3

Introduction to Classes and Objects

◀ ▶

You will see something new. Two things. And I call them Thing One and Thing Two.

— Dr. Theodor Seuss Geisel

Nothing can have value without being an object of utility.

— Karl Marx

Your public servants serve you right.

— Adlai E. Stevenson

Knowing how to answer one who speaks, To reply to one who sends a message.

— Amenemope

◀ ▶

OBJECTIVES

- In this chapter you will learn:
- What classes, objects, member functions and data members are.
- How to define a class and use it to create an object.
- How to define member functions in a class to implement the class's behaviors.
- How to declare data members in a class to implement the class's attributes.
- How to call a member function of an object to make that member function perform its task.
- The differences between data members of a class and local variables of a function.
- How to use a constructor to ensure that an object's data is initialized when the object is created.
- How to engineer a class to separate its interface from its implementation and encourage reuse.

Outline

3.1 Introduction

3.2 Classes, Objects, Member Functions and Data Members

3.3 Overview of the Chapter Examples

3.4 Defining a Class with a Member Function

3.5 Defining a Member Function with a Parameter

3.6 Data Members, *set* Functions and *get* Functions

3.7 Initializing Objects with Constructors

3.8 Placing a Class in a Separate File for Reusability

3.9 Separating Interface from Implementation

3.10 Validating Data with set Functions

3.11 (Optional) Software Engineering Case Study: Identifying the Classes in the ATM Requirements Document

3.12 Wrap-Up

3.1 Introduction

- **Programs from Chapter 2**
  - **All statements were located in function `main`**
- **Typically**
  - **Programs will consist of**
    - **Function `main` and**
    - **One or more classes**
      - **Each containing data members and member functions**

3.2 Classes, Objects, Member Functions and Data Members

- Review of classes: Car example
  - Functions describe the mechanisms that perform a tasks, such as acceleration
    - Hides complex tasks from user, just as a driver can use the pedal to accelerate without needing to know how the acceleration is performed
  - Classes must be defined before they can be used, car must be built before it can be driven
  - Many car objects created from same class, many cars built from same engineering drawing

3.2 Classes, Objects, Member Functions and Data Members (Cont.)

- **Review of classes: Car example (Cont.)**
  - **Member-function calls send messages to an object to perform tasks, just like pressing the gas pedal sends a message to the car to accelerate**
  - **Objects and cars both have attributes, like color and miles driven**

◀ ▶

3.3 Overview of the Chapter Examples

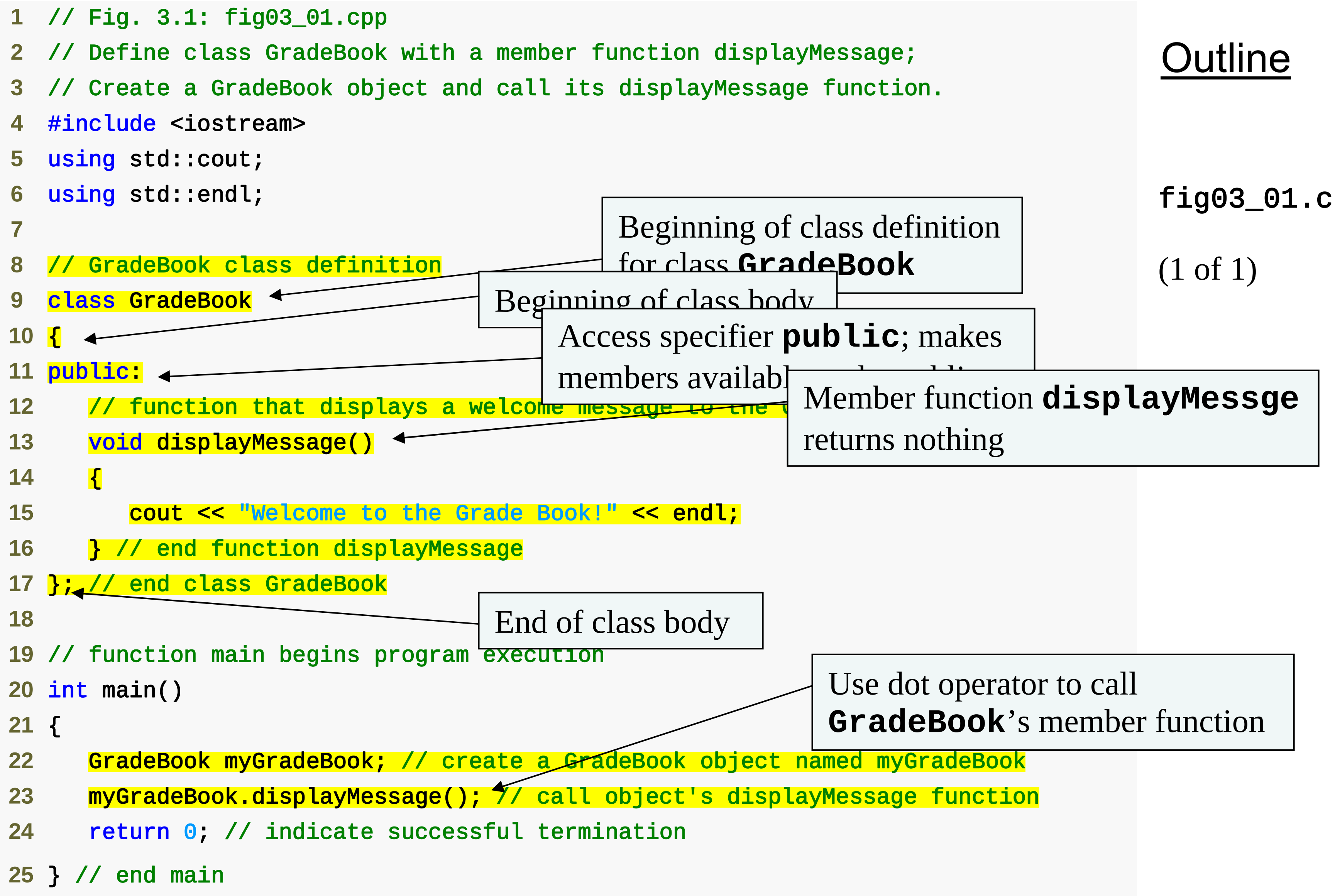
- **Seven simple examples**
  - **Examples used to build a `GradeBook` class**
- **Topics covered:**
  - **Member functions**
  - **Data members**
  - **Clients of a class**
    - **Other classes or functions that call the member functions of this class’s objects**
  - **Separating interface from implementation**
  - **Data validation**
    - **Ensures that data in an object is in a particular format or range**

◀ ▶

3.4 Defining a Class With a Member Function

- **Class definition**
  - **Tells compiler what member functions and data members belong to the class**
  - **Keyword `class` followed by the class's name**
  - **Class body is enclosed in braces (`{}`)**
    - **Specifies data members and member functions**
    - **Access-specifier `public:`**
      - **Indicates that a member function or data member is accessible to other functions and member functions of other classes**

Outline



```cpp
// Fig. 3.1: fig03_01.cpp
// Define class GradeBook with a member function displayMessage;
// Create a GradeBook object and call its displayMessage function.
#include <iostream>
using std::cout;
using std::endl;

// GradeBook class definition
class GradeBook
{
public:
   // function that displays a welcome message to the
   void displayMessage()
   {
      cout << "Welcome to the Grade Book!" << endl;
   } // end function displayMessage
}; // end class GradeBook

// function main begins program execution
int main()
{
   GradeBook myGradeBook; // create a GradeBook object named myGradeBook
   myGradeBook.displayMessage(); // call object's displayMessage function
   return 0; // indicate successful termination
} // end main
```

Beginning of class definition for class **GradeBook**

Beginning of class body

Access specifier **public**; makes members availabl

Member function **displayMessge** returns nothing

End of class body

Use dot operator to call **GradeBook**'s member function

```
Welcome to the Grade Book!
```

fig03_01.cpp

(1 of 1)

Common Programming Error 3.1

Forgetting the semicolon at the end of a class definition is a syntax error.

3.4 Defining a Class With a Member Function (Cont.)

- Member function definition
  - Return type of a function
    - Indicates the type of value returned by the function when it completes its task
    - `void` indicates that the function does not return any value
  - Function names must be a valid identifier
  - Parentheses after function name indicate that it is a function
  - Function body contains statements that perform the function’s task
    - Delimited by braces (`{}`)

Common Programming Error 3.2

Returning a value from a function whose return type has been declared `void` is a compilation error.

Common Programming Error 3.3

Defining a function inside another function is a syntax error.

3.4 Defining a Class With a Member Function (Cont.)

- Using a class
  - A class is a user-defined type (or programmer-defined type)
    - Can be used to create objects
      - Variables of the class type
    - C++ is an extensible language
  - Dot operator (`.`)
    - Used to access an object’s data members and member functions
    - Example
      - `myGradeBook.displayMessage()`
        - Call member function `displayMessage` of `GradeBook` object `myGradeBook`

◀ ▶

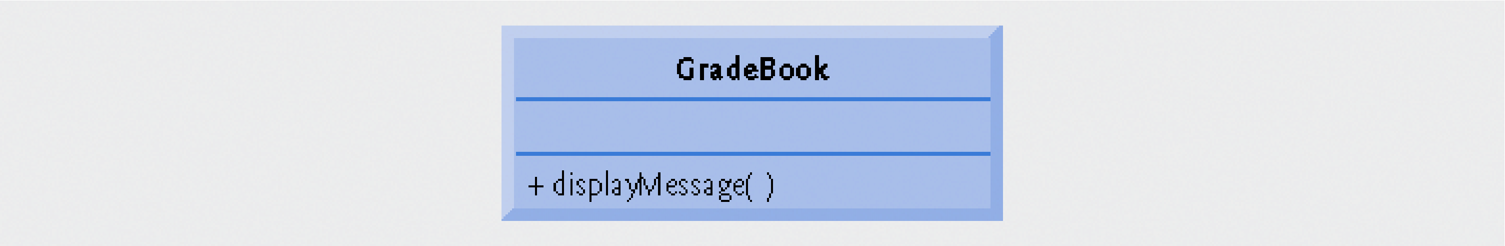


Fig.3.2 | UML class diagram indicating that class `GradeBook` has a public `displayMessage` operation.

3.4 Defining a Class With a Member Function (Cont.)

- UML class diagram
  - A rectangle with three compartments
    - Top compartment contains the name of the class
    - Middle compartment contains the class's attributes
    - Bottom compartment contains the class's operations
      - A (+) in front of an operation indicates it is public

3.5 Defining a Member Function with a Parameter

- **Function parameter(s)**
  - **Information needed by a function to perform its task**
- **Function argument(s)**
  - **Values supplied by a function call for each of the function’s parameters**
    - **Argument values are copied into function parameters**

3.5 Defining a Member Function with a Parameter (Cont.)

- A `string`
  - Represents a string of characters
  - An object of C++ Standard Library class `std::string`
    - Defined in header file `<string>`
- Library function `getline`
  - Used to retrieve input until newline is encountered
  - Example
    - `getline( cin, nameOfCourse );`
      - Inputs a line from standard input into string `object nameOfCourse`

Outline

```cpp
// Fig. 3.3: fig03_03.cpp
// Define class GradeBook with a member function that takes a parameter;
// Create a GradeBook object and call its displayMessage function.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include <string> // program uses C++ standard string class
using std::string;
using std::getline;

// GradeBook class definition
class GradeBook
{
public:
   // function that displays a welcome message to the GradeBook user
   void displayMessage( string courseName )
   {
      cout << "Welcome to the grade book for\n" << courseName << "!"
         << endl;
   } // end function displayMessage
}; // end class GradeBook

// function main begins program execution
int main()
{
   string nameOfCourse; // string of characters to store the course name
   GradeBook myGradeBook; // create a GradeBook object named myGradeBook

```

Include `string` class definition

Member function parameter

Use the function parameter as a variable

fig03_03.cpp

(1 of 2)

Outline

fig03_03.cpp

(2 of 2)

```
   // prompt for and input course name
   cout << "Please enter the course name:" << endl;
   getline( cin, nameOfCourse ); // read a course name with blanks
   cout << endl; // output a blank line

   // call myGradeBook's displayMessage function
   // and pass nameOfCourse as an argument
   myGradeBook.displayMessage( nameOfCourse );
   return 0; // indicate successful termination
} // end main
```

Passing an argument to the member function

```
Please enter the course name:
CS101 Introduction to C++ Programming

Welcome to the grade book for
CS101 Introduction to C++ Programming!
```

3.5 Defining a Member Function with a Parameter (Cont.)

- Parameter Lists
  - Additional information needed by a function
  - Located in parentheses following the function name
  - Function may have any number of parameters
    - Parameters separated by commas
  - Number, order and types of arguments in a function call must match the number, order and types of parameters in the called function’s parameter list
  - Modeled in UML
    - Parameter name, followed by a colon and the parameter type in the member function’s parentheses

Common Programming Error 3.4

Placing a semicolon after the right parenthesis enclosing the parameter list of a function definition is a syntax error.

Common Programming Error 3.5

Defining a function parameter again as a local variable in the function is a compilation error.

Good Programming Practice 3.1

To avoid ambiguity, do not use the same names for the arguments passed to a function and the corresponding parameters in the function definition.

Good Programming Practice 3.2

Choosing meaningful function names and meaningful parameter names makes programs more readable and helps avoid excessive use of comments.

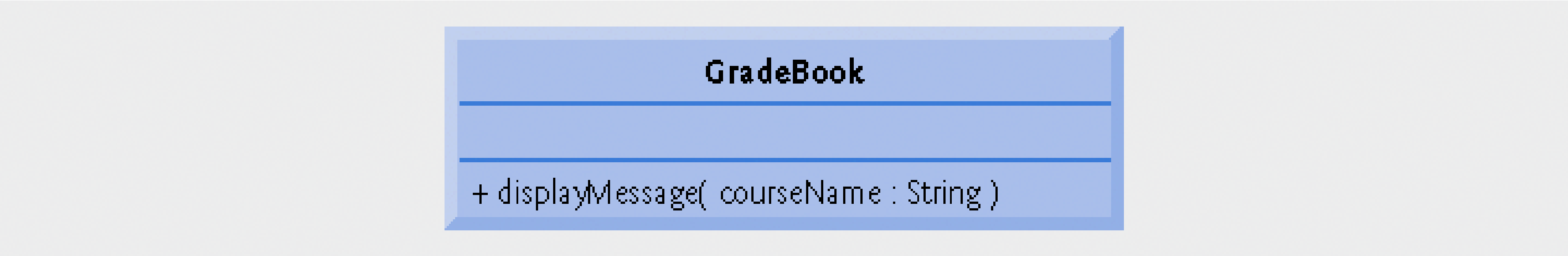


Fig.3.4 | UML class diagram indicating that class `GradeBook` has a `displayMessage` operation with a `courseName` parameter of UML type `String`.

3.6 Data Members, *set* Functions and *get* Functions

- Local variables
  - Variables declared in a function definition’s body
    - Cannot be used outside of that function body
  - When a function terminates
    - The values of its local variables are lost
- Attributes
  - Exist throughout the life of the object
  - Represented as data members
    - Variables in a class definition
  - Each object of class maintains its own copy of attributes

Outline

fig03_05.cpp

(1 of 3)

```
// Fig. 3.5: fig03_05.cpp
// Define class GradeBook that contains a courseName data member
// and member functions to set and get its value;
// Create and manipulate a GradeBook object with these functions.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include <string> // program uses C++ standard string class
using std::string;
using std::getline;

// GradeBook class definition
class GradeBook
{
public:
   // function that sets the course name
   void setCourseName( string name )
   {
      courseName = name; // store the course name in the object
   } // end function setCourseName

   // function that gets the course name
   string getCourseName()
   {
      return courseName; // return the object's courseName
   } // end function getCourseName

```

set function modifies **private** data

get function accesses **private** data

Outline

```
   // function that displays a welcome message
   void displayMessage()
   {
      // this statement calls getCourseName to get the
      // name of the course this GradeBook represents
      cout << "Welcome to the grade book for\n" << getCourseName() << "!"
         << endl;
   } // end function displayMessage
private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook

// function main
int main()
{
   string nameOfCourse; // string of characters to store the course name
   GradeBook myGradeBook; // create a GradeBook object named myGradeBook

   // display initial value of courseName
   cout << "Initial course name is: " << myGradeBook.getCourseName()
      << endl;

```

fig03_05.cpp

(2 of 3)

Use *set* and *get* functions, even within the class

`private` members accessible only to member functions of the class

Accessing **`private`** data outside class definition

Outline

```
   // prompt for, input and set course name
   cout << "\nPlease enter the course name:" << endl;
   getline( cin, nameOfCourse ); // read a course name with blanks
   myGradeBook.setCourseName( nameOfCourse ); // set the course name

   cout << endl; // outputs a blank line
   myGradeBook.displayMessage(); // display message with new course name
   return 0; // indicate successful termination
} // end main
```

Modifying **`private`** data outside class definition

`fig03_05.cpp`

of 3)

```
Initial course name is:

Please enter the course name:
CS101 Introduction to C++ Programming

Welcome to the grade book for
CS101 Introduction to C++ Programming!
```

Good Programming Practice 3.3

Place a blank line between member-function definitions to enhance program readability.

3.6 Data Members, *set* Functions and *get* Functions (Cont.)

- Access-specifier `private`
  - Makes a data member or member function accessible only to member functions of the class
  - `private` is the default access for class members
  - Data hiding
- Returning a value from a function
  - A function that specifies a return type other than `void`
    - Returns a value to its calling function

Software Engineering Observation 3.1

As a rule of thumb, data members should be declared `private` and member functions should be declared `public`. (We will see that it is appropriate to declare certain member functions `private`, if they are to be accessed only by other member functions of the class.)

Common Programming Error 3.6

An attempt by a function, which is not a member of a particular class (or a `friend` of that class, as we will see in Chapter 10), to access a `private` member of that class is a compilation error.

Good Programming Practice 3.4

Despite the fact that the `public` and `private` access specifiers may be repeated and intermixed, list all the `public` members of a class first in one group and then list all the `private` mem-bers in another group. This focuses the client’s attention on the class’s `public` interface, rather than on the class’s implementation.

Good Programming Practice 3.5

If you choose to list the `private` members first in a class definition, explicitly use the `private` access specifier despite the fact that `private` is assumed by default. This improves pro-gram clarity.

◀ ▶

Software Engineering Observation 3.2

We will learn in Chapter 10, Classes: Part 2, that functions and classes declared by a class to be `friends` can access the `private` members of the class.

Error-Prevention Tip 3.1

Making the data members of a class `private` and the member functions of the class `public` facilitates debugging because problems with data manipulations are localized to either the class's member functions or the `friend`s of the class.

Common Programming Error 3.7

Forgetting to return a value from a function that is supposed to return a value is a compilation error.

◀ ▶

3.6 Data Members, *set* Functions and *get* Functions (Cont.)

- **Software engineering with *set* and *get* functions**
  - **`public` member functions that allow clients of a class to set or get the values of `private` data members**
  - ***set* functions sometimes called mutators and *get* functions sometimes called accessors**
  - **Allows the creator of the class to control how clients access `private` data**
  - **Should also be used by other member functions of the same class**

Good Programming Practice 3.6

Always try to localize the effects of changes to a class’s data members by accessing and manipulating the data members through their get and set functions. Changes to the name of a data member or the data type used to store a data member then affect only the corresponding get and set functions, but not the callers of those functions.

◀ ▶

Software Engineering Observation 3.3

It is important to write programs that are understandable and easy to maintain. Change is the rule rather than the exception. Programmers should anticipate that their code will be modified.

Software Engineering Observation 3.4

The class designer need not provide set or get functions for each `private` data item; these capabilities should be provided only when appropriate. If a service is useful to the client code, that service should typically be provided in the class’s `public` interface.

3.6 Data Members, *set* Functions and *get* Functions (Cont.)

- UML diagram
  - Indicating the return type of an operation
    - Place a colon and the return type after the parentheses following the operation name
  - Minus sign used to indicate `private` members

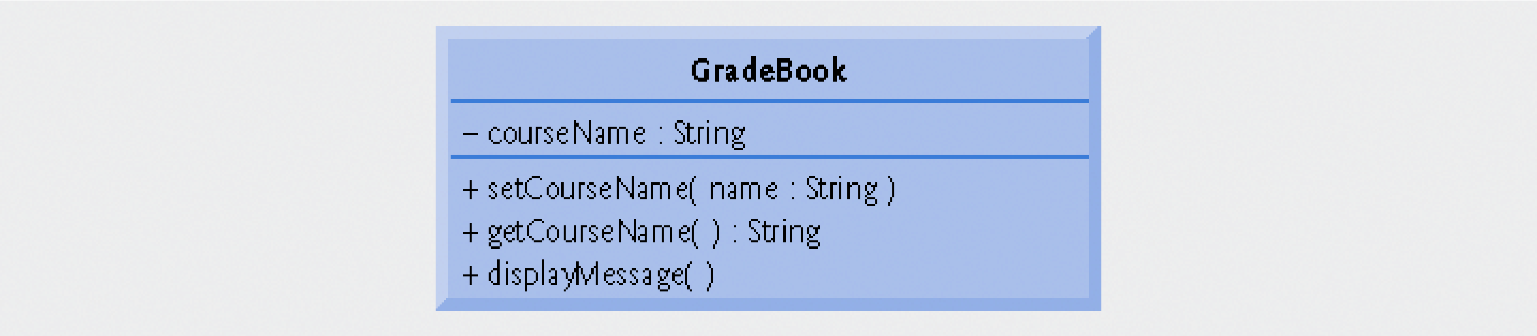


Fig.3.6 | UML class diagram for class `GradeBook` with a private `courseName` attribute and public operations `setCourseName`, `getCourseName` and `displayMessage`.

3.7 Initializing Objects with Constructors

- Constructors
  - Functions used to initialize an object’s data when it is created
    - Call made implicitly when object is created
    - Must be defined with the same name as the class
    - Cannot return values
      - Not even `void`
  - Default constructor has no parameters
    - The compiler will provide one when a class does not explicitly include a constructor
      - Compiler’s default constructor only calls constructors of data members that are objects of classes

```cpp
// Fig. 3.7: fig03_07.cpp
// Instantiating multiple objects of the GradeBook class and using
// the GradeBook constructor to specify the course name
// when each GradeBook object is created.
#include <iostream>
using std::cout;
using std::endl;

#include <string> // program uses C++ standard string class
using std::string;

// GradeBook class definition
class GradeBook
{
public:
   // constructor initializes courseName with string supplied as argument
   GradeBook( string name )
   {
      setCourseName( name ); // call set function to initialize courseName
   } // end GradeBook constructor

   // function to set the course name
   void setCourseName( string name )
   {
      courseName = name; // store the course name in the object
   } // end function setCourseName

```

Constructor has same name as class and no return type

Initialize data member

Outline

fig03_07.cpp

(1 of 3)

```
   // function to get the course name
   string getCourseName()
   {
      return courseName; // return object's courseName
   } // end function getCourseName

   // display a welcome message to the GradeBook user
   void displayMessage()
   {
      // call getCourseName to get the courseName
      cout << "Welcome to the grade book for\n" << getCourseName()
         << "!" << endl;
   } // end function displayMessage
private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook

```

Outline

fig03_07.cpp

(2 of 3)

Outline

fig03_07.cpp

(3 of 3)

```cpp
// function main begins program execution
int main()
{
   // create two GradeBook objects
   GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
   GradeBook gradeBook2( "CS102 Data Structures in C++" );

   // display initial value of courseName for each GradeBook
   cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
      << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
      << endl;
   return 0; // indicate successful termination
} // end main
```

Creating objects implicitly calls the constructor

```
gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++
```

Error-Prevention Tip 3.2

Unless no initialization of your class’s data members is necessary (almost never), provide a constructor to ensure that your class’s data members are initialized with meaningful values when each new object of your class is created.

Software Engineering Observation 3.5

Data members can be initialized in a constructor of the class or their values may be set later after the object is created. However, it is a good software engineering practice to ensure that an object is fully initialized before the client code invokes the object’s member functions. In general, you should not rely on the client code to ensure that an object gets initialized properly.

3.7 Initializing Objects with Constructors (Cont.)

- Constructors in a UML class diagram
  - Appear in third compartment, with operations
  - To distinguish a constructor from a class's operations
    - UML places the word "constructor" between guillemets before the constructor's name
      - <<constructor>>
  - Usually placed before other operations

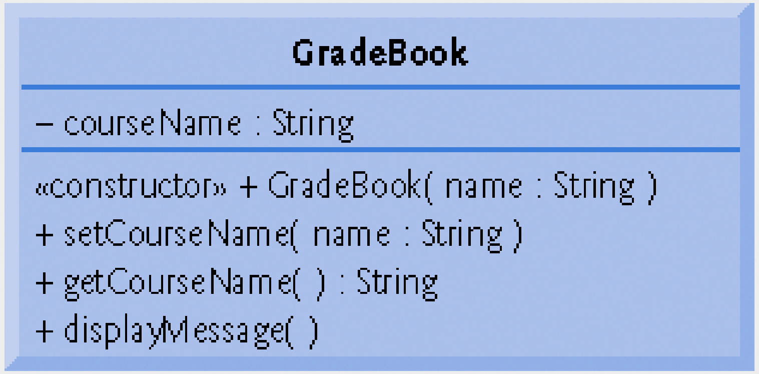

GradeBook
– courseName : String
«constructor» + GradeBook(name : String)
+ setCourseName(name : String)
+ getCourseName( ) : String
+ displayMessage( )

Fig.3.8 | UML class diagram indicating that class `GradeBook` has a constructor with a name parameter of UML type `String`.

3.8 Placing a Class in a Separate File for Reusability

- `.cpp` file is known as a source-code file
- Header files
  - Separate files in which class definitions are placed
    - Allow compiler to recognize the classes when used elsewhere
  - Generally have `.h` filename extensions
- Driver files
  - Program used to test software (such as classes)
  - Contains a `main` function so it can be executed

```
// Fig. 3.9: GradeBook.h
// GradeBook class definition in a separate file from main.
#include <iostream>
using std::cout;
using std::endl;

#include <string> // class GradeBook uses C++ standard string class
using std::string;

// GradeBook class definition
class GradeBook
{
public:
   // constructor initializes courseName with string supplied as argument
   GradeBook( string name )
   {
      setCourseName( name ); // call set function to initialize courseName
   } // end GradeBook constructor

   // function to set the course name
   void setCourseName( string name )
   {
      courseName = name; // store the course name in the object
   } // end function setCourseName

```

Class definition is in a header file

Outline

fig03_09.cpp

(1 of 2)

Outline

```
   // function to get the course name
   string getCourseName()
   {
      return courseName; // return object's courseName
   } // end function getCourseName

   // display a welcome message to the GradeBook user
   void displayMessage()
   {
      // call getCourseName to get the courseName
      cout << "Welcome to the grade book for\n" << getCourseName()
         << "!" << endl;
   } // end function displayMessage
private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook
```

fig03_09.cpp

(2 of 2)

Outline

fig03_10.cpp

(1 of 1)

```
// Fig. 3.10: fig03_10.cpp
// Including class GradeBook from file GradeBook.h for use in main.
#include <iostream>
using std::cout;
using std::endl;

#include "GradeBook.h" // include definition of class GradeBook

// function main begins program execution
int main()
{
   // create two GradeBook objects
   GradeBook gradeBook1( "CS101 Introduct
   GradeBook gradeBook2( "CS102 Data Structures in C++" );

   // display initial value of courseName for each GradeBook
   cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
      << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
      << endl;
   return 0; // indicate successful termination
} // end main
```

Including the header file causes the class definition to be copied into the file

```
gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++
```

3.8 Placing a Class in a Separate File for Reusability (Cont.)

- `#include` preprocessor directive
  - Used to include header files
    - Instructs C++ preprocessor to replace directive with a copy of the contents of the specified file
  - Quotes indicate user-defined header files
    - Preprocessor first looks in current directory
      - If the file is not found, looks in C++ Standard Library directory
  - Angle brackets indicate C++ Standard Library
    - Preprocessor looks only in C++ Standard Library directory

3.8 Placing a Class in a Separate File for Reusability (Cont.)

- Creating objects
  - Compiler must know size of object
    - C++ objects typically contain only data members
    - Compiler creates one copy of class’s member functions
      - This copy is shared among all the class’s objects

Error-Prevention Tip 3.3

To ensure that the preprocessor can locate header files correctly, `#include` preprocessor directives should place the names of user-defined header files in quotes (e.g., `"GradeBook.h"`) and place the names of C++ Standard Library header files in angle brackets (e.g., `<iostream>`).

◀ ▶

3.9 Separating Interface from Implementation

- **Interface**
  - **Describes what services a class's clients can use and how to request those services**
    - **But does not reveal how the class carries out the services**
    - **A class definition that lists only member function names, return types and parameter types**
      - **Function prototypes**
  - **A class's interface consists of the class's `public` member functions (services)**
- **Separating interface from implementation**
  - **Client code should not break if implementation changes, as long as interface stays the same**

3.9 Separating Interface from Implementation (Cont.)

- Separating interface from implementation (Cont.)
  - Define member functions outside the class definition, in a separate source-code file
    - In source-code file for a class
      - Use binary scope resolution operator (`::`) to tie each member function to the class definition
    - Implementation details are hidden
      - Client code does not need to know the implementation
  - In header file for a class
    - Function prototypes describe the class's `public` interface

fig03_11.cpp

(1 of 1)

```cpp
// Fig. 3.11: GradeBook.h
// GradeBook class definition. This file presents GradeBook's public
// interface without revealing the implementations of GradeBook's member
// functions, which are defined in GradeBook.cpp.
#include <string> // class GradeBook uses C++ standard string class
using std::string;

// GradeBook class definition
class GradeBook
{
public:
   GradeBook( string ); // constructor that initializes courseName
   void setCourseName( string ); // function that sets the course name
   string getCourseName(); // function that gets the course name
   void displayMessage(); // function that displays a welcome message
private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook
```

Interface contains data members and member function prototypes

Common Programming Error 3.8

Forgetting the semicolon at the end of a function prototype is a syntax error.

Good Programming Practice 3.7

Although parameter names in function prototypes are optional (they are ignored by the compiler), many programmers use these names for documentation purposes.

◀ ▶

Error-Prevention Tip 3.4

Parameter names in a function prototype (which, again, are ignored by the compiler) can be misleading if wrong or confusing names are used. For this reason, many programmers create function prototypes by copying the first line of the corresponding function definitions (when the source code for the functions is available), then appending a semicolon to the end of each prototype.

Common Programming Error 3.9

When defining a class's member functions outside that class, omitting the class name and binary scope resolution operator (`::`) preceding the function names causes compilation errors.

Outline

3_12.cpp

(1 of 2)

```
// Fig. 3.12: GradeBook.cpp
// GradeBook member-function definitions. This file contains
// implementations of the member functions prototyped in GradeBook.h.
#include <iostream>
using std::cout;
using std::endl;

#include "GradeBook.h" // include definition of class GradeBook

// constructor initializes courseName with string supplied as argu
GradeBook::GradeBook( string name )
{
   setCourseName( name ); // call set function to initialize courseName
} // end GradeBook constructor

// function to set the course name
void GradeBook::setCourseName( string name )
{
   courseName = name; // store the course name in the object
} // end function setCourseName

```

`GradeBook` implementation is placed in a separate source-code file

Include the header file to access the class name **`GradeBook`**

Binary scope resolution operator ties a function to its class

```
// function to get the course name
string GradeBook::getCourseName()
{
   return courseName; // return object's courseName
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   // call getCourseName to get the courseName
   cout << "Welcome to the grade book for\n" << getCourseName()
      << "!" << endl;
} // end function displayMessage
```

Outline

fig03_12.cpp

(2 of 2)

Outline

```cpp
// Fig. 3.13: fig03_13.cpp
// GradeBook class demonstration after separating
// its interface from its implementation.
#include <iostream>
using std::cout;
using std::endl;

#include "GradeBook.h" // include definition of class GradeBook

// function main begins program execution
int main()
{
   // create two GradeBook objects
   GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
   GradeBook gradeBook2( "CS102 Data Structures in C++" );

   // display initial value of courseName for each GradeBook
   cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
      << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
      << endl;
   return 0; // indicate successful termination
} // end main
```

```
gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++
```

fig03_13.cpp

(1 of 1)

3.9 Separating Interface from Implementation (Cont.)

- The Compilation and Linking Process
  - Source-code file is compiled to create the class's object code (source-code file must `#include` header file)
    - Class implementation programmer only needs to provide header file and object code to client
  - Client must `#include` header file in their own code
    - So compiler can ensure that the `main` function creates and manipulates objects of the class correctly
  - To create executable application
    - Object code for client code must be linked with the object code for the class and the object code for any C++ Standard Library object code used in the application

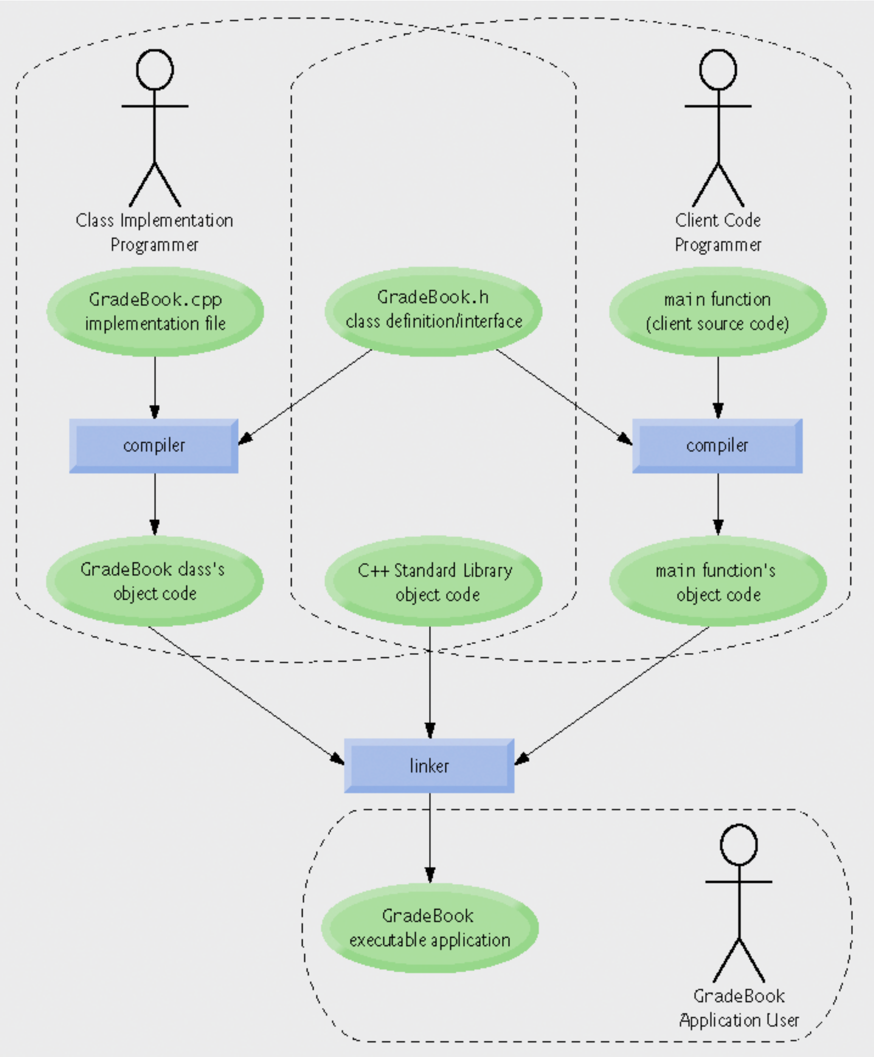

Class Implementation Programmer
Client Code Programmer
GradeBook.cpp implementation file
GradeBook.h class definition/interface
main function (client source code)
compiler
compiler
GradeBook class's object code
C++ Standard Library object code
main function's object code
linker
GradeBook executable application
GradeBook Application User

Fig.3.14 | Compilation and linking process that produces an executable application.

3.10 Validating Data with *set* Functions

- *set* functions can validate data
  - Known as validity checking
  - Keeps object in a consistent state
    - The data member contains a valid value
  - Can return values indicating that attempts were made to assign invalid data
- `string` member functions
  - `length` returns the number of characters in the `string`
  - `Substr` returns specified substring within the `string`

```
// Fig. 3.15: GradeBook.h
// GradeBook class definition presents the public interface of
// the class. Member-function definitions appear in GradeBook.cpp.
#include <string> // program uses C++ standard string class
using std::string;

// GradeBook class definition
class GradeBook
{
public:
   GradeBook( string ); // constructor that initializes a GradeBook object
   void setCourseName( string ); // function that sets the course name
   string getCourseName(); // function that gets the course name
   void displayMessage(); // function that displays a welcome message
private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook
```

Outline

fig03_15.cpp

(1 of 1)

Outline

```
// Fig. 3.16: GradeBook.cpp
// Implementations of the GradeBook member-function definitions.
// The setCourseName function performs validation.
#include <iostream>
using std::cout;
using std::endl;

#include "GradeBook.h" // include definition of class GradeBook

// constructor initializes courseName with string supplied as argument
GradeBook::GradeBook( string name )
{
   setCourseName( name ); // validate and store courseName
} // end GradeBook constructor

// function that sets the course name;
// ensures that the course name has at most 25 characters
void GradeBook::setCourseName( string name )
{
   if ( name.length() <= 25 ) // if name has 25 or fewer characters
      courseName = name; // store the course name in the object

```

fig03_16.cpp

(1 of 2)

Constructor calls *set* function to perform validity checking

set functions perform validity checking to keep **courseName** in a consistent state

```
   if ( name.length() > 25 ) // if name has more than 25 characters
   {
      // set courseName to first 25 characters of parameter name
      courseName = name.substr( 0, 25 ); // start at 0, length of 25

      cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
         << "Limiting courseName to first 25 characters.\n" << endl;
   } // end if
} // end function setCourseName

// function to get the course name
string GradeBook::getCourseName()
{
   return courseName; // return object's courseName
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   // call getCourseName to get the courseName
   cout << "Welcome to the grade book for\n" << getCourseName()
      << "!" << endl;
} // end function displayMessage
```

Outline

fig03_16.cpp

(2 of 2)

Outline

fig03_17.cpp

```
// Fig. 3.17: fig03_17.cpp
// Create and manipulate a GradeBook object; illustrate validation.
#include <iostream>
using std::cout;
using std::endl;

#include "GradeBook.h" // include definition of class Gr

// function main begins program execution
int main()
{
   // create two GradeBook objects;
   // initial course name of gradeBook1 is too long
   GradeBook gradeBook1( "CS101 Introduction to Programming in C++" );
   GradeBook gradeBook2( "CS102 C++ Data Structures" );

```

Constructor will call *set* function to perform validity checking

Outline

fig03_17.cpp

(2 of 2)

```
   // display each GradeBook's courseName
   cout << "gradeBook1's initial course name is: "
      << gradeBook1.getCourseName()
      << "\ngradeBook2's initial course name is: "
      << gradeBook2.getCourseName() << endl;

   // modify myGradeBook's courseName (with a valid-length string)
   gradeBook1.setCourseName( "CS101 C++ Programming" );

   // display each GradeBook's courseName
   cout << "\ngradeBook1's course name is: "
      << gradeBook1.getCourseName()
      << "\ngradeBook2's course name is: "
      << gradeBook2.getCourseName() << endl;
   return 0; // indicate successful termination
} // end main
```

Call *set* function to perform validity checking

```
Name "CS101 Introduction to Programming in C++" exceeds maximum length (25).
Limiting courseName to first 25 characters.

gradeBook1's initial course name is: CS101 Introduction to Pro
gradeBook2's initial course name is: CS102 C++ Data Structures

gradeBook1's course name is: CS101 C++ Programming
gradeBook2's course name is: CS102 C++ Data Structures
```

Software Engineering Observation 3.6

Making data members `private` and controlling access, especially write access, to those data members through `public` member functions helps ensure data integrity.

Error-Prevention Tip 3.5

The benefits of data integrity are not automatic simply because data members are made `private`—the programmer must provide appropriate validity checking and report the errors.

Software Engineering Observation 3.7

Member functions that *set* the values of `private` data should verify that the intended new values are proper; if they are not, the *set* functions should place the `private` data members into an appropriate state.

3.11 (Optional) Software Engineering Case Study: Identifying the Classes in the ATM Requirements Document

- **Identifying the classes in a system**
  - **Key nouns and noun phrases in requirements document**
    - **Some are attributes of other classes**
    - **Some do not correspond to parts of the system**
    - **Some are classes**
      - **To be represented by UML class diagrams**

Nouns and noun phrases in the requirements document		
bank	money / fund	account number
ATM	screen	PIN
user	keypad	bank database
customer	cash dispenser	balance inquiry
transaction	$20 bill / cash	withdrawal
account	deposit slot	deposit
balance	deposit envelope	

Fig.3.18 | **Nouns and noun phrases in the requirements document.**

3.11 (Optional) Software Engineering Case Study: Identifying the Classes in the ATM Requirements Document (Cont.)

- Modeling classes with UML class diagrams
  - Top compartment contains name of the class
  - Middle compartment contains attributes
  - Bottom compartment contains operations
  - An elided diagram
    - Suppress some class attributes and operations for readability
  - An association
    - Represented by a solid line that connects two classes
    - Association can be named
    - Numbers near end of each line are multiplicity values
    - Role name identifies the role an object plays in an association

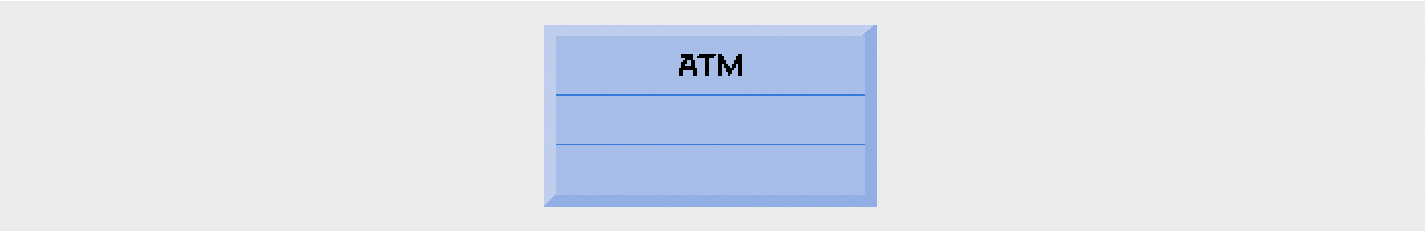


Fig.3.19 | **Representing a class in the UML using a class diagram.**

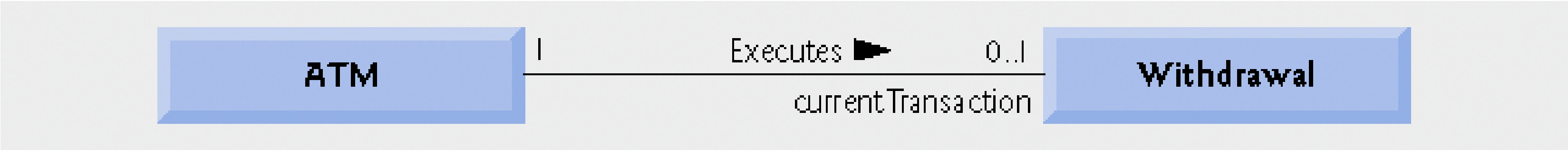


Fig.3.20 | Class diagram showing an association among classes.

Symbol	Meaning
0	None
1	One
m	An integer value
0..1	Zero or one
m, n	*m* or *n*
m..n	At least *m*, but not more than *n*
*	Any nonnegative integer (zero or more)
0..*	Zero or more (identical to *)
1..*	One or more

Fig.3.21 | **Multiplicity types.**

3.11 (Optional) Software Engineering Case Study: Identifying the Classes in the ATM Requirements Document (Cont.)

- **Composition relationship**
  - **Indicated by solid diamonds attached to association lines**
  - **Composition properties**
    - **Only one class can represent the whole**
    - **Parts only exist while whole exists, whole creates and destroys parts**
    - **A part may only belong to one whole at a time**
- **Hollow diamonds indicate aggregation**
  - **A weaker form of composition**
- **Types of associations**
  - **One-to-one**
  - **One-to-many**
  - **Many-to-one**

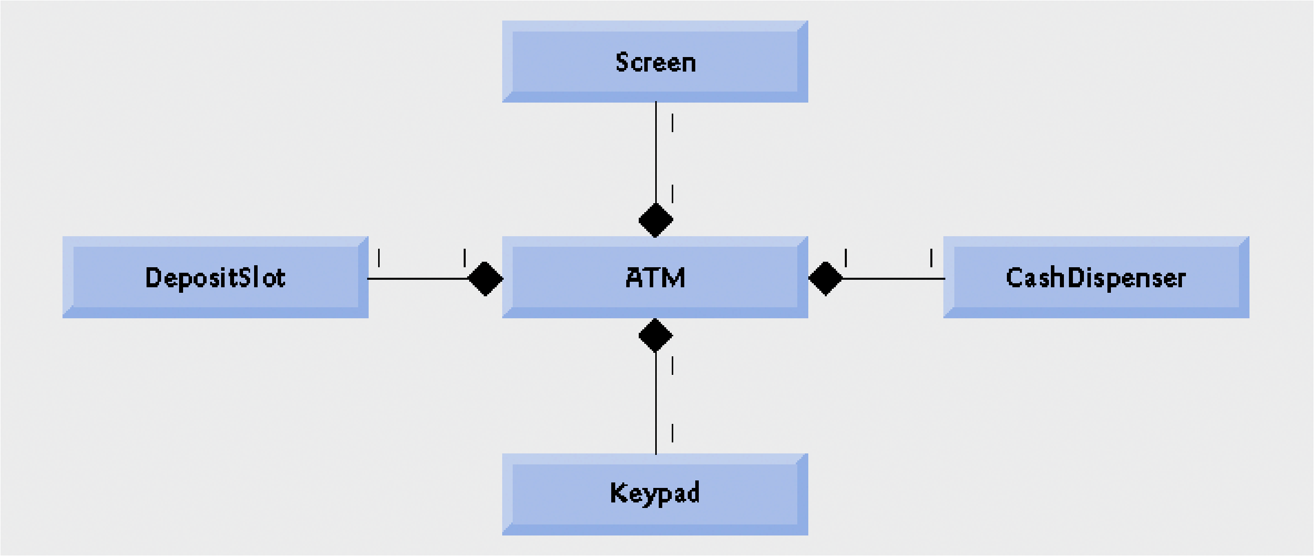


Fig.3.22 | Class diagram showing composition relationships.

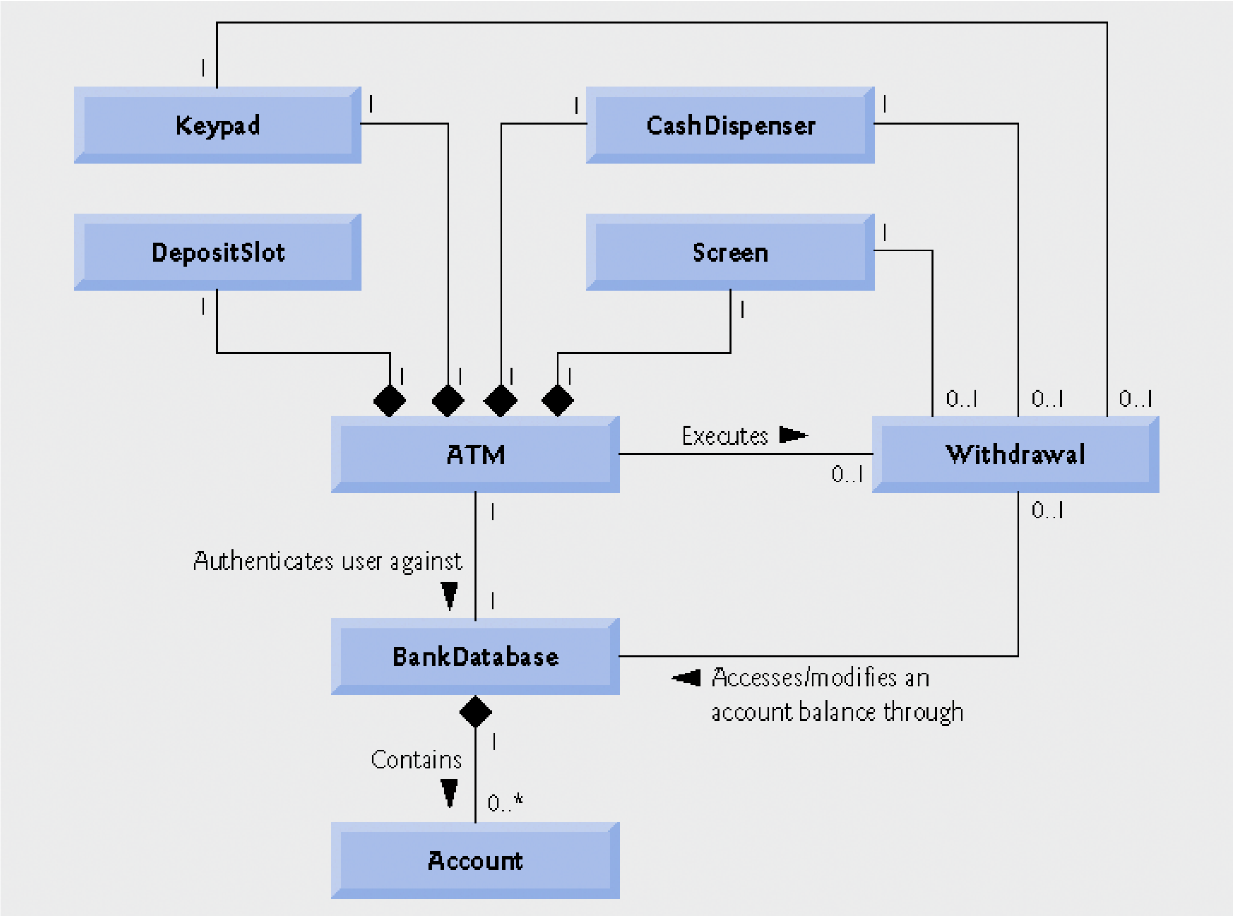


Fig.3.23 | **Class diagram for the ATM system model**

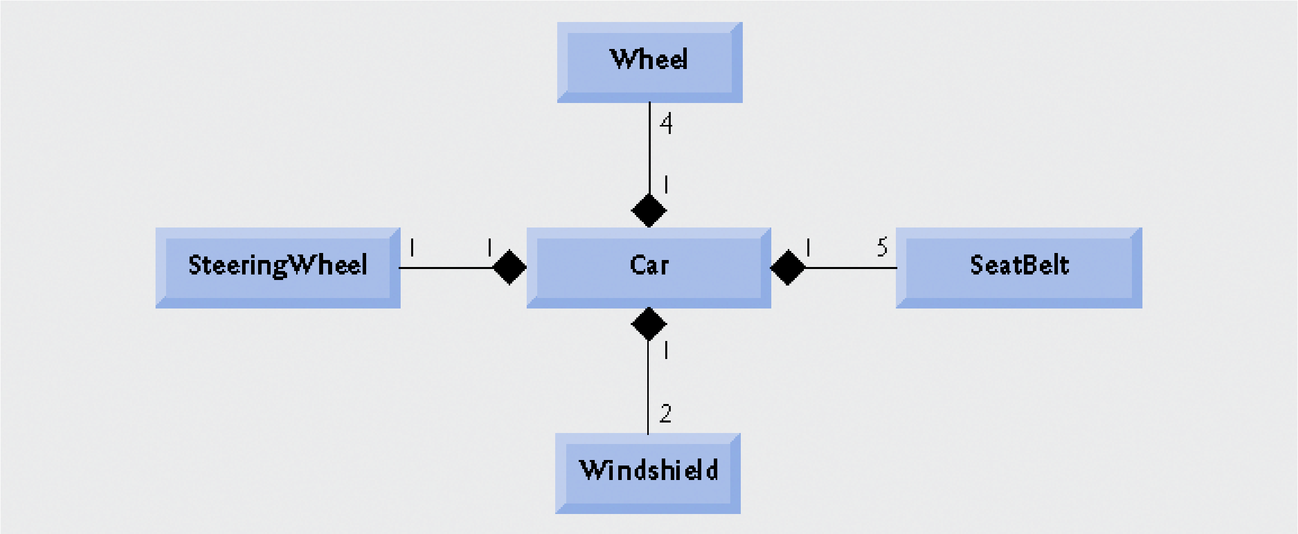


Fig.3.24 | Class diagram showing composition relationships of a class `Car`.

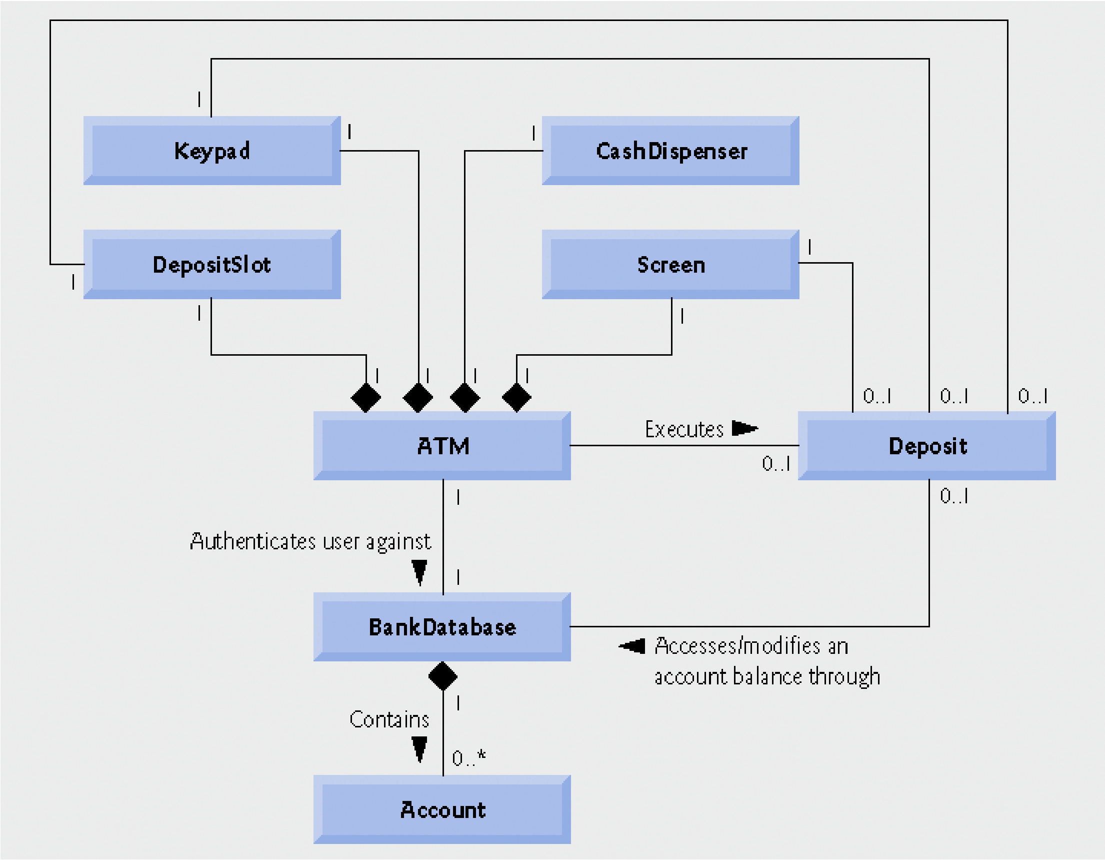


Fig.3.25 | Class diagram for the ATM system model including class `Deposit`.