

Object-Oriented Programming

Dr. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

08, 2016

- 1 **OOP Introduction**
- 2 **Case Study: Scala**
- 3 **Case Study: Python**

- Programs as collections of collaborating objects
- Object has public interface, hidden implementation
- Objects are classified according to their behavior
- Objects may represent real-world entities or entities that produce services for a program
- Objects may be reusable across programs

There are many object-oriented programming (OOP) languages

- Some are pure OOP language (e.g., Smalltalk).
- Newer languages do not support other paradigms but use their imperative structures (e.g., Java and C#).
- Some support procedural and data-oriented programming (e.g., Ada and C++).
- Some support functional program (e.g., Scala)

- Abstract data types
 - Encapsulation
 - Information Hiding
- Class Hierarchy
- Inheritance
- Polymorphism - Dynamic Binding

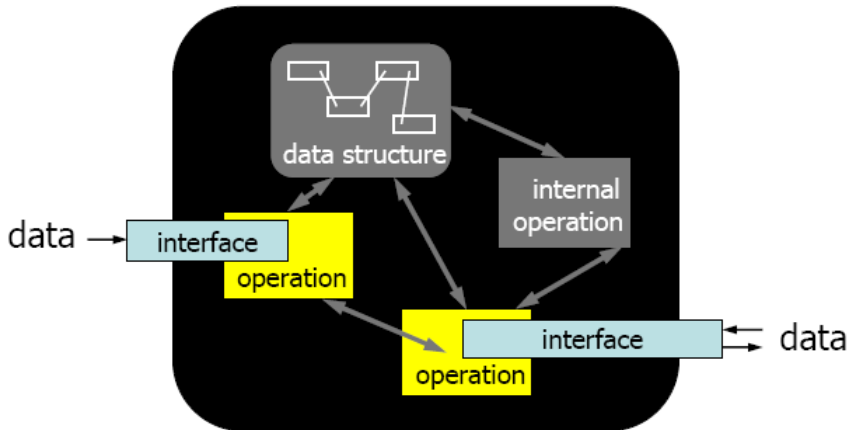
An abstract data type is data type that satisfies the following two conditions:

- **Encapsulation:**

- Attributes and operations are combined in a single syntactic unit
- compiled separately

- **Information Hiding:**

- The access to members are controlled
- Reliability increased



Operations on a stack:

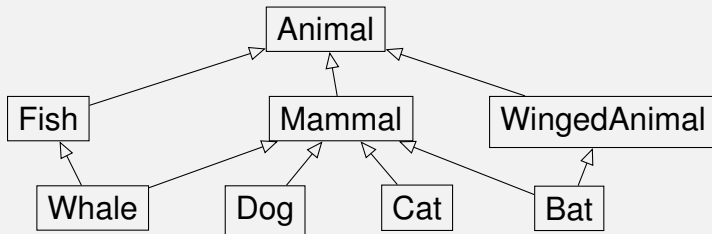
- `create(stack)`
- `destroy(stack)`
- `empty(stack)`
- `push(stack,element)`
- `pop(stack)`
- `top(stack)`

Client code:

```
...  
create ( stk1 );  
push( stk1 , color1 );  
push( stk1 , color2 );  
if ( ! empty( stk1 ) )  
    temp = top( stk1 );  
...
```

Implementation can be adjacent or linked list. Client: don't care!

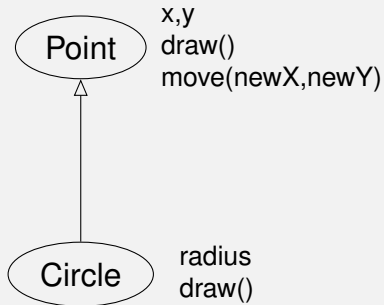
- A class may have some Subclasses
- A class may have one/many Superclass(es)

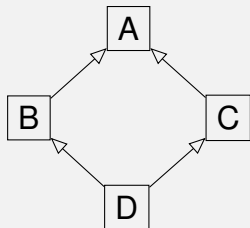


- A subclass is able to inherit **non-private** members of its superclasses
- A subclass can add its own members and **override** its inherited members
- Single vs. multiple inheritance
- Inheritance increases the reusability in OOP



Example of Inheritance





- if D inherits from B and C different versions of the same behaviour, which version will be effective in D?
- this problem, called diamond problem, is solved differently in different OO language
- is there the diamond problem in Java?

Class vs. Object

- A class defines the abstract characteristics of a thing
 - its attributes or properties and
 - its behaviors or methods or features.

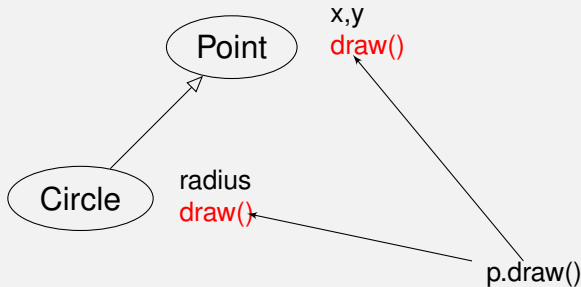
A Dog has *fur* and is able *to bark*

- An object is a particular instance of a class.
Lassie is a dog

Method vs. Message

- A method describes a behavior of an object.
A dog can bark
- A message is a process at which a method of an object is invoked.
Lassie barks

- Polymorphism: different objects can respond to the same message in different ways.



- Dynamic binding

- There are two kinds of variables in a class:
 - **Class variables**
 - Instance variables
- There are two kinds of methods in a class:
 - Class methods – accept messages to the class
 - Instance methods – accept messages to objects
- Abstract class vs. Concrete class:
 - Abstract Class – no instance
 - Concrete Class - able to have its instances

- Invented by Martin Odersky at EPFL, Lausanne, Switzerland.
- Similar to Java
- Work smoothly with Java
- Run on Java Virtual Machine
- OOP + FP
- Include lexer and parser generator



Class

- class [1]
- abstract class [5]
- trait [2]
- case class [3]

Object

- new <class name>
- <case class name>
- object

Example [7]

```
class Rational(n: Int, d: Int){  
  require(d != 0)  
  
  private val g = gcd(n.abs, d.abs)  
  private def gcd(a: Int, b: Int): Int =  
    if (b == 0) a else gcd(b, a % b)  
  val numer = n / g  
  val denom = d / g  
  
  def this(n: Int) = this(n, 1)  
  
  def + (that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
  
  override def toString = numer + "/" + denom  
}
```

Example on Abstract class [4]

```
abstract class Element {  
    def contents: Array[String] //no body: abstract  
    val height = contents.length  
    val width =  
        if (height == 0) 0 else contents(0).length  
}  
class ArrayElement(cons: Array[String])  
    extends Element {  
    def contents: Array[String] = cons  
}  
class LineElement(s: String)  
    extends ArrayElement(Array(s)) {  
    override def width = s.length  
    override def height = 1  
}
```

```
object Element {  
  
    def elem(contents: Array[String]): Element =  
        new ArrayElement(contents)  
  
    def elem(line: String): Element =  
        new LineElement(line)  
}  
  
val space = Element.elem(" ")  
val hello = Element.elem(Array("hello ", "world"))
```

Which kind of Element will be assigned to *space* and *hello*?

```
abstract class Expr
```

```
case class Var(name: String) extends Expr
```

```
case class Number(num: Double) extends Expr
```

```
case class UnOp(operator: String, arg: Expr)  
                                extends Expr
```

```
case class BinOp(operator: String,  
                  left: Expr, right: Expr) extends Expr
```

```
val v = Var("x")
```

```
val op = BinOp("+", Number(1), v)
```

```
v.name
```

```
op.left
```

Example 1 on Traits [9]

Reusability:

```
abstract class Bird
```

```
trait Flying {  
    def flyMess: String  
    def fly() = println(flyMess)  
}
```

```
trait Swimming {  
    def swim() = println("I'm swimming")  
}
```

```
class Penguin extends Bird with Swimming
```

```
class Hawk extends Bird with Swimming with Flying {  
    val flyMess = "I'm a good flyer"  
}
```

```
class Frigatebird extends Bird with Flying {  
    val flyMess = "I'm an excellent flyer"  
}
```



Example 2 on Traits [2]

Stackable Modifications:

```
abstract class IntQueue {  
  def get(): Int  
  def put(x: Int): Unit}  
class BasicIntQueue extends IntQueue {  
  private val buf = new ArrayBuffer[Int]  
  def get() = buf.remove(0)  
  def put(x: Int): Unit = { buf += x }  
trait Doubling extends IntQueue {  
  abstract override def put(x: Int) { super.put(2*x) }  
trait Incrementing extends IntQueue {  
  abstract override def put(x: Int) { super.put(x+1) }  
val queue =  
  new BasicIntQueue with Incrementing with Doubling  
queue.put(10)  
queue.get() //???
```

Trait Linearization: Doubling-> Incrementing->
BasicIntQueue-> IntQueue-> AnyRef-> Any

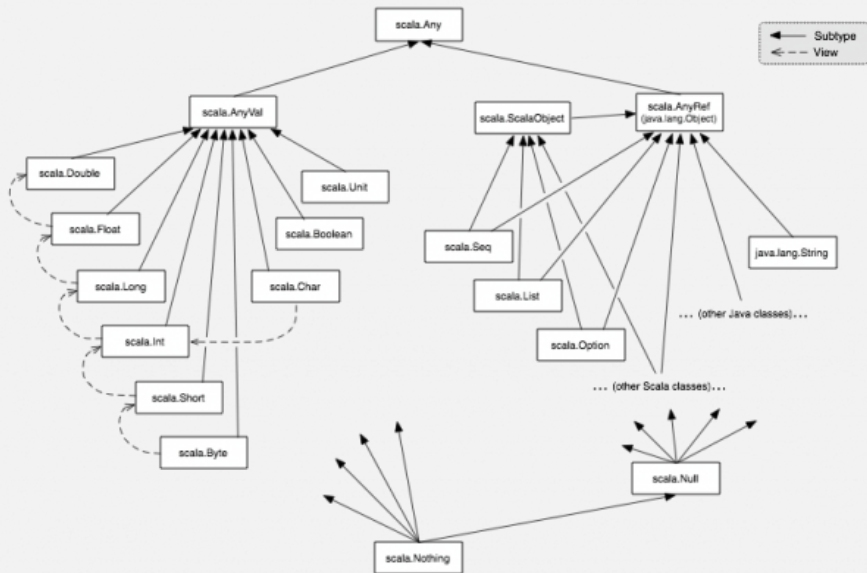
- A class extends just a single class but is able to have many "with" traits
- If these traits have the same method, which method is inherited
- Solution in Scala: Trait linearization
 - 1 Take the first extended trait/class and write its complete inherited hierarchy in vertical form, store this hierarchy as X
 - 2 Take the next trait/class after the with clause, write its complete hierarchy and cancel the classes or traits that are repeated in hierarchy X. Add the remaining traits/classes to the front of the hierarchy X.
 - 3 Go to step 2 and repeat the process, until no trait/class is left out.
 - 4 Place the class itself in front of hierarchy as head for which the hierarchy is being written.

Example on Trait Linearization

```
trait A {  
    def name: String  
}  
trait B extends A {  
    override def name: String = "class b"  
}  
trait C extends A {  
    override def name: String = "class c"  
}  
class D extends B with C {}  
var class_d = new D  
println(class_d.name)
```

- B: B->A->AnyRef->Any
- C: C->A->AnyRef->Any
- C->B->A->AnyRef->Any
- D: D->C->B->A->AnyRef->Any

Scala Hierarchy [8]



- protected
- private
- protected[<name>]
- private[<name>]

Example,

package Assignment

...

protected[Assignment] **val** field1 = 100

private[this] **val** field2 = 2;

- Class vs. Object
- Instance vs. static fields
- Instance, Class, Static methods
- Encapsulation
- Inheritance
- Polymorphism

- Class is a user-defined prototype for an object

```
class <Clsname>: => Clsname: name of the class  
    'Optional_class_documentation_string '  
    <class_suite>    => fields, methods
```

- Object: an instance of the class

```
class Employee:  
    'Common_base_class_for_all_employees '  
    empCount = 0    => empCount: static field  
    def __init__(self, n, s): => constructor  
        self.name = n    => name: instance field  
        self.salary = s    => salary: instance field  
        Employee.empCount += 1  
obj = Employee("Nam", 30) => create object
```

- Instance Method: method belongs to the instance

```
class Employee:
    def displayEmployee( self ): => first param-
        eter for instance
        print( "Name_:_", self.name,
                ",_Salary:_", self.salary )
obj = Employee( "Nam",30)
obj.displayEmployee() =>obj is passed to self
```

- able to access to instance fields through first parameter and '.' => self.name
- able to access to static field through class name and '.' => Employee.empCount

```
class Employee:
    @classmethod
    def create(cls, n, s):
        print(cls.empCount)
        return cls(n, s)
    @staticmethod
    def isHighSal(s):
        if s > 8:
            print("High_Salary")
obj = Employee.create("Nam", 30)
Employee.isHighSal(30)
```

=>to define class method
=>first parameter for class
=>access to static fields
=>to define static method
=>no parameter for class
=>unable to access any fields
=>Employee is passed to cls
=>Employee is used to resolve

- to hide fields and methods
- based on name of fields and methods
 - Protected: prefix by a single underscore (`_example`)
 - Private: prefix by a double underscores (`__example`)
 - Public: begin with a letter

- Python 3: root is **object**
- Multiple Inheritance

class <clsname>(<parent>(,<parent>)*)?:

- For example,

class A: => superclass is **object**

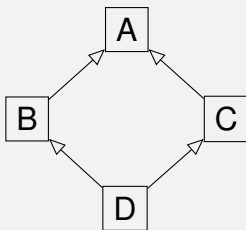
class Rectangle(Parallelogram):

=> superclass is **Parallelogram**

class Square(Rhombus, Rectangle):

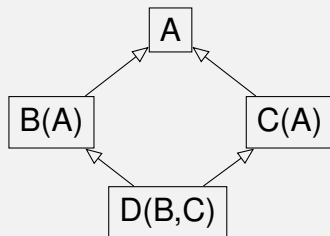
=> superclasses are **Rhombus** and **Rectangle**

- Subclass inherits non-private fields and methods from super-classes
- Diamond problem



- Method Resolution Order: determine search sequence of a class

- MRO is used to determine search sequence $L(M)$ of class M
 - $L(\text{object}) = [\text{object}]$
 - $L(M(A,B,C)) = [M] + \text{merge}(L(A),L(B),L(C),[A,B,C])$
- $\text{merge}([H_1|T_1],[H_2|T_2],[H_3|T_3])$
 - Step 1: if H_1 is a **good** head which is NOT in the tail of other lists, take H_1 out as an output, remove H_1 out of all lists, back to Step 1.
 - Step 2: if H_1 is not a good head, check if H_2 is a good head. If it is, apply Step1 for H_2 . If it is not, check for H_3 and so on. If there is no good head, give an error.



- $L(A) = [A, \text{object}]$
- $L(B) = [B] + \text{merge}(L(A), [A]) = [B, A, \text{object}]$
- $L(C) = [C] + \text{merge}(L(A), [A]) = [C, A, \text{object}]$
- $L(D) = [D] + \text{merge}(L(B), L(C), [B, C]) = [D, B, C, A, \text{object}]$
- $\text{merge}([B, A, o], [C, A, o], [B, C]) \Rightarrow$
 - B is a good head $\Rightarrow [B] + \text{merge}([A, o], [C, A, o], [C])$
 - A is NOT a good head, check C, and C is a good head $\Rightarrow [B, C] + \text{merge}([A, o], [A, o], [])$
 - A is a good head $\Rightarrow [B, C, A] + \text{merge}([o], [o], [])$
 - o is a good head $\Rightarrow [B, C, A, o]$

- **super()** => refer to the superclass

```
1 class A:
2     def foo(self, x):
3         print(x)
4 class B(A):
5     def foo(self, x):
6         super().foo(x)
7 B().foo(3) => 3
```

- **isinstance(o,T)** => check if object **o** is of type **T**

```
class A: pass
class B(A): pass
x = B()
isinstance(x,B) => True
isinstance(x,A) => True
type(x) is B => True
type(x) is A => False
```

- **type(o)** => return the type of object **o**

- **Overloading:**

```
def func(param1, param2=0): pass
func(1, 2)
func("asbc")
```

- **Universal Polymorphism:**

Parametric Polymorphism

```
class A:
    def func1(self):
        print("A")
    def func2(self): pass
class B:
    def func1(self):
        print("B")
for x in [A(), B()]:
    x.func1()
    x.func2()
```

Subtyping Polymorphism

```
class A:
    def func1(self):
        print("A")
    def func2(self): pass
class B(A):
    def func1(self):
        print("B")
for x in [A(), B()]:
    x.func1()
    x.func2()
```

- **id()**: address of the specified object

x,y,z = 3,3,4

id(x) => address of object 3 which x points to

id(y) => same above address => x, y -> same object

id(z) => z points to different object

- **is** vs. **==**

- **is**: True just when they are same object

- **==**: True even when they are different objects but their attributes are equal

x,y = [1,2,3],[1,2,3]

x **is** y => False

id(x) == **id(y)** => False

x == y => True

- Context manager by **with** statement

with <expression> **as** <variable>:
 <stmt-list>

- <**expression**> must return an object which has **__enter__** and **__exit__** methods
- The **with** statement is executed like:
 <variable> = <expression>
 <variable>.__enter__()
 <stmt_list>
 <variable>.__exit__()
- __exit__** method is always executed before the control goes out of the <**stmt_list**>
- used for managing resources: file, database,...

```
with open('abc.txt','r') as f:  
    print(f.read())  
f.closed => True
```


What are still in your mind?

- [1] **Classes and Objects**, <http://www.artima.com/pinsled/classes-and-objects.html>, **19 06 2014**.
- [2] **Traits**, <http://www.artima.com/pinsled/traits.html>, **19 06 2014**.
- [3] **Case Class and Pattern Matching**, <http://www.artima.com/pinsled/case-classes-and-pattern-matching.html>, **19 06 2014**.
- [4] **Abstract Members**, <http://www.artima.com/pinsled/abstract-members.html>, **19 06 2014**.
- [5] **Compositions and Inheritance**, <http://www.artima.com/pinsled/composition-and-inheritance.html>, **19 06 2014**.
- [6] **Packages and Imports**, <http://www.artima.com/pinsled/packages-and-imports.html>, **19 06 2014**.

- [7] **Functional Objects**, <http://www.artima.com/pinsled/functional-objects.html>, 19 06 2014.
- [8] **Scala Hierarchy**, <http://www.artima.com/pinsled/scalas-hierarchy.html>, 19 06 2014.
- [9] **Learning Scala part seven -Traits**, <http://joelabrahamsson.com/learning-scala-part-seven-traits/>, 19 06 2014.
- [10] **Object-Oriented Programming in Python 3**, <https://realpython.com/python3-object-oriented-programming/>, 01-10-2020.
- [11] **Python - Object-Oriented**, <https://realpython.com/python3-object-oriented-programming/>, 01-10-2020.
- [12] **The Python 2.3 Method Resolution Order**, <https://www.python.org/download/releases/2.3/mro/>, 01-10-2020.