CS225
Data Structures
Notes Packet #7
Inheritance and Virtual Functions

Jason Zych

Chapter 7

Inheritance and virtual functions

7.1 Inheritance

Inheritance allows you to declare one class, with certain member variables and functions, and then to declare a new class which is an extension of that class, and which can make use of the first class's variables and functions without having to redefine them.

```
class Coord  // this is the first class
{
public:
   double xCoord, yCoord;
   void Initialize(double xInit, double yInit);
   void Print();
};

// this is the "extension" class
class LabelledCoord : public Coord
{
public:
   int label;
   void SetLabel(int theLabel);
};
```

Conceptually, what we have done here is allow `Coord`'s variables and functions to be copied into `LabelledCoord`, as follows:

```
  The class Coord
---------------------
  memory needed by Coord object:
     enough to hold two doubles

  functions supported by Coord object:
     Initialize(double xInit, double yInit)
     Print()


  The class LabelledCoord
 ----------------------
  memory needed by LabelledCoord object:
        enough to hold two doubles            // inherited from Coord
        and an int                            // specific to LabelledCoord

  functions supported by LabelledCoord object:
       Initialize(double xInit, double yInit) // inherited from Coord
       Print()                                // inherited from Coord
       SetLabel(int theColorID)               // specific to LabelledCoord
```

Now, if we create an object of type `LabelledCoord`:

```
LabelledCoord L1;             // local object
LabelledCoord* Lptr;          // local pointer
Lptr = new LabelledCoord();   // dynamic object
```

then in lines 1 and 3 above (in those two lines, actual objects are created), enough memory is set aside to hold two doubles and an int. The `LabelledCoord` object is more than just the `int` declared by `LabelledCoord`; the syntax of inheritance tells the compiler that a `LabelledCoord` consists of all the same things we already put into `Coord`, *plus* an extra `int` to store a label.

The class that gets extended (`Coord` above) is known in C++ as the *base* class. The class that is the extension (`LabelledCoord` above) is known in C++ as the *derived* class. (Note that the concept of a base class is identical to the concept of a superclass in Java, and likewise a derived class is the same as a subclass.)
The specific syntax of inheritance in C++is:

```
class DerivedClassName : public BaseClassName
```

(For this course, we will always have the word `public` above, though there's a few other things it could be as well.)

7.2 Access permissions

When you write a class, the keywords `public` and `private` indicate which member variables and member functions of the class are directly available to clients. (When we say some variable or function is "directly available" to a client, we mean that the client can use it by name, rather than having to call some other function in order to get that other function to use the variable or function in question.) When a member variable of a class is declared to be `private`, it doesn't mean that variable doesn't exist – certainly, it does exist, and when an object of the class is created, space is set aside for that variable. It just means that clients of that class cannot use that variable by name – they have to ask member functions (which *do* have access to that variable by name) to read or write the variable.

It works the same way with inheritance. If there are `private` member variables in the base class, then member functions in the derived class cannot access those varibles by name. For example, if in our base and derived class example from a few pages back, we had made the member variables `private`:

```
class Coord
{
private:
   double xCoord, yCoord;
public:
   void Initialize(double xInit, double yInit);
   void Print();
};

class LabelledCoord : public Coord
{
private:
   int label;
public:
   void SetLabel(int theLabel);
};
```

then because `xCoord` and `yCoord` are `private`, not only can clients of `Coord` not use those variables by directly by name, but the member functions of `LabelledCoord` cannot use those variables directly by name, either. They are marked `private` in `Coord`, and so *nothing* outside of `Coord` can use those variables directly by name.

The important point, though, is this: those variables are *still* part of `LabelledCoord`. That is, when you create a `LabelledCoord` object, there is still space set aside for two `double` variables and an `int` variable. Whether or not a variable exists, and whether or not you can access it, are two entirely different things. This is no different than clients using objects with `private` variables. The clients of `Coord` can't get to `xCoord` and `yCoord` directly by name. However, those two variables are still part of any `Coord` object; the fact that the clients of `Coord` cannot reach them by name does not negate the fact that they are there. Likewise here, there are three variables that are part of `LabelledCoord`, but the clients of `LabelledCoord` cannot access *any* of them directly by name, and the `LabelledCoord` member functions themselves can only access *one* of them by name. The fact that `LabelledCoord` cannot access `xCoord` and `yCoord` by name does not negate the fact that they are part of `LabelledCoord` anyway.

So, if `LabelledCoord` *does* want to access those two variables, it will have to use `Coord` functions to do so, just as clients of `Coord` would have had to do.

What if you did indeed want a derived class to have access to base class variables directly by name? Well, you could have made them `public` in the base class, but if you do that, then not only does the derived class has access to that data, but so does the world at large. One can imagine situations where you'd want to allow derived classes to access certain information from base classes, while still keeping the data out of the hands of the world at large.

Fortunately, there is a way to do this : the keyword `protected`. The access permission `protected` is used just like `public` and `private` are. Member functions and member variables marked `protected` in a base class are available directly by name in derived classes. Furthermore, they are automatically considered `protected` in those derived classed as well – so classes derived from the derived class can still access those variables directly by name, but clients of the derived class will not be able to access those variables directly by name.

To recap the three access statements then:

```
class Coord {
public:
   // These are accessible by any object, including those of derived
   // classes (where they also have public access).
protected:
   // These members can be used only by this class and its
   // derived classes (where they also have protected access).
private:
   // Anything declared here is inaccessible outside of this class. Even
   // derived classes cannot use these members.
};
```

7.3 Constructors and the initializer list

In Java, you had the keyword `super` to enable you to invoke base class constructors from derived class constructors. In C++, you accomplish the same thing through the use of a constructor's *initializer list*. If the following were the no-argument constructor for `LabelledCoord`:

```
// definition for LabelledCoord no-argument constructor
LabelledCoord::LabelledCoord()
{
   label = 0;
}
```

then you can have the no-argument base class constructor called by adding a colon after the close-parenthesis, and then following that colon with the constructor call:

```
// definition for LabelledCoord no-argument constructor
LabelledCoord::LabelledCoord() : Coord()
{
   label = 0;
}
```

In the above code, the call to `Coord()` is part of the `LabelledCoord` constructor's initializer list. The initializer list for a constructor always begins with a colon, placed after the closing parenthesis of that constructor. Then, after the colon, you list the specific initialization code. (Above, we have only shown a base class initializer; we'll see some others a bit later on in this section of the notes.) Once all your initialization code is done, *then* you have the opening curly brace of the actual constructor (the opening curly brace of the `LabelledCoord` constructor, in this case), followed by whatever code should go inside the curly braces. Note that the above example follows this description – the parenthesis of the `LabelledCoord` constructor comes first, followed by a colon, followed by the call to the base class constructor (to initialize variables inherited from the base classs), followed by the open curly brace.

Writing the constructor in this fashion works the same way the `super` call did in Java – namely, it ensures that a base class constructor is called, for the purpose of initializing variables inherited from the base class, before any code is run for initializing the variables added by the derived class. If you wanted to initialize the base class variables using a constructor other than the no-argument constructor, then you would simply place those arguments into the base class constructor call, just as you would have placed those arguments into the call to `super` in Java:

```
// alternate definition for LabelledCoord no-argument constructor;
// calls the two-parameters-of-type-double Coord constructor
LabelledCoord::LabelledCoord() : Coord(5.6, 2.7)
{
   label = 0;
}
```

Recall that in Java, a call to the superclass constructor, using `super`, had to be the first line of the constructor. This forced the superclass constructor to be run before any of the other code of the subclass constructor was run. Likewise in C++, the base class constructor must run before any of the other derived class code can be executed – so that the variables inherited from the base class can be initialized before the variables added by the derived class get initialized. So in the example above, the call to the base class constructor on the initializer list gets run before any of the code inside the curly braces of the derived class constructor gets run. That is, the `Coord()` call gets run before the `label = 0;` line of code is executed.

In fact, there is another way that this works like Java. In Java, not only did you have to have the `super` call as the first line of your constructor, but if you didn't, a `super();` line (i.e. calling the no-argument constructor of the superclass) was automatically inserted by the compiler as the first line of the subclass constructor. You didn't have the option of not calling a superclass constructor as the first line of a subclass constructor – if you didn't explicitly do this, then the compiler did it for you. Likewise in C++, this also happens. If you don't have the colon and a base class constructor call after the close parenthesis of the derived class constructor, then the colon and a call to the base class no-argument constructor are inserted automatically. That is, these two definitions are equivalent:

```
// first definition from examples above
// Coord() is called by default
LabelledCoord::LabelledCoord()      // : Coord()
{
   label = 0;
}

// second definition from examples above
// call to Coord() is explicitly written in
LabelledCoord::LabelledCoord() : Coord()
{
   label = 0;
}
```

So, if the way you want to initialize variables inherited from your base class is by calling the base class no-argument constructor, then you don't need to put any constructor call on the initializer list, since the no-argument constructor of the base class is called by default. On the other hand, if you wanted to initialize those variables inherited from the base class using some other base class constructor instead, you'll need to put a specific call to that constructor on the initializer list.

Also just like in Java, if you rely on the default, and your base class doesn't have a no-argument constructor, you will get a compiler error. If your base class doesn't have a no-argument constructor, then you'll need to choose what other base class constructor to use to initialize the variables inherited from the base class, and you'll have to place a call to that base class constructor on the initializer list of your derived class constructor.

This "initializer list" idea doesn't get used merely for base class constructor calls. You can also use it to initialize the member variables of the derived class. For example, if you wanted to initialize the integer `label` to `0`, you could have written this code instead of the code we've been writing above:

```
LabelledCoord::LabelledCoord() : Coord(), label(0)
{
   // no code here, but you still need curly braces
}
```

This is especially useful if you have non-primitive types as member variables. For example, let's suppose you had some hypothetical `String` class you've written, and so the member variable declaration part of `LabelledCoord` looked like this:

```
class LabelledCoord : public Coord
{
private:
   int label;
   String name1;
   String name2;
public:
      ... // member functions go here
};
```

(Never mind what purpose those two `String` variables might serve; I can't think of any either. They're just here for the sake of the example.) For both of the variables that are of non-primitive type – our two `String` variables in this case – the default constructor is called on them by default on the initializer list. That is, the following two code segments are equivalent, in this case:

```
LabelledCoord::LabelledCoord()
{
   label = 0;
}


LabelledCoord::LabelledCoord() : Coord(), name1(), name2()
{
   label = 0;
}
```

because in the first case, not only is the base class no-argument constructor called by default, but the no-argument constructors for the `String` member variables are called by default as well. So, if you wanted to initialize them using something other than the `String` no-argument constructor – if, for example, there was a constructor that took a double-quoted literal and you want to use that – you would do this as follows:

```
LabelledCoord::LabelledCoord() : Coord(), name1("hello"),
                                          name2("world")
{
   label = 0;
}
```

You are also free to initialize `label` on that list as well:

```
LabelledCoord::LabelledCoord() : Coord(), label(0),
                  name1("hello"), name2("world")
{
   // no code here, but curly braces must remain
}
```

So, if you choose to try and initialize such non-primitive-type objects within the curly braces and not bother with the initializer list, you have to keep in mind that then there is an automatic call to the no-argument constructor for that object on the initializer list anyway. This means it gets initialized once by default, and then reassigned within the curly braces using your code. And in that case, since you are assigning it twice, that is less efficient than just assigning it to the correct value the first time, and therefore you are better off choosing the correct constructor and explicitly initializing the member variable using that constructor, on the initializer list.

Note that this "inititialize your non-primitive member data" use of the initializer list has nothing to do with inheritance. You'll only be calling base class constructors in inheritance situations, but *any* class – whether derived class or not – can use the initializer list to initialize non-primitive-type member variables.

7.4 Dynamic binding...or not

As you might recall from Java, a reference to a superclass can point to a superclass object, but can also point to subclass object instead. Likewise, in C++, a pointer to a base class can point to a base class object, but can also point to a derived class object instead. For example, the following code is legal:

```
Coord* cPtr;
cPtr = new LabelledCoord();
```

Ordinarily, if you tried writing the address of one type of object into the pointer to another type of object, the compiler would complain. But if the type of the object is derived from the type of the pointer (as `LabelledCoord` is derived from `Coord` in the example above), then the compiler is okay with this.

This becomes something we care about when both the base and derived classes have functions with similar names:

```
class Coord
{
private:
   double xCoord, yCoord;
public:
   void Print() {cout << "Coord!" << endl;}
};

class LabelledCoord : public Coord
{
private:
   int label;
public:
   void Print() {cout << "LabelledCoord!" << endl;}
};
```

Note that both classes have a `Print()` function. At first glance, you might think this would be a compiler error. After all, `LabelledCoord` inherits the functions from `Coord`, and so it already has a `Print()` function. Now, we have given it another `Print()` function, in addition to the one inherited from `Coord`. Since the name and the parameter list are identical for the two functions, this would seem to be a problem. However, it is not. In cases such as this, where a function we've specifically written into a derived class would clash with a function which that derived class inherited from the base class, the derived class function simply "covers up" the inherited base class function. That is, the compiler lets this slide, and then whenever we call `Print()` off a `LabelledCoord` object, the `LabelledCoord` version is chosen, rather than the `Coord` version.

That means that when you are invoking the `Print()` function off a local variable, the type of the local variable indicates the `Print()` function that is chosen:

```
Coord c1;
LabelledCoord lc1;
c1.Print();    // prints "Coord!"
lc1.Print();   // prints "LabelledCoord!"
```

Note that the compiler could make those decisions at compile-time. It knows the type of the variable, and so it can use that type to figure out which `Print()` function should get called. Likewise, if you have a pointer of one type, to an object of that same type, the compiler makes the decision based on the type of the pointer:

```
Coord* cPtr = new Coord();
LabelledCoord* lcPtr = new LabelledCoord();
cPtr->Print();   // prints "Coord!"
lcPtr->Print();  // prints "LabelledCoord!"
```

The pointer `cPtr` is of type `Coord`, so the `Coord` version of `Print()` is chosen in the third line in the above example. Likewise, `lcPtr` is of type `LabelledCoord`, so the `LabelledCoord` version of `Print()` is chosen in the fourth line in the above example.

The question is, what happens in this case:

```
Coord* cPtr = new LabelledCoord();
cPtr->Print();
```

Here, the pointer, and the object it points to, are of different types. So which version of `Print()` gets chosen?

If you remember the rules of dynamic binding from Java, you would say that the function that runs is based on the object type, and thus the `LabelledCoord` version of `Print()` is chosen above.

And you'd be wrong.

If this were Java, that is what would happen, but that is not what happens in the above C++ code. The compiler still wants to make all the decisions itself, and so it bases its choice of `Print()` function off the pointer type. Since the type of `cPtr` is `Coord`, it is the `Coord` version of `Print()` that is chosen. It would not make any difference if you had a conditional involved:

```
Coord* cPtr;
int x;
cin >> x;
if (x == 0)
   cPtr = new Coord();
else
   cPtr = new LabelledCoord();
cPtr->Print();
```

In the above case, the compiler doesn't even know what type of object `cPtr` points to, since that depends on user input, which isn't available until run-time. So, the compiler doesn't try to base any decisions on the object type. When you have a base class pointer, the compiler chooses the base class version of the function, without worrying at all about whether that pointer will

eventually point to a base class object or a derived class object. The compiler uses the type of the pointer to determine which class's function gets called.

This is a definite change from Java. In Java, it would not be the compiler that would select the version of `Print()` in the code above. Instead, in Java the run-time environment would select the version of `Print()`, based on the type of the object rather than the pointer. This run-time selection based on object type was something that was referred to as *dynamic binding*, because the proper function defintion was chosen (i.e. bound to the function call) at run-time (i.e. dynamically) rather than at compile time (i.e. statically). Above, we are saying that C++ uses `static binding` – the compiler makes the decision, based on the pointer type, and thus no decisions are made at run-time, since they have already been made by the time the program begins running.

It would seem, then, that all the advantages dynamic binding gives us in Java are absent in C++. However, this is not so. We *do* have dynamic binding in C++, just as in Java. However, in C++, it is necessary to "turn dynamic binding on", whereas in Java, it is on by default. Since we have not turned it on, we have static binding above – the proper function definition is chosen based on the type of the pointer. Once we turn dynamic binding on (which we will discuss in the next section of these notes), then the choice of `Print()` function will be made at run-time, and will be based on the type of the objcet `cPtr` points to, rather than being based off the type of `cPtr` as the compiler does.

Why does C++ do things differently than Java? Well, if the run-time environment must make the choice of function binding, reading various information to make that choice will take time. Therefore, the program will take slightly longer to run. On the other hand, if the compiler makes that choice, then there is no choice to make at run-time and so the program can run slightly faster. In Java, you automatically have dynamic binding on, and so you automatically need to wait until run-time to have these bindings done and so your program will automatically take a bit longer to run. Whereas in C++, the fastest way to do things is what is done by default, and if *you* happen to want the flexibility that dynamic binding provides, then *you* need to make the choice to turn dynamic binding on yourself. Remember, in C++, nothing is done for your that you do not specifically request yourself. Having to turn on dynamic binding is one example of something that you need to do yourself – the language doesn't assume you want to take the extra time needed to support that feature unless you specifically say so.

7.5 Virtual Functions

We turn on dynamic binding in C++ using the keyword `virtual`. This keyword is placed directly in front of the function for which we want to turn on dynamic binding:

```
class Coord
{
private:
   double xCoord, yCoord;
public:
   virtual void Print() {cout << "Coord!" << endl;}
};
```

```
class LabelledCoord : public Coord
{
private:
   int label;
public:
   void Print() {cout << "LabelledCoord!" << endl;}
};
```

If we declare a member function of a base class as virtual, then when we encounter code such as the code we listed in the previous section of these notes:

```
Coord* cPtr;
int x;
cin >> x;
if (x == 0)
   cPtr = new Coord();
else
   cPtr = new LabelledCoord();
cPtr->Print();
```

then in that situation, the compiler does not note that `cPtr` is of type `Coord` and thus choose the `Coord` version of `Print()`. Rather, the compiler merely verifies that the line `cPtr->Print();` will work no matter what `cPtr` points to, and after that, the decision of what version of `Print()` gets called is left to the run-time environment, which makes the decision based on the type of the object that `cPtr` points to at run-time. This is known as *dynamic binding*.

Once you declare a function to be virtual, it remains virtual no matter how many times it gets redeclared in derived classes. So, you don't need to declare it virtual in each derived class. However, it's probably a good idea to do so for documentation purposes.

```
class Coord
{
private:
   double xCoord, yCoord;
public:
   virtual void Print() {cout << "Coord!" << endl;}
};

class LabelledCoord : public Coord
// This class is a derived class
{
private:
   int label;
public:
   virtual void Print() {cout << "LabelledCoord!" << endl;}
};
```

Three points regarding this issue:

- You don't put the keyword `virtual` in front of the function definition. You only put it in front of the function declaration inside the class declaration. In practice, this means you put the keyword `virtual` in the `.h` file but not the `.C` file.
- Each individual function must be declared virtual. That is, declaring one function as virtual does not suddenly make all the other functions in that class virtual as well. Therefore, it is possible to have some functions in a class be virtual, and some not be virtual – i.e. to have dynamic binding for some functions in a class and static binding for other functions in that same class.
- You can never have virtual constructors, but destructors *can* be virtual. In general, if a class has virtual functions, its destructor should be virtual. This is because you want dynamic binding to work on the destructor as well. If we run the line `delete cPtr;`, we want the `Coord` destructor to run if `cPtr` points to a `Coord` object, and we want the `LabelledCoord` destructor to run if `cPtr` points to a `LabelledCoord` object.

7.6 Abstract classes and pure virtual functions

In some instances, you will have one or more functions in a base class which don't have any definition that makes any sense. You would have declarations for these functions only so that the function name appears in the base class to be overwritten later by identical function names in the derived classes. But you would have only the declaration, and no definition. In Java, these were known as abstract methods. In C++, they are known as *pure virtual functions*, and they are declared as follows:

```
virtual function_declaration = 0;

// for example:
virtual void Print() = 0;
```

A class with at least one pure virtual function is called an abstract class. Unlike in Java, we don't specifically notate that the class is abstract; it simply is abstract, with no syntax to signify this. As with Java, you cannot create objects of classes which are abstract.

7.7 Proper Inheritance

Just because you *can* use inheritance in certain ways doesn't mean you *should* use it in those ways. Many serious design errors can result from using inheritance improperly. *Proper Inheritance* is defined to be a use of inheritance in which derived classes "promise no less and require no more" than the base class they are derived from.

An excellent example of this is the class "Ellipse/Circle" inheritance problem. Imagine you have the following inheritance hierarchy:

```
class Ellipse
{


   double xAxis, yAxis;
};

class Circle : public Ellipse
{


};
```

Is this inheritance okay? Certainly we might *prefer* to implement `Circle` using a single "radius" variable rather than two axis variables, but since a circle is basically an ellipse with an equal-sized x-axis and y-axis, why not save ourselves the work of writing an entirely new `Circle` class and reuse the `Ellipse` code by deriving `Circle` from `Ellipse`? The various functions of `Ellipse` would then be functions for `Circle` as well, and so, for example, we could read either `xAxis` or `yAxis` (presumably using public functions of `Ellipse` that were written for that purpose) to get the radius of the `Circle` object.

The problem arises in that some of the functions of `Ellipse` don't work nicely with the idea of a `Circle`. For example:

```
class Ellipse
{
   // scales xAxis by a factor of xFac and y-axis by
   //  a factor of yFac
   virtual void Scale(xFac, yFac);

   double xAxis, yAxis;
};

class Circle : public Ellipse
{


};
```

What do we do with `Scale` in `Circle`? We appear to have three options:

1. We can accept `Scale` as is, as a public function of `Circle`. This means that we will have to let `Circle` scale differently in the x and y directions, since the `Scale` function permits that to occur. But, if `Circle` objects can have different x and y axis lengths, then what differentiates a (deformed) `Circle` from an `Ellipse`? Nothing, really. If we allow `Scale` to operate on `Circle` as it currently exists, then the resultant ability to deform `Circle` objects means we may as well not even have had a `Circle` class, since it isn't behaving much differently than a regular `Ellipse`.

2. We can try and shut the `Scale` function off somehow for the `Circle` class:

```
class Circle : public Ellipse
{
public:
   // does nothing at all
   virtual void Scale(xFac, yFac)  {}
};

or, alternatively,

class Circle : public Ellipse
{
public:

private:
   // scales circle exactly as ellipse's scale function
   //  does, so we hide it in the private section so the
   //  user can't get to it
   virtual void Scale(xFac, yFac) { whatever in here;}
};
```

In the first case, we simply make `Scale` an empty function; in the second, we restrict access to it.

The problem with both of these methods is that if we had the following code:

```
Ellipse* elPtr;
// some more lines of code, such as maybe elPtr = new Ellipse();
elPtr->Scale(2,3);
```

then we would like to see a consistent result. That is, when we wrote the above code, we assumed first that we could call `Scale` off of an `Ellipse` object or an object of a class derived from `Ellipse`, second that it would do something, and third that the particular something it would do would be to scale the ellipse to the appropriate factors. We assumed all of that because it was in the specification of the `Scale` function in the `Ellipse` class. Later, we come along and add the `Circle` class, and if we add one new line of code inside the section of code indicated with the comment above:

```
elPtr = new Circle();
```

then the `elPtr->Scale(2,3);` line of code won't work anymore. The point of inheritance is to allow old code to call new code, but here adding the new code breaks the old code and so we have to go in and mess with things. This is where design problems and huge later-maintenance issues arise. And the reason we have a problem here is because the `Circle` class "promised less" than its base class (`Ellipse`) did – namely, `Circle` said that "even though `Ellipse` promised you that `Scale` was call-able, that it at least did something,

and that specifically it did what the function specification describes, my version of `Scale` either doesn't work, doesn't do anything, or doesn't do what the original specification described." We have promised less and thus have run into trouble.

3. Finally, we could simply require that if `elPtr` points to a `Circle`, you can still use the `Scale` function but you must pass it two equal values as parameters (so that the `Circle` scales in equal magnitudes both ways and remains a circle). However, just as with the second option, this option will break the current `elPtr->Scale(2,3);` line of code. Here, we have "required more" – we have tacked on additional requirements to calling the `Scale` function that were not in the original specification. So, we again run into trouble.

So, what do we do? Well, we admit to ourselves that, as nice as the mathematical abstraction may be, in this case a `Circle` should NOT inherit from an `Ellipse`. That is, inheritance is NOT about a derived class being more specific; it is about a derived class being *substitutable* for the base class. In our examples above, `Circle` is a more specific form of `Ellipse`, but it is not perfectly substitutable for `Ellipse` and thus it is not inheritance we should attempt. If you promise no less in a derived class and require no more in a derived class, then that insures it is substitutable for a base class and that is proper inheritance – when derived class objects are perfectly substitutable for base class objects. You can still add new functions to the derived class, of course – that would be promising more and that is fine. But your derived class has to at least support the behavior in the specification of the base class – that is the bare minimum it must do.

Your intuition does not matter. Code reuse does not matter, as that is only the second purpose of inheritance. Inheritance is first and foremost about substitutability and you should NOT violate that rule even if "intuition" says you should (for example, even though intution says that circles are more specific forms of ellipses, you should not derive `Circle` from `Ellipse` if `Ellipse` has a `Scale` function).