Cape Peninsula
University of Technology

Software Design 2 (SDN260S)

Introduction to Object-Oriented Programming

H. Mataifa

Department of Electronic, Electrical and Computer Engineering

Object-Oriented Technology

- **Class**: a reusable software component, defined in terms of attributes and behaviours; Much like a blueprint for an engineering design, a **class** can be used to create **objects** that possess the attributes and behaviours defined in the class

- **Method**: performing a task in a (computer) program requires a method, which defines the program statements that actually perform the task

- An object has **attributes** that define its properties or characteristics (e.g. a car has a specific colour, number of doors, etc.). These attributes are defined as **instance variables** in the class definition. Instance variables are specific to an object, specified when the object is created

- Classes **encapsulate** (i.e. wrap) attributes and methods into objects – an object’s attributes and methods are intimately related. Objects may interact with one another, but they’re not normally allowed to know how other objects are implemented – implementation details are hidden within the objects themselves. Information hiding is crucial to good software engineering

- **Instantiation**: we need to create an object of a class before a program can perform the tasks that the class’ methods define. The process of doing so is called instantiation. An object is referred to as an instance of its class.

- **Method call**: can be thought of as a message passed to an object requesting it to perform a certain task by means of the method defined as part of its class

Object-Oriented Technology

- **Object-Oriented Analysis and Design (OOAD)**: developing large software projects requires following a detailed *analysis* process for determining the project's *requirements* (i.e. defining what the system is supposed to do) and developing a *design* that satisfies them (i.e. defining how the system should do it). This is usually an iterative process that involves review of the design against the requirements to ensure its correctness. If this process involves analysing and designing the system from an object-oriented point of view, it's called an *object-oriented analysis and design (OOAD)* process, and is implemented using an object-oriented programming language, such as **Java**.

- **Unified Modelling Language (UML)** : the most widely used graphical scheme for modelling object-oriented systems

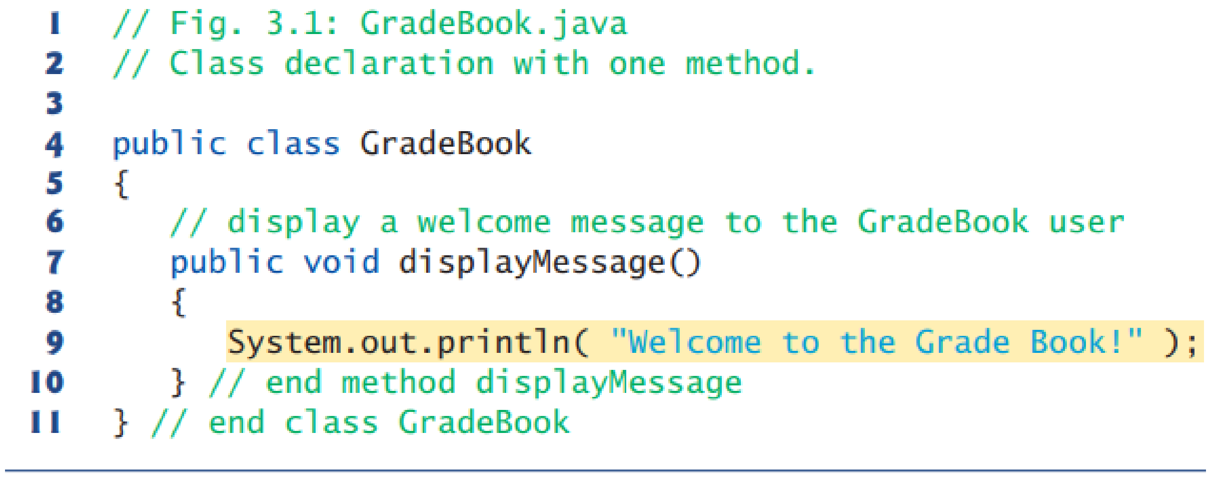
Declaring a class with a method and instantiating an object

- Class GradeBook contains method displayMessage that displays a message on the screen

- To be able to make use of the method defined in class GradeBook, we need to create an object of this class

- Notice the use of keyword public (referred to as an *access modifier*) in class GradeBook's declaration. This means the class can *directly interact with other classes*, and it needs to be saved in a file bearing its name, and having the file extension ".java" (i.e. GradeBook.java)

- Notice the components of the definition of method displayMessage. The *method header* begins with *access modifier* (public), followed by *return type* (void), then by *method name* (displayMessage), finally by *parameter list* in round brackets (empty in this case). The body of the method is delimited by curly brackets



```
// Fig. 3.1: GradeBook.java
// Class declaration with one method.

public class GradeBook
{
   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      System.out.println( "Welcome to the Grade Book!" );
   } // end method displayMessage
} // end class GradeBook
```

Fig. 3.1 | Class declaration with one method.

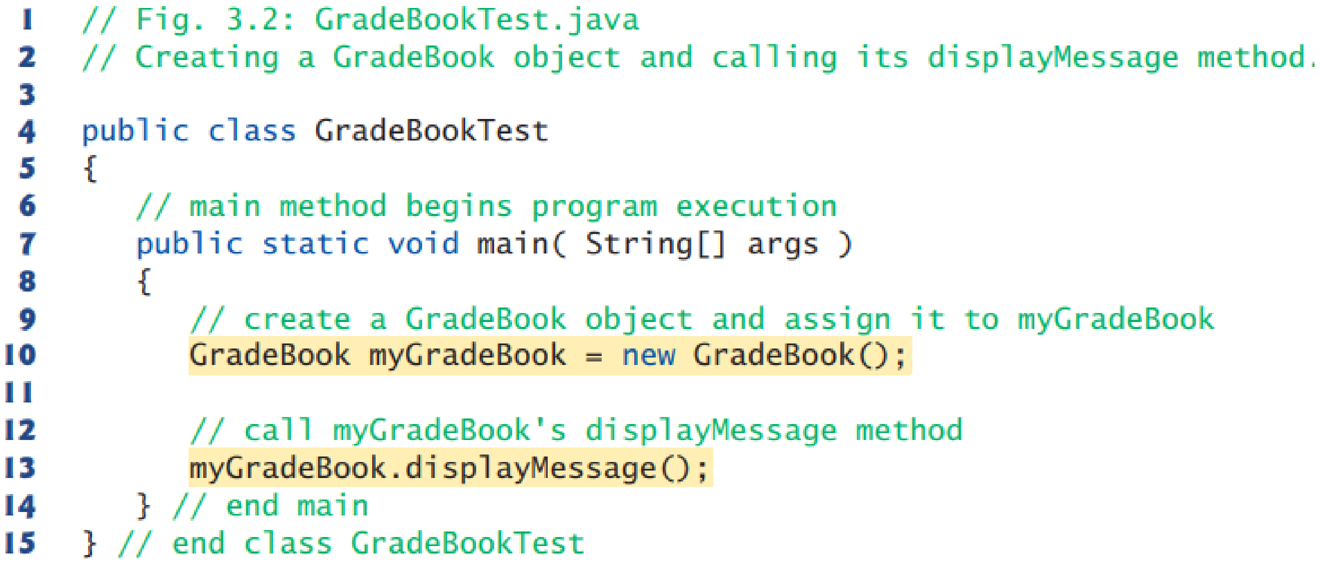
Declaring a class with a method and instantiating an object

- Class GradeBookTest is used as the Java application program that we use to test the operation of class GradeBook. It can also be referred to as a *driver class*. It contains method main that is required by the **JVM** to run any Java application

- Note the *method header* for main. The additional component it has (compared with method displayMessage defined earlier) is the keyword static.

- A *static method* is special, because we can call it without first creating an object of the class in which the method is declared

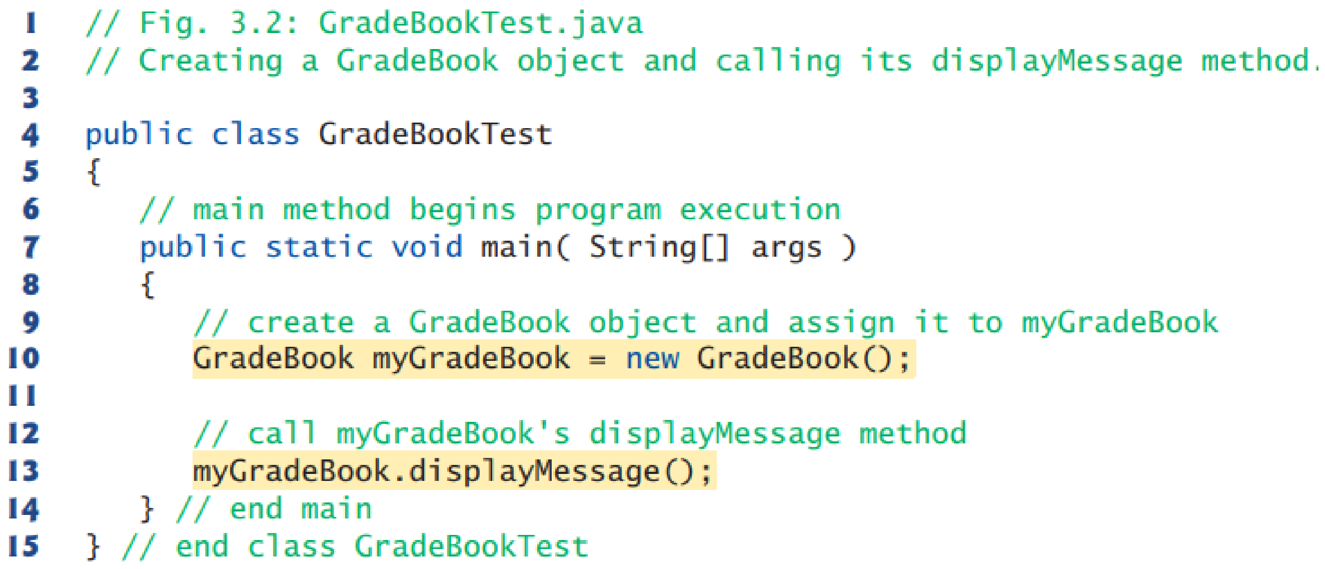
- In order to call a method declared in another class, we need to instantiate an object of that class. This is done in line 10 of Fig. 3.2



```
 1  // Fig. 3.2: GradeBookTest.java
 2  // Creating a GradeBook object and calling its displayMessage method.
 3
 4  public class GradeBookTest
 5  {
 6     // main method begins program execution
 7     public static void main( String[] args )
 8     {
 9        // create a GradeBook object and assign it to myGradeBook
10        GradeBook myGradeBook = new GradeBook();
11
12        // call myGradeBook's displayMessage method
13        myGradeBook.displayMessage();
14     } // end main
15  } // end class GradeBookTest
```

Declaring a class with a method and instantiating an object

- Line 10 makes use of a constructor, a *special method* used to instantiate an object of a class
- Line 13 calls method displayMessage using an instance of class (i.e. object of type) GradeBook. This is referred to as *method invocation*(*object name* followed by *dot* followed by *method name* and *parameter list*)



```
 1  // Fig. 3.2: GradeBookTest.java
 2  // Creating a GradeBook object and calling its displayMessage method.
 3
 4  public class GradeBookTest
 5  {
 6     // main method begins program execution
 7     public static void main( String[] args )
 8     {
 9        // create a GradeBook object and assign it to myGradeBook
10        GradeBook myGradeBook = new GradeBook();
11
12        // call myGradeBook's displayMessage method
13        myGradeBook.displayMessage();
14     } // end main
15  } // end class GradeBookTest
```

UML diagram for class GradeBook

- UML (*Unified Modelling Language*) is the most widely used graphical scheme for modelling object-oriented systems. Fig. 3.3 shows a **UML** class diagram for class GradeBook. It contains three compartments: top one for the class name, middle one for the class attributes (also, *instance variables*), and bottom one for the class methods (or *class operations*)

- Class attributes and methods are usually either public or private. **UML** indicates this by preceding the attribute/method name by a plus/minus sign for public/private access respectively

- Class GradeBook has no attributes, thus middle compartment is empty

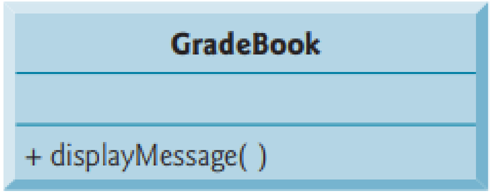


Fig. 3.3 | UML class diagram indicating that class `GradeBook` has a `public` `displayMessage` operation.

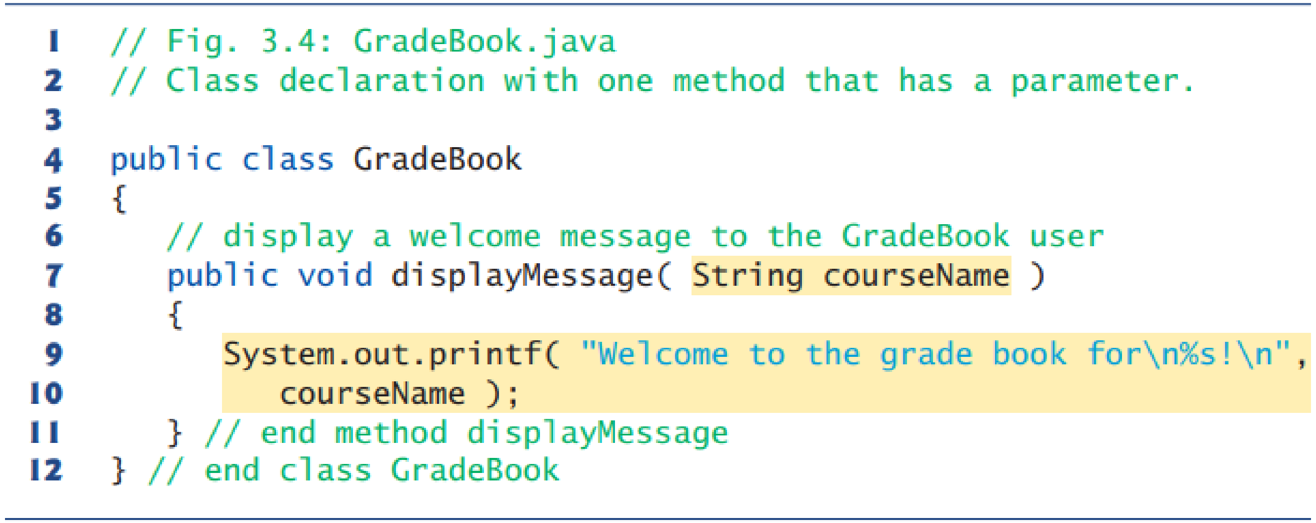
Declaring a method with a parameter

- A method may require additional information from the calling environment to perform its task(s). This additional information should be specified in the method definition as parameters

- Parameters are defined in a comma-separated list enclosed within parentheses following the method name. Each parameter must specify the data type and variable name

- When a method is defined with parameters, the call to the method (i.e. method invocation) must supply an appropriate argument corresponding to each parameter in the method definition

- Class GradeBook's method displayMessage is defined with parameter courseName of type String

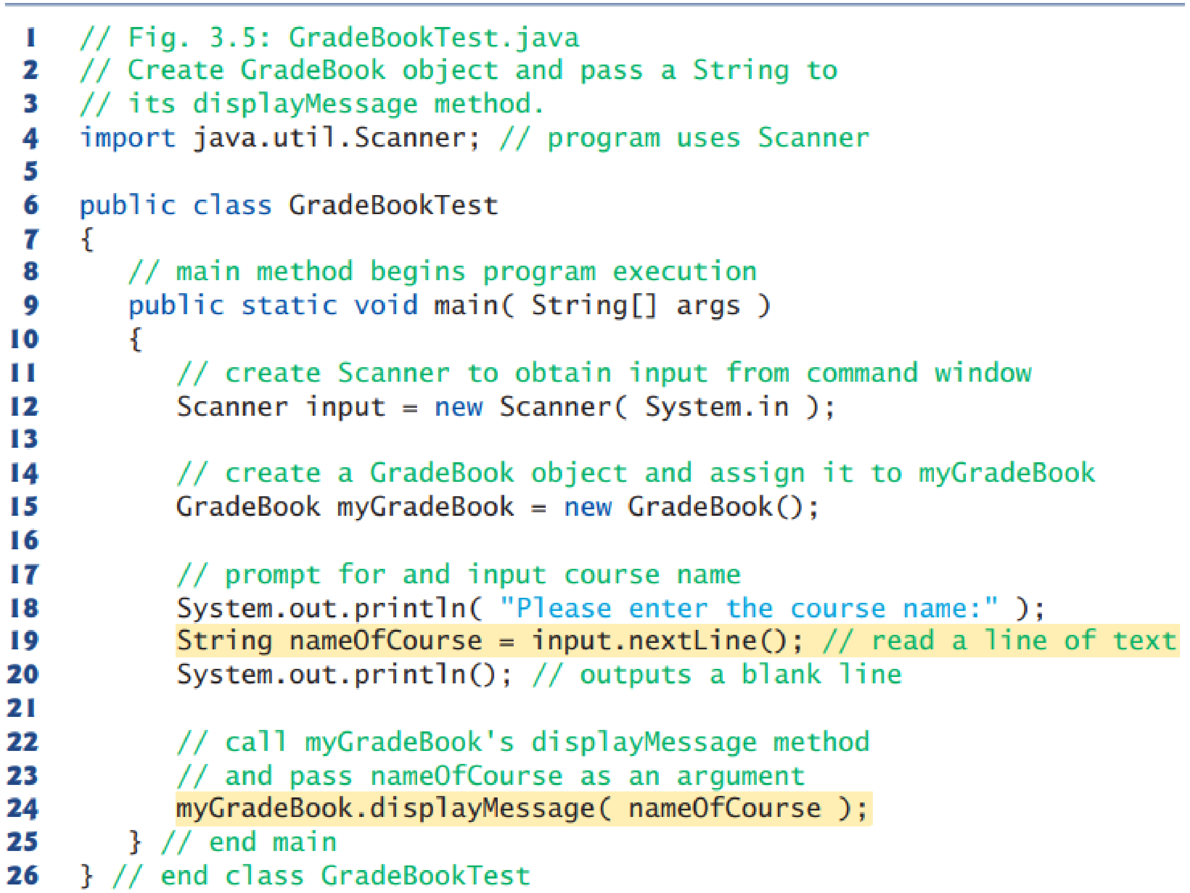


```java
// Fig. 3.4: GradeBook.java
// Class declaration with one method that has a parameter.

public class GradeBook
{
   // display a welcome message to the GradeBook user
   public void displayMessage( String courseName )
   {
      System.out.printf( "Welcome to the grade book for\n%s!\n",
         courseName );
   } // end method displayMessage
} // end class GradeBook
```

Fig. 3.4 | Class declaration with one method that has a parameter.

Declaring a method with a parameter

- Class GradeBookTest is used as the Java application program to test the modified class GradeBook
- The programmer must supply an appropriate argument to method displayMessage when calling it (line 24)
- The number of arguments in a method call must match the number of parameters in the parameter list of the method declaration, and the argument type must be "consistent" with the corresponding parameter type



```
 1  // Fig. 3.5: GradeBookTest.java
 2  // Create GradeBook object and pass a String to
 3  // its displayMessage method.
 4  import java.util.Scanner; // program uses Scanner
 5
 6  public class GradeBookTest
 7  {
 8     // main method begins program execution
 9     public static void main( String[] args )
10     {
11        // create Scanner to obtain input from command window
12        Scanner input = new Scanner( System.in );
13
14        // create a GradeBook object and assign it to myGradeBook
15        GradeBook myGradeBook = new GradeBook();
16
17        // prompt for and input course name
18        System.out.println( "Please enter the course name:" );
19        String nameOfCourse = input.nextLine(); // read a line of text
20        System.out.println(); // outputs a blank line
21
22        // call myGradeBook's displayMessage method
23        // and pass nameOfCourse as an argument
24        myGradeBook.displayMessage( nameOfCourse );
25     } // end main
26  } // end class GradeBookTest
```

Updated UML class diagram for GradeBook

- **UML** class diagram for the modified class differs from the original one only in terms of method displayMessage having a parameter

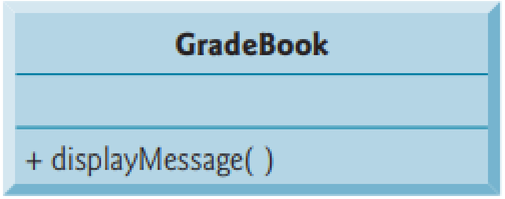


Fig. 3.3 | UML class diagram indicating that class `GradeBook` has a `public` `displayMessage` operation.

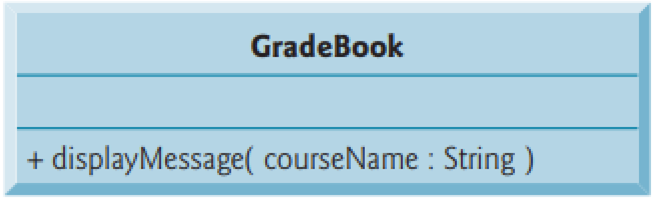


Fig. 3.6 | UML class diagram indicating that class `GradeBook` has a `displayMessage` operation with a `courseName` parameter of UML type `String`.

Instance variables, set methods and get methods

- Local variables are variables that are declared within the body of a method (i.e. they are local to that method, and can only be used within that method; they are lost once the method terminates)

- Class attributes are declared as variables in a class declaration, and outside of the bodies of the class' method declarations. When each object of a class maintains its own copy of a class attribute, that attribute is referred to as an instance variable

- Class GradeBook is redefined so that it has a class attribute courseName; every instance (i.e. object of type) GradeBook will now have courseName as an instance variable, which can be manipulated at any time during the program execution

- Whenever a class has instance variables, it is customary to provide methods that will be used to modify or access the instance variables; these are referred to as set and get methods

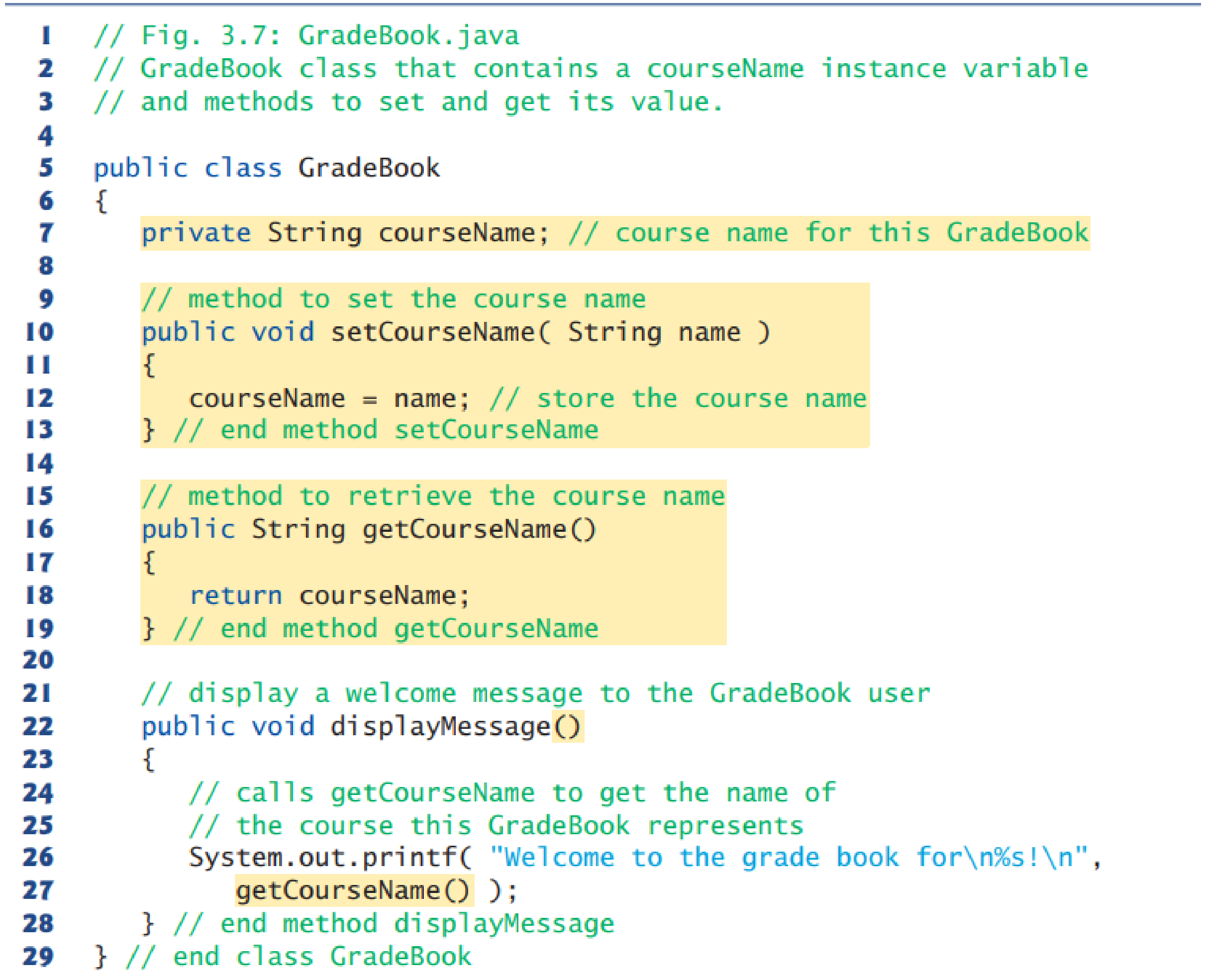
Instance variables, set methods and get methods

- Modified class GradeBook has two additional methods: setCourseName (to assign a value to instance variable courseName) and getCourseName (to obtain the value stored in instance variable courseName)

Instance variable courseName is declared private to restrict access to it

This is referred to as information hiding (or encapsulation), and is considered good programming practice

The only way to access instance variable courseName outside of class GradeBook is via (public) methods setCourseName and getCourseName

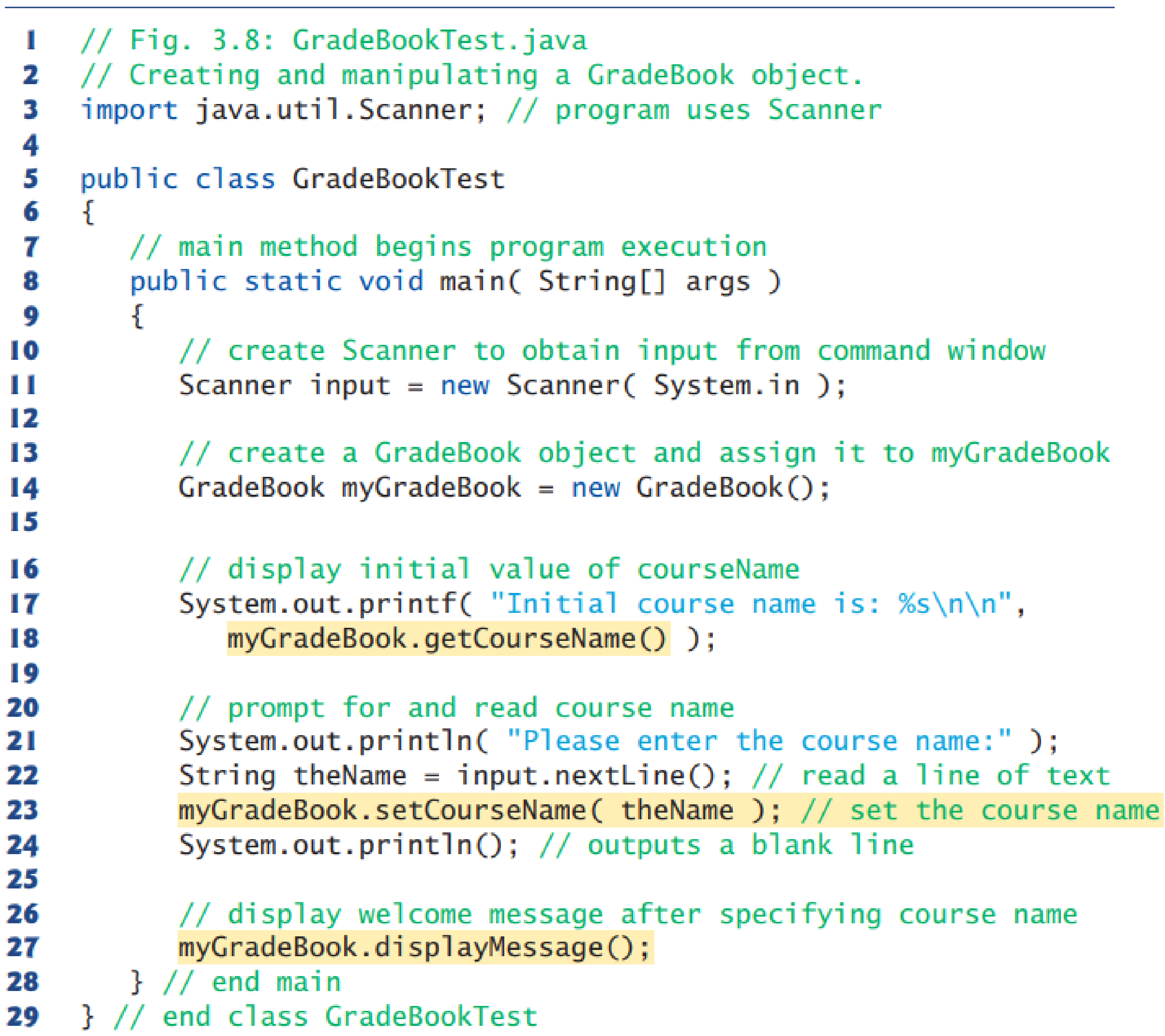


```java
// Fig. 3.7: GradeBook.java
// GradeBook class that contains a courseName instance variable
// and methods to set and get its value.

public class GradeBook
{
   private String courseName; // course name for this GradeBook

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // calls getCourseName to get the name of
      // the course this GradeBook represents
      System.out.printf( "Welcome to the grade book for\n%s!\n",
         getCourseName() );
   } // end method displayMessage
} // end class GradeBook
```

Fig. 3.7 | GradeBook class that contains a courseName instance variable and methods to set and get its value.

Instance variables, set methods and get methods

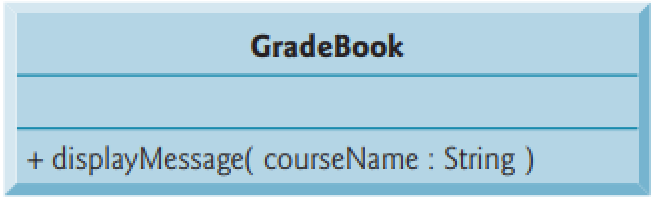
- Class GradeBookTest is used to test the modified GradeBook class



```java
// Fig. 3.8: GradeBookTest.java
// Creating and manipulating a GradeBook object.
import java.util.Scanner; // program uses Scanner

public class GradeBookTest
{
   // main method begins program execution
   public static void main( String[] args )
   {
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      // create a GradeBook object and assign it to myGradeBook
      GradeBook myGradeBook = new GradeBook();

      // display initial value of courseName
      System.out.printf( "Initial course name is: %s\n\n",
         myGradeBook.getCourseName() );

      // prompt for and read course name
      System.out.println( "Please enter the course name:" );
      String theName = input.nextLine(); // read a line of text
      myGradeBook.setCourseName( theName ); // set the course name
      System.out.println(); // outputs a blank line

      // display welcome message after specifying course name
      myGradeBook.displayMessage();
   } // end main
} // end class GradeBookTest
```

Updated UML class diagram for GradeBook

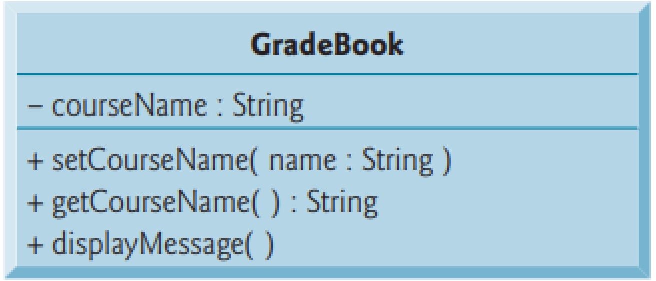
- Updated **UML** class diagram now contains an instance variable and two additional methods



GradeBook

+ displayMessage(courseName : String)

Fig. 3.6 | UML class diagram indicating that class GradeBook has a displayMessage operation with a courseName parameter of UML type String.



GradeBook

– courseName : String

+ setCourseName(name : String)
+ getCourseName( ) : String
+ displayMessage( )

Fig. 3.9 | UML class diagram indicating that class GradeBook has a private courseName attribute of UML type String and three public operations—setCourseName (with a name parameter of UML type String), getCourseName (which returns UML type String) and displayMessage.

Initializing objects with constructors

- Java requires each class to have a constructor, a special method that is used to instantiate (i.e. create an instance of) the class

- Keyword new is used when instantiating an object of a class by means of the class' constructor; this amounts to requesting memory to be allocated for storing the instantiated object

- A constructor has the same name as the class, and has no return type

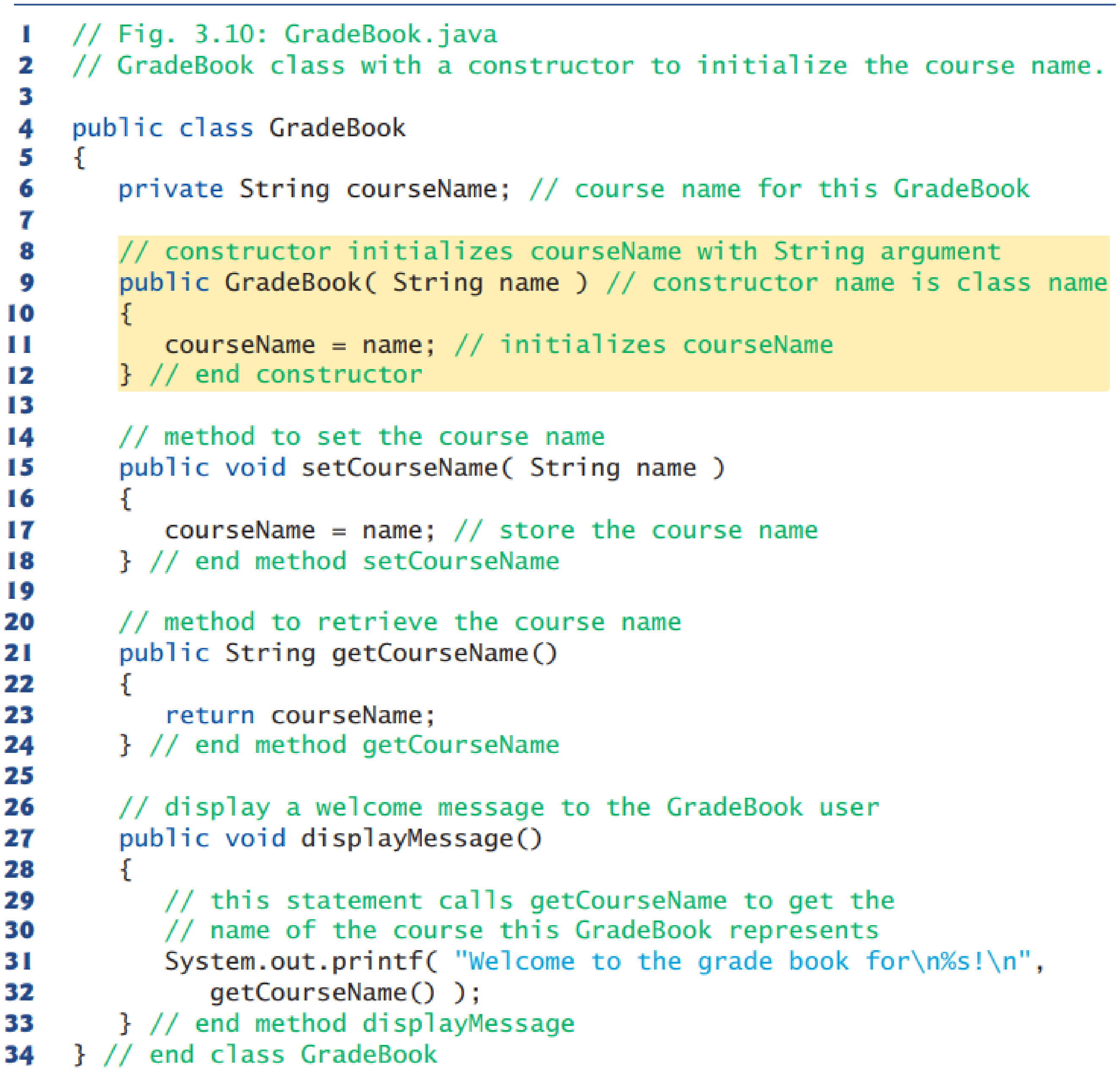
- By default, the compiler provides a default constructor with no parameters, which can be used to instantiate an object when no explicit constructor has been defined for the class.

- When a default constructor is used, instance variables are initialized to their default values. To be able to initialize instance variables to custom values when instantiating an object, a constructor can be (explicitly) defined for the class that initializes the object appropriately

- Constructors are normally declared public (since they have to be used by other classes to instantiate objects)

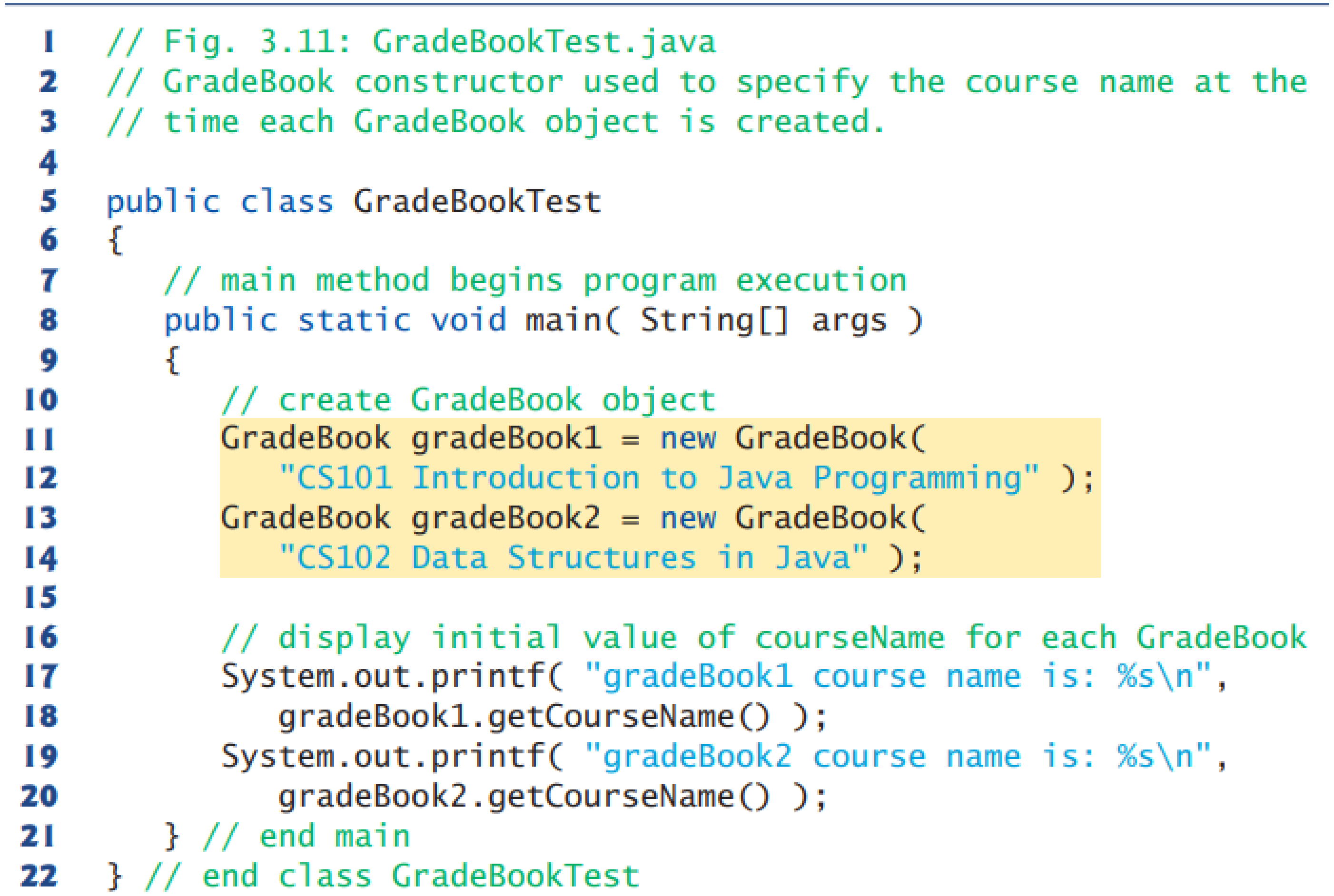
- Once you define a constructor for a class, the compiler no longer supplies the default constructor

Initializing objects with constructors



```java
// Fig. 3.10: GradeBook.java
// GradeBook class with a constructor to initialize the course name.

public class GradeBook
{
   private String courseName; // course name for this GradeBook

   // constructor initializes courseName with String argument
   public GradeBook( String name ) // constructor name is class name
   {
      courseName = name; // initializes courseName
   } // end constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // this statement calls getCourseName to get the
      // name of the course this GradeBook represents
      System.out.printf( "Welcome to the grade book for\n%s!\n",
         getCourseName() );
   } // end method displayMessage
} // end class GradeBook
```

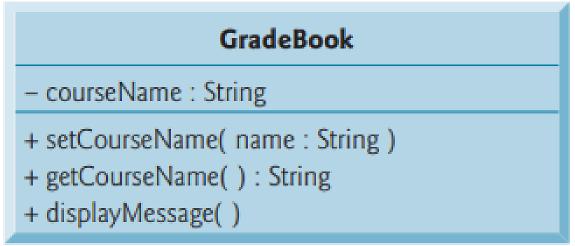
Initializing objects with constructors



```java
// Fig. 3.11: GradeBookTest.java
// GradeBook constructor used to specify the course name at the
// time each GradeBook object is created.

public class GradeBookTest
{
   // main method begins program execution
   public static void main( String[] args )
   {
      // create GradeBook object
      GradeBook gradeBook1 = new GradeBook(
         "CS101 Introduction to Java Programming" );
      GradeBook gradeBook2 = new GradeBook(
         "CS102 Data Structures in Java" );

      // display initial value of courseName for each GradeBook
      System.out.printf( "gradeBook1 course name is: %s\n",
         gradeBook1.getCourseName() );
      System.out.printf( "gradeBook2 course name is: %s\n",
         gradeBook2.getCourseName() );
   } // end main
} // end class GradeBookTest
```

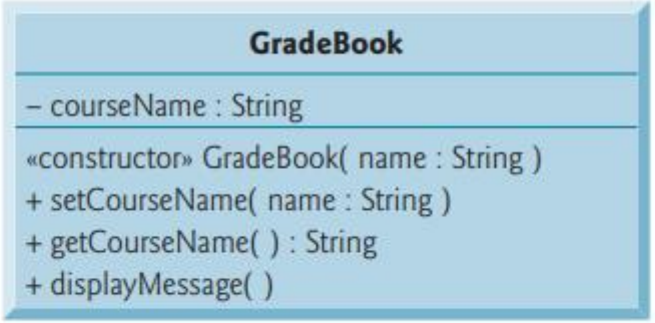
Updated UML class diagram for GradeBook

- Updated **UML** class diagram now contains an instance variable and two additional methods



```
GradeBook
– courseName : String
+ setCourseName( name : String )
+ getCourseName( ) : String
+ displayMessage( )
```

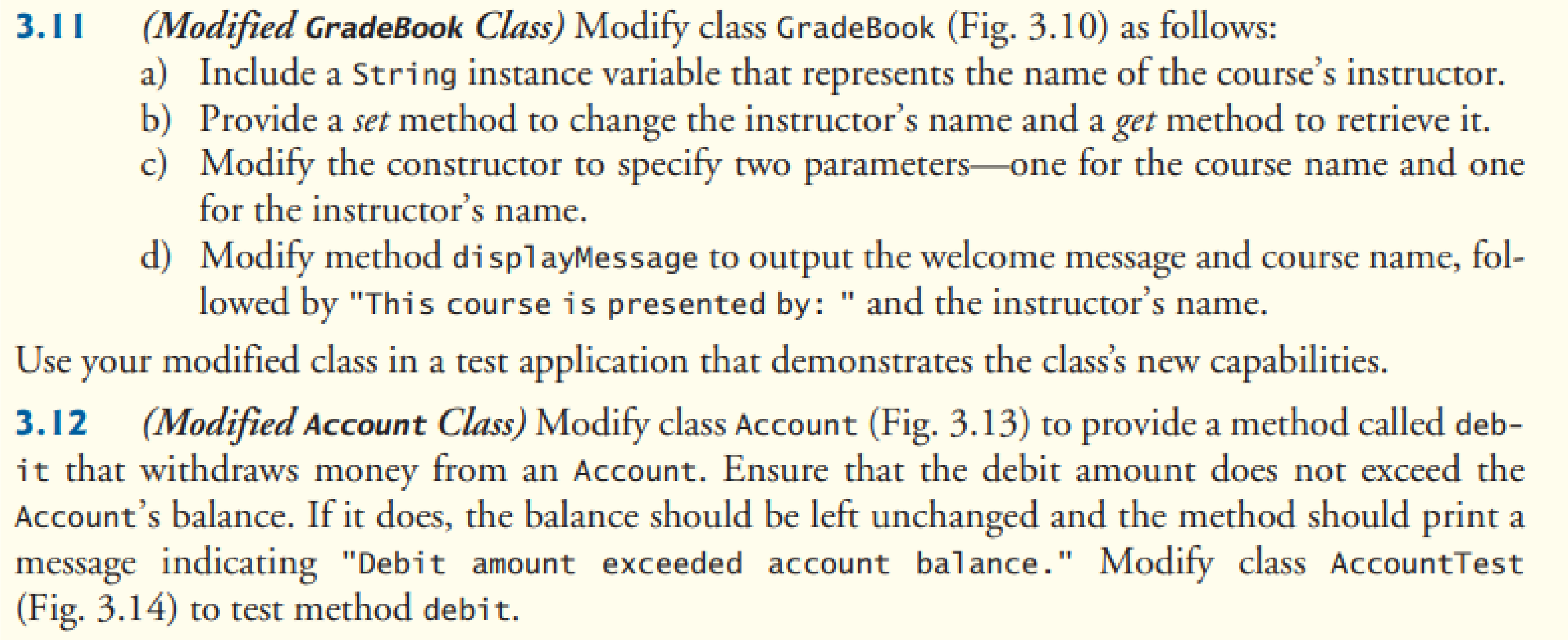
Fig. 3.9 | UML class diagram indicating that class GradeBook has a private courseName attribute of UML type String and three public operations—setCourseName (with a name parameter of UML type String), getCourseName (which returns UML type String) and displayMessage.



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

Fig. 3.12 | UML class diagram indicating that class GradeBook has a constructor that has a name parameter of UML type String.

Exercises



3.11 *(Modified* **`GradeBook`** *Class)* Modify class `GradeBook` (Fig. 3.10) as follows:

a) Include a `String` instance variable that represents the name of the course's instructor.

b) Provide a *set* method to change the instructor's name and a *get* method to retrieve it.

c) Modify the constructor to specify two parameters—one for the course name and one for the instructor's name.

d) Modify method `displayMessage` to output the welcome message and course name, followed by `"This course is presented by: "` and the instructor's name.

Use your modified class in a test application that demonstrates the class's new capabilities.

3.12 *(Modified* **`Account`** *Class)* Modify class `Account` (Fig. 3.13) to provide a method called `debit` that withdraws money from an `Account`. Ensure that the debit amount does not exceed the `Account`'s balance. If it does, the balance should be left unchanged and the method should print a message indicating `"Debit amount exceeded account balance."` Modify class `AccountTest` (Fig. 3.14) to test method `debit`.