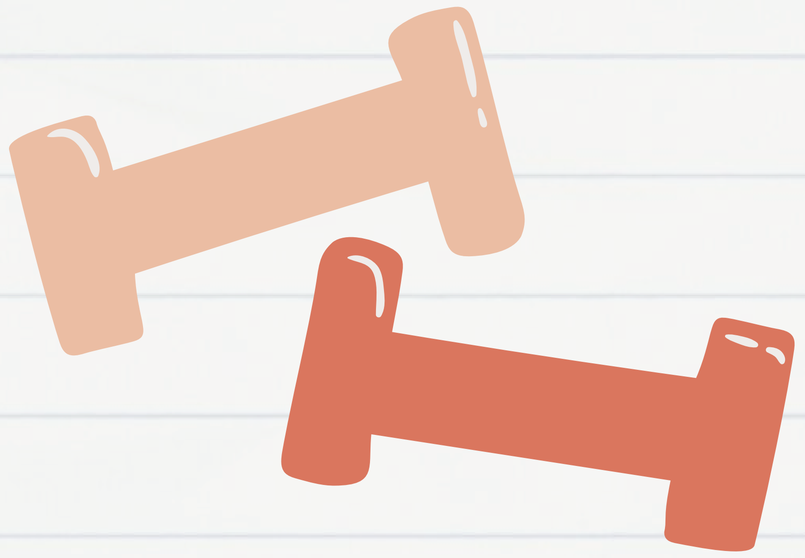
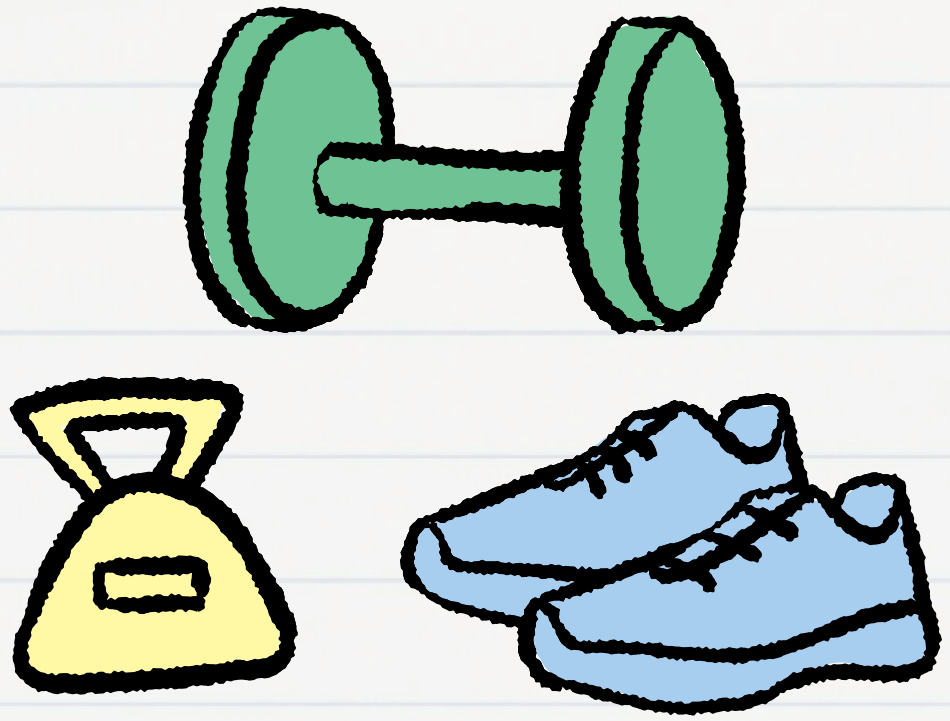
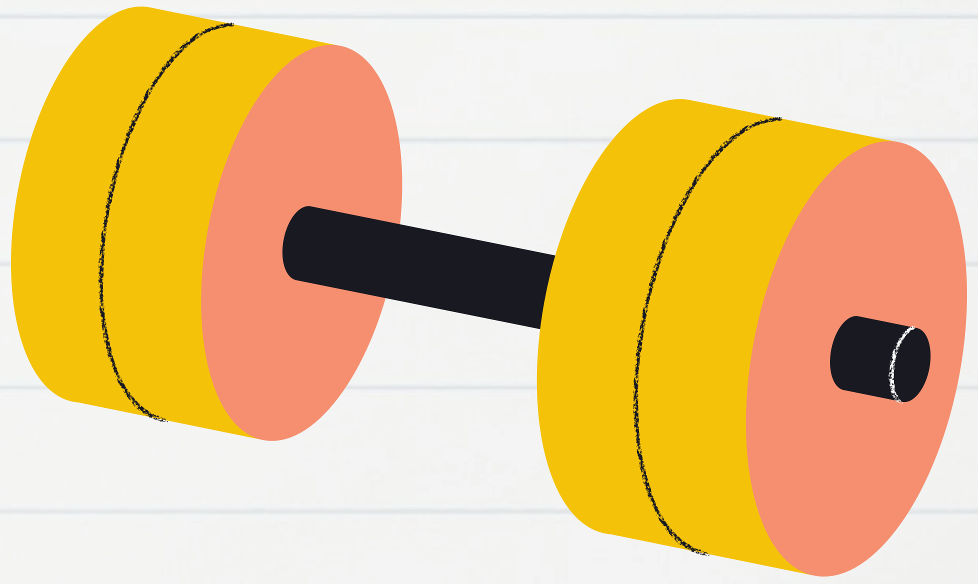


Fitness Progress Tracker System

Presented by: Group 6

Introduction

Fitness Progress Tracker System

Fitness Progress Tracker is designed to support individuals in achieving their health and fitness goals, such as weight loss or muscle gain. Key features include:

- **Goal Setting and Progress Tracking:** Users can set specific fitness goals and track their progress .
- **Activity Recording**: The system allows users to record various fitness activities, including workouts and daily meals.
- **Physical Changes Monitoring:** Efficient tracking of physical changes, particularly weight loss, to ensure users stay on track with their goals.
- **Personalized Feedback**: Offers personalized feedback to guide users toward realizing their fitness objectives.
- **User Experience and Data Security**: Prioritizes a user-friendly interface and incorporates strong security measures to protect user data.

Objective/Purpose

Tracking Fitness Activities

Provide a user-friendly platform for logging various fitness activities in detail

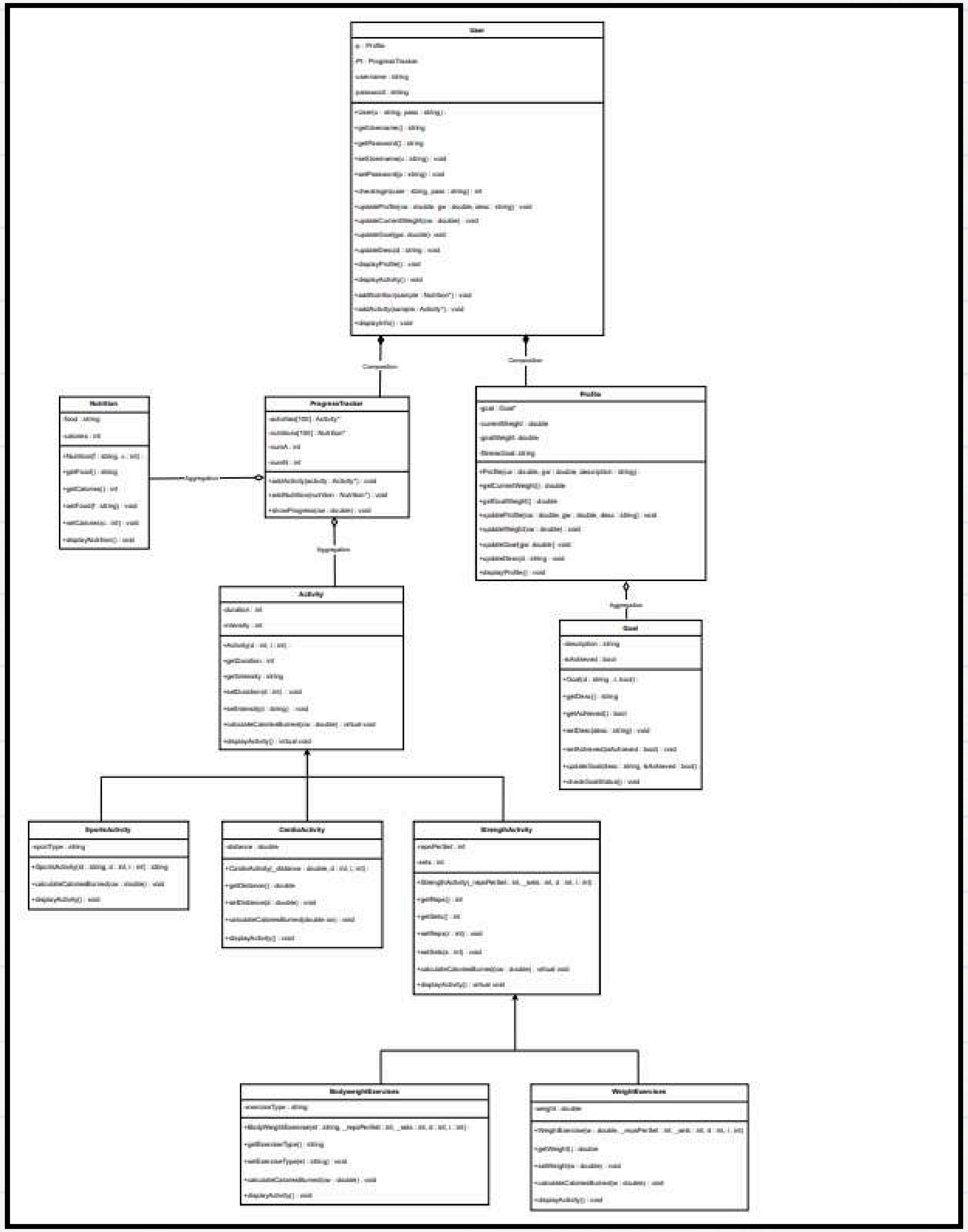
Monitoring Progress

Enable users to set and track personal fitness goals for weight loss and muscle gain

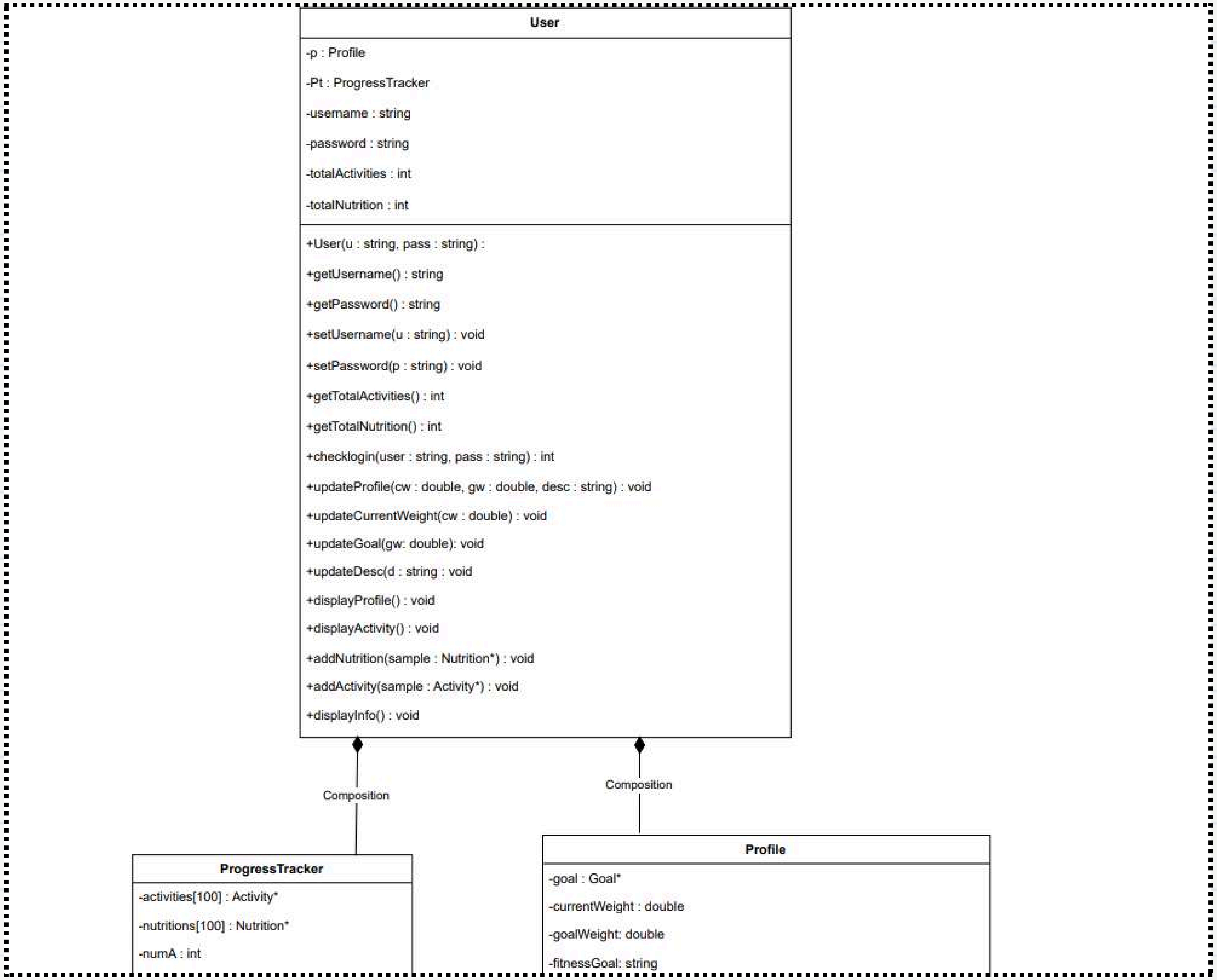
Providing Feedback

Generate basic feedback to keep users engaged and motivated

UML Diagram


User
-p : Profile
-Pt : ProgressTracker
-username : string
-password : string
+User(u : string, pass : string)
+getUsername() : string
+getPassword() : string
+setUsername(u : string) : void
+setPassword(p : string) : void
+checkIsgn(user : string, pass : string) : int
+updateProfile(cw : double, gw : double, desc : string) : void
+updateCurrentWeight(cw : double) : void
+updateGoal(gw : double) : void
+updateDesc(d : string) : void
+displayProfile() : void
+displayActivity() : void
+addNutrition(sample : Nutrition*) : void
+addActivity(sample : Activity*) : void
+displayInfo() : void
Composition
Composition
Nutrition
-food : string
-calories : int
+Nutrition(f : string, c : int)
+getFood() : string
+getCalories() : int
+setFood(f : string) : void
+setCalories(c : int) : void
+displayNutrition() : void
Aggregation
ProgressTracker
-activities[100] : Activity*
-nutritions[100] : Nutrition*
-numA : int
-numN : int
+addActivity(activity : Activity*) : void
+addNutrition(nutrition : Nutrition*) : void
+showProgress(cw : double) : void
Aggregation
Profile
-goal : Goal*
-currentWeight : double
-goalWeight : double
-fitnessGoal : string
+Profile(cw : double, gw : double, description : string)
+getCurrentWeight() : double
+getGoalWeight() : double
+updateProfile(cw : double, gw : double, desc : string) : void
+updateWeight(cw : double) : void
+updateGoal(gw : double) : void
+updateDesc(d : string) : void
+displayProfile() : void
Aggregation
Goal
-description : string
-isAchieved : bool
+Goal(d : string, i : bool)
+getDesc() : string
+getAchieved() : bool
+setDesc(desc : string) : void
+setAchieved(isAchieved : bool) : void
+updateGoal(desc : string, isAchieved : bool)
+checkGoalStatus() : void
Activity
-duration : int
-intensity : int
+Activity(d : int, i : int)
+getDuration : int
+getIntensity : string
+setDuration(d : int) : void
+setIntensity(i : string) : void
+calculateCaloriesBurned(cw : double) : virtual void
+displayActivity() : virtual void
SportsActivity
-sportType : string
+SportsActivity(st : string, d : int, i : int) : string
+calculateCaloriesBurned(cw : double) : void
+displayActivity() : void
CardioActivity
-distance : double
+CardioActivity(_distance : double, d : int, i : int)
+getDistance() : double
+setDistance(d : double) : void
+calculateCaloriesBurned(double cw) : void
+displayActivity() : void
StrengthActivity
-repsPerSet : int
-sets : int
+StrengthActivity(_repsPerSet : int, _sets : int, d : int, i : int)
+getReps() : int
+getSets() : int
+setReps(r : int) : void
+setSets(s : int) : void
+calculateCaloriesBurned(cw : double) : virtual void
+displayActivity() : virtual void
BodyweightExercises
-exerciseType : string
+BodyWeightExercise(et : string, _repsPerSet : int, _sets : int, d : int, i : int)
+getExerciseType() : string
+setExerciseType(et : string) : void
+calculateCaloriesBurned(cw : double) : void
+displayActivity() : void
WeightExercises
-weight : double
+WeightExercise(w : double, _repsPerSet : int, _sets : int, d : int, i : int)
+getWeight() : double
+setWeight(w : double) : void
+calculateCaloriesBurned(w : double) : void
+displayActivity() : void

UML Diagram



User

-p : Profile
-Pt : ProgressTracker
-username : string
-password : string
-totalActivities : int
-totalNutrition : int

+User(u : string, pass : string) :
+getUsername() : string
+getPassword() : string
+setUsername(u : string) : void
+setPassword(p : string) : void
+getTotalActivities() : int
+getTotalNutrition() : int
+checklogin(user : string, pass : string) : int
+updateProfile(cw : double, gw : double, desc : string) : void
+updateCurrentWeight(cw : double) : void
+updateGoal(gw: double): void
+updateDesc(d : string : void
+displayProfile() : void
+displayActivity() : void
+addNutrition(sample : Nutrition*) : void
+addActivity(sample : Activity*) : void
+displayInfo() : void

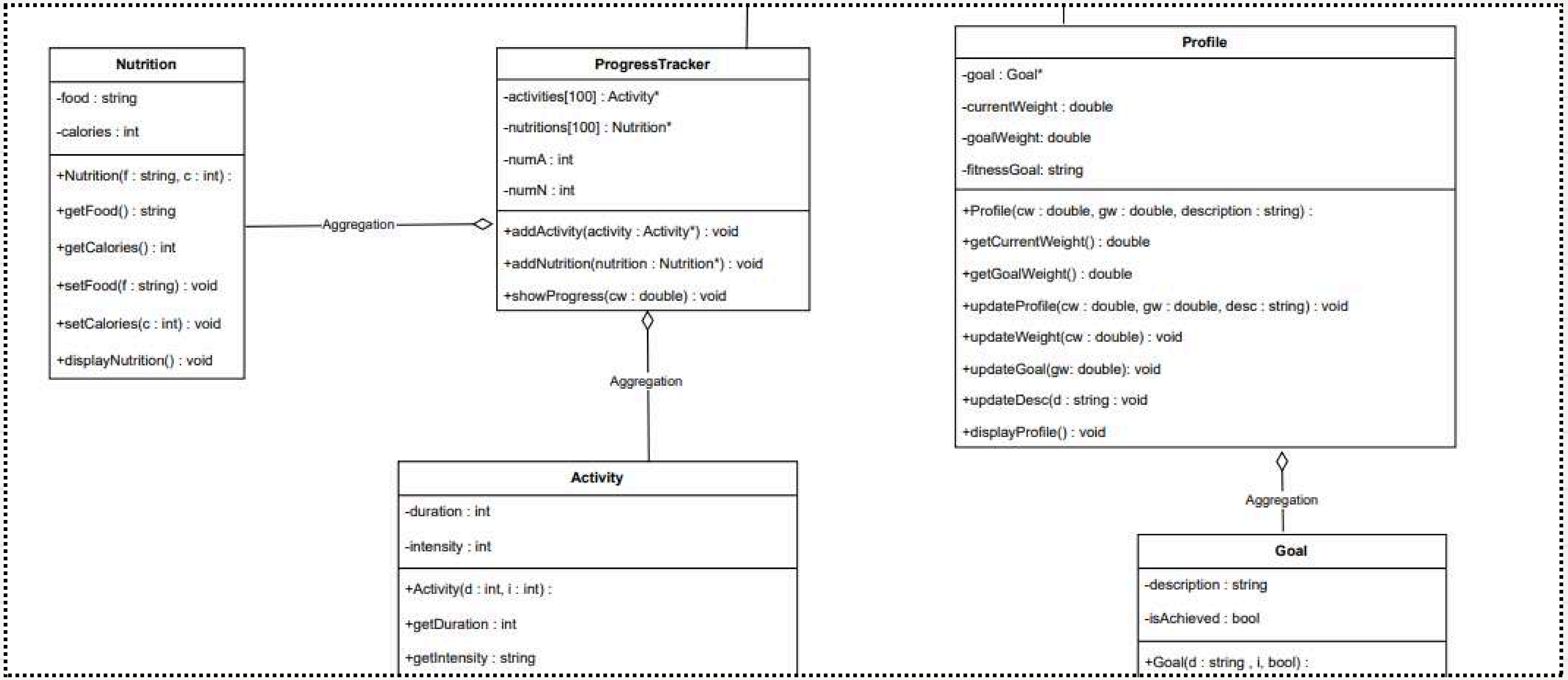
ProgressTracker

-activities[100] : Activity*
-nutritions[100] : Nutrition*
-numA : int

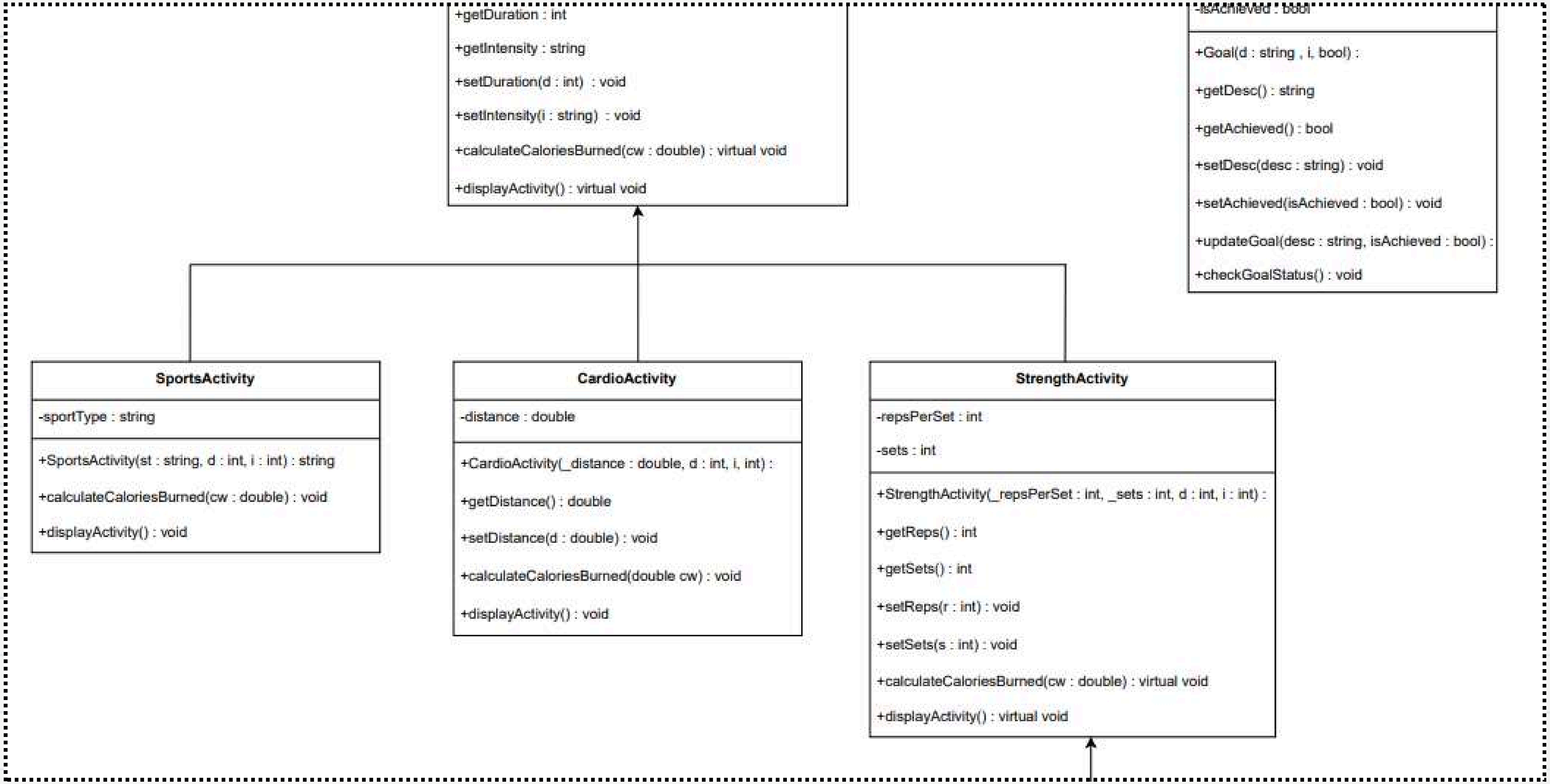
Profile

-goal : Goal*
-currentWeight : double
-goalWeight: double
-fitnessGoal: string

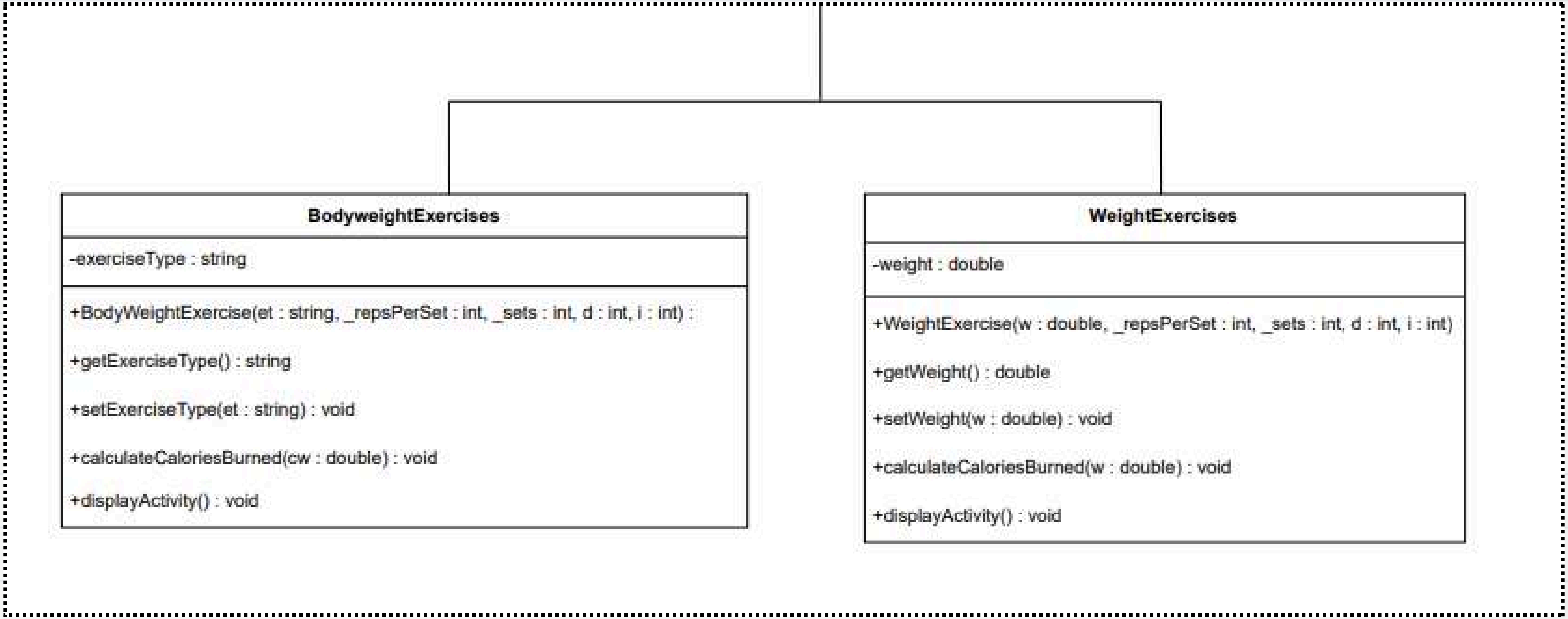
UML Diagram


Nutrition
-food : string
-calories : int
+Nutrition(f : string, c : int) :
+getFood() : string
+getCalories() : int
+setFood(f : string) : void
+setCalories(c : int) : void
+displayNutrition() : void
Aggregation
ProgressTracker
-activities[100] : Activity*
-nutritions[100] : Nutrition*
-numA : int
-numN : int
+addActivity(activity : Activity*) : void
+addNutrition(nutrition : Nutrition*) : void
+showProgress(cw : double) : void
Aggregation
Activity
-duration : int
-intensity : int
+Activity(d : int, i : int) :
+getDuration : int
+getIntensity : string
Profile
-goal : Goal*
-currentWeight : double
-goalWeight: double
-fitnessGoal: string
+Profile(cw : double, gw : double, description : string) :
+getCurrentWeight() : double
+getGoalWeight() : double
+updateProfile(cw : double, gw : double, desc : string) : void
+updateWeight(cw : double) : void
+updateGoal(gw: double): void
+updateDesc(d : string : void
+displayProfile() : void
Aggregation
Goal
-description : string
-isAchieved : bool
+Goal(d : string , i, bool) :

UML Diagram

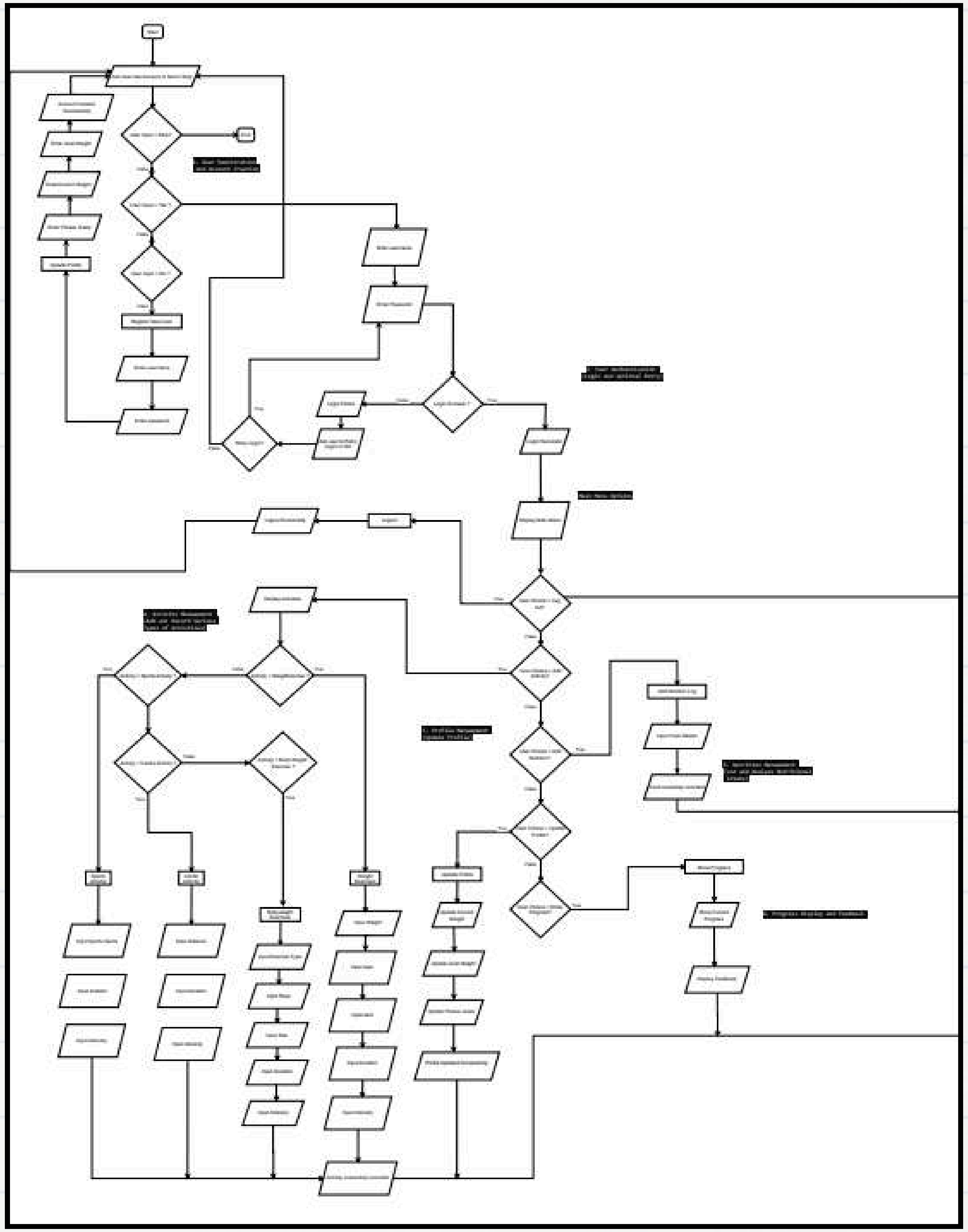

+getDuration : int
+getIntensity : string
+setDuration(d : int) : void
+setIntensity(i : string) : void
+calculateCaloriesBurned(cw : double) : virtual void
+displayActivity() : virtual void
-isAchieved : bool
+Goal(d : string , i, bool) :
+getDesc() : string
+getAchieved() : bool
+setDesc(desc : string) : void
+setAchieved(isAchieved : bool) : void
+updateGoal(desc : string, isAchieved : bool) :
+checkGoalStatus() : void
SportsActivity
-sportType : string
+SportsActivity(st : string, d : int, i : int) : string
+calculateCaloriesBurned(cw : double) : void
+displayActivity() : void
CardioActivity
-distance : double
+CardioActivity(_distance : double, d : int, i, int) :
+getDistance() : double
+setDistance(d : double) : void
+calculateCaloriesBurned(double cw) : void
+displayActivity() : void
StrengthActivity
-repsPerSet : int
-sets : int
+StrengthActivity(_repsPerSet : int, _sets : int, d : int, i : int) :
+getReps() : int
+getSets() : int
+setReps(r : int) : void
+setSets(s : int) : void
+calculateCaloriesBurned(cw : double) : virtual void
+displayActivity() : virtual void

UML Diagram

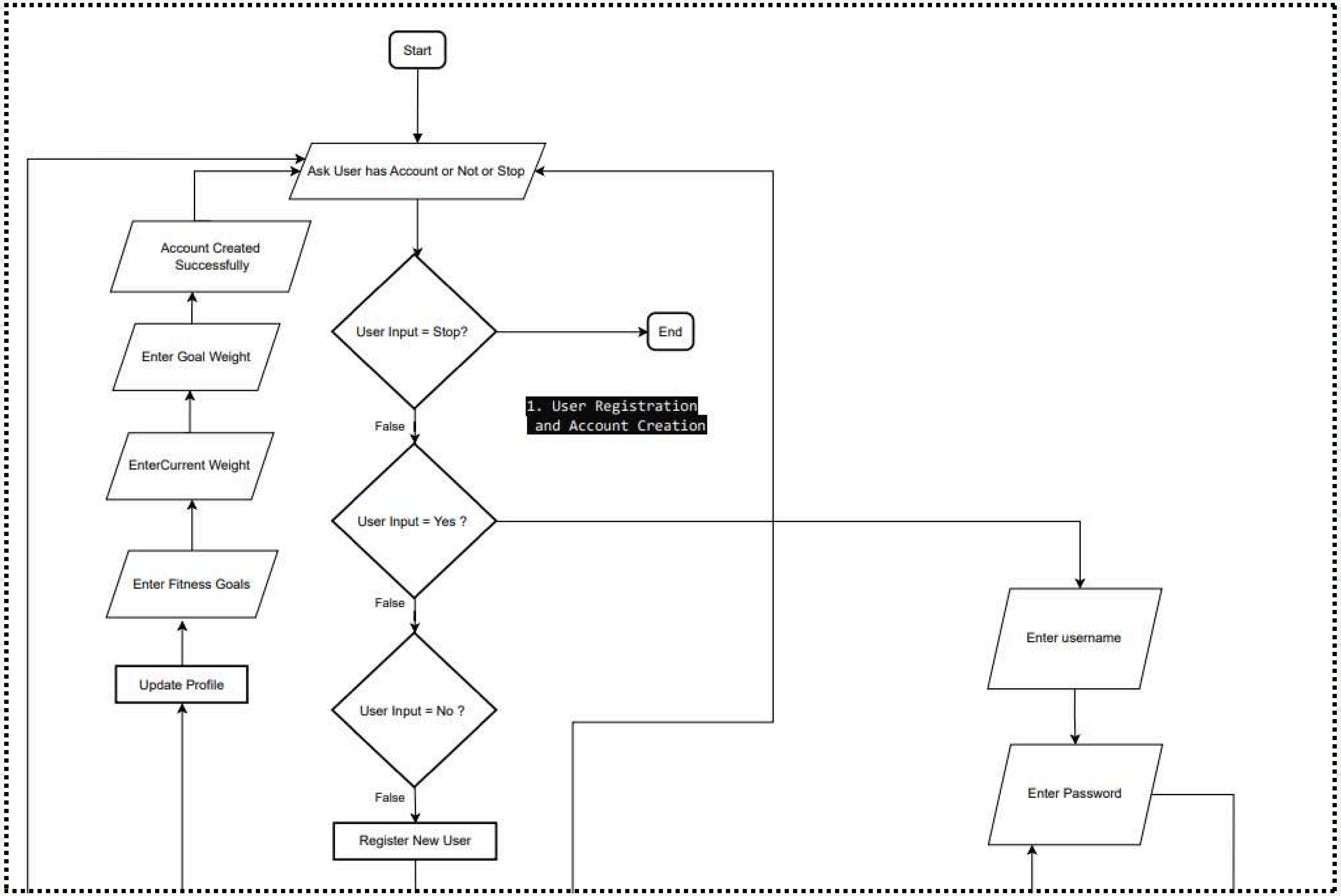


BodyweightExercises
-exerciseType : string
+BodyWeightExercise(et : string, _repsPerSet : int, _sets : int, d : int, i : int) :
+getExerciseType() : string
+setExerciseType(et : string) : void
+calculateCaloriesBurned(cw : double) : void
+displayActivity() : void
WeightExercises
-weight : double
+WeightExercise(w : double, _repsPerSet : int, _sets : int, d : int, i : int)
+getWeight() : double
+setWeight(w : double) : void
+calculateCaloriesBurned(w : double) : void
+displayActivity() : void

Flowchart

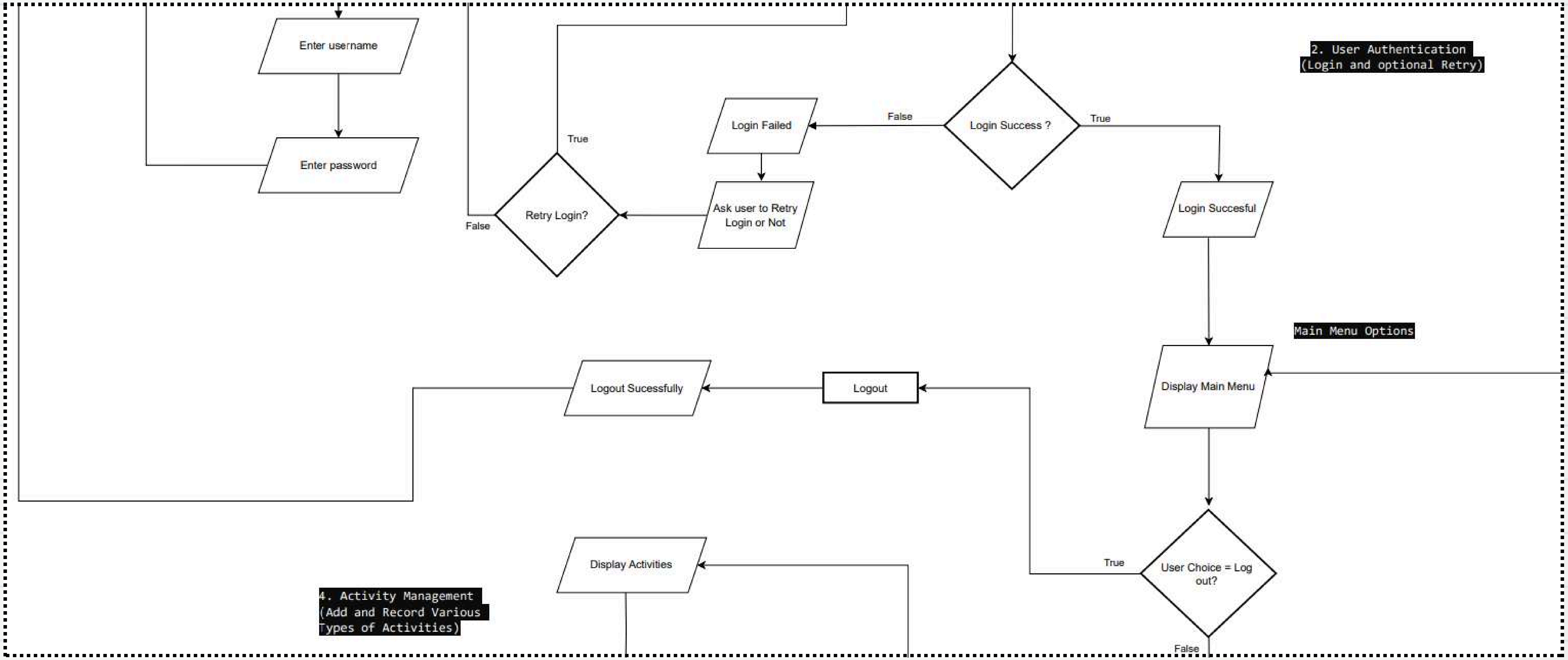


Flowchart



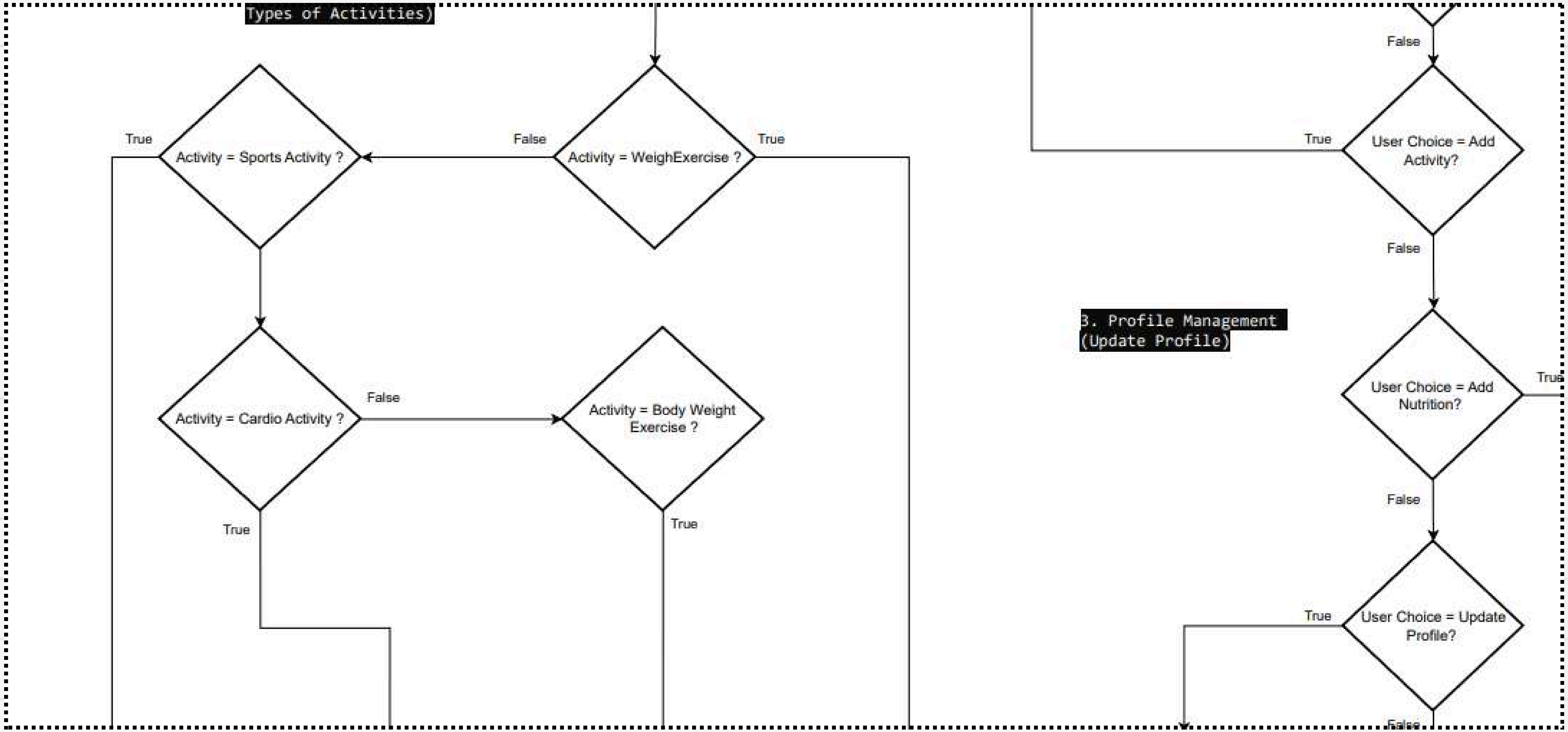
Start
Ask User has Account or Not or Stop
Account Created Successfully
Enter Goal Weight
EnterCurrent Weight
Enter Fitness Goals
Update Profile
User Input = Stop?
End
False
1. User Registration and Account Creation
User Input = Yes ?
False
User Input = No ?
False
Register New User
Enter username
Enter Password

Flowchart



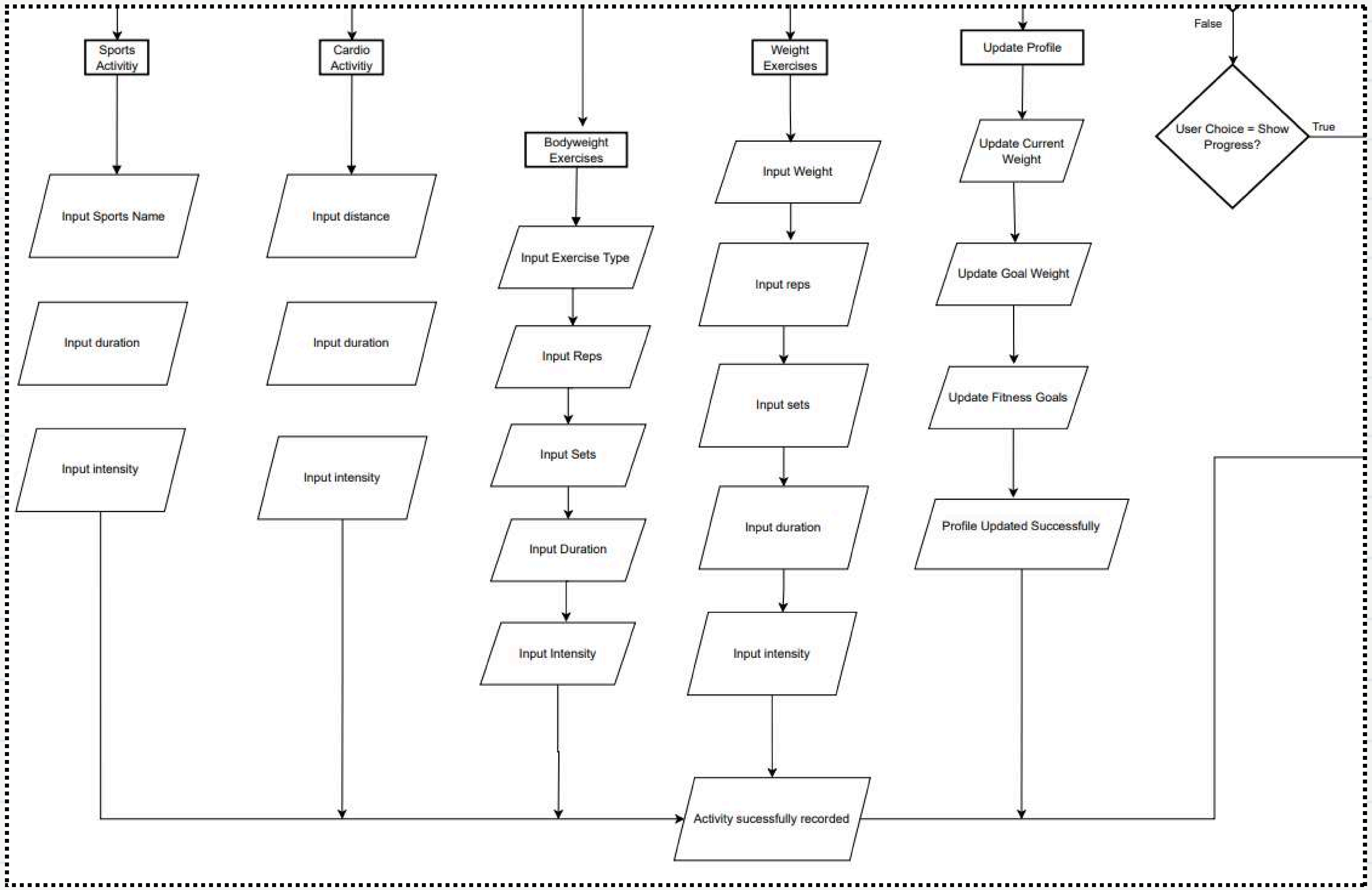
Enter username
Enter password
Retry Login?
True
False
Login Failed
Ask user to Retry Login or Not
False
Login Success ?
True
2. User Authentication (Login and optional Retry)
Login Succesful
Main Menu Options
Display Main Menu
Logout
Logout Sucessfully
User Choice = Log out?
True
False
Display Activities
4. Activity Management (Add and Record Various Types of Activities)

Flowchart

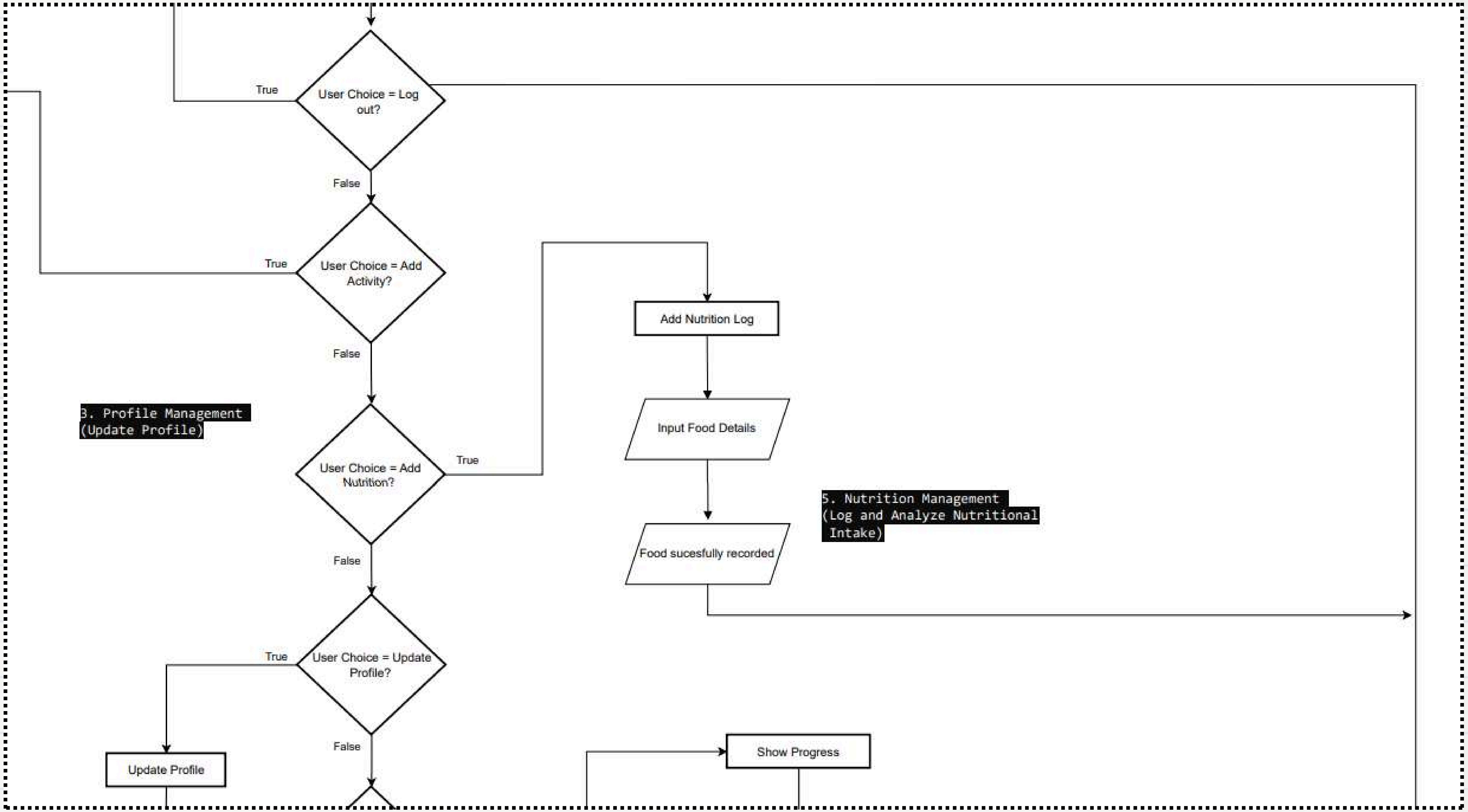


Types of Activities)
Activity = Sports Activity ?
True
False
Activity = WeighExercise ?
True
Activity = Cardio Activity ?
False
Activity = Body Weight Exercise ?
True
True
3. Profile Management (Update Profile)
False
User Choice = Add Activity?
True
False
User Choice = Add Nutrition?
True
False
User Choice = Update Profile?
True
False

Flowchart

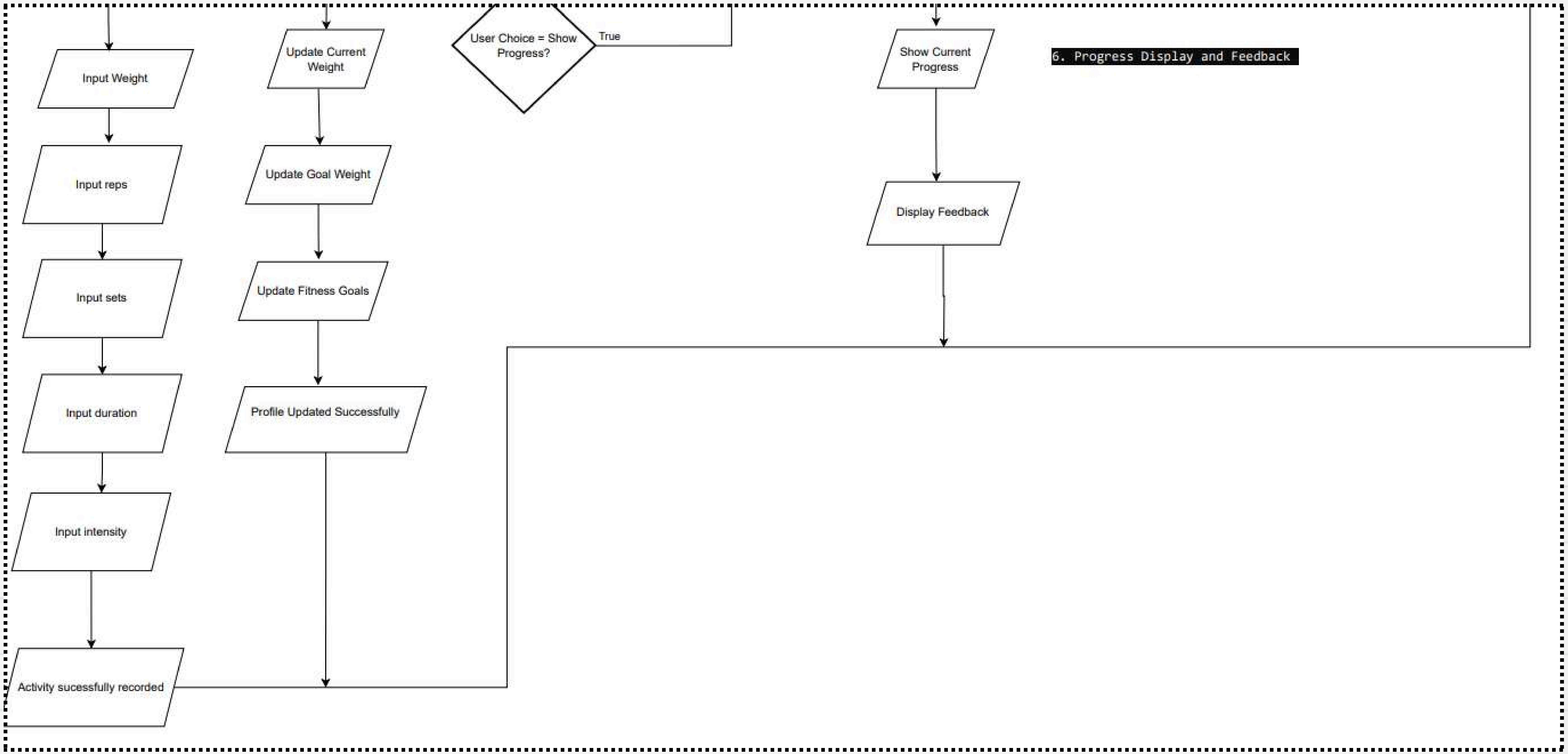

False
Sports Activitiy
Cardio Activitiy
Bodyweight Exercises
Weight Exercises
Update Profile
User Choice = Show Progress?
True
Input Sports Name
Input distance
Input Exercise Type
Input Weight
Update Current Weight
Update Goal Weight
Input duration
Input duration
Input Reps
Input reps
Update Fitness Goals
Input intensity
Input intensity
Input Sets
Input sets
Profile Updated Successfully
Input Duration
Input duration
Input Intensity
Input intensity
Activity sucessfully recorded

Flowchart



User Choice = Log out?
True
False
User Choice = Add Activity?
True
False
User Choice = Add Nutrition?
True
False
Add Nutrition Log
Input Food Details
Food sucesfully recorded
5. Nutrition Management (Log and Analyze Nutritional Intake)
3. Profile Management (Update Profile)
User Choice = Update Profile?
True
False
Update Profile
Show Progress

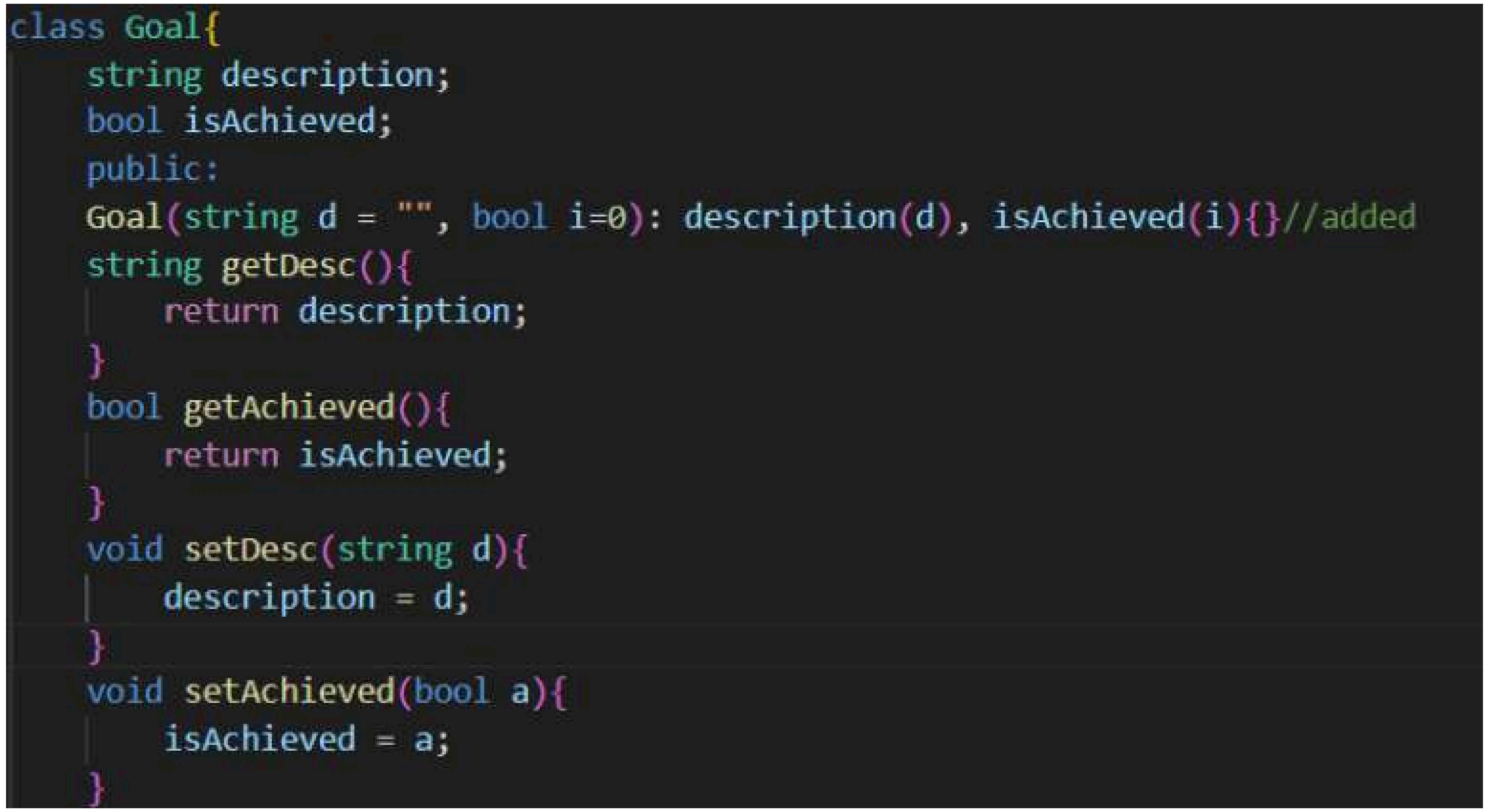
Flowchart


Input Weight
Input reps
Input sets
Input duration
Input intensity
Activity sucessfully recorded
Update Current Weight
Update Goal Weight
Update Fitness Goals
Profile Updated Successfully
User Choice = Show Progress?
True
Show Current Progress
Display Feedback
6. Progress Display and Feedback

Encapsulation

- Encapsulation is the bundling of data attributes and methods that operate on that data within a single unit or class, and restricting access to some of the object's components.
- In this project,all of the classes like Goal, Profile, Activity Nutrition and User have private data members and public Accessor and Mutator methods for outside the class to interact with their private members.

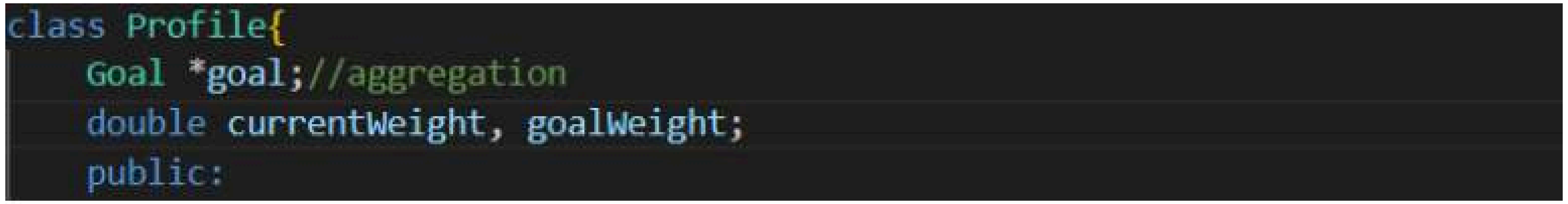
Encapsulation: Goal Class



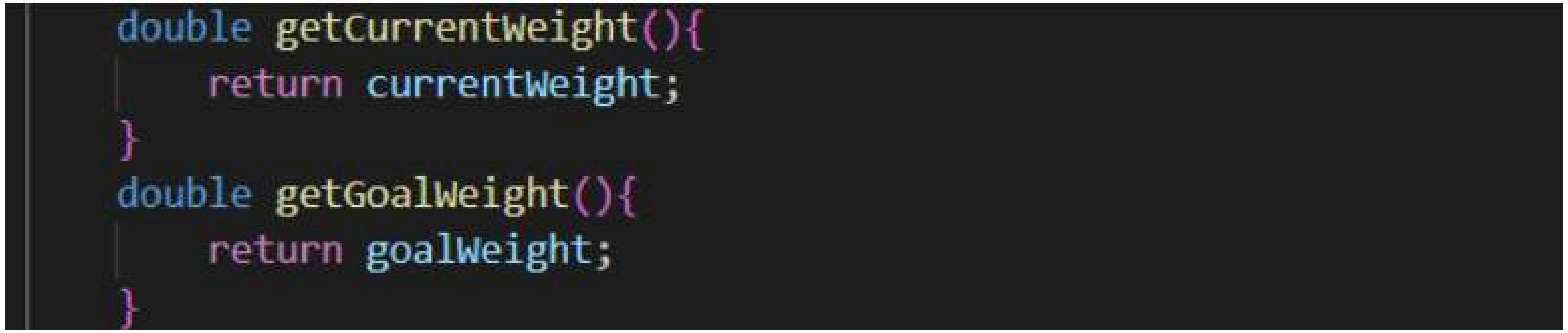
```
class Goal{
    string description;
    bool isAchieved;
    public:
    Goal(string d = "", bool i=0): description(d), isAchieved(i){}//added
    string getDesc(){
        return description;
    }
    bool getAchieved(){
        return isAchieved;
    }
    void setDesc(string d){
        description = d;
    }
    void setAchieved(bool a){
        isAchieved = a;
    }
```

- Private Data Members: description and isAchieved
- Accessor and Mutator : getDesc(), getAchieved), setDesc(), setAchieved()

Encapsulation: Profile Class



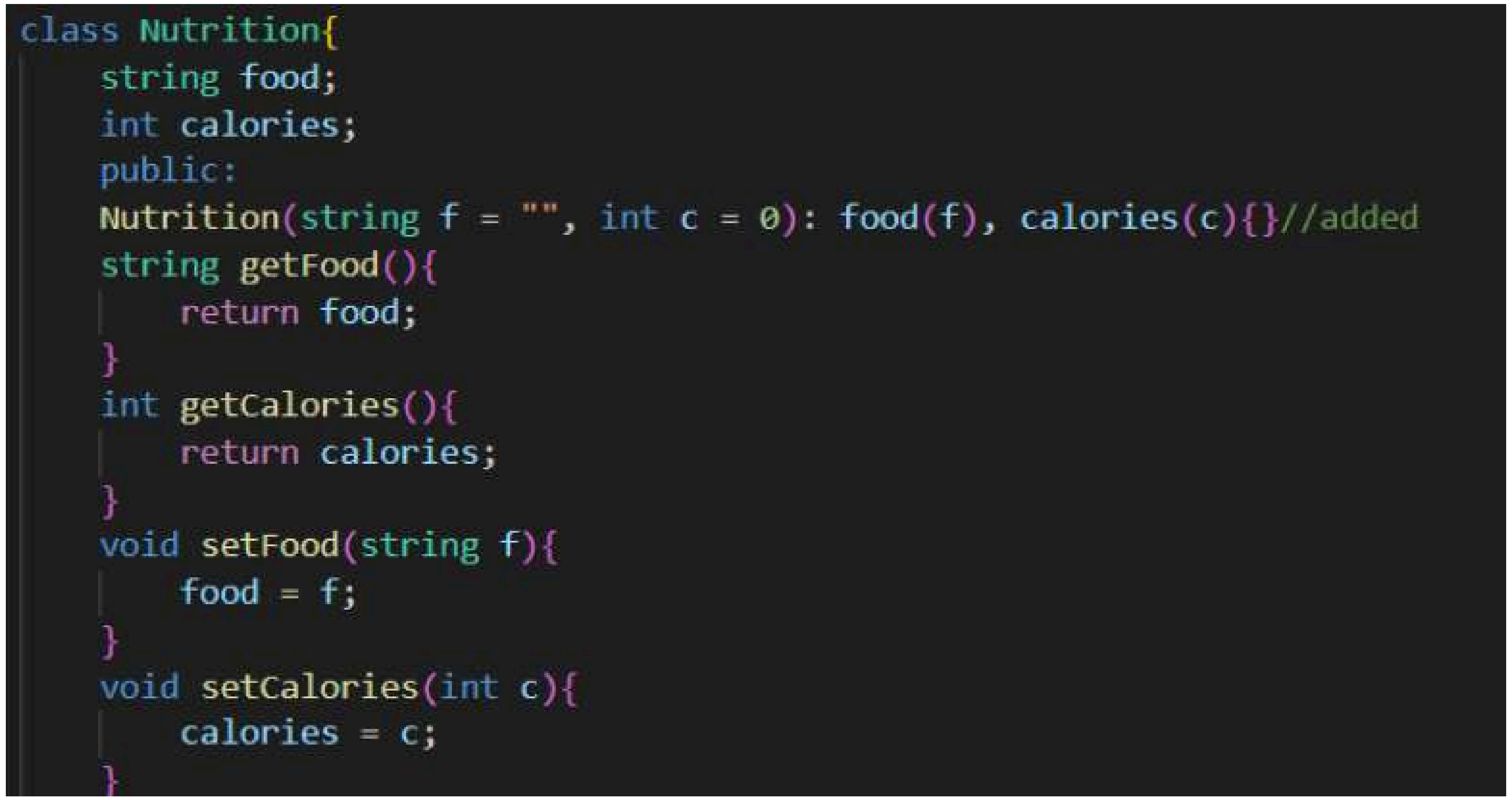
```
class Profile{
    Goal *goal;//aggregation
    double currentWeight, goalWeight;
    public:
```



```
      double getCurrentWeight(){
          return currentWeight;
      }
      double getGoalWeight(){
          return goalWeight;
      }
```

- Private Data Members: currentWeight, goalWeight
- Accessor: getCurrentWeight(), getGoalWeight()

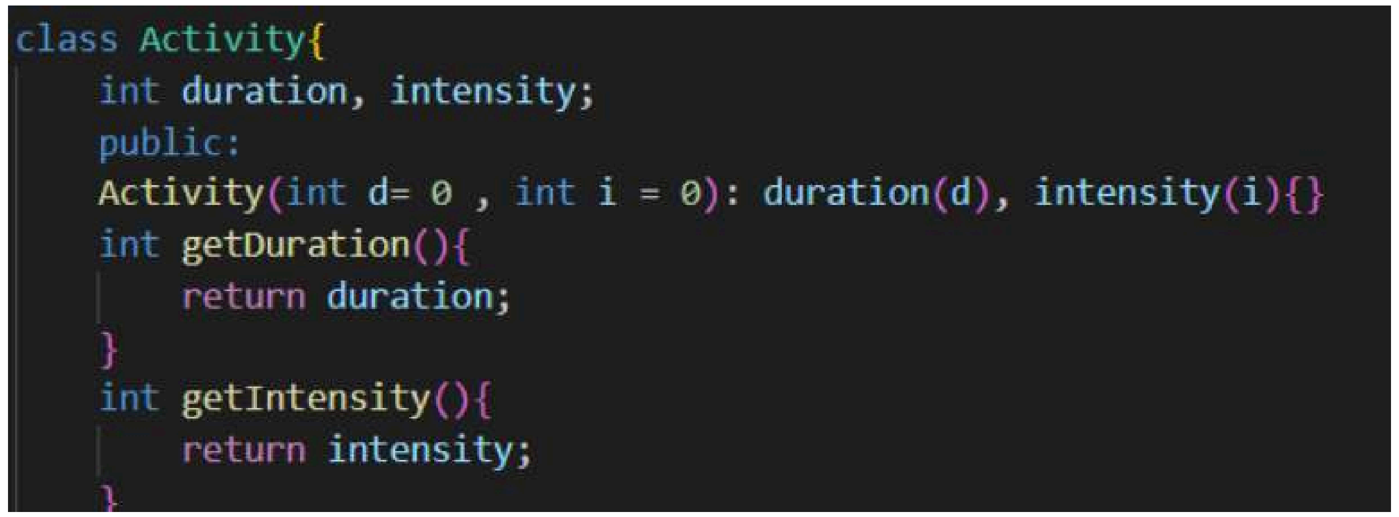
Encapsulation: Nutrition Class



```
class Nutrition{
    string food;
    int calories;
    public:
    Nutrition(string f = "", int c = 0): food(f), calories(c){}//added
    string getFood(){
        return food;
    }
    int getCalories(){
        return calories;
    }
    void setFood(string f){
        food = f;
    }
    void setCalories(int c){
        calories = c;
    }
```

- Private Data Members: food, calories
- Accessor and Mutator: getFood(), getCalories(), setFood(), setCalories()

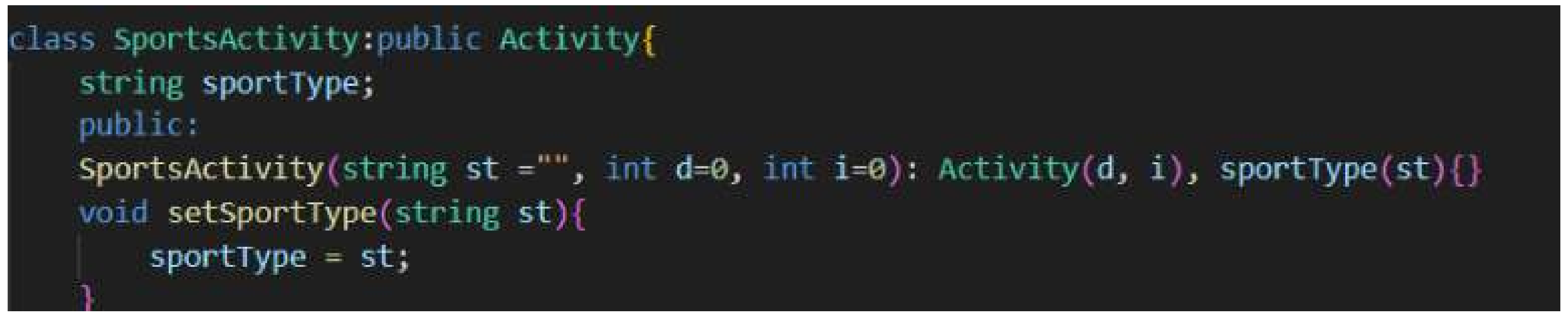
Encapsulation: Activity Class



```
class Activity{
    int duration, intensity;
    public:
    Activity(int d= 0 , int i = 0): duration(d), intensity(i){}
    int getDuration(){
        return duration;
    }
    int getIntensity(){
        return intensity;
```

- Private Data Members: duration, intensity
- Accessor: getDuration(), getIntensity()

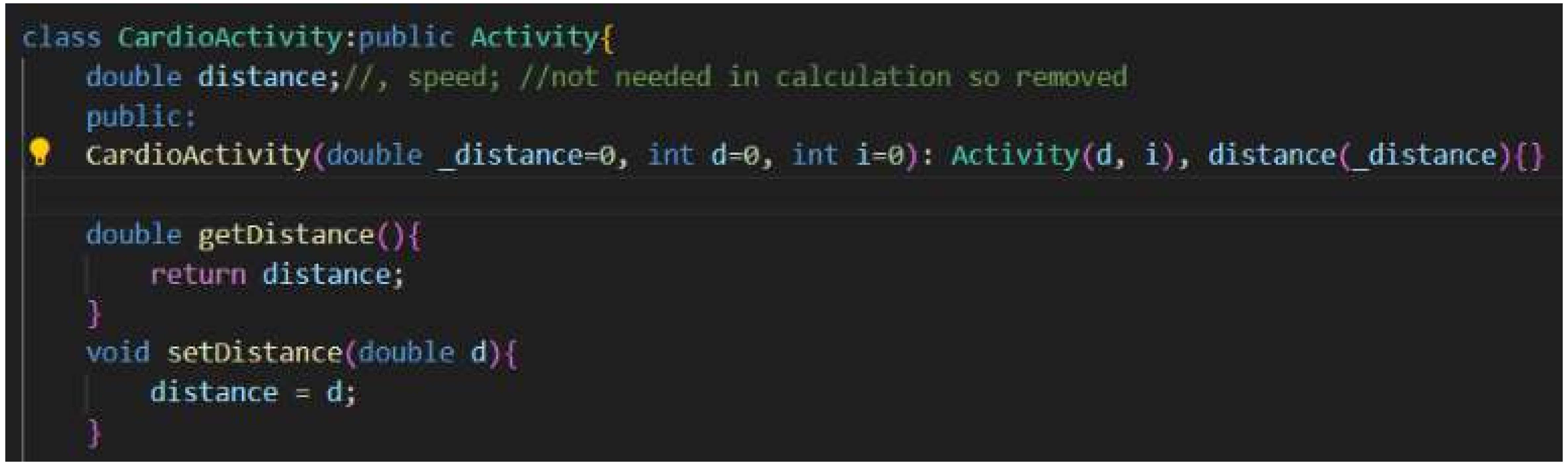
Encapsulation: SportActivity Class



```
class SportsActivity:public Activity{
    string sportType;
    public:
    SportsActivity(string st ="", int d=0, int i=0): Activity(d, i), sportType(st){}
    void setSportType(string st){
        sportType = st;
```

- Private Data Members: sportType
- Mutator: setSportType()

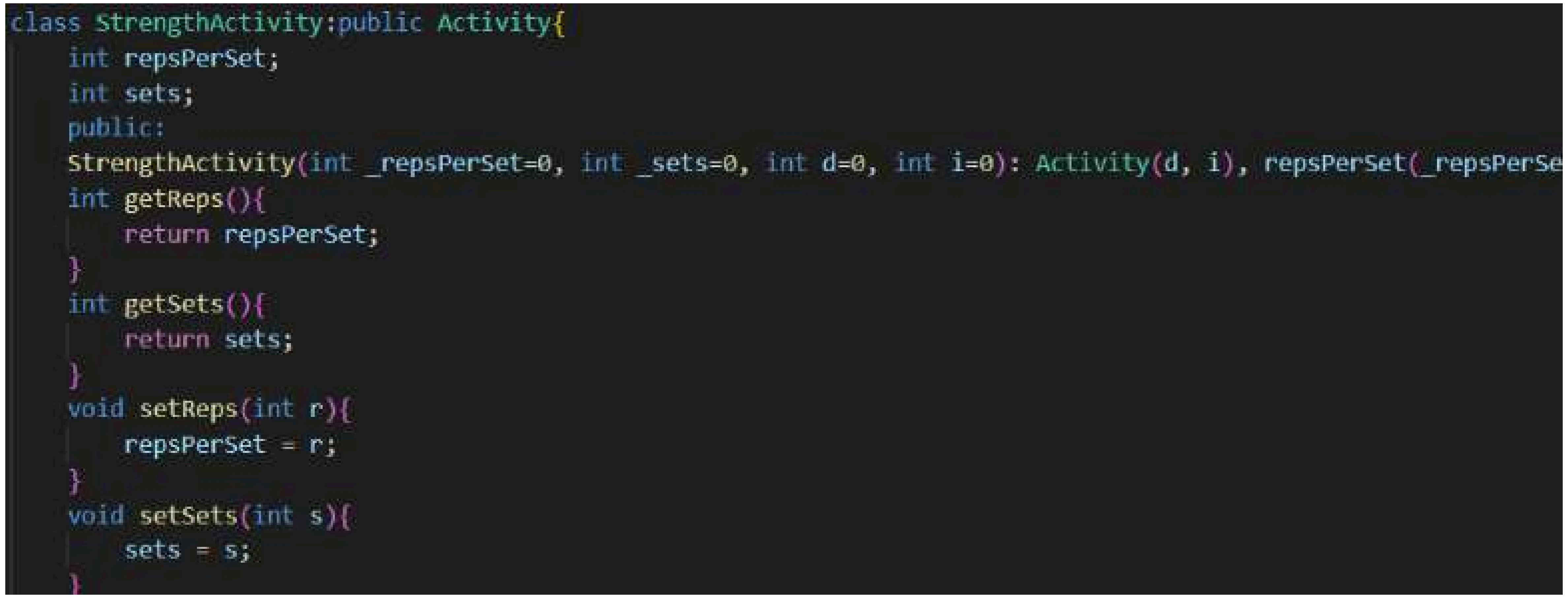
Encapsulation: CardioActivity Class



```
class CardioActivity:public Activity{
    double distance;//, speed; //not needed in calculation so removed
    public:
    CardioActivity(double _distance=0, int d=0, int i=0): Activity(d, i), distance(_distance){}

    double getDistance(){
        return distance;
    }
    void setDistance(double d){
        distance = d;
    }
```

- Private Data Members: distance
- Accessor and Mutator: getDistance(), setDistance()

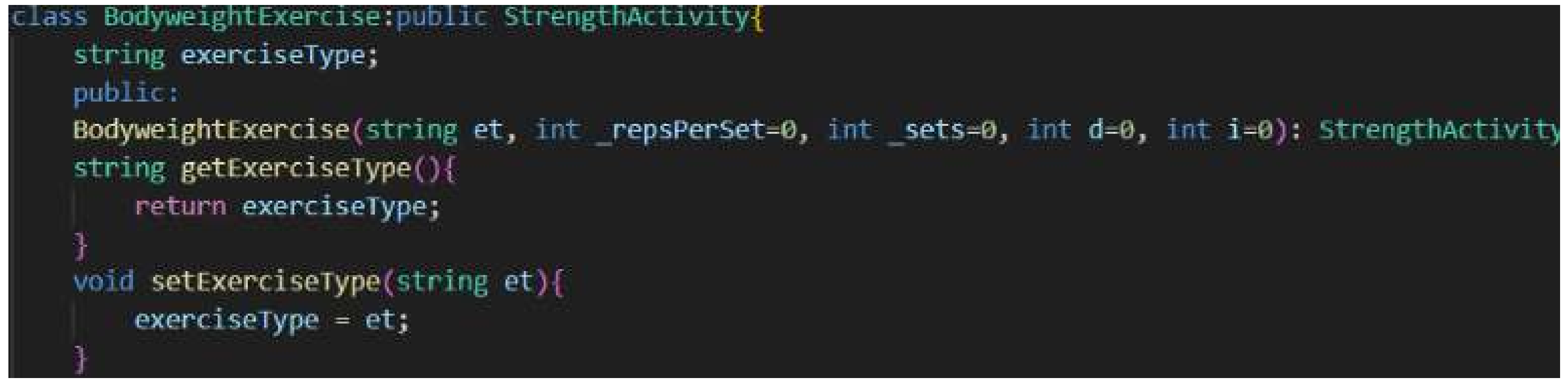
Encapsulation: StrengthActivity Class



```
class StrengthActivity:public Activity{
    int repsPerSet;
    int sets;
    public:
    StrengthActivity(int _repsPerSet=0, int _sets=0, int d=0, int i=0): Activity(d, i), repsPerSet(_repsPerSe
    int getReps(){
        return repsPerSet;
    }
    int getSets(){
        return sets;
    }
    void setReps(int r){
        repsPerSet = r;
    }
    void setSets(int s){
        sets = s;
    }
```

- Private Data Members: repsPerSet, sets
- Accessor and Mutator: getReps(), getSets(), setReps(), setSets()

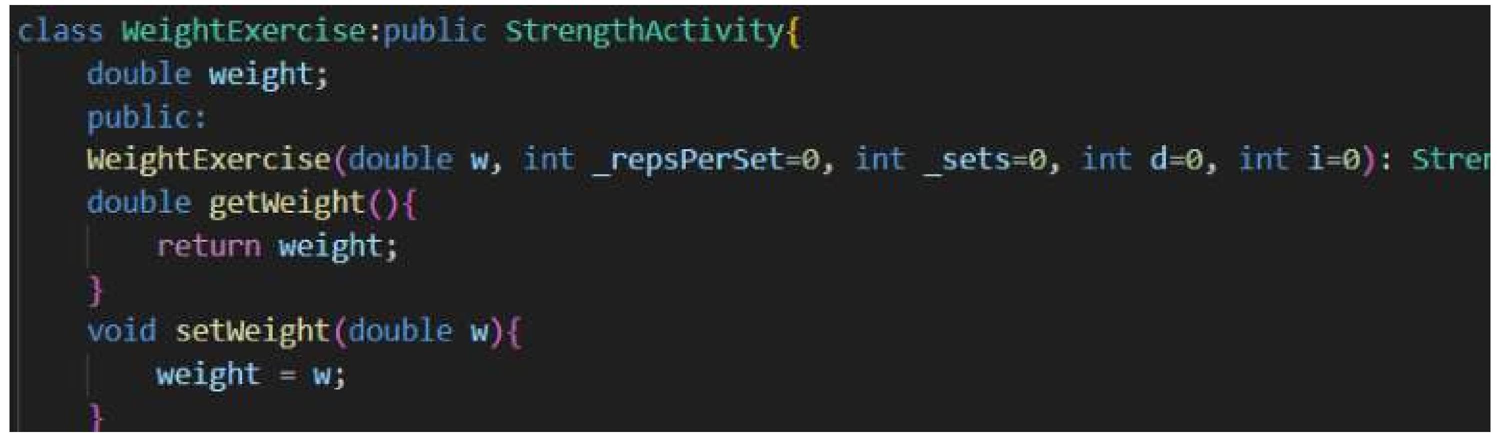
Encapsulation: BodyWeightExercise Class



```
class BodyweightExercise:public StrengthActivity{
    string exerciseType;
    public:
    BodyweightExercise(string et, int _repsPerSet=0, int _sets=0, int d=0, int i=0): StrengthActivity
    string getExerciseType(){
        return exerciseType;
    }
    void setExerciseType(string et){
        exerciseType = et;
    }
```

- Private Data Members: exerciseType
- Accessor and Mutator: getExerciseType(), setExerciseType()

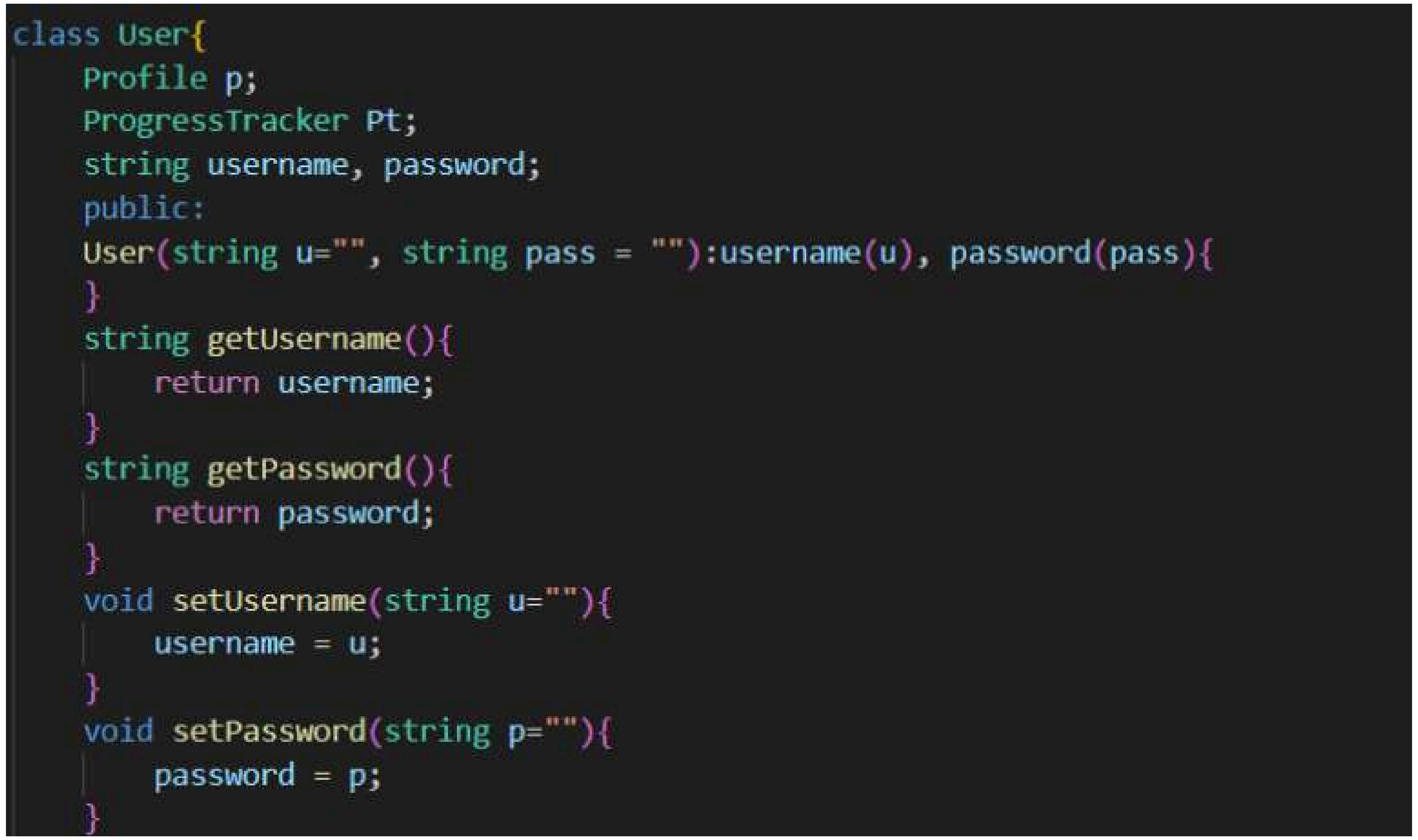
Encapsulation: WeightExercise Class



```
class WeightExercise:public StrengthActivity{
    double weight;
    public:
    WeightExercise(double w, int _repsPerSet=0, int _sets=0, int d=0, int i=0): Stren
    double getWeight(){
        return weight;
    }
    void setWeight(double w){
        weight = w;
    }
```

- Private Data Members: weight
- Accessor and Mutator: getWeight(), setWeight()

Encapsulation: UserClass



```
class User{
    Profile p;
    ProgressTracker Pt;
    string username, password;
    public:
    User(string u="", string pass = ""):username(u), password(pass){
    }
    string getUsername(){
        return username;
    }
    string getPassword(){
        return password;
    }
    void setUsername(string u=""){
        username = u;
    }
    void setPassword(string p=""){
        password = p;
    }
```

- Private Data Members: username, password
- Accessor and Mutator: getUsername(), getPassword(), setUsername(), setPassword()

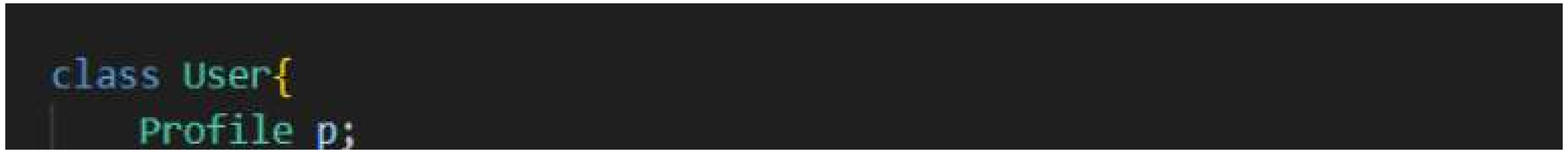
Encapsulation:

We use encapsulation to protect the internal state of objects and ensure that they can only be modified through well-defined interfaces. This helps in maintaining data integrity and hiding the complexity of the implementation from the user. By using encapsulation, it allows us to control how data is accessed and modified, ensuring that the object's state remains consistent and valid.

Composition

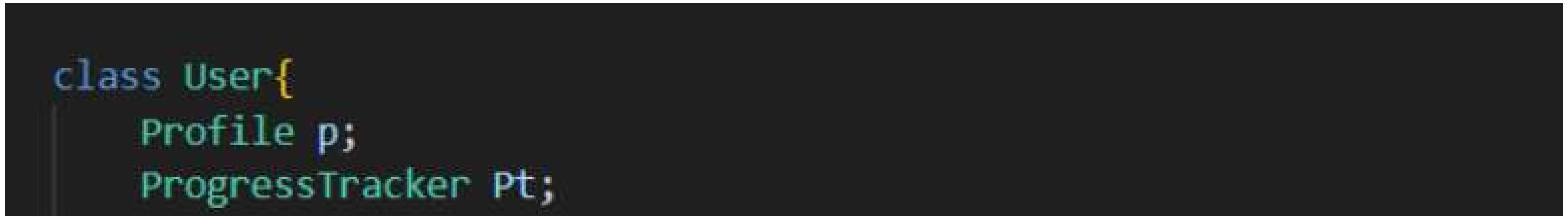
- Composition is a whole-part relationship with a strong ownership..
- .It represents that the enclosing object (whole) “consists of” enclosed objects (parts).The existence of the enclosed objects are determined by the enclosing objects.
- In this case, the User class has a composition relationship with the Profile class and ProgressTracker Class.

Compostition: User and Profile


```
class User{
    Profile p;
```

- The relationship between User and Profile is a composition, indicating that each User must have a Profile to manage personal fitness goals and related information.
- This ensures that the lifecycle of the Profile is tightly coupled with the User. For example if the User (whole) is deleted, the associated Profile (part) is also deleted.
- This composition relationship models the real-world scenario where a user's profile, which includes their goals, current weight, and target weight, is inherently a part of the user’s overall data and cannot exist independently

Compostition: User and ProgressTracker



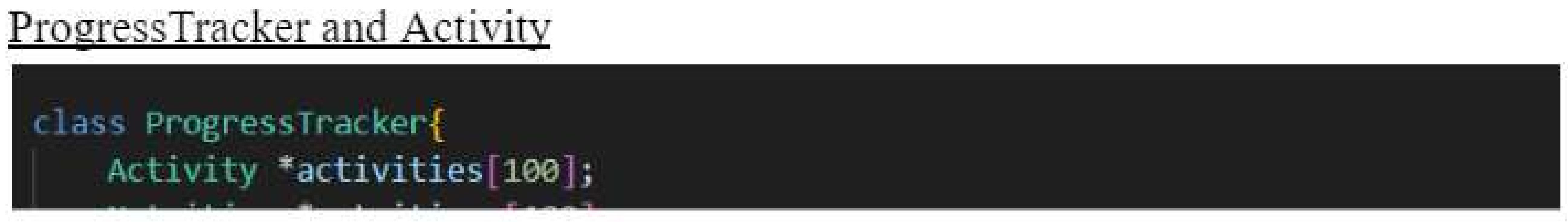
```
class User{
    Profile p;
    ProgressTracker Pt;
```

- The relationship between User and ProgressTracker is a composition because it demonstrates that each User must have a ProgressTracker to monitor and manage their fitness activities and nutrition.
- The ProgressTracker is a part of the user's fitness management system and its existence is dependent on the User.
- If the User is deleted, the associated ProgressTracker is also deleted.
- This ensures that the tracking of fitness progress is inherently linked to the user and cannot function independently.

Aggregation

- Aggregation is a special form of association which is a one way relationship. It models a 'has-a" relationship between classes where the enclosing class "has-a" enclosed class.
- The existence of both enclosing and enclosed objects are independent.
- In this project, the are three aggregation relationships which are ProgressTracker and Activity, Progress Tracker and Nutrition as well as Profile and Goal.

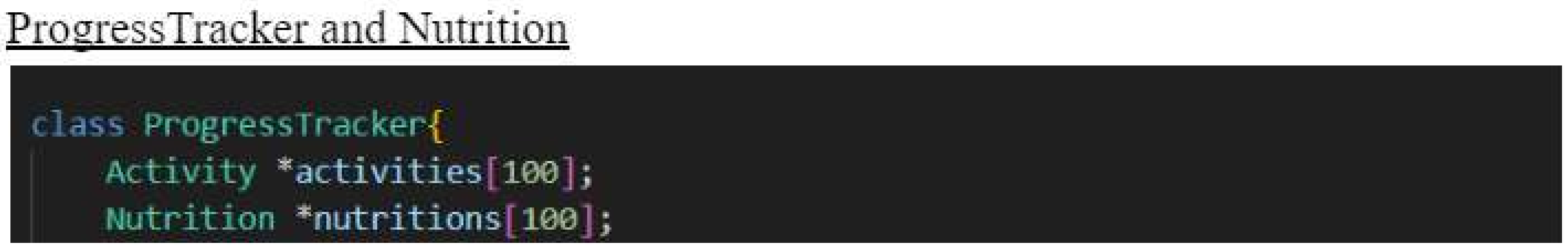
Aggregation: ProgressTracker and Activity


ProgressTracker and Activity

```
class ProgressTracker{
    Activity *activities[100];
```

- ProgressTracker aggregates Activity.
- In this context, the ProgressTracker class “has-a” and manages multiple Activity objects, but these Activity objects can exist independently of the ProgressTracker.
- The ProgressTracker class is designed to keep track of a user's fitness progress by recording various activities they perform which hold arrays of Activity objects to store and manage these activities.

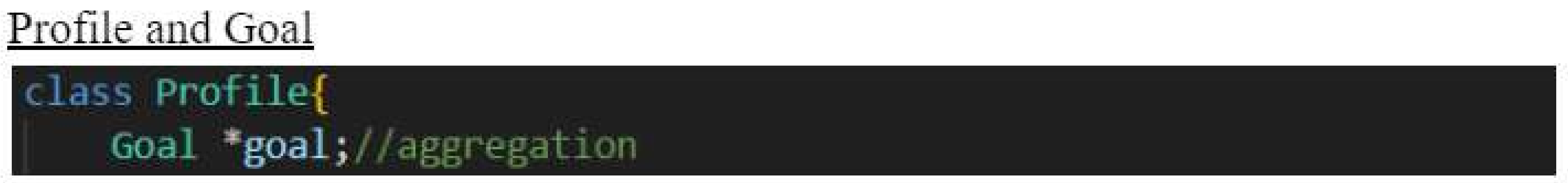
Aggregation: ProgressTracker and Nutrition


ProgressTracker and Nutrition

```
class ProgressTracker{
    Activity *activities[100];
    Nutrition *nutritions[100];
```

- ProgressTracker aggregates Nutrition.
- In this context, the ProgressTracker class “has-a” and manages multiple Nutrition objects, but these Nutrition objects can exist independently of the ProgressTracker.
- The ProgressTracker class is designed to keep track of a user's fitness progress by recording various nutritional intakes which holds arrays of Nutrition objects to store and manage these nutritional entries.

Aggregation: Profile and Goal


Profile and Goal

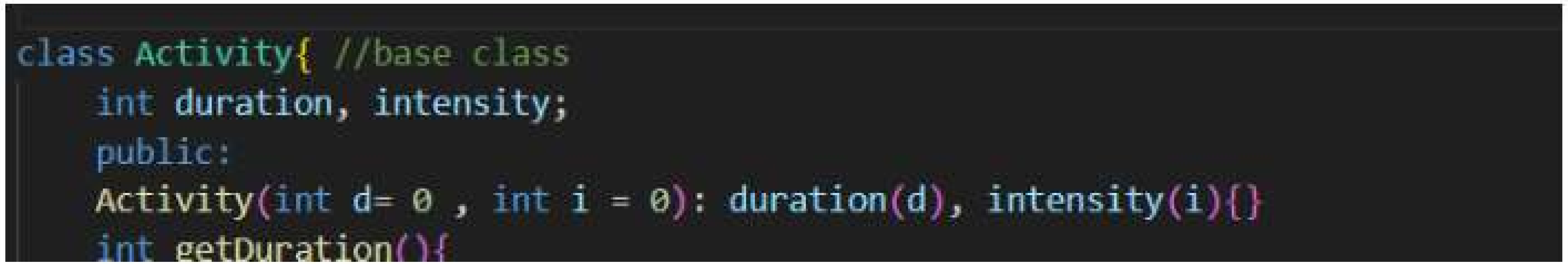
```
class Profile{
    Goal *goal;//aggregation
```

- Profile has aggregation with Goal.
- Profile manages Goal objects but does not strictly control their lifecycle.
- Profile objects (enclosing objects) do not necessarily have Goal objects (enclosed objects).
- Both of them can exist independently regardless of if one party is destroyed.
- The Profile class is designed to manage user details related to their fitness journey, including current weight, goal weight, and fitness goals which holds a pointer to a Goal object to represent the user's fitness goal.

Inheritance

- Inheritance is a concept where a new class (derived class) inherits the properties and behavior of another class (base class).
- In this project, there are derived classes from the Activity (base class) and StrengthActivity (intermediate base class).

Base Class : Activity

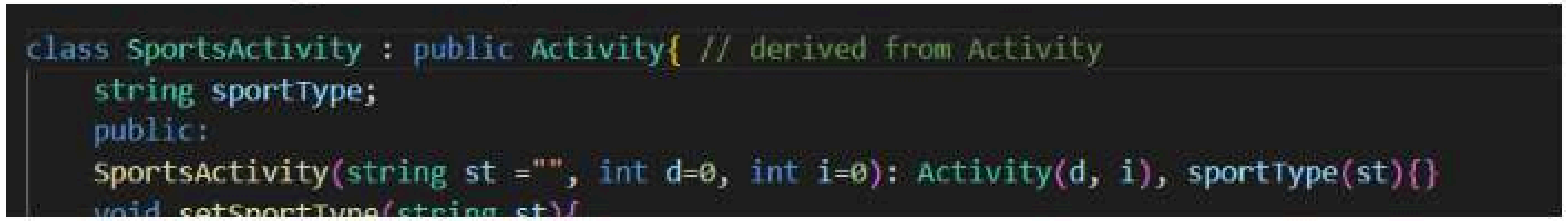


```
class Activity{ //base class
    int duration, intensity;
    public:
    Activity(int d= 0 , int i = 0): duration(d), intensity(i){}
    int getDuration(){
```

- Activity will be the base class where the derived classes such as SportsActivity, CardioActivity and StrengthActivity inherits its attributes and behavior.

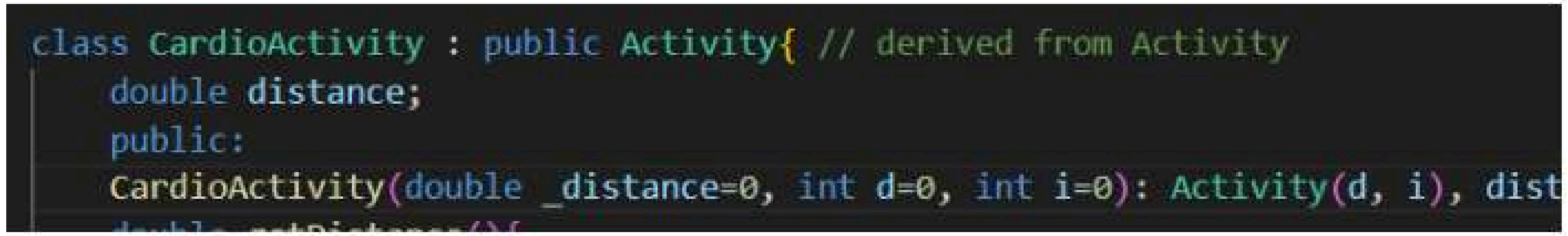
Inheritance

<u>Derived Class : SportsActivity</u>



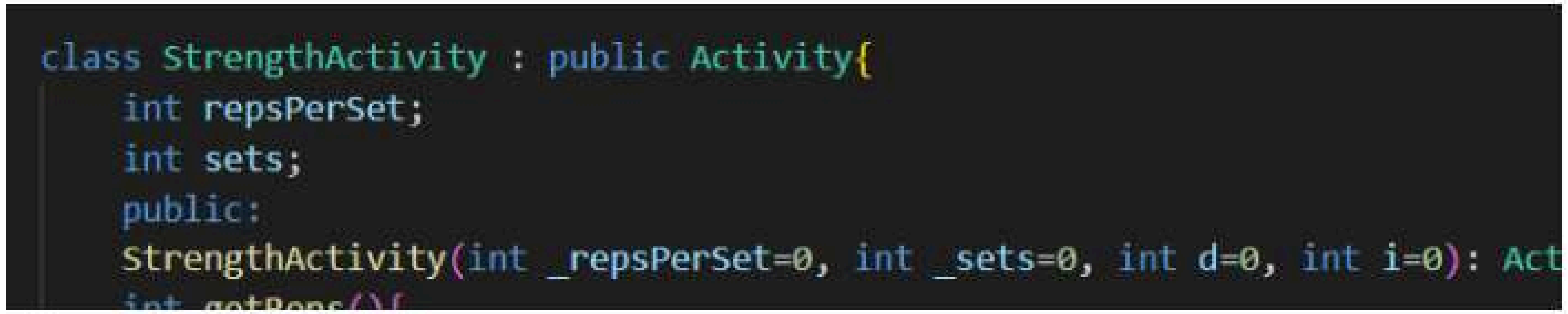
```
class SportsActivity : public Activity{ // derived from Activity
    string sportType;
    public:
    SportsActivity(string st ="", int d=0, int i=0): Activity(d, i), sportType(st){}
    void setSportType(string st){
```

<u>Derived Class : CardioActivity</u>



```
class CardioActivity : public Activity{ // derived from Activity
    double distance;
    public:
    CardioActivity(double _distance=0, int d=0, int i=0): Activity(d, i), dist
    double getDistance(){
```

<u>Derived Class : StrengthActivity</u>

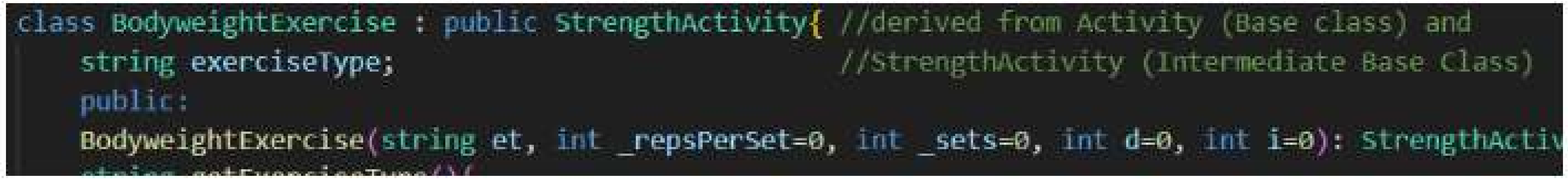


```
class StrengthActivity : public Activity{
    int repsPerSet;
    int sets;
    public:
    StrengthActivity(int _repsPerSet=0, int _sets=0, int d=0, int i=0): Act
    int getReps(){
```

- These classes are specialized forms of Activity, each with unique attributes and behaviors that extend the base class Activity.
- Inheritance is used here to promote code reuse and polymorphic behavior.
- Each subclass extends the functionality of the base class and introduces specialized behaviors without duplicating code.
- For example are specific calorie burning calculations and activity-specific attributes.

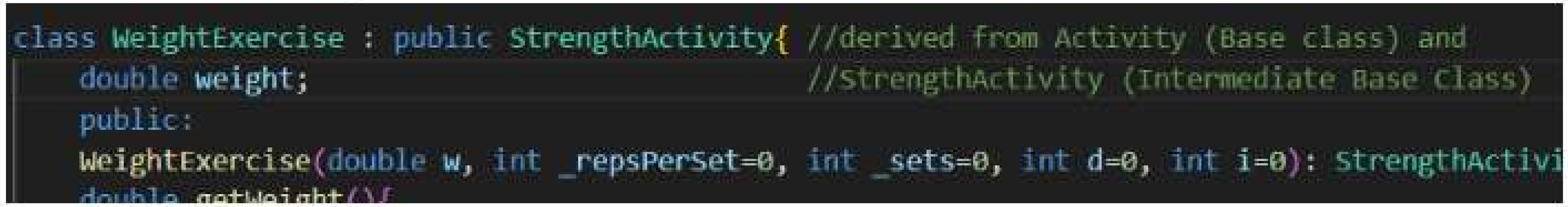
Inheritance

Derived Class : BodyWeightExercise



```
class BodyweightExercise : public StrengthActivity{ //derived from Activity (Base class) and
    string exerciseType;                             //StrengthActivity (Intermediate Base Class)
    public:
    BodyweightExercise(string et, int _repsPerSet=0, int _sets=0, int d=0, int i=0): StrengthActiv
```

Derived Class : WeightExercise



```
class WeightExercise : public StrengthActivity{ //derived from Activity (Base class) and
    double weight;                               //StrengthActivity (Intermediate Base Class)
    public:
    WeightExercise(double w, int _repsPerSet=0, int _sets=0, int d=0, int i=0): StrengthActivi
```

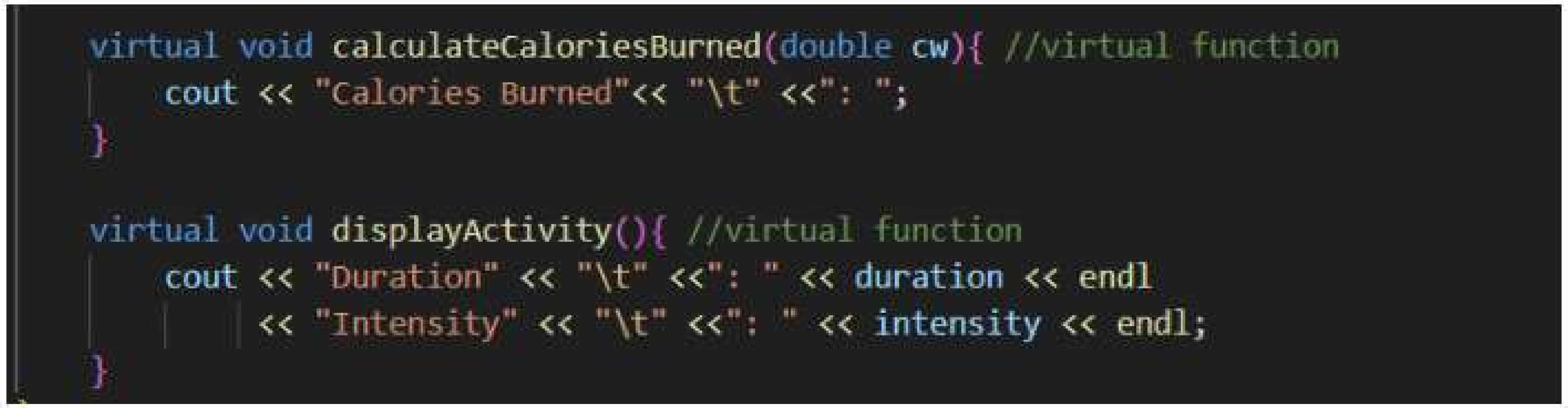
- Specialized forms of Activity and StrengthActivity, each with unique attributes and behaviors that extend the Activity (base class) and StrengthActivity (intermediate base class).
- The BodyWeightExerciseand WeightExercise are inherited from the StrengthActivity and through it, indirectly from Activity.
- By inheriting from StrengthActivity, they not only gain the general attributes of Activity (duration, intensity) but also the specialized attributes of StrengthActivity (sets, repsPerSet).

Polymorphism

- Polymorphism is one of the most important concepts in OOP that describes the ability of objects to take or to be displayed in different forms.
- It performs the same actions but different behaviors. It allows objects of different classes to be treated as objects of a common superclass.
- In this case, we implement the virtual member functions on base classes like Activity and StrengthActivity that will do the dynamic binding where the bindings are decided at runtime.

Polymorphism

Base Class : Activity

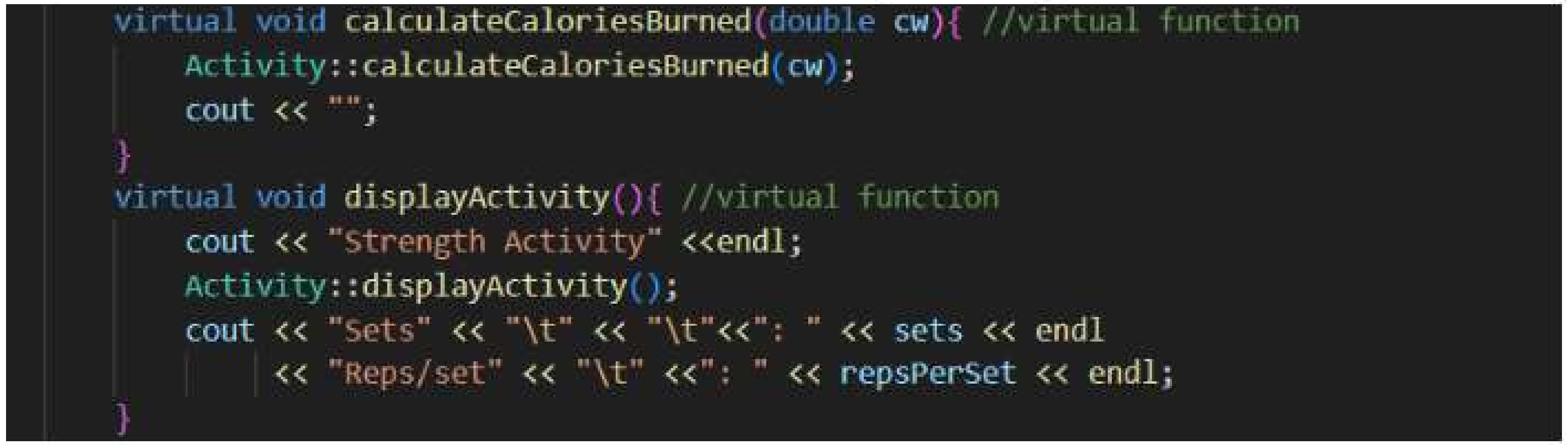


```
virtual void calculateCaloriesBurned(double cw){ //virtual function
    cout << "Calories Burned"<< "\t" <<": ";
}

virtual void displayActivity(){ //virtual function
    cout << "Duration" << "\t" <<": " << duration << endl
         << "Intensity" << "\t" <<": " << intensity << endl;
}
```

- Base class
- the calculateCaloriesBurned() and displayActivity() methods in the Activity (base class) are declared as virtual
- - This allows derived classes from the Activity (base class) such as StrengthActivity, SportsActivity and CardioActivity to provide their own implementation.

- Intermediate base class
- Other than overrides methods from the Activity (base class), the StrengthActivity class also will be an intermediate base class.
- - The calculateCaloriesBurned() and displayActivity() methods in the StrengthActivity (intermediate base class) will be declared as virtual too.
- Allows derived classes from the Activity (base class) and StrengthActivity (intermediate base class) which are WeightExercise and BodyWeightExercise classes to provide their own implementation.

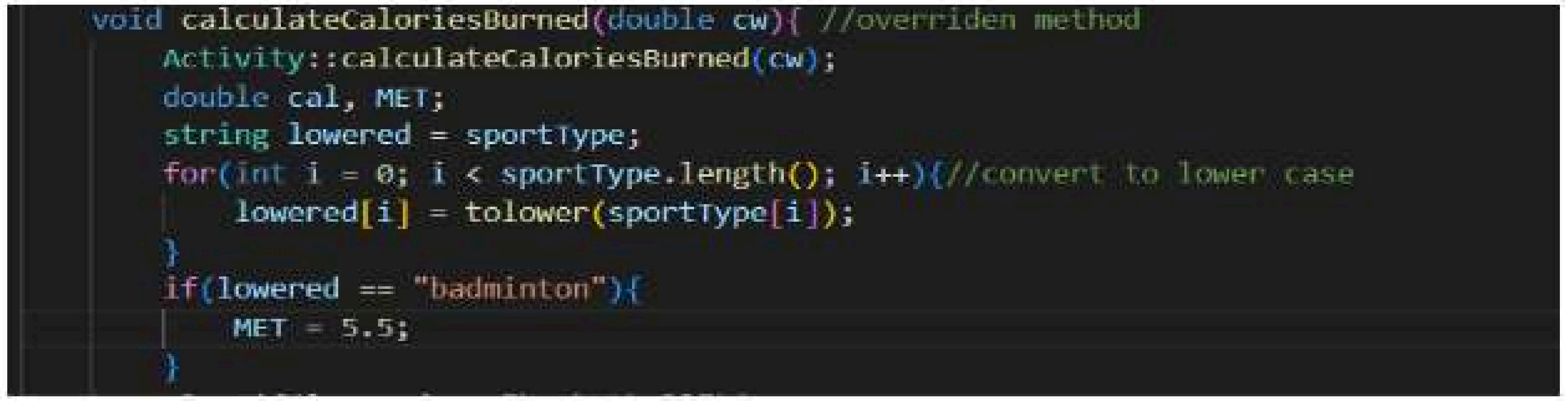
Derived Class and Intermediate Base Class : StrengthActivity



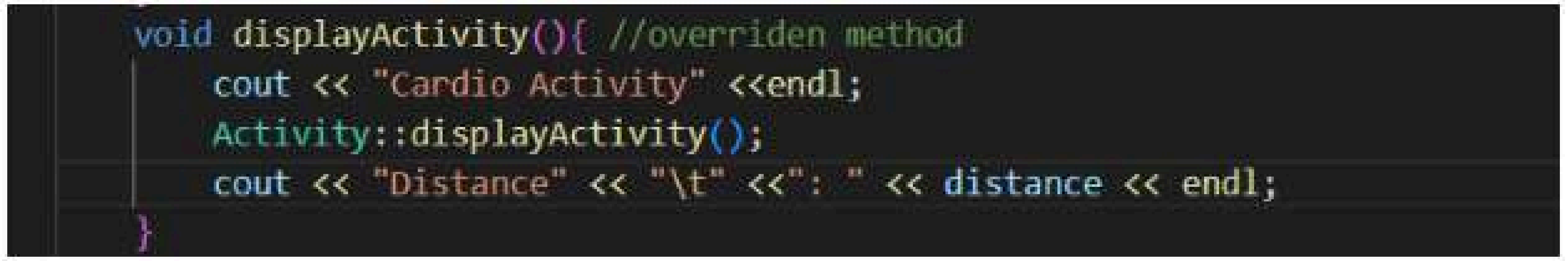
```
virtual void calculateCaloriesBurned(double cw){ //virtual function
    Activity::calculateCaloriesBurned(cw);
    cout << "";
}
virtual void displayActivity(){ //virtual function
    cout << "Strength Activity" <<endl;
    Activity::displayActivity();
    cout << "Sets" << "\t" << "\t"<<": " << sets << endl
         << "Reps/set" << "\t" <<": " << repsPerSet << endl;
}
```

Polymorphism

Derived Class : SportsActivity

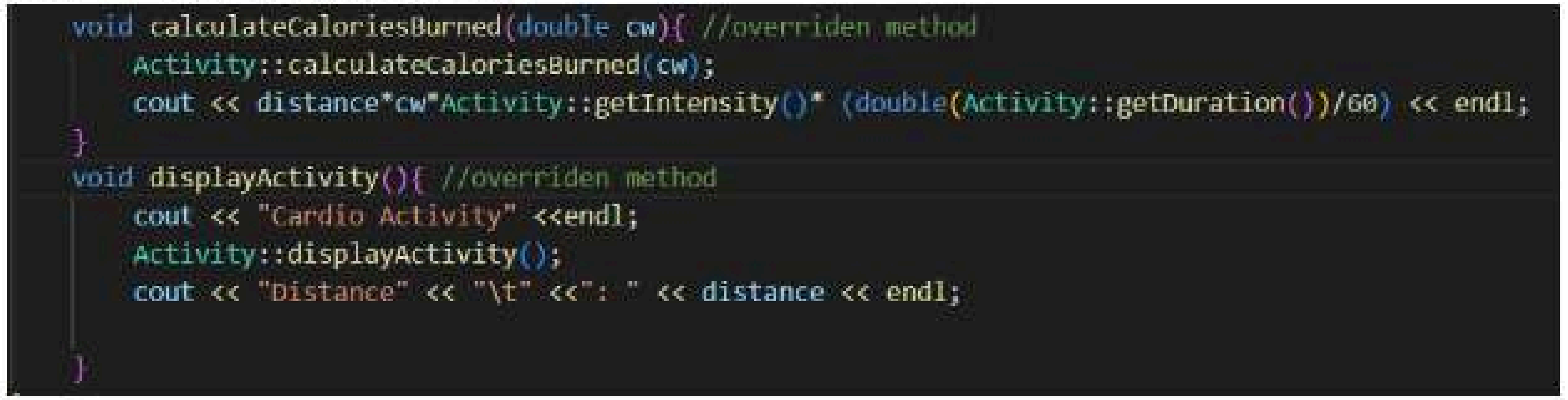


```
void calculateCaloriesBurned(double cw){ //overriden method
    Activity::calculateCaloriesBurned(cw);
    double cal, MET;
    string lowered = sportType;
    for(int i = 0; i < sportType.length(); i++){//convert to lower case
        lowered[i] = tolower(sportType[i]);
    }
    if(lowered == "badminton"){
        MET = 5.5;
    }
```



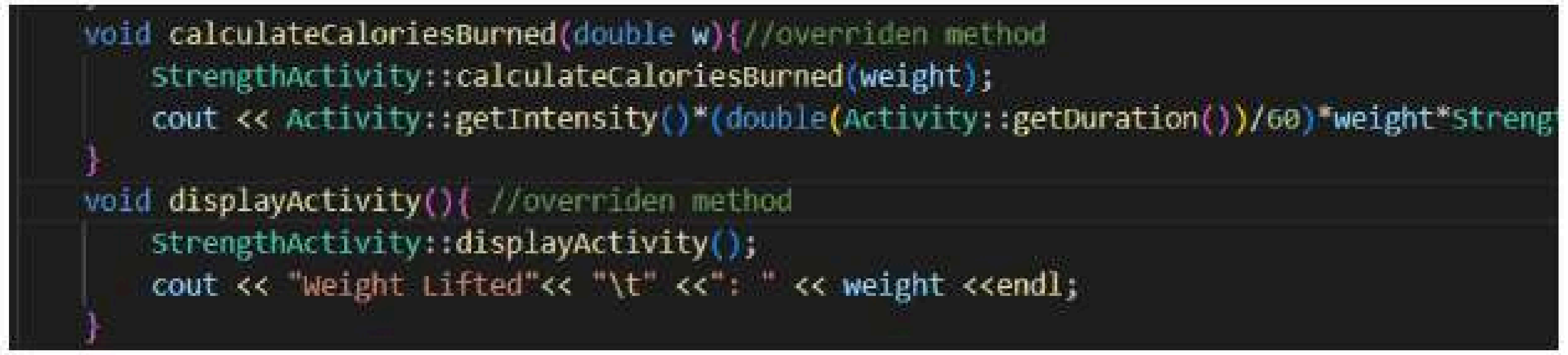
```
void displayActivity(){ //overriden method
    cout << "Cardio Activity" <<endl;
    Activity::displayActivity();
    cout << "Distance" << "\t" <<": " << distance << endl;
}
```

Derived Class : CardioActivity



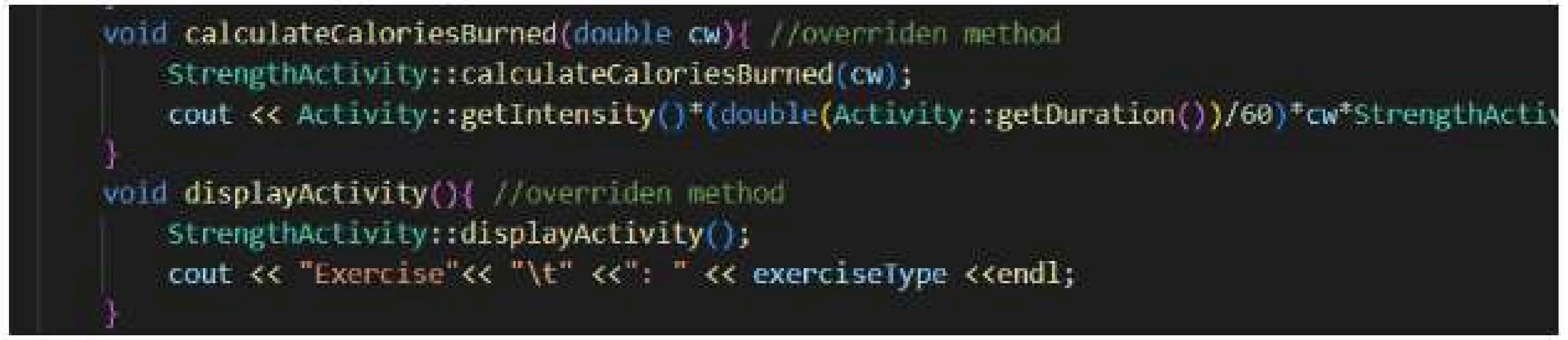
```
void calculateCaloriesBurned(double cw){ //overriden method
    Activity::calculateCaloriesBurned(cw);
    cout << distance*cw*Activity::getIntensity()* (double(Activity::getDuration())/60) << endl;
}
void displayActivity(){ //overriden method
    cout << "Cardio Activity" <<endl;
    Activity::displayActivity();
    cout << "Distance" << "\t" <<": " << distance << endl;

}
```

Derived Class : WeightExercise



```
void calculateCaloriesBurned(double w){//overriden method
    StrengthActivity::calculateCaloriesBurned(weight);
    cout << Activity::getIntensity()*(double(Activity::getDuration())/60)*weight*Streng
}
void displayActivity(){ //overriden method
    StrengthActivity::displayActivity();
    cout << "Weight Lifted"<< "\t" <<": " << weight <<endl;
}
```

Derived Class : BodyWeightExercise



```
void calculateCaloriesBurned(double cw){ //overriden method
    StrengthActivity::calculateCaloriesBurned(cw);
    cout << Activity::getIntensity()*(double(Activity::getDuration())/60)*cw*StrengthActiv
}
void displayActivity(){ //overriden method
    StrengthActivity::displayActivity();
    cout << "Exercise"<< "\t" <<": " << exerciseType <<endl;
}
```

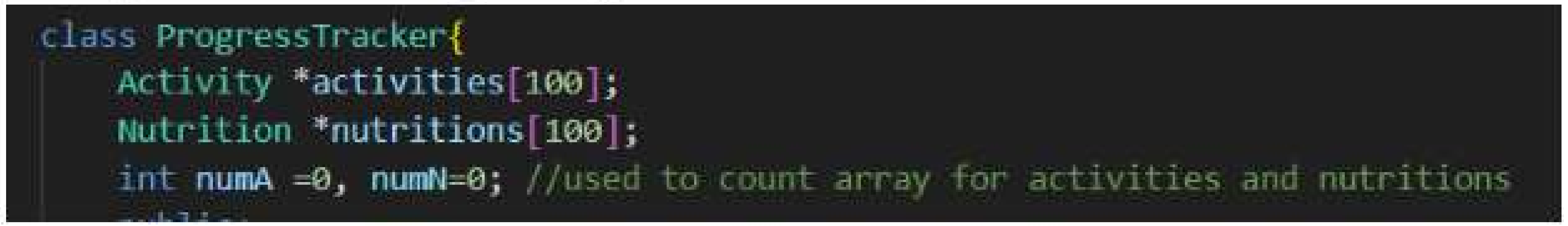
- SportsActivity and CardioActivity overrides the calculateCaloriesBurned() and displayActivity() methods from the base class, providing specific implementations.
- WeightExercise and BodyWeightExercise overrides the calculateCaloriesBurned() and displayActivity() methods from the Activity (base class) and the StrengthActivity (intermediate base class) to provide specific implementations.

Array of objects

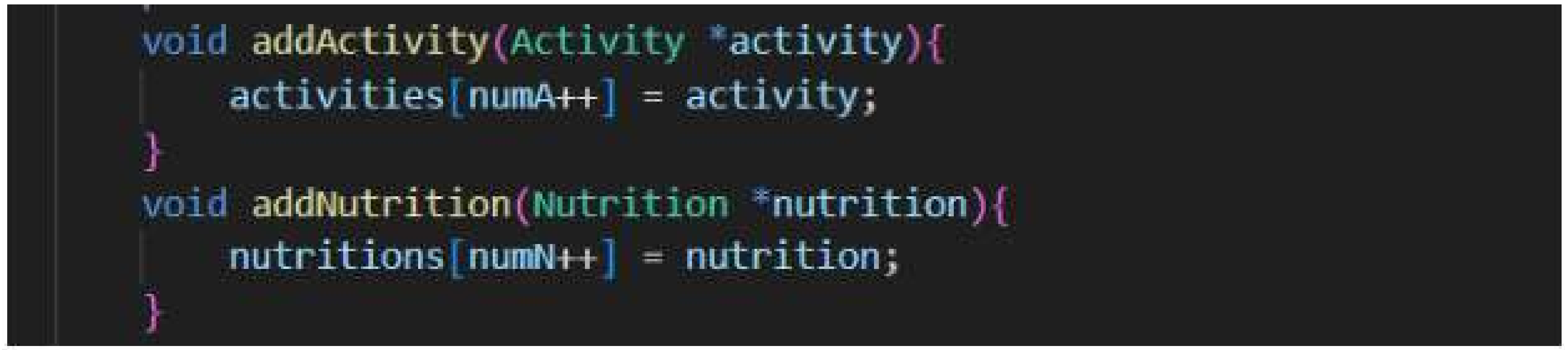
- An array of objects is a collection of instances of the same class type. It allows for the management of multiple objects of the same type in a structured way.
- The arrays of objects can be used to store and manipulate collections of objects using indices, providing a convenient way to handle multiple data items of the same type.
- This concept is implemented in the project to store the users, activities and nutrition information

Array of objects: Activities and Nutrition

Activity and Nutrition array in ProgressTracker Class

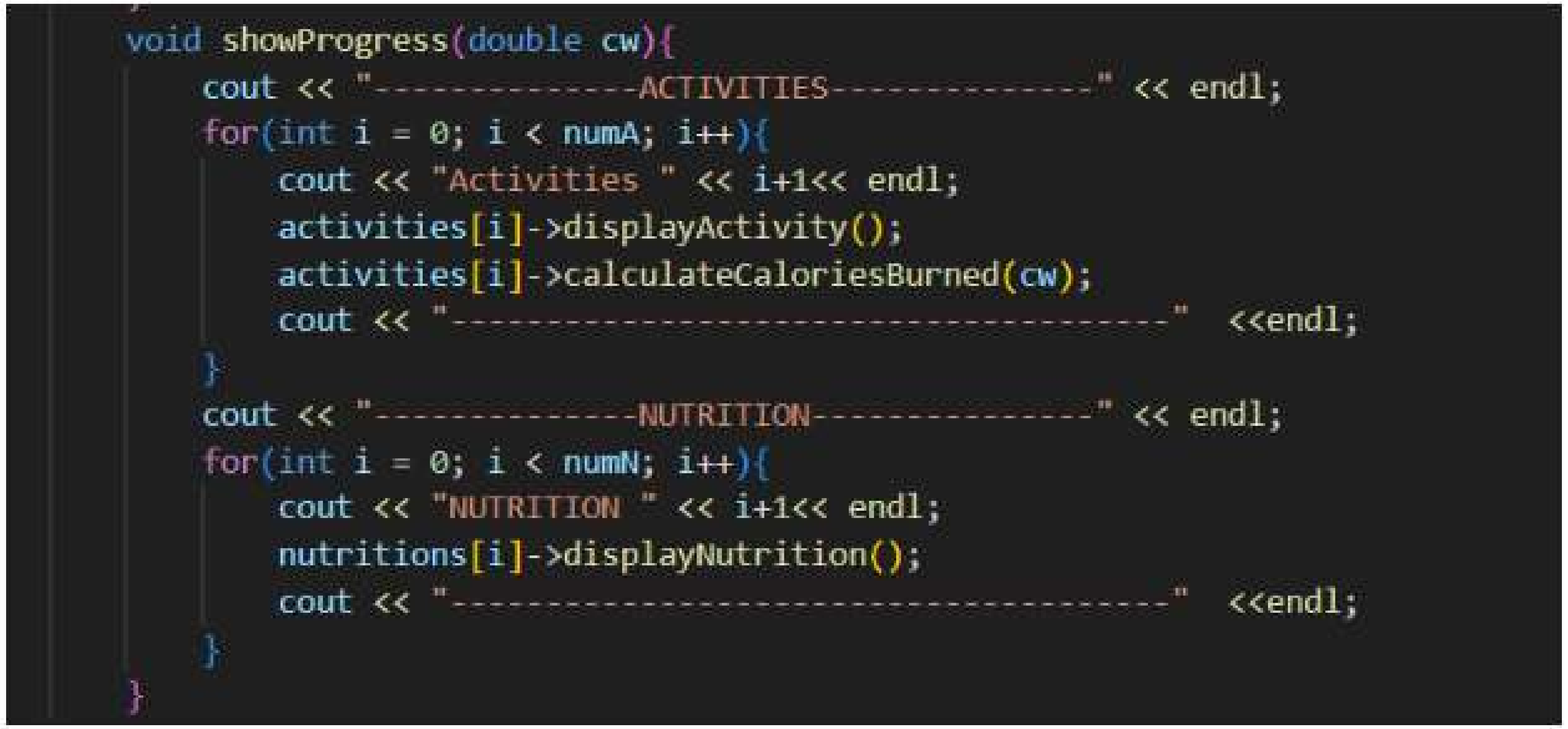

```
class ProgressTracker{
    Activity *activities[100];
    Nutrition *nutritions[100];
    int numA =0, numN=0; //used to count array for activities and nutritions
```

- Activity *activities[100] are an array that can hold up to 100 pointers to Activity objects.
- Nutrition *nutritions[100]; are an array that can hold up to 100 pointers to Nutrition objects.
- numA and numN variables used to count the array of activities and nutrition respectively.
- These arrays allow ProgressTracker to keep track of multiple activities and nutrition entries for users.


```
    void addActivity(Activity *activity){
        activities[numA++] = activity;
    }
    void addNutrition(Nutrition *nutrition){
        nutritions[numN++] = nutrition;
    }
```

- addActivity() method takes a pointer to an Activity object and adds it to the activities array, incrementing the numA counter.
- addNutrition() method takes a pointer to a Nutrition object and adds it to the nutritions array, incrementing the numN counter

Array of objects: Activities and Nutrition

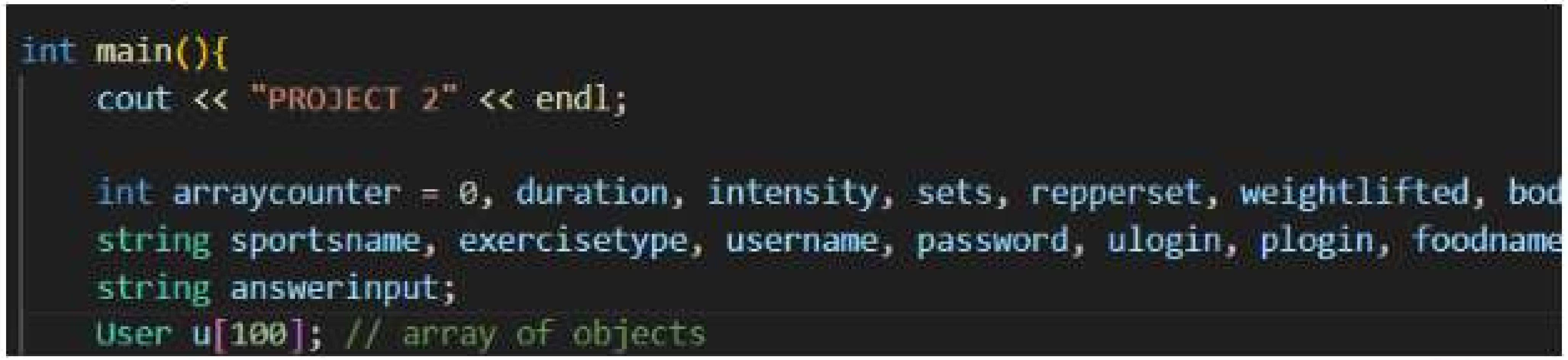


```
void showProgress(double cw){
    cout << "-------------ACTIVITIES-------------" << endl;
    for(int i = 0; i < numA; i++){
        cout << "Activities " << i+1<< endl;
        activities[i]->displayActivity();
        activities[i]->calculateCaloriesBurned(cw);
        cout << "-----------------------------------"  <<endl;
    }
    cout << "-------------NUTRITION--------------" << endl;
    for(int i = 0; i < numN; i++){
        cout << "NUTRITION " << i+1<< endl;
        nutritions[i]->displayNutrition();
        cout << "-----------------------------------"  <<endl;
    }
}
```

- showProgress() method iterates over the activities array using a for loop, displaying each activity and calculating calories burned.
- it also iterates over the nutritions array to display each nutrition entry.
- The use of loops to iterate over arrays allows for processing each object stored in the array efficiently.

Array of objects: Users

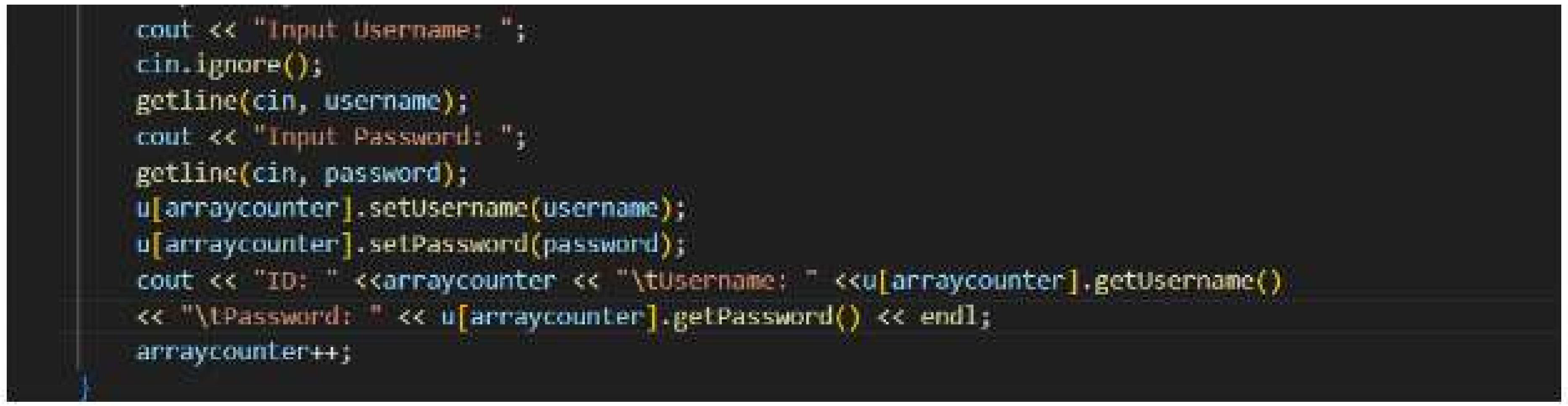
<u>Users array in Main Function</u>



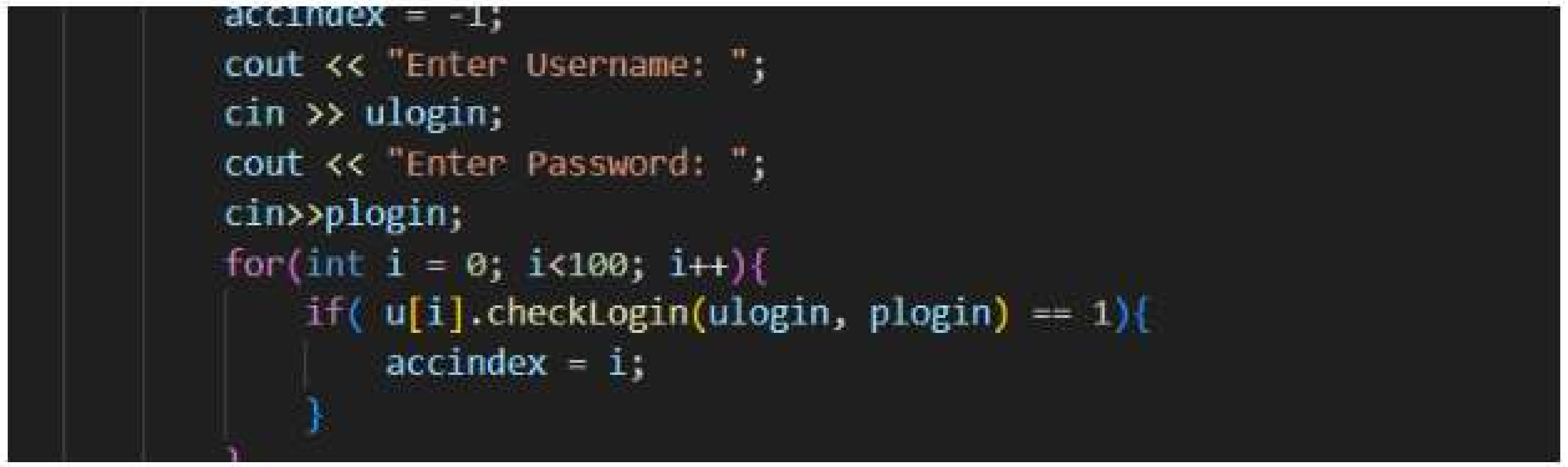
```
int main(){
    cout << "PROJECT 2" << endl;

    int arraycounter = 0, duration, intensity, sets, repperset, weightlifted, bod
    string sportsname, exercisetype, username, password, ulogin, plogin, foodname
    string answerinput;
    User u[100]; // array of objects
```

- User u[100] is an array of User objects that can hold up to 100 users.
- Each element in the array is an instance of the User class.
- The arraycounter variable are used to count the array of users.

- User Registration
- When a new user creates an account, their information is stored in the next available slot in the User array
- arraycounter is incremented to point to the next available slot in the array for the next new user.



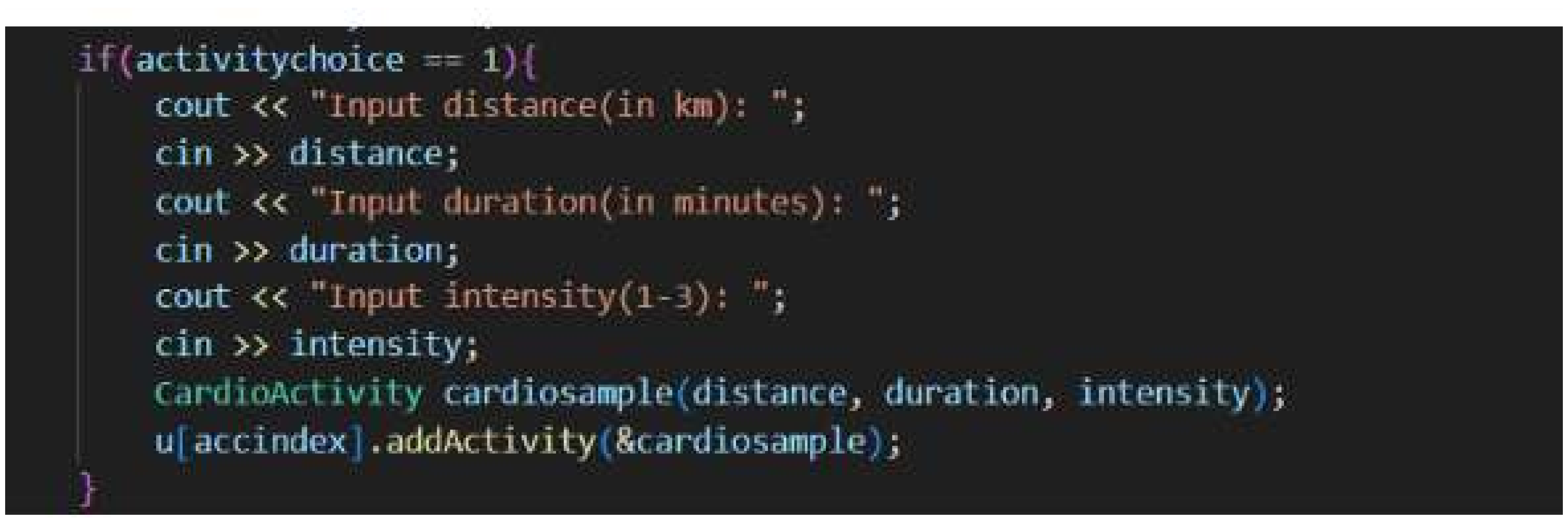
```
        cout << "Input Username: ";
        cin.ignore();
        getline(cin, username);
        cout << "Input Password: ";
        getline(cin, password);
        u[arraycounter].setUsername(username);
        u[arraycounter].setPassword(password);
        cout << "ID: " <<arraycounter << "\tUsername: " <<u[arraycounter].getUsername()
        << "\tPassword: " << u[arraycounter].getPassword() << endl;
        arraycounter++;
    }
```



```
            accindex = -1;
            cout << "Enter Username: ";
            cin >> ulogin;
            cout << "Enter Password: ";
            cin>>plogin;
            for(int i = 0; i<100; i++){
                if( u[i].checkLogin(ulogin, plogin) == 1){
                    accindex = i;
                }
```

- User Login
- When an existing user attempts to log in, the program iterates through the User array to verify the username and password:
- If a match is found, accindex is set to the index of the matching user.

Array of objects: Users



```
if(activitychoice == 1){
    cout << "Input distance(in km): ";
    cin >> distance;
    cout << "Input duration(in minutes): ";
    cin >> duration;
    cout << "Input intensity(1-3): ";
    cin >> intensity;
    CardioActivity cardiosample(distance, duration, intensity);
    u[accindex].addActivity(&cardiosample);
}
```

- Once logged in, users can perform various actions such as adding nutrition, adding activities, updating their profile, and showing progress.
- These actions are managed using the User object at the accindex.

Thank you