**Q1. What is the meaning of multiple inheritance?**

Multiple inheritance means that a subclass can inherit from two or more superclasses. C++ allows multiple inheritance, but Java allows only single inheritance, that is, a subclass can inherit only one superclass.

Multiple Inheritance

source character.tcl

oo::class create warrior {

variable State

 constructor {name} {

 puts "Warrior constructor"

 next $name

 incr State(defense) 2

 incr State(attack) 2

 }

}

oo::class create human {

 variable State

 constructor {name} {

 puts "Human constructor"

 next $name

 incr State(hitpoints) 2

 }

}

oo::class create humanwarrior {

 superclass human warrior character

 variable State

 constructor {name} {

 puts "Human Warrior constructor"

 next $name

 }

}

puts "Creating a human warrior"

humanwarrior create elmer Sigfried

puts ""

puts "The character's attributes are:"

elmer show

puts ""

puts "Attack value 8 with no armor"

puts [elmer defense 8]

#### Script Output

**Creating a human warrior**

**Human Warrior constructor**

**Human constructor**

**Warrior constructor**

**Character constructor**

**The character's attributes are:**

**State(attack) = 4**

**State(defense) = 4**

**State(hitpoints) = 7**

**State(name) = Sigfried**

**Attack value 8 with no armor**

**Final Attack is: 8**

**8 is larger than 4, Sigfried is Hit**

**Q2. What is the concept of delegation?**

Delegation is commonly defined as the shifting of authority and responsibility for particular functions, tasks or decisions from one person (usually a leader or manager) to another.

**Delegation at its core**

Regardless of how widely or narrowly one defines the term, delegation is the leader’s decision to:

1. **Lead** by providing vision and direction.
2. **Trust others** and empowering them to deliver on that trust.
3. **Develop beyond a leaders’ personal capacity** by tapping into others’ unique abilities and opinions.
4. **Build capacity in others** through training and new experiences.
5. **Replace him/her self** with others who can do the same job, freeing them to contribute elsewhere.

**Delegation-friendly (and not so friendly) situations**

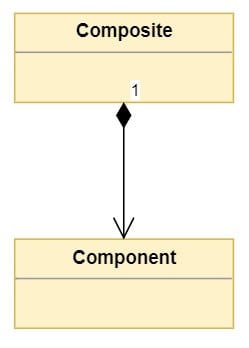
The Situational Leadership Model factors in more than a subordinate’s experience and expertise. Environment also plays a role in determining whether a leader should direct, collaborate with or delegate to a subordinate. Let’s analyze these along the continuum of crisis to stable environments.

1. **Crisis :** Leaders who are dealing with crises have neither the time not the bandwidth to work subordinates through the process detailed above. In most cases, the leader will need to assume an authoritarian or directive role in mobilizing others toward desired outcomes. When dealing with very experienced, expert subordinates, a more participative approach is recommended.
2. **Changing/High-growth :** In fluid environments that are active but not crisis-ridden, leaders should seek to use a more collaborative approach so long as the subordinate possesses at least moderate levels of capacity and know-how.
3. **Stable :** This is the kind of environment in which delegation is most effective.

Q**3. What is the concept of composition?**

Composition is a concept that models a has a relationship. It enables creating complex types by combining objects of other types. This means that a class Composite can contain an object of another class Component . This relationship means that a Composite has a Component ..

UML represents composition as follows:

[](https://files.realpython.com/media/ic-basic-composition.8a15876f7db2.jpg)

Composition is represented through a line with a diamond at the composite class pointing to the component class. The composite side can express the cardinality of the relationship. The cardinality indicates the number or valid range of Component instances the Composite class will contain.

In the diagram above, the 1 represents that the Composite class contains one object of type Component. Cardinality can be expressed in the following ways:

* **A number** indicates the number of Component instances that are contained in the Composite.
* **The \* symbol** indicates that the Composite class can contain a variable number of Component instances.
* **A range 1..4** indicates that the Composite class can contain a range of Component instances. The range is indicated with the minimum and maximum number of instances, or minimum and many instances like in **1..\***.

For example, your Horse class can be composed by another object of type Tail. Composition allows you to express that relationship by saying a Horse **has a** Tail.

**Q4. What are bound methods and how do we use them?**

A bound method is the one which is dependent on the instance of the class as the first argument. It passes the instance as the first argument which is used to access the variables and functions. In Python 3 and newer versions of python, all functions in the class are by default bound methods.

Let’s understand this concept with an example:

|  |
| --- |
| **# Python code to demonstrate**  **# use of bound methods**    **class A:**    **def func(self, arg):**  **self.arg = arg**  **print("Value of arg = ", arg)**      **# Creating an instance**  **obj = A()**    **# bound method**  **print(obj.func)** |

**Q5. What is the purpose of pseudoprivate attributes?**

One of the main problems that the pseudoprivate attribute feature is meant to alleviate has to do with the way instance attributes are stored. In Python, all instance attributes wind up in the single instance object at the bottom of the class tree. This is different from the C++ model, where each class gets its own space for data members it defines.

Within a class method in Python, whenever a method assigns to a self attribute (e.g., self.attr = value), it changes or creates an attribute in the instance ([inheritance searches](https://www.pythonstudio.us/object-oriented/attribute-inheritance-search.html) happen only on reference, not on assignment). Because this is true even if multiple classes in a hierarchy assign to the same attribute, collisions are possible.

Why Use Pseudo-Private Attributes?

The problem that the pseudo-private attribute feature is meant to alleviate has to do with the way instance attributes are stored. In Python, all instance attributes wind up in the single instance object at the bottom of the class tree. This is very different from the C++ model, where each class gets its own space for data members it defines.

Within a class method in Python, whenever a method assigns to a self attribute (e.g., self.attr=value), it changes or creates an attribute in the instance (inheritance search only happens on reference, not assignment). Because this is true even if multiple classes in a hierarchy assign to the same attribute, collisions are possible.

For example, suppose that when a programmer codes a class, she assumes that she owns the attribute name X in the instance. In this class's methods, the name is set and later fetched:

class C1: def meth1(self): self.X = 88 # Assume X is mine. def meth2(self): print self.X