The Object REXX Class Hierarchy

Simon Nash
IBM

The Object REXX Class Hierarchy

Simon C. Nash
IBM UK Laboratories Ltd, Hursley Park,
Winchester, Hants SO21 2JN, England
Internet: nash@vnet.ibm.com

Abstract

Object REXX, an object-oriented extension of the popular REXX language, includes a class hierarchy. The design of this hierarchy posed some interesting challenges in providing mechanisms that would serve the needs of the base hierarchy together with probable user extensions to it. This paper presents the chosen design in the form of a tutorial introduction to the concepts and mechanisms involved, including abstract classes, mixins, and multiple inheritance. It also gives examples of how the mechanisms provided by REXX might be used by class users and implementers.

Objects and Classes

REXX objects are grouped into classes. For example, all character strings (whatever their content) belong to the String class, all directories belong to the Directory class, and so on. The class of an object indicates what "kind" of object it is — that is, what methods it provides to respond to messages sent to it. For example, string objects provide string-related methods such as POS and SUBSTR, and directory objects provide methods for collections such as ITEMS and SUPPLIER. You can look at the descriptions of the String and Directory classes to find out what methods are available on string and directory objects.

In REXX, everything is an object, so classes are objects too. Class objects are used in a number of ways, the most important of which is their role in creating other objects. They support this by providing NEW and INHERITED methods which create objects of the kind defined by the class. For example, the Directory class object returns a new directory object in response to the message

```
.directory~new
```

Classes and Instances

The objects created by a class are known as its instances. They are given methods that match the specification defined by the class for its instances. For example, a Rectangle class might define methods AREA and PERIMETER using the directives

```
::class Rectangle
::method area
expose width height
return width*height
::method perimeter
expose width height
return (width+height)*2
```

Then, when rectangle objects (instances of the Rectangle class) are created by sending NEW messages to the Rectangle class object, they will have methods AREA and PERIMETER with the REXX code shown above.

Object and Instance Methods

There's an important difference between the AREA and PERIMETER methods of a rectangle object and the AREA and PERIMETER method definitions in the Rectangle class. A rectangle object can respond to AREA and PERIMETER messages by running the methods shown above, and we say that it has these as *object methods.* The Rectangle class cannot respond to AREA and PERIMETER messages itself, but its instances can, and we say that it has AREA and PERIMETER as *instance methods.* The Rectangle class responds to other (class-related) messages, such as the NEW message that creates a rectangle object, so it has object methods (like NEW) as well as its instance methods.

Since only classes have instance methods, there's no need to distinguish between object methods and instance methods when talking about other kinds of objects, such as strings or rectangles. We usually just say plain "methods" when talking about the object methods of these objects.

Subclasses, Superclasses, and Inheritance

Every class in the system could be defined independently, with a complete set of instance methods. However, many classes have a lot in common. An example of this may be Student and Graduate classes — a graduate object has the same information as a student object (name, ID, course, etc.) and also some additional information (graduation details). We'd prefer not to repeat most of the instance methods of the Student class in the definition of the Graduate class, and we can avoid this (and express the close relationship between these two kinds of objects) by making the Graduate class a subclass of the Student class. This gives the Graduate class all the instance methods of the Student class, and the Graduate class can then add or override any necessary instance methods.

If Graduate is a subclass of Student, we call Student a superclass of Graduate. The subclass-superclass relationship is also called inheritance, so we say that Graduate inherits the NAME method from Student.

The inheritance relationship can be used to arrange the classes into a class hierarchy — a diagram in which superclasses are drawn above subclasses, with lines connecting them. The class at the top of this hierarchy is the Object class. Its instance methods (COPY, "=", STRING, etc.) are inherited (directly or indirectly) by all other classes and so become object methods of all objects. Most objects have additional object methods — for example, the Supplier class is a subclass of the Object class and has instance methods AVAILABLE, INDEX, ITEM, and NEXT, so all supplier objects have these object methods as well as COPY, "=", STRING, etc.

You can create a subclass by specifying the name of the superclass on the SUBCLASS option of the ::CLASS directive (which is equivalent to sending a SUBCLASS message to a class object). For example, to make Graduate a subclass of Student, you would write

```
::class Graduate subclass Student
```

If Student were itself a subclass of Person (inheriting some Person methods), this makes Grasuate a subclass of Person too (through the intermediate class Student). We sometimes use the terms direct and indirect superclasses (or subclasses) to distinguish these. If you don't specify the SUBCLASS option, your class becomes a subclass of the Object class.

When talking about a class's instance methods, which ones do we mean — just the ones it defines itself, or those and the ones it inherits from its superclasses? It's usually more convenient to take this as meaning the methods defined by the class itself, and we will follow this convention from now on. However, it's important to remember that when the class creates instances, the object methods of the instances include not only the instance methods of the class itself, but also those of all the superclasses from which it inherits.

Abstract Classes and Object Classes

Some classes have a close inheritance relationship, like Graduate and Student. Others are related in a slightly more distant way — more like siblings than parents and children. You can appreciate why the term "inheritance" is used to describe the class family! For example, array and list objects share a number of methods: FIRST, LAST, NEXT, PREVIOUS, SECTION, and SUPPLIER. Even so, neither is a subclass of the other — arrays have a DIMENSION method, but lists don't, and lists have a FIRSTITEM method, but arrays don't. So how can we express the common nature of arrays and lists?

The answer is an *abstract class*. Abstract classes are special classes that don't create instances (unlike "normal" classes, like Student and Graduate). Instead, they provide a set of instance method definitions that can be shared by a number of other classes. It's helpful if abstract classes define meaningful properties, with a collection of methods that relate to that property. For example, the property shared between arrays and lists is that of having some internal sequence which can be used to step through the items of the array or list — the idea of a "first" item, "next" item and so on. This leads naturally to the idea of a Sequenced class — but it's not a "normal" class, since it isn't meaningful to think about making instances of the Sequenced class. That's because the Sequenced class doesn't provide enough capability for a functional standalone "sequenced" object. Array and List inherit from Sequenced and add the missing pieces that Sequenced doesn't have.

We need a name for "normal" classes (that can create instances) to distinguish them from abstract classes. We call them *object classes* because these are the classes whose members (instances) are real live objects. Their names are usually nouns, such as Array, List, and Rectangle. In contrast, abstract classes define properties (or abstractions) that describe objects — they have no instances, and their names are usually adjectives like Sequenced.

Because of the way abstract classes factor out the common methods from their subclasses, you'd expect them to always have more than one subclass. This is true for all the abstract classes that REXX provides except the Condition and Supplier classes. These aren't object

classes because they don't provide NEW or ENHANCED methods for creating instances — REXX provides other ways to create condition objects and supplier objects. They're a very special case (since no user-created classes would be able to work like this), and it's convenient to use abstract classes for them.

To create an abstract class, use the ABSTRACT option on a ::CLASS directive. For example, to create an abstract class Visual which is a subclass of the Object class, you would write

```
::class Visual abstract
```

Multiple Inheritance

As well as sharing some methods with arrays, lists also share some methods with queues: MAKEARRAY, PEEK, PULL, PUSH and QUEUE. Again, it makes sense to create an abstract class for these. We call it Queuelike, since its instance methods apply to all objects that function as queues. So we need the List class to inherit from both the Sequenced and Queuelike abstract classes. This is called multiple inheritance, and although it may look quite simple (at least in this case), it is very powerful. It also raises some rather complicated issues — see More on Multiple Inheritance below.

You can use multiple inheritance by specifying the INHERIT option on a ::CLASS directive (which is equivalent to sending one or more INHERIT messages to a class object). For example, to create a class Window which is a subclass of Visual and also inherits from Movable and Sizeable, you would write

```
::class Window subclass Visual inherit Movable Sizeable
```

There's no limit to the number of classes you can inherit from in this way.

Class Methods

We've seen that class objects have both instance methods and object methods. How are their object methods (like NEW) defined? The CLASS option on a ::METHOD directive indicates that the method being defined is a *class method*, not an instance method. For an object class, this means that the class will have that method as one of its object methods. For example, the Array class defines OF as a class method, and this allows OF messages to be sent to the Array class object to create array objects whose contents are specified by the arguments to OF.

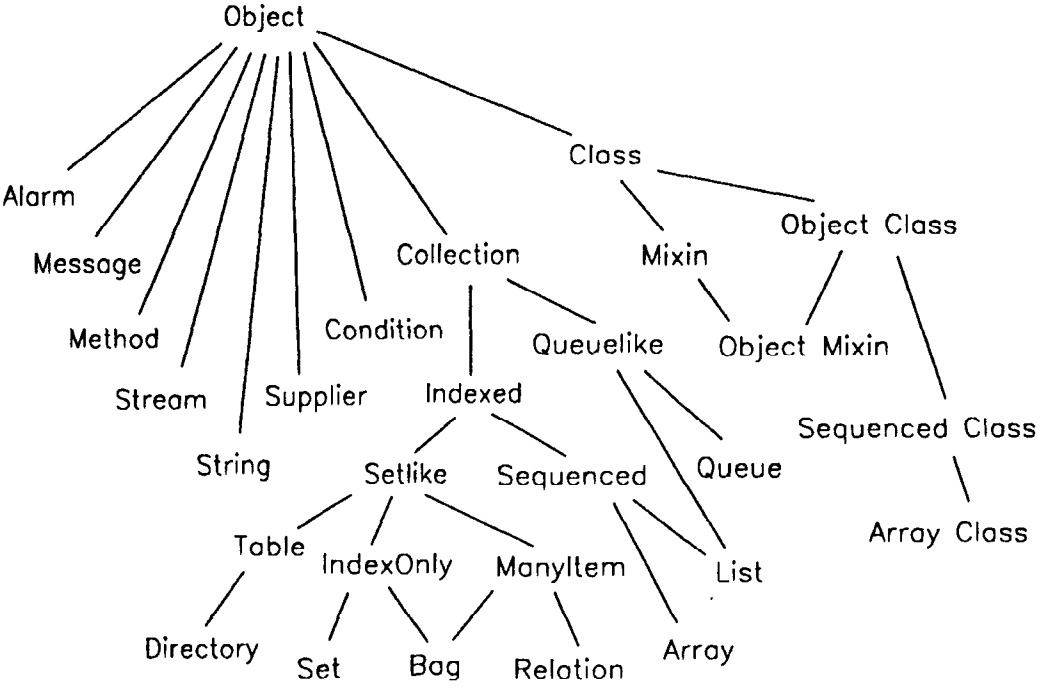
What about abstract classes — can they have class methods too? They can, but their class methods work slightly differently than those of object classes. They are defined in the same way, with the CLASS option on a ::METHOD directive, but they don't become object methods of the abstract class itself. Instead, they become object methods of any object classes that inherit (directly or indirectly) from the abstract class in which they are defined. For example, the Sequenced class also has an OF class method, but OF messages can't be sent to the Sequenced class to create "sequenced" objects (because the Sequenced class is abstract and so can't create objects). Instead, OF becomes an object method of any object classes that inherit from the Sequenced class, such as the List class. The List class doesn't have to do anything (except inherit from the Sequenced class) to make this happen.

So what object methods do abstract classes have? They all have the same ones: DEFINE, DELETE, ID, INHERIT, INITA, METHOD, METHODS, SUBCLASS, SUBCLASSES, SUPERCLASSES, and UNINHERIT. Of course, like all objects, their object methods include the instance methods of the Object class: COPY, "=", STRING, etc. Object classes have two additional object methods: NEW and ENHANCED, the methods that create objects.

Class methods are inherited in exactly the same way as instance methods. For example, the List class inherits the OF class method from the Sequenced class, just as it inherits the FIRST, LAST, MAKEARRAY, NEXT, PREVIOUS, SECTION, and SUPPLIER instance methods.

The Class Hierarchy

We've mentioned a number of REXX classes and the inheritance relationships between them. Let's take a look at the complete hierarchy for the classes provided and used by REXX.



That looks a bit daunting, but the REXX user doesn't have to be concerned with many of these classes. A number of them (Collection, Indexed, IndexOnly, ManyItem, Queuelike, Sequenced, and Setlike) are abstract classes used only for internal factoring out of common methods. In addition, the metaclass section of the hierarchy (Class, Mixin, Object Mixin, Object Class, Sequenced Class, and Array Class) is shown for completeness but isn't for general use (see the section on Metaclasses below). That leaves us with the classes that represent other objects: Alarm, Array, Bag, Condition, Directory, List, Message, Method, Object, Queue, Relation, Set, Stream, String, Supplier, and Table.

More on Multiple Inheritance

We used the List class to introduce multiple inheritance. The List class inherits from the Sequenced and Queuelike abstract classes, which means that lists have both the Sequenced and Queuelike properties (collections of methods). Another way of saying this is that a list can be used whenever either a queuelike or sequenced object is expected, and it will work correctly.

Multiple inheritance can be very useful if used properly, but it can cause a lot of problems when it's used incorrectly. This has made it quite controversial in object-oriented circles! A common mistake is to think of it as a "magic" way to combine two different objects into one — such as a hybrid of the Rectangle and List classes. That probably sounds rather ridiculous, but other examples can seem more plausible. For example, if I have classes Directory (which keeps names and some information for each name) and Phone (which dials a number passed to it), can't I create a Phone Directory class (which keeps names and phone numbers, and dials the number when given a name) simply by inheriting from Directory and Phone?

Unfortunately, it's not usually that simple. The reason is that Directory and Phone were each designed to do a specific job, with a set of methods appropriate to that job, but neither was designed (probably) to "mix in" with the other. For example, the DIAL method of Phone was designed to be given a number to dial, and multiple inheritance won't make it smart enough to change its behaviour to take a name instead and look it up in the directory. (By the way, the right way to do this is called aggregation, which means creating a new object that contains Directory and Phone and provides the right connections between them.)

So when is multiple inheritance useful? Actually, the answer's in the paragraph above. It's useful for classes that have been specially designed to "mix in" with other classes — like the Sequenced and Queuelike classes, which were specially designed to mix in with each other. However, these classes were not designed to mix more generally with other classes. You can try mixing them with other classes (REXX won't stop you), but it's unlikely that anything useful will result. For more general "mix in" classes, we'll have to look elsewhere in the hierarchy.

Mixins

A class that is designed to be mixed in with other classes in a general fashion is called a *mixin*. For example, the ManyItem mixin can be inherited by any setlike class to allow multiple items to have the same index, and is used by the Bag and Relation classes.

It's important to understand the difference between mixins and abstract classes. Both can be used with multiple inheritance, but their purposes are very different. Abstract classes are for the convenience of a class hierarchy implementer, to prevent the same methods being duplicated among more than one object class. They are of little use in themselves, but enable the construction of object classes below them in the hierarchy. They are not part of the public interface of the class hierarchy for inheritance.

A mixin, in the other hand, allows some class (and all the classes below it in the hierarchy) to be enhanced in some way. For example, if Persistent is a mixin to the Object class, all classes

in the hierarchy may exist in persistent and non-persistent versions. The persistent versions inherit the Persistent mixin, but the non-persistent versions don't. For example, to make a persistent directory class, you would write

```
::class PersistentDirectory subclass Directory inherit Persistent
```

which tells REXX that the PersistentDirectory class inherits from the Persistent mixin as well as the Directory class. Any number of mixins may be inherited, and a combination of inherited mixins and other inherited classes may be specified.

Since Persistent is a mixin to the Object class, it applies to all subclasses of the Object class — that is, all classes. Some mixins are more specialized — for example, the ManyItem mixin is a subclass of the Setlike class and so only applies to classes that inherit from the Setlike class. No other class (for example, a subclass of the Stream class) is allowed to inherit the ManyItem mixin. This is because the ManyItem mixin has been designed specifically to enhance the Setlike class (it's "tailor made" to fit this class only) and won't fit any other class. The Setlike class is called the base class of the ManyItem mixin, and we say that ManyItem is a mixin to the Setlike class.

So mixins, like object classes but unlike abstract classes, are intended for users of the class hierarchy and are part of its public interface for inheritance. They provide enhancing options for the object classes in the hierarchy, to be included or excluded at the user's discretion.

Some mixins provide a complete set of methods for some property, so that a class can acquire that property just by inheriting from the mixin. Other mixins define a property, and provide some of the methods required, but depend on subclasses to provide other necessary methods. For example, a Persistent mixin may provide methods that take care of saving object data to stable storage and restoring it when needed, but not the methods that actually extract the object's essential data (when saving) and recreate its state from saved data (when restoring). Those methods may be left as placeholders in the Persistent mixin, needing to be filled in by classes that inherit the persistent property from it, since only they know the intimate details of how they are constructed.

To create a mixin, use the MIXIN option on a ::CLASS directive. For example, to create a mixin OrderedSet which has a base class of Set, you would write

```
::class OrderedSet mixin subclass Set
```

The Class Search Order for Methods

In a single-inheritance hierarchy, classes inherit methods from their ancestors in the hierarchy. Since every class has exactly one superclass (except the root class Object, which has none), there is a simple line of inheritance from each class up to the root class Object, through any intermediate ancestor classes. This line of inheritance defines a search order for methods (the class search order). The order is important because more than one ancestor class may have an instance method with the same name — like PRINT. When a PRINT message arrives at an instance of the class, it's important to know which PRINT method will be run. The search order starts with the lowest class in the hierarchy (the class to which the instance that received the PRINT message belongs) and proceeds upwards to its superclass, then its superclass's

superclass, and so on up to the root class Object. The first PRINT method found is the one that gets run.

With multiple inheritance, the situation is quite a bit more complicated. Classes may have many superclasses (direct and indirect), and there may not be an obvious "right" order of searching them for a method. The rules REXX uses are:

1. A subclass is always searched before its superclasses.
2. Mixins are searched immediately before their base class.
3. Where multiple classes appear on the INHERIT option of the ::CLASS directive, the classes are searched in the order they appear (leftmost first).

If there is no search order that satisfies all these rules, or if a mixin is inherited without its base class already in the search order, the inheritance is in error.

What about multiple inheritance from object classes? It's this sort of thing that gave multiple inheritance a bad name. There are very few cases (if any) when it would be appropriate, but REXX doesn't prevent it – it doesn't seem right to limit the powers of object classes (compared with abstract classes) by making a special restriction here. Beware, though! Before doing this, you should see if your hierarchy can be restructured to make one of the superclasses an abstract class or a mixin, or consider whether aggregation (combining two objects into a composite object, as in the Phone Directory example) isn't a more appropriate way to accomplish what you want. It usually will be.

Metaclasses

For most users of Object REXX, the concepts and mechanisms presented so far will be all they need to create instances, subclasses, abstract classes, and mixins – making full use of the facilities that REXX provides for using and extending the class hierarchy. This section and the next one complete the picture for those who are curious to know more about how all this works, or need to understand or reprogram the underlying mechanisms of the class hierarchy.

Are class objects instances of some class? For completeness and consistency, it would be nice if they were. We call these special classes metaclasses. Their instances are classes, like the Supplier class and the Sequenced class. How many metaclasses are there? There could be one for each class (as in Smalltalk), but it's not necessary to go this far. However, we do need a metaclass for each class that has a different collection of class methods. To see why this is, let's look more closely at how class methods work.

The class methods of the Object class are NEW and ENHANCED. This means that they will be object methods of the Object class and every other object class that inherits from the Object class (that is, all object classes). A metaclass is needed to create these classes, and this metaclass needs NEW and ENHANCED instance methods so that its instances (the object classes) will have NEW and ENHANCED object methods. Let's call this class the Object Class metaclass.

Suppose we create a subclass of the Object class with another class method – for example, a Database class with a RESTORE class method to restore the previously saved state of an

object. There will have to be a new metaclass to create this class, since the Object Class metaclass doesn't have our RESTORE method. Let's call this new metaclass (with a RESTORE instance method) the Database Class metaclass. The Database class is an instance of the Database Class metaclass, and the RESTORE instance method of the Database Class metaclass becomes the RESTORE object method of the Database class.

How do these metaclasses fit into the hierarchy? As well as their specialized instance methods that correspond to the class methods of their instance classes, they have instance methods for all the standard object methods of classes: DEFINE, DELETE, ID, INHERIT, INITA, METHOD, METHODS, SUBCLASS, SUBCLASSES, SUPERCLASSES, and UNINHERIT. We need a class with these as its instance methods (they need to be instance methods somewhere) and we call this class the Class class. It's natural to make the Object Class metaclass and Database Class metaclass subclasses of the Class class, since they can then inherit all its instance methods listed above.

Which class is the metaclass for abstract classes? Their object methods are the ones that are shared by all classes: DEFINE, DELETE, etc. Since these are the instance methods of the Class class, the Class class is the metaclass for all abstract classes.

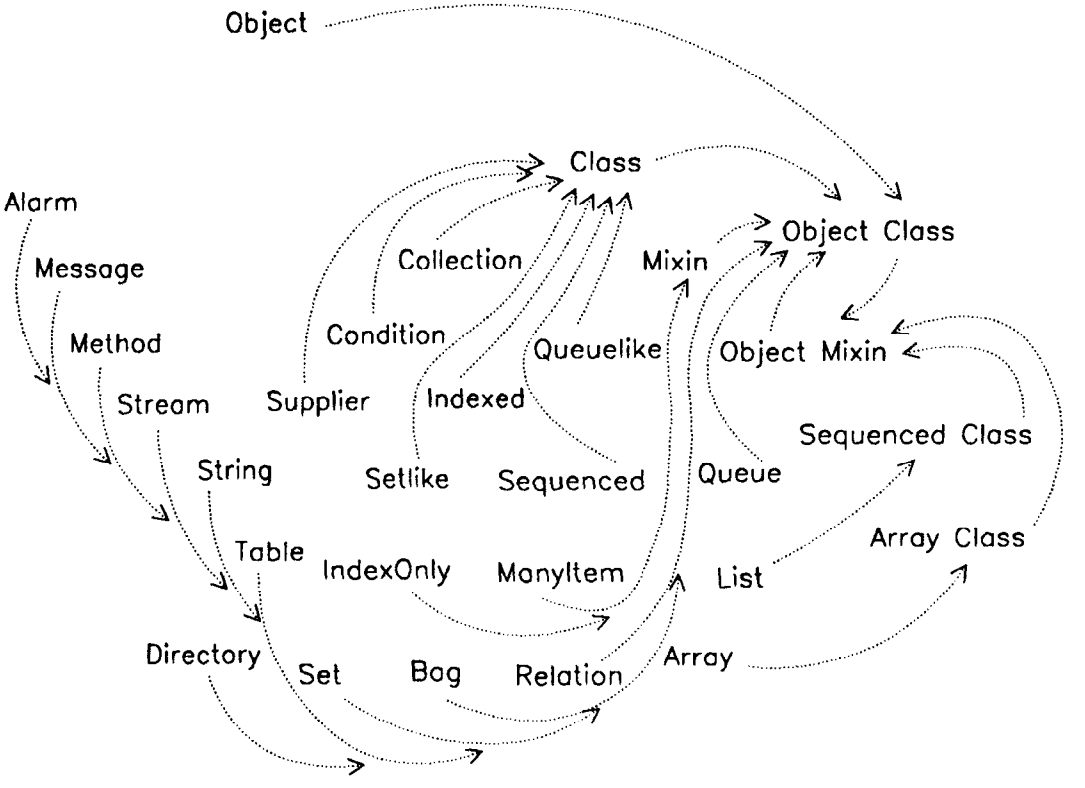
What about metaclasses for mixins? Mixins are very similar to classes, only differing in their inheritance rules, so we make the Mixin class (the class whose instance methods are object methods of all mixins) a subclass of the Class class. Is the Mixin class the metaclass for all mixins? It isn't, for the same reason that the Class class isn't the metaclass for all classes — just as different classes have different object methods, mixins do too.

Let's take an example to see why different mixins have different object methods. If we create a Relational mixin to our Database class, with instance methods but no class methods, what object methods does the Relational mixin have? They include all the standard mixin object methods (the instance methods of the Mixin class) as well as the inherited class methods: NEW, ENHANCED, and RESTORE. We want these as object methods because we want the Relational mixin (as a mixin to an object class, or an *object mixin*) to be able to create instances in its own right. If it couldn't, we'd have to create another object class (inheriting from Database and Relational) which could create these instances — adding an unnecessary class to the hierarchy.

It looks as though we might need a Relational Mixin metaclass to create the Relational mixin. In theory, we do; in practice, we don't. By making metaclasses mixins (with a base class of the Class class), they can also be inherited by subclasses of the Mixin class (since Mixin is a subclass of Class). So the Database Class metaclass becomes the Database Class mixin, and REXX can construct the Relational Mixin metaclass simply by inheriting from the Mixin class and the Database Class mixin. That's what mixins are all about!

Classes and Metaclasses

The earlier hierarchy diagram showed the superclass-subclass relationship of the REXX classes. It didn't show the class-instance relationships. Of course, these connections will be quite different, so it could be confusing to try to show both on the same diagram. We'll use a separate diagram here to show the class-instance relationships of these classes.



The classes are shown in the same positions as before, but the inheritance connections have been replaced by arrows which point from each class to its metaclass (from instance to class). There's an interesting circularity between the Object Class mixin and the Object Mixin class — each class is an instance of the other. This is a bit of a mindbender, and reminds me of the chicken and the egg question — how did these classes get created? Let's just say that someone had to do a little bit of cheating here.

Last Words

Don't worry if multiple inheritance, abstract classes, or mixins seem difficult or unnecessary. The simplest classes are the object classes. They create objects that do a particular job which is well-defined by their class definition. They are the place to start in familiarizing yourself with object-oriented programming, and in creating your own classes. Start by subclassing object classes, with single inheritance. Override a few methods and get a feel for how subclasses can be different from, yet similar to their superclasses. Then try multiple inheritance with a mixin, getting a feel for how that works. When you have developed a few object classes, you may start to notice relationships between them that don't match the

hierarchy — similar or identical methods cropping up in different places. That's the time to think about making use of abstract classes — to bring the relationships between your object classes into clearer focus.

This ongoing refinement of the class hierarchy is a hallmark of good object-oriented programming — seeing new relationships between your classes, and finding better ways to structure the hierarchy to express those relationships. Don't try to start out by designing a set of 20 abstract classes, 50 object classes, 15 mixins and all the relationships between them. You won't get it right at the first attempt! Far better to develop your hierarchy gradually, refining it as you acquire a feel through hands-on experience of how the classes relate to each other.

Summary

We have seen how object methods, instance methods, and class methods are used in Object REXX. The need for object classes, abstract classes, and mixins has been explained, together with guidelines for when they should be used and how they relate to single and multiple inheritance. The use of all the above facilities of the REXX language has been illustrated with examples from the class hierarchy provided by REXX. Finally, the role of metaclasses in completing the picture has been shown.

Acknowledgements

The main structure of the REXX class hierarchy was developed in a meeting of the REXX Architecture Review Board, with contributions from Jim Babka, Mike Cowlishaw, Brian Marks, Rick McGuire, and the other board members. The details took shape over several design iterations, with vital contributions and encouragement to continue from Jim Babka, Brian Marks, and Dave Renshaw.