Ministry of Education of Republic of Moldova

Technical University of Moldova

CIM Faculty

Anglophone Department

**Report**

*on APPOO*

Laboratory Work Nr. 0

Performed by st. gr. FAF-131 **Danilescu Dan**

Verified by lect.univ **Drahnea Vlad**

Chișinău 2016

**Inheritance**

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

## Base & Derived Classes:

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

**C++**

Consider a base class **Shape** and its derived class **Rectangle** as follows:

#include <iostream>

using namespace std;

// Base class

class Shape

{

public:

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

// Derived class

class Rectangle: public Shape

{

public:

int getArea()

{

return (width \* height);

}

};

int main(void)

{

Rectangle Rect;

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

return 0;

}

Swift:

 Inheritance is a fundamental behavior that differentiates classes from other types in Swift.

A simple example of inheritance:

**class** AClass {

**func** doSomething() {

println("Hello from AClass")

}

}

**class** **Subclass**: AClass {

}

**let** base\_object **=** AClass()

base\_object.doSomething()

*//> Hello from AClass*

**let** enhanced\_object **=** Subclass()

enhanced\_object.doSomething()

*// > Hello from AClass*

**Encapsulation**

Encapsulation is one of the most important object-oriented design principles: It hides the internal states and functionality of objects.

**Swift:**

You can achieve this by using the access control features of Swift.

*“You can assign specific access level to individual types (classes, structures, and enumerations), as well to properties, methods, initilizers, and subscripts belonging to those types. Protocols can be restricted to a certain context, as can global constants, variables, and functions.”*

So in essence this means you can use access control for everything you define except local variables

public class SomePublicClass {}

internal class SomeInternalClass {}

private class SomePrivateClass {}

public var somePublicVariable = 0

internal let someInternalConstant = 0

private func somePrivateFunction() {}

**Access Control for SubClasses:**

Swift allows the user to subclass any class that can be accessed in the current access context. A subclass cannot have a higher access level than its superclass. The user is restricted from writing a public subclass of an internal superclass.

public class cricket {

private func print() {

println("Welcome to Swift Super Class")

}

}

internal class tennis: cricket {

override internal func print() {

println("Welcome to Swift Sub Class")

}

}

let cricinstance = cricket()

cricinstance.print()

let tennisinstance = tennis()

tennisinstance.print()

**C++**

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example:

#include <iostream>

using namespace std;

class Adder{

public:

// constructor

Adder(int i = 0)

{

total = i;

}

// interface to outside world

void addNum(int number)

{

total += number;

}

// interface to outside world

int getTotal()

{

return total;

};

private:

// hidden data from outside world

int total;

};

int main( )

{

Adder a;

a.addNum(10);

a.addNum(20);

a.addNum(30);

cout << "Total " << a.getTotal() <<endl;

return 0;

}

**Polymorphism**

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

**C++**

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

#include <iostream>

using namespace std;

class Shape {

protected:

int width, height;

public:

Shape( int a=0, int b=0)

{

width = a;

height = b;

}

int area()

{

cout << "Parent class area :" <<endl;

return 0;

}

};

class Rectangle: public Shape{

public:

Rectangle( int a=0, int b=0):Shape(a, b) { }

int area ()

{

cout << "Rectangle class area :" <<endl;

return (width \* height);

}

};

class Triangle: public Shape{

public:

Triangle( int a=0, int b=0):Shape(a, b) { }

int area ()

{

cout << "Triangle class area :" <<endl;

return (width \* height / 2);

}

};

// Main function for the program

int main( )

{

Shape \*shape;

Rectangle rec(10,7);

Triangle tri(10,5);

// store the address of Rectangle

shape = &rec;

// call rectangle area.

shape->area();

// store the address of Triangle

shape = &tri;

// call triangle area.

shape->area();

return 0;

}

Swift:

Here is a simple example in which multiple instances can be used as a GraphicObject.

**class** GraphicObject {

**func** draw() {

println("does nothing")

}

}

**class** **SpaceShip**: GraphicObject {

}

**class** **EmpireSpaceShip**: SpaceShip {

**override** **func** draw() {

println("draws an empire space ship")

}

}

**class** **RebellionSpaceShip**: SpaceShip {

**override** **func** draw() {

println("draws a rebellion space ship")

}

}

**class** **DeathStar**: GraphicObject {

**override** **func** draw() {

println("draws the Death Star")

}

}

**var** spaceShips **=** [EmpireSpaceShip(), RebellionSpaceShip(), DeathStar()]

**for** spaceShip **in** spaceShips {

spaceShip.draw()

}

**Task 2:**

|  |  |
| --- | --- |
| **Swift** | **C++** |
| **Encapsulation** | |
| friend class and friend functions | Mixin’s |
| **Polymorphism** | |
| full separation of polymorphism and inheritance | polymorphism and inheritance are linked |
| **Inheritance** | |
| one inheritance | multi-inheritance |
| **Other differences** | |
| all methods are always virtual | not all |
| it’s *self* instead of *this* | must be *this* |
| parentheses for method calls are optional | we have to put them |