

# **OBJECT ORIENTED PROGRAMMING**

## **Using OOSimL**

### **An Overview - Part 1**

**José M. Garrido**  
**Department of Computer Science**

**Updated November 2016**

**College of Computing and Software Engineering**  
**Kennesaw State University**

**© 2015, J. M. Garrido**

## 1 Introduction

The hardware in a computer system is driven and controlled by software, which consist of the set of programs that execute in the computer. These control, manage the hardware, and carry out system and user tasks.

A program consists of data descriptions and instructions to manipulate this data. The types of programs that carry out system tasks are known as *system programs* and control the activities and functions of the various hardware components. An example of system programs is the operating system, such as Unix, Windows, MacOS, OS/2, and others.

User programs, also known as *application programs*, carry out tasks that solve specific problems for the users. These programs execute under control of the system programs. Application programs are developed by individuals and organizations for solving specific problems.

## 2 Programming Languages

A programming language is a set of syntax and semantic rules for writing programs. It is a formal notation that is used to write the data description and the instructions of a program. The programming language's syntax rules describe how to write sentences. The semantic rules describe the meaning of the sentences. These two types of rules must be consistent. There are several levels of computer programming languages, basically low-level and high-level languages.

### 2.1 Assembly Languages

The symbolic machine languages are also known as assembly languages and are low-level programming languages. In these languages, various mnemonic symbols represent operations and addresses in memory. These languages are hardware dependent; there is a different assembly language for every computer type. Assembly language is still used today for detailed control of hardware devices; it is also used when extremely efficient execution is required.

## 2.2 High-Level Programming Languages

A high-level programming language provides a programmer the facility to write instructions to the computer in the form of a program. A programming language must be expressive enough to help programmers in the writing of programs for a large family of problems.

High-level programming languages are hardware independent and more problem-oriented (for a given family of problems). These languages allow more readable programs, and are easier to write and maintain. Examples of these languages are Pascal, C, Cobol, Fortran, Algol, Ada, Smalltalk, C++, Eiffel, and Java.

Simula was the first object-oriented language developed in the mid-sixties and used to write simulation models. The original language was an extension of Algol. In a similar manner, C++ was developed as an extension to C in the early eighties.

Java was developed by Sun Microsystems in the mid-nineties and has far more capabilities than any other object-oriented programming language to date, unfortunately, it has a relatively low-level syntax.

Languages like C++ and Java can require considerable effort to learn and master. Several newer and experimental, higher-level, object-oriented programming languages have been developed. Each one has a particular goal. One such language is *OOSimL* (Object Oriented Simulation Language); its main purpose is to develop simulation models, make it easier to learn object-oriented programming principles, and help students transition to Java.

## 2.3 Compiling a Java Program

A program written in a programming language is compiled or translated to an equivalent program in machine language, which is the only programming language that the computer accepts for processing.

In addition to *compilation*, another step known as *linking* is required before a program can be executed. Programming languages like Java, require compilation and interpretation. This last step involves execution of the compiled program.

Two special programs are needed, the compiler and the interpreter. The Java compiler checks for syntax errors in the source program and translates it into *bytecode*, which is the program in an intermediate form. The Java bytecode is not dependent on any particular platform or computer system. To execute this bytecode, the Java Virtual Machine (JVM), carries out the

interpretation of the bytecode.

## 2.4 Compiling OOSimL Programs

Programs written in OOSimL (an object-oriented language that is higher level than Java), are compiled with the OOSimL compiler. This is a translator that checks for syntax errors in the language, and translates the program from OOSimL to Java. This compiler is a special program that is freely available from the following Web page:

<http://ksuweb.kennesaw.edu/~jgarrido/oosiml.html>

The OOSimL compilation is illustrated in Figure 1. Separate documents describe in further detail how to use the OOSimL compiler.



Figure 1: OOSimL Compilation to Java

## 3 Software Development

The main goal of software development is to solve a given real-world problem. This involves the following general tasks:

1. Understanding and describing the problem in a clear and unambiguous form
2. Designing a solution to the problem
3. Developing a computer solution to the problem.

### 3.1 Software Life Cycle

A software development process involves carrying out a sequence of activities. These activities represent the life of the software from birth to retirement. In most cases, the development process is also known as the *software life cycle*.

The software life cycle basically consists of the following phases:

1. Software development
2. Operational activities that applies the software in the problem environment
3. Maintenance activities that report defects in the software and the subsequent fixes and releases of new versions of the software
4. Software retirement when it can no longer be maintained.

The simplest model of the software life cycle is the *waterfall model*. This model represents the sequence of phases or activities to develop the software system through installation and maintenance of the software. In this model, the activity in a given phase cannot be started until the activity of the previous phase has been completed.

Figure 2 illustrates the sequence of activities or phases involved in the waterfall model of the software life cycle.

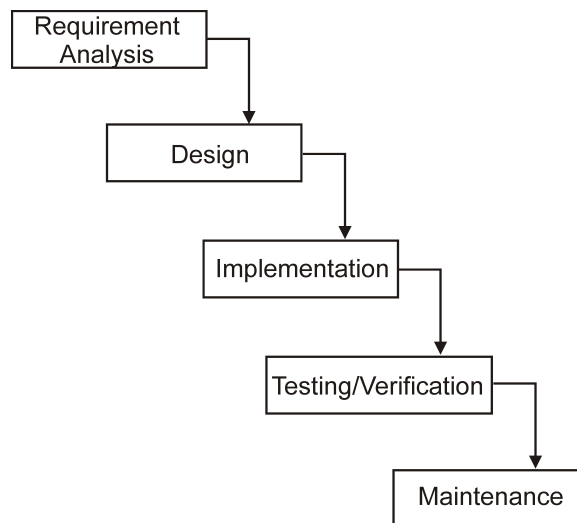


Figure 2: The waterfall model

The various phases of the software life cycle are the following:

1. Analysis that results in documenting the problem description and what the problem solution is supposed to accomplish.

2. Design, which involves describing and documenting the detailed structure and behavior of the system model.
3. Implementation of the software using a programming language.
4. Testing and verification of the programs.
5. Installation and maintenance that results in delivery, installation and maintenance of the programs.

In practice, the waterfall model is argued by many to be very inconvenient and difficult to use, mainly because for any non-trivial project, to get one phase of a software product's lifecycle completed and correct before moving on to the next phases.

There are some variations proposed for the waterfall model of the life cycle. These include returning to the previous phase when necessary. Recent trends in system development have emphasized an iterative approach, in which previous stages can be revised and enhanced.

A more complete model of software life cycle is the *spiral model* that incorporates the construction of *prototypes* in the early stages. A prototype is an early version of the application that does not have all the final characteristics. Other development approaches involve prototyping and rapid application development (RAD).

## 4 Modules

For complex problems or systems, the general approach is to divide the problem into smaller problems that are easier to solve. The partitioning of a problem into smaller parts is known as *decomposition*. These small parts are known as modules, which are easier to manage.

System design usually emphasizes modular structuring, also called modular decomposition. A problem is divided into smaller problems (or subproblems), and a solution is designed for each subproblem. Therefore, the solution to a problem consists of several smaller solutions corresponding to each of the subproblems. This approach is called modular design. Object-oriented design enhances modular design by providing classes as the most important decomposition (modular) unit. A program structure is basically organized as an assembly of classes.

## 5 Modeling Objects

*Abstraction* is used to model the objects in a problem domain. This involves the elimination of unessential characteristics. A model includes only the relevant aspects of the real-world system. Therefore, only the relevant objects and only the essential characteristics of these objects are included in the model.

Several levels of detail are needed to completely define objects and the collections of objects in a model. The activities in object-oriented modeling are:

1. Identifying the relevant objects for the model
2. Using abstraction to describe these objects
3. Defining collections of similar objects

An abstract representation is a simplified description and includes only the relevant or essential properties of part of a real system. A model is such an abstract representation of some part of the problem domain.

Real-world entities or real-world objects are the fundamental components of a real world system. Identifying and modeling real-world entities in the problem domain are the central focus of the object-oriented approach. A real-world entity has the responsibility of carrying out a specific task. Real-world entities identified in the real-world environment are modeled as objects.

Objects with similar characteristics are grouped into collections, and these are modeled as *classes*.

Objects and classes are described in a standard notation, the Unified Modeling Language (UML). This chapter and throughout the book simplified UML diagrams are employed when describing classes and objects.

### 5.1 Defining Classes

In modeling a real-world problem, collections of similar objects are identified. Classes are then defined as abstract descriptions of these collections of objects, which are objects with the same structure and behavior. A class defines the attributes, and behavior for all the objects of the class.

Figure 3 shows two collections of real-world objects and the modeling of the classes.

An object belongs to a collection or class, and any object of the class is an *instance* of the class.

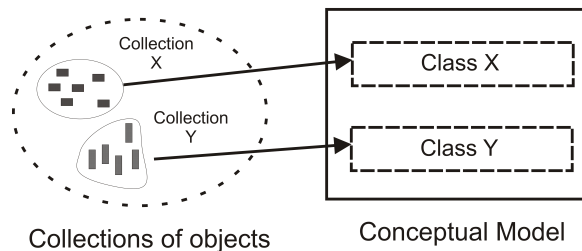


Figure 3: Collections of objects

A software implementation of class consists of:

- Data declarations that represent the attributes of the class
- Behavior representation as one or more operations (also known as functions and methods)

A class is represented graphically in UML as a simplified class diagram. The following is an example of a representation of class *Employee*, in a simplified UML diagram that shows the structure and behavior for all objects of this class. The attributes defined for this class are *salary*, *emp\_number*, *name*, and *emp\_date*. Figure 4 shows the diagram that describes class *Employee*.

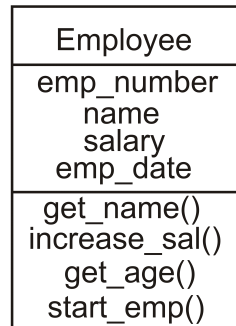
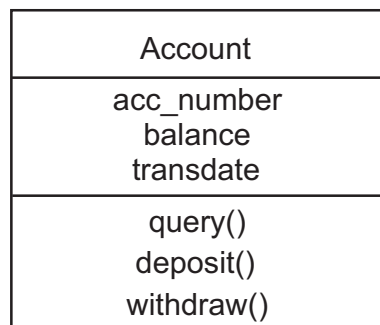
Another example of a class is shown next. Figure 5 shows the diagram that describes class *Account*.

## 5.2 Describing Objects

Three basic categories of real-world objects can be identified:

- Physical objects such as persons, animals, cars, balls, traffic lights, and so on
- Non-tangible objects such as contracts, accounts, and so on
- Conceptual objects that are used to represent part of the components or part of the behavior of the problem.



Figure 4: Diagram of class *Employee*Figure 5: Diagram of class *Account*

Objects exhibit independent behavior and interact with one another. Objects communicate by sending messages to each other. Every object has:

- State, represented by the set of properties (or attributes) and their associated values
- Behavior, represented by the operations, also known as methods, of the object
- Identity, which is an implicit or explicit property that can uniquely identify an object.

Two objects of class *Employee* are shown in Figure 6. This is the UML diagram that shows the state of the objects. The diagram is basically a

rectangle divided into three sections. The top section indicates the class of the object, the middle section includes the list of the attributes and their current values, and the bottom section includes the list of object operations.

:Employee	:Employee
879156 "John Martin" 45000.00 7/08/2001	4651961 "Nancy Dow" 38766.50 10/5/2004
get_name() increase_sal() get_age() start_emp()	get_name() increase_sal() get_age() start_emp()

Figure 6: Two objects of class *Employee*

The state of an object is defined by the values of its attributes. The two objects of class *Employee* have different states because their attributes have different values.

An object of class *Account* is shown in Figure 5. This object has three attributes, *acc\_num*, *balance*, and *transdate*. The object also has and three operations, *deposit* and *withdraw*, and *query*. The values of these attributes are also shown. The behavior of this object is dependent on another object that can start any of these operations.

## 6 Object Interactions

Object interaction involves objects sending *messages* to each other. The object that sends the message is the requestor of a service that can be provided by the receiver object.

An object sends a message to request a service, which is provided by the object receiving the message. The sender object is known as the *client* of a service, and the receiver object is known as the *supplier* of the service. Passive objects perform operations in response to messages.

An object of class *Employee* interacts with an object of class *Account*, by invoking an operation of the object of class *Account*.

These interactions are object-specific; a message is always sent to a specific object. This is also known as *method invocation*. A message normally contains three parts:

- The operation to be invoked or started
- The input data required by the operation to perform
- The result of the operation

The standard UML diagram known as the *communication diagram* is used to describe the general interaction between two or more objects (the sending of messages between objects).

Figure 7 shows a simple communication diagram to describe the interaction between an object of class *Employee* with an object of class *Account*. In this example, the object of class *Employee* invokes the *deposit* operation of the object of class *Account*. The first object sends a message to the second object (of class *Account*). As a result of this message, the object of class *Account* performs its *deposit* operation.

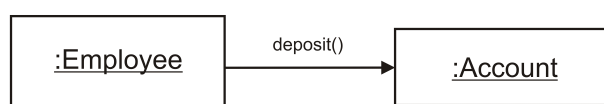


Figure 7: Communication diagram with two objects

## 7 Other Concepts of Object Orientation

Object orientation provides enhanced modularity in developing software systems. An application is basically a set of well-structured and related *modules*. The class is the most basic module of a program, and is the main decomposition unit of a program. In other words, a program is decomposed into a set of classes. The application is organized as an assembly of classes. Figure 8 illustrates the notion of an assembly of classes.

This is the *static view* of a program (that implements an application). The *dynamic view* of the application is a set of objects performing their behavior and interacting among themselves.

### 7.1 Encapsulation

The encapsulation principle describes an object as the integration of its attributes and behavior in a single unit. There is an imaginary protecting

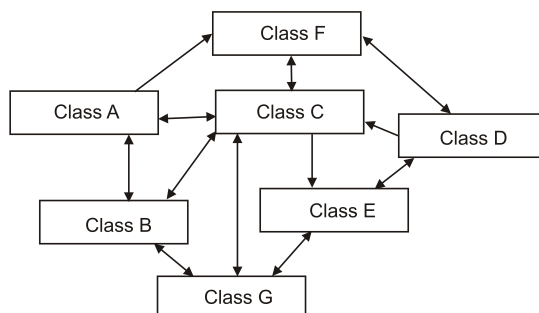


Figure 8: An assembly of classes

wall surrounding the object. This is considered a protection mechanism. To protect the features of an object, an access mode is specified for every feature.

The access mode specifies which features of the object can be accessed from other objects. If access to a feature (attribute or operation) is not allowed, the access mode of the feature is specified to be *private*. If a feature of an object is *public*, it is accessible from any other objects.

## 7.2 Information Hiding

As mentioned previously, an object that provides a set of services to other objects is known as a *provider* object, and all other objects that request these services by sending messages are known as *client* objects. An object can be a service provider for some services, and it can also be a client for services that it requests from other (provider) objects.

The principle of information hiding provides the description of a class only to show the services the objects of the class provide to other objects and hiding all implementation details. In this manner, a class description presents two views:

1. The *external view* of the objects of a class. This view consists of the list of services (or operations) that other objects can invoke. The list of services can be used as a service contract between the provider object and the client objects.

2. The *internal view* of the objects of the class. This view describes the implementation details of the data and the operations of the objects in a class. This information is hidden from other objects.

These two views of an object are described at two different levels of abstraction. The external view is at a higher level of abstraction. The external view is often known as the *class specification*, and the internal view as the *class implementation*.

With the external view, information about an object is limited to that only necessary for the object's features to be invoked by other objects. The rest of the knowledge about the object is not revealed.

In general, the external view of the objects should be kept separate from the internal view. The internal view of properties and operations of an object are hidden from other objects. The object presents its external view to other objects and shows what features (operations and attributes) are accessible.

## 8 Programs

An object-oriented program is an implementation of the classes that were modeled in the analysis and design phase of the software development process. This is the static view of a program that describes the structure of the program composed of one or more modules known as *classes*.

When the program executes, objects of these classes are created and made to interact among them. This is the dynamic view of a program, which describes the behavior of the program while it executes. This behavior consists of the set of *objects*, each one exhibiting individual behavior and interacting with other objects.

## 9 Definition of Classes

A class defines the structure and the behavior of the objects in that class. The software definition of a class consists of:

- The attribute declarations of the class
- Descriptions of the operations, known as methods or functions of the class

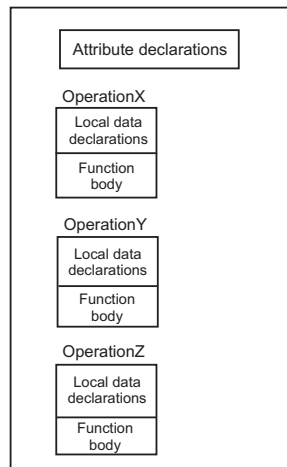


Figure 9: Structure of a typical class

The attributes in a class are defined as data declarations of the class. The behavior of the objects of the class is represented by the descriptions of the operations of the class.

The general structure of a class is shown in Figure 9 that illustrates the declarations of the attributes and the definitions of three operations: *OperationX*, *OperationY*, and *OperationZ*. Each of these operations consists of local data declarations and its instructions.

The smallest modular unit is an operation, also known as a function or method; it carries out a single task. It is not standalone unit because every function belongs to a class; a function is an internal decomposition unit.

## 10 Implementation of Programs

The software implementation of a program is carried out writing the code in a suitable programming language. Detailed design is often written in pseudo-code, which is a high-level notation at the level between the modeling diagrams and the programming language.

### 10.1 Programming Languages

Programming languages have well-defined syntax and semantic rules. The syntax is defined by a set of grammar rules and a vocabulary (a set of words). The legal sentences are constructed using sentences in the form of

*statements*. There are two groups of words: *reserved words* and *identifiers*.

Reserved words are the keywords of the language and have a predefined purpose in the language. These are used in most statements with precise meaning, for example, **class**, **inherits**, **variables**, **while**, **if**, and others.

Identifiers are names for variables, constants, functions, and classes that the programmer chooses, for example, *emp\_number*, *employee\_name*, *salary*, *emp\_date*, and on.

The main variable declarations in a class are the attributes. The other data declarations are the ones inside a function and are known as *local* variable declarations. The structure of object-oriented programs includes the following kinds of statements:

- Class definitions
- Declaration of data constants
- Declaration of simple variables
- Declaration of object references
- Definition of methods or functions

Within the body of function definitions, language instructions are included to create objects and to manipulate the objects created.

## 10.2 Implementing Classes

A class definition is implemented with a class header and additional statements; these are:

1. The **description** statement, which may be used to include a textual documentation or description of the class. This section ends with a start-slash (\*).
2. The **class** statement is used for the class header. This includes assigning a name to the class and other information related to the class.
3. The **private** keyword starts the section for the declarations of the private attributes of the class or definitions of the private operations.
4. The **protected** keyword starts the section for the declaration of protected attributes of the class or definitions of the projected operations.

5. The **public** keyword starts section for the declarations of the private attributes or the definition of the public operations of the class.
6. The **endclass** statement ends the class definition.

With OOSimL, the general syntactic definition of a class is:



**description**

```

      . . .
class < class_name > is
  private
    . . . [data declarations]
  protected
    . . . [data declarations]
  public
    . . . [data declarations]
  private
    [definitions of private operations]
  protected
    [definitions of private operations]
  public
    [definitions of public operations]
endclass < class_name >

```

## 11 Data Declaration

Data descriptions are the data declarations that define the class attributes. Three groups of data attributes can be defined in a class: constants, variables, and object references.

**constants**

```

      . . .

```

**variables**

```

      . . .

```

**object references**

```

      . . .

```

Each data declaration consists of:

- A unique name to identify the data item
- An optional initial value
- The type of the data item

The name of a data item is an identifier and is given by the programmer; it must be different from any keyword in OOSimL (or in Java).

## 11.1 Data Names

Text symbols are used in all algorithm description and in the source program. The special symbols that indicate essential parts of an algorithm are called *keywords*. These are reserved words and cannot be used for any other purpose. The symbols used for uniquely identifying the data items and are called *identifiers* and are defined by the programmer. For example, *x*, *y*, *z*, *s*, and *area*.

The data items normally change their values when they are manipulated by the instructions in the various operations. These data items are known as *variables* because their values change when instructions are applied on them. Those data items that do not change their values are called *constants*. These data items are given an initial value that will never change.

## 11.2 Data Types

Data types can be of two groups: *elementary* (also known as primitive) data types and classes.

Elementary types are classified into the three following categories:

- Numeric
- Text
- Boolean

There are three groups of numeric types: **integer**, **float**, and **double**. The non-integer types are also known as real (or fractional), which means that the numerical values have a fractional part.

Integer values are countable to a finite value, for example, age, number of automobiles, number of pages in a book, and so on. Real values (of type **float**) have a decimal point; for example, cost amount of an item, the height of a building, current temperature in a room, a time interval (period). These values cannot be expressed as integers. Values of type **double** provide more precision than type **float**.

There are of two basic types of text data items: **character** and type **string**. Data items of type **string** consist of a sequence of characters. The values for these two types of data items are textual values.

Another data type is the one in which the values of the variables can take a truth-value (true or false); these variables are of type **boolean**.

Complex types are the classes and are used as types for object reference variables in all object-oriented programs. Values of these variables are references to objects.

### 11.3 Data Declarations in OOSimL

Each data description includes the name or identifier for variable or constant and its type. The initial value of a variable or constant is optional. There are two general categories of variables:

- Variables of elementary or primitive type
- Object reference variables

In object-oriented programming languages, the programmer defines classes as types for object reference variables, then creating objects of these classes.

#### 11.3.1 Variables of Elementary Types

The OOSimL statement for the declaration of variables of elementary types has the following form:

```
define < variable_name > of type < elementary_type >
```

Examples of data declarations in OOSimL of two constants of types **integer** and **double**, and three variables of type **integer**, **float**, and **boolean**.

```
constants
  define MAX_PERIOD = 24 of type integer
  define PI = 3.1416 of type double
variables
  define number_elements of type integer
  define salary of type float
  define act_flag of type boolean
```

#### 11.3.2 Object References

An object reference variable can refer to an object. The following OOSimL statement declares object references variables and has the structure:

```
define < object_ref_name > of class < class_name >
```

The following statements declare an object reference variable called *server\_obj* of class *Server*, and an object reference variable *car1* of class *Car*.

```

object references
  define serverobj of class Server
  define car1 of class Car

```

## 11.4 Temporal and Spatial Properties

Additional properties of data items declared are important to fully understand identifying data items in software development, these concepts are:

- The *scope* of a data item is that portion of a program in which statements can reference that data item. Variables and constants declared as attributes of the class can be accessed from anywhere in the class. Instructions in any function of the class can use these data items. Local declarations define data items that can only be used by instructions in the function in which they have been declared.
- The *persistence* of a data item is the interval of time that the data item exists—the lifetime of the data item. The lifetime of a variable declared as an attribute of a class, exists during the complete life of an object of the class. Data items declared locally will normally have a life time only during which the function executes.

## 12 Functions

Classes consist of attribute declarations and definitions of functions (or methods).

A function carries out a specific task of a class. Figure 10 illustrates the general structure of a function. The definition of a function consists of two basic parts:

- Local data declarations
- The sequence of statements with instructions that manipulate the data

The body of a function is composed of data declaration and instructions that manipulate the data. The scope of the data declared in the function is *local* to the function. The data in a function only exists during execution of the function; their persistence is limited to the lifetime of the function.

The brief documentation about the purpose of the function is described in the paragraph that starts with the keyword **description** and ends with a

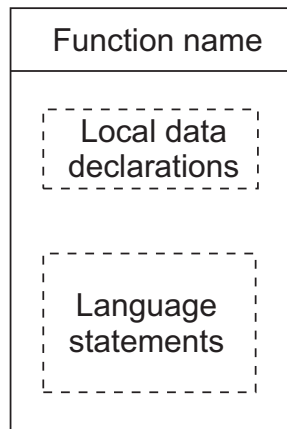


Figure 10: Structure of a function

star-slash (`*/`). Other comments can be included at any point in the function to clarify and/or document the function. Line comments start with a double forward slash (`//`) and end with the line.

The name of the function is declared in the **function** statement. This name is used when the function is called or invoked by some other function. In OOSimL, the basic form of a function definition is as follows.

```

description
. . .
function < function_name > is
  constants
  . . .
  variables
  . . .
  object references
  . . .
  begin
  . . . [instructions]
endfun < function_name >

```

The keyword **function** starts the header followed by the name of the function; the body of the function starts with the keyword **is**. The function definition ends with the **endfun** keyword and the name of the function.

Similarly to the data declarations in the class, the data declarations are

divided into constant, variable, and object references. Constant, variable, and object declarations are optional. The instructions in the body of the function appear between the keywords **begin** and **endfun**.

The special function, *main*, starts and terminates the execution of the entire program; it is the control function of the program. One of the classes of the program must include this function.