COMP642

Object Oriented Programming

Lectorial 5

Book Class

- Consider a class for keeping information on books:

```
class Book:
    def __init__(self, cnum, pdate, title, author):
        bookCatNumber = cnum
        bookPurchaseDate = pdate
        bookTitle = title
        bookAuthor = author

    def info(self):
        return "Book Title: " + self.bookTitle + " Author:   " + self.bookAuthor
```

Different Types of Books?

- Sometime later we may add “talking books”. The information about these is exactly the same as an ordinary book but with the additional information of a Playing Time.
- We could:

  Create an entirely new class **TalkingBook**

  Or

  Add a **playTime** property to the Book class

Both of these solutions have problems –

What are they?

Problems

- Extra Class:

  Will now have two classes with similar properties. Awkward to update them both.

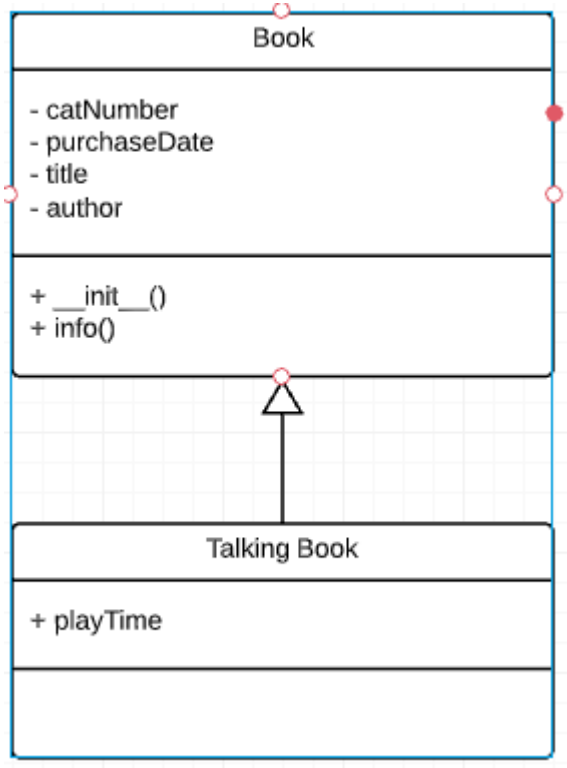
- Extra Property:

  What about the books that don’t have a playing time?

  Is it a talking book for which we don’t know the playing time or is it an ordinary book?

Inheritance

- **`TalkingBook`** derives from (or is inherited from, or extends) **Book**
- **Book** is the Base (or parent or super) class
- **`TalkingBook`** is a derived (or child or sub) class
- A **`TalkingBook`** **IS A** **Book**

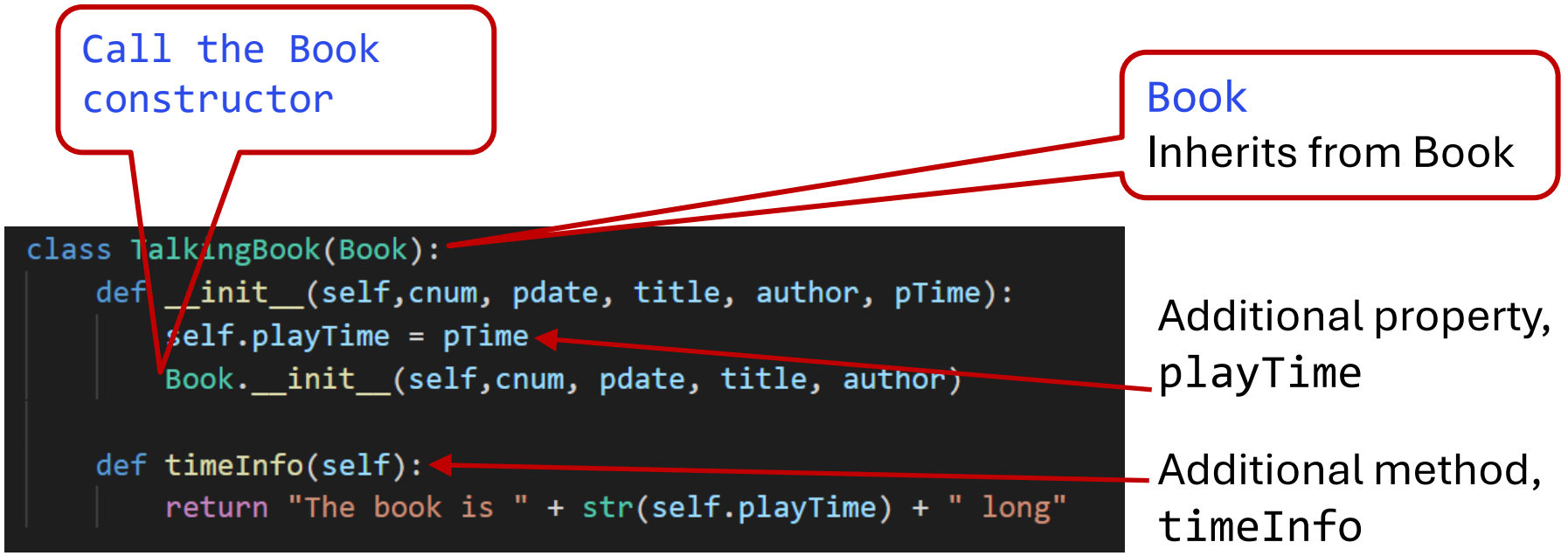


Inherited Properties

- A derived class inherits all the properties and methods of its base class.
- A **`TalkingBook`** **IS A** **`Book`** therefore it has a title, author, catalogue number, purchase date and an Info method.

Additional Properties in Child Class

- A derived class can add additional properties and methods.
- A **`TalkingBook`** also has a playTime property.
- We can also add methods e.g. `timeInfo`.



A TalkingBook Object

- We can create an object of the **`TalkingBook`** class.

```
talkBook = TalkingBook(1001, '1/1/2021', 'The Big Giant', 'Unknown', 500)
```

- `talkBook` will have 5 private properties:

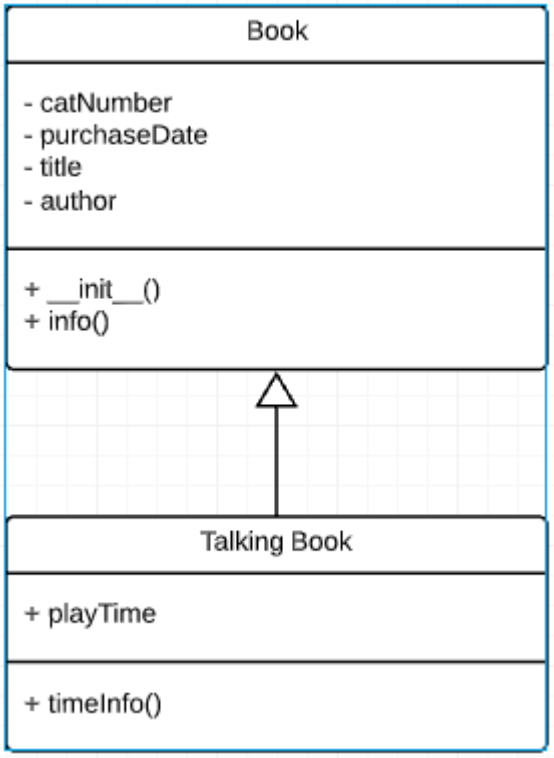
  4 from its parent

  1 additional one

  and

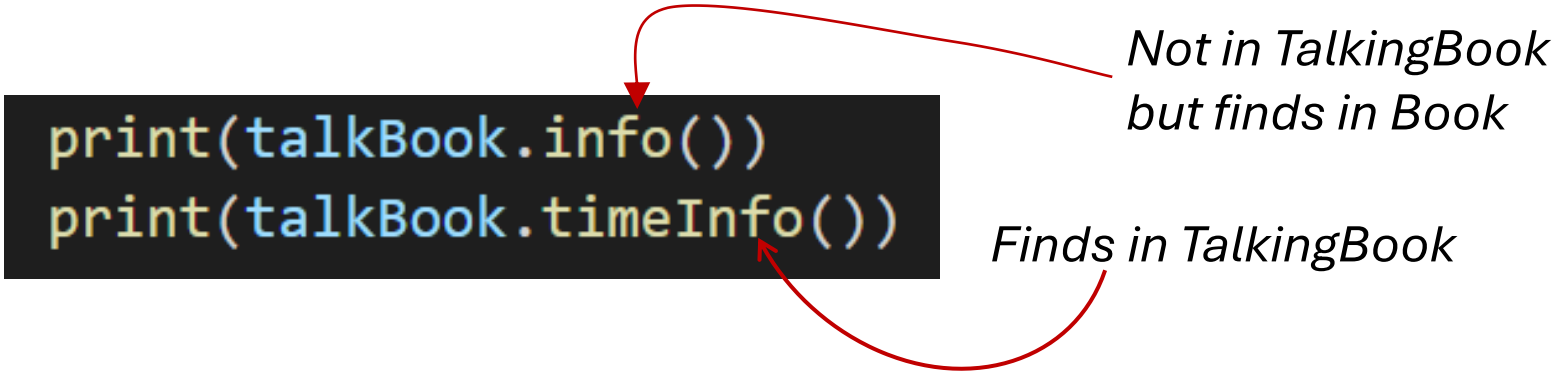
  1 public method inherited from its parent

  1 public method of its own



Calling Methods

- If we call a method of the **`TalkingBook`** object the program will:
  - Look in **`TalkingBook`** class for the method.
  - If it can't find it, it will look in the parent class (**Book**).



  - It will keep searching up the hierarchy until it finds the method.

Overriding Methods (1)

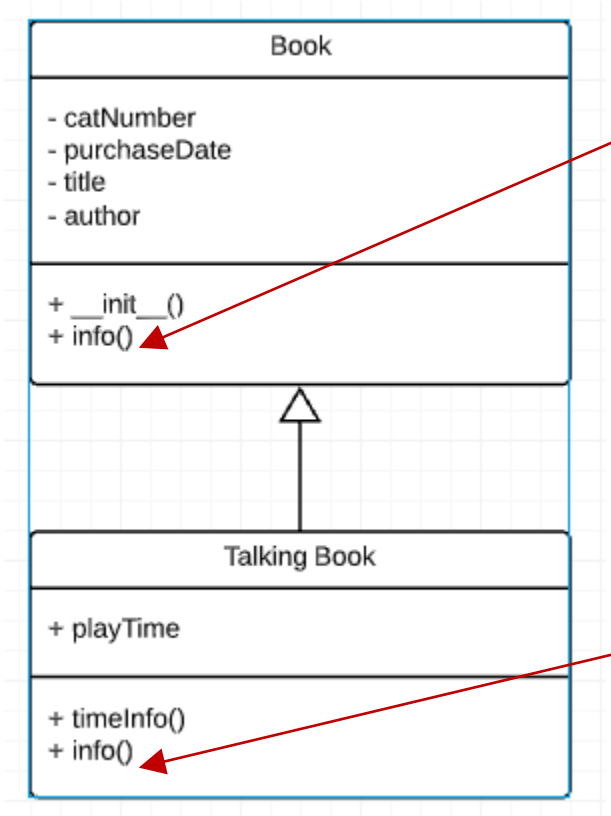
- The information generated by a **`TalkingBook`** may need to be different from its parent class **Book**.
- E.g.
  - We may want to add a playing time to the method info().
  - We can **Override** a method of a base class by redefining it in the derived class.

Overriding Methods (2)

- In `TalkingBook` class: declare function with same signature

```
def info(self):
    return "I am a Talking Book"
```

Overriding Methods (3)



Book *objects will use this info method.*

`TalkingBook` *now has its own (overridden)* info *method.*

`TalkingBook` *objects will now use this method (not an inherited one from Book).*

Calling Overridden Methods

- Depending on the **type** of object the appropriate method will be called.

```
aBook = Book(1002, '1/8/2021', 'The Small Giant', 'Unknown')
print(aBook.info())
```

```
Book Title: The Small Giant Author:  Unknown
```

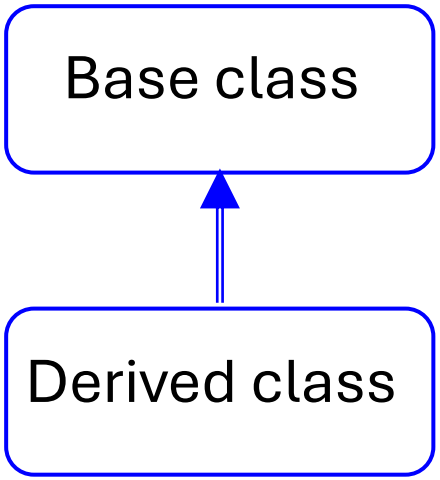
```
talkBook = TalkingBook(1001, '1/1/2021', 'The Big Giant', 'Unknown', 500)
print(talkBook.info())
```

```
I am a Talking Book
```

Inheritance

- Provides code reusability – use existing class to create new class instead of creating it from scratch.
- Child class can access all data and methods defined in the parent class.
- Child class can have additional data and methods.
- Child can also provide specific implementations to the methods of the parent class (overriding).

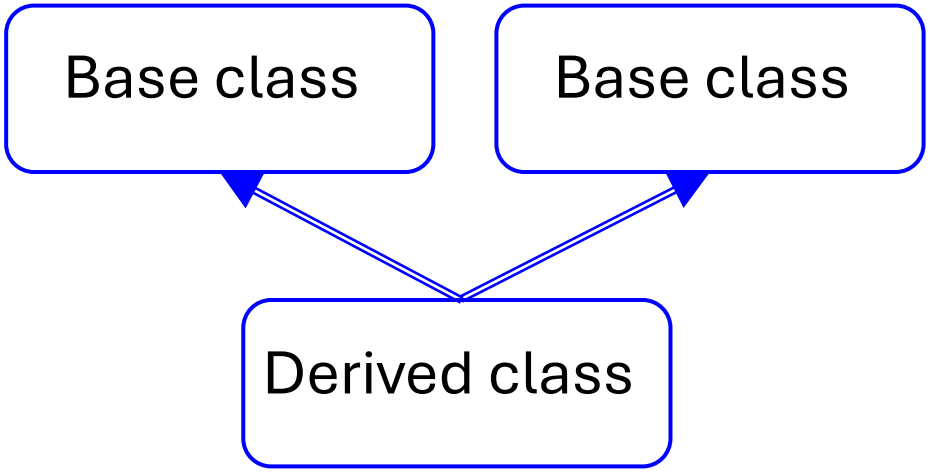
Single Inheritance




```python
class Animal:
    def speak(self):
        print("Animal Speaking")

class Dog(Animal):
    def bark(self):
        print("Dog Barking")

aDog = Dog()
aDog.speak()
aDog.bark()
```

```
Animal Speaking
Dog Barking
```

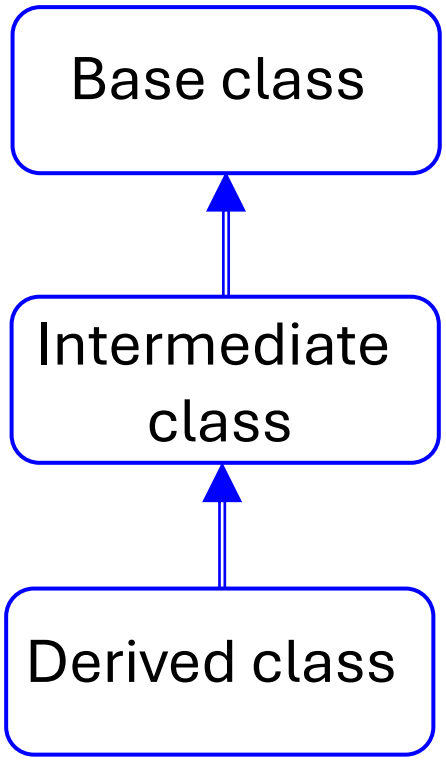
Multiple Inheritance




```python
class Mother:
    def mother(self):
        print("Mother")

class Father:
    def father(self):
        print("Father")

class Son(Mother, Father):
    def parent(self):
        self.father()
        self.mother()

aSon = Son()
aSon.parent()
```

```
Father
Mother
```

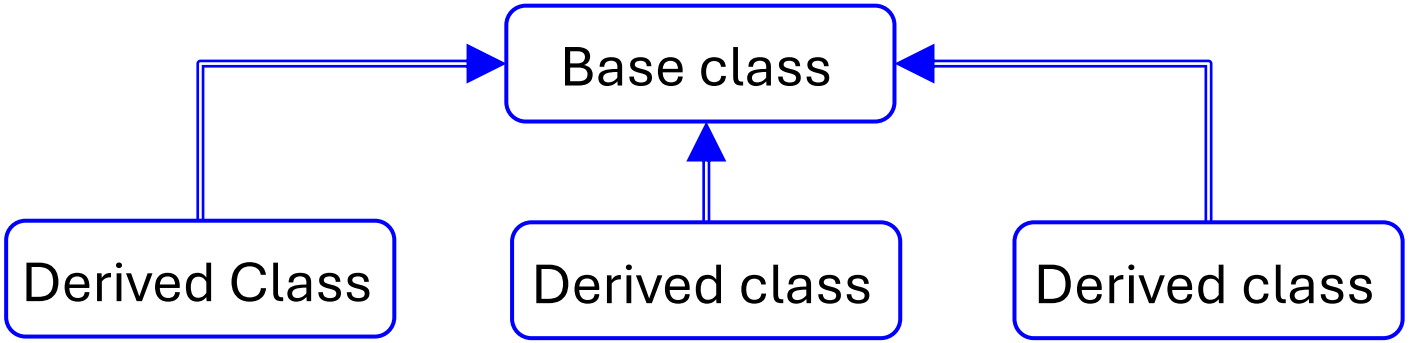
Multilevel Inheritance




```python
class Grandfather:
    def __init__(self, gname):
        self.gName = gname

class Father(Grandfather):
    def __init__(self, fname, gname):
        self.fName = fname
        Grandfather.__init__(self,gname)

class Son(Father):
    def __init__(self, sname, fname, gname):
        self.sName = sname
        Father.__init__(self,fname,gname)

    def fullName(self):
        return self.sName + " " + self.fName + " " + self. gName

aSon = Son('Andy', 'Smith', 'Parker')
print(aSon.fullName())
```

```
Andy Smith Parker
```

Hierarchical Inheritance




```python
class Parent:
    def func(self):
        print("Parent class function")

class Child1(Parent):
    def func1(self):
        print("Child1 class function")

class Child2(Parent):
    def func2(self):
        print("Child2 class function")

class Child3(Parent):
    def func3(self):
        print("Child3 class function")
```

Test Your Knowledge

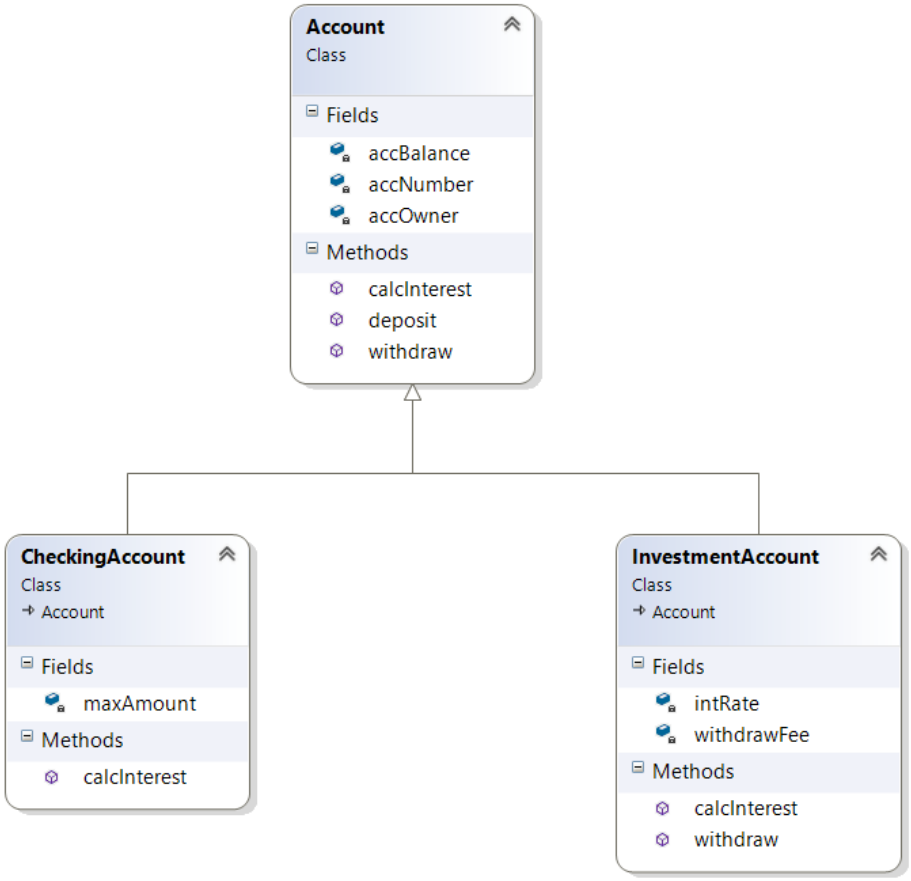
A bank has several different accounts:
All accounts have a number, an owner, and a balance.
All accounts have withdrawal, deposit, and interest methods.

Ordinary accounts pay interest of 6%.
Cheque accounts pay interest (6%) only on amounts over a specified value (e.g. $2000).
Investment accounts pay interest of 9%. Withdrawal fee is $20.

How could we set up inherited classes to do this?

(There are lots of ways.)

Bank Account



Polymorphism (1)

- The ability to take different forms.
- The same method name can be used for different types.
- Inbuilt polymorphic functions:

```
print(len("Python"))
print(len([10, 20, 30]))
```

- User-defined polymorphic functions (method overloading):

```
def sum(x, y, z = 0):
    return x + y + z

print(sum(10, 20))
print(sum(10, 20, 30))
```

```
30
60
```

Polymorphism (2)

- Polymorphism with class methods:

```
class Cat:
    def speak(self):
        print("Cat Speak: MEOW")

class Dog:
    def speak(self):
        print("Dog Speak: WOOF")

class Duck:
    def speak(self):
        print("Duck Speak: QUACK")
```

```
animalList = []
aCat = Cat()
animalList.append(aCat)

aDog = Dog()
animalList.append(aDog)

aDuck = Duck()
animalList.append(aDuck)

for animal in animalList:
    animal.speak()
```

```
Cat Speak: MEOW
Dog Speak: WOOF
Duck Speak: QUACK
```

Methods called based on the type of object.

Polymorphism (3)

- Polymorphism with inheritance:

  Define methods in the child class that has the same name as the methods in the parent class (overriding).

```
aBook = Book(1002, '1/8/2021', 'The Small Giant', 'Unknown')
talkBook = TalkingBook(1001, '1/1/2021', 'The Big Giant', 'Unknown', 500)
pictBook = PictureBook(1003,'2/2/2021', "Barney the Dinosaur", 'Robert Smith', 'Animal' )

aList = []
aList.append(aBook)
aList.append(talkBook)
aList.append(pictBook)

for book in aList:
    print(book.info())
```

```
Book Title: The Small Giant Author:  Unknown
I am a Talking Book
I am a Picture Book
```

Private Variables and Child Classes

- Private members cannot be accessed by derived class members.

```
class Book:
    def __init__(self, aTitle, anAuthor):
        self.__bookTitle = aTitle
        self.bookAuthor = anAuthor

class TalkingBook(Book):
    def __init__(self, aTitle, anAuthor, pTime):
        self.__playTime = pTime
        Book.__init__(self, aTitle,anAuthor)

    def bookInfo(self):
        return "A talking book called " + self.__bookTitle

tBook = TalkingBook("Cinderella", "Unknown", 500)
print(tBook.bookInfo())
```

```
    return "A talking book called " + self.__bookTitle
AttributeError: 'TalkingBook' object has no attribute '_TalkingBook__bookTitle'
```

Even though a `TalkingBook` has a `myTitle` property through inheritance, it cannot refer to it because it is **private** to the parent class.

Private Variables and Child Classes

- Parent class can make this property public.

```
class Book:
    def __init__(self, aTitle, anAuthor):
        self.bookTitle = aTitle
        self.bookAuthor = anAuthor

class TalkingBook(Book):
    def __init__(self, aTitle, anAuthor, pTime):
        self.__playTime = pTime
        Book.__init__(self, aTitle,anAuthor)

    def bookInfo(self):
        return "A talking book called " + self.bookTitle
```

This allows public access to the `bookTitle`.

No special privilege for child classes.

What if we want to allow child classes to access the data but not the public?

Protected Variables

- Declaring a variable as **protected** means that it is available to inherited classes but is still invisible to other classes.
- Single underscore to denote protected. However, this is actually accessible outside class.
- Responsible programmer should refrain from accessing and modifying variables prefixed with _ from outside its class.
- Define getter setter using property decorator

```
class Book:
    def __init__(self, aTitle, anAuthor):
        self._bookTitle = aTitle
        self.bookAuthor = anAuthor
```

super()

- `super` is a key word that refers to the object's base (parent) class.

```
class Book:
    def __init__(self, aTitle, anAuthor):
        self._bookTitle = aTitle
        self.bookAuthor = anAuthor

    def info(self):
        return "Title: " + self._bookTitle + " Author: " + self.bookAuthor

class TalkingBook(Book):
    def __init__(self, aTitle, anAuthor, pTime):
        self.__playTime = pTime
        Book.__init__(self,aTitle,anAuthor)

    def bookInfo(self):
        return super().info() +  " Talk time: " + str(self.__playTime)
```

use super to access info() of Book class

Constructors and Inheritance (1)

- Because a child object **IS A** parent object, when we create a child, we need to create a parent object first and then add on the extra bits.
- This means we need to run a constructor in the parent class before running any constructor in a child class.

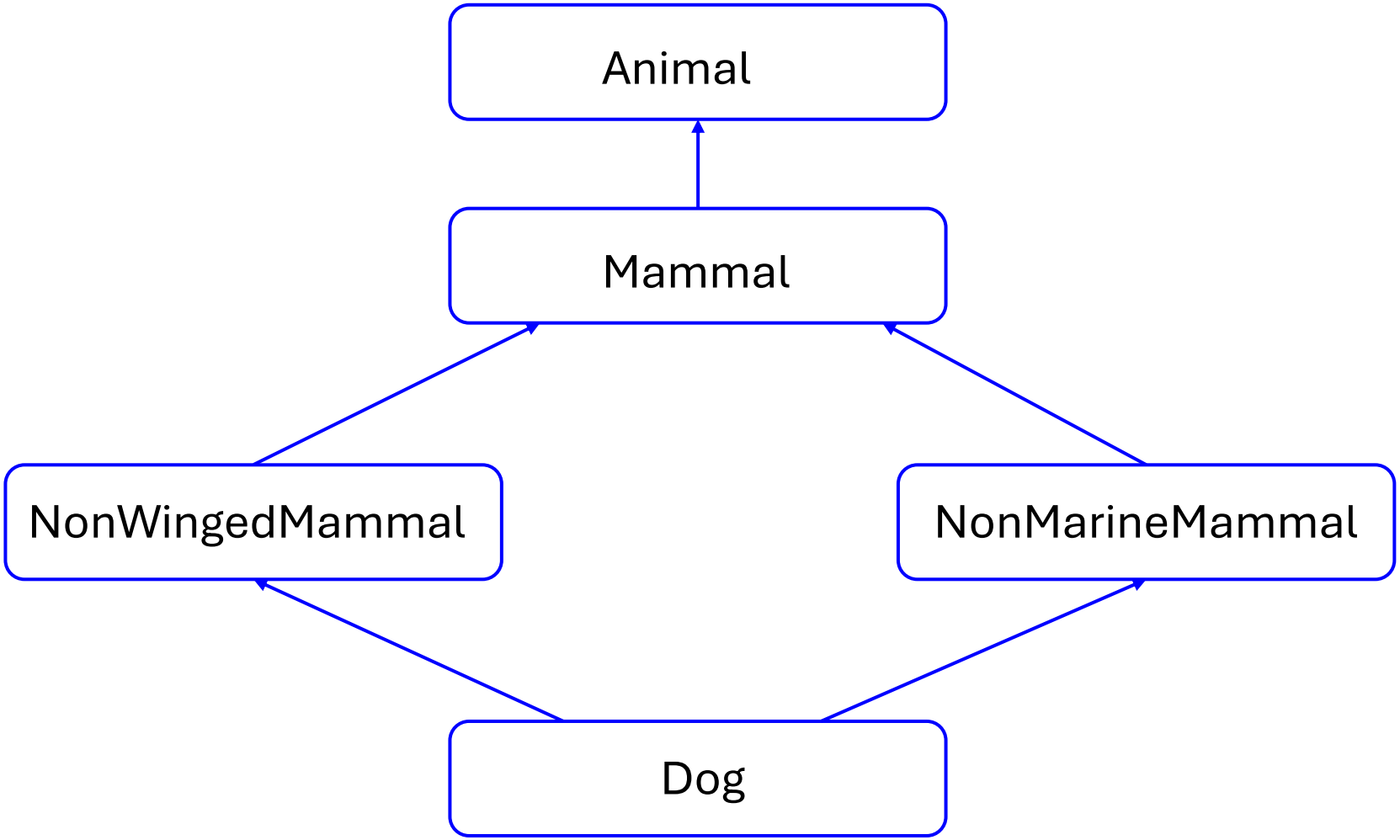
```
class TalkingBook(Book):
    def __init__(self, aTitle, anAuthor, pTime):
        self.__playTime = pTime
        Book.__init__(self,aTitle,anAuthor)
```

Constructors and Inheritance (2)

```
class TalkingBook(Book):
    def __init__(self, aTitle, anAuthor, pTime):
        self.__playTime = pTime
        super().__init__(aTitle,anAuthor)
```

- A more elegant way of calling the parent's class constructor.
- If the parent's class name is changed, the child class is not affected.

super() with Multiple Inheritance (1)



super() with Multiple Inheritance (2)

```
class Animal:
  def __init__(self, Animal):
    print(Animal, 'is an animal.');

class Mammal(Animal):
  def __init__(self, mammalName):
    print(mammalName, 'is a warm-blooded anima
    super().__init__(mammalName)

class NonWingedMammal(Mammal):
  def __init__(self, NonWingedMammal):
    print(NonWingedMammal, "can't fly.")
    super().__init__(NonWingedMammal)

class NonMarineMammal(Mammal):
  def __init__(self, NonMarineMammal):
    print(NonMarineMammal, "can't swim.")
    super().__init__(NonMarineMammal)

class Dog(NonMarineMammal, NonWingedMammal):
  def __init__(self):
    print('Dog has 4 legs.');
    super().__init__('Dog')

d = Dog()
```

```
Dog has 4 legs.
Dog can't swim.
Dog can't fly.
Dog is a warm-blooded animal.
Dog is an animal.
```

Constructors are called based on the Method Resolution Order (MRO).

Each constructor will only be called once in a specific order based on the MRO.

super() with Multiple Inheritance (3)

```
class Animal:
  def __init__(self, Animal):
    print(Animal, 'is an animal.');

class Mammal(Animal):
  def __init__(self, mammalName):
    print(mammalName, 'is a warm-blooded anima
    super().__init__(mammalName)

class NonWingedMammal(Mammal):
  def __init__(self, NonWingedMammal):
    print(NonWingedMammal, "can't fly.")
    super().__init__(NonWingedMammal)

class NonMarineMammal(Mammal):
  def __init__(self, NonMarineMammal):
    print(NonMarineMammal, "can't swim.")
    super().__init__(NonMarineMammal)

class Dog(NonMarineMammal, NonWingedMammal):
  def __init__(self):
    print('Dog has 4 legs.');
    super().__init__('Dog')

d = Dog()
```

```
Dog has 4 legs.
Dog can't swim.
Dog can't fly.
Dog is a warm-blooded animal.
Dog is an animal.
```

```
print(Dog.__mro__)
```

<class '__main__.Dog'>,
<class '__main__.NonMarineMammal'>,
<class '__main__.NonWingedMammal'>,
<class '__main__.Mammal'>, <class
'__main__.Animal'>, <class 'object'>)

super() with Multiple Inheritance (4)

```
bat = NonMarineMammal('Bat')
print(NonMarineMammal.__mro__)
```

```
Bat can't swim.
Bat is a warm-blooded animal.
Bat is an animal.
```

class '__main__.NonMarineMammal'>, <class '__main__.Mammal'>, <class '__main__.Animal'>, <class 'object'>

How MRO Works:

- A method in the derived calls is always called before the method of the base class.
- If there are multiple parents, methods of the parent class that appears first is invoked first.

Test Your Knowledge

- What differences would these ideas make to the bank account example from last lecture if we wanted to supply an owner when we created the objects?