Şafak AKINCI **26.12.2016**
152120141003

Experiment 7
Inheritence

Objectives

To create classes by inheriting from existing classes.
The notions of base classes and derived classes and the relationships between them.

Prelab Activities

Correct The Code

1-	10	:	protected:	:	compilation error
	15	:	class X : public Y	:	compilation error
2-	7	:	public	:	compilation error
3-	24	:	class B : public A	:	compilation error
4-	16	:	class Y : public X	:	logic error
	20	:	Y()	:	compilation error
	28	:	yObject	:	compilation error

Lab Exercises

Lab Exercise 1 - Football Players

```
/*****************************************
 * Player.h                              *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence             *
 *****************************************/


//Player.h includes base class, including iostream header in this file
//allows that not include the iostream header for each header file.
#include <iostream>

//To declare string type variables.
#include <string>

//Define string type under the namespace of std.
using namespace std;


class Player {
//Protected variables can be accessed from derived classes.
protected:
    int age;
    string name;
//Public variables can be accessed from derived classes too.
public:
    enum Direction
    {
        EAST,
        NORTH,
        NORTH_EAST,
        NORTH_WEST,
        SOUTH,
        SOUTH_EAST,
        SOUTH_WEST,
        WEST
    };

    //Returns age which is the PROTECTED member variable of Player Class.
    int getAge() const;

    //Returns name which is the PROTECTED member variable of Player Class.
    const string& getName() const;

    //Calls private functions of Player Class.
    void Move(Direction direction);

    //Constructor Function of Player Class.
    //Directly derived classes from Player class must initialize name and age(PROTECTED) variables!
    Player(const string& name, int age);

    //Destructor Function of Player Class.
    ~Player();

//Private Functions CANNOT BE ACCESSED out of the class.
private:
    //Move functions print that player is moving to given direction in MOVE FUNCTION(public).
    void MoveEast();
    void MoveNorth();
    void MoveNorthEast();
    void MoveNorthWest();
    void MoveSouth();
    void MoveSouthEast();
    void MoveSouthWest();
    void MoveWest();
};


/******************************************
 * Player.cpp                             *
 ******************************************
 * IDE : Xcode                          *
 * Author : Şafak AKINCI                *
 * Experiment 7: Inheritence            *
 ****************************************/

#include "Player.h"             //To know function prototypes of Player Class.

/*  using namespace std;  was included in "Player.h" header, that's why we didn't include it again here.  */


//Constructor Function of Player Class.
//Directly derived classes from Player class must initialize name and age(PROTECTED) variables!
Player::Player(const string& name, int age):name(name),age(age)
{
}

//Destructor Function of Player Class.
Player::~Player()
{
}

//Returns age which is the PROTECTED member variable of Player Class.
//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
int Player::getAge() const
{
    return age;
}

//Returns name which is the PROTECTED member variable of Player Class.
//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
const string& Player::getName() const
{
    return name;
}

//Calls private functions of Player Class.
void Player::Move(Player::Direction direction)
{
    switch (direction) {
        case Direction::EAST:
            MoveEast();
            break;
        case Direction::NORTH:
            MoveNorth();
            break;
        case Direction::NORTH_EAST:
            MoveNorthEast();
```

```cpp
            break;
        case Direction::NORTH_WEST:
            MoveNorthWest();
            break;
        case Direction::SOUTH:
            MoveSouth();
            break;
        case Direction::SOUTH_EAST:
            MoveSouthEast();
            break;
        case Direction::SOUTH_WEST:
            MoveSouthWest();
            break;
        case Direction::WEST:
            MoveWest();
            break;
    }//end switch
}//end Player::Move()

//Prints that player is moving to given direction in MOVE FUNCTION (public).
void Player::MoveEast()
{
    //name is protected member of Player Class, it can be accessed from FootballPlayer Class and its functions 
because of the PROTECTED defined.
    cout<<name<<" is moving to EAST\n";
}

//Prints that player is moving to given direction in MOVE FUNCTION (public).
void Player::MoveNorth()
{
    //name is protected member of Player Class, it can be accessed from FootballPlayer Class and its functions 
because of the PROTECTED defined.
    cout<<name<<" is moving to NORTH\n";
}

//Prints that player is moving to given direction in MOVE FUNCTION (public).
void Player::MoveNorthEast()
{
    //name is protected member of Player Class, it can be accessed from FootballPlayer Class and its functions 
because of the PROTECTED defined.
    cout<<name<<" is moving to NORTH-EAST\n";
}

//Prints that player is moving to given direction in MOVE FUNCTION (public).
void Player::MoveNorthWest()
{
    //name is protected member of Player Class, it can be accessed from FootballPlayer Class and its functions
```

```
because of the PROTECTED defined.
    cout<<name<<" is moving to NORTH-WEST\n";
}

//Prints that player is moving to given direction in MOVE FUNCTION (public).
void Player::MoveSouth()
{
    //name is protected member of Player Class, it can be accessed from FootballPlayer Class and its functions
because of the PROTECTED defined.
    cout<<name<<" is moving to SOUTH\n";
}

//Prints that player is moving to given direction in MOVE FUNCTION (public).
void Player::MoveSouthEast()
{
    //name is protected member of Player Class, it can be accessed from FootballPlayer Class and its functions
because of the PROTECTED defined.
    cout<<name<<" is moving to SOUTH-EAST\n";
}

//Prints that player is moving to given direction in MOVE FUNCTION (public).
void Player::MoveSouthWest()
{
    //name is protected member of Player Class, it can be accessed from FootballPlayer Class and its functions
because of the PROTECTED defined.
    cout<<name<<" is moving to SOUTH-WEST\n";
}

//Prints that player is moving to given direction in MOVE FUNCTION (public).
void Player::MoveWest()
{
    //name is protected member of Player Class, it can be accessed from FootballPlayer Class and its functions
because of the PROTECTED defined.
    cout<<name<<" is moving to WEST\n";
}


/******************************************
 * DefensivePlayer.h                      *
 ******************************************
 * IDE : Xcode                            *
 * Author : Şafak AKINCI                  *
 * Experiment 7: Inheritence              *
 ******************************************/

/*
 This is a preprocessor technique of preventing a header file from being included multiple times,
 which can be problematic for various reasons.

 During compilation of the project, each .cpp file (usually) is compiled.

 The compiler will take .cpp file, open any files #included by it, concatenate them all into one massive text file,
 and then perform syntax analysis and finally it will convert it to some intermediate code,
 optimize/perform other tasks, and finally generate the assembly output for the target architecture.

 ******************************************************************************************
 *    Because of this, if a file is #included multiple times under one .cpp file,        *
 *    the compiler will append its file contents twice,                                   *
 *    so if there are definitions within that file,                                       *
 *    we will get a compiler error telling us that we redefined a variable.               *
 ******************************************************************************************

 When the file is processed by the preprocessor step in the compilation process,
 the first time its contents are reached the first two lines will check if FILE_H has been defined for the
preprocessor.
 If not, it will define FILE_H and continue processing the code between it and the #endif directive.

 The next time that file's contents are seen by the preprocessor, the check against FILE_H will be false,
 so it will immediately scan down to the #endif and continue after it.
 THIS PREVENTS REDEFINITION ERRORS.
 */

#ifndef DefensivePlayer_h
#define DefensivePlayer_h

#include "FootballPlayer.h"             //To derive DefensivePlayer class from FootballPlayer Class.

//DefensivePlayer class is derived from FootballPlayer class.
//DefensivePlayer class inherits FootballPlayer Class' functions and variables as public.

//  The important part is that FootballPlayer Class is also derived from Player Class so,
//DefensivePlayer Class is indirectly derived from Player Class.

/*
            Player Class
                ^
                |
                |
        FootballPlayer Class
                ^
                |
                |
        DefensivePlayer Class
 */
```

```
//This means, DefensivePlayer Class is also inherits functions and members of Player Class too.

class DefensivePlayer:public FootballPlayer
{

//Private variables CANNOT BE ACCESSED out of the class.
private:
    double paymentPerMatch;
    int playedMatchCount;

//Public variables can be accessed from derived classes too.
public:

    //Constructor Function of DefensivePlayer Class.
        //  This class is derived from FootballPlayer Class,
        //so this constructor function has to call its base class' constructor function(FootballPlayer Class'
Construction).
    DefensivePlayer(const string& name, int age, double paymentPerYear, double paymentPerMatch);

    //Calculates the payment for DefensivePlayer for this year.
    double CalculatePaymentForThisYear() const;

    //Prints that player who called this function is defending.
    void Defense() const;

    //Returns playedMatchCount which is the PRIVATE member variable of DefensivePlayer Class.
    int getPlayedMatchCount() const;

    //Prints that player who called this function is tripped up.
    void TripUp(const Player* player) const;

    //Sets playedMatchCount which is the PRIVATE member variable of DefensivePlayer Class.
    void setPlayedMatchCount(int playedMatchCount);

    //Destructor Function of DefensivePlayer Class.
    ~DefensivePlayer();
};

#endif /* DefensivePlayer_hpp */

/*****************************************
 * DefensivePlayer.cpp                   *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence             *
 *****************************************/

#include "DefensivePlayer.h"            //To know function prototypes of DefensivePlayer Class.
//DefensivePlayer.h also includes FootballPlayer.h header, that's why we don't also include FootballPlayer.h header
here (also Player.h).

//Constructor Function of DefensivePlayer Class.
//This class is derived from FootballPlayer Class, so this constructor function has to call its base class'
constructor function(FootballPlayer Class' Construction).
DefensivePlayer::DefensivePlayer(const string& name, int age, double paymentPerYear, double
paymentPerMatch):FootballPlayer(name,age,paymentPerYear)
{
    (*this).paymentPerMatch = paymentPerMatch;
}

//Destructor Function of DefensivePlayer Class.
DefensivePlayer::~DefensivePlayer()
{

}

//Calculates the payment for DefensivePlayer for this year.
//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
double DefensivePlayer::CalculatePaymentForThisYear() const
{
    return paymentPerYear + paymentPerMatch * playedMatchCount;
}

//Prints that player who called this function is defending.
//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
void DefensivePlayer::Defense() const
{
    //name is protected member of Player Class, it can be accessed from FootballPlayer Class and its functions
because of the PROTECTED defined.
    cout<<name<<" is defending...\n";
}

//Returns playedMatchCount which is the PRIVATE member variable of DefensivePlayer Class.
//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
int DefensivePlayer::getPlayedMatchCount() const
{
    return playedMatchCount;
}

//Prints that player who called this function is tripped up.
//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
void DefensivePlayer::TripUp(const Player* player) const
```

```
{
    //name is protected member of Player Class, it can be accessed from FootballPlayer Class and its functions
because of the PROTECTED defined.
    cout<<name<<" tripped up "<<player->getName()<<endl;
}

//Sets playedMatchCount which is the PRIVATE member variable of DefensivePlayer Class.
void DefensivePlayer::setPlayedMatchCount(int playedMatchCount)
{
    (*this).playedMatchCount = playedMatchCount;
}

/*****************************************
 * FootballPlayer.h                      *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence             *
 *****************************************/

/*
    This is a preprocessor technique of preventing a header file from being included multiple times,
 which can be problematic for various reasons.

    During compilation of the project, each .cpp file (usually) is compiled.

    The compiler will take .cpp file, open any files #included by it, concatenate them all into one massive text
file,
 and then perform syntax analysis and finally it will convert it to some intermediate code,
 optimize/perform other tasks, and finally generate the assembly output for the target architecture.

        *************************************************************************************
        *    Because of this, if a file is #included multiple times under one .cpp file,        *
        *    the compiler will append its file contents twice,                                  *
        *    so if there are definitions within that file,                                      *
        *    we will get a compiler error telling us that we redefined a variable.              *
        *************************************************************************************

    When the file is processed by the preprocessor step in the compilation process,
 the first time its contents are reached the first two lines will check if FILE_H has been defined for the
preprocessor.
    If not, it will define FILE_H and continue processing the code between it and the #endif directive.

    The next time that file's contents are seen by the preprocessor, the check against FILE_H will be false,
                    so it will immediately scan down to the #endif and continue after it.
                                THIS PREVENTS REDEFINITION ERRORS.
 */
```

```
#ifndef FootballPlayer_h
#define FootballPlayer_h

#include "Player.h"       //To derive FootballPlayer Class from Player Class which defined in Player.h header.

//FootballPlayer class is derived from Player Class.
//FootballPlayer class inherits Player Class' functions and variables as public.
//In this case Player Class is base class.
class FootballPlayer:public Player
{
//Protected variables can be accessed from derived classes.
protected:
    double paymentPerYear;

//Public variables can be accessed from derived classes too.
public:
    //Constructor Function of FootballPlayer Class.
        //  This class is derived from Player Class,
        //so this constructor function has to call its base class' constructor function(Player Class' Construction).
    FootballPlayer(const string& name, int age, double paymentPerYear);

    //Returns paymentPerYear which is the PROTECTED member variable of FootballPlayer Class.
    double getPaymentPerYear() const;

    //Sets paymentPerYear which is the PROTECTED member variable of FootballPlayer Class.
    void setPaymentPerYear(double paymentPerYear);

    //Prints that player who called this function is passing to given player.
    void Pas (const Player* player);

    //Prints that player who called this function is shotting.
    void Shot();

    //Destructor Function of FootballPlayer Class.
    ~FootballPlayer();
};

#endif /* FootballPlayer_h */

/*****************************************
 * ForwardPlayer.cpp                     *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence             *
 *****************************************/
```

```
#include "ForwardPlayer.h"              //To know function prototypes of ForwardPlayer Class.
//ForwardPlayer.h also includes FootballPlayer.h header, that's why we don't also include FootballPlayer.h header
here (also Player.h).


//Constructor Function of ForwardPlayer Class.
//  This class is derived from FootballPlayer Class,so this constructor function has to call its base class'
constructor function(FootballPlayer Class' Construction).
ForwardPlayer::ForwardPlayer(const string& name, int age, double paymentPerYear, double
paymentPerGoal):FootballPlayer(name,age,paymentPerYear)
{
    (*this).paymentPerGoal = paymentPerGoal;
}

//Destructor Function of ForwardPlayer Class.
ForwardPlayer::~ForwardPlayer()
{

}

//Calculates the payment for ForwardPlayer for this year.
//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
double ForwardPlayer::CalculatePaymentForThisYear() const
{
    return paymentPerYear + paymentPerGoal * goalCount;
}

//Returns goalCount which is the PRIVATE member variable of ForwardPlayer Class.
//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
int ForwardPlayer::getGoalCount() const
{
    return goalCount;
}

//Sets goalCount which is the PRIVATE member variable of ForwardPlayer Class.
void ForwardPlayer::setGoalCount(int goalCount)
{
    (*this).goalCount = goalCount;
}

//Prints that player who called this function is throwing himself.
//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
void ForwardPlayer::ThrowYourSelf() const
{
    //name is protected member of Player Class, it can be accessed from FootballPlayer Class and its functions
because of the PROTECTED defined.
    cout<<name<<" is throwing himself...\n";
}

//Prints that player who called this function is trying a dribble past.
//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
void ForwardPlayer::TryDribblePast(const Player* player) const
{
    //name is protected member of Player Class, it can be accessed from FootballPlayer Class and its functions
because of the PROTECTED defined.
    cout<<name<<" is trying a dribble past on "<<player->getName()<<endl;
}

/*****************************************
 * GoalKeeper.h                          *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence             *
 *****************************************/

/*
 This is a preprocessor technique of preventing a header file from being included multiple times,
 which can be problematic for various reasons.

 During compilation of the project, each .cpp file (usually) is compiled.

 The compiler will take .cpp file, open any files #included by it, concatenate them all into one massive text file,
 and then perform syntax analysis and finally it will convert it to some intermediate code,
 optimize/perform other tasks, and finally generate the assembly output for the target architecture.

 *****************************************************************************************
 *    Because of this, if a file is #included multiple times under one .cpp file,       *
 *    the compiler will append its file contents twice,                                  *
 *    so if there are definitions within that file,                                      *
 *    we will get a compiler error telling us that we redefined a variable.              *
 *****************************************************************************************

 When the file is processed by the preprocessor step in the compilation process,
 the first time its contents are reached the first two lines will check if FILE_H has been defined for the
preprocessor.
 If not, it will define FILE_H and continue processing the code between it and the #endif directive.

 The next time that file's contents are seen by the preprocessor, the check against FILE_H will be false,
 so it will immediately scan down to the #endif and continue after it.
 THIS PREVENTS REDEFINITION ERRORS.
 */

#ifndef GoalKeeper_h
```

```
#define GoalKeeper_h

#include "FootballPlayer.h"         //To derive GoalKeeper class from FootballPlayer Class.

//GoalKeeper class is derived from FootballPlayer class.
//GoalKeeper class inherits FootballPlayer Class' functions and variables as public.

//  The important part is that FootballPlayer Class is also derived from Player Class so,
//GoalKeeper Class is indirectly derived from Player Class.

/*
            Player Class
                ^
                |
                |
        FootballPlayer Class
                ^
                |
                |
         GoalKeeper Class
 */

//This means, GoalKeeper Class is also inherits functions and members of Player Class too.
class GoalKeeper:public FootballPlayer
{

//Public variables can be accessed from derived classes too.
public:
    //Construction Function of GoalKeeper Class.
    //This class is derived from FootballPlayer Class and it was derived from Player Class.
    //  name, age and paymentPerYear variables that function take should be initialized by FootballPlayer
Constructor,
    //  and it will also call the constructor function of Player Class to initialize name and age members.
    //**  paymentPerYear is PROTECTED member variable of FootballPlayer Class.
    GoalKeeper(const string name, int age, double paymentPerYear);

    //Destructor Function of GoalKeeper Class.
    ~GoalKeeper();

    //Calculates the payment for GoalKeeper for this year.
    double CalculatePaymentForThisYear() const;

    //Prints that player who called this function is flying and trying to catch.
    void FlyAndTryCatch() const;
};
#endif /* GoalKeeper_h */
```

```
/****************************************
 * DefensivePlayer.cpp                  *
 ****************************************
 * IDE : Xcode                          *
 * Author : Şafak AKINCI                *
 * Experiment 7: Inheritence            *
 ****************************************/

#include "GoalKeeper.h"         //To know function prototypes of GoalKeeper Class.
//GoalKeeper.h also includes FootballPlayer.h header, that's why we don't also include FootballPlayer.h header here
(also Player.h).

//Constructor function of GoalKeeper class.
//This class is derived from FootballPlayer Class, so this constructor function has to call its base class'
constructor function(FootballPlayer Class' Construction).
GoalKeeper::GoalKeeper(const string name, int age, double paymentPerYear):FootballPlayer(name,age,paymentPerYear)
{

}

//Destructor function of GoalKeeper class.
GoalKeeper::~GoalKeeper()
{

}

//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
double GoalKeeper::CalculatePaymentForThisYear() const
{
    return paymentPerYear;
}

//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
void GoalKeeper::FlyAndTryCatch() const
{

    //name is protected member of Player Class, it can be accessed from GoalKeeper Class and its functions because of
the PROTECTED defined.
    cout<<name<<" is flying and trying to catch...\n";
}

/****************************************
 * MiddleFielderPlayer.h                *
 ****************************************
 * IDE : Xcode                          *
 * Author : Şafak AKINCI                *
 * Experiment 7: Inheritence             *
 ******************************************/

/*
 This is a preprocessor technique of preventing a header file from being included multiple times,
 which can be problematic for various reasons.

 During compilation of the project, each .cpp file (usually) is compiled.

 The compiler will take .cpp file, open any files #included by it, concatenate them all into one massive text file,
 and then perform syntax analysis and finally it will convert it to some intermediate code,
 optimize/perform other tasks, and finally generate the assembly output for the target architecture.

 ****************************************************************************************
 *    Because of this, if a file is #included multiple times under one .cpp file,        *
 *    the compiler will append its file contents twice,                                  *
 *    so if there are definitions within that file,                                      *
 *    we will get a compiler error telling us that we redefined a variable.              *
 ****************************************************************************************

 When the file is processed by the preprocessor step in the compilation process,
 the first time its contents are reached the first two lines will check if FILE_H has been defined for the
preprocessor.
 If not, it will define FILE_H and continue processing the code between it and the #endif directive.

 The next time that file's contents are seen by the preprocessor, the check against FILE_H will be false,
 so it will immediately scan down to the #endif and continue after it.
 THIS PREVENTS REDEFINITION ERRORS.
 */

#ifndef MiddleFielderPlayer_h
#define MiddleFielderPlayer_h

#include "FootballPlayer.h"             //To derive MidFielderPlayer class from FootballPlayer Class.

//MidFielderPlayer class is derived from FootballPlayer class.
//MidFielderPlayer class inherits FootballPlayer Class' functions and variables as public.

//  The important part is that FootballPlayer Class is also derived from Player Class so,
//MidFielderPlayer Class is indirectly derived from Player Class.

/*
            Player Class
                ^
                |
                |
        FootballPlayer Class
```

```
                ^
                |
                |
        MidFielderPlayer Class
 */

//This means, MidFielderPlayer Class is also inherits functions and members of Player Class too.

class MidFielderPlayer:public FootballPlayer
{

//Private variables CANNOT BE ACCESSED out of the class.
private:
    int assistCount;
    double paymentPerAssist;

//Public variables can be accessed from derived classes too.
public:

    //Constructor Function of MidFielderPlayer Class.
        //  This class is derived from FootballPlayer Class,
        //so this constructor function has to call its base class' constructor function(FootballPlayer Class'
Construction).
    MidFielderPlayer(const string& name, int age, double paymentPerYear, double paymentPerAssist);

    //Destructor Function of MidFielderPlayer Class.
    ~MidFielderPlayer();

    //Calculates the payment for MidFielderPlayer for this year.
    double CalculatePaymentForThisYear() const;

    //Returns assistCount is the PRIVATE member variable of MidFielderPlayer Class.
    int getAssistCount() const;

    //Prints that player who called this function is pressing.
    void Press() const;

    //Prints that player who called this function is trying a through ball.
    void TryThroughBall(const Player* player) const;

    //Sets assistCount which is the PRIVATE member variable of MidFielderPlayer Class.
    void setAssistCount(int assistCount);
};

#endif /* MiddleFielderPlayer_hpp */

/*****************************************
```

```
 * MiddleFielderPlayer.cpp               *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence             *
 *****************************************/

#include "MiddleFielderPlayer.h"            //To know function prototypes of MiddleFielderPlayer Class.
//MiddleFielderPlayer.h also includes FootballPlayer.h header, that's why we don't also include FootballPlayer.h
header here (also Player.h).


//Constructor Function of MidFielderPlayer Class.
    //  This class is derived from FootballPlayer Class,
    //so this constructor function has to call its base class' constructor function(FootballPlayer Class'
Construction).
MidFielderPlayer::MidFielderPlayer(const string& name, int age, double paymentPerYear, double
paymentPerAssist):FootballPlayer(name,age,paymentPerYear)
{
    (*this).paymentPerAssist = paymentPerAssist;
}

//Destructor Function of MidFielderPlayer Class.
MidFielderPlayer::~MidFielderPlayer()
{

}

//Calculates the payment for MidFielderPlayer for this year.
//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
double MidFielderPlayer::CalculatePaymentForThisYear() const
{
    return paymentPerYear + paymentPerAssist * assistCount;
}

//Returns assistCount is the PRIVATE member variable of MidFielderPlayer Class.
//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
int MidFielderPlayer::getAssistCount() const
{
    return assistCount;
}

//Prints that player who called this function is pressing.
//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
void MidFielderPlayer::Press() const
{
    //name is protected member of Player Class, it can be accessed from FootballPlayer Class and its functions
```

```
because of the PROTECTED defined.
    cout<<name<<" is pressing...\n";
}

//Prints that player who called this function is trying a through ball.
//Function should be const because it is an accessor function (It shouldn't change the class' variables.)
void MidFielderPlayer::TryThroughBall(const Player* player) const
{
    //name is protected member of Player Class, it can be accessed from FootballPlayer Class and its functions
because of the PROTECTED defined.
    cout<<name<<" is trying a through ball to "<<player->getName()<<endl;
}

//Sets assistCount which is the PRIVATE member variable of MidFielderPlayer Class.
void MidFielderPlayer::setAssistCount(int assistCount)
{
    (*this).assistCount = assistCount;
}

/*****************************************
 * TestMain.cpp                          *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence             *
 *****************************************/

//Derived Classes
#include "DefensivePlayer.h"
#include "GoalKeeper.h"
#include "MiddleFielderPlayer.h"
#include "ForwardPlayer.h"

void TEST_GoalKeeper()
{
    cout <<"+-------------------+" << endl
    << "| GOAL KEEPER TEST |" << endl
    << "+-------------------+" << endl;

    //keeper pointer object is created from GoalKeeper Class in HEAP MEMORY.
    GoalKeeper* keeper = new GoalKeeper ("Iker Casillas", 34, 7.5);

    //keeper is moving given direction.
    keeper->Move(Player::Direction::EAST);

    //keeper is moving given direction.
    keeper->Move(Player::Direction::NORTH_EAST);
```

```cpp
    //keeper is shotting.
    keeper->Shot();

    //player pointer object is created from DefensivePlayer Class in HEAP MEMORY.
    DefensivePlayer* player = new DefensivePlayer ("Raphael Varane", 22, 1.2, 0.1);

    //keeper is passing.
    keeper->Pas(player);

    //keeper is flying and trying to catch.
    keeper->FlyAndTryCatch();

    //Payment for keeper will print to console.
    cout<<"Payment for this year: "<<keeper->CalculatePaymentForThisYear()<<" million EUR"<<endl;

}//end TEST_GoalKeeper ()

void TEST_DefensivePlayer()
{
    cout <<"+-----------------------+" << endl
    << "| DEFENSIVE PLAYER TEST |" << endl
    << "+-----------------------+" << endl;

    //defense pointer object is created from DefensivePlayer Class in HEAP MEMORY.
    DefensivePlayer* defense = new DefensivePlayer ("Raphael Varane", 22, 1.2, 0.1);
    defense->Move(Player::Direction::EAST);
    defense->Move(Player::Direction::NORTH_EAST);

    defense->Shot();

    MidFielderPlayer* player = new MidFielderPlayer ("Gareth Bale", 25, 15, 0.2);
    defense->Pas(player);

    defense->Defense();

    ForwardPlayer* opponent = new ForwardPlayer ("Lionel Messi", 28, 20, 0.3);
    defense->TripUp(opponent);

    defense->setPlayedMatchCount(11);

    //Payment for defense will print to console.
    cout<<"Payment for this year: "<<defense->CalculatePaymentForThisYear()<<" million EUR"<<endl;

}//TEST_DefensivePlayer()

void TEST_MiddleFieldPlayer()
```

```cpp
{
    cout <<"+----------------------------+" << endl
    << "| MIDDLE FIELDER PLAYER TEST |" << endl
    << "+----------------------------+" << endl;

    //middleFielder pointer object is created from MidFielderPlayer Class in HEAP MEMORY.
    MidFielderPlayer* middleFielder = new MidFielderPlayer ("Gareth Bale", 25, 15, 0.2);
    middleFielder->Move(Player::Direction::EAST);
    middleFielder->Move(Player::Direction::NORTH_EAST);

    middleFielder->Shot();

    ForwardPlayer* player = new ForwardPlayer ("Cristiano Ronaldo", 30, 17, 0.3);
    middleFielder->Pas(player);

    middleFielder->Press();
    middleFielder->TryThroughBall(player);

    middleFielder->setAssistCount(22);

    //Payment for keeper will print to console.
    cout<<"Payment for this year: "<<middleFielder->CalculatePaymentForThisYear()<<" million EUR"<<endl;

}//end TEST_MiddleFieldPlayer()

void TEST_ForwardPlayer()
{
    cout<<"+---------------------+" << endl
    <<"| FORWARD PLAYER TEST |" << endl
    <<"+---------------------+" << endl;

    //forward pointer object is created from ForwardPlayer Class in HEAP MEMORY.
    ForwardPlayer* forward = new ForwardPlayer ("Cristiano Ronaldo", 30, 17, 0.3);
    forward->Move(Player::Direction::EAST);
    forward->Move(Player::Direction::NORTH_EAST);

    forward->Shot();

    ForwardPlayer* player = new ForwardPlayer ("Karim Benzema", 27, 8, 0.3);
    forward->Pas(player);

    forward->ThrowYourSelf();

    DefensivePlayer* opponent = new DefensivePlayer ("Gerard Pique", 28, 5.8, 0.1);
    forward->TryDribblePast(opponent);

    forward->setGoalCount(42);
    cout<<"Payment for this year: "<<forward->CalculatePaymentForThisYear()<<" million EUR"<<endl;

}//end TEST_ForwardPlayer()



int main ()
{

    TEST_GoalKeeper();
    TEST_DefensivePlayer();
    TEST_MiddleFieldPlayer();
    TEST_ForwardPlayer();

    //Indicates that program ended successfully.
    return 0;
}//end main()
```

AT QUIZ

```
/******************************************
 * Animal.h                               *
 ******************************************
 * IDE : Xcode                            *
 * Author : Şafak AKINCI                  *
 * Experiment 7: Inheritence QUIZ         *
 ******************************************/

/*
 This is a preprocessor technique of preventing a header file from being included multiple times,
 which can be problematic for various reasons.

 During compilation of the project, each .cpp file (usually) is compiled.

 The compiler will take .cpp file, open any files #included by it, concatenate them all into one massive text file,
 and then perform syntax analysis and finally it will convert it to some intermediate code,
 optimize/perform other tasks, and finally generate the assembly output for the target architecture.

 *****************************************************************************************
 *    Because of this, if a file is #included multiple times under one .cpp file,        *
 *    the compiler will append its file contents twice,                                  *
 *    so if there are definitions within that file,                                      *
 *    we will get a compiler error telling us that we redefined a variable.              *
 *****************************************************************************************
 When the file is processed by the preprocessor step in the compilation process,
 the first time its contents are reached the first two lines will check if FILE_H has been defined for the
preprocessor.
 If not, it will define FILE_H and continue processing the code between it and the #endif directive.

 The next time that file's contents are seen by the preprocessor, the check against FILE_H will be false,
 so it will immediately scan down to the #endif and continue after it.
 THIS PREVENTS REDEFINITION ERRORS.
 */

#ifndef Animal_h
#define Animal_h

//Animal.h includes base class, including iostream header in this file
//allows that not include the iostream header for each file.
#include <iostream>

//To declare string type variables.
#include <string>

//Define string type under the namespace of std.
using namespace std;


//Animal class is the base class.
class Animal
{

//Protected variables can be accessed from derived classes.
protected:
    string m_name;
    int m_age;

//Public variables can be accessed from derived classes too.
public:

    //Constructor Function of Animal Class.
    Animal(const string& name, int age);

    //Returns m_name which is the PROTECTED member variable of Animal Class.
    string getName() const;

    //Returns m_age which is the PROTECTED member variable of Animal Class.
    int getAge() const;

    //Sets m_name which is the PROTECTED member variable of Animal Class.
    void setName(string name);
    //Sets m_age which is the PROTECTED member variable of Animal Class.
    void setAge(int age);
};

#endif


/*****************************************
 * Animal.cpp                            *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence QUIZ        *
 *****************************************/

//To know function prototypes of Animal Class.
#include "Animal.h"


//Constructor Function of Animal Class.
//m_name and m_age are initialized in memory initializer scope.
Animal::Animal(const string& name, int age):m_name(name),m_age(age)
{

}

//Returns m_name which is the PROTECTED member variable of Animal Class.
//Function should be const, because it is an accessor function, it won't change any member of Class.
string Animal::getName() const
{
    return m_name;
}

//Returns m_age which is the PROTECTED member variable of Animal Class.
//Function should be const, because it is an accessor function, it won't change any member of Class.
int Animal::getAge() const
{
    return m_age;
}

//Sets m_name which is the PROTECTED member variable of Animal Class.
void Animal::setName(string name)
{
    m_name = name;
}
```

```
//Sets m_age which is the PROTECTED member variable of Animal Class.
void Animal::setAge(int age)
{
    m_age = age;
}

/*****************************************
 * Cat.h                                 *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence QUIZ        *
 *****************************************/
/*
 This is a preprocessor technique of preventing a header file from being included multiple times,
 which can be problematic for various reasons.

 During compilation of the project, each .cpp file (usually) is compiled.

 The compiler will take .cpp file, open any files #included by it, concatenate them all into one massive text file,
 and then perform syntax analysis and finally it will convert it to some intermediate code,
 optimize/perform other tasks, and finally generate the assembly output for the target architecture.

 *****************************************************************************************
 *    Because of this, if a file is #included multiple times under one .cpp file,       *
 *    the compiler will append its file contents twice,                                  *
 *    so if there are definitions within that file,                                      *
 *    we will get a compiler error telling us that we redefined a variable.              *
 *****************************************************************************************

 When the file is processed by the preprocessor step in the compilation process,
 the first time its contents are reached the first two lines will check if FILE_H has been defined for the
preprocessor.
 If not, it will define FILE_H and continue processing the code between it and the #endif directive.

 The next time that file's contents are seen by the preprocessor, the check against FILE_H will be false,
 so it will immediately scan down to the #endif and continue after it.
 THIS PREVENTS REDEFINITION ERRORS.
 */

#ifndef Cat_h
#define Cat_h

#include "Mammal.h"         //To derive Cat class from Mammal Class.


//Cat class is derived from Mammal Class.
```

```
//Cat class inherits Mammal Class' functions and variables as public.
//In this case Mammal Class is base class.

//  The important part is that Mammal Class is also derived from Animal Class so,
//Cat Class is indirectly derived from Animal Class.

/*
        Animal Class
            ^
            |
            |
        Mammal Class
            ^
            |
            |
        Cat Class
 */

//This stands for Cat Class is also inherits functions and members of Animal Class too.
class Cat: public Mammal
{
    //Private variables CAN NOT BE ACCESSED from derived classes.
private:
    int m_miaowSoundLevel;

    //Public variables can be accessed from derived classes too.
public:

    //Constructor Function of Cat Class.
    Cat(const string& name, int age, int miaowSoundLevel);

    //Prints that cat which called this function is miaowing and its sound level.
    void Miaow() const;
};


#endif /* Cat_h */

/*****************************************
 * Cat.cpp                               *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence QUIZ        *
 *****************************************/

#include "Cat.h"        //To know function prototypes of Cat Class.
```

```
//Constructor Function of Cat Class.
//Cat Class is derived from Mammal Class, so Cat Class' Constructor Function HAS TO CALL MAMMAL CLASS' CONSTRUCTOR.
Cat::Cat(const string& name, int age, int miaowSoundLevel):Mammal(name,age)
{
    //m_miaowSoundLevel is the private member of Cat Class.
    m_miaowSoundLevel = miaowSoundLevel;
}


//Prints that cat which called this function is miaowing and its sound level.
//Function should be const, because it is an accessor function, it won't change any member of Class.
void Cat::Miaow() const
{
    //m_name is the protected member of Animal Class, we can reach it from Cat class because of the PROTECTED 
DEFINED.
    cout<<m_name<<" is miaowing and its sound level is:\t"<<m_miaowSoundLevel<<endl;
}

/*****************************************
 * Crocodile.h                           *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence QUIZ        *
 *****************************************/

/*
 This is a preprocessor technique of preventing a header file from being included multiple times,
 which can be problematic for various reasons.

 During compilation of the project, each .cpp file (usually) is compiled.

 The compiler will take .cpp file, open any files #included by it, concatenate them all into one massive text file,
 and then perform syntax analysis and finally it will convert it to some intermediate code,
 optimize/perform other tasks, and finally generate the assembly output for the target architecture.

 *****************************************************************************************
 *    Because of this, if a file is #included multiple times under one .cpp file,       *
 *    the compiler will append its file contents twice,                                  *
 *    so if there are definitions within that file,                                      *
 *    we will get a compiler error telling us that we redefined a variable.              *
 *****************************************************************************************

 When the file is processed by the preprocessor step in the compilation process,
 the first time its contents are reached the first two lines will check if FILE_H has been defined for the 
preprocessor.
```

```
 If not, it will define FILE_H and continue processing the code between it and the #endif directive.

 The next time that file's contents are seen by the preprocessor, the check against FILE_H will be false,
 so it will immediately scan down to the #endif and continue after it.
 THIS PREVENTS REDEFINITION ERRORS.
 */

#ifndef Crocodile_h
#define Crocodile_h

#include "Reptile.h"        //To derive Crocodile class from Reptile Class.


//Crocodile class is derived from Reptile Class.
//Crocodile class inherits Reptile Class' functions and variables as public.
//In this case Reptile Class is base class.

//  The important part is that Reptile Class is also derived from Animal Class so,
//Crocodile Class is indirectly derived from Animal Class.

/*
        Animal Class
            ^
            |
            |
        Reptile Class
            ^
            |
            |
        Crocodile Class
 */

//This means, Crocodile Class is also inherits functions and members of Animal Class too.

class Crocodile:public Reptile
{

//Protected variables can be accessed from derived classes.
protected:
    int m_roarSoundLevel;

//Public variables can be accessed from derived classes too.
public:

    //Constructor Function of Crocodile Class.
    Crocodile(const string& name, int age, int roarSoundLevel);
    //Prints that crocodile which called this function is roaring and its sound level.
    void Roar() const;

};

#endif /* Crocodile_h */


/*****************************************
 * Crocodile.cpp                         *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence QUIZ        *
 *****************************************/

#include "Crocodile.h"          //To know function prototypes of Crocodile Class.

//Constructor Function of Crocodile Class.
//Crocodile Class is derived from Reptile Class, so Crocodile Class' Constructor Function HAS TO CALL REPTILE CLASS'
CONSTRUCTOR.
Crocodile::Crocodile(const string& name, int age, int roarSoundLevel):Reptile(name,age)
{
    //m_roarSoundLevel is the protected member of Crocodile Class.
    m_roarSoundLevel = roarSoundLevel;
}


//Prints that crocodile which called this function is roaring and its sound level.
//Function should be const, because it is an accessor function, it won't change any member of Class.
void Crocodile::Roar() const
{
    //m_name is the protected member of Animal Class, we can reach it from Crocodile class because of the PROTECTED
DEFINED.
    cout<<m_name<<" is roaring and its sound level is:\t"<<m_roarSoundLevel<<endl;
}

/*****************************************
 * Dog.h                                 *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence QUIZ        *
 *****************************************/

/*
 This is a preprocessor technique of preventing a header file from being included multiple times,
 which can be problematic for various reasons.

 During compilation of the project, each .cpp file (usually) is compiled.

 The compiler will take .cpp file, open any files #included by it, concatenate them all into one massive text file,
 and then perform syntax analysis and finally it will convert it to some intermediate code,
 optimize/perform other tasks, and finally generate the assembly output for the target architecture.

 *****************************************************************************************
 *    Because of this, if a file is #included multiple times under one .cpp file,        *
 *    the compiler will append its file contents twice,                                  *
 *    so if there are definitions within that file,                                      *
 *    we will get a compiler error telling us that we redefined a variable.              *
 *****************************************************************************************

 When the file is processed by the preprocessor step in the compilation process,
 the first time its contents are reached the first two lines will check if FILE_H has been defined for the
preprocessor.
 If not, it will define FILE_H and continue processing the code between it and the #endif directive.

 The next time that file's contents are seen by the preprocessor, the check against FILE_H will be false,
 so it will immediately scan down to the #endif and continue after it.
 THIS PREVENTS REDEFINITION ERRORS.
 */
#ifndef Dog_h
#define Dog_h

#include "Mammal.h"             //To derive Dog class from Mammal Class.


//Dog class is derived from Mammal Class.
//Dog class inherits Mammal Class' functions and variables as public.
//In this case Mammal Class is base class.

//  The important part is that Mammal Class is also derived from Animal Class so,
//Dog Class is indirectly derived from Animal Class.

/*
            Animal Class
                ^
                |
                |
            Mammal Class
                ^
                |
                |
            Dog Class
 */

//This stands for Dog Class is also inherits functions and members of Animal Class too.

class Dog:public Mammal
{
//Protected variables can be accessed from derived classes.
protected:
    int m_barkSoundLevel;

//Public variables can be accessed from derived classes too.
public:

    //Constructor Function of Dog Class.
    Dog(const string& name, int age, int barkSoundLevel);

    //Prints that dog which called this function is barking and its sound level.
    void Bark() const;
};

#endif /* Dog_h */

/*****************************************
 * Dog.cpp                               *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence QUIZ        *
 *****************************************/

#include "Dog.h"            //To know function prototypes of Dog Class.

//Constructor Function of Dog Class.
//Dog Class is derived from Mammal Class, so Dog Class' Constructor Function HAS TO CALL MAMMAL CLASS' CONSTRUCTOR.
Dog::Dog(const string& name, int age, int barkSoundLevel):Mammal(name,age)
{
    //m_barkSoundLevel is the protected member of Dog Class.
    m_barkSoundLevel = barkSoundLevel;
}

//Prints that dog which called this function is barking and its sound level.
//Function should be const, because it is an accessor function, it won't change any member of Class.
void Dog::Bark() const
{
    //m_name is the protected member of Animal Class, we can reach it from Dog class because of the PROTECTED
DEFINED.
    cout<<m_name<<" is barking and its sound level is:\t"<<m_barkSoundLevel<<endl;
}


/*****************************************
 * Mammal.h                             *
 *****************************************
 * IDE : Xcode                          *
 * Author : Şafak AKINCI                *
 * Experiment 7: Inheritence QUIZ       *
 *****************************************/

/*
 This is a preprocessor technique of preventing a header file from being included multiple times,
 which can be problematic for various reasons.

 During compilation of the project, each .cpp file (usually) is compiled.

 The compiler will take .cpp file, open any files #included by it, concatenate them all into one massive text file,
 and then perform syntax analysis and finally it will convert it to some intermediate code,
 optimize/perform other tasks, and finally generate the assembly output for the target architecture.

 *****************************************************************************************
 *    Because of this, if a file is #included multiple times under one .cpp file,       *
 *    the compiler will append its file contents twice,                                 *
 *    so if there are definitions within that file,                                     *
 *    we will get a compiler error telling us that we redefined a variable.             *
 *****************************************************************************************

 When the file is processed by the preprocessor step in the compilation process,
 the first time its contents are reached the first two lines will check if FILE_H has been defined for the
preprocessor.
 If not, it will define FILE_H and continue processing the code between it and the #endif directive.

 The next time that file's contents are seen by the preprocessor, the check against FILE_H will be false,
 so it will immediately scan down to the #endif and continue after it.
 THIS PREVENTS REDEFINITION ERRORS.
 */

#ifndef Mammal_h
#define Mammal_h

#include "Animal.h"                 //To derive Mammal class from Animal Class.


//Mammal class is derived from Animal Class.
//Mammal class inherits Animal Class' functions and variables as public.
//In this case Animal Class is base class.
```

```
class Mammal:public Animal
{
//Public variables can be accessed from derived classes too.
public:

    //Constructor Function of Mammal Class.
    Mammal(const string& name, int age);

    //Prints that mammal that called this function is breeding.
    void Breed() const;
};


#endif /* Mammal_h */


/*****************************************
 * Mammal.cpp                            *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence QUIZ        *
 *****************************************/

#include "Mammal.h"             //To know function prototypes of Mammal Class.

//Constructor Function of Mammal Class.
//Mammal Class is derived from Animal Class, so Mammal Class' Constructor Function HAS TO CALL ANIMAL CLASS'
CONSTRUCTOR.
Mammal::Mammal(const string& name, int age):Animal(name,age)
{}

//Prints that mammal which called this function is breeding.
//Function should be const, because it is an accessor function, it won't change any member of Class.
void Mammal::Breed() const
{
    //m_name is the protected member of Animal Class, we can reach it from Mammal class because of the PROTECTED
DEFINED.
    cout<<m_name<<" is breeding!\n";
}

/*****************************************
 * Reptile.h                             *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence QUIZ        *
 ****************************************/

/*
 This is a preprocessor technique of preventing a header file from being included multiple times,
 which can be problematic for various reasons.

 During compilation of the project, each .cpp file (usually) is compiled.

 The compiler will take .cpp file, open any files #included by it, concatenate them all into one massive text file,
 and then perform syntax analysis and finally it will convert it to some intermediate code,
 optimize/perform other tasks, and finally generate the assembly output for the target architecture.

 ****************************************************************************************
 *    Because of this, if a file is #included multiple times under one .cpp file,        *
 *    the compiler will append its file contents twice,                                  *
 *    so if there are definitions within that file,                                      *
 *    we will get a compiler error telling us that we redefined a variable.              *
 ****************************************************************************************

 When the file is processed by the preprocessor step in the compilation process,
 the first time its contents are reached the first two lines will check if FILE_H has been defined for the
preprocessor.
 If not, it will define FILE_H and continue processing the code between it and the #endif directive.

 The next time that file's contents are seen by the preprocessor, the check against FILE_H will be false,
 so it will immediately scan down to the #endif and continue after it.
 THIS PREVENTS REDEFINITION ERRORS.
 */

#ifndef Reptile_h
#define Reptile_h

#include "Animal.h"             //To derive Reptile class from Animal Class.


//Reptile class is derived from Animal Class.
//Reptile class inherits Animal Class' functions and variables as public.
//In this case Animal Class is base class.
class Reptile:public Animal
{

//Public variables can be accessed from derived classes.
public:

    //Constructor Function of Reptile Class.
    Reptile(const string& name, int age);
    //Prints that reptile which called this function is ovulating.
    void Ovulate() const;
};

#endif /* Reptile_h */

/******************************************
 * Reptile.cpp                            *
 ******************************************
 * IDE : Xcode                            *
 * Author : Şafak AKINCI                  *
 * Experiment 7: Inheritence QUIZ         *
 ******************************************/

#include "Reptile.h"        //To know function prototypes of Reptile Class.


//Constructor Function of Reptile Class.
//Reptile Class is derived from Animal Class, so Reptile Class' Constructor Function HAS TO CALL ANIMAL CLASS'
CONSTRUCTOR.
Reptile::Reptile(const string& name, int age):Animal(name,age)
{

}

//Prints that reptile which called this function is ovulating.
//Function should be const, because it is an accessor function, it won't change any member of Class.
void Reptile::Ovulate() const
{
    //m_name is the protected member of Animal Class, we can reach it from Reptile class because of the PROTECTED
DEFINED.
    cout<<m_name<<" is ovulating!\n";
}


/******************************************
 * Snake.h                                *
 ******************************************
 * IDE : Xcode                            *
 * Author : Şafak AKINCI                  *
 * Experiment 7: Inheritence QUIZ         *
 ******************************************/

/*
 This is a preprocessor technique of preventing a header file from being included multiple times,
 which can be problematic for various reasons.
 During compilation of the project, each .cpp file (usually) is compiled.

 The compiler will take .cpp file, open any files #included by it, concatenate them all into one massive text file,
 and then perform syntax analysis and finally it will convert it to some intermediate code,
 optimize/perform other tasks, and finally generate the assembly output for the target architecture.

 *****************************************************************************************
 *    Because of this, if a file is #included multiple times under one .cpp file,       *
 *    the compiler will append its file contents twice,                                  *
 *    so if there are definitions within that file,                                      *
 *    we will get a compiler error telling us that we redefined a variable.              *
 *****************************************************************************************

 When the file is processed by the preprocessor step in the compilation process,
 the first time its contents are reached the first two lines will check if FILE_H has been defined for the
preprocessor.
 If not, it will define FILE_H and continue processing the code between it and the #endif directive.

 The next time that file's contents are seen by the preprocessor, the check against FILE_H will be false,
 so it will immediately scan down to the #endif and continue after it.
 THIS PREVENTS REDEFINITION ERRORS.
 */

#ifndef Snake_h
#define Snake_h

#include "Reptile.h"            //To derive Snake class from Reptile Class.

//Snake class is derived from Reptile Class.
//Snake class inherits Reptile Class' functions and variables as public.
//In this case Reptile Class is base class.

//  The important part is that Reptile Class is also derived from Animal Class so,
//Cat Class is indirectly derived from Animal Class.

/*
            Animal Class
                ^
                |
                |
            Reptile Class
                ^
                |
                |
            Snake Class
 */
```

```
//This stands for Snake Class is also inherits functions and members of Animal Class too.

class Snake:public Reptile
{
//Private variables CAN NOT BE ACCESSED from derived classes.
private:
    int m_sizzleSoundLevel;

//Public variables can be accessed from derived classes too.
public:

    //Constructor Function of Snake Class.
    Snake(const string& name, int age, int sizzleSoundLevel);

    //Prints that snake which called this function is sizzling and its sound level.
    void Sizzle() const;
};

#endif /* Snake_h */

/*****************************************
 * Snake.cpp                            *
 *****************************************
 * IDE : Xcode                          *
 * Author : Şafak AKINCI                *
 * Experiment 7: Inheritence QUIZ       *
 *****************************************/

#include "Snake.h"          //To know function prototypes of Snake Class.


//Constructor Function of Snake Class.
//Snake Class is derived from Reptile Class, so Snake Class' Constructor Function HAS TO CALL REPTILE CLASS'
CONSTRUCTOR.
Snake::Snake(const string& name, int age, int sizzleSoundLevel):Reptile(name,age)
{
    //m_sizzleSoundLevel is the private member of Snake Class.
    m_sizzleSoundLevel = sizzleSoundLevel;
}

//Prints that snake which called this function is sizzling and its sound level.
//Function should be const, because it is an accessor function, it won't change any member of Class.
void Snake::Sizzle() const
{
    //m_name is the protected member of Animal Class, we can reach it from Snake class because of the PROTECTED
DEFINED.
    cout<<m_name<<" is sizzling and its sound level is:\t"<<m_sizzleSoundLevel<<endl;
```

```
}

/*****************************************
 * TestMain.cpp                          *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 7: Inheritence QUIZ        *
 *****************************************/

//Only Derived Classes are included, we don't have the include the header where the base class defined.
//Because, derived classes' header files have already included them, there is no need to do it again.
#include "Dog.h"
#include "Cat.h"
#include "Snake.h"
#include "Crocodile.h"

void TEST_Animal()
{
    cout <<"+-------------+" << endl
    << "| ANIMAL TEST |" << endl
    << "+-------------+" << endl;


    //animal is created from Animal Class. It should take name and age parameters.
    Animal animal("Mouse",10);

    //animal's name and age will print to console.
    cout<<animal.getName()<<" is "<<animal.getAge()<<" years"<<endl;

    //animal's name will change as the given parameter.
    animal.setName("FARE");

    //animal's age will change as the given parameter.
    animal.setAge(11);

    //animal's name and age will print to console.
    cout<<animal.getName()<<" is "<<animal.getAge()<<" years"<<endl;
}

void TEST_Mammal()
{
    cout <<"+-------------+" << endl
    << "| MAMMAL TEST |" << endl
    << "+-------------+" << endl;

    //animal is created from Mammal Class. It should take name and age parameters.
    Mammal animal("Dolphin",10);

    //animal's name and age will print to console.
    cout<<animal.getName()<<" is "<<animal.getAge()<<" years"<<endl;

    //animal's name will change as the given parameter.
    animal.setName("DOLPHIN");

    //animal's age will change as the given parameter.
    animal.setAge(11);

    //animal's name and age will print to console.
    cout<<animal.getName()<<" is "<<animal.getAge()<<" years"<<endl;

    //Prints that, animal is breeding.
    animal.Breed();
}

void TEST_Reptile()
{
    cout <<"+---------------+" << endl
    << "| REPTILE TEST |" << endl
    << "+---------------+" << endl;

    //animal is created from Reptile Class. It should take name and age parameters.
    Reptile animal("Reptile Crocodile",10);

    //animal's name and age will print to console.
    cout<<animal.getName()<<" is "<<animal.getAge()<<" years"<<endl;

    //animal's name will change as the given parameter.
    animal.setName("REPTILE CROCODILE");

    //animal's age will change as the given parameter.
    animal.setAge(11);

    //animal's name and age will print to console.
    cout<<animal.getName()<<" is "<<animal.getAge()<<" years"<<endl;

    //Prints that animal is ovulating.
    animal.Ovulate();
}

void TEST_Dog()
{
    cout <<"+-----------+" << endl
    << "| DOG TEST |" << endl
```

```cpp
    << "+-----------+" << endl;

    //animal is created from Dog Class. It should take name, age and soundLevel parameters.
    Dog animal("Little Dog",10,90);

    //animal's name and age will print to console.
    cout<<animal.getName()<<" is "<<animal.getAge()<<" years"<<endl;

    //animal's name will change as the given parameter.
    animal.setName("LITTLE DOG");

    //animal's age will change as the given parameter.
    animal.setAge(11);

    //animal's name and age will print to console.
    cout<<animal.getName()<<" is "<<animal.getAge()<<" years"<<endl;

    //Prints that animal is barking.
    animal.Bark();
}

void TEST_Cat()
{
    cout <<"+-----------+" << endl
    << "| CAT TEST |" << endl
    << "+-----------+" << endl;

    //animal is created from Cat Class. It should take name, age and soundLevel parameters.
    Cat animal("Panthera Tulliana Pardus",10,90);

    //animal's name and age will print to console.
    cout<<animal.getName()<<" is "<<animal.getAge()<<" years"<<endl;

    //animal's name will change as the given parameter.
    animal.setName("PANTHERA TULLIANA PARDUS");

    //animal's age will change as the given parameter.
    animal.setAge(11);

    //animal's name and age will print to console.
    cout<<animal.getName()<<" is "<<animal.getAge()<<" years"<<endl;

    //Prints that animal is miaowing.
    animal.Miaow();
}

void TEST_Snake()
```

```
{
    cout <<"+------------+" << endl
    << "| SNAKE TEST |" << endl
    << "+------------+" << endl;

    //animal is created from Snake Class. It should take name, age and soundLevel parameters.
    Snake animal("Black Mamba",10,90);

    //animal's name will change as the given parameter.
    cout<<animal.getName()<<" is "<<animal.getAge()<<" years"<<endl;

    //animal's name will change as the given parameter.
    animal.setName("BLACK MAMBA");

    //animal's age will change as the given parameter.
    animal.setAge(11);

    //animal's name and age will print to console.
    cout<<animal.getName()<<" is "<<animal.getAge()<<" years"<<endl;

    //Prints that animal is sizzling.
    animal.Sizzle();
}

void TEST_Crocodile()
{
    cout <<"+----------------+" << endl
    << "| CROCODILE TEST |" << endl
    << "+----------------+" << endl;

    //animal is created from Crocodile Class. It should take name, age and soundLevel parameters.
    Crocodile animal("Komodo Dragon",10,90);

    //animal's name and age will print to console.
    cout<<animal.getName()<<" is "<<animal.getAge()<<" years"<<endl;

    //animal's name will change as the given parameter.
    animal.setName("KOMODO DRAGON");

    //animal's age will change as the given parameter.
    animal.setAge(11);

    //animal's name and age will print to console.
    cout<<animal.getName()<<" is "<<animal.getAge()<<" years"<<endl;

    //Prints that animal is roaring.
    animal.Roar();
}

int main ()
{
    TEST_Animal();
    TEST_Mammal();
    TEST_Reptile();
    TEST_Dog();
    TEST_Cat();
    TEST_Snake();
    TEST_Crocodile();


    //Indicates that function ended succesfully.
    return 0;
}//end main ()
```

Conclusion

→ **A class can contain an object that is created from another class. This situation is called as COMPOSITION.**

→ **A class that has the basic elements of objects is called Base Class.**
And, we derive another class from Base Class (generally public).
Derived class inherits the Base Class' functions and variables which can be used in derived class when they are defined as PROTECTED or PUBLIC in Base Class.
Deriving a class from another class is called INHERITENCE.

→ **Inheritence allows us to define a class in terms of another class, which makes it easier to create and maintain application.**
This also provides an opportunity to reuse the code functionality and fast implementation time.