LECTURE 7 OBJECT RELATIONSHIPS

Fundamentals of Programming - COMP1005

Department of Computing
Curtin University

Updated 27th September 2019

Copyright Warning

Learning Outcomes

- Understand and apply class relationships: composition, aggregation and inheritance
- Understand and use exception handling
- Write code to work with and test classes

CLASS RELATIONSHIPS

Fundamentals of Programming

Lecture 7

Goals of Object-Orientation

- Reuse / Extensibility
  - Reuse: each class provides its functionality to other classes
  - Can inherit from a class to reuse/extend its functionality
- Modularization - low coupling, high cohesion
  - Objects should be responsible for their own data state
  - Objects should represent a single concept and all methods should relate to that concept (high cohesion)
  - Only the object’s interface should matter to a user of that object, not the details of its implementation (low coupling)

- ***Note****: many of these slides are from Object-Oriented Program Design*

Class Relationships

- The classes of objects which communicate with each other via message passing share some form of relationship (association):
  - Aggregation
  - Composition
  - Inheritance
  - Other

Class Relationships

- Aggregation:
  - One class is declared as a class field within the other class
  - Communication is one way (most of the time?), from class to class field
- Composition:
  - One class is included as part of the other class
  - The included class does not exist without the host class

Class Relationships

- Inheritance:
  - One class is a descendant of another class
  - Uses polymorphism, method overloading or direct references to the superclass to communicate.
  - Communication is one way, from child to parent (sound familiar!!)
- Other:
  - Where objects of one class are related to another in a manner which is NOT aggregation or inheritance.
  - These other relationships will be discussed in future units.

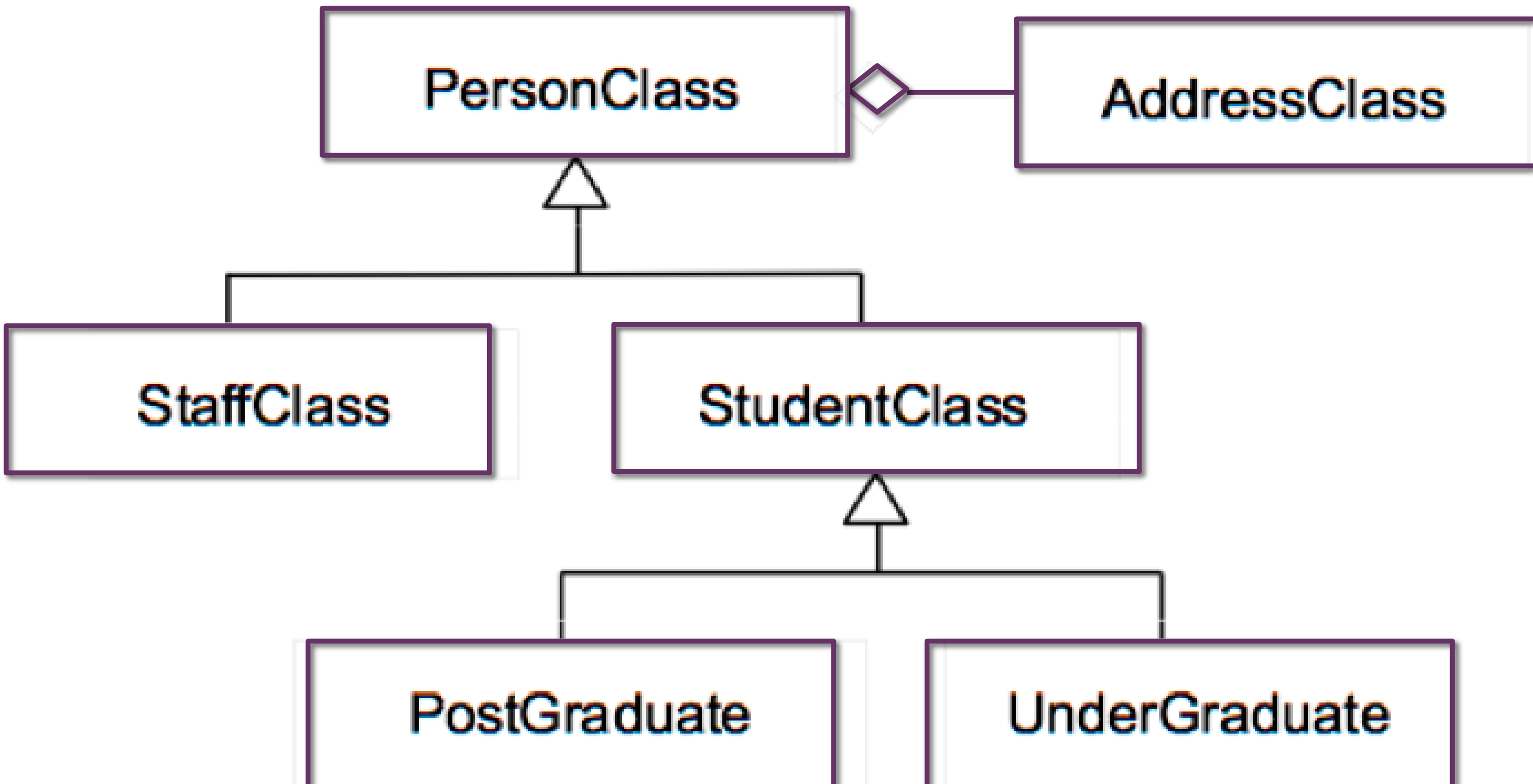
Object Communication

- Also referred to as message passing:
- When an object of one class calls a method in an object of another class it is passing a message
- A request to the object to perform some task

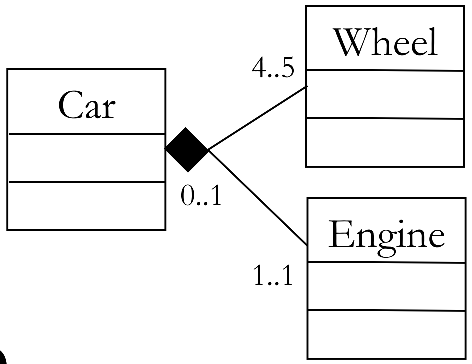
Modelling Languages

- Used to show the relationships between different classes and different instances of classes (i.e. objects) in a particular software
- Usually graphical
- Most commonly adopted methodology is known as **UML**:
  - **Unified:** a union of the approaches put forward by Grady Booch, James Rumbaugh and Ivar Jacobson
  - **Modelling:** a graphical representation (or model) of an OO software design
  - **Language:** provides a standard way of expressing object relationships (i.e. contains rules for syntax & semantics)
- Software Engineering units teach UML and OO software design.
- For now we will simply look at the UML notation for class diagrams - describing inheritance and aggregation/composition.

Uni People Example – Class Diagram

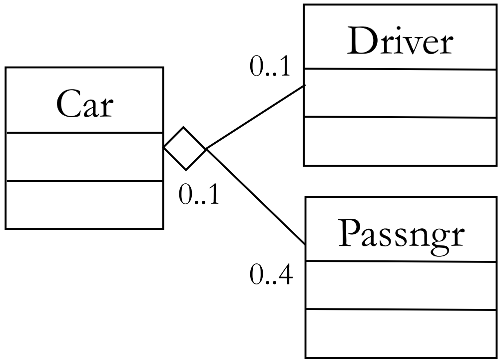


Class Relationships (1)



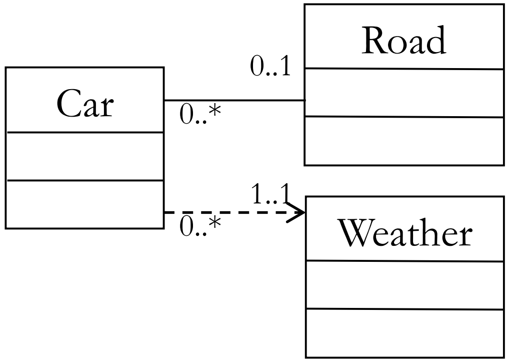
- Composition
  - **“has-a”** or “whole-part” relationship
    - UML: Shown with solid diamond beside container class
    - *e.g.*, Car “has-a” Wheel
  - Strong lifecycle dependency between classes
    - Car is not a car without four Wheels and an Engine
    - When Car is destroyed, so are the Wheels and Engine
  - In code:
    - Car would have Wheel and Engine as class fields

Class Relationships (2)



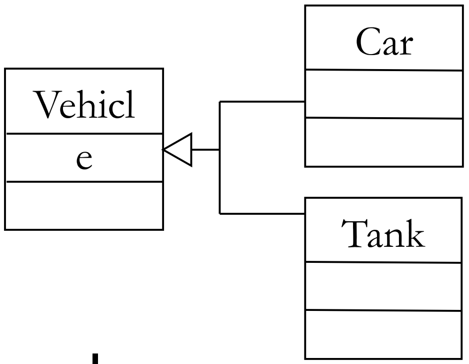
- Aggregation
  - Weaker form of composition, but is still **“has-a”**
    - UML: Shown with open/unfilled diamond beside container
  - Lifecycle dependency usually not strong
    - Car does not always have a driver
    - When Car is destroyed driver and passengers are not
    - Drivers can drive different cars
  - In code:
    - Car would have Driver and Passenger as class fields
    - …exactly like composition!

Class Relationships (3)



- Association and Dependency
  - Indicates interaction between classes
    - Association = solid line, Dependency = dashed line
    - Difference is murky: UML is a *guide*, not a *law*
  - Used to show that one class invokes methods on another
    - … but that there is no other relationship beyond this
    - With arrow, implies *unidirectional* (Car calls Weather, not vice-versa)
    - No arrow implies *bidirectional* (Car and Road call each other)
  - In code: Any way that a method call can be set up and made
    - *e.g.*, Weather object is passed as a parameter to a Car method
      - *e.g.*, Car.setAggressiveness(Weather currentConditions)
    - *e.g.*, Road has a class field of all Cars on that Road (aggregation?)

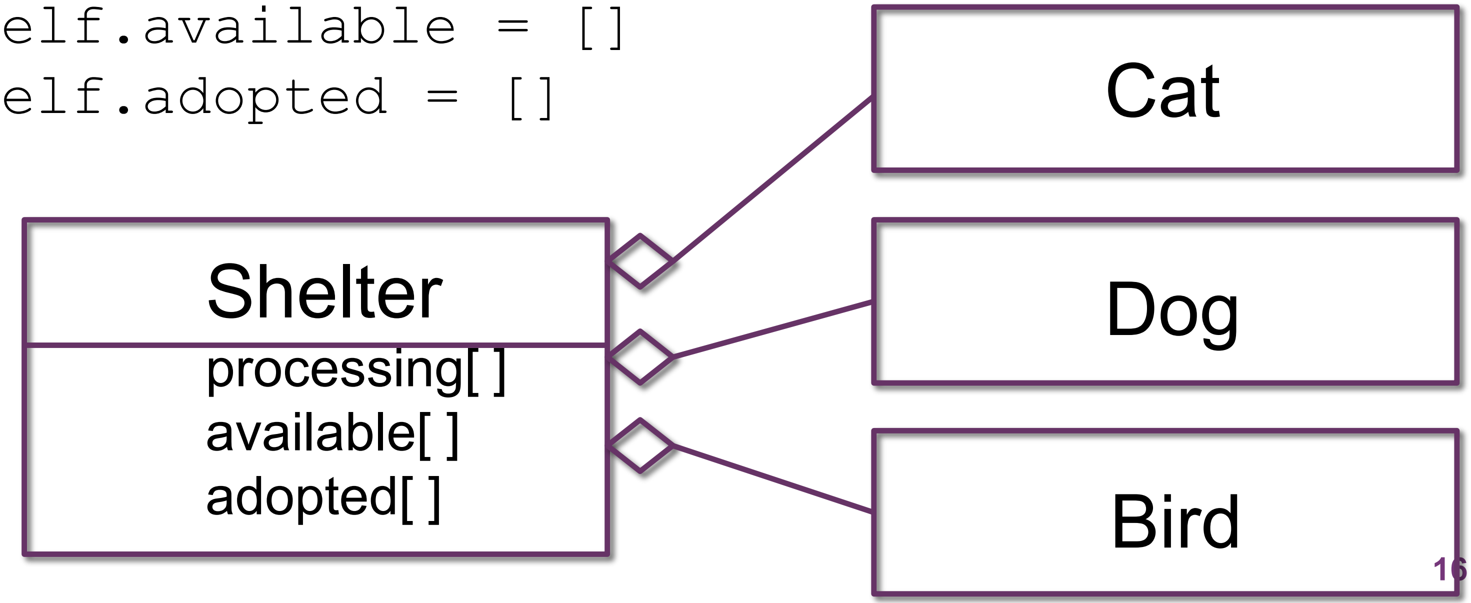
Class Relationships (4)



- Inheritance
  - **“is-a”** relationship
    - Indicates one class is a sub-type of another class
    - Shown with an open triangle arrowhead beside super-type
  - Implies the specialisation of the super-type
    - Super-type synonyms: ‘parent’, ‘base’
    - Sub-type synonyms: ‘child’, ‘derived’
  - In code: During class declaration; syntax is language-specific
    - Python: class Car(Vehicle):
    - Java: public class Car **extends** Vehicle
    - C++/C#: public class Car **:** Vehicle

Example: Pet Shelter (animals.py)

```
class Shelter():

    def __init__(self, name, address, phone):
        self.name = name
        self.address = address
        self.phone = phone
        self.processing = []
        self.available = []
        self.adopted = []
```



Example: Pet Shelter (animals.py)

```
def newAnimal(self, type, name, dob, colour, breed):
       temp = None
       if type == 'Dog':
           temp = Dog(name, dob, colour, breed)
       elif type == 'Cat':
           temp = Cat(name, dob, colour, breed)
       elif type == 'Bird':
           temp = Bird(name, dob, colour, breed)
       else:
           print('Error, unknown animal type: ', type)
       if temp:
           self.processing.append(temp)
           print('Added ',name, ' to processing list')
```

Example: Pet Shelter (shelters.py)

```
from animals import Dog, Cat, Bird, Shelter

print('\nPet shelter program...\n')

rspca = Shelter('RSPCA', 'Serpentine Meander', '123456')
rspca.newAnimal('Dog', 'Dude', '1/1/2011', 'Brown', 'Jack Russell')
rspca.newAnimal('Dog', 'Brutus', '1/1/1982', 'Brown',
                'Rhodesian Ridgeback')
rspca.newAnimal('Cat', 'Oogie', '1/1/2006', 'Grey', 'Fluffy')
rspca.newAnimal('Bird', 'Big Bird', '10/11/1969', 'Yellow', 'Canary')
rspca.newAnimal('Bird', 'Dead Parrot', '1/1/2011', 'Dead', 'Parrot')

print('\nAnimals added\n')
```

Example: Pet Shelter (shelters.py)

```
print('Listing animals for processing...\n')

rspca.displayProcessing()

print('Processing animals...\n')

rspca.makeAvailable('Dude')
rspca.makeAvailable('Oogie')
rspca.makeAvailable('Big Bird')
rspca.makeAdopted('Oogie')

print('\nPrinting updated list...\n')

rspca.displayAll()
```

Example: Pet Shelter (output)

```
Processing animals...

Added  Dead Parrot to available list
Added  Oogie  to available list
Added  Big Bird  to available list
Added  Oogie  to adopted list

Printing updated lists...

Current processing list:
DOG   :  Dude   DOB:  1/1/2011  Colour:  Brown  Breed:  Jack Russell
DOG   :  Brutus          DOB:  1/1/1982  Colour:  Brown  Breed:  
Rhodesian Ridgeback

Current available list:
BIRD  :  Dead Parrot     DOB:  1/1/2011  Colour:  Dead   Breed:  Parrot
BIRD  :  Big Bird        DOB:  10/11/1969        Colour:  Yellow
        Breed:  Canary

Current adopted list:
CAT   :  Oogie  DOB:  1/1/2006  Colour:  Grey   Breed:  Fluffy
```

Inheritance

- Inheritance is the ability of a new class of object to take on all of the properties of an existing class
  - i.e. the state and the functionality
- **Super Class**: The original class
- **Sub Class**: The new class which inherits all of the functionality of the super class
- The sub class can then:
  - Introduce additional state (class fields)
  - Modify the inherited functionality.
  - Introduce new functionality
    - i.e. more specialised
- The super class generally has less functionality than the sub class
    - i.e. more generalised

Aggregation v's Inheritance

- An aggregation relationship is implied by the class field declarations
- An inheritance relationship is explicitly stated (given in brackets on the class definition)

- Note that BOTH relationships encapsulate the functionality of one class within another:
  - Any inheritance relationship can be re-expressed as an aggregation relationship and vice versa.
  - The choice is based upon which relationship is most appropriate.

Class Responsibility

- Each class has a designated role or responsibility in the software system
- It may be that some classes have duplicated functionality
- This duplicated functionality can be removed and placed into a super class which the original classes inherit from
- It is important to ensure that a sub class never assumes the role of its super class
- If the sub class requires some super class functionality then it should call the appropriate super class method

Super Class - Sub Class Communication

- Communication is one way:
  - Sub class calls super class methods but not the other way around
- The word *super is used to refer to the super class*
- super() by itself is a call to the super class' __init__ method

- super().methodName() is a call to a public method in the super class
- Example:
  - In a super class there is a toString() method
    - outStr = super().toString()
  - The sub class toString method wishes to generate a string containing its own state plus the super class state:
    - outStr = super().toString() + self.state

The Base Class

- All classes except one inherit from another class
- A special class, known as the *base class, is the only class that does not*
- In Python this base class is called *object*
- If no inheritance relationship is specified then it automatically inherits from the base class
  - Note: In Python 2, a class definition needed to state it inherited from object – def class person(object)

Super Class / Sub Class
Object Construction

- In order to construct a sub class object, a super class object must also be created
- The order of object construction is from the base class through to the sub class

animals.py - Dog Class (Lecture 6)

```
class Dog():

    myclass = "Dog"

    def __init__(self, name, dob, colour, breed):
        self.name = name
        self.dob = dob
        self.colour = colour
        self.breed = breed

    def printit(self):
        print('Name: ', self.name)
        print('DOB: ', self.dob)
        print('Colour: ', self.colour)
        print('Breed: ', self.breed)
        print('Class: ', self.myclass)
```

animals.py - Cat Class (Lecture 6)

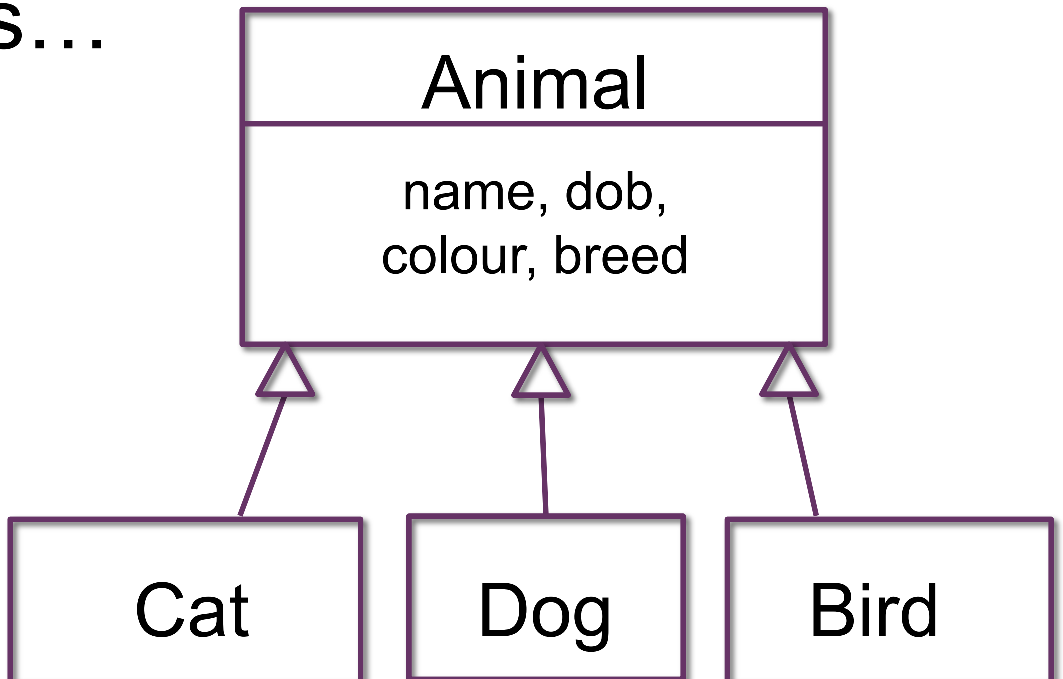
```
class Cat():

    myclass = "Cat"

    def __init__(self, name, dob, colour, breed):
        self.name = name
        self.dob = dob
        self.colour = colour
        self.breed = breed

    def printit(self):
        print('Name: ', self.name)
        print('DOB: ', self.dob)
        print('Colour: ', self.colour)
        print('Breed: ', self.breed)
        print('Class: ', self.myclass)
```

animals.py - Bird Class (Lecture 6)

```
class Bird():

    myclass = "Bird"

    def __init__(self, name, dob, colour, breed):
        self.name = name
        self.dob = dob
        self.colour = colour
        self.breed = breed

    def printit(self):
        print('Name: ', self.name)
        print('DOB: ', self.dob)
        print('Colour: ', self.colour)
        print('Breed: ', self.breed)
        print('Class: ', self.myclass)
```

Example: Inheritance

- Repetition should be avoided if possible
- Cat, Dog and Bird are nearly identical
- Factor out the duplicated fields and methods…



Example: animals.py

```
class Animal():

    myclass = "Animal"

    def __init__(self, name, dob, colour, breed):
        self.name = name
        self.dob = dob
        self.colour = colour
        self.breed = breed

    def __str__(self):
        return(self.name + '|' + self.dob + '|' + self.colour+'|'+self.breed)

    def printit(self):
        spacing = 5 - len(self.myclass)
        print(self.myclass.upper(), spacing*' ' + ': ', self.name,'\tDOB: ',
                 self.dob,'\tColour: ', self.colour,'\tBreed: ', self.breed)
```

Example: animals.py – magic!

```
class Dog(Animal):

    myclass = "Dog"

class Cat(Animal):

    myclass = "Cat"

class Bird(Animal):

    myclass = "Bird"
```

Just the differences between the **Animal** superclass and the subclasses

These changes would have no impact on Shelter.py or pets.py

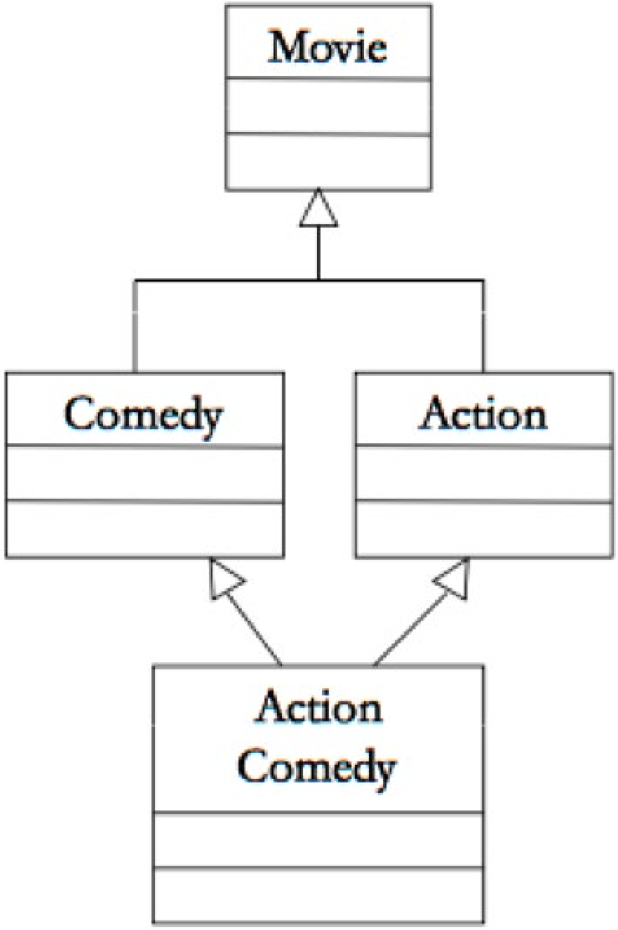
Multiple Inheritance

- So what do we do if a class is required to inherit the state and functionality of more than one super class?
- So Tank “is-a” Vehicle
  - But Tank “is-an” Artillery as well, not just a Vehicle and Artillery is not always a Vehicle, so can’t put Artillery in between Tank and Vehicle
  - ie: Tank really has more than one base class
- One solution: allow multiple inheritance (eg: Python, C++)
  - Tank inherits from *both* Vehicle and Artillery

Multiple Inheritance – Problems

- Theoretically, multiple inheritance is fine
- But in practice (in the code), things can get messy
- Say both Vehicle and Artillery define a method getSize()
  - If Tank does not override getSize(), which getSize() version should the compiler call? Vehicle’s? Artillery’s?
  - Worse, what if Artillery.getSize() refers to the size of the *shells* it fires, but Vehicle.getSize() refers to the *vehicle’s* size?
- In more complicated inheritance hierarchies, you can even inherit from the same class more than once!
  - The next slide shows an example of this

Multiple Inheritance - Example



EXCEPTION HANDLING

Fundamentals of Programming

Lecture 7

Errors and exceptions

- Errors or mistakes in a program are often referred to as bugs
- They are almost always the fault of the programmer
- The process of finding and eliminating errors is called debugging
- Errors can be classified into three major groups:
  - Syntax errors
  - Runtime errors
  - Logical errors

http://python-textbok.readthedocs.io/en/1.0/Errors_and_Exceptions.html

Syntax Errors

- Python will find these kinds of errors when it tries to parse your program, and exit with an error message without running anything.
- Syntax errors are mistakes in the use of the Python language, and are like spelling or grammar mistakes
- Common Python syntax errors include:
  - leaving out a keyword
  - putting a keyword in the wrong place
  - leaving out a symbol, such as a colon, comma or brackets
  - misspelling a keyword
  - incorrect indentation
  - empty block

Runtime Errors

- If a program is syntactically correct – that is, free of syntax errors – it will be run by the Python interpreter
- Some problems are only revealed when a particular line is executed
- When a program comes to a halt because of a runtime error, we say that it has crashed
- Some examples of Python runtime errors:
  - division by zero
  - performing an operation on incompatible types
  - using an identifier which has not been defined
  - accessing a list element, dictionary value or object attribute which doesn't exist
  - trying to access a file which doesn't exist

Logical Errors

- Logical errors are the most difficult to fix
- They occur when the program runs without crashing, but produces an incorrect result
- The error is caused by a mistake in the program's logic. You won't get an error message, because no syntax or runtime error has occurred.
- Here are some examples of mistakes which lead to logical errors:
  - using the wrong variable name
  - indenting a block to the wrong level
  - using integer division instead of floating-point division
  - getting operator precedence wrong
  - making a mistake in a boolean expression
  - off-by-one, and other numerical errors

Exceptions

- Error handling is a necessary task, but how can you do it elegantly?
  - Errors aren’t ‘normal’ - you don’t make a system that *expects* errors! But you must handle error situations
  - One solution: return an error code. Used in C programs

Exceptions

- O-O languages solve error handling with **exceptions**
  - An independent ‘return path’ designed specifically for notifying the caller of an exceptional situation (=error)
  - On an error, a method ‘throws’ an exception
  - The calling method can ‘catch’ the exception
    - If caller doesn’t catch it, the exception is thrown to the next-higher caller
    - If no-one catches it, the exception causes the program to crash

Exceptions

- Python only lets objects of type Exception or its descendants to be thrown
- Python has a range of classes descending (inheriting, extends) from Exception
  - eg: ValueError, ZeroDivisionError
- You can define your own exception class, as long as it inherits from Exception (or one of it's subclasses)

Catching Exceptions

- Exceptions from different methods in different objects are often all caught at the one place in the calling method
  - Convenient: all error handling happens in one place
- Most languages use

  try .. **except** (catch) .. [**finally**] blocks:
  - **try**: define the set of statements whose exceptions will all be handled by the catch block associated with this try
  - **except**: processing to do if an exception is thrown in the try
  - **finally**: processing to always do regardless of whether an exception occurs or not.
    - Good for clean-up, eg: closing open files
    - This block is optional and executes after the try and catch blocks

Example

```
try:
    dividend = int(input("Please enter the dividend: "))
except ValueError:
    print("The dividend has to be a number!")

try:
    divisor = int(input("Please enter the divisor: "))
except ValueError:
    print("The divisor has to be a number!")

try:
    print("%d / %d = %f" % (dividend, divisor,
          dividend/divisor))
except ZeroDivisionError:
    print("The dividend cannot be zero!")
```

Error checks v's exception handling

```
# with checks
n = None
while n is None:
    s = input("Enter an integer: ")
    if s.lstrip('-').isdigit():
        n = int(s)
    else:
        print("%s is not an integer." % s)
```

```
# with exception handling
n = None
while n is None:
    try:
        s = input("Enter an integer: ")
        n = int(s)
    except ValueError:
        print("%s is not an integer." % s)
```

Raising Exceptions

- Python uses the **raise** keyword to throw exceptions
  - FYI - Java uses "throw"

```
if (invalid):
    raise ValueError(“invalid import");
```

- Note that we are creating an object and then throwing it = raise

Example

```
try:
    age = int(input("Please enter your age: "))
    if age < 0:
        raise ValueError(str(age) + " is not valid")

except ValueError as err:
    print("You entered incorrect age input:", err)

else:
    print("I see that you are", age, " years old.")

finally:
    print("It was really nice talking to you.\
            Goodbye!")
```

Exception Types

- Here are a few common exception types which we are likely to raise in our own code:
  - **TypeError**: this is an error which indicates that a variable has the wrong *type* for some operation.
    We might raise it in a function if a parameter is not of a type that we know how to handle.
  - **ValueError**: this error is used to indicate that a variable has the right *type* but the wrong *value*.
    For example, we used it when age was an integer, but the wrong *kind* of integer.
  - **NotImplementedError**: we will see in the next chapter how we use this exception to indicate that a class's method has to be implemented in a child class.

Example - Stacks

- stack + capacity + count
+ __init__() + getCount() + isEmpty() + isFull() + push() + pop() + top() + display()

- Week 2 of COMP1002
- Simulate a stack of items
- First in – first out
- Operations/behaviour
  - Push()
  - Pop()
  - Top()
  - isEmpty, isFull()
- ← UML Class Diagram

Unit testing example - Stacks

```
import numpy as np

class DSAStack ():

    def __init__(self, max_capacity=100):
        self.capacity = max_capacity
        self.stack = np.zeros(self.capacity)
        print(self.stack)
        self.count = 0

    def getCount(self):
        return self.count

    def isEmpty(self):
        return self.count==0

    def isFull(self):
        return self.count==self.capacity
```

```python
def push(self,value):
    if self.isFull():
        raise StackOverflowError("Stack is full")
    else:
        temp = self.count
        self.stack[temp] = value
        self.count = self.count + 1

def pop(self):
    topVal = self.top()
    self.count = self.count - 1
    return topVal

def top(self):
    topVal = -1
    if self.isEmpty():
        raise StackUnderflowError("Stack is already empty")
    else:
        temp = self.count
        topVal = self.stack[temp - 1]
    return topVal

def display(self):
    print(self.stack)
```

Stack exceptions – used in push() and top()

```
class Error(Exception):
    pass

class StackOverflowError(Error):
    """Exception raised if stack is overfull.

    Attributes:
        message -- explanation of the error
    """
    def __init__(self, message):
        self.message = message

class StackUnderflowError(Error):
    """Exception raised if stack is underfull.

    Attributes:
        message -- explanation of the error
    """
    def __init__(self, message):
        self.message = message
```

Stack Unit Tests (1/2)

```
from stacks import *

print("\nUnit tests for Stack class\n")

numpassed = 0
numtests = 0

# Test case 1 - create stack
numtests += 1
s = DSAStack()
if s.getCount() == 0:
    print("PASSED: Created stack successfully")
    numpassed += 1
else:
    print("FAILED: Could not create stack")


# Test case 2 - push values

numtests += 1
s.push(100)
s.push(200)
if s.getCount() == 2:
    print("PASSED: pushed two values on successfully")
    numpassed += 1
else:
    print("FAILED: Could not push values")
```

Stack Unit Tests (2/2)

```
# Test case 2 - pop values

numtests += 1
value1 = s.pop()
value2 = s.pop()
if value1 == 200 and value2 == 100:
    print("PASSED: popped two values off successfully")
    numpassed += 1
else:
    print("FAILED: Could not pop values")

# Test case 3 -  pop empty stack

numtests += 1
try:
    print(s.pop())
except StackUnderflowError as e:
    print("PASSED: Exception thrown as expected")
    numpassed += 1
else:
    print("FAILED: Popped value from empty stack")

# Results

print("\nPassed ", numpassed, " of ", numtests, " tests: ",
     100*numpassed/numtests, "%\n")
```

Summary

- Understand and apply object relationships: composition, aggregation and inheritance
- Understand and use exception handling
- Write code to work with and test classes

Practical Sessions

- We'll be coding object relationships and using them
- We'll do testing and use exception handling

Looking Ahead

- Next topic is Scripts and Automation
- Assignment has been released
- Tests will be marked during the week free

Next week…

- Developing programs
- Working with packages
- Version Control