTANCE, routine_id("SimpleClass_setName_1"))
end_class()

```

This class definition is almost exactly like the one in **STEP 10**, except for the function **SimpleClass_setName_1()**, which takes one parameter (**anyName**) of type **sequence**. This function calls the procedure **set_property()**, which sets this class' property **myName** to the value passed in as **anyName**. The procedure **method()** then registers the name of the method as "**setName**".

Turning now to our application, we might think of calling **SimpleClass_setName_1()** directly, like this:

```

-- SimpleDemo.ex v1.4

include SimpleClass.e

procedure main()
  entity MySimpleObject
  sequence itsValue

  MySimpleObject = call_method(SimpleClass, "new", NONE)

  -- delete this – it doesn't work!
  -- myName = "Christopher"

  SimpleClass_setName_1("Christopher")
  itsValue = call_method(MySimpleObject, "getName", NONE)
  puts(1, "\nMySimpleObject.myName = " & itsValue)
end procedure

main()

```

This results in an error message: **SimpleClass_setName_1() has not been declared**
 Clearly this identifier is outside the scope of the application.

Or we could even try calling **set_property()** direct, like this:

```

-- SimpleDemo.ex v1.5

include SimpleClass.e

procedure main()
  entity MySimpleObject
  sequence itsValue

  MySimpleObject = call_method(SimpleClass, "new", NONE)

  -- delete these – they don't work!
  -- myName = "Christopher"
  -- SimpleClass_setName_1("Christopher")

  set_property(this(), "myName", "Christopher")

  itsValue = call_method(MySimpleObject, "getName", NONE)
  puts(1, "\nMySimpleObject.myName = " & itsValue)
end procedure

main()

```

We get another error message: **In this(). Not allowed.**

This is because neither **set_property()** nor **this()** are allowed during main program context – only during method context. The correct approach, of course, is to use **call_method()**:

```

-- SimpleDemo.ex v1.6

include SimpleClass.e

```



```

procedure main()
  entity MySimpleObject
  sequence itsValue

  MySimpleObject = call_method(SimpleClass, "new", NONE)

  -- delete the folowing statements – none of them work!
  --myName = "Christopher"
  --SimpleClass_setName_1("Christopher")
  --set_property(this(), "myName", "Christopher")

  VOID = call_method(MySimpleObject, "setName", {"Christopher"})

  itsValue = call_method(MySimpleObject, "getName", NONE)
  puts(1, "\nMySimpleObject.myName = " & itsValue)
end procedure

main()

```

Run the application with **DL.e** and note the output:

```

DL: method()
DL: method()
DL: method()
DL: class() calls new_class(): returns ref to new class { 4,0,M}
DL: property()
DL: method()
DL: method()
DL: end_class()
DL: call_method() calls..
DL: Entity_new(), which returns ref to new entity { 4,2,M}
DL: call_method() calls..
DL: setName(), whose target is..
DL: this(), which returns ref to target entity: { 4,2,M}
DL: set_property() sets value of instance property
DL: call_method() calls..
DL: getName(), whose target is..
DL: this(), which returns ref to target entity: { 4,2,M}
DL: get_property() returns value of instance property
EX: MySimpleObject.myName = Christopher

```

The greyed-out portions are the same as those in the previous step(s), and should be looking quite familiar by now. Notice that we have had to register one more **method()** in the class definition – **setName()**. Notice that when **call_method()** is invoked by the main program, it calls the method **setName()** on its target instance **MySimpleObject**. Note that this entity's reference is **{ 4,2,M}**, and that this is the sequence that is consistently returned by the DL routine **this()** – as it should be, since **setName()** and **getName()** are called (in turn) on the same entity. Notice also that whereas **getName()** is coded before **setName()** in the file, they are executed by the application in the reverse order – illustrating that *program context* does not necessarily follow the layout of the code.

Oh and notice the value of the property – it isn't "**Alex**" any more!

Now suppose that we wanted to display the value of the property within the function **SimpleClass_setName_1()**, immediately after the property's new value had been set. Our intuition might be to code it like this:

-- in **SimpleClass.e v1_3**

```
function SimpleClass_setName_1(sequence anyName)
  puts(1, "\nDL: In the setName method...\n")

  set_property(this(), "myName", anyName)
  puts(1, "\nDL: " & myName)

  return NIL
end function
method("setName", 1, INSTANCE, routine_id("SimpleClass_setName_1"))
```

Make this change to your function in **SimpleClass.e**, and run **SimpleDemo.ex (v1.7)**.
You should get an error message: **myName has not been declared**

The property is not within the scope of this function, even though it's in the same class definition. The correct way to achieve this functionality, is to use the routine **get_property()**:

```
function SimpleClass_setName_1(sequence anyName)
  puts(1, "\nDL: In the setName method...\n")

  set_property(this(), "myName", anyName)
  printf(1, "The property = %s\n", {get_property(this(), "myName")})

  return NIL
end function
method("setName", 1, INSTANCE, routine_id("SimpleClass_setName_1"))
```

Change your **SimpleClass.e** file to incorporate the **printf()** routine (call it **v1.4**), and run **SimpleDemo.ex** (call it **v 1.8**) with **DL.e**. Examine the screen display using the discussion above, and get a feel for the dynamic way the program context changes as the application interacts with the class definition and with DL itself.

THE INEVITABLE "HELLO WORLD" PROGRAM!

We now know enough to try our hand at writing an OOP version of the **Hello World** program. Start using **diamondlite.e** from now on, but feel free to go back to **DL.e** if you get stuck, want to clarify what your program is doing, or want to revise something.

STEP 12: A GREETING CLASS WITH A PARAMETERISED CONSTRUCTOR

If you think about **InertClass** and **SimpleClass**, you'll realise that their construction (and, for that matter, their destruction) has been determined by the code in the file defining the class. This hasn't been a terrible limitation, because we've been able to set the initial value of properties using the procedure **property()**, and we've learnt how to design (*accessor*) methods to assign new values to properties (*setters*) and retrieve those values (*getters*).

But sometimes it is necessary or desirable to have the application initialise the properties of the class, either through statements in the code itself, or else by user input. Moreover, if the class has to keep track of its objects in some way, any book-keeping tasks that depend on input can be done here. The *parameterised constructor* is a programmer-defined method with which to accomplish such tasks.

We'll design a class that contains a property whose initial value is an empty sequence – a string of length 0. We'll also incorporate in our design the capability to have the property

initialised by the application the moment it instantiates the class (rather than at some later time – by using a *setter*, for instance).

We will create a file called **GreetingClass.e** – it will look very similar to the minimal versions of our previous classes. (The comments are numbered in the order I wrote them.)

```
-- GreetingClass.e v1.0

-- 1 include the library
include diamondlite.e

-- 2 get a reference to the class, whose superclass is Entity
global constant GreetingClass = class("GreetingClass", Entity)

-- 4 register a property, and initialise it to an empty sequence
property("message", INSTANCE, NONE)

-- 5 define a (redundant) default constructor; it takes no arguments;
-- when called, it will override the automatic constructor inherited from Entity
function GreetingClass_new_0()
    entity newGreeting
    newGreeting = call_method(super(), "new", NONE)

    return newGreeting
end function
method("new", 0, CLASS, routine_id("GreetingClass_new_0"))

-- 6 define a parameterised constructor; it takes one argument;
-- when called it will not override the automatic constructor,
-- because it has a different signature: new( param )
function GreetingClass_new_1(sequence msg)
    entity newGreeting
    newGreeting = call_method(super(), "new", NONE)

    set_property(newGreeting, "message", msg)

    return newGreeting
end function
method("new", 1, CLASS, routine_id("GreetingClass_new_1"))

-- 7 define a method to get the property's value
function GreetingClass_getMessage_0()
    sequence text
    text = get_property(this(), "message")

    return text
end function
method("getMessage", 0, INSTANCE, routine_id("GreetingClass_getMessage_0"))

-- 8 define a (redundant) default destructor; it takes no arguments;
-- when called, it will override the automatic destructor inherited from Entity
function GreetingClass_delete_0()
    return call_method(super(), "delete", NONE)
end function
method("delete", 0, INSTANCE, routine_id("GreetingClass_delete_0"))

-- 3 end the class definition
```

```
end_class()
```

There should be no surprises here – we've encountered all this code before. We will now write a file **GreetingDemo.ex** to instantiate the class and implement its functionality.

```
-- GreetingDemo.ex v1.0

-- 1 include the file with the class definition
include GreetingClass.e

-- 2 define the procedure main()
procedure main()
    -- 3 declare the variable myGreetingObject of type entity
    entity myGreetingObject

    myGreetingObject = call_method(GreetingClass, "new", {"Hello World!"})
end procedure

-- 4 call the procedure main()
main()
```

Run the application. You'll see a blank screen – all we've done is to create a new entity (**myGreetingObject**) and initialise its property (**message**) to **"Hello World"**.

Now that we've got the skeleton set down, let's make some changes to help us see how the object of the class behaves under the control of the application. For a start let's create two objects of the class – one to be constructed by the *default* constructor; the other by the *parameterised* constructor. And let's add some messages to inform us which stage of the program we're in at any given moment.

```
-- GreetingClass.e v1.1

include diamondlite.e

global constant GreetingClass = class("GreetingClass", Entity)
    property("message", INSTANCE, NONE)

    function GreetingClass_new_0()
        entity newGreeting

        puts(1, "\nNow in default destructor...")
        newGreeting = call_method(super(), "new", NONE)

        return newGreeting
    end function
    method("new", 0, CLASS, routine_id("GreetingClass_new_0"))

    function GreetingClass_new_1(sequence msg)
        entity newGreeting

        puts(1, "\nNow in parameterised constructor...")
        newGreeting = call_method(super(), "new", NONE)
        set_property(newGreeting, "message", msg)

        return newGreeting
    end function
```

```

method("new", 1, CLASS, routine_id("GreetingClass_new_1"))

function GreetingClass_getMessage_0()
    sequence text

    puts(1, "\nNow in method getMessage()....")
    text = get_property(this(), "message")

    return text
end function
method("getMessage", 0, INSTANCE, routine_id("GreetingClass_getMessage_0"))

function GreetingClass_delete_0()
    puts(1, "\nNow in default destructor....")

    return call_method(super(), "delete", NONE)
end function
method("delete", 0, INSTANCE, routine_id("GreetingClass_delete_0"))
end_class()

```

These simple messages will tell us whether the different components of the class definition execute on cue. Now let's change **GreetingDemo.ex** to construct two objects in different ways:

```

-- GreetingDemo.ex v1.1

include GreetingClass.e

procedure main()
    entity myDefGreetObj,      -- to be constructed by default constructor
           myParamGreetObj    -- to be constructed by parameterised constructor

    puts(1, "\nNow in main(), before object construction....")

    myDefGreetObj = call_method(GreetingClass, "new", NONE)
    myParamGreetObj = call_method(GreetingClass, "new", {"Hello World!"})

    puts(1, "\nNow in main(), after object construction....")
end procedure

main()

```

When you run this application you should see four lines displayed on the screen:

```

Now in main(), before object construction....
Now in default constructor....
Now in parameterised constructor....
Now in main(), after object construction....

```

You can confirm that the application ran as desired. You may be wondering why we didn't see the message **Now in default destructor....** That's because we haven't made a call to this destructor, so our objects were destructed by the *automatic* destructor inherited from **Entity**. Change **GreetingDemo.ex** to call the *default* destructor, and then run the application.

```

-- GreetingDemo.ex v1.2

```

```

include GreetingClass.e

procedure main()
    entity myDefGreetObj, myParamGreetObj

    puts(1, "\nNow in main(), before object construction....")

    myDefGreetObj = call_method(GreetingClass, "new", NONE)
    myParamGreetObj = call_method(GreetingClass, "new", {"Hello World!"})

    puts(1, "\nNow in main(), after object construction....")

    VOID = call_method(myDefGreetObj, "delete", NONE)
    VOID = call_method(myParamGreetObj, "delete", NONE)
end procedure

main()

```

You should now see the four previous lines, followed by two lines saying:

Now in default destructor....

(... one line for each object being destructed.)

Now let's display some values, so that we can confirm that our objects are being constructed with the correct properties. Let's change **GreetingClass.e** as follows:

```

-- GreetingClass.e v1.2

include diamondlite.e

global constant GreetingClass = class("GreetingClass", Entity)
    property("message", INSTANCE, NONE)

    function GreetingClass_new_0()
        entity newGreeting

        puts(1, "\nNow in default destructor....")
        newGreeting = call_method(super(), "new", NONE)
        set_property(newGreeting, "message", NONE)
        printf(1, "\nmessage = %s", {get_property(newGreeting, "message")})

        return newGreeting
    end function
    method("new", 0, CLASS, routine_id("GreetingClass_new_0"))

    function GreetingClass_new_1(sequence msg)
        entity newGreeting

        puts(1, "\nNow in parameterised constructor....")
        newGreeting = call_method(super(), "new", NONE)
        set_property(newGreeting, "message", msg)

        -- two syntaxes for achieving the same purpose: display message = .....
        printf(1, "\nmessage = %s", {get_property(newGreeting, "message")})
        printf(1, "\nmessage = %s", {msg})

        return newGreeting
    end function

```

```

method("new", 1, CLASS, routine_id("GreetingClass_new_1"))

function GreetingClass_getMessage_0()
    sequence text

    puts(1, "\nNow in method getMessage()....")
    text = get_property(this(), "message")
    printf(1, "\nmessage = %s", {text})

    return text
end function
method("getMessage", 0, INSTANCE, routine_id("GreetingClass_getMessage_0"))

function GreetingClass_delete_0()
    puts(1, "\nNow in default destructor...")

    return call_method(super(), "delete", NONE)
end function
method("delete", 0, INSTANCE, routine_id("GreetingClass_delete_0"))
end_class()

```

Run **GreetingDemo.ex** (v1.3), and note the series of messages that confirm that the class is behaving correctly:

- ❖ We start in **main()**, in main program context, before any object is constructed.
- ❖ One object is constructed by the *default* constructor, which sets the property **message** = (ie an empty sequence).
- ❖ The second object is constructed by the *parameterised* constructor, which sets the property **message** = **Hello World!** (and displays it twice, according to the two syntaxes used in the constructor).
- ❖ Then we go to the method **getMessage()**, which displays **message** = **Hello World!**.
- ❖ And finally each object is destructed in the *default* destructor (hence two messages).

STEP 13: A GREETING CLASS WITH A PARAMETERISED DESTRUCTOR

Just as we can provide arguments with which a parameterised constructor can initialise our objects, so we can provide arguments for a parameterised destructor to decommission our object in a manner of our choosing. We don't really need this destructor for our present purposes; but just as in the section above, it gives us an opportunity to practise coding and demonstrate that the syntax works as it should. We will modify **GreetingClass.e** to include a parameterised destructor:

```

-- GreetingClass.e v1.3

include diamondlite.e

global constant GreetingClass = class("GreetingClass", Entity)
    property("message", INSTANCE, NONE)

    function GreetingClass_new_0()
        entity newGreeting

        puts(1, "\nNow in default destructor...")
        newGreeting = call_method(super(), "new", NONE)
        set_property(newGreeting, "message", NONE)
        printf(1, "\nmessage = %s", {get_property(newGreeting, "message")})

        return newGreeting
    end function
end_class()

```



```

end function
method("new", 0, CLASS, routine_id("GreetingClass_new_0"))

function GreetingClass_new_1(sequence msg)
    entity newGreeting

    puts(1, "\nNow in parameterised constructor...")
    newGreeting = call_method(super(), "new", NONE)
    set_property(newGreeting, "message", msg)
    printf(1, "\nmessage = %s", {get_property(newGreeting, "message")})
    printf(1, "\nmessage = %s", {msg})

    return newGreeting
end function
method("new", 1, CLASS, routine_id("GreetingClass_new_1"))

function GreetingClass_getMessage_0()
    sequence text

    puts(1, "\nNow in method getMessage()...")
    text = get_property(this(), "message")
    printf(1, "\nmessage = %s", {text})

    return text
end function
method("getMessage", 0, INSTANCE, routine_id("GreetingClass_getMessage_0"))

function GreetingClass_delete_0()
    puts(1, "\nNow in default destructor...")

    return call_method(super(), "delete", NONE)
end function
method("delete", 0, INSTANCE, routine_id("GreetingClass_delete_0"))

-- 9 define a (redundant) parameterised destructor; it takes one argument;
--   when called it will not override the automatic destructor,
--   because it has a different signature: delete( param )
function GreetingClass_delete_1(sequence finish)
    puts(1, "\nNow in parameterised destructor...")
    printf(1, "\ndestructing %s", {finish})

    set_property(this(), "message", "Goodbye World!")

    printf(1, "\nmessage = %s",
            {get_property(this(), "message")})

    return call_method(super(), "delete", NONE)
end function
method("delete", 1, INSTANCE, routine_id("GreetingClass_delete_1"))
end_class()

```

The syntax should be familiar by now, as we have met it all before. All that remains is for us to make the appropriate modification to **GreetingDemo.ex** to demonstrate that our second object (**myParamGreetObj**) is decommissioned appropriately:

```
-- GreetingDemo.ex v1.4
```

```

include GreetingClass.e

procedure main()
    entity myDefGreetObj, myParamGreetObj

    puts(1, "\nNow in main(), before object construction....")

    myDefGreetObj    = call_method(GreetingClass, "new", NONE)
    myParamGreetObj = call_method(GreetingClass, "new", {"Hello World!"})

    puts(1, "\nNow in main(), after object construction....")

    VOID = call_method(myDefGreetObj, "delete", NONE)
    VOID = call_method(myParamGreetObj, "delete", {"myParamGreetObj"})
end procedure

main()

```

You'll see the same messages displayed as before, verifying that each component of the class behaves as dictated by the code. However in place of the previous final message

Now in default destructor....

you will see three new messages:

**Now in parameterised destructor....
 destructing myParamGreetObj
 message = Goodbye World!**

STEP 14: A GREETING CLASS WITH A SETTER METHOD

We know how to let the constructor initialise properties of our class; we've learnt how to pass arguments to our destructor; and we know how to design a getter accessor method in case we want to access data. A lot of the code for this class hasn't been truly necessary. And we haven't really used our class to best advantage, because much of its functionality is still hard-coded in the class definition itself. Nevertheless the exercise has been useful, to give us coding practice and to demonstrate that the class really works as designed.

We have one more coding task to achieve – the design and testing of a setter method to change the property's value. We don't really need it for this particular program, but we'll do it in anticipation of a time when we'll be asking our application – rather than the class definition – to set new values for properties. As before, we'll incorporate statements that will demonstrate that the class behaves as it should. Again, we'll be getting more coding practice.

We won't need much discussion about this task – we've met all this code before. Here is the new version of **GreetingClass.e**

```

-- GreetingClass.e v1.4

include diamondlite.e

global constant GreetingClass = class("GreetingClass", Entity)
    property("message", INSTANCE, NONE)

    function GreetingClass_new_00()
        entity newGreeting

```

```

        puts(1, "\nNow in default destructor...")
        newGreeting = call_method(super(), "new", NONE)
        set_property(newGreeting, "message", NONE)
        printf(1, "\nmessage = %s", {get_property(newGreeting, "message")})

        return newGreeting
    end function
    method("new", 0, CLASS, routine_id("GreetingClass_new_0"))

function GreetingClass_new_1(sequence msg)
    entity newGreeting

    puts(1, "\nNow in parameterised constructor...")
    newGreeting = call_method(super(), "new", NONE)
    set_property(newGreeting, "message", msg)
    printf(1, "\nmessage = %s", {get_property(newGreeting, "message")})
    printf(1, "\nmessage = %s", {msg})

    return newGreeting
end function
method("new", 1, CLASS, routine_id("GreetingClass_new_1"))

-- 10 define a method to set the property's value
function GreetingClass_setMessage_1(sequence new_msg)
    puts(1, "\nNow in method setMessage()...")

    set_property(this(), "message", new_msg)

    printf(1, "\nmessage = %s", {get_property(this(), "message")})
    printf(1, "\nmessage = %s", {new_msg})

    return NIL
end function
method("setMessage", 1, INSTANCE, routine_id("GreetingClass_setMessage_1"))

function GreetingClass_getMessage_0()
    sequence text

    puts(1, "\nNow in method getMessage()...")
    text = get_property(this(), "message")
    printf(1, "\nmessage = %s", {text})

    return text
end function
method("getMessage", 0, INSTANCE, routine_id("GreetingClass_getMessage_0"))

function GreetingClass_delete_0()
    puts(1, "\nNow in default destructor...")

    return call_method(super(), "delete", NONE)
end function
method("delete", 0, INSTANCE, routine_id("GreetingClass_delete_0"))

function GreetingClass_delete_1(sequence finish)
    puts(1, "\nNow in parameterised destructor...")
    printf(1, "\ndestructing %s", {finish})

```

```

        set_property(this(), "message", "Goodbye World!")

        printf(1, "\nmessage = %s",
                {get_property(this(), "message")})

        return call_method(super(), "delete", NONE)
    end function
    method("delete", 1, INSTANCE, routine_id("GreetingClass_delete_1"))
end_class()

```

And now for the corresponding changes to our application, **GreetingDemo.ex**

```

-- GreetingDemo.ex v1.5

include GreetingClass.e

procedure main()
    entity myDefGreetObj, myParamGreetObj

    puts(1, "\nNow in main(), before object construction....")

    myDefGreetObj    = call_method(GreetingClass, "new", NONE)
    myParamGreetObj = call_method(GreetingClass, "new", {"Hello World!"})
    puts(1, "\nNow in main(), after object construction....")
    VOID = call_method(myDefGreetObj,    "setMessage", {"Greetings Earthlings!"})
    VOID = call_method(myDefGreetObj,    "delete", NONE)
    VOID = call_method(myParamGreetObj, "delete", {"myParamGreetObj"})
end procedure

main()

```

We see a longer list of messages, each demonstrating the progress of the application as it executes each component of the class:

- ❖ we begin in **main()**
- ❖ we construct two objects (whose properties are initialised to an **empty sequence** and **"Hello World!"** respectively)
- ❖ we go back to **main()**
- ❖ we change the property's value to **"Greetings Earthlings!"** (which we display twice, using two different syntaxes)
- ❖ we destruct the objects – one via the *default* destructor; the other by passing an argument to the *parameterised* destructor (which changes the value of the property to **"Goodbye World!"**).

STEP 15: THE GREETING CLASS STRIPPED DOWN

So far we've designed our class tediously, giving it redundant features and unnecessary work to do. We did this to practise building up the basic components of a class, get experience in coding in DL, and prove to ourselves that the components of the class work in the manner and order they were supposed to.

In real-world programs, however, we want the class to confine itself to providing a blueprint for properties and methods which our application's objects can acquire, leaving it up to the application to do the job the user desires.

We'll suppose that our task is to write a program to display three messages on the screen: **"Hello World!"**; **"Greetings Earthlings!"**; and **"Goodbye World!"**.

Using the *procedural* programming paradigm, we would think in terms of doing something to (in this case *displaying*) three separate pieces of data (the three messages). We could do this in our application by coding something like:

```
puts(1, "Hello World!")
puts(1, "\nGreetings Earthlings!")
puts(1, "\nGoodbye World!")
```

Using the *OOP* paradigm, however, we would think in terms of there being one piece of data (a message) that may be assigned different values, each of which may be accessed for display on the screen. We could then think of this piece of data (ie a *property*) as having three states: an *initial value* ("Hello World!"); a *reset value* ("Greetings Earthlings!"); and a *final value* ("Goodbye World!"). We would have to provide a mechanism (ie *methods*) for achieving these states – let's say a *constructor* (for the initial value); a *destructor* (for the final value); and a *setter* (for any other values). We would then have to provide a mechanism for retrieving any of these values at any time – ie a *getter* method. We would think in terms of putting all of this functionality into one class. Finally we would code an application that instantiated the class, and used the methods to access the property (in its different states), which it would display on the screen.

To show you how this might be achieved I've taken **GreetingClass.e** and stripped it right down to its bare essentials. We've met all of this code before, so I won't elaborate on it. But I've commented the steps I took, in the order I took them.

```
-- GreetingClass.e v1.5

-- 1 include the DL library
include diamondlite.e

-- 2 get a reference to the class, whose superclass is Entity
global constant GreetingClass = class("GreetingClass", Entity)
-- 4 register a property, and initialise it to an empty sequence
property("message", INSTANCE, NONE)

-- 5 define a parameterised constructor; it takes one argument
-- it does not override the inherited automatic (zero parameter) constructor
function GreetingClass_new_1(sequence msg)
    entity newGreeting

    newGreeting = call_method(super(), "new", NONE)
    set_property(newGreeting, "message", msg)

    return newGreeting
end function
method("new", 1, CLASS, routine_id("GreetingClass_new_1"))

-- 6 define a method to set the property's value
function GreetingClass_setMessage_1(sequence new_msg)
    set_property(this(), "message", new_msg)

    return NIL
end function
method("setMessage", 1, INSTANCE, routine_id("GreetingClass_setMessage_1"))

-- 7 define a method to get the property's value
function GreetingClass_getMessage_0()
    sequence text
```

```

        text = get_property(this 0, "message")

        return text
    end function
    method("getMessage", 0, INSTANCE, routine_id("GreetingClass_getMessage_0"))

    -- 8 define a default destructor to override the inherited automatic destructor
    -- it displays a final message
    function GreetingClass_delete_0 0
        set_property(this 0, "message", "Goodbye World!")

        printf(1, "\nDuring destruct, it is: %s", {get_property(this 0, "message")})

        return call_method(super0, "delete", NONE)
    end function
    method("delete", 0, INSTANCE, routine_id("GreetingClass_delete_0"))

    -- 3 end the class definition
end_class0

```

Now the corresponding application file, **GreetingDemo.ex**, becomes:

```

-- GreetingDemo.ex v1.6

-- 1 include the file with the class definition
include GreetingClass.e

-- 2 define the procedure main()
procedure main0
    -- 3 declare a variable of type entity
    entity myParamGreetObj

    -- 4 create an instance of the class using a parameterised constructor,
    -- initialise its property, and return a reference to the object
    myParamGreetObj = call_method(GreetingClass, "new", {"Hello World!"})

    -- 5 display the value of the initialised property
    printf(1, "\nInitialise greeting to: %s",
        {call_method(myParamGreetObj, "getMessage", NONE)})

    -- 6 now assign a new value to the property
    VOID = call_method(myParamGreetObj, "setMessage", {"Greetings Earthlings!"})

    -- 7 and display the new value
    printf(1, "\nAfter resetting, it is: %s",
        {call_method(myParamGreetObj, "getMessage", NONE)})

    -- 8 use a default destructor to destroy the object
    -- and display a final message
    VOID = call_method(myParamGreetObj, "delete", NONE)
end procedure

-- 9 call the procedure main()
main0

```

Now we're letting the application do most of the work. It instantiates the class with an object, initialises the property (**message**) to **"Hello World!"**, and returns a reference

(**myParamGreetObj**) to the object. It then displays the value of that property using Eu's **printf()**. Next, it resets the value of the property, and again uses **printf()** to display it ("Greetings Earthlings!"). Finally it invokes the default destructor to decommission the object, and while doing so it resets the value of the property and immediately displays it ("Goodbye World!"). Note that this message could not be displayed by our application after the object's destruction, because by that time the object and all its components have gone out of scope, and are beyond the reach of our application code.

Although our application is one step closer to the real world, it lacks one important capability – it cannot interact with the user at run-time. The next step will examine how we can achieve this task.

INTERACTING WITH THE USER

STEP 16: A PRODUCT CLASS WITH USER INPUT

We've progressed a step at a time, initially placing a lot of the executable code in the class definition, then replacing it with executable code in the application itself. Doing this has allowed the application to do more of the work. Now it's time to let the user do some of the work, interacting with the application by supplying values for some of the data required by the program.

We will exemplify this functionality by designing an application to do the following:

1. prompt the user to input two numbers
2. calculate their product
3. display the product on the screen

Using the *procedural paradigm*, we would think of the task in terms of actions to be performed on data:

1. `get_first_number`
2. `get_second_number`
3. `calculate_product`
4. `display_product`

Using the *OO paradigm*, we would think more like this:

1. we have a class
2. the class contains two properties (ie states, or pieces of data) – `num_1`; `num_2`
3. the class consists of at least four methods –
 - one to set the first number
 - one to set the second number
 - one to calculate the product
 - one to get the product

Actually, there are two more methods – a constructor and a destructor – but we'll just use DL's automatic defaults. And if we wanted to, we could add three more methods –

- ❖ one to get the first number
 - ❖ one to get the second number
 - ❖ and a third to display the product
4. we then design an application to instantiate the class, get the user's input, and accomplish the job using the tools provided by the class specification.

Using the OO paradigm for such a simple program might seem excessively tedious – but it's worth doing for practice on the way to more complex programs.

We have already seen this code before, so I won't go over it again – the comments should be clear enough. Let's begin with the class specification for **ProductClass.e**:

```
-- ProductClass.e v1.0
```

```

-- 1 include the DL library
include diamondlite.e

-- 2 define a class called Product, whose superclass is Entity,
-- and get a reference to be assigned to a global constant
global constant Product = class("Product", Entity)
    -- 3 register two properties, num1 and num2,
    -- and give each of them a default value of zero
    property("num1", INSTANCE, 0)
    property("num2", INSTANCE, 0)

    -- 4 a method to set the first number
    function Product_setFirstNum_1(integer n1)
        set_property(this(), "num1", n1)
        return NIL
    end function
    method("setFirstNum", 1, INSTANCE, routine_id("Product_setFirstNum_1"))

    -- 5 a method to set the second number
    function Product_setSecondNum_1(integer n2)
        set_property(this(), "num2", n2)
        return NIL
    end function
    method("setSecondNum", 1, INSTANCE, routine_id("Product_setSecondNum_1"))

    -- 6 a method to get the first number
    function Product_getFirstNum_0()
        return get_property(this(), "num1")
    end function
    method("getFirstNum", 0, INSTANCE, routine_id("Product_getFirstNum_0"))

    -- 7 a method to get the second number
    function Product_getSecondNum_0()
        return get_property(this(), "num2")
    end function
    method("getSecondNum", 0, INSTANCE, routine_id("Product_getSecondNum_0"))

    -- 8 a method to calculate the product
    function Product_calcProduct_0()
        atom product
        product = get_property(this(), "num1") * get_property(this(), "num2")
        return product
    end function
    method("calcProduct", 0, INSTANCE, routine_id("Product_calcProduct_0"))

    -- 9 a method to display the product
    function Product_showProduct_0()
        puts(1, "\nTheir product is: ")
        print(1, call_method(this(), "calcProduct", NONE))
        return NIL
    end function
    method("showProduct", 0, INSTANCE, routine_id("Product_showProduct_0"))

-- 10 end the class definition
end_class()

```


We've designed a class that contains the two properties, with methods to set them and get them, as well as methods to calculate and display their product. Now let's design an application **ProductDemo.ex** to instantiate and use this class:

```
-- ProductDemo.ex v1.0

-- 1 include the class definition and other libraries
include ProductClass.e
include get.e

-- 2 define procedure main()
procedure main()
    -- 3 declare variables
    sequence first, second

    -- 4 create an object of the class Product, and get a reference of type entity
    entity multiply
    multiply = call_method(Product, "new", NONE)

    -- 5 input the first integer
    puts(1, "Enter the first integer: ")    first = get(0)

    -- 6 set the first property to the value input by the user
    VOID = call_method(multiply, "setFirstNum", {first[2]})

    -- 7 input the second integer
    puts(1, "\nEnter the second integer: ")    second = get(0)

    -- 8 set the second property to the value input by the user
    VOID = call_method(multiply, "setSecondNum", {second[2]})

    -- 9 get and display the value of each property
    puts(1, "\n\nThe first integer is: ")
    print(1, call_method(multiply, "getFirstNum", NONE))

    puts(1, "\nThe second integer is: ")
    print(1, call_method(multiply, "getSecondNum", NONE))

    -- 10 display the product on the screen
    VOID = call_method(multiply, "showProduct", NONE)
end procedure

-- 11 call procedure main()
main()
```

Notice how our application has taken charge, as it were, calling method after method to get the job done –

- ❖ create an instance of the class
- ❖ input and set the first number
- ❖ input and set the second number
- ❖ get and display the first number
- ❖ get and display the second number
- ❖ display the product.

You might be wondering why our application hasn't called a method to calculate the product ("calcProduct" in our class definition file). Well it turns out that it doesn't need to, because **calcProduct** is invoked internally, in the class definition, within the routine responsible for

implementing the method **showProduct**. It would be redundant to have our application call **calcProduct** all over again.

The point to notice is this: it's a good idea for the application to do as much of the work as possible; but there may be some work that's best done quietly within the class itself. Calculating the product is an example of such a task. Why? Because the point of our **ProductClass** is to go ahead and calculate the product of two data – our application shouldn't have to call it to do the very job it was created for!

Run the application, responding to the prompts by entering integers. Notice that the application displays your entries to confirm that the properties have been set to the values you entered, and confirm that the product displayed on your screen is correct. If you're curious to "trace" the execution of your program, use **DL.e** and examine the display output against a hard copy of your class definition file and application file. The exercise may be tedious, but worth doing from time to time to help you become familiar with the flow of your programs as they go from one program context to the next.

At last, we've created an application that interacts with the user on the one hand, and with a class on the other hand. We're now ready to accomplish more ambitious tasks.

MAKING COPIES OF OBJECTS

STEP 17: COPYING OBJECTS BY ASSIGNING A REFERENCE

So far each of our applications has used only one object of its corresponding class. But since one advantage of OOP is to make it possible for us to reuse code, and since a class is often likened to a blueprint from which to create objects, there will be times when we'll want our application to create multiple instances of any given class. It's time for us to learn how to do this.

DL provides us with a method called **clone()**, with which we can create multiple objects of a class. We introduced it in **STEP 5** – it will help you to read that section again now. I will discuss how to use this method, but I want to get to it in a roundabout way, to point out some interesting things along the way.

Let's start with the basic concept behind **ProductClass**. It consisted of two properties **num1** and **num2**, which could be multiplied together to yield a variable that we called **product**. (There were supporting methods to help us set, get, calculate, and display these values, but we'll ignore them for now.) All these elements made up a *data structure* called a class.

Let's start the discussion by thinking of the three variables as a "unit", and let's consider how we might use procedural programming to produce something like a "copy" of it. Look at this sample of code in Eu:

```
-- Purpose: to make a "copy" of x
integer x, y
x = 10
y = x  -- now y is assigned the same value as x
? y    -- displays the integer 10
```

If you run this code you will confirm that **y** now has the same value as **x** (ie 10) – so we can think of **y** as a "copy" of **x**. Of course once "copied", the two variables are free to take different values. For example, continuing the above code:

```
x = 20
? x    -- now x is 20
```

```
? v      -- but v is still 10
y = 30
? y      -- now y is 30
? x      -- but x is still 20
```

Now let's take our "unit" (num1, num2, product), think of it as a sequence, and use the syntax above to create a "copy" of it. Look at **CopySequence.ex** below:

```
-- CopySequence.ex v1.0

integer num1, num2, product

num1 = 10
num2 = 20
product = (num1 * num2) -- ie 200

sequence source, copy

source = {num1, num2, product}
copy = source -- assign to copy all the elements of source

puts(1, "The source sequence is: ")      ? source -- ie {10, 20, 200}
puts(1, "The copied sequence is: ")      ? copy   -- ie {10, 20, 200}

-- now change source in some way; eg double every element
source = (source * 2)
puts(1, "The source sequence is now: ")  ? source -- ie {20, 40, 400}

-- and confirm that copy remains unchanged
puts(1, "The copied sequence is still: ") ? copy   -- ie {10, 20, 200}

-- now change copy in some way; eg halve every element
copy = (copy / 2)
puts(1, "The copied sequence is now: ")  ? copy   -- ie {5, 10, 100}

-- and confirm that source remains unchanged
puts(1, "The source sequence is still: ") ? source -- ie {20, 40, 400}
```

What have we done? We've declared a variable called **source**, and assigned to it a "unit" of data (in this case a sequence of three integers – num1, num2, product). Then we've declared another, independent variable – called **copy** – to which we've assigned the same values as **source**. We've demonstrated that each sequence could be manipulated independently, such that changing **source** wouldn't automatically change **copy** (and vice versa). So we can feel justified in thinking of **copy** as an "identical twin" of **source**.

Now let's apply this reasoning to the task of making a copy of an object of **ProductClass**. Let's go back to our application **ProductDemo.ex**, and let's create a second instance ("copy") of our class (**Product**) using the model we've already tested in **CopySequence.ex**. We'll call our new file **AssignCopyDemo.ex**:

```
-- AssignCopyDemo.ex v1.0

include ProductClass.e
include get.e

procedure main()
```

```

sequence first, second
entity source, copy

source = call_method(Product, "new", NONE)

puts(1, "Enter the first integer: ")      first = get(0)
VOID = call_method(source, "setFirstNum", {first[2]})

puts(1, "\nEnter the second integer: ")  second = get(0)
VOID = call_method(source, "setSecondNum", {second[2]})

puts(1, "\n\nSource's first integer is: ")
print(1, call_method(source, "getFirstNum", NONE))

puts(1, "\nSource's second integer is: ")
print(1, call_method(source, "getSecondNum", NONE))

VOID = call_method(source, "showProduct", NONE)

copy = source

-- get and display the value of each property of copy
puts(1, "\n\nCopy's first integer is: ")
print(1, call_method(copy, "getFirstNum", NONE))

puts(1, "\nCopy's second integer is: ")
print(1, call_method(copy, "getSecondNum", NONE))

-- display copy's product on the screen
VOID = call_method(copy, "showProduct", NONE)
end procedure

main()

```

Run **AssignCopyDemo.ex** to confirm that **copy** and **source** do indeed display the same values. Then let's do what we did in **CopySequence.ex** –

- ❖ change **source**, and confirm that **copy** remains unchanged
- ❖ then change **copy**, and confirm that **source** remains unchanged

We'll change **AssignCopyDemo.ex** as follows:

```

-- AssignCopyDemo.ex v1.1

include ProductClass.e
include get.e

procedure main()
sequence first, second
entity source, copy

source = call_method(Product, "new", NONE)

puts(1, "Enter the first integer: ")      first = get(0)
VOID = call_method(source, "setFirstNum", {first[2]})

puts(1, "\nEnter the second integer: ")  second = get(0)
VOID = call_method(source, "setSecondNum", {second[2]})

```

```

puts(1, "\n\nSource's first integer is: ")
print(1, call_method(source, "getFirstNum", NONE))

puts(1, "\n\nSource's second integer is: ")
print(1, call_method(source, "getSecondNum", NONE))

VOID = call_method(source, "showProduct", NONE)

copy = source

puts(1, "\n\nCopy's first integer is: ")
print(1, call_method(copy, "getFirstNum", NONE))

puts(1, "\n\nCopy's second integer is: ")
print(1, call_method(copy, "getSecondNum", NONE))

VOID = call_method(copy, "showProduct", NONE)

puts(1, "\n\nNow change source's properties....")
puts(1, "\nEnter the first integer: ") first = get(0)
VOID = call_method(source, "setFirstNum", {first[2]})

puts(1, "\nEnter the second integer: ") second = get(0)
VOID = call_method(source, "setSecondNum", {second[2]})

puts(1, "\n\nSource's first integer is: ")
print(1, call_method(source, "getFirstNum", NONE))

puts(1, "\n\nSource's second integer is: ")
print(1, call_method(source, "getSecondNum", NONE))

VOID = call_method(source, "showProduct", NONE)

puts(1, "\n\nNow back to copy....")
puts(1, "\n\nCopy's first integer is: ")
print(1, call_method(copy, "getFirstNum", NONE))

puts(1, "\n\nCopy's second integer is: ")
print(1, call_method(copy, "getSecondNum", NONE))

VOID = call_method(copy, "showProduct", NONE)
end procedure

main()

```

Run the application. You should find that when you input new values for **first** and **second** in source, those same values unexpectedly appear in copy! This isn't what we predicted from the model we developed in **CopySequence.ex**, so clearly something's not right. Before we get to that, however, let's see out of curiosity what'll happen to source, if we reset **first** and **second** in copy. Look at the following version of **AssignCopyDemo.ex**:

```
-- AssignCopyDemo.ex v1.2
```

```
include ProductClass.e
include get.e
```

```

procedure main()
    sequence first, second
    entity source, copy

    source = call_method(Product, "new", NONE)

    puts(1, "Enter the first integer: ")      first = get(0)
    VOID = call_method(source, "setFirstNum", {first[2]})

    puts(1, "\nEnter the second integer: ")  second = get(0)
    VOID = call_method(source, "setSecondNum", {second[2]})

    puts(1, "\n\nSource's first integer is: ")
    print(1, call_method(source, "getFirstNum", NONE))

    puts(1, "\nSource's second integer is: ")
    print(1, call_method(source, "getSecondNum", NONE))

    VOID = call_method(source, "showProduct", NONE)

    copy = source

    puts(1, "\n\nCopy's first integer is: ")
    print(1, call_method(copy, "getFirstNum", NONE))

    puts(1, "\nCopy's second integer is: ")
    print(1, call_method(copy, "getSecondNum", NONE))

    VOID = call_method(copy, "showProduct", NONE)

    puts(1, "\n\nNow change copy's properties....")

    puts(1, "\nEnter the first integer: ")      first = get(0)
    VOID = call_method(copy, "setFirstNum", {first[2]})

    puts(1, "\nEnter the second integer: ")  second = get(0)
    VOID = call_method(copy, "setSecondNum", {second[2]})

    puts(1, "\n\nCopy's first integer is: ")
    print(1, call_method(copy, "getFirstNum", NONE))

    puts(1, "\nCopy's second integer is: ")
    print(1, call_method(copy, "getSecondNum", NONE))

    VOID = call_method(copy, "showProduct", NONE)

    puts(1, "\n\nNow back to source....")
    puts(1, "\nSource's first integer is: ")
    print(1, call_method(source, "getFirstNum", NONE))

    puts(1, "\nSource's second integer is: ")
    print(1, call_method(source, "getSecondNum", NONE))

    VOID = call_method(source, "showProduct", NONE)
end procedure

main()

```

Again we find that when we change **copy**, the changes automatically appear in **source**. We're forced to conclude that while we can produce an independent "copy" of an ordinary variable by assigning to **copy** the value(s) of **source**, we can't use the same syntax to produce independent multiple instances of the same class.

This is because the identifiers we've been using for variables of type **entity** (ie the names we've been giving to our instances – "**InertEntity**", "**MySimpleObject**", "**multiply**", "**source**", "**copy**", "**MyGreetingObject**"), and even the identifiers for the global constants returned by the routine **class()** (ie "**InertClass**", "**SimpleClass**", "**GreetingClass**", "**Product**"), are really *handles*, or *references* (or an "alias") with respect to the class.

Let's make a detour to examine what this means. Interestingly, this digression will lead us back to the topic of making multiple instances of a class.

A DETOUR: DL's HANDLES

We introduced the topic of *handles* (or perhaps more formally, *references*) in **AN ORIENTATION TO DL**, and watched them in action for awhile beginning at **STEP 2**. Normally DL (like Eu) deals with them silently on our behalf, leaving us free to think about more direct programming tasks. But understanding them can help us see why **AssignCopyDemo.ex** didn't work in the way we had expected. Moreover handles can help us better understand DL's class hierarchy, and how DL keeps track of the classes and instances we create. And they can suggest another way by which we might create multiple instances – our main task, after all. So we examine them in detail now.

We can think of a *handle* as a reference that identifies (or stands for, or is an alias for) a program element, by means of which we can gain access to the program element itself. Anything we do by use of the reference, we are actually doing to the program element to which it refers. It's like having a mansion (the program element) that opens out onto (let's say) the main road (one reference), a side road (a second reference), and a back laneway (a third reference). Each entrance (reference) is different – but each gives us access to the same mansion (program element), and having gained access we can do all sorts of things to it.

In **AssignCopyDemo.ex** the identifier **source** wasn't "the object itself" – it was a reference (doorway) to it. So when we declared the identifier **copy** and assigned it the value of **source**, it was as if we were saying "Let **copy** give us access to the same object as **source**." – ie we gave our mansion two entrances, **source** and **copy**. That's why any change we made to the object by using **source**, could be displayed by using **copy** (and vice versa).

You already know what these references "look like" in DL. They're sequences of three integer elements – **{a, b, c}** – where:

a is the class number

b is the instance (ie the object's) number (for a class, this will be 0)

c is Eu's largest negative integer (a constant called **MARKER** = -1073741824)

Each reference is unique. Each element (except **MARKER**) is incremented automatically by Eu according to a predefined plan. You will recall that DL predefines three classes – **Entity** (with its three methods – **new()**, **delete()**, and **clone()**); **Exception** (with no properties or methods); and **Null_Class** (again, with no properties or methods). **Entity** is automatically inherited by each normal class we declare. **Exception** is automatically inherited by each exception we declare. We can define *subclasses* (but not properties or methods) of **Exception**; but we cannot define *subclasses* of **Null_Class**. And we can't create instances of **Exception** or **Null_Class**. DL automatically creates **Null_Instance** (an instance of **Null_Class**) for us. Have a look at **APPENDIX A** again, for a pictorial representation of this.

You will recall the following:

- ❖ **Entity**'s reference is {1, 0, **MARKER**}
- ❖ **Exception**'s reference is {2, 0, **MARKER**}
- ❖ **Null_Class**' reference is {3, 0, **MARKER**}
- ❖ programmer's classes are referenced as {4, 0, ...}, {5, 0, ...}, etc
- ❖ **Null_Instance**' reference is {3, 1, **MARKER**}
- ❖ the reference for any class' **delete()** method is also {3, 1, **MARKER**}
- ❖ **new()** and **clone()** methods are referenced by consecutive integers

To help make sense of this, have a look at the following program **HandleNums.ex**. The class definition and application code are all in the one file, which is amply commented. Run the application and check out all the numbers!

```
-- HandleNums.ex

include diamondlite.e

entity newEntity

puts(1, "\nHandle to Entity      = ")          print(1, Entity)      -- {1, 0, MARKER}

puts(1, "\nHandle to Entity.new()  = ")
newEntity = call_method(Entity, "new", NONE)  print(1, newEntity) -- {1, 2, MARKER}

puts(1, "\nHandle to Entity.clone() = ")
print(1, call_method(newEntity, "clone", NONE)) -- {1, 3, MARKER}

puts(1, "\nHandle to Entity.delete() = ")
print(1, call_method(newEntity, "delete", NONE)) -- {3, 1, MARKER}

puts(1, "\n\nHandle to Exception      = ")          print(1, Exception) -- {2, 0, MARKER}

puts(1, "\n\nHandle to Null_Class      = ")          print(1, Null_Class) -- {3, 0, MARKER}

puts(1, "\nHandle to Null_Instance  = ")          print(1, Null_Instance) -- {3, 1, MARKER}

global constant MyClass = class("MyClass", Entity)
    puts(1, "\n\nHandle to MyClass      = ")
    print(1, MyClass) -- {4, 0, MARKER}
end_class()

-----

entity myOb_1, myOb_2, myOb_3

myOb_1 = call_method(MyClass, "new", NONE)
puts(1, "\n\nHandle to myOb_1.new()  = ")          print(1, myOb_1) -- {4, 3, MARKER}

puts(1, "\nHandle to myOb_1.clone() = ")
print(1, call_method(myOb_1, "clone", NONE)) -- {4, 4, MARKER}

puts(1, "\nHandle to myOb_1.delete() = ")
print(1, call_method(myOb_1, "delete", NONE)) -- {3, 1, MARKER}

myOb_2 = call_method(MyClass, "new", NONE)
puts(1, "\n\nHandle to myOb_2.new()  = ")          print(1, myOb_2) -- {4, 5, MARKER}
```



```

puts(1, "\nHandle to myOb_2.clone() = ")
print(1, call_method(myOb_2, "clone", NONE))           -- {4, 6, MARKER}

puts(1, "\nHandle to myOb_2.delete() = ")
print(1, call_method(myOb_2, "delete", NONE))         -- {3, 1, MARKER}

myOb_3 = call_method(MyClass, "new", NONE)
puts(1, "\n\nHandle to myOb_3.new() = ")               print(1, myOb_3) -- {4, 7, MARKER}

puts(1, "\nHandle to myOb_3.clone() = ")
print(1, call_method(myOb_3, "clone", NONE))         -- {4, 8, MARKER}

puts(1, "\nHandle to myOb_3.delete() = ")
print(1, call_method(myOb_3, "delete", NONE))         -- {3, 1, MARKER}

```

As I said before, this detour into handles/references can also lead us back to our main topic. Look at the application portion of the file, and notice the declaration:

```
entity myOb_1, myOb_2, myOb_3
```

We now know that each of these is a *reference to a class*. Now look at what is assigned to each reference. It's the value returned by invoking `call_method()`. And what's the target of each of these calls? It's **myClass**. And what's that? A reference to the class we defined earlier in the file, when we invoked `class()`. Could this syntax illustrate another way of creating multiple instances of a class?

BACK TO MAKING COPIES OF OBJECTS

STEP 17a: COPYING OBJECTS BY DECLARING NEW INSTANCES OF A CLASS

Our previous attempt to create multiple instances of a class failed because we didn't know about references at the time. But while learning about them, we may have hit upon a solution. The solution appears to be this: declare multiple identifiers of type **entity**, and assign to each of them a unique reference returned from `call_method()` using as target (a reference to) the class we wish to instantiate (and which we would have already defined in our class file).

Let's explore this idea by writing an application **NewCopyDemo.ex** using **ProductClass**.

```

-- NewCopyDemo.ex v1.0

include ProductClass.e

procedure main()
    entity source, copy

    source = call_method(Product, "new", NONE)
    copy   = call_method(Product, "new", NONE)

    -- get and display the value of each property of source
    puts(1, "Source's details....")
    puts(1, "\nSource's first integer is: ")
    print(1, call_method(source, "getFirstNum", NONE))

    puts(1, "\nSource's second integer is: ")
    print(1, call_method(source, "getSecondNum", NONE))

```

```

-- display source's product on the screen
VOID = call_method(source, "showProduct", NONE)

-- get and display the value of each property of copy
puts(1, "\n\nCopy's details....")
puts(1, "\nCopy's first integer is: ")
print(1, call_method(copy, "getFirstNum", NONE))

puts(1, "\nCopy's second integer is: ")
print(1, call_method(copy, "getSecondNum", NONE))

-- display copy's product on the screen
VOID = call_method(copy, "showProduct", NONE)
end procedure

main()

```

Running this application doesn't produce startling results – each property, and therefore the product, is **0** (the default values). But at least both **source** and **copy** are the same. Let's test the code a bit more by setting the properties of **source** and seeing whether they are reflected in **copy**.

```

-- NewCopyDemo.ex v1.1

include ProductClass.e
include get.e

procedure main()
sequence first, second
entity source, copy

source = call_method(Product, "new", NONE)

puts(1, "Enter the first integer: ") first = get(0)
VOID = call_method(source, "setFirstNum", {first[2]})

puts(1, "\nEnter the second integer: ") second = get(0)
VOID = call_method(source, "setSecondNum", {second[2]})

puts(1, "Source's details....")
puts(1, "\nSource's first integer is: ")
print(1, call_method(source, "getFirstNum", NONE))

puts(1, "\nSource's second integer is: ")
print(1, call_method(source, "getSecondNum", NONE))

VOID = call_method(source, "showProduct", NONE)

-- now create a new instance ("copy") of the class
copy = call_method(Product, "new", NONE)

puts(1, "\n\nCopy's details....")
puts(1, "\nCopy's first integer is: ")
print(1, call_method(copy, "getFirstNum", NONE))

puts(1, "\nCopy's second integer is: ")

```

```

    print(1, call_method(copy, "getSecondNum", NONE))

    VOID = call_method(copy, "showProduct", NONE)
end procedure

main()

```

When you run this application you will see that irrespective of what you do to **source**, you can't get **copy** to change from the default values. A little thought reveals why – **copy** is the reference returned from the routine **call_method()**, whose target is the method **new()**, the default constructor. In other words using this syntax, we will succeed in creating a new instance of the class – but only in its default, initialised state. This isn't the kind of copy we had in mind. It's like seeing a house that you really like, and asking the builder to construct one just like it for you – only to find that (s)he builds a house based on the original plans, without the extensions and renovations that were subsequently added to the house you liked so much. To accomplish that job, we'll have to turn to the method **clone()**.

STEP 17b: COPYING OBJECTS USING THE METHOD **clone()**

Our task is to create multiple instances without having to use the (constructor) method **new()** (which as we've seen, can only create instances with their default property values). We do this by using the method **clone()**, which is inherited from the universal base class **Entity**, by each class we design. This method will make and return a copy of each property of its target class, giving us a new instance with the same property values. We get the job done by using the routine **call_method()**, and passing to the method **clone()** and its target, the name of the class.

To see how this is done let's create a new application **CloneCopyDemo.ex**, based on our previous file **NewCopyDemo.ex**, and note the additional syntax:

```

-- CloneCopyDemo.ex v1.0

include ProductClass.e
include get.e

procedure main()
    sequence first, second
    entity source, copy

    source = call_method(Product, "new", NONE)

    puts(1, "Enter the first integer: ")    first = get(0)
    VOID = call_method(source, "setFirstNum", {first[2]})

    puts(1, "\nEnter the second integer: ")    second = get(0)
    VOID = call_method(source, "setSecondNum", {second[2]})

    puts(1, "Source's details...")
    puts(1, "\nSource's first integer is: ")
    print(1, call_method(source, "getFirstNum", NONE))

    puts(1, "\nSource's second integer is: ")
    print(1, call_method(source, "getSecondNum", NONE))

    VOID = call_method(source, "showProduct", NONE)

    -- delete this – it doesn't work!

```

```

        -- copy = call_method(Product, "new", NONE)

        -- invoke clone() on source to make copy
        copy = call_method(source, "clone", NONE)

        puts(1, "\n\nCopy's details....")
        puts(1, "\nCopy's first integer is: ")
        print(1, call_method(copy, "getFirstNum", NONE))

        puts(1, "\nCopy's second integer is: ")
        print(1, call_method(copy, "getSecondNum", NONE))

        VOID = call_method(copy, "showProduct", NONE)
    end procedure

main()

```

When you run this application you should find that the values you gave to **source**'s properties are reflected in the values of **copy**'s properties. So far, so good. Let's now change **source**, and see whether **copy** remains unchanged. Here's the next version of **CloneCopyDemo.ex**:

```

-- CloneCopyDemo.ex v1.1

include ProductClass.e
include get.e

procedure main()
    sequence first, second
    entity source, copy

    source = call_method(Product, "new", NONE)

    puts(1, "Enter the first integer: ")    first = get(0)
    VOID = call_method(source, "setFirstNum", {first[2]})

    puts(1, "\nEnter the second integer: ")    second = get(0)
    VOID = call_method(source, "setSecondNum", {second[2]})

    puts(1, "Source's details....")
    puts(1, "\nSource's first integer is: ")
    print(1, call_method(source, "getFirstNum", NONE))

    puts(1, "\nSource's second integer is: ")
    print(1, call_method(source, "getSecondNum", NONE))

    VOID = call_method(source, "showProduct", NONE)

    copy = call_method(source, "clone", NONE)

    -- reset source's property values:
    VOID = call_method(source, "setFirstNum", {10})
    VOID = call_method(source, "setSecondNum", {20})

    -- get and display the new value of each property of source
    puts(1, "\n\nSource's new details....")
    puts(1, "\nSource's first integer is: ")
    print(1, call_method(source, "getFirstNum", NONE))

```

```

puts(1, "\nSource's second integer is: ")
print(1, call_method(source, "getSecondNum", NONE))

-- display source's product on the screen
VOID = call_method(source, "showProduct", NONE)

puts(1, "\n\nCopy's details....")
puts(1, "\nCopy's first integer is: ")
print(1, call_method(copy, "getFirstNum", NONE))

puts(1, "\nCopy's second integer is: ")
print(1, call_method(copy, "getSecondNum", NONE))

VOID = call_method(copy, "showProduct", NONE)
end procedure

main()

```

Run this application. You should find that this time **copy**'s values remain unchanged whatever we do to **source**'s values. The final test is to determine whether we can change **copy**'s values without those changes being reflected automatically in **source**. Let's make the following changes to **CloneCopyDemo.ex**:

```

-- CloneCopyDemo.ex v1.2

include ProductClass.e
include get.e

procedure main()
sequence first, second
entity source, copy

source = call_method(Product, "new", NONE)

puts(1, "Enter the first integer: ") first = get(0)
VOID = call_method(source, "setFirstNum", {first[2]})

puts(1, "\nEnter the second integer: ") second = get(0)
VOID = call_method(source, "setSecondNum", {second[2]})

puts(1, "Source's details....")
puts(1, "\nSource's first integer is: ")
print(1, call_method(source, "getFirstNum", NONE))

puts(1, "\nSource's second integer is: ")
print(1, call_method(source, "getSecondNum", NONE))

VOID = call_method(source, "showProduct", NONE)

copy = call_method(source, "clone", NONE)

-- get and display the value of each property of copy
puts(1, "\n\nCopy's details....")
puts(1, "\nCopy's first integer is: ")
print(1, call_method(copy, "getFirstNum", NONE))

```

```

puts(1, "\nCopy's second integer is: ")
print(1, call_method(copy, "getSecondNum", NONE))

VOID = call_method(copy, "showProduct", NONE)

-- reset copy's property values:
VOID = call_method(copy, "setFirstNum", {10})
VOID = call_method(copy, "setSecondNum", {20})

-- get and display the new value of each property of copy
puts(1, "\n\nCopy's new details...")
puts(1, "\nCopy's first integer is: ")
print(1, call_method(copy, "getFirstNum", NONE))

puts(1, "\nCopy's second integer is: ")
print(1, call_method(copy, "getSecondNum", NONE))

-- display source's product on the screen
VOID = call_method(copy, "showProduct", NONE)

-- now check the value of each property of source
puts(1, "\n\nSource's details...")
puts(1, "\nSource's first integer is: ")
print(1, call_method(source, "getFirstNum", NONE))

puts(1, "\nSource's second integer is: ")
print(1, call_method(source, "getSecondNum", NONE))

-- display source's product on the screen
VOID = call_method(source, "showProduct", NONE)
end procedure

main()

```

When you run this application you'll be able to confirm that we've finally succeeded in creating a copy of our original object – a true copy that is free to vary independently of the source object from which it was created.

A RECAP AND A LOOK AHEAD....

We have discussed several important concepts here, which we can summarise as follows:

1. the identifiers we use for *entities* are actually *handles* or references, implemented as three-element sequences of integers; they are generated automatically by DL (according to a predetermined plan), and are the means by which the entities are located and used.
2. by using the modified version of **diamondlite.e** we have been able to see an *interaction* between our application, our class definition, and DL – backwards and forwards – gaining access to DL's routines, our own methods, and our own properties as the need arises. This *dynamic* view of program execution complements the static, *diagrammatic* view of classes and objects as "having" or "containing" things.
3. we have been able to appreciate the importance of *program context* – our application begins in *main program context* up to the point where the routine **class()** executes; we then enter *class definition context*, where properties and methods are registered and the class is set up; after **end_class()** executes we go back to *main program context*, from which point the application will enter *instance/class method context* as the program dictates.
4. we noted that the method **new()** actually creates a brand new entity in its *default* state, with properties set to their initial values, and methods poised to execute when called.

5. we noted that the method `clone()` makes a (bitwise) copy of an entity in its *present* state, with properties set at their current value; the source and the copy are then free to take different paths during program execution.

We are now able to write an outline for a generic class – `GenericClass.e`

```
include diamondlite.e

global constant Generic = class("Generic", Entity)
  property("aNumber", INSTANCE, NIL)
  property("aString", INSTANCE, NONE)
  property("anEntity", INSTANCE, Null_Instance)

  function Generic_new_0()
    entity newGeneric
    newGeneric = call_method(super(), "new", NONE)
    return newGeneric
  end function
  method("new", 0, CLASS, NULL_METHOD)

  function Generic_clone_0()
    entity cloneGeneric
    cloneGeneric = call_method(super(), "clone", NONE)
    return cloneGeneric
  end function
  method("clone", 0, INSTANCE, NULL_METHOD)

  function Generic_setProperty_1(object x)
    set_property(this(), <property_name>, x)
    return NIL
  end function
  method("setProperty", 1, INSTANCE, NULL_METHOD)

  function Generic_getProperty_0()
    return get_property(this(), <property_name>)
  end function
  method("getProperty", 0, INSTANCE, NULL_METHOD)
end_class()
```

Its corresponding application could look something like this – `GenericDemo.ex`

```
include GenericClass.e

procedure main()
  entity myGeneric, myClone

  myGeneric = call_method(Generic, "new", NONE)
  VOID = call_method(myGeneric, "setProperty", {<property_value>})
  myClone = call_method(myGeneric, "clone", NONE)
  -- invoke call_method(myClone, "getProperty", NONE) to return a property_value
  -- display or otherwise use this value
end procedure

main()
```

Most of this should look familiar by now. Notice the syntax **NULL_METHOD** – it's another place-holder, that will eventually be occupied by a valid **routine_id()** when a proper and functioning program is coded. As it stands now, it represents a method that does nothing and returns **NIL**.

You may be puzzled by the property whose value is **Null_Instance**. We haven't come across this before – it means that this property is designed to contain an instance (rather than an Eu fundamental data type). A class containing an entity? We explore this functionality in the next sections.

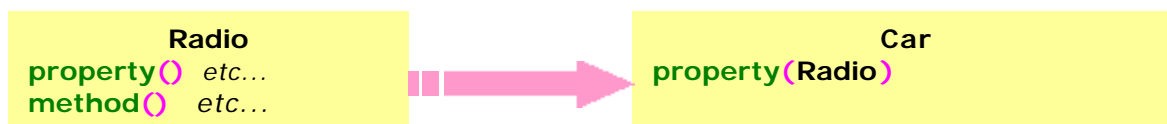
A QUICK LOOK AT COMPOSITION

So far our classes' properties have been one or other of Eu's fundamental data types. But we are now looking at a class with a property that is itself an entity, ie a unit that has certain qualities and capabilities. This might surprise you at first, but it makes sense when you think about it - many real-life objects are *composed* of (or "contain") other objects "within" them, that are meant to work together, but that can also exist and function independently. My car has a radio, which is designed to function seamlessly within the car – the radio and the car could still function independently of one another, but the car benefits from having the added functionality provided by the radio. Looked at like this, we could say that the radio is an attribute (a "feature") of the car, even though the radio could potentially function without the car (eg on a boat, instead).

How should we code such a situation? Your intuition might be to simply place the radio inside the car, like this:

```
global constant Car = class("Car", Entity)
    global constant Radio = class("Radio", Entity)
    end_class()
end_class()
```

That is not legal syntax, and for an important reason: you don't just want to dump the radio on the back seat of the car – you want to put it in its correct place, wire it up, and make it an integral, functioning part of the car. That's why we have to make the radio a *property* of the car. We can picture our task as below. How might we achieve it?



In real life we would:

1. make a new radio, and have it ready
2. make a new car, with space for a radio
3. set the radio in the space in the car

We could achieve this result using the following point-form algorithm:

1. design the blueprint for a radio
2. design the blueprint for a car
3. allocate space for a radio
4. specify the construction of a new car
5. specify the construction of a new radio
6. set the new radio in its allocated space

We could code this process in three separate files. First, **Radio.e**:


```

-- Radio.e v1.0

-- 1  design the radio blueprint
global constant Radio = class("Radio", Entity) -- {4,0,M}
    -- to keep things simple, nothing in here!
    -- our class will use Entity's methods new(), clone(), and delete()
end_class()

```

Then **Car.e**....

```

-- Car.e v1.0

-- 2  design the car blueprint
global constant Car = class("Car", Entity) -- {5,0,M}
    -- 3  allocate space for a radio
    property("aRadio", INSTANCE, Null_Instance)

    function Car_new_0()
        entity newCar, newRadio

        -- 4  specify the construction of a new car
        newCar = call_method(super(), "new", NONE) -- {5,2,M}

        -- 5  specify the construction of a new radio
        newRadio = call_method(Radio, "new", NONE) -- {4,3,M}

        -- 6  set the new radio in its allocated space
        set_property(newCar, "aRadio", newRadio)

        return newCar -- {5,2,M}
    end function
    method("new", 0, CLASS, routine_id("Car_new_0"))
end_class()

```

And finally an application to create the finished product – **CarRadioDemo.ex**:

```

-- CarRadioDemo.ex v1.0

-- don't forget these include files!
include diamondlite.e
include Radio.e
include Car.e

procedure main()
    entity myCar
    myCar = call_method(Car, "new", NONE) -- {5,2,M}
end procedure

main()

```

Run the application with **DL.e**. We've met all this syntax in previous steps, but we haven't used it in quite this way before. Compare the class definition files with the point-form algorithm to ensure you understand the steps. Notice the detailed code in the constructor **Car.new()** – it really does create the new entity (two, in this case), and handles the assignment to the property. Also notice the screen display - particularly the handles:

1. for class **Radio**: **{4,0,M}**
2. for class **Car**: **{5,0,M}**
3. for **Null_Instance**: **{3,1,M}** -- you don't see this – it is implied!
4. for **newCar** (ie myCar): **{5,2,M}** -- an instance of Car, therefore handle = **{5,....}**
5. for **newRadio**: **{4,3,M}** -- an instance of Radio, therefore handle = **{4,....}**

Being able to create a data structure that represents an entity with a property that is itself an entity, will enable us to simulate more complex real-world objects – even objects that we see on our computer screen. We explore this idea in the next steps.

DEEP AND SHALLOW CLONING

Now that we can model the creation of more complex entities, we are in a position to examine how to copy them. In **STEP 17b** we considered how to use the method **clone()** and we found that it produced a bitwise copy of an entity as it was at the moment of cloning – the copy had all the property values of the source, and the same capabilities.

In the next few steps we will be exploring several ways of creating and copying entities, illustrating them with simplistic simulations of GUI designs. I am indebted to Michael Nelson for suggesting this approach to the topic, and for some of the code I use below.

The objective is not to learn how to create GUI's, but rather to use the *idea* of GUI's to illustrate some of the things we can achieve with cloning. A secondary objective is to give us practice in OO thinking, and in coding using DL. I'll present the material incrementally, but here is an overview of the simulations we'll consider:

1. create a single window – **Window_1**
2. clone a window to produce two identical entities – **Window_1** and **Window_1¹**
3. create a single window (**Window_1**) with a single button (**Button_1**) in it
4. create two windows, each with a button in it, using different approaches:
 - create two new windows (**Window_1**; **Window_2**) and two new buttons (**Button_1**; **Button_2**)
 - create one new window with one new button (**Window_1**; **Button_1**), and clone them – **Window_1¹**; **Button_1¹**
 - create two new windows (**Window_1**; **Window_2**), a new button (**Button_1**), and its (shallow) clone (**Button_1¹**)
 - create two new windows (**Window_1**; **Window_2**), a new button (**Button_1**), and its (deep) clone (**Button_2¹**)
5. create three windows, each with a button in it, using different approaches:
 - **Window_1** with **Button_1**
 - **Window_2** with **Button_1¹** (a shallow clone of **Button_1**)
 - **Window_3** with **Button_2¹** (a deep clone of **Button_1**)
6. create one window, with two buttons in it, using different approaches:
 - create a new button (**Button_1**) and clone it (**Button_1¹**)
 - create a new button, clone it twice, and use the clones (**Button_1¹**; **Button_1²**)
 - create a new button, clone it once, and use it twice (**Button_1¹**; **Button_1¹**)

The details will become clear as our discussion unfolds.

STEP 18: CLONING A WINDOW

Let's begin by imagining that we wish to model a blank window which we eventually intend to copy. We can think of the window as an entity, with properties and capabilities. To keep things simple, let's give it one property (an id number) and three basic capabilities (of being opened, of being copied, and of being closed).

We can describe this functionality using the point-form algorithm below:

1. design the blueprint for a window
2. allocate space for its id
3. create a new window
4. give it its own id
5. clone this window
6. decommission both windows

We can now specify a class definition to achieve this functionality – **Window.e**

(NOTE: I've supplied handle numbers here and there, to help you keep track of the process.)

```
-- Window.e v1.0

-- 1 design the window blueprint
global constant Window = class("Window", Entity) -- {4,0,M}
-- 2 allocate space for its id
property("idNum", INSTANCE, NIL)

-- * something new here: a class property, "counter". It is discussed below.
property("counter", CLASS, 0)

function Window_new_0()
    entity newWindow
    integer id

    -- 3 create a new window
    newWindow = call_method(super(), "new", NONE)

    -- 4 give it its own id; update the class counter
    id = get_property(this(), "counter") + 1 -- this() returns {4,0,M}
    set_property(newWindow, "idNum", id)
    set_property(this(), "counter", id) -- this() returns {4,0,M}

    return newWindow
end function
method("new", 0, CLASS, routine_id("Window_new_0"))

-- 5 clone this window using Entity's clone() method
-- 6 delete this window using Entity's delete() method

-- 7 a method to allow us to access a window's id number
function Window_getID_0()
    integer id
    id = get_property(this(), "idNum")
    return id
end function
method("getID", 0, INSTANCE, routine_id("Window_getID_0"))
end_class()
```

We have introduced a *class property* ("counter") for the first time. This is a property that pertains to the class as a whole – it is not automatically available to instances of the class. In this particular case it's there to keep count of the number of instances that have been created; it is incremented each time the method **new()** is invoked. Notice that its default value is **NIL** – 0 – to which it returns each time the application is rerun.

We have met the *instance property* ("idNum", in this case) before. Every instance of Window will have this property, whose default value is **NIL** – to which it returns each time we create a

new entity. Accordingly each time we invoke method `new()` we have to set "idNum" with the current value of "counter", or else we would only know its default value.

Notice that our *default constructor* creates a new entity and takes responsibility for incrementing the class counter, and setting the new entity's "idNum". We didn't create a separate method for this because in DL all methods are *public*, and we don't want an application to be able to meddle with our object's id.

And notice the use of the routine `this()` within the class method `new()`. As we've seen before, `this()` returns a reference to the current entity – from within an *instance method*, it returns a reference to the current *instance entity*; but from within a *class method*, it returns a reference to the current *class entity* (in this case, the class Window). So we could have written:

`get_property(Window, "counter")` and `set_property(Window, "counter", id)`
in place of: `get_property(this(), "counter")` and `set_property(this(), "counter", id)`

Now let's go about creating and displaying two identical windows on the screen:



We have two choices – we can create a new window, clone it, and display them both; or we can create two new windows and display them. Let's create an application to simulate (very simplistically!) these options – **WindowDemo.ex**:

```
-- WindowDemo.ex v1.0

include diamondlite.e
include Window.e

procedure main()
    entity Window1, Window2

    -- create two new windows, and show their ID numbers
    Window1 = call_method(Window, "new", NONE) -- {4,2,M}
    Window2 = call_method(Window, "new", NONE) -- {4,3,M}
    puts (1, "\nTwo brand new windows:")
    printf(1, "\nWindow1.id = %d", call_method(Window1, "getID", NONE))
    printf(1, "\nWindow2.id = %d", call_method(Window2, "getID", NONE))

    -- create a new window and clone it; show the ID numbers
    -- use Window1 as our new window – code 'reuse'
    Window2 = call_method(Window1, "clone", NONE) -- {4,4,M}
    puts (1, "\n\nNew and cloned window:")
    printf(1, "\nWindow1.id = %d", call_method(Window1, "getID", NONE))
    printf(1, "\nWindow2.id = %d", call_method(Window2, "getID", NONE))
end procedure

main()
```

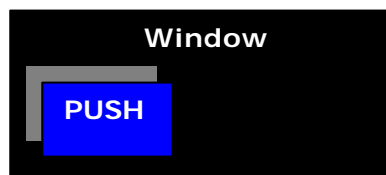
If you run the application with `diamondlite.e` you will see:

Two brand new windows:	-- BTW - HERE ARE THE ENTITY HANDLES:
Window1.id = 1	-- Window1 is {4,2,M}
Window2.id = 2	-- Window2 is {4,3,M}
 New and cloned window:	
Window1.id = 1	-- Window1 is {4,2,M}
Window2.id = 1	-- Window2 (ie clone of Window1) is {4,4,M}

It confirms that whereas **new()** creates two *different* windows (hence different id's) with their default values, **clone()** creates a second window that is a *copy* of the first (with property values equal to those at the moment of copying). We could go on like this, creating as many copies of our window as we need. (If you're not sure of the flow of execution, run the application with **DL.e** and follow the handle numbers.)

STEP 19: A WINDOW WITH A BUTTON

Now let's go a step further, and imagine that we are looking at a screen with a window that has a button on it; the button has the word "PUSH" on it.



We can think of the button as an entity, with properties (eg text, colour, size etc) and capabilities. This situation is similar to the one we met before – a car with a radio – so we should be able to use similar syntax.

The point form algorithm for achieving this functionality would then be something like this:

1. design the blueprint for a button
2. allocate space for text; initialise it to an empty string
3. construct a new button
4. set its text to the string "PUSH"
5. design the blueprint for a window
6. allocate space for an entity of class button
7. create a new window
8. create a new button
9. assign the new button to window's property "button"

We will begin by coding the class definition for the button – **Button.e**:

```
-- Button.e v1.0

-- 1 design the button blueprint
global constant Button = class("Button", Entity) -- {4,0,M}
-- 2 allocate space for text; initialise it to an empty string
property("text", INSTANCE, NONE)

function Button_new_0()
    entity newButton

    -- 3 construct a new button
    newButton = call_method(super(), "new", NONE) -- {4,3,M}

    -- 4 set its text to the string "PUSH"
    set_property(newButton, "text", "PUSH")
```

```

        return newButton
    end function
    method("new", 0, CLASS, routine_id("Button_new_0"))

    function Button_getText_0()
        return get_property(this(), "text")
    end function
    method("getText", 0, INSTANCE, routine_id("Button_getText_0"))
end_class()

```

We could have taken a different approach. We could have initialised the property "text" to "PUSH" - like this: `property("text", INSTANCE, "PUSH")` - and then used the automatic default constructor instead of coding our own default constructor. We have also supplied a method `getText()` to make it possible for us to retrieve the value in property "text".

And now we'll code the class definition for the window – **Window.e**

```

-- Window.e v1.1

-- 5 design the window blueprint
global constant Window = class("Window", Entity) -- {5,0,M}
    property("counter", CLASS, NIL)
    property("idNum", INSTANCE, 0)

    -- 6 allocate space for an entity of class Button
    property("button", INSTANCE, Null_Instance)

    -- allocate space for the text in window's button
    property("buttonText", INSTANCE, NONE)

    function Window_new_0()
        entity newWindow, newButton
        integer id
        sequence itsText -- the button's text

        -- 7 create a new window
        newWindow = call_method(super(), "new", NONE) -- {5,2,M}
        id = get_property(this(), "counter") + 1 -- this() returns {5,0,M}
        set_property(newWindow, "idNum", id)
        set_property(this(), "counter", id) -- this() returns {5,0,M}

        -- 8 call Button.new() to create a new button with text "PUSH"
        newButton = call_method(Button, "new", NONE) -- {4,3,M}

        -- 9 assign the new button to window's property "button"
        set_property(newWindow, "button", newButton)

        -- call newButton.getText() to return the button's text
        itsText = call_method(newButton, "getText", NONE)

        -- assign that text to window's property
        set_property(newWindow, "buttonText", itsText)

        return newWindow
    end function
    method("new", 0, CLASS, routine_id("Window_new_0"))

```

```

-- to clone the window use the automatic default clone() method
-- to delete the window use the automatic default delete() method

-- a method to allow us to access a window's id number
function Window_getID_0()
    integer id
    id = get_property(this(), "idNum")
    return id
end function
method("getID", 0, INSTANCE, routine_id("Window_getID_0"))

-- a method to allow us to access the handle of window's button
function Window_getButton_0()
    return get_property(this(), "button")
end function
method("getButton", 0, INSTANCE, routine_id("Window_getButton_0"))

-- a method to allow us to access the text of window's button
function Window_getText_0()
    return get_property(this(), "buttonText")
end function
method("getText", 0, INSTANCE, routine_id("Window_getText_0"))
end_class()

```

Notice that we've coded a default constructor to do the job of creating entities for us, like this:

1. create a new window
2. increment the class counter
3. assign a value to the window's ID number
4. create a new button using **Button.new()** - which we coded previously in **Button.e**
5. assign the (handle of the) new button to the new window's "button" property
6. use the new button's **getText()** method to return the button's text (ie "PUSH")
7. assign that text to the new window's property "buttonText"
8. return a reference to the new window

We have also added a method **getText()** to make it possible for us to retrieve the text of the button on the window. And of course we still have the method **getID()** to give us access to the window's id number.

And finally we code the application file – **WindowDemo.ex**:

```

-- WindowDemo.ex v1.1

include diamondlite.e
include Button.e
include Window.e

procedure main()
    entity myWindow

    -- create a new window, show its ID number, its button's handle,
    -- and its button's text
    myWindow = call_method(Window, "new", NONE) -- {5,2,M}

    puts (1, "\nA new window:")
    printf(1, "\nmyWindow.id = %d", call_method(myWindow, "getID", NONE))

```

```

puts(1, "\nmyWindow.getButton() returns the button's handle: ")
print(1, call_method(myWindow, "getButton", NONE)) -- {4,3,M}

puts(1, "\nmyWindow.getText() returns the button's text: " &
      call_method(myWindow, "getText", NONE))

end procedure

main()

```

Run the application with **diamondlite.e**, and note the screen display:

```

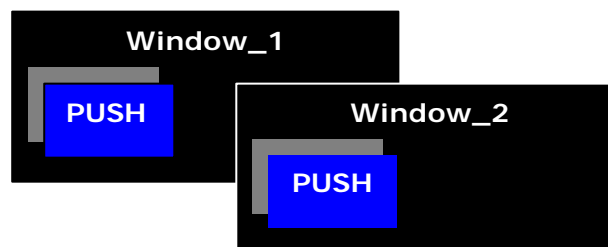
A new window:      -- this is myWindow, whose handle is {5,2,M}
myWindow.id = 1
myWindow.getButton() returns the button's handle: {4,3,M}
myWindow.getText() returns the button's text: PUSH

```

It confirms that we have created a new window; that it contains a (reference to a) button; and that the button says "PUSH".

STEP 20: TWO WINDOWS, EACH WITH AN IDENTICAL BUTTON - ALL NEW

Now let's go a step further and simulate a situation in which we have two windows on our screen, each with an identical button - like this:



There are a couple of ways of approaching this task, depending on what we want to achieve.

For a start we can create a new entity Window_1 with its corresponding button, and then we can create another new entity - Window_2 - with its own (new) button. This would be acceptable if we were prepared to accept the creation of new entities in their default state. This solution would only require a small change to the application - **WindowDemo.ex**:

```

-- WindowDemo.ex v1.2

include diamondlite.e
include Button.e
include Window.e

procedure main()
  entity myWindow1, myWindow2

  -- create a new window, show its ID number, its button's handle,
  -- and its button's text
  myWindow1 = call_method(Window, "new", NONE) -- {5,2,M}

  puts (1, "\nWindow1:")
  printf(1, "\nmyWindow1.id = %d", call_method(myWindow1,"getID", NONE))

  puts(1, "\nmyWindow1.getButton() returns the button's handle: ")
  print(1, call_method(myWindow1, "getButton", NONE)) -- {4,3,M}

```



```

puts(1, "\nmyWindow1.getText() returns the button's text: " &
      call_method(myWindow1, "getText", NONE))

-- create a second window, show its ID number, its button's handle,
-- and its button's text
myWindow2 = call_method(Window, "new", NONE) -- {5,4,M}

puts (1, "\n\nWindow2:")
printf(1, "\nmyWindow2.id = %d", call_method(myWindow2,"getID", NONE))

puts(1, "\nmyWindow2.getButton() returns the button's handle: ")
print(1, call_method(myWindow2, "getButton", NONE)) -- {4,5,M}

puts(1, "\nmyWindow2.getText() returns the button's text: " &
      call_method(myWindow2, "getText", NONE))

end procedure

main()

```

If we run the application we will be able to confirm that both windows are new entities (they have different id's), and that both buttons are new entities (they have different handles); the output is presented below:

```

Window1:    -- this is myWindow1, whose handle is {5,2,M}
myWindow1.id = 1
myWindow1.getButton() returns the button's handle: {4,3,M}
myWindow1.getText()   returns the button's text: PUSH

Window2:    -- this is myWindow2, whose handle is {5,4,M}
myWindow2.id = 2
myWindow2.getButton() returns the button's handle: {4,5,M}
myWindow2.getText()   returns the button's text: PUSH

```

STEP 20a: TWO WINDOWS, EACH WITH AN IDENTICAL BUTTON - SHALLOW CLONE

Another solution might be to start with a new window/button combination, and clone everything. We would then have two identical windows with two identical buttons. We can achieve this result using the automatic default constructor already available to the class **Window**. We would need to modify our application as follows:

```

-- WindowDemo.ex v1.3

include diamondlite.e
include Button.e
include Window.e

procedure main()
  entity myWindow, clonedWindow

  myWindow = call_method(Window, "new", NONE) -- {5,2,M}

  puts (1, "\nA new window:")
  printf(1, "\nmyWindow.id = %d", call_method(myWindow,"getID", NONE))

  puts(1, "\nmyWindow.getButton() returns the button's handle: ")

```

```

print(1, call_method(myWindow, "getButton", NONE)) -- {4,3,M}

puts(1, "\nmyWindow.getText() returns the button's text: " &
      call_method(myWindow, "getText", NONE))

-- create a cloned window, show its ID number, its button's handle,
-- and its button's text
clonedWindow = call_method(myWindow, "clone", NONE) -- {5,4,M}

puts (1, "\n\nA cloned window:")
printf(1, "\nclonedWindow.id = %d", call_method(clonedWindow, "getID", NONE))

puts(1, "\nclonedWindow.getButton() returns the button's handle: ")
print(1, call_method(clonedWindow, "getButton", NONE)) -- {4,3,M}

puts(1, "\nclonedWindow.getText() returns the button's text: " &
      call_method(clonedWindow, "getText", NONE))

end procedure

main()

```

If we now run the application we can confirm that the windows are copies (they have the same id), as are the buttons (they have the same handle):

```

A new window:      -- this is myWindow, whose handle is {5,2,M}
myWindow.id = 1
myWindow.getButton() returns the button's handle: {4,3,M}
myWindow.getText()   returns the button's text: PUSH

A cloned window:  -- this is clonedWindow, whose handle is {5,4,M}
clonedWindow.id = 1
clonedWindow.getButton() returns the button's handle: {4,3,M}
clonedWindow.getText() returns the button's text: PUSH

```

In some situations this would be exactly what we need. The second window's button is a "shallow" copy of the first window's button - it is really a reference to the first window's button, rather than a true, "stand-alone" copy in its own right. When we need to create a copy that is an entity in its own right, we need to produce what is known as a "deep" clone.

STEP 20b: TWO WINDOWS, EACH WITH AN IDENTICAL BUTTON - DEEP CLONE

We saw that we could create a clone of a window containing a reference to a clone of a button that itself resides on the first window. We achieved this as follows:

1. we created a new entity called **Window_1**, using **Entity.new()**
2. we created **Button_1** ("PUSH") using **Button.new()**, and got its handle
3. using this handle, we assigned **Button_1** to **Window_1**'s "button" property
4. using **Button_1**'s method **getText()**, we retrieved its text "PUSH"
5. we assigned that string to **Window_1**'s "buttonText" property
6. using **Window_1.clone()** we created **Window_2**, and found that it was a bitwise copy of **Window_1** - containing the same id, the same reference to the button, and the same text, as in **Window_1**

Now suppose that we want **Window_2** to have its "own" copy of the original button, rather than merely a handle to **Window_1**'s cloned button. We achieve this by designing a clone method in class **Window**, to override the class' automatic default **clone()** method. We can describe the necessary steps using the following point-form algorithm:

1. create a copy of a **Window** entity

2. get the value in its property "button" - this value will be a reference to a button entity
3. use this reference to create a clone of the button
4. set the value of this button entity to Window's "button" property
5. get the button's text
6. set this value to Window's "buttonText" property

We can code this process as follows - **Window.e**:

```
-- Window.e v1.2

global constant Window = class("Window", Entity) -- {5,0,M}
  property("counter", CLASS, 0)
  property("idNum", INSTANCE, NIL)
  property("button", INSTANCE, Null_Instance)
  property("buttonText", INSTANCE, NONE)

  function Window_new_0()
    entity newWindow, newButton
    integer id
    sequence itsText -- the button's text

    newWindow = call_method(super(), "new", NONE) -- {5,2,M}

    id = get_property(this(), "counter") + 1
    set_property(newWindow, "idNum", id)
    set_property(this(), "counter", id)
    newButton = call_method(Button, "new", NONE) -- {4,3,M}
    set_property(newWindow, "button", newButton)
    itsText = call_method(newButton, "getText", NONE)
    set_property(newWindow, "buttonText", itsText)

    return newWindow
  end function
  method("new", 0, CLASS, routine_id("Window_new_0"))

  function Window_clone_0()
    entity clonedWindow, refButton, clonedButton
    sequence itsText

    -- use Entity.clone() to create a cloned window;
    -- it will contain a button with "PUSH"
    clonedWindow = call_method(super(), "clone", NONE) -- {5,4,M}

    -- get the value in the cloned window's "button" property
    -- it will be a reference to a button entity
    refButton = get_property(clonedWindow, "button") -- {4,3,M}

    -- use the clone() method of the button entity in the cloned window's
    -- "button" property to create a new cloned button
    clonedButton = call_method(refButton, "clone", NONE) -- {4,5,M}

    -- assign the new cloned button to the cloned window's "button" property
    set_property(clonedWindow, "button", clonedButton)

    -- use the new cloned button's getText() method to return its text
    itsText = call_method(clonedButton, "getText", NONE)
```

```

        -- set that text to the cloned window's "buttonText" property
        set_property(clonedWindow, "buttonText", itsText)

        return clonedWindow
    end function
    method("clone", 0, INSTANCE, routine_id("Window_clone_0"))

    function Window_getID_0()
        integer id
        id = get_property(this(), "idNum")
        return id
    end function
    method("getID", 0, INSTANCE, routine_id("Window_getID_0"))

    function Window_getButton_0()
        return get_property(this(), "button")
    end function
    method("getButton", 0, INSTANCE, routine_id("Window_getButton_0"))

    function Window_getText_0()
        return get_property(this(), "buttonText")
    end function
    method("getText", 0, INSTANCE, routine_id("Window_getText_0"))
end_class()

```

Now when we run **WindowDemo.ex** (call it **v1.4**) we get the following result:

```

A new window:    -- this is myWindow, whose handle is {5,2,M}
myWindow.id = 1
myWindow.getButton() returns the button's handle: {4,3,M}
myWindow.getText() returns the button's text: PUSH

A cloned window: -- this is clonedWindow, whose handle is {5,4,M}
clonedWindow.id = 1
clonedWindow.getButton() returns the button's handle: {4,5,M}
clonedWindow.getText() returns the button's text: PUSH

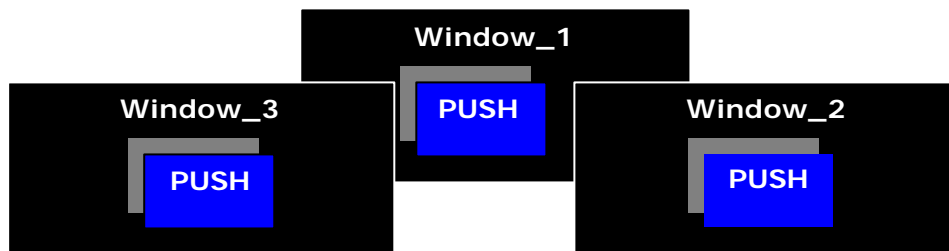
```

By following the handle numbers we can confirm that the second window is a copy of the first (it has the same id), but that it has its own copy of a button - not just a reference to the first window's button. (If you are unsure about this, run the application with **DL.e**, and trace the appearance of new handle numbers.) This is the essence of "deep" copying - it makes it possible for us to "go back", "deep" into the original entity, and create a cloned entity of that.

AN INTRODUCTION TO METHOD OVERLOADING

STEP 20c: DEEP AND SHALLOW CLONING TOGETHER

We can now look at how we might change our code in order to give us maximum flexibility - to do shallow cloning, or to do deep cloning as and when we want to. For instance, we might want to achieve the following result:



And we might want to stipulate that the button in Window_2 will be a shallow clone of the button in Window_1, whereas the button in Window_3 will be a deep clone of the button in Window_1.

We can achieve this functionality by using DL's support for method *overriding* and method *overloading*.

We are familiar with method overriding - where one method supplants another method of the same name and parameter-list. For example the `new()` and `clone()` methods that we have written, are used in place of the corresponding `new()` and `clone()` methods inherited from **Entity**. Our own methods have the same name and parameter-list as the automatically inherited methods.

Method overloading refers to a situation in which our file contains several methods with the *same name*, but which differ in either or both of the following ways:

1. one is a class method while the other is an instance method
2. the methods have a different number of parameters

We are therefore able to write code in which methods that do a similar job are more easily recognised by being given the same name, even if they use different parameters, in different contexts.

So we can modify our file `Window.e` such that it contains two `clone()` methods - one with no parameters (this one will override the corresponding method inherited from **Entity**), and an overloaded method with one parameter (with which to do deep cloning).

```
-- Window.e v1.3

global constant Window = class("Window", Entity) -- {5,0,M}
  property("counter", CLASS, 0)
  property("idNum", INSTANCE, NIL)
  property("button", INSTANCE, Null_Instance)
  property("buttonText", INSTANCE, NONE)

  function Window_new_0()
    entity newWindow, newButton
    integer id
    sequence itsText -- the button's text

    newWindow = call_method(super(), "new", NONE) -- {5,2,M}

    id = get_property(this(), "counter") + 1
    set_property(newWindow, "idNum", id)
    set_property(this(), "counter", id)

    newButton = call_method(Button, "new", NONE) -- {4,3,M}
    set_property(newWindow, "button", newButton)
```

```

        itsText = call_method(newButton, "getText", NONE)
        set_property(newWindow, "buttonText", itsText)

    return newWindow
end function
method("new", 0, CLASS, routine_id("Window_new_0"))

-- this method overrides the one inherited from Entity;
-- it is redundant here, but is included for illustration purposes
function Window_clone_0()
    entity clonedWindow
    clonedWindow = call_method(super(), "clone", NONE)
    return clonedWindow
end function
method("clone", 0, INSTANCE, routine_id("Window_clone_0"))

-- an overloaded method clone(sequence deep); it differs from clone() above
function Window_clone_1(object deep)
    entity clonedWindow, refButton, clonedButton
    sequence itsText

    VOID = deep -- to discard "deep"

    clonedWindow = call_method(super(), "clone", NONE) -- {5,4,M}
    refButton = get_property(clonedWindow, "button") -- {4,3,M}
    clonedButton = call_method(refButton, "clone", NONE) -- {4,5,M}
    itsText = call_method(clonedButton, "getText", NONE)

    set_property(clonedWindow, "button", clonedButton)
    set_property(clonedWindow, "buttonText", itsText)

    return clonedWindow
end function
method("clone", 1, INSTANCE, routine_id("Window_clone_1"))

function Window_getID_0()
    integer id
    id = get_property(this(), "idNum")
    return id
end function
method("getID", 0, INSTANCE, routine_id("Window_getID_0"))

function Window_getButton_0()
    return get_property(this(), "button")
end function
method("getButton", 0, INSTANCE, routine_id("Window_getButton_0"))

function Window_getText_0()
    return get_property(this(), "buttonText")
end function
method("getText", 0, INSTANCE, routine_id("Window_getText_0"))
end_class()

```

We can change **WindowDemo.ex** accordingly, to mediate the new functionality:

```
-- WindowDemo.ex v1.5
```

```

include diamondlite.e
include Button.e
include Window.e

procedure main()
    entity myWindow, clonedWindow, deepClonedWindow

    myWindow = call_method(Window, "new", NONE) -- {5,2,M}

    -- Window_1
    puts (1, "\nA new window:")
    printf(1, "\nmyWindow.id = %d", call_method(myWindow, "getID", NONE))

    puts(1, "\nmyWindow.getButton() returns the button's handle: ")
    print(1, call_method(myWindow, "getButton", NONE)) -- {4,3,M}

    puts(1, "\nmyWindow.getText() returns the button's text: " &
           call_method(myWindow, "getText", NONE))

    -- Window_2
    clonedWindow = call_method(myWindow, "clone", NONE) -- {5,4,M}

    puts (1, "\n\nA cloned window:")
    printf(1, "\nclonedWindow.id = %d", call_method(clonedWindow, "getID", NONE))

    puts(1, "\nclonedWindow.getButton() returns the button's handle: ")
    print(1, call_method(clonedWindow, "getButton", NONE)) -- {4,3,M}

    puts(1, "\nclonedWindow.getText() returns the button's text: " &
           call_method(clonedWindow, "getText", NONE))

    -- Window_3
    deepClonedWindow = call_method(myWindow, "clone", {"deep"}) -- {5,4,M}

    puts (1, "\n\nAnother cloned window:")
    printf(1, "\ndeepClonedWindow.id = %d",
           call_method(deepClonedWindow, "getID", NONE))

    puts(1, "\ndeepClonedWindow.getButton() returns the button's handle: ")
    print(1, call_method(deepClonedWindow, "getButton", NONE)) -- {4,3,M}

    puts(1, "\ndeepClonedWindow.getText() returns the button's text: " &
           call_method(deepClonedWindow, "getText", NONE))

end procedure

main()

```

When we run this application we see the following screen display:

```

A new window:
myWindow.id = 1
myWindow.getButton() returns the button's handle: {4,3,M}
myWindow.getText() returns the button's text: PUSH

A cloned window:
clonedWindow.id = 1
clonedWindow.getButton() returns the button's handle: {4,3,M}

```

```
clonedWindow.getText()    returns the button's text: PUSH
```

Another cloned window:

```
deepClonedWindow.id = 1
```

```
deepClonedWindow.getButton() returns the button's handle: { 4,6,M}
```

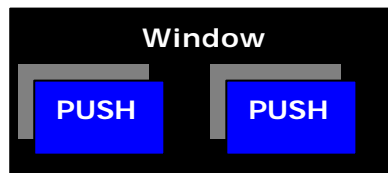
```
deepClonedWindow.getText()  returns the button's text: PUSH
```

We can confirm that all three windows are clones – they have the same id (**1**). We see that the button in the second window is merely a shallow clone of the button in the first window – ie it has the same reference (**{ 4,3,M}**) as the first button. We see that the button in the third window comes from the same class (**Button – { 4,0,M}**) as the other two buttons, but that it is a different entity – ie it has a different reference value. Finally, we observe that all three buttons have the same text – **PUSH**.

SOME EXTENSION EXERCISES

STEP 21: A WINDOW WITH TWO IDENTICAL BUTTONS

Now let's say that we want to have two identical buttons (same text, same colour etc) on the same window - like this:



Our task is to simulate this situation (very simplistically). Since we have all the code necessary to create a new window with a new button bearing the text "PUSH", it would seem that the best thing to do is simply to clone the button we already have. (We could have decided to create a new button, but we would have ended up with a button in its default state, which is not what we need.)

We could achieve this functionality using the following algorithm:

1. design the button prototype
2. allocate space for text; initialise it to an empty string
3. construct a new button
4. set its text to the string "PUSH"
5. design the window prototype
6. allocate space for an entity of class button
7. create a new window
8. create a new button
9. assign the new button to window's property "button"
10. make a copy of the button we already have

We will have to alter **Window.e** accordingly. For one thing the property "button" will have to be changed to something like "allButtons", to accommodate a sequence of entities (two buttons in this case). And the property "buttonText" could be changed to "allTexts", to accommodate a sequence of text strings (in this case, {"PUSH", "PUSH"}) corresponding to each button.

```
-- Window.e v1.4
```

```
-- 5 design the window prototype
```

```
global constant Window = class("Window", Entity) -- {5,0,M}
```

```
    property("counter", CLASS, NIL) -- the class counter
```



```

property("idNum", INSTANCE, NIL) -- a place for window's ID number
property("allButtons", INSTANCE, NONE) -- a place for all button entities
property("allTexts", INSTANCE, NONE) -- a place for all buttons' texts

function Window_new_0()
    entity newWindow, newButton, cloneButton
    integer id
    sequence itsText, -- the button's text
              buttons, -- to accommodate all buttons in the property
              texts    -- to accommodate all texts in the property

    buttons = {} texts = {} -- initialise the sequences

    newWindow = call_method(super(), "new", NONE) -- {5,2,M}
    -- give it its own id; update the class counter
    id = get_property(this(), "counter") + 1 -- this() returns {5,0,M}
    set_property(newWindow, "idNum", id)
    set_property(this(), "counter", id) -- this() returns {5,0,M}

    -- populate the sequences; there's nothing in there yet!
    buttons = get_property(newWindow, "allButtons")
    texts    = get_property(newWindow, "allTexts")

    newButton = call_method(Button, "new", NONE) -- {4,3,M}; new entity
    buttons = append(buttons, newButton) -- grow the sequence of buttons
    set_property(newWindow, "allButtons", buttons) -- assign to property
    itsText = call_method(newButton, "getText", NONE) -- get button's text
    texts = append(texts, itsText) -- grow the sequence of texts
    set_property(newWindow, "allTexts", texts) -- assign to property

    -- 10 make a copy of the button we already have
    cloneButton = call_method(newButton, "clone", NONE) -- {4,4,M}; clone
    buttons = append(buttons, cloneButton) -- grow the sequence of buttons
    set_property(newWindow, "allButtons", buttons) -- assign to property
    itsText = call_method(cloneButton, "getText", NONE) -- get button's text
    texts = append(texts, itsText) -- grow the sequence of texts
    set_property(newWindow, "allTexts", texts) -- assign to property

    return newWindow
end function
method("new", 0, CLASS, routine_id("Window_new_0"))

function Window_getID_0()
    integer id
    id = get_property(this(), "idNum")
    return id
end function
method("getID", 0, INSTANCE, routine_id("Window_getID_0"))

function Window_getText_0()
    return get_property(this(), "allTexts") -- note the change here!
end function
method("getText", 0, INSTANCE, routine_id("Window_getText_0"))
end_class()

```

We will have to modify **WindowDemo.ex** a bit to demonstrate what we've achieved:

```

-- WindowDemo.ex v1.6

include diamondlite.e
include Button.e
include Window.e

procedure main()
    entity myWindow
        sequence texts      texts = {}
        -- create a new window, show its ID number, and show its buttons' texts
        myWindow = call_method(Window, "new", NONE) -- {5,2,M}
        texts = call_method(myWindow, "getText", NONE)
        puts (1, "\nA new window:")
        printf(1, "\nmyWindow.id = %d", call_method(myWindow, "getID", NONE))
        printf(1, "\nNew button's text is %s", {texts[1]})
        printf(1, "\nCloned button's text is %s", {texts[2]})
    end procedure

main()

```

Run the application with **diamondlite.e** and confirm that we've simulated the existence of a single window, with two identical buttons bearing the word "PUSH".

A COUPLE OF ALTERNATIVES

As often happens in programming there are several different ways of accomplishing the same task. You were probably dissatisfied with **Window.e** - eg some code was repeated in the body of method **new()**; and you had to be aware of two entities (a new button and a cloned button) instead of one entity "twice".

What we did was:

1. create a new button (with its initialised text)
2. assign to the corresponding property...
 - the (new) button
 - its text
3. create a cloned button (with its initialised text)
4. assign to the corresponding property...
 - the (clone) button
 - its text

STEP 21a: START WITH A NEW BUTTON, AND CLONE IT TWICE

We could improve the code by doing this:

1. create a new button (with its initialised text)
2. clone it twice, on each occasion assigning to the corresponding property...
 - the (cloned) button
 - its text

Here is a stripped-down version of **Window.e** to illustrate this process:

```

-- Window.e v1.5

global constant Window = class("Window", Entity) -- {5,0,M}
    property("allButtons", INSTANCE, NONE)
    property("allTexts", INSTANCE, NONE)

```

```

function Window_new_0()
    entity newWindow, newButton, cloneButton
    sequence itsText, buttons, texts

    buttons = {} texts = {} -- initialise the sequences

    newWindow = call_method(super(), "new", NONE)      -- {5,2,M}

    buttons = get_property(newWindow, "allButtons")
    texts = get_property(newWindow, "allTexts")

    newButton = call_method(Button, "new", NONE)      -- {4,3,M}
    for i = 1 to 2 do
        -- the handles for these buttons will be {4,4,M} and {4,5,M}
        cloneButton = call_method(newButton, "clone", NONE)
        buttons = append(buttons, cloneButton)
        set_property(newWindow, "allButtons", buttons)
        itsText = call_method(cloneButton, "getText", NONE)
        texts = append(texts, itsText)
        set_property(newWindow, "allTexts", texts)
    end for
    return newWindow
end function
method("new", 0, CLASS, routine_id("Window_new_0"))

function Window_getText_0()
    return get_property(this(), "allTexts")
end function
method("getText", 0, INSTANCE, routine_id("Window_getText_0"))
end_class()

```

STEP 21b: START WITH A NEW BUTTON, CLONE IT ONCE, AND ASSIGN IT TWICE

Another solution might go like this:

1. create a new button (with its initialised text)
2. clone it once
3. assign it, and its text, to the corresponding property twice

Here is another stripped-down version of **Window.e** to illustrate this process:

```

-- Window.e v1.6

global constant Window = class("Window", Entity) -- {5,0,M}
    property("allButtons", INSTANCE, NONE)
    property("allTexts", INSTANCE, NONE)

    function Window_new_0()
        entity newWindow, newButton, cloneButton
        sequence itsText, buttons, texts

        buttons = {} texts = {} -- initialise the sequences

        newWindow = call_method(super(), "new", NONE)      -- {5,2,M}

        buttons = get_property(newWindow, "allButtons")
        texts = get_property(newWindow, "allTexts")
    
```

```

newButton = call_method(Button, "new", NONE) -- {4,3,M}
cloneButton = call_method(newButton, "clone", NONE) -- {4,4,M} only!

for i = 1 to 2 do
    buttons = append(buttons, cloneButton)
    set_property(newWindow, "allButtons", buttons)
    itsText = call_method(cloneButton, "getText", NONE)
    texts = append(texts, itsText)
    set_property(newWindow, "allTexts", texts)
end for
return newWindow
end function
method("new", 0, CLASS, routine_id("Window_new_0"))

function Window_getText_0()
    return get_property(this(), "allTexts")
end function
method("getText", 0, INSTANCE, routine_id("Window_getText_0"))
end_class()

```

Here is a stripped-down version of **WindowDemo.ex** to demonstrate how they work:

```

-- WindowDemo.ex v1.7 & v1.8

include diamondlite.e
include Button.e
include Window.e

procedure main()
    entity myWindow
        sequence texts      texts = {}
    myWindow = call_method(Window, "new", NONE)
    texts = call_method(myWindow, "getText", NONE)
    puts (1, "\nA new window:")
    printf(1, "\nOne button's text is %s", {texts[1]})
    printf(1, "\nThe other button's text is %s", {texts[2]})
end procedure

main()

```

On each occasion the screen will display:

```

A new window:
One button's text is PUSH
The other button's text is PUSH

```

These two class definitions certainly reduce the amount of repetitious code; and they are easier to read and understand. Ultimately all three approaches appear to produce the same result. But if you look at the handle numbers I've added along the way (or if you run the applications with **DL.e**), you will realise that we have created different entities. It turns out that these three alternatives are not necessarily interchangeable.

In **STEP 21** we created a new button entity in its default state (this became our first button), and then cloned it (to give us our second button). In some situations this might be exactly what we need – two separate entities (an "original" and its "copy") representing the default state, that are initially identical in their properties and methods, but that may differ later on.

In **STEP 21a** we created a new button entity, and then cloned it twice – assigning each clone to a corresponding button. Again, we ended up with two separate entities; but each is now a "copy" of an original button entity in its default state.

In **STEP 21b** we created a new button entity, cloned it once, then assigned it twice. We still had only one entity (the clone) – it's just that it has been used twice. Sometimes this will be enough for our needs.

And this discussion will be enough for our needs!

EXCEPTION HANDLING

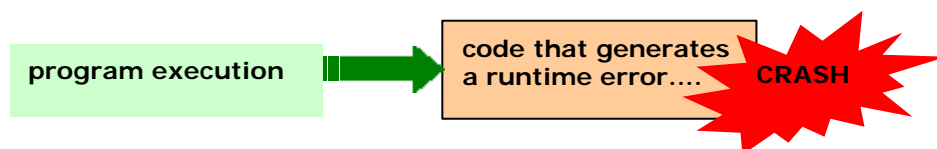
So far we've avoided dealing with errors that might have occurred during the execution of our applications. For instance when we asked for user input, we didn't check that it was of the correct type - we just trusted that it would be.

But in real-world programming we cannot make any such assumptions, so we will need to anticipate *run-time errors* - known as **exceptions** (because they represent the exceptional scenario; the exception to the rule) - and we will need a way of dealing with them logically and systematically.

Some run-time errors are *fatal* - unrecoverable - and nothing can be done to rescue execution; the program must stop. But sometimes we can prevent a situation becoming irreversible by empowering our application to issue a warning, or by forcing the user to take corrective action. There is a range of deliberate decisions we can make about what to do in such circumstances. Let's consider them one at a time.

STEP 22: DO NOTHING - LET THE LANGUAGE DEAL WITH IT!

Consider the example of trying to divide by zero. We might decide to do nothing other than to let the language deal with the problem in whatever way it can. We can picture the situation like this:



For example:

```
-- DivideByZero.ex v1.0

include get.e

procedure main()
  sequence input
  integer   numerator, denominator
  atom      answer

  puts(1,"Enter numerator: ")    input = get(0) -- enter the number 5
  numerator = input[2]

  puts(1,"\nEnter denominator: ") input = get(0) -- enter the number 0
  denominator = input[2]
```

```

        answer = numerator / denominator

        printf(1, "\nAnswer = %f", answer)
end procedure

main()

```

If we run this application (with the numbers suggested above) Eu will terminate it abruptly, and we will see something like this:

```

attempt to divide by 0
—> see ex.err

```

We can make the display a little more appealing by doing this:

```

-- DivideByZero.ex v1.1

include get.e
include machine.e

crash_message("\nAttempt to divide by zero is not allowed.")

procedure main()
    sequence input
    integer    numerator, denominator
    atom      answer

    puts(1, "Enter numerator: ")    input = get(0) -- enter the number 5
    numerator = input[2]
    puts(1, "\nEnter denominator: ") input = get(0) -- enter the number 0
    denominator = input[2]
    answer = numerator / denominator
    printf(1, "\nAnswer = %f", answer)
end procedure

main()

```

The output will be **Attempt to divide by zero is not allowed.** - but this is not really an advance on what we did before.

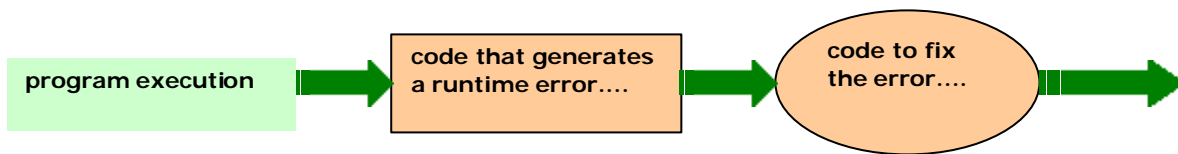
STEP 22a: RETURN AN ERROR CODE.

Notice that I've passed up the opportunity to check for an input error: I haven't tested the error-status value represented by the first element of the sequence named **input**, to verify that the user didn't do something cheeky - eg enter **zero**.

Returning error codes from functions is an established method of dealing with errors, but it has its limitations: there is only so much information that the error-code (usually an integer) can contain; it can be hard to remember the meaning of each error-code (was it 1? -1? or 0?), and how to tell the difference between the error and its alternative; and of course ultimately, the language can't force the programmer to test the error-code.

STEP 22b: COMBINE ERROR-HANDLING CODE WITH THE NORMAL CASE.

As an alternative, we could test the input and inform the user about the error, like this:



For example:

-- DivideByZero.ex v1.2

include get.e

procedure main()

sequence input

integer numerator, denominator

atom answer

puts(1,"Enter numerator: ") input = get(0) -- enter the number 5
numerator = input[2]

puts(1,"\nEnter denominator: ") input = get(0)
denominator = input[2]

if denominator = 0 then

puts(1,"\nError - denominator cannot be zero!")

else

answer = numerator / denominator

printf(1,"\nAnswer = %f", answer)

end if

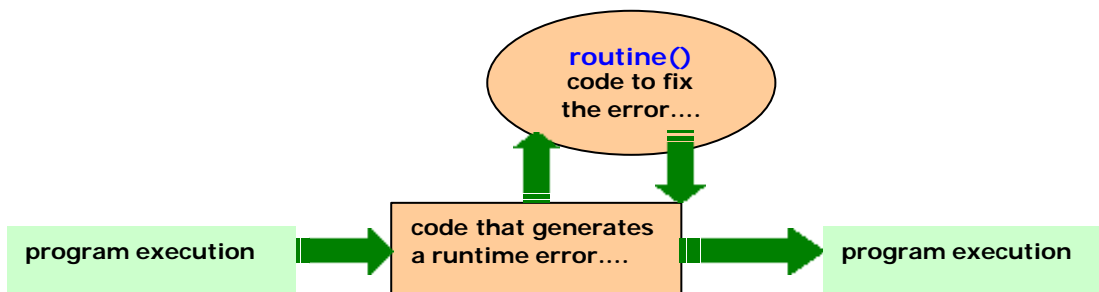
end procedure

main()

This approach does the job of informing the user about the error, but since the error-testing code is mixed with the "normal-execution" code, the program can become hard to read, understand, and update – not here, but in real, complex applications.

STEP 22c: BUNDLE ERROR-HANDLING CODE INTO SEPARATE ROUTINES

Going one step further, we could make our application more readable by bundling the error-handling code into a routine of its own - for instance:



To give an example:

-- DivideByZero.ex v1.3

include get.e

```

function nonzero_denominator(object d)
    -- error-handling code: we force user to enter non-zero denominator
    return nonzero_d
end function

procedure main()
    sequence input
    integer    numerator, denominator, nonzero
    atom       answer

    puts(1,"Enter numerator: ")      input = get(0)  -- enter the number 5
    numerator = input[2]

    puts(1,"\nEnter denominator: ")  input = get(0)
    denominator = input[2]
    nonzero = nonzero_denominator(denominator)

    answer = numerator / nonzero

    printf(1,"\nAnswer = %f", answer)
end procedure

main()

```

We come to appreciate this manner of coding when we must do a number of error-checks - eg:

```

-- DivideByZero.ex v1.4

include get.e

function integer_input(object input)
    -- error-handling code: we force user to enter an integer
    return intgr_inpt
end function

function nonzero_denominator(object d)
    -- error-handling code: we force user to enter non-zero denominator
    return nonzero_d
end function

procedure main()
    sequence input
    integer    numerator, denominator, int_input, nonzero
    atom       answer

    puts(1,"Enter numerator: ")      input = get(0)  -- enter the number 5
    numerator = input[2]
    int_input = integer_input(numerator)

    puts(1,"\nEnter denominator: ")  input = get(0)
    denominator = input[2]
    int_input = integer_input(denominator)
    nonzero = nonzero_denominator(int_input)

    answer = int_input / nonzero

```



```

    printf(1, "\nAnswer = %f", answer)
end procedure

main()

```

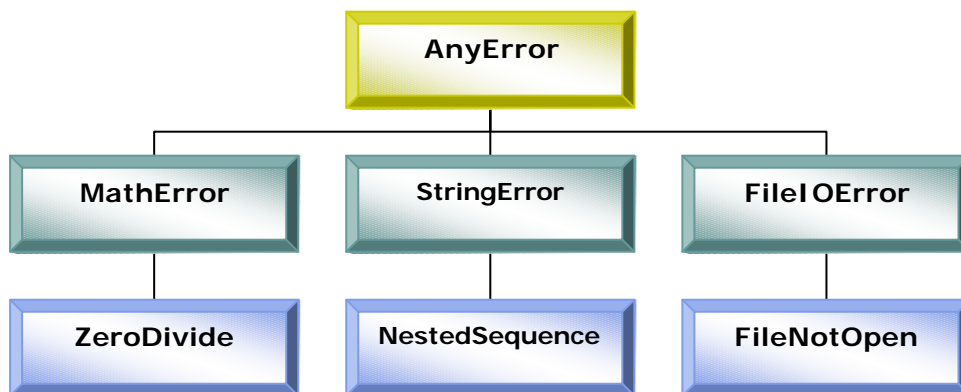
By choosing our identifiers carefully, we can make our application code clearer, more readable, and more systematic. But this method has its limits too: it doesn't enforce any kind of system by which to categorise or organise errors of different types. For example notice that both of our error-handling routines are *mathematical errors* (as distinct from, say, *file i/o errors* or *string errors*). In a large project we wouldn't be able to guarantee that the routines containing error-handling code for different types of errors, were systematically and logically organised.

An OO *exception-handling system* is designed to address these limitations, and to provide a system for dealing with a variety of errors. To understand how, we need to take a detour....

A DETOUR: AN INTRODUCTION TO INHERITANCE

Let's begin by remembering that a **class** is a complex programming element. We've described it as a "model" of external objects, or as a "blueprint" from which entities may arise, or as a "data structure" defining relationships between properties and methods; and we've seen how it can "contain" (or *incorporate*) other classes within it.

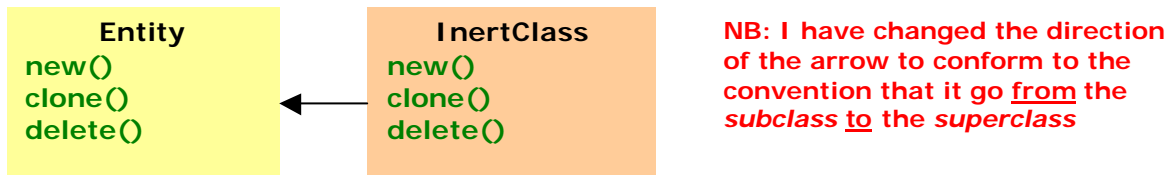
But a class is also a way of classifying objects - of saying that certain entities belong to this or that category by virtue of having certain characteristics in common. This quality of classes will prove useful in our attempt to categorise our errors. Consider the following diagram:



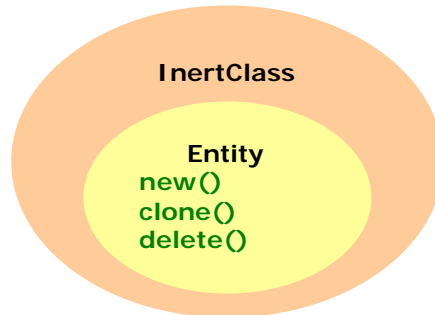
Each box represents a class. If we take any one class and follow the lines radiating from it, we trace a *hierarchy*. If we take class **MathError**, for example, we see that it has class **AnyError** above it and class **ZeroDivide** below it. With respect to **MathError** we can say the following:

- **AnyError** is its *superclass* or *parent class* (in fact it's a *base class* for all errors)
- **ZeroDivide** is its *subclass* or *child class*
- **MathError** *derives from* (or *extends*) **AnyError**
- **ZeroDivide** *derives from* (or *extends*) **MathError**
- **MathError** *inherits* all the properties and methods of **AnyError** and adds some of its own
- **ZeroDivide** *inherits* all the properties & methods of **MathError** and adds some of its own

As it happens, we have been using *inheritance* already - as early as **STEP 7**, when we created **InertClass** from **Entity**. Recall the diagram we used then:



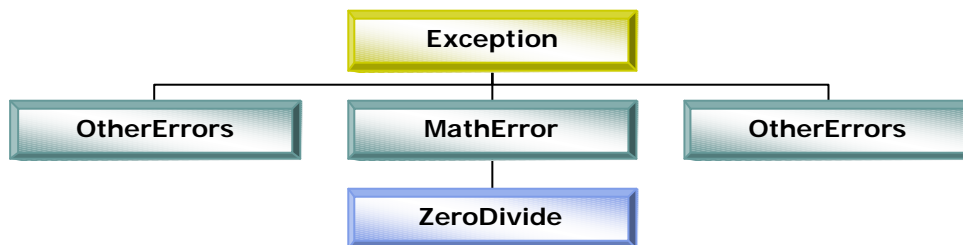
This was a pictorial way of showing that **InertClass** derived all of the functionality of **Entity** (ie three methods; no properties). Another way of representing the relationship is like this:



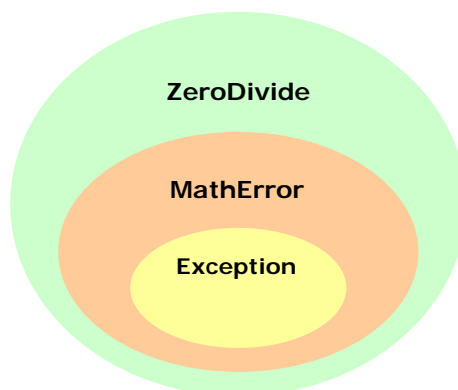
Every class we have defined so far has been a subclass of **Entity**, and the syntax that has launched the process of class creation has been of the form:

```
[global] constant SubclassName = class(SubclassName, SuperclassName)
-- where SuperclassName is Entity
```

If you read through **AN ORIENTATION TO DL** as well as **STEP 1, 2, and 3**, you'll be reminded of another DL base class - **Exception**. It's a superclass that has no properties or methods, and it's inherited by any error class that we write. It will help us to organise our class hierarchy of run-time errors (exceptions), like this:



Another way of depicting it is as follows:



And the syntax for creating these classes is:

```
[global] constant MathError = exception(MathError, Exception)
[global] constant ZeroDivide = exception(ZeroDivide, MathError)
```

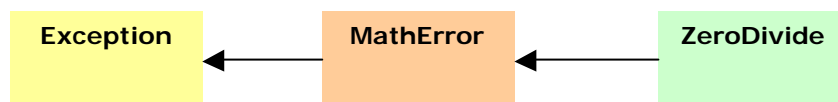
This syntax uses the DL routine `exception()` to enable us to define class **MathError** as a subclass of base class **Exception**, and class **ZeroDivide** as a subclass of class **MathError**. Our next task is to learn how to apply these concepts using DL.

DL's EXCEPTION HANDLING SYSTEM

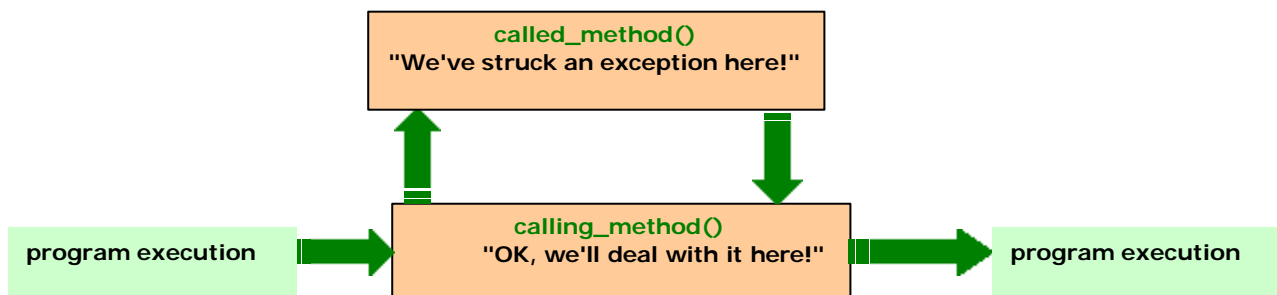
The DL system for exception handling offers us a certain amount of predefined functionality by which we can define, name, and systematically categorise exceptions that our application might encounter, and then offers us explicit alternatives by which we might deal with them.

We can summarise its capabilities as follows:

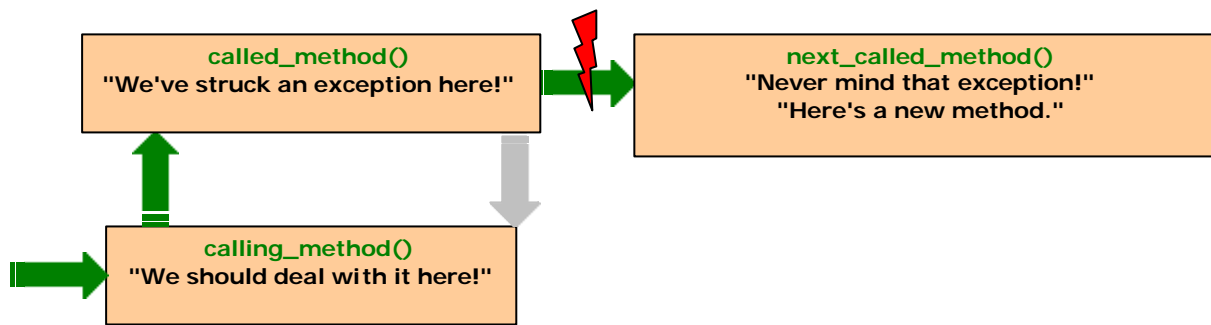
1. It allows us to identify and name various exceptions, and to categorise them as subclasses of **Exception**, or indeed as subclasses of one another. For example referring to the class hierarchy in the section above, we can represent classes of exceptions as follows:



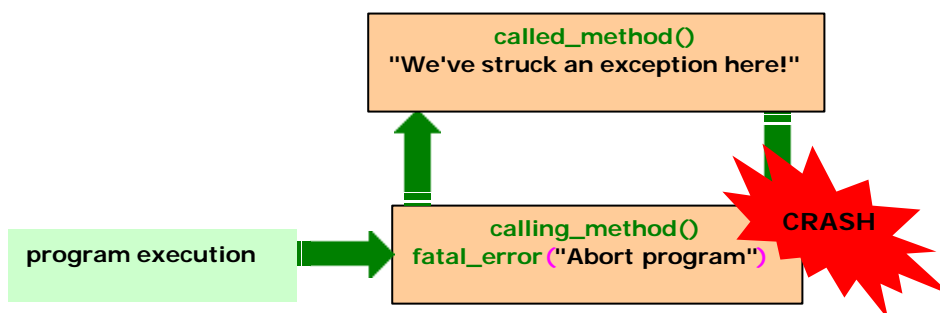
2. It offers a called method a way of *raising* an exception – "signalling" to the calling method or program that *a run-time error has been encountered*. And correspondingly offers the calling method or program a systematic approach for *clearing* ("processing") the error. Until that's done, the exception will be *pending* – "waiting to be cleared". Eg:



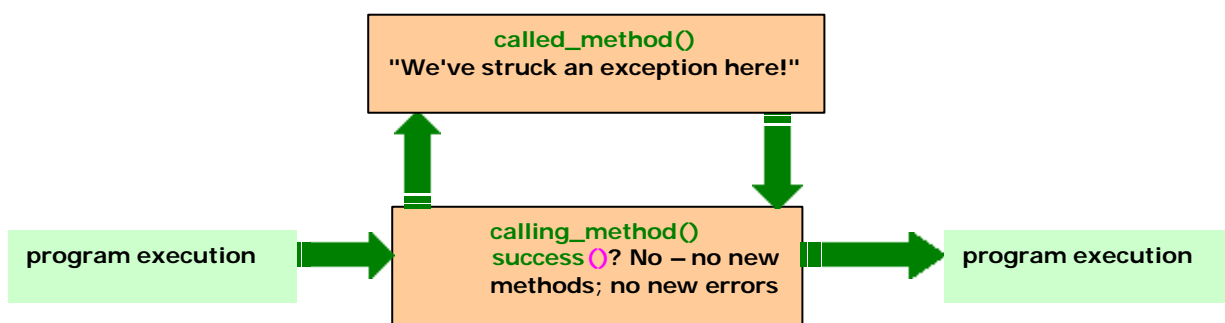
3. DL will not allow us to have more than one exception pending at a time – there's no point encountering new errors before we've dealt with the old ones! – so if a new exception is raised before an old one has been cleared, our program will stop immediately. In addition, while it's permissible to exit from a method with an exception pending, we can't enter a new method (where a new error may be encountered) without first clearing the old error. We can picture it as on the following page:



4. It allows us to choose how we will respond to our exceptions, according to the needs of our application:
- Sometimes we will want our program to stop as soon as a non-recoverable exception is encountered. We will use the procedure `fatal_error()` for that, in any program context, as for example:



- Sometimes we won't need to clear an exception, and we won't care which one it is – we'll only want to know whether or not there is one pending. We can use the function `success()` for that, in any context other than class definition. (But note that if an exception is pending, we won't be able to enter any new methods, or raise any more exceptions, until it's been processed.) For example:



- Before we can deal with an exception, we have to identify it somehow. In any program context other than class definition, we use the procedure `throw()` for this – it will set the current pending exception to the value of the handle of an exception class.
- And before we go ahead and process an exception, we'll want to verify that this class is the pending exception (or that it is a superclass of the pending exception). We use the function `catch()` for that, in any program context other than class definition. And once it has done its job, that exception class no longer has "pending" status.

expression, whose value is the handle of an exception class that was thrown); and we add the code necessary to handle that exception. In effect, **catch()** says to DL: "If there is an error of this type, I'm taking care of it – you can forget about it!". On the other hand, we won't have to do anything about a **fatal error** – it will be handled by an abrupt termination of the application.

The procedure **throw()** takes (the handle of) an exception as its argument, and makes it the current, *pending exception*, that will wait to be processed. If another *pending exception* already exists, the program will terminate immediately – you can't keep more than one exception waiting! The same thing will happen if you call a new method while you have pending exceptions from other methods. The rule is that you may leave one or more methods with an exception pending, but you may not enter a new method with an exception pending.

The function **catch()** works in partnership with the procedure **throw()**. It takes (the handle of) an exception as its argument, and asks: "Is this exception class (or any class derived from it) the current *pending exception*?". If it is, then it returns **TRUE** and strips that exception class (or the class derived from it) of its "current pending" status. Otherwise it returns **FALSE**, and makes no changes to the status quo – which could be to leave in existence another *pending exception* (for some other invocation of **catch()** to deal with).

Note that a call to **catch(Exception)** will process any *pending exception*, since it's asking: "this class, or any of its derived classes, the *pending exception*?" – "Yes!", since all exceptions are subclasses of **Exception**. Note also that in DL **catch()** is not a statement (as it is in Java, for instance) – it's a function that returns a boolean value, which needs to be processed – eg by an *if-else-[elsif]* construct. And be aware that it doesn't cause what's called *stack unwinding* – it doesn't change the execution path by passing the exception up the chain of called functions until it finds one that can deal with the error.

Sometimes you need to know if an exception is pending, but don't want to clear it and don't care what it is. You can check for an exception while leaving it pending, by calling **success()**. This is a function that takes no parameters, and which returns **TRUE** if there is no exception pending, and **FALSE** if there is an exception pending.

Sometimes you want to catch an exception, and do your method's local cleanup, but rethrow the exception to signal the error to the method that called yours. The routine **caught()** is a function that takes no parameters and returns the last exception that was processed by **catch()**. We need it because once **catch()** has been invoked, it will clear the pending exception whereas there are situations in which you want to know what was the exact exception, and do something, but not clear the exception.

I've discussed all this in one place under the one heading, so that you can refer to it from time to time. Our next task is to make sense of it by studying some simple examples.

STEP 23: HANDLING FATAL ERRORS

Recall **STEP 12**, where we defined **GreetingClass** with a parameterised constructor to take a string greeting as its argument. Let's modify the class definition, to check that the argument really is a string and, if it's not, to issue a fatal error and terminate the application. Here is what we could do:

```
-- GreetingClass.e v1.6

include diamondlite.e

-- check for type string
type string(object text)
  if atom(text) then return FALSE end if
```

```

    for i = 1 to length(text) do
        if not integer(text[i]) then return FALSE end if
        if text[i] < 32 or text[i] > 255 then return FALSE end if
    end for
    return TRUE
end type

global constant GreetingClass = class("GreetingClass", Entity)
    property("message", INSTANCE, NONE )

    function GreetingClass_new_1(object msg)
        entity newGreeting

        -- if the argument isn't a string, terminate immediately with an error message
        if not string(msg) then
            fatal_error("\nArgument must be a string!")
        end if

        newGreeting = call_method(super(), "new", NONE)
        set_property(newGreeting, "message", msg)
        return newGreeting
    end function
    method("new", 1, CLASS, routine_id("GreetingClass_new_1"))

    function GreetingClass_setMessage_1(object msg)
        -- if the argument isn't a string, terminate; otherwise set the property
        if not string(msg) then
            fatal_error("\nArgument must be a string!")
        else
            set_property(this(), "message", msg)
        end if
        return NIL
    end function
    method("setMessage", 1, INSTANCE, routine_id("GreetingClass_setMessage_1"))

    function GreetingClass_getMessage_0()
        return get_property(this(), "message")
    end function
    method("getMessage", 0, INSTANCE, routine_id("GreetingClass_getMessage_0"))
end_class()

```

And we could demonstrate how this might work with a modified version of **GreetingDemo.ex**:

```

-- GreetingDemo v1.7

include diamondlite.e
include GreetingClass.e

procedure main()
    entity myGreeting
    myGreeting = call_method(GreetingClass, "new", {12})
end procedure

main()

```

When we try to create a new entity by passing a number (ie 12) instead of a string, we get:

FATAL ERROR

Argument must be a string!

In GreetingClass class method new#1 called from main program

We see the error message we coded, and we are told that it has been issued from our class method `new()`, which takes 1 parameter (hence `#1`), which was called by `call_method()` during main program context.

STEP 23a: WHEN WE ONLY WANT TO KNOW IF THERE'S A PENDING EXCEPTION

Sometimes all we want to do is find out whether there's an exception pending at this moment – if there is, we'll know not to call any new methods – but we don't intend to process any pending exception, and we don't care which exception it is. We can use the function `success()` to answer the question: "Have we been **successful** in avoiding any exceptions so far?". If **TRUE**, we can proceed to call new methods. (Since we're looking at this from the point of view of *errors*, rather than emphasising being *error-free*, we usually use this function in the negative – ie **if not success() then <do this> else <continue the program> end if.**)

We'll illustrate this aspect of DL's exception-handling system by defining a class which will allow us to do division (given a valid numerator, and a valid, non-zero denominator), and display the answer. The program will crash if we enter an invalid character (eg `@`); if it finds any other errors – such as a string instead of a number; or a zero denominator – it will tell us that we've encountered some error somewhere in the program, and that we won't be able to continue.

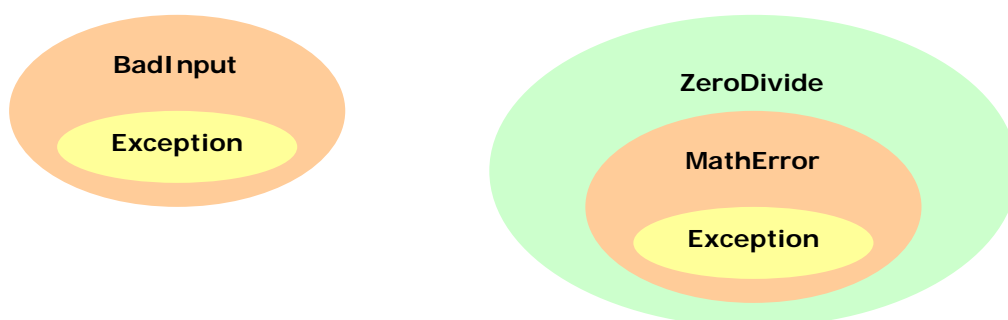
We can summarise our task like this:

- prompt and get user's numerator input
- if it's an invalid object (eg an ampersand sign - `&`) crash the program; otherwise...
- if it isn't a number, make `BadInput` the *pending exception*; otherwise...
- accept the input, and...
- prompt and get user's denominator input
- if it's an invalid object (eg a hash sign - `#`) crash the program; otherwise...
- if it isn't a number, make `BadInput` the *pending exception*; otherwise...
- accept the input, and...
- create a new entity, assigning the input values to their respective properties
- calculate (and display) the answer to the division
- if there is a *pending exception* prevent the program from continuing

One of our first tasks is to define our error classes. I've chosen to do this:

1. a *fatal error* will be triggered by entering an invalid Eu object
2. a `BadInput` subclass of **Exception**, for errors such as entering a string instead of a number
3. a `MathError` subclass of **Exception**, for any type of mathematical exception
4. a `ZeroDivide` subclass of `MathError`, for a zero denominator

We can display this schematically as follows:



This represents hierarchies of exception classes. Each class ultimately derives functionality from **Exception**, but adds something of its own (eg **MathError**) to be inherited by its subclasses (eg **ZeroDivide**).

To make it all work, we'll define a class, **Division.e**: its constructor will be responsible for creating an entity if all inputs are correct, and setting the entity's properties; it will have setter methods (which will check the input); and getter methods for each of the properties and for the answer to the division.

```
-- Division.e v1.0

include diamondlite.e
include get.e

without warning

-- define the exception classes
global constant BadInput = exception("BadInput", Exception)
global constant MathError = exception("MathError", Exception)
global constant ZeroDivide = exception("ZeroDivide", MathError)

-- begin defining the class Division
global constant Division = class("Division", Entity)
    property("numerator", INSTANCE, NIL)      -- Register two properties,
    property("denominator", INSTANCE, NIL)    -- with default values of 0.

    function Division_new_00()
        entity newDivision
        sequence input
        atom numerator, denominator

        puts(1, "\nEnter the numerator: ")    input = get(0)

        if (input[1] = GET_FAIL) then          -- If invalid input: fatal error
            fatal_error("\nYou entered an invalid Eu object.")
            newDivision = Null_Instance        -- don't create an entity.
        elseif sequence(input[2]) then        -- If input is a string
            throw(BadInput)                   -- make this the pending exception
            newDivision = Null_Instance        -- don't create an entity.
        else                                   -- Otherwise....
            numerator = input[2]              -- accept the numerator.

            puts(1, "\nEnter the denominator: ")    input = get(0)

            if (input[1] = GET_FAIL) then
                fatal_error("\nYou entered an invalid Eu object.")
                newDivision = Null_Instance
            elseif sequence(input[2]) then
                throw(BadInput)
                newDivision = Null_Instance
            elseif (input[2] = 0) then         -- If denominator is 0 make
                throw(ZeroDivide)             -- this the pending exception
                newDivision = Null_Instance    -- don't create an entity.
            else                               -- Accept the denominator, create an entity, & set the properties.
                denominator = input[2]
                newDivision = call_method(super(), "new", NONE)
                set_property(newDivision, "numerator", numerator)
```

```

        set_property(newDivision, "denominator", denominator)
    end if
end if

return newDivision
end function
method("new", 0, CLASS, routine_id("Division_new_0"))

function Division_setNumerator_1(object input) -- similar checks to constructor
    if (input[1] = GET_FAIL) then
        fatal_error("\nYou entered an invalid Eu object.")
    elseif sequence(input[2]) then
        throw(BadInput)
    else
        set_property(this(), "numerator", input[2])
    end if

    return NIL
end function
method("setNumerator", 1, INSTANCE, routine_id("Division_setNumerator_1"))

function Division_setDenominator_1(object input) -- similar checks to constructor
    if (input[1] = GET_FAIL) then
        fatal_error("\nYou entered an invalid Eu object.")
    elseif sequence(input[2]) then
        throw(BadInput)
    elseif (input[2] = 0) then
        throw(ZeroDivide)
    else
        set_property(this(), "denominator", input[2])
    end if

    return NIL
end function
method("setDenominator", 1, INSTANCE, routine_id("Division_setDenominator_1"))

function Division_getNumerator_0()
    return get_property(this(), "numerator")
end function
method("getNumerator", 0, INSTANCE, routine_id("Division_getNumerator_0"))

function Division_getDenominator_0()
    return get_property(this(), "denominator")
end function
method("getDenominator", 0, INSTANCE,
        routine_id("Division_getDenominator_0"))

function Division_getAnswer_0()
    atom answer
    -- do the division, and return the quotient
    answer = get_property(this(), "numerator") /
            get_property(this(), "denominator")

    return answer
end function
method("getAnswer", 0, INSTANCE, routine_id("Division_getAnswer_0"))
end_class()

```

The corresponding application file, **DivisionDemo.ex**, will use the syntax **not success()** at strategic points, to ascertain whether we can continue execution. If it finds a pending exception, it will tell us that we can't go on. On its own, it won't be able to tell us which exception we've encountered – only that there is one – and it won't be able to clear it, either.

```
-- DivisionDemo.ex v1.0

include Division.e

procedure main()
  entity myDivision
  sequence input

  myDivision = call_method(Division, "new", NONE)
  if not success() then      -- Did we succeed in creating a new entity?
    puts(1, "\nThere's an exception somewhere in your program.")
    puts(1, "\nYou won't be able to call any new methods.")
  else                      -- Yes. Now get the answer, and get a new numerator.
    printf(1, "\nThe answer is %.2f",
           call_method(myDivision, "getAnswer", NONE))

    puts(1, "\nEnter the numerator: ")      input = get(0)
    VOID = call_method(myDivision, "setNumerator", {input})
    if not success() then      -- Was the numerator OK?
      puts(1, "\nThere's an exception somewhere in your program.")
      puts(1, "\nYou won't be able to call any new methods.")
    else                      -- Yes. Now get the denominator.
      puts(1, "\nEnter the denominator: ")  input = get(0)
      VOID = call_method(myDivision, "setDenominator", {input})
      if not success() then    -- Was it OK too?
        puts(1, "\nThere's an exception somewhere in your program.")
        puts(1, "\nYou won't be able to call any new methods.")
      else                    -- Yes. Now get the answer.
        printf(1, "\nThe answer is %.2f",
               call_method(myDivision, "getAnswer", NONE))
      end if
    end if
  end if
end if
end procedure

main()
```

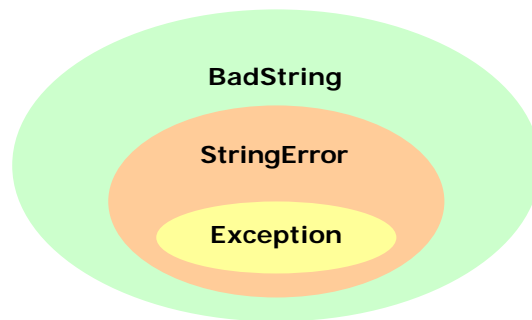
Familiarise yourself with the application, running it with different types of input.

STEP 23b: WHEN WE'LL CLEAR WHICHEVER EXCEPTION WE HAPPEN TO CATCH

Sometimes it isn't enough for us to become aware that there's an exception pending – we want to clear it, so that we can get on with the rest of our program. It mightn't be critical for us to know exactly which exception we've encountered – perhaps because we'll deal with it in the same way, whatever error it is.

We can use the syntax **catch(Exception)** to answer the question: "Is this class, or any of its subclasses, the pending exception?". Since **Exception** is at the heart of all exceptions, it will catch any exception.

For example we might decide to define a class of error called **BadString** which is a subclass of **StringError**, which in turn is a subclass of **Exception**. We could picture the classes like this:



In our class definition file we would call the DL routine **exception()**, to enable these classes to inherit from the base class. When we attempt to create a new instance of our **GreetingClass**, we would test whether the argument we pass in is a valid string. If it were not, we wouldn't create a new instance, but we would call **throw()** to identify the particular error we had encountered. Assuming that in this application we didn't care which particular error we process – only that we deal with whatever error we encounter – we would call **catch(Exception)** to indicate that fact. We could code it like this:

```
-- GreetingClass.e v1.7

include diamondlite.e

type string(object text )
  if atom(text) then return FALSE end if
  for i = 1 to length(text) do
    if not integer(text [i]) then return FALSE end if
    if text [i] < 32 or text [i] > 255 then return FALSE end if
  end for
  return TRUE
end type

global constant StringError = exception("StringError", Exception)
global constant BadString = exception("BadString", StringError)

global constant GreetingClass = class("GreetingClass", Entity)
  property("message", INSTANCE, NONE)

  function GreetingClass_new_1(object msg)
    entity newGreeting

    if not string(msg) then
      throw(BadString)          -- the handle of the exception
      return Null_Instance      -- don't create a new entity
    end if

    newGreeting = call_method(super(), "new", NONE)
    set_property(newGreeting, "message", msg)
    return newGreeting
  end function
  method("new", 1, CLASS, routine_id("GreetingClass_new_1"))

  function GreetingClass_setMessage_1(object msg)
    if not string(msg) then
      throw(BadString)          -- the handle of the exception
```

```

        else                -- otherwise, go ahead and set the property
            set_property(this(), "message", msg)
        end if
        return NIL
    end function
    method("setMessage", 1, INSTANCE, routine_id("GreetingClass_setMessage_1"))

    function GreetingClass_getMessage_0()
        return get_property(this(), "message")
    end function
    method("getMessage", 0, INSTANCE, routine_id("GreetingClass_getMessage_0"))
end_class()

```

We will now code our application in such a way that it clears any exception it encounters. As an extra, however, and in order to give us an opportunity to use a couple of other DL routines, we will ask our application to tell us the name of the class from which the error derived. We can do this by using the functions `caught()` and `class_name()`. By using them together – `class_name(caught())` – we can answer the question: "What is the name of the class of the exception that was last cleared by the function `catch()`?". Here is how we could do it:

```

-- AnyErrorDemo.ex v1.0

include GreetingClass.e

procedure main()
    entity myGreeting
    myGreeting = call_method(GreetingClass, "new", {12})
    if catch(Exception) then -- if this, or one of its subclasses, is the pending exception
        puts(1, "\nAn instance could not be created!")
        printf(1, "\nThe exception's class name is %s",
                {class_name(caught())})
    else
        printf(1, "\nmyGreeting's property is %s",
                {call_method(myGreeting, "getMessage", NONE)})
    end if
end procedure

main()

```

When you run this application, you'll get the following output:

```

An instance could not be created!
The exception's class name is BadString

```

STEP 23c: WHEN EACH EXCEPTION IS CLEARED IN ITS OWN PARTICULAR WAY

But what if it matters to us (and to the user), what kind of error the application encounters? Imagine a situation in which a program may begin only if the user enters a particular word – like a password. In real life: if the password is correct the program will commence; otherwise a message would be displayed – perhaps **Have you forgotten your password? Try again** – and the user would get another chance (say up to a total of three attempts, before being shut out of the application altogether).

For teaching purposes, let's relax the requirements somewhat. Let's say that the password to be entered is "BEGIN", and that the user has an unlimited number of attempts to enter it.

After each entry the application checks the input for errors, and displays a description of the kind of error that it found. When the correct password is entered the program displays:

Valid Password – begin the program.

A point form algorithm to achieve these requirements might go as follows:

1. prompt the user to enter the password
2. get the user's input
3. perform a series of checks on the input:
 - if there was no input (eg user just pressed the <Enter> key)
display "No Input" and go back to #1
 - otherwise if the input was invalid (eg entered something other than valid letters)
display "Invalid Input" and go back to #1
 - otherwise if the input wasn't the correct word (eg user entered "START")
display "Incorrect Input" and go back to #1
 - otherwise
display "Correct Input" and move on
4. let the program begin

Let's think how we might code this functionality in non-OO Eu first. Here's a first draft:

```
-- SecurityDemo.ex v1.0

-- All the letters of the alphabet; they make up a valid password
constant VALID_CHARS =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

constant FALSE = 0
constant TRUE  = 1

-- A function to test whether a password is valid (but not necessarily correct!)
type valid_string(object x)
    if atom(x) then return FALSE end if -- If it isn't a sequence, then it's invalid!
    for i = 1 to length(x) do           -- If any of its letters
        if not find(x[i], VALID_CHARS) then -- isn't a valid character, then
            return FALSE                -- the password is invalid.
        end if
    end for
    return TRUE -- If it has passed the tests above, then it must be a valid password.
end type

procedure main()
    sequence pwd

    while 1 do
        puts(1, "\nEnter the password: ")
        pwd = gets(0)      pwd = pwd[1..length(pwd)-1] -- remove '\n'

        if length(pwd) = 0 then
            puts(1, "\nNo Input")
        elsif not valid_string(pwd) then
            puts(1, "\nInvalid Input")
        elsif not equal(pwd, "BEGIN") then
            puts(1, "\nIncorrect Input")
        else
            puts(1, "\nCorrect Input")
            exit
        end if
    end while
end procedure
```

```

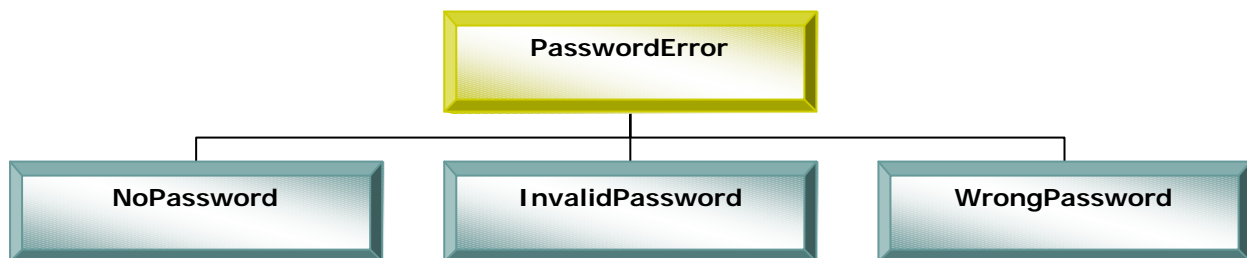
end while
puts(1, "\nThe program can now begin...")
-- other method calls and application processing would be placed here
end procedure

main()

```

Run the application a few times, entering different passwords and noting the error messages until you enter the correct password, at which point the program will stop. Notice that the error messages are not irrelevant – they matter to you, because (presumably!) you are paying attention to them and correcting your mistakes.

Our task now is to create a class definition that will incorporate all this functionality. We'll tackle the errors first. We can think of each of these errors as *a type of PasswordError*, inheriting all its attributes and adding others particular to the specific error in question:



And in turn, we can think of **PasswordError** as *a type of Exception*, inheriting all its attributes and adding any others that are particular to PasswordError itself. We can therefore expect part of the code in our class file to contain the following statements:

```

-- in *.e

global constant PasswordError = exception("PasswordError", Exception)
global constant NoPassword    = exception("NoPassword", PasswordError)
global constant InvalidPassword = exception("InvalidPassword", PasswordError)
global constant WrongPassword = exception("WrongPassword", PasswordError)

```

We can also expect that the class file will contain code signalling that it has encountered an error. From what we have seen in **STEP 23a**, we should expect it to look something like this:

```

-- in *.e

if <we encounter this condition> then
    throw(NoPassword)
elsif <we encounter that condition> then
    throw(InvalidPassword)
elsif <we encounter another condition> then
    throw(WrongPassword)
else
    <we encounter no error condition>
end if

```

Correspondingly we should expect our application file to contain code to respond to error conditions; and we can look at **STEP 23b** for a clue to the syntax:

```

-- in *.ex

if catch(NoPassword) then
    <do this thing>
elseif catch(InvalidPassword) then
    <do that thing>
elseif catch(WrongPassword) then
    <do another thing>
else
    <do what should be done if no error was encountered>
end if

```

To help us put these ideas together, let us design a class whose constructor becomes responsible for getting the user's input, checking it for one of the errors above, and bringing it to the attention of the application so that it can deal with the problem. Only when there is no error will the constructor create a new entity – otherwise it will create only a **Null_Instance**. Here is one way of coding the class definition file:

```

-- SecureClass.e v1.0

-- create subclasses of PasswordError, whose parent class is Exception
global constant PasswordError = exception("PasswordError", Exception)
global constant NoPassword = exception("NoPassword", PasswordError)
global constant InvalidPassword = exception("InvalidPassword", PasswordError)
global constant WrongPassword = exception("WrongPassword", PasswordError)

constant VALID_CHARS =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

type valid_string(object x)
    if atom(x) then return FALSE end if
    for i = 1 to length(x) do
        if not find(x[i], VALID_CHARS) then
            return FALSE
        end if
    end for
    return TRUE
end type

-- class definition for SecureClass
global constant Secure = class("Secure", Entity)
    property("password", INSTANCE, NONE)

    function Secure_new_0()
        entity newSecure
        sequence pwd

        -- we get the user's input
        puts(1, "\nEnter the password: ")
        pwd = gets(0)    pwd = pwd[1..length(pwd)-1]

        -- and test it for errors; we create an entity only if there are no errors
        if length(pwd) = 0 then
            throw(NoPassword)
            newSecure = Null_Instance
        elseif not valid_string(pwd) then
            throw(InvalidPassword)

```



```

        newSecure = Null_Instance
    elsif not equal(pwd, "BEGIN") then
        throw(WrongPassword)
        newSecure = Null_Instance
    else
        newSecure = call_method(super(), "new", NONE)
        set_property(newSecure, "password", pwd)
    end if

    return newSecure
end function
method("new", 0, CLASS, routine_id("Secure_new_0"))
end_class()

```

The corresponding application file could then be as follows:

```

-- SecurityDemo.ex v1.1

include diamondlite.e
include SecureClass.e

procedure main()
    entity mySecure

    while 1 do
        -- call the constructor each time to get and test input
        -- if find an error, call constructor all over again
        -- stop calling when correct password is entered
        mySecure = call_method(Secure, "new", NONE)

        if catch(NoPassword) then
            puts(1, "\nNo Input")
        elsif catch(InvalidPassword) then
            puts(1, "\nInvalid Input")
        elsif catch(WrongPassword) then
            puts(1, "\nIncorrect Input")
        else
            puts(1, "\nCorrect Input")
            exit
        end if
    end while
    puts(1, "\nThe program can now begin...")
    -- other method calls and application processing would be placed here
end procedure

main()

```

Run the application several times, to confirm that you get the error messages that correspond to your input.

STEP 23d: AN EXTENSION EXERCISE

We can introduce a slightly more realistic feel to the behaviour of our application, and at the same time experiment with our code a little more. Let's say that we will give the user a maximum of three attempts to enter the correct password – after that, the program will automatically abort. We can preserve the basic structure of our code, making a couple of modifications:

```

-- code outline in *.ex

for i = 1 to max_attempts do
    -- initial statements, eg a method call
    if <catch an exception> then
        <deal with it>
    elsif <catch another exception> then
        <deal with that too>
    else
        <no exception to process>
        exit -- the loop
    end if

    if i = max_attempts then
        return -- from the procedure; end of program
    end if
end for
-- continue with rest of program

```

We can use this outline to modify **SecurityDemo.ex** as follows:

```

-- SecurityDemo.ex v1.2

include diamondlite.e
include SecureClass.e

constant MAX_ATTEMPTS = 3 -- give user up to 3 attempts

procedure main()
    entity mySecure

    for counter = 1 to MAX_ATTEMPTS do
        mySecure = call_method(Secure, "new", NONE)

        if catch(NoPassword) then
            puts(1, "\nNo Input")
        elsif catch(InvalidPassword) then
            puts(1, "\nInvalid Input")
        elsif catch(WrongPassword) then
            puts(1, "\nIncorrect Input")
        else
            puts(1, "\nCorrect Input")
            exit
        end if

        if counter = MAX_ATTEMPTS then
            puts(1, "\nLogin failure – abort program")
            return -- from the procedure; end the program
        end if
    end for
    puts(1, "\nThe program can now begin...")
    -- other method calls and application processing would be placed here
end procedure

main()

```

Run the application again, and confirm that it behaves as it did before.

STEP 23e: RETHROWING EXCEPTIONS

We can now take this opportunity to bring together some of the ideas we've already discussed while we have a look at a very simple example of *rethrowing* exceptions.

Sometimes we catch an exception, but for one reason or another we don't want to clear it just yet – perhaps the current method needs to do some local processing and still keep the exception pending so that it can make its caller aware of it. The problem with using `catch()` on its own, is that it will automatically clear the exception. In DL we can use `throw(caught())` to allow us to `throw()` (again) the last exception (ie `caught()`) that was processed by `catch()`.

To illustrate how to do this, we will suppose that we need an application to get from the user a *non-zero, positive integer*. Let's say that if the user enters something totally invalid – or for that matter a string or a zero (0) – that the program crashes, or at least fails to create an entity or proceed any further. But let's also say that the program tries to help the user a bit:

- if a *negative integer* is entered, the program replaces it with a (positive) absolute value, and uses that value instead; eg **-3** becomes **3**
- if a *floating point number* is entered, the program does one of two things:
 - if a *negative float* is entered, the program rounds it to the closest positive integer (1) and uses that value instead; eg **-3.2** becomes **1**
 - if a *positive (non-zero) float* is entered, the program rounds it up or down to the nearest non-zero integer and uses that value instead; eg **0.2** becomes **1**; **0.7** becomes **1**; **1.3** becomes **1**; **1.6** becomes **2**

Let's say the class has a single property called "inputNum", and that the class constructor will be responsible for getting and checking the user's input, and creating a new entity according to the above conditions. The class will contain two methods – "setNumber" and "getNumber".

We can outline the definition for an **InputCheck** class as follows:

```
-- InputCheck.e v1.0

include diamondlite.e
include get.e

global constant Input = class("Input", Entity)
    property("inputNum", INSTANCE, NIL)

    function Input_new_0()
        entity newInput
        -- <code>
        return newInput
    end function
    method("new", 0, CLASS, routine_id("Input_new_0"))

    function Input_setNumber_1(object in)
        -- <code>
        return NIL
    end function
    method("setNumber", 1, INSTANCE, routine_id("Input_setNumber_1"))

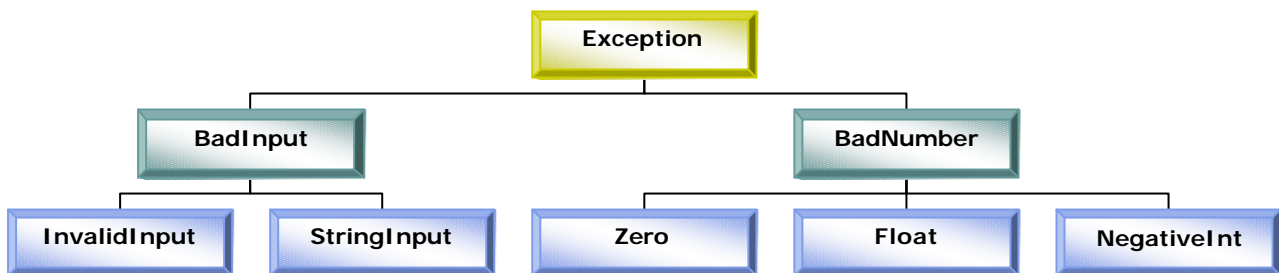
    function Input_getNumber_0()
        return get_property(this(), "inputNum")
    end function
```

```

method("getNumber",0,INSTANCE,routine_id("Input_getNumber_0"))
end_class()

```

Now let's consider a hierarchy of errors that our class might need to define. I've chosen to create two broad classes of exception – **BadInput**, and **BadNumber**. **BadInput** has two subclasses – **InvalidInput** (for rubbish input) and **StringInput** (for a string input in place of a number) – and **BadNumber** has three subclasses: **Zero** (for input of 0); **Float** (for input of a float); and **NegativeInt** (for input of a negative integer). They all have **Exception** as their "ultimate" superclass.



We can now consider how to raise these exceptions. Below is an outline of an algorithm I've chosen to implement:

```

if (user enters an invalid character) then
    <tell caller that an InvalidInput exception has been encountered>
    <don't create a new entity>
otherwise if (user enters a string) then
    <tell caller that a StringInput exception has been encountered>
    <don't create a new entity>
otherwise if (user enters a 0) then
    <tell caller that a Zero exception has been encountered>
    <don't create a new entity>
otherwise
    <create a new entity>
    <set its property to the value of the input>
    if (the input is not an integer) then
        <tell caller that a Float exception has been encountered>
    otherwise if (the input is a negative integer) then
        <tell caller that a NegativeInt exception has been encountered>
    end if
end if
end if

```

I'll bypass discussing it here, and move on to present an outline of an algorithm to handle these exceptions, using *rethrowing*:

```

if (encounter any kind of BadInput exception) then
    <warn user>
    (rethrow whichever exception has been encountered)
end if

if (it is an InvalidInput exception) then
    <fatal error: crash the program>
otherwise if (it is a StringInput exception) then
    <tell the user about it>
end if

```

```

if <encounter any kind of BadNumber exception> then
    <warn the user>
    (rethrow whichever exception has been encountered)
otherwise
    <tell the user that input was fine and needed no correction>
    <get value of the property>
end if

if (it is a ZeroException) then
    <tell the user that no entity was created>
otherwise if (it is a Float exception) then
    <tell the user>
    <get the value of the property>
    <if it's negative, round it up to 1>
    <otherwise round it up or down to the nearest integer larger than 1>
    <reset the property to the new value>
otherwise if (it is a NegativeInt exception) then
    <tell the user>
    <get the value of the property>
    <get its absolute value>
    <reset the property to the new value>
end if

```

The really important concept here is that we can throw an exception – eg by way of its superclass – and then catch it somewhere else; but that instead of processing the exception then and there, we do one or two things and *rethrow* the exception for clearing somewhere else.

We can now look at the full class definition, comparing the code against the outlines above:

```

-- InputCheck.e v1.0

include diamondlite.e
include get.e

global constant BadInput    = exception("BadInput", Exception)
global constant InvalidInput = exception("InvalidInput", BadInput)
global constant StringInput = exception("StringInput", BadInput)
global constant BadNumber   = exception("BadNumber", Exception)
global constant Zero        = exception("Zero", BadNumber)
global constant Float       = exception("Float", BadNumber)
global constant NegativeInt = exception("NegativeInt", BadNumber)

global constant Input = class("Input", Entity)
    property("inputNum", INSTANCE, NIL)

    function Input_new_0()
        entity newInput
        sequence in

        puts(1, "\nEnter a positive, non-zero integer: ") in = get(0)

        if (in[1] = GET_FAIL) then
            throw(InvalidInput)
            newInput = Null_Instance
        elsif (sequence(in[2])) then
            throw(StringInput)

```

```

        newInput = Null_Instance
    elsif (in[2] = 0) then
        throw(Zero)
        newInput = Null_Instance
    else
        newInput = call_method(super(), "new", NONE)
        set_property(newInput, "inputNum", in[2])

        if not integer(in[2]) then
            throw(Float)
        elsif (in[2] < 0) then
            throw(NegativeInt)
        end if
    end if
    return newInput
end function
method("new", 0, CLASS, routine_id("Input_new_0"))

function Input_setNumber_1(object in)
    set_property(this(), "inputNum", in)
    return NIL
end function
method("setNumber", 1, INSTANCE, routine_id("Input_setNumber_1"))

function Input_getNumber_0()
    return get_property(this(), "inputNum")
end function
method("getNumber", 0, INSTANCE, routine_id("Input_getNumber_0"))
end_class()

```

And we can examine the application file, again comparing it against the outlines above:

```

-- RethrowDemo.ex v1.0

include InputCheck.e

procedure main()
    entity myEntity
    atom input

    myEntity = call_method(Input, "new", NONE)
    if catch(BadInput) then
        puts(1, "\nWarning: your input is incorrect!")
        throw(caught())
    end if

    if catch(InvalidInput) then
        fatal_error("You did not enter a valid Eu object. The program will close now.")
        return
    elsif catch(StringInput) then
        puts(1, "\nYou entered a string. An entity was not be created.")
        return
    end if

    if catch(BadNumber) then
        puts(1, "\nYou entered a number, but it was not quite right.")
        throw(caught())
    end if
end procedure

```

```

else
    printf(1, "\nYou entered correctly." &
           "\nAn entity was created." &
           "\nIts property was set at %d",
           call_method(myEntity, "getNumber", NONE))
end if

if catch(Zero) then
    puts(1, "\nYou entered zero. An entity was not created.")
elseif catch(Float) then
    puts(1, "\nYou entered a floating point number.")
    input = call_method(myEntity, "getNumber", NONE)
    if (input < 0) then
        input = 1
        printf(1, "\nIt was a negative number." &
               "\nIts closest positive integer is %d", input)
    else
        input = floor(input)
        if (input = 0) then
            input += 1
        elseif (remainder(input, 10) >= 5) then
            input += 1
        end if
        printf(1, "\nIts closest positive integer is %d", input)
    end if

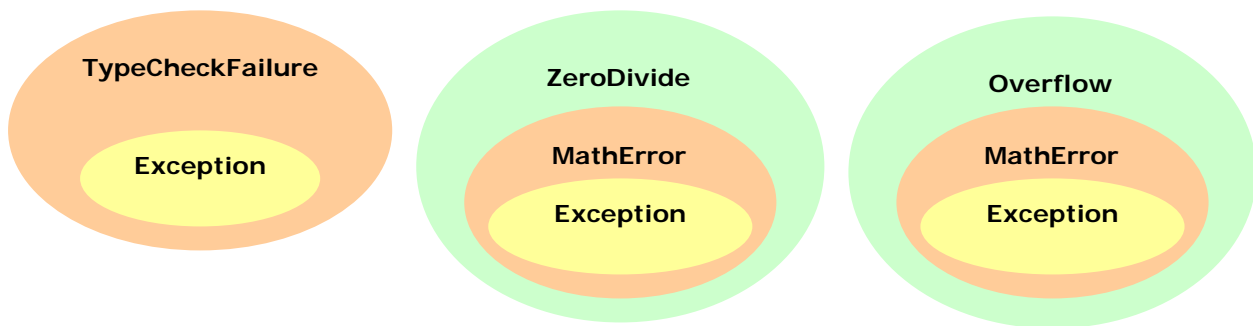
    VOID = call_method(myEntity, "setNumber", {input})
    printf(1, "\nAn entity was created." &
           "\nIts property was set at %d",
           call_method(myEntity, "getNumber", NONE))
elseif catch(NegativeInt) then
    puts(1, "\nYou entered a negative integer.")
    input = - (call_method(myEntity, "getNumber", NONE))
    printf(1, "\nIts absolute value is %d", input)
    VOID = call_method(myEntity, "setNumber", {input})
    printf(1, "\nAn entity was created." &
           "\nIts property was set at %d",
           call_method(myEntity, "getNumber", NONE))
end if
end procedure

main()

```

STEP 24: A FULL CLASS DEFINITION, COMPLETE WITH EXCEPTIONS

We should now be able to write a fully functional class definition, complete with exceptions. I couldn't do any better than to include this code by Michael Nelson. It's a reworking of **Product Class** from **STEP 16**, renamed to **IntMath**. As an extension exercise, you might like to write your own application to use this class. Note that it defines two broad exception classes – **TypeCheckFailure**, and **MathError** – and that the latter class is a superclass for two other classes: **ZeroDivide**, and **Overflow**. We can represent them as on the following page:



The class **IntMath**, has two instance properties, "first" and "second" (which are initially set to 0), and uses the automatic default constructor, destructor, and clone methods. It has methods to set and get each property, and methods to carry out the arithmetic operations sum, difference, product, quotient, and remainder. The setter methods test input, and throw a TypeCheckFailure exception if they detect something other than an integer. The "calculator" methods test the answers they derive, and throw an Overflow exception if they detect something other than an integer. The Quotient method also tests the divisor, and throws a ZeroDivide exception if it detects 0. Notice that the Quotient and the Remainder methods test for two exceptions each.

```

-- Class: IntMath
-- input two integers, and calculate and display their IntMath

include diamondlite.e

global constant TypeCheckFailure = exception("TypeCheckFailure", Exception)
global constant MathError       = exception("MathError", Exception)
global constant ZeroDivide      = exception("ZeroDivide", MathError)
global constant Overflow        = exception("Overflow", MathError)

global constant IntMath = class("IntMath", Entity)
  property("first", INSTANCE, 0)
  property("second", INSTANCE, 0)

  function IntMath_setFirst_1(object n1)
    if integer(n1) then
      set_property(this 0, "first", n1)
    else
      throw(TypeCheckFailure)
    end if

    return NIL
  end function
  method("setFirst", 1, INSTANCE, routine_id("IntMath_setFirst_1"))

  function IntMath_setSecond_1(object n2)
    if integer(n2) then
      set_property(this 0, "second", n2)
    else
      throw(TypeCheckFailure)
    end if

    return NIL
  end function
  method("setSecnd", 1, INSTANCE, routine_id("IntMath_setSecond_1"))

```



```

function IntMath_getFirst_0()
    return get_property(this 0, "first")
end function
method("getFirst", 0, INSTANCE, routine_id("IntMath_getFirst_0"))

function IntMath_getSecond_0()
    return get_property(this 0, "second")
end function
method("getSecond", 0, INSTANCE, routine_id("IntMath_getSecond_0"))

function IntMath_Sum_0()
    atom sum

    sum = get_property(this 0, "first") + get_property(this 0, "second")
    if not integer(sum) then
        throw(Overflow)
        return 0
    end if

    return sum
end function
method("Sum", 0, INSTANCE, routine_id("IntMath_Sum_0"))

function IntMath_Difference_0()
    atom diff

    diff = get_property(this 0, "first") - get_property(this 0, "second")
    if not integer(diff) then
        throw(Overflow)
        return 0
    end if

    return diff
end function
method("Difference", 0, INSTANCE, routine_id("IntMath_Difference_0"))

function IntMath_Product_0()
    atom prod

    prod = get_property(this 0, "first") * get_property(this 0, "second")
    if not integer(prod) then
        throw(Overflow)
        return 0
    end if

    return prod
end function
method("Product", 0, INSTANCE, routine_id("IntMath_Product_0"))

function IntMath_Quotient_0()
    integer second
    atom quot

    second = get_property(this 0, "second")
    if second = 0 then
        throw(ZeroDivide)
        return 0
    end if
    quot = first / second
    return quot
end function
method("Quotient", 0, INSTANCE, routine_id("IntMath_Quotient_0"))

```

```

        end if

        quot = get_property(this(), "first") / second
        if quot < 0 then
            quot = -floor(-quot)
        else
            quot = floor(quot)
        end if

        if not integer(quot) then
            throw(Overflow)
            return 0
        end if

        return quot
    end function
    method("Quotient", 0, INSTANCE, routine_id("IntMath_Quotient_0"))

function IntMath_Remainder_0()
    integer second
    atom rem

    second = get_property(this(), "second")
    if second = 0 then
        throw(ZeroDivide)
        return 0
    end if

    rem = remainder(get_property(this(), "first"), second)
    if not integer(rem) then
        throw(Overflow)
        return 0
    end if

    return rem
end function
    method("Remainder", 0, INSTANCE, routine_id("IntMath_Remainder_0"))
end_class()

```

In your application file, the code for calling the setter methods will need to look something like this:

```

VOID = call_method(className, methodName, {argument})
if catch(TypeCheckFailure) then
    -- code to handle the error
else
    -- code to continue with program
end if

```

The code for calling the methods that do the calculations – except for the methods **quotient()** and **remainder()** – will look something like this:

```

VOID = call_method(className, methodName, NONE)
if catch(Overflow) then
    -- code to handle the error
else


```

```
end if -- code to continue with program
```

The code for calling the last two methods could be a bit different, since there are two errors to be tested. For instance you might decide to test for `MathError`, which will return **TRUE** whether `Overflow` or `ZeroDivide` is encountered, and then rethrow the exception to handle it explicitly. For instance:

```
VOID = call_method(className, methodName, NONE)
if catch(MathError) then
    -- do some local cleanup
    -- perhaps even stop execution
    throw(caught()) -- rethrow the error: either ZeroDivide or Overflow
end if

if catch(ZeroDivide) then
    -- code to handle this error
elseif catch(Overflow) then
    -- code to handle this error
else
    -- code to continue with program
end if
```



This section has given us lots of opportunities to think about classes and the way they are inherited. Indeed this is the topic of our final chapter.

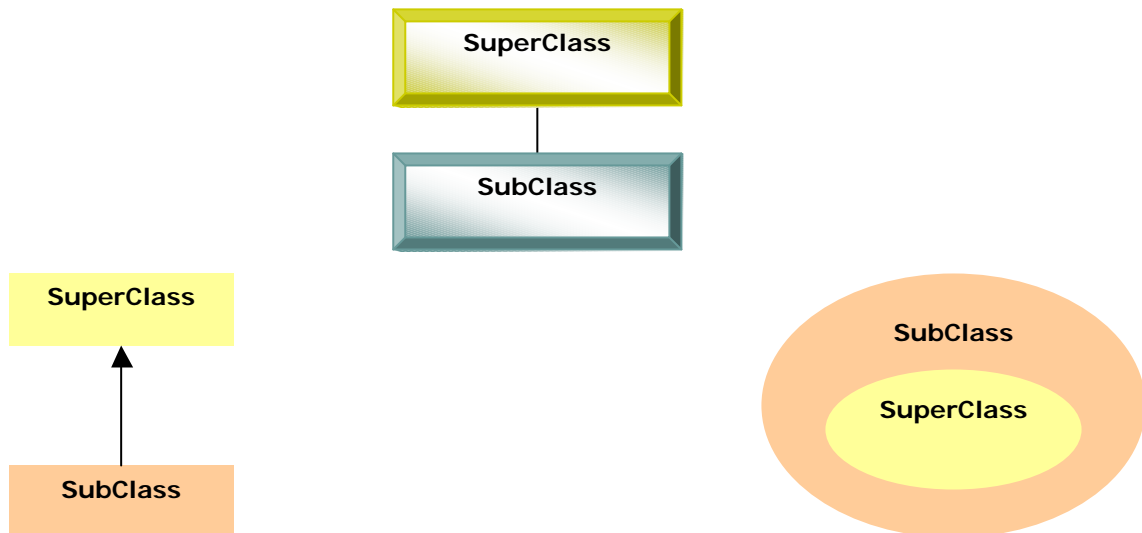
INHERITANCE

Dealing with exceptions has given us an opportunity to discuss the concept of *inheritance*, and the idea that we can structure our classes in a hierarchy based on the relationship between their classes. In fact we've touched on inheritance almost from the beginning of this Guide. We can now consolidate what we've learnt, and add one or two more topics that will finish off our introductory learning.

At its heart, *inheritance* refers to a capability offered to us by the OO approach – the ability to take what's already available in a fully-functional stand-alone class, and make that the "core" of a new, more specialised, more extended class that has certain unique qualities of its own. The "extended" class therefore **is** a form of the "core" class – usually with upgraded, additional features. (Note that the "core" class doesn't lose any of its functionality – it remains fully serviceable, and may even be the preferred class to use in some situations.) Because of this relationship between the "core" and the "extended" class, it becomes possible for us to create a well-structured, hierarchical category of classes that have a logical relationship to one another.

Note that we have already met situations in which one class was incorporated into the design of another (look at the section called **A FIRST LOOK AT COMPOSITION**), but that was different: the two classes were fundamentally separate, and there was no "core" / "extension" relationship between them – one class simply **had** the other class within it. Those classes could not form a logically related category of entities.

Recall that the inherited class (from which another class inherits functionality) is known as a *superclass* or *parent class* or *base class*, and that the inheriting class is known as a *subclass* or *child class* or *derived class*. And recall that we can represent the relationship between them schematically in various ways - eg:



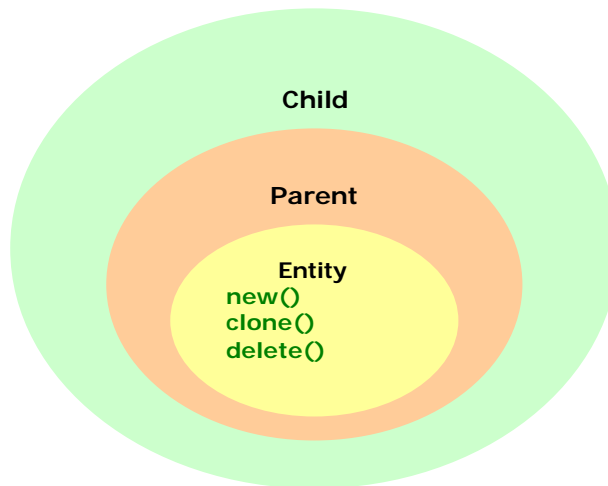
In DL each class that we define may have only one *superclass*. Moreover a *subclass* is potentially allowed access to all the properties and methods available to its corresponding superclass – but the superclass doesn't know about any extra properties or methods available in the subclass. Note that this access is not automatic, at least for properties (remember that in DL all properties are *private* by default) – the superclass must have accessor methods (which in DL are *public* by default) to allow such access. All the superclass' methods will be public and therefore accessible. Sometimes that's just what we want. When we don't want them to be accessible, we can use **NULL_METHOD** to do nothing and return **NIL**. (See **GenericClass.e** in **A RECAP AND A LOOK AHEAD...** after **STEP 17b**.)

Now might be a good time to have another look at **APPENDIX: DL's CLASS SYSTEM**, to remind yourself how the classes we write inherit from DL's predefined classes. Reading through **INTRODUCTORY CONCEPTS** will remind you how entities are created and referenced. And a read through **AN INTRODUCTION TO INHERITANCE** (in the chapter called **EXCEPTION HANDLING**) will remind you about class hierarchies.

Armed with this background, we can now go about implementing inheritance in our own normal classes.

STEP 25: A CHILD CLASS INHERITING FROM ITS PARENT CLASS

The simplest thing we can do is to create a child class that inherits from its parent class. We have already done something similar several times before – eg in creating **Inert Class** from **Entity**. (In fact this might be a good time to look through **STEPS 7ff**, to understand the process from the point of view of inheritance.) We will now go one step further, to create a **Parent Class** (that inherits from **Entity**), and a **Child Class** (that inherits from **Parent**). We can picture our task as on the following page:



Here is the minimal class definition that will mediate this functionality:

```
-- ParentClass v1.0

global constant Parent = class("Parent", Entity)
    -- other code can go here
end_class()
```

```
-- ChildClass v1.0

global constant Child = class("Child", Parent)
    -- other code can go here
end_class()
```

And here is an application file that will use the inherited automatic constructor and destructor to create and then decommission an instance of **Child**:

```
-- ChildDemo.ex v1.0

include diamondlite.e -- or in our case: DL.e
include ParentClass.e
include ChildClass.e

procedure main()
    entity myChild
    myChild = call_method(Child, "new", NONE)
end procedure

main()
```

Run the application with **DL.e**, and confirm that we have created the class **Parent** (whose handle is **{4,0,M}**), the class **Child** (whose handle is **{5,0,M}**), and the entity **myChild** (with handle **{5,2,M}**). Notice that the superclass is created first, and then the subclass. Notice the syntax that makes inheritance possible (and recall that we used a similar approach when we defined our exception classes):

```
global constant Parent = class("Parent", Entity) -- Parent inherits from Entity
```

```
global constant Child = class("Child", Parent) -- Child inherits from Parent
```

Parent has inherited all the methods of **Entity**, and **Child** stands to inherit all the properties and methods available to **Parent**. Note that we haven't created a new instance of **Parent** – only of **Child** (viz **myChild**)

STEP 25a: A CHILD CLASS INHERITING ITS PARENT'S PROPERTY

The very next thing we can do is to add a property to **Parent** and code an accessor method with which to get it:

```
-- ParentClass v1.1

global constant Parent = class("Parent", Entity)
    property("parentAge", INSTANCE, 100)

    function Parent_getAge_0()
        return get_property(this(), "parentAge")
    end function
    method("getAge", 0, INSTANCE, routine_id("Parent_getAge_0"))
end_class()
```

We can keep **Child** just as it is, and modify **ChildDemo.ex** to display the property's value:

```
-- ChildDemo.ex v1.1

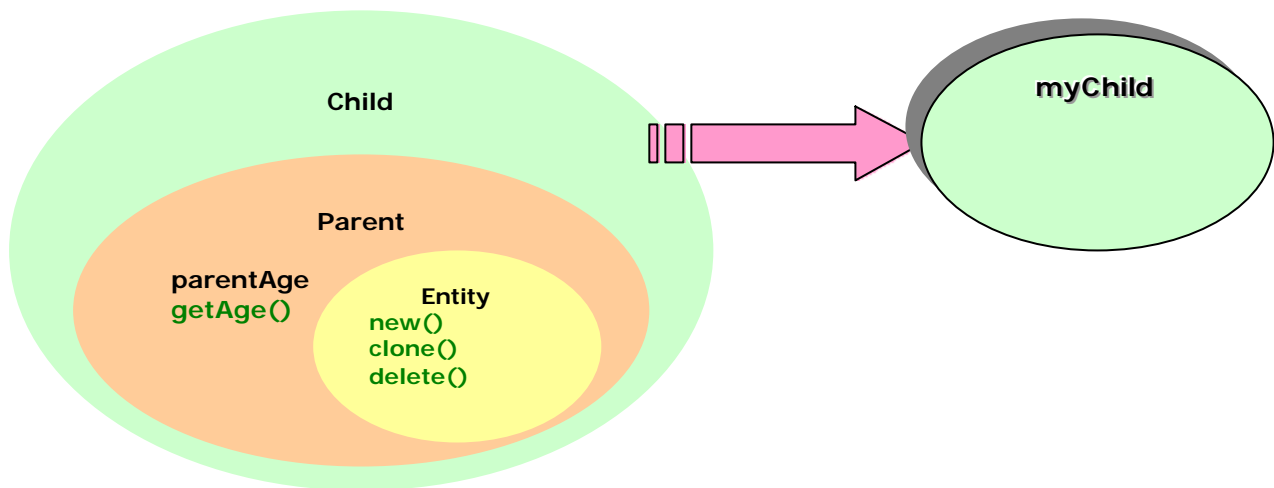
include DL.e
include ParentClass.e
include ChildClass.e

procedure main()
    entity myChild

    myChild = call_method(Child, "new", NONE)
    printf(1, "\nEX: child's parent's age is %d",
        call_method(myChild, "getAge", NONE))
end procedure

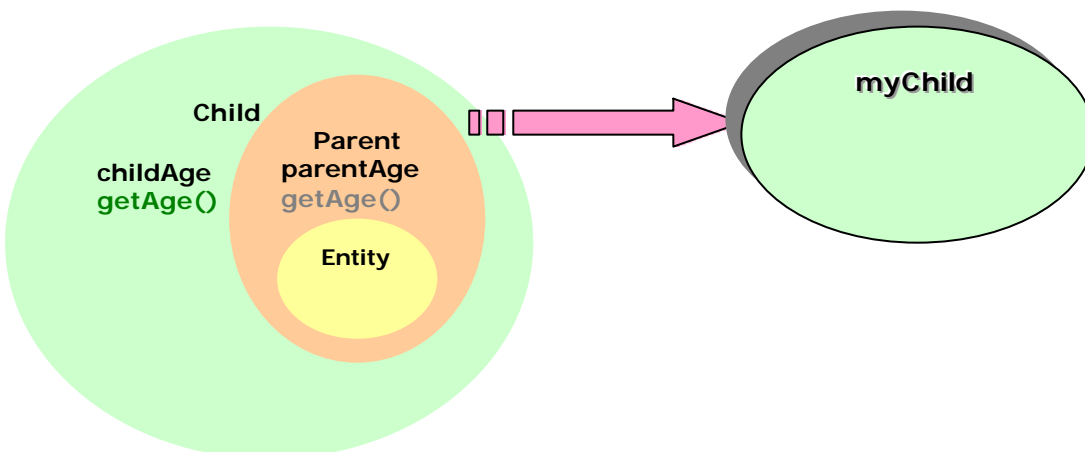
main()
```

When we run this application we are be able to confirm that by calling **getAge()** on **myChild**, we have been able to access the value of the property in **Parent** – ie, we have achieved the purpose that would have been achieved by calling **getAge()** on an instance of **Parent**, *without explicitly creating that instance*. Speaking loosely, it's as if we've been able to "get to" something in **Parent**, "through" **Child**. (This is what inheritance permits us to do, that we couldn't do using *composition*, which we studied earlier.) We can represent our current position as on the following page:



STEP 25b: A METHOD IN CHILD CLASS OVERRIDING A METHOD IN PARENT CLASS

We can now go one step further. Assuming that **Child** inherits from **Parent**, how would we go about coding a situation in which class **Child** had a property (eg **childAge**) similar to that of **Parent** (ie **parentAge**), and we needed to access **childAge** by calling method **getAge()**? In other words we are looking to override **Parent.getAge()** with **Child.getAge()** – ie:



The solution will involve *overriding* one method by another, and it will be achieved by coding two methods with the same signature. Now might be a good time to read through the section called **OVERRIDING METHODS** (after **STEP's 7 and 8**), to remind yourself about the basic ideas.

In our present case we would keep **ParentClass.e** just as it is (ie **v1.1**), and modify **ChildClass.e** as follows:

```
-- ChildClass v1.1

global constant Child = class("Child", Parent)
  property("childAge", INSTANCE, 50)

  function Child_getAge_0()
    return get_property(this, "childAge")
  end function
  method("getAge", 0, INSTANCE, routine_id("Child_getAge_0"))
end_class()
```

We would then alter the application file as follows:

```
-- ChildDemo.ex v1.2

include diamondlite.e -- in our case: DL.e
include ParentClass.e
include ChildClass.e

procedure main()
    entity myChild

    myChild = call_method(Child, "new", NONE)
    printf(1, "\nEX: child's age is %d",
            call_method(myChild, "getAge", NONE))
end procedure

main()
```

Confirm that **ChildDemo.ex v1.1** and **v1.2** are (essentially) identical – they just return a different value because of overriding of the method **getAge()**.

You might be wondering how we would access the values of both properties in the one application. We already know how to create a new entity of each class separately (ie **myParent** and **myChild**), and how to call **myParent.getAge()** and **myChild.getAge()** independently – well, we would just do that! In other words:

```
-- ChildDemo.ex v1.3

include diamondlite.e -- in our case: DL.e
include ParentClass.e
include ChildClass.e

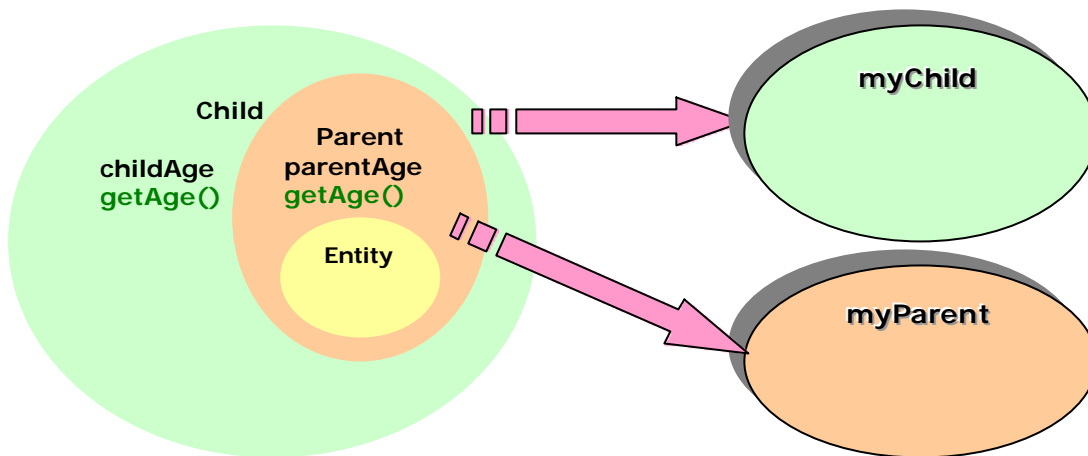
procedure main()
    entity myParent, myChild

    myParent = call_method(Parent, "new", NONE)
    myChild = call_method(Child, "new", NONE)

    printf(1, "\nEX: parent's age is %d",
            call_method(myParent, "getAge", NONE))
    printf(1, "\nEX: child's age is %d",
            call_method(myChild, "getAge", NONE))
end procedure

main()
```

Run the application to confirm that we have created two separate entities (they have different handles), and that we can specify which **getAge()** method we want to use by specifying its target. We can represent what we've done as on the following page:



STEP 25c: INHERITANCE INVOLVING PARAMETERISED CONSTRUCTORS

So far each of our classes has had a property whose value was hard-coded in the class definition. We would like to have the option of creating entities in which we could set these properties ourselves – either hard-coded within the application, or by way of user input. To achieve this, we will need to use *parameterised constructors*. (Now might be a good time to revise **STEP 12**, where we developed **GreetingClass**.)

First we would need to change **Parent** as follows:

```
-- ParentClass v1.2

global constant Parent = class("Parent", Entity)
    property("parentAge", INSTANCE, NIL)

    function Parent_new_1 (integer p_age)
        entity newParent

        newParent = call_method(super(), "new", NONE)
        set_property(newParent, "parentAge", p_age)
        return newParent
    end function
    method("new", 1, CLASS, routine_id("Parent_new_1"))

    function Parent_getAge_0()
        return get_property(this(), "parentAge")
    end function
    method("getAge", 0, INSTANCE, routine_id("Parent_getAge_0"))
end_class()
```

And then we would need to change **Child** accordingly, like this:

```
-- ChildClass v1.2

global constant Child = class("Child", Parent)
    property("childAge", INSTANCE, NIL)

    function Child_new_2 (integer p_age, integer c_age)
        entity newChild

        newChild = call_method(super(), "new", {p_age})
    end function
    method("new", 2, CLASS, routine_id("Child_new_2"))

    function Child_getAge_0()
        return get_property(this(), "childAge")
    end function
    method("getAge", 0, INSTANCE, routine_id("Child_getAge_0"))
end_class()
```

```

        set_property(newChild, "childAge", c_age)
    return newChild
end function
method("new", 2, CLASS, routine_id("Child_new_2"))

function Child_getAge_0()
    return get_property(this(), "childAge")
end function
method("getAge", 0, INSTANCE, routine_id("Child_getAge_0"))
end_class()

```

Notice the line `newChild = call_method(super(), "new", {p_age})`. It means the following: "Call the method `new()` on the current entity's (ie **Child**) superclass (ie **Parent**), passing as argument the (integer) value `p_age`, and when you've done all that return the reference to the new entity (**newChild**).". In executing this statement, control will pass to the function identified as **Parent new 1** (which takes one integer parameter), where the following will be executed: `newParent = call_method(super(), "new", NONE)`. This will mean the following: "Call the method `new()` on the current entity's (ie **Parent**) superclass (ie **Entity**), passing no argument, and when you've done all that return a reference to the new entity (**newParent**).". In addition before returning from each constructor in turn, each function will see to it that the value of the property is set to the value of the corresponding argument.

To see how this will work, let's modify the application file accordingly:

```

-- ChildDemo.ex v1.4

include diamondlite.e -- in our case: DL.e
include ParentClass.e
include ChildClass.e

procedure main()
    entity myParent, myChild

    myParent = call_method(Parent, "new", NONE)
    myChild = call_method(Child, "new", {100, 50})

    printf(1, "\nEX: parent's age is %d",
            call_method(myParent, "getAge", NONE))
    printf(1, "\nEX: child's age is %d",
            call_method(myChild, "getAge", NONE))
end procedure

main()

```

Run the application with **DL.e**, trace the execution of the program, and look at the handles to the various entities. Notice that `myParent = call_method(Parent, "new", NONE)` calls the automatic default constructor, passing no arguments to it, and therefore creating an entity whose **parentAge** property is initially set at **0**. When the next statement is executed – a call to **Child.new()**, passing two arguments (the first [100] going to **Parent**'s property, and the second [50] going to **Child**'s property as described previously) – the new entity will have access to a new value for **Parent**'s property.

But be careful: notice that a call to `myParent.getAge()` doesn't yield a result of **100** – it returns **0**, the default value. This reminds us that **myParent** and **myChild** are two separate entities, even though the latter inherits from the same class as the former. So each entity has a different value for **parentAge** – **0** for **myParent.parentAge**,

and 100 for myChild. parentAge

What's more, we can't access that property via a call to **myChild.getAge()**, because this method overrides the similarly-named method in **myParent**. In order for us to have access to both properties from within an entity of the subclass, we will have to restructure the code in the class definition.

One obvious thing we can do is to give the getter methods different names. For example we can do this:

-- ParentClass v1.3

```
global constant Parent = class("Parent", Entity)
  property("parentAge", INSTANCE, NIL)

  function Parent_new_1(integer p_age)
    entity newParent

    newParent = call_method(super(), "new", NONE)
    set_property(newParent, "parentAge", p_age)
    return newParent
  end function
  method("new", 1, CLASS, routine_id("Parent_new_1"))

  -- change the name of the method to getParentAge()
  function Parent_getParentAge_0()
    return get_property(this(), "parentAge")
  end function
  method("getParentAge", 0, INSTANCE, routine_id("Parent_getParentAge_0"))
end_class()
```

-- ChildClass v1.3

```
global constant Child = class("Child", Parent)
  property("childAge", INSTANCE, NIL)

  function Child_new_2(integer p_age, integer c_age)
    entity newChild

    newChild = call_method(super(), "new", {p_age})
    set_property(newChild, "childAge", c_age)
    return newChild
  end function
  method("new", 2, CLASS, routine_id("Child_new_2"))

  -- change the name of the method to getChildAge()
  function Child_getChildAge_0()
    return get_property(this(), "childAge")
  end function
  method("getChildAge", 0, INSTANCE, routine_id("Child_getChildAge_0"))
end_class()
```

-- ChildDemo.ex v1.5

```
include diamondlite.e -- in our case: DL.e
include ParentClass.e
```

```

include ChildClass.e

procedure main()
    entity myParent, myChild

    myParent = call_method(Parent, "new", NONE)
    myChild  = call_method(Child, "new", { 100, 50 })

    puts(1, "\nEx: In myParent:")
    printf(1, "\nEX: parent's age is %d",
            call_method(myParent, "getParentAge", NONE))
    puts(1, "\n\nEx: In myChild:")
    printf(1, "\nEX: parent's age is %d",
            call_method(myChild, "getParentAge", NONE))
    printf(1, "\nEX: child's age is %d",
            call_method(myChild, "getChildAge", NONE))
end procedure

main()

```

An alternative is to create another additional method, **getAge#1** (ie **getAge(parameter)**), and make it responsible for "choosing" the correct age. We would keep **ParentClass v1.2**, and modify **ChildClass v1.2** as follows:

```

-- ChildClass v1.4

global constant Child = class("Child", Parent)
    property("childAge", INSTANCE, NIL)

    function Child_new_2(integer p_age, integer c_age)
        entity newChild

        newChild = call_method(super(), "new", {p_age})
        set_property(newChild, "childAge", c_age)
        return newChild
    end function
    method("new", 2, CLASS, routine_id("Child_new_2"))

    function Child_getAge_0()
        return get_property(this(), "childAge")
    end function
    method("getAge", 0, INSTANCE, routine_id("Child_getAge_0"))

    function Child_getAge_1(object whichMethod)
        if equal(whichMethod, "self") then
            return call_method(this(), "getAge", NONE)
        elsif equal(whichMethod, "parent") then
            return call_method(super(), "getAge", NONE)
        else
            fatal_error("\nFatal Error: invalid parameter")
        end if
    end function
    method("getAge", 1, INSTANCE, routine_id("Child_getAge_1"))
end_class()

```

Any program or method that calls **myChild.getAge()** will now have an option:

1. if it does not pass an argument to the method, then the **getAge#0** method will execute, and the child's age will be returned
2. if it does pass an argument to the method, then the **getAge#1** method will execute –
 - a. if the argument is "self", then this class' **getAge#0** method will be called, just like option 1 above
 - b. if the argument is "parent", then the superclass' (**Parent**) **getAge#0** method will be called instead

We can demonstrate how this can work by modifying **ChildDemo.ex** as follows:

```
-- ChildDemo.ex v1.6

include diamondlite.e -- in our case: DL.e
include ParentClass.e
include ChildClass.e

procedure main()
  entity myParent, myChild

  myParent = call_method(Parent, "new", NONE)
  myChild  = call_method(Child, "new", { 100, 50 })

  puts(1, "\nEx: In myParent:")
  printf(1, "\nEX: parent's age is %d",
          call_method(myParent, "getAge", NONE))

  puts(1, "\n\nEx: In myChild:")

  -- delete this – it is inapplicable
  printf(1, "\nEX: parent's age is %d",
          call_method(myChild, "getParentAge", NONE))

  -- keep this, with a modification
  printf(1, "\nEX: child's age is %d",
          call_method(myChild, "getAge", NONE))

  puts(1, "\n\nEx: Alternatively, using getAge#1 in myChild:")
  printf(1, "\nEX: parent's age is %d",
          call_method(myChild, "getAge", {"parent"}))
  printf(1, "\nEX: child's age is %d",
          call_method(myChild, "getAge", {"self"}))

end procedure

main()
```

Run the program with **diamondlite.e** to satisfy yourself that it works; if you need to trace through the program use **DL.e** instead.

STEP 25d: COMPLETING THE CLASSES – AN EXTENSION EXERCISE

You might like to take this opportunity to fill out these classes, as an extension exercise. By this stage you won't need much explanation from me, so I'll just sketch out some suggestions.

1 Add setters to the class definitions:

These would look something like this:

```
function Class_setAge_1(object age)
```

```

    set_property(this (), <"propertyName">, age)
    return NIL
end function
method("setAge", 1, INSTANCE, routine_id("Class_setAge_1"))

```

2 Add exceptions to the class definitions:

For example, you could create an exception class **BadInput**, and define it as follows:

```

global constant BadInput = exception("BadInput", Exception)

```

You could declare the parameter(s) as type **object** – eg: **foo(object age)**

In the appropriate methods (ie the constructors and the setters), you could test the input:

```

if not integer(age) then
    throw(BadInput)
else
    -- go ahead and create the entity or set the property
end if

```

3 Let the application get user input:

For example let the user enter a value for each age, and let the entity test it as above.

4 Let the application deal with erroneous input:

For example:

```

myChild = call_method(<Class>, <"new">, <{argument(s)}>)
if catch(BadInput) then
    -- deal with the error
else
    -- continue with the application
end if

```

POLYMORPHISM

Mention of *overriding* gives us an opportunity to say something about *polymorphism* since it is overriding that makes polymorphism possible.

As an example of *polymorphism*, consider the steering wheels of cars: they all present a standardised outward form, and they are all used in much the same way (you turn them; if you turn them clockwise the vehicle turns to the right; etc) – irrespective of whether their internal mechanism mediates manual steering, automatic steering, or rack-and-pinion steering. Note that you, the driver, don't have to know these internal details in order to steer each type of vehicle – once you know how to handle a steering wheel you can use the same skill in any of those vehicles, and obtain the same result. The correct steering mechanism comes into play on its own – in this case, it's determined by what's in the car.

So basically *polymorphism* refers to the concept and the practice of designing classes whose instances have methods with a standardised outward form (the *interface*), and where those methods may be implemented quite differently by different manifestations of the class: one ultimate form (the *interface*) – many methods, each with its own implementation details. The user of the class doesn't have to choose the correct implementation details – this is done "automatically" by the translator or interpreter, thereby reducing a lot of complexity.

Some languages (eg C++ and Java) have quite sophisticated support for polymorphism, and so does Diamond; DL reduces much of the complexity surrounding the topic, by supporting a very simple implementation of it.

We used it unknowingly in **STEP 25a**, where we were able to call `myChild.getAge()` and access `Parent.parentAge`, just as if we had called `myParent.getAge()` itself. As far as DL is concerned, if class **A** is the superclass of class **B**, then wherever it's permissible to use the class itself (eg `Parent`) or an instance of it (eg `myParent`), we may use the subclass (eg `Child`) or an instance of that (eg `myChild`) instead. Note that this capability doesn't necessarily go both ways, since the superclass might not know much about its subclasses.

In **STEP 25b** we glimpsed another sense of polymorphism – one interface; many methods – where `Child`'s interface included a method for *getting an age*, but it could do one of two things: get `parentAge`, or get `childAge`. Provided the code was there, the interpreter would automatically choose the correct `getAge()` method.

Even earlier – in **STEP 23b** – when we considered the function call `catch(Exception)`, where we asked the question: "Is this class, or any of its subclasses, the pending exception?", we were dealing with yet another expression of polymorphism.

Actually, we can take the concept a little further using DL. Imagine coding a generic `Shape` class, with properties such as `length`, `width`, `height`, `radius` etc, and the usual accessor methods. We could also incorporate "generic" methods `getArea()` and `getVolume()` into the class definition – we can think of them as part of the interface of the class. Then, when the class is inherited by classes defining specific shapes (eg `Triangle` or `Circle`), they can contain their own versions of `getArea()` and `getVolume()` that will override the interface ("generic") methods. The interpreter will choose the particular method to be called when the time comes.

CLASS HIERARCHIES

We are now able to use the knowledge we've gained above, to go to the next step – starting with a superclass, and inheriting it to define subclasses that have a logical, hierarchical relationship to one another, based on shared characteristics. Actually we have already been introduced to the idea – when we dealt with exceptions – but at that point we couldn't add properties or methods to those classes (because exception classes are defined that way in DL).

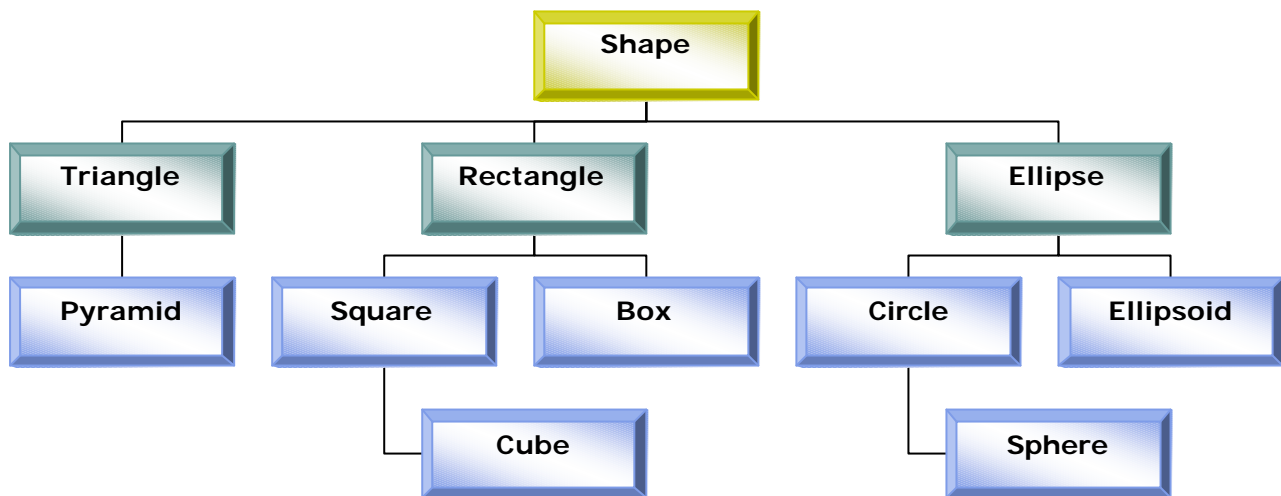
Let's assume that we want to define a base class called `Shape`, that will contain properties and methods that will allow us to calculate and display the `area` and `volume` of **2D** and **3D** shapes – such as `Rectangle`, `Square`, `Triangle`, `Pyramid`, `Ellipse`, `Circle`, `Box`, `Cube`, `Sphere`, `Ellipsoid` etc.

I will discuss how we might build up the various classes we need, what properties and methods we might give them, and how the classes might be inherited. This discussion will be based on ideas suggested by Michael Nelson, and will rely heavily on his code. My contribution will be in the method of presenting the material: instead of supplying all the code myself, I will outline an approach you might adopt and will leave it to you to fill in the details and make the classes fully functional – think of it as an extension exercise.

STEP 26: DEFINE THE CLASS HIERARCHY

Let's begin with the base class `Shape`, which will inherit from `Entity`. It will need to have all the properties – eg `length`, `width`, `height`, `depth`, `major` and `minor radius` – and methods with which to calculate and display areas (for 2D shapes) and volumes (for 3D shapes). It will have three subclasses – `Triangle`, `Rectangle`, and `Ellipse`. `Square` will be a subclass of `Rectangle`, and `Circle` will be a subclass of `Ellipse`. So much for 2D shapes. The 3D shapes will consist of `Pyramid` (from `Triangle`), `Cube` (from `Square`), `Box` (from `Rectangle`),

Sphere (from **Circle**), and **Ellipsoid** (from **Ellipse**). We can represent this arrangement as follows:



STEP 26a: AN OVERVIEW OF THE CLASS DEFINITIONS

The arrangement depicted above suggests that our class definitions will have the following general structure:

```
-- ShapeClass.e

global constant Shape = class("Shape", Entity)
    -- details of the class definition
end_class()
```

```
-- TriangleClass.e

global constant Triangle = class("Triangle", Shape)
    -- details of the class definition
end_class()
```

```
-- PyramidClass.e

global constant Pyramid = class("Pyramid", Triangle)
    -- details of the class definition
end_class()
```

```
-- RectangleClass.e

global constant Rectangle = class("Rectangle", Shape)
    -- details of the class definition
end_class()
```

```
-- BoxClass.e

global constant Box = class("Box", Rectangle)
```



```
-- details of the class definition
end_class()
```

-- SquareClass.e

```
global constant Square = class("Square", Rectangle)
-- details of the class definition
end_class()
```

-- CubeClass.e

```
global constant Cube = class("Cube", Square)
-- details of the class definition
end_class()
```

-- EllipseClass.e

```
global constant Ellipse = class("Ellipse", Shape)
-- details of the class definition
end_class()
```

-- EllipsoidClass.e

```
global constant Ellipsoid = class("Ellipsoid", Ellipse)
-- details of the class definition
end_class()
```

-- CircleClass.e

```
global constant Circle = class("Circle", Ellipse)
-- details of the class definition
end_class()
```

-- SphereClass.e

```
global constant Sphere = class("Sphere", Circle)
-- details of the class definition
end_class()
```

STEP 26b: DEFINE THE BASE CLASS – SHAPE

Instead of detailing every line of code for this task, I will only outline the necessary steps – by now you will be quite capable of filling in the details yourself.

➤ The overall class definition will have the following components:

-- ShapeClass

```
global constant Shape = class("Shape", Entity)
-- 1 register the properties
-- 2 define a parameterised constructor
-- 3 define a setter method for each property
-- 4 define a getter method for each property
-- 5 define a generic method to calculate each value – area and volume
-- 6 define a function that will be shared by subsequent methods
```

```

-- 7  define a method to display each property; it will use the shared function
-- 8  define a method to display the area and the volume
end_class()

```

➤ Here are the properties we will probably need:

```

-- 1  register the properties
property("length", INSTANCE, NIL)
property("width", INSTANCE, NIL)
property("height", INSTANCE, NIL)
property("depth", INSTANCE, NIL)
property("radius", INSTANCE, NIL)
property("majorRadius", INSTANCE, NIL)
property("minorRadius", INSTANCE, NIL)

```

➤ Here is a parameterised constructor (the major and minor radii are relevant to calculations pertaining to ellipses):

```

-- 2  define a parameterised constructor
function Shape_new_7(atom L, atom W, atom H, atom D, atom R,
                    atom majR, atom minR)
    entity newShape
    newShape = call_method(super(), "new", NONE)
    set_property(newShape, "length", L)
    set_property(newShape, "width", W)
    set_property(newShape, "height", H)
    set_property(newShape, "depth", D)
    set_property(newShape, "radius", R)
    set_property(newShape, "majorRadius", majR)
    set_property(newShape, "minorRadius", minR)
    return newShape
end function
method("new", 7, CLASS, routine_id("Shape_new_7"))

```

➤ Define a setter for each property – no surprises here (as an exercise, you might consider checking the parameter to ensure it is an atom, and throwing an exception if it isn't):

```

-- 3  define a setter method for each property
function Shape_<setProperty>_1(atom <param>)
    set_property(this(), <"propertyName">, <param>)
    return NIL
end function
method(<"setProperty">, 1, INSTANCE, routine_id("Shape_<setProperty>_1"))

```

➤ And define a getter for each property – again, no surprises:

```

-- 4  define a getter method for each property
function Shape_<getProperty>_0()
    return get_property(this(), <"propertyName">)
end function
method(<"getProperty">, 0, INSTANCE, routine_id("Shape_<getProperty>_0"))

```

➤ Define "generic" methods for area and volume. In this class, they do nothing and return **NIL** – but remember they will be overridden by the appropriate method according to the particular class that's called:

```
-- 5  define "generic" or "polymorphic" methods to calculate area and volume
function Shape_getArea_0()
    return NIL
end function
method("getArea",0, INSTANCE,routine_id("Shape_getArea_0"))

function Shape_getVolume_0()
    return NIL
end function
method("getVolume",0, INSTANCE,routine_id("Shape_getVolume_0"))
```

- This is a function – not a method – that will be shared by various methods further along:

```
-- 6  define a function that will be shared by subsequent methods
function Shape_show(sequence prprty)
    atom its_value
    sequence its_name
    its_name = lower(class_name(this))
    its_value = get_property(this, prprty)
    printf(1, "\nThe %s's %s is %.2f", {its_name, prprty, its_value})
    return NIL
end function
```

- These methods will call the shared function, passing the name of the property, in order to display the class name, the property name, and the property value:

```
-- 7  define a method to display each property; it will use the shared Shape_show() function
function Shape_<showProperty>_0()
    return Shape_show(<"propertyName">)
end function
method(<"showProperty",0, INSTANCE,
      routine_id("Shape_<showProperty>_0"))
```

- And now for methods to display the (previously calculated) area and volume:

```
-- 8  define methods to show the area and volume of the particular shape

function Shape_showArea()
    sequence its_name
    its_name = lower(class_name(this))
    printf(1, "\nThe %s's area is %.2f",
           {its_name, call_method(this, getArea, NONE)})
    return NIL
end function
method("showArea, 0, INSTANCE, routine_id("Shape_showArea"))

function Shape_showVolume()
    sequence its_name
    its_name = lower(class_name(this))
    printf(1, "\nThe %s's volume is %.2f",
           {its_name, call_method(this, getVolume, NONE)})
    return NIL
end function
method("showVolume, 0, INSTANCE, routine_id("Shape_showVolume"))
```

STEP 26c: DEFINE THE SUBCLASSES – eg RECTANGLE

I won't go through each and every subclass, but will discuss the issues that need to be addressed in **Rectangle** (and its subclass **Square**), and you will be able to apply the same concepts in writing the complete code for the other classes.

- The first thing to get clear is the syntax for defining one class as a subclass of another. We discussed this earlier – use `class("subclassName", superclassName)` to return a reference to the new class.
- The second step is to address the construction of the subclass. As it stands, our **Shape** constructor takes 7 parameters. In constructing an instance of **Rectangle**, for example, we will want to give values to only two of them – length and width. One approach might be to code the constructor as follows:

```
-- RectangleClass.e

global constant Rectangle = class("Rectangle", Shape)
  function Rectangle_new_0()
    entity newRectangle
      -- get a value for length
      -- get a value for width
      newRectangle = call_method(super(), "new",
                                {length, width, NIL, NIL, NIL, NIL, NIL})
    return newRectangle
  end function
  method("new", 0, CLASS, routine_id("Rectangle_new_0"))

  -- other method definitions
end_class()
```

When you then call this constructor in the application, like this....

```
procedure main()
  entity myRectangle
  myRectangle = call_method(Rectangle, "new", NONE)

  -- other method calls
end procedure
```

....**Rectangle.new()** will call its superclass' parameterised constructor

Shape.new(L, W, H, D, R, majR, minR)

and assign **length** and **width** to L and H respectively, and **NIL** to each of the other parameters. In the case of the other classes inheriting from **Shape**, we can use similar syntax:

```
newTriangle = call_method(super(), "new",
                          {length, NIL, height, NIL, NIL, NIL, NIL})
```

```
newEllipse = call_method(super(), "new",
                         {NIL, NIL, NIL, NIL, NIL, majorRadius, minorRadius})
```

```
newRectangle = call_method(super(), "new",
                          {length, width, NIL, NIL, NIL, NIL, NIL})
```

- Now for classes inheriting from one of the above classes, eg **Square** (from **Rectangle**), we can code the constructor like this:

```
-- SquareClass.e

global constant Square = class("Square", Rectangle)
  function Square_new_0()
    entity newSquare
    -- get a value for length
    newSquare = call_method(super(), "new",
                           {length, NIL, NIL, NIL, NIL, NIL, NIL})

    return newSquare
  end function
  method("new", 0, CLASS, routine_id("Square_new_0"))

  -- other method definitions
end_class()
```

And following the same pattern, we can expect the constructor for **Box** to contain a statement such as this:

```
newBox = call_method(super(), "new",
                    {length, width, NIL, depth, NIL, NIL, NIL})
```

- Perhaps the next step should be to override the setters of properties that aren't relevant to the particular class in question. For example we'll want to be sure that when we create an object of **Rectangle** there will be no way of setting a value for the properties **depth**, **radius**, **majorRadius**, **minorRadius**. We can do this as follows:

```
function Rectangle_<setProperty>_1(atom <param>)
  set_property(this(), <"propertyName">, <param>)
  return NIL
end function
method(<"setProperty">, 1, INSTANCE, NULL_CLASS)
```

Similarly the class **Triangle** will need to ensure that its caller cannot set values for **width**, **depth**, or any of the **radii**. The class **Square** will need to ensure that **length** and **width** will always have the same value – no matter which setter is used (**setLength()** or **setWidth()**). And **Ellipsoid** will need to ensure that only the setters for the two **radii** are used – the others are rendered inactive or overridden.

- Another important task will be to code the methods that will calculate the area (for 2D shapes) and volume (for 3D shapes). For instance in the case of **Rectangle**, we will need to code something like this to override **Shape.getArea()**:

```
function Rectangle_getArea_0()
  atom area
  area = get_property(this(), "length") *
        get_property(this(), "width")
  return area
end function
method("getArea", 0, INSTANCE, routine_id("Rectangle_getArea_0"))
```

Similarly to calculate the volume of a **Box** entity, we would need to code a method such as:

```
function Box_getVolume_0()
```

```

    atom volume
    volume = get_property(this(), "length") *
              call_method(super(), "getArea", NONE)
    return volume
end function
method("getVolume", 0, INSTANCE, routine_id("Box_getVolume_0"))

```

In order to complete these class definitions fully, you will have to know the formulae for the areas and volumes of the various shapes. The comments outlined above should be sufficient to get you started – my aim has been to focus on a general approach to coding inherited classes, rather than to emphasise arithmetic details.

STEP 26d: WRITE THE APPLICATION FILE

There are no surprises here – you will need to create new instances of each class that you need, supplying the correct property values and extracting the corresponding areas or volumes, by calling the appropriate methods to achieve the task. I'm sure that you've learnt the basics so well that you can see how to continue from here yourself.

A FINAL CLASS JUST FOR FUN – SELF-AWARE CLASS

I couldn't help including the following class definition – for your entertainment and as another extension exercise. It was written by Michael Nelson in response to my question: "Can a class be aware of itself?" The following code models a simplified form of self-awareness:

```

-- SelfAwareClass.e v1.0

include diamondlite.e

global constant SelfAware = class("SelfAware", Entity)
  property("me", INSTANCE, Null_Instance)
  property("myClass", INSTANCE, SelfAware)
  property("mySuperclass", INSTANCE, Entity)
  property("myMethods", INSTANCE, {"getMe#0", "getMyClass#0",
    "getMySuperclass#0", "getMyMethods#0", "getMyProperties#0"})
  property("myProperties", INSTANCE, {"me", "myClass", "mySuperclass",
    "myMethods", "myProperties"})

  function SelfAware_new_0()
    entity newSelfAware
    newSelfAware = call_method(super(), "new", NONE)
    set_property(newSelfAware, "me", newSelfAware)
    return newSelfAware
  end function
  method("new", 0, CLASS, routine_id("SelfAware_new_0"))

  function SelfAware_getMe_0()
    return get_property(this(), "me")
  end function
  method("getMe", 0, INSTANCE, routine_id("SelfAware_getMe_0"))

  function SelfAware_getMyClass_0()
    return get_property(this(), "myClass")
  end function
  method("getMyClass", 0, INSTANCE,
    routine_id("SelfAware_getMyClass_0"))

```

```

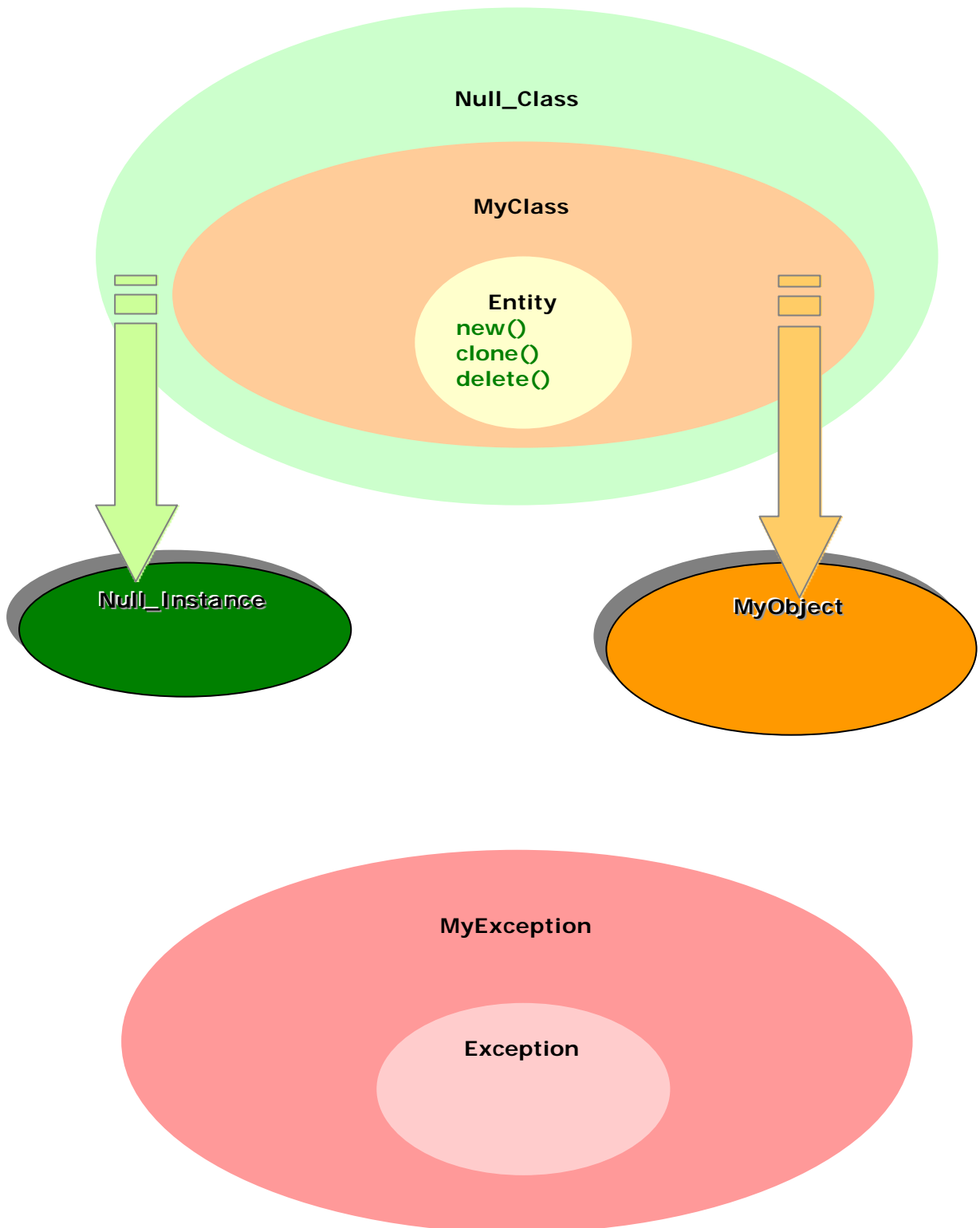
function SelfAware_getMySuperclass_0 ()
    return get_property(this 0, "mySuperclass")
end function
method("getMySuperclass", 0, INSTANCE,
    routine_id("SelfAware_getMySuperclass_0"))

function SelfAware_getMyMethods_0 ()
    return get_property(this 0, "myMethods")
end function
method("getMyMethods", 0, INSTANCE,
    routine_id("SelfAware_getMyMethods_0"))

function SelfAware_getMyProperties_0 ()
    return get_property(this 0, "myProperties")
end function
method("getMyProperties", 0, INSTANCE,
    routine_id("SelfAware_getMyProperties_0"))
end_class 0

```

APPENDIX A: DL's CLASS SYSTEM



APPENDIX B: DL ROUTINES BY PROGRAM CONTEXT

ROUTINE	MAIN PROGRAM	CLASS DEFINITION	CLASS METHOD	INSTANCE METHOD
<code>call_method()</code>	*		*	*
<code>catch()</code>	*		*	*
<code>caught()</code>	*		*	*
<code>class()</code>	*			
<code>class_entity()</code>	*	*	*	*
<code>class_name()</code>	*	*	*	*
<code>deleted_instance()</code>	*	*	*	*
<code>end_class()</code>		*		
<code>entity()</code>	*	*	*	*
<code>error_screen_width()</code>	*	*	*	*
<code>exception()</code>	*			
<code>extends()</code>	*	*	*	*
<code>fatal_error()</code>	*	*	*	*
<code>get_class()</code>	*	*	*	*
<code>get_property()</code>			*	*
<code>identifier()</code>	*	*	*	*
<code>instance_entity()</code>	*	*	*	*
<code>method()</code>		*		
<code>property()</code>		*		
<code>same_class()</code>	*	*	*	*
<code>set_property()</code>			*	*
<code>success()</code>	*		*	*
<code>super()</code>			*	*
<code>this()</code>			*	*
<code>this_class()</code>			*	*
<code>throw()</code>	*		*	*

APPENDIX C: DL CONSTANTS

CONSTANT	MEANING
TRUE	integer 1 used as a boolean
FALSE	integer 0 used as a boolean
NIL	integer 0 used as no meaningful data
NONE	an empty sequence
INSTANCE	indicates an instance property or method
CLASS	indicates a class property or method
NULL_METHOD	a do-nothing method used in place of routine_id(); returns NIL

APPENDIX D: DL VARIABLE

VARIABLE	MEANING
VOID	to discard an unwanted or meaningless return value

APPENDIX E: DL ROUTINES THAT TEST FOR TYPES

ROUTINE	MEANING
<code>identifier(o)</code>	TRUE if o is a valid Eu identifier (string)
<code>entity(o)</code>	TRUE if o is an entity
<code>instance_entity(o)</code>	TRUE if o is an instance entity
<code>class_entity(o)</code>	TRUE if o is a class entity
<code>deleted_instance(o)</code>	TRUE if o is an instance entity that has been deleted

APPENDIX F: DL ROUTINES THAT RELATE TO CLASSES

ROUTINE	MEANING
<code>class()</code>	begin class definition; return handle of new class
<code>end_class()</code>	end class definition
<code>class_name()</code>	return name of an entity's class
<code>get_class()</code>	return handle of an (instance) entity's class
<code>this_class()</code>	return handle of currently executing (instance) method's class
<code>same_class()</code>	TRUE if two entities belong to same class
<code>extends()</code>	TRUE if two entities have same class OR... first entity's class is a subclass of second entity's class

APPENDIX G: DL ROUTINES THAT RELATE TO PROPERTIES

ROUTINE	MEANING
<code>property()</code>	register an instance or class property and set its default value
<code>set_property()</code>	assign a value to an instance or class property
<code>get_property()</code>	return the value of an instance or class property

APPENDIX H: DL ROUTINES THAT RELATE TO METHODS

ROUTINE	MEANING
<code>method()</code>	register an instance or class method
<code>call_method()</code>	call an instance or class method and return its value
<code>this()</code>	return the handle to the target of an instance or class method
<code>super()</code>	call the overridden superclass method of this target's instance or class (overriding) method; return an integer constant

APPENDIX I: DL ROUTINES THAT RELATE TO ERROR HANDLING

ROUTINE	MEANING
<code>exception()</code>	return handle of new exception
<code>throw()</code>	set the pending exception
<code>catch()</code>	TRUE if an entity is the class or superclass of a pending exception
<code>caught()</code>	return the last exception processed by <code>catch()</code>
<code>fatal_error()</code>	immediately terminate program with an error message
<code>error_screen_width()</code>	set the width of the error screen
<code>success()</code>	TRUE if no exception pending

APPENDIX J: DL FATAL ERROR MESSAGES

*** Invalid error message. ***

The error message you used in `fatal_error()` is not a valid character string

Not allowed.

The DL routine you called is not allowed in this program context

Invalid class name.

The name you have given to your class is invalid

Invalid method name.

The name you have given to your method is invalid

Invalid property name.

The name you have given to your property is invalid

Invalid exception name.

The name you have given to your exception is invalid

Invalid exception.

This is not a valid exception

Invalid superclass.

This is not a valid superclass of a class you have declared

Method `MethodName` has invalid entity type.

The identified method is not of the correct type (ie instance vs class)

Method `MethodName` has invalid parameter count.

The identified method has the wrong number of parameters

Property `PropertyName` has invalid entity type.

The identified property is not of the correct type (ie instance vs class)

Instance property `PropertyName` has already been defined.

The identified instance property has already been defined

Class property `PropertyName` has already been defined.

The identified class property has already been defined

Target is not an entity.

The target of your method call is not an entity

Target is a deleted instance.

The target of your method call is an instance that has already been deleted

`ClassType` class `ClassName` not allowed as target.

It is not permissible for the identified class, of this type, to be the target of your method call

Class `ClassName` does not define `instance | class` property `PropertyName`.

The identified class does not define the instance or class property you have identified

Access to **ClassName** instance | class property **PropertyName** denied.

You cannot have access to the identified instance or class property of this class

Parameter list must be a sequence.

The parameter list you coded must be a sequence

Class **ClassName** does not define instance | class **MethodName#N**.

The identified class does not define the identified instance or class method that takes N parameters

Superclass **SuperclassName** is not a normal class.

The identified superclass is not a normal class

Superclass **SuperclassName** is not an exception.

The identified superclass is not an exception

Attempt to override **ClassName** instance property **PropertyName**.

You have tried to override the identified instance property of class

Attempt to override **Classname** class property **PropertyName**.

You have tried to override the identified class property of class

instance | class method **MethodName** has invalid routine_id.

The identified instance or class method has an invalid routine_id() value

instance | class method **MethodName** has already been defined.

The identified instance or class method has already been defined

Class **ClassName** is not an exception.

The identified class not an exception

Exception **ExceptionName** thrown.

The identified exception has already been thrown

With **ClassName** exception pending | caught
in **MethodName** | main program | class definition
called from call chain

An exception has been caught or is pending in the identified method, main program, or class definition; the call chain is given if there is one