Week 7

Object-oriented Programming (OOP)

Bank Accounts

`bank_account.py`

Deposit

- Add a function `deposit()` to deposit some money into the account

```
>>> myAcc = BankAccount('Alan',1000)
>>> myAcc.showBalance()
Your balance is $1000.00
>>> myAcc.deposit(200)
>>> myAcc.deposit(400)
>>> myAcc.showBalance()
Your balance is $1600.00
```

Secure Withdrawal

- Add a control measure when you withdraw where a name must be provided
  - It must match the name in the account

```
>>> myAcc = BankAccount('Alan',1000)
>>> myAcc.withdraw('Mary',100)
You are not authorized for this account
>>> myAcc.withdraw('Alan',10000)
Money not enough! You do not have $10000.00
0
>>> myAcc.withdraw('Alan',100)
100
>>> myAcc.showBalance()
Your balance is $900.00
```

Interest

- Add an attribute for interest rate
  - Initialize it at the **constructor**
- Add a function `oneYearHasPassed()` which adds the interest gained after one year to the account

```
>>> myAcc = BankAccount('Alan',1000,0.04)
>>> myAcc.showBalance()
Your balance is $1000.00
>>> myAcc.oneYearHasPassed()
>>> myAcc.showBalance()
Your balance is $1040.00
>>> myAcc.oneYearHasPassed()
>>> myAcc.showBalance()
Your balance is $1081.60
```

Minimal Account

- Create a class `MinimalAccount`
- Behaves the same as the normal `BankAccount`, except:
  - Whenever one year has passed, if its balance is less than $1000, a $20 administration fee will be deducted from the account
  - The fee will be deducted BEFORE the annual interest is gained
  - The balance will never be below zero
- Discuss with your neighbor, how will you design this class?
  - **Directly modify** `BankAccount`? Or…
  - **Duplicate** `BankAccount` and modify it? Or…
  - What else?

Minimal Account

```
>>> mySonAcc = MinimalAccount('John',40,0.04)
>>> mySonAcc.oneYearHasPassed()
>>> mySonAcc.showBalance()
Your balance is $20.80
>>> mySonAcc.oneYearHasPassed()
>>> mySonAcc.showBalance()
Your balance is $0.83
>>> mySonAcc.oneYearHasPassed()
>>> mySonAcc.showBalance()
Your balance is $0.00
```

Extra Tasks

- Method `transferTo()` in class `BankAccount`
  - Given another account, you can transfer your money to the account

```
>>> myAcc.transferTo(myWifeAcc,500)
```

- Method `setupGiro()` in class `BankAccount`
  - Money will be deducted annually before interest is gained

```
>>> myAcc = BankAccount('Alan',1100,0.04)
>>> myAcc.setupGiro(40)
>>> myAcc.setupGiro(60)
>>> myAcc.oneYearHasPassed()
>>> myAcc.showBalance()
Your balance is $1040.00
```

- A new class `JointAccount`
  - An account has two owners and both owners can withdraw money

Vehicles

`vehicle.py`

Recap: Lecture

Vehicle

- Attributes: `pos,velocity`
- Methods: `setVelocity(),move()`

Cannon

- Attributes: `numAmmo`
- Methods: `fire()`

Sportscar

- Methods: `__init__()`, `turnOnTurbo()`

Lorry

- Attributes: `cargo`
- Methods: `__init__(),load(), unload(),inventory()`

Tank

Bisarca

- Methods: `load()`

Petrol

- Let's try to be more realistic: every vehicle needs some petrol
  - Sportscar, Lorry, etc.
- Add a method `addPetrol(n)` that adds n liters of petrol into a vehicle
- For every "move", the vehicle will consume 1 litre of petrol
- What attribute do you need to add? And where?

```
>>> myCar.addPetrol(2)
>>> myCar.move()
Move to (0, 80)
>>> myCar.move()
Move to (0, 160)
>>> myCar.move()
Out of petrol. Cannot move.
```

Add where?

Vehicle

- Attributes: `pos,velocity`
- Methods: `setVelocity(),move()`

Cannon

- Attributes: `numAmmo`
- Methods: `fire()`

Sportscar

- Methods: `__init__()`, `turnOnTurbo()`

Lorry

- Attributes: `cargo`
- Methods: `__init__(),load(), unload(),inventory()`

Tank

Bisarca

- Methods: `load()`

Add Red and Modify Green

Vehicle

- Attributes: pos,velocity,petrol
- Methods: setVelocity(),move(),addPetrol()

Cannon

- Attributes: numAmmo
- Methods: fire()

Sportscar

- Methods: __init__(), turnOnTurbo()

Lorry

- Attributes: cargo
- Methods: __init__(),load(), unload(),inventory()

Tank

Bisarca

- Methods: load()

Solar Tank

- How about a Tank that can survive on solar power?
  - Don’t need petrol

Pulling your leg???

How to Design a Solar Tank?

Vehicle

- Attributes: `pos,velocity,petrol`
- Methods: `setVelocity(),move(),addPetrol()`

Cannon

- Attributes: `numAmmo`
- Methods: `fire()`

Sportscar

- Methods: `__init__()`, `turnOnTurbo()`

Lorry

- Attributes: `cargo`
- Methods: `__init__(),load(), unload(),inventory()`

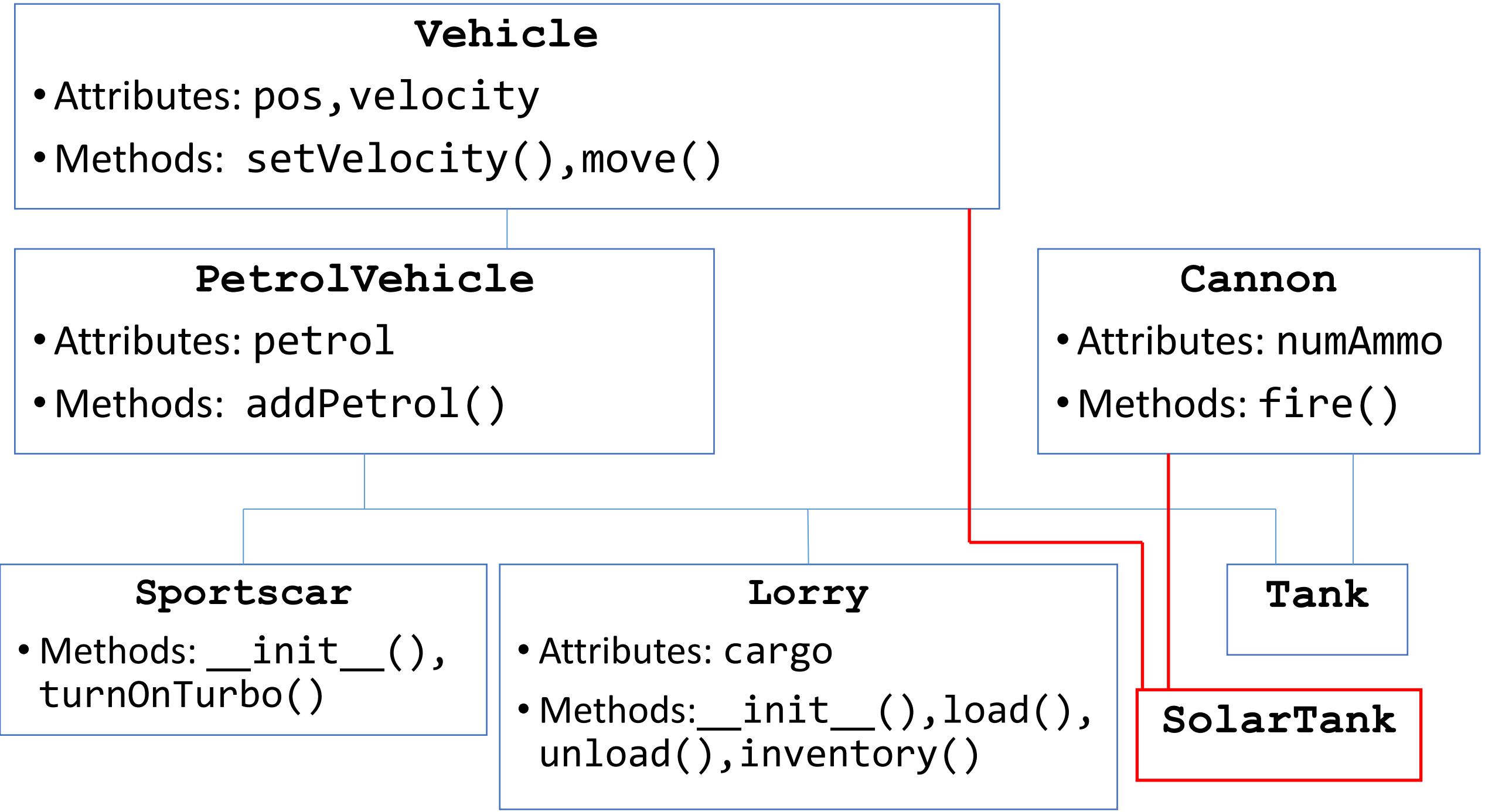
Tank

Bisarca

- Methods: `load()`

Solution?

- Separate the current “petrol” vehicle into
  - A superclass Vehicle and a Subclass PetrolVehicle
  - Then the solar tank will be a subclass of both Vehicle and Cannon


Vehicle
• Attributes: pos,velocity
• Methods: setVelocity(),move()
PetrolVehicle
• Attributes: petrol
• Methods: addPetrol()
Cannon
• Attributes: numAmmo
• Methods: fire()
Sportscar
• Methods: __init__(), turnOnTurbo()
Lorry
• Attributes: cargo
• Methods:__init__(),load(), unload(),inventory()
Tank
SolarTank

Solution?

- Get into trouble with
  - SolarBattleBisarca

```
class Lorry(PetrolVehicle):
    …
class Bisarca(Lorry):
    …
class BattleBisarca(Bisarca,Cannon):
    …
```

  - You are forced to re-implement a SolarBisarca first? or…?

You want the “load()” in Bisarca but don’t want petrol

PetrolVehicle

- Attributes: `pos,velocity,petrol`
- Methods: `setVelocity(),move(),addPetrol()`

Cannon

- Attributes: `numAmmo`
- Methods: `fire()`

Sportscar

- Methods: `__init__()`, `turnOnTurbo()`

Lorry

- Attributes: `cargo`
- Methods: `__init__(),load(), unload(),inventory()`

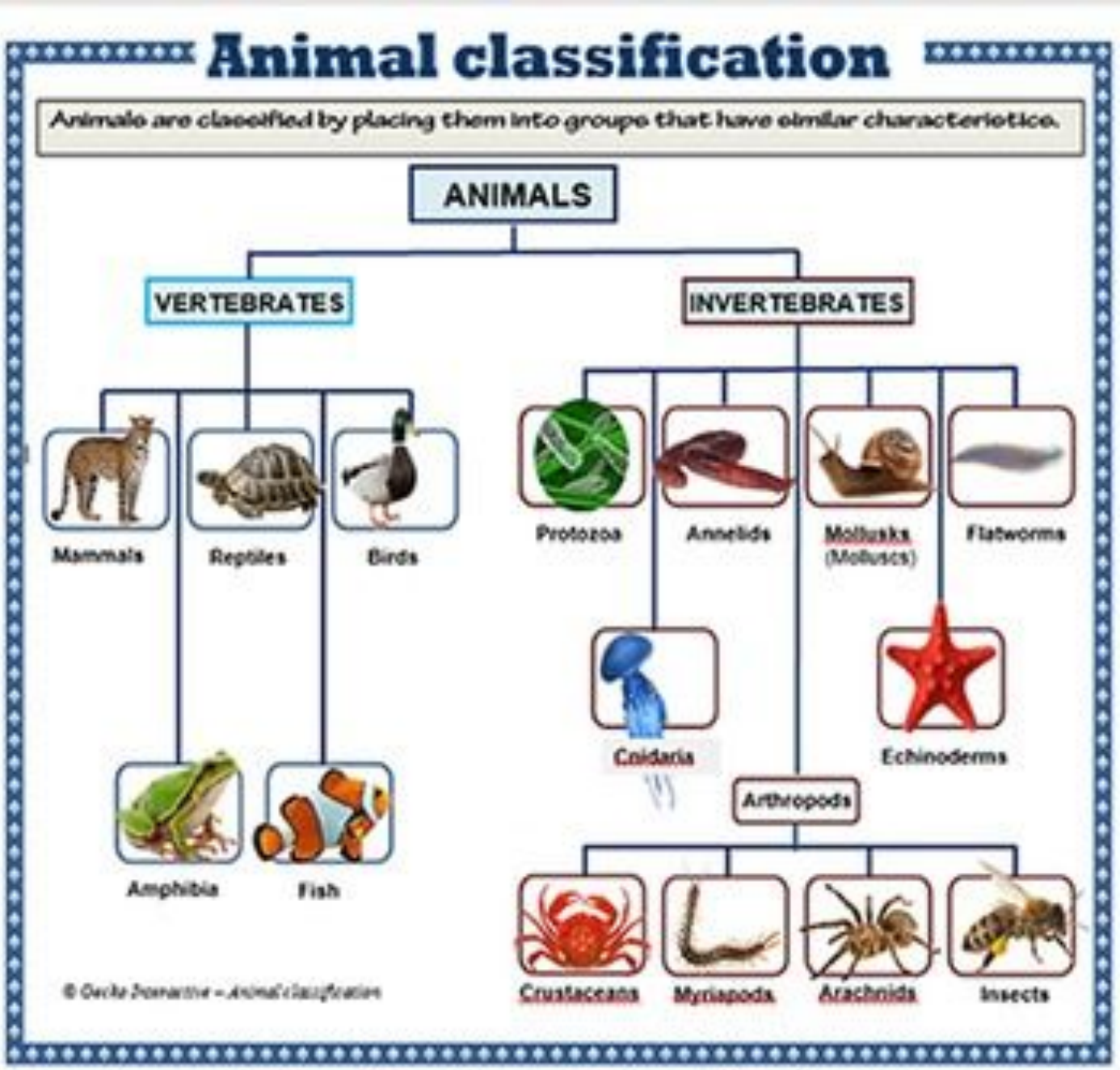
Tank

Bisarca

- Methods: `load()`

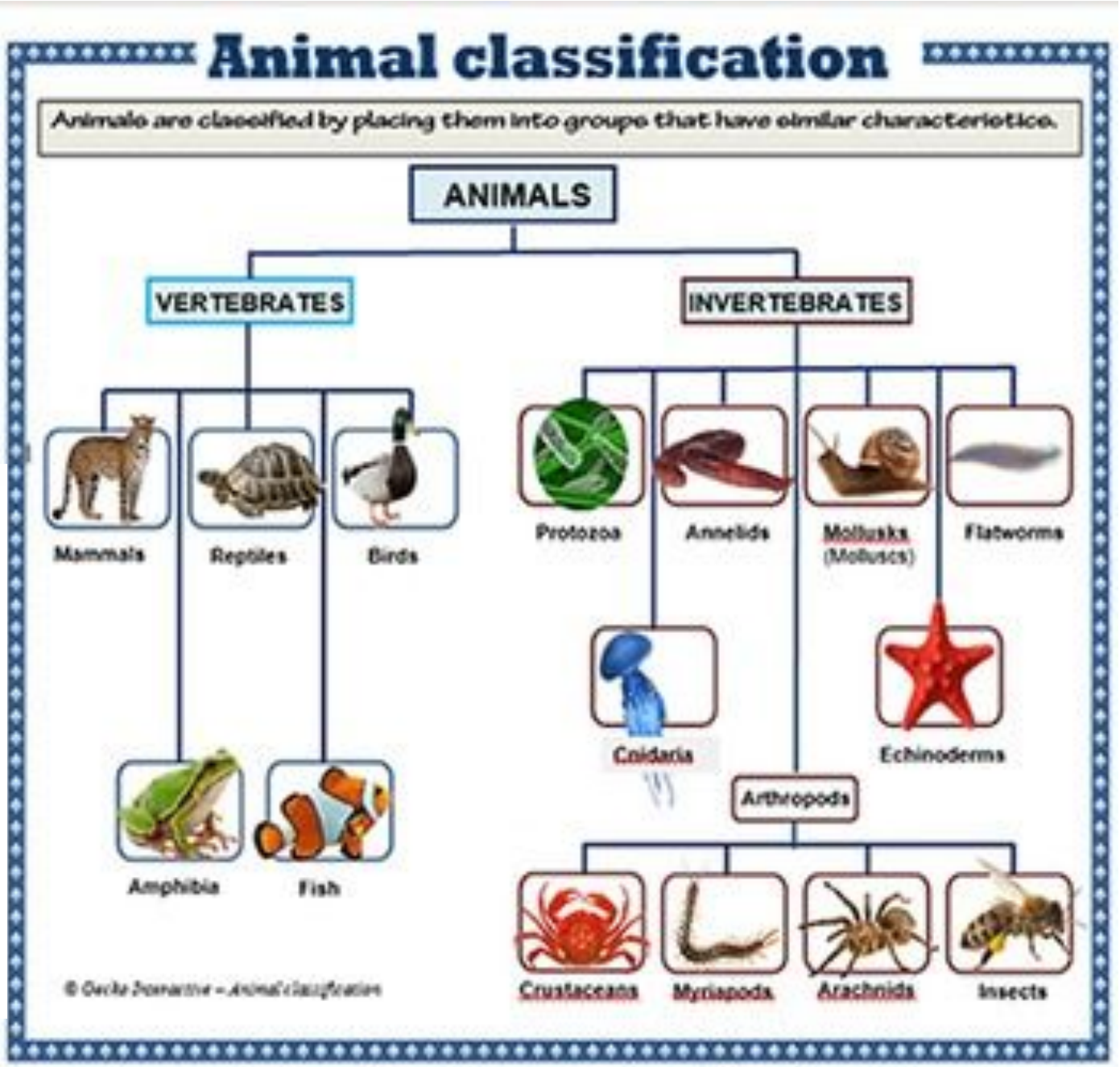
Design Issue

- If every object can be classified nicely, the world would be beautiful
  - Every subclass is a subset of its superclass
  - Every subclass in the same level is distinct
  - Not like …



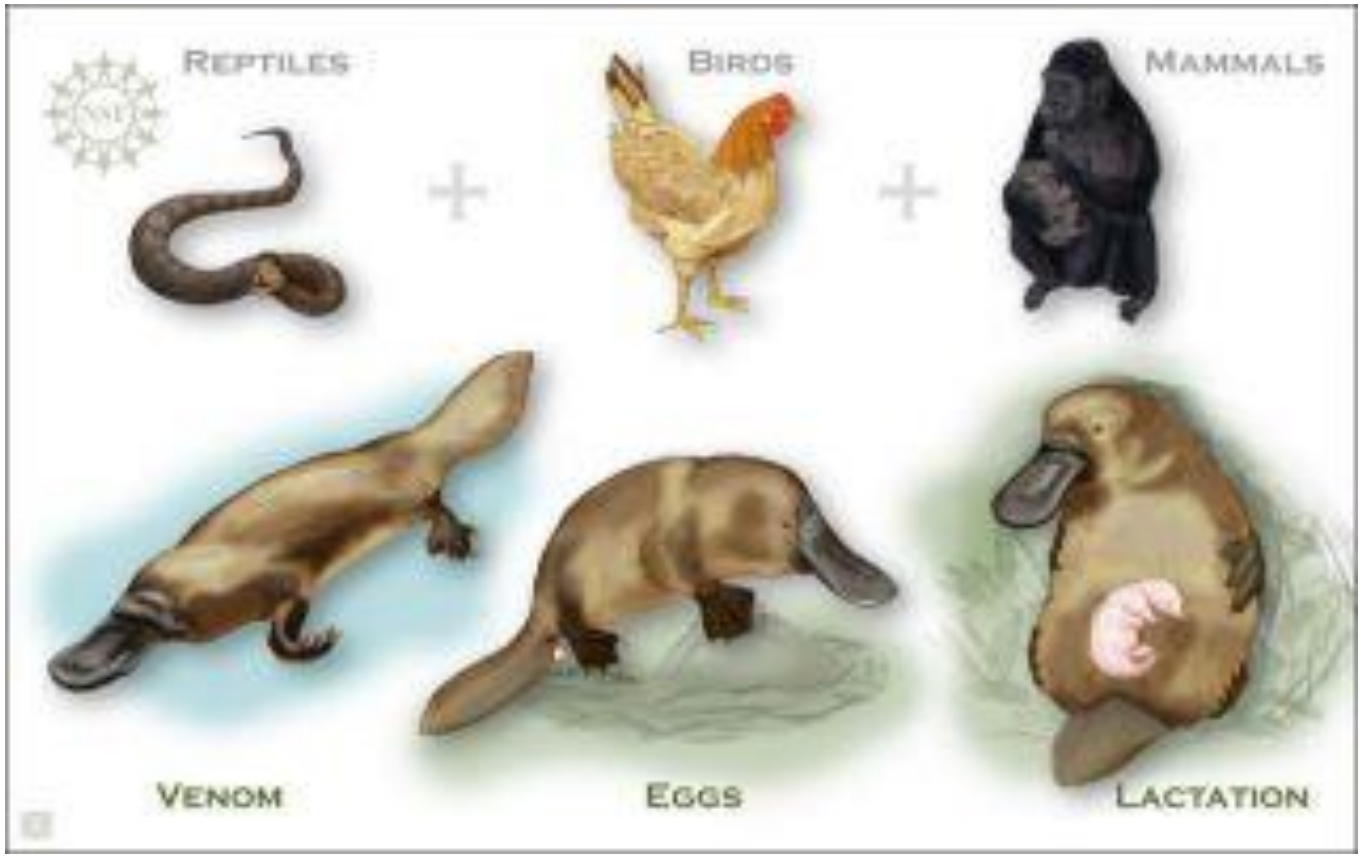
Design Issue

- Where will you fit platypus in the classification?



Where will you fit platypus ?

- Platypus
  - Has venom like reptiles
  - Lays eggs like birds
  - Produces milk like mammals



Design Issue

- Where will you fit platypus in the classification?