2a

Introduction to Classes and Objects

OBJECTIVES

In this lecture you will learn:

- What classes, objects, methods and instance variables are.
- How to declare a class and use it to create an object.
- How to implement a class’s behaviors as methods.
- How to implement a class’s attributes as instance variables and properties.
- How to call an object’s methods to make them perform their tasks.

OBJECTIVES

- The differences between instance variables of a class and local variables of a method.
- How to use a constructor to ensure that an object’s data is initialized when the object is created.
- The differences between value types and reference types.

Outline

4.1 Introduction

4.2 Classes, Objects, Methods, Properties and Instance Variables

4.3 Declaring a Class with a Method and Instantiating an Object of a Class

4.4 Declaring a Method with a Parameter

Statements

4.5 Instance Variables and Properties

4.6 UML Class Diagram with a Property

4.7 Software Engineering with Properties and `set` and `get` Accessors

Outline

4.8 Auto-implemented Properties

4.9 Value Types vs. Reference Types

4.10 Initializing Objects with Constructors

4.11 Floating-Point Numbers and Type `decimal`

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

◀ ▶

4.2 Classes, Objects, Methods, Properties and Instance Variables

- A car begins as engineering drawings, similar to the blueprints used to design a house.
- An accelerator pedal “hides” the complex mechanisms that actually make the car go faster.
- Before you can drive a car, it must be built from the engineering drawings that describe it.

4.2 Classes, Objects, Methods, Properties and Instance Variables (Cont.)

- A **method** describes the internal mechanisms that actually perform its tasks.
- A **class** is used to house a method, just as a car's drawings house the design of an accelerator pedal.
- A class that represents a bank account might contain one method to deposit money in an account, another to withdraw money from an account and a third to inquire what the current account balance is.

4.2 Classes, Objects, Methods, Properties and Instance Variables (Cont.)

- Just as someone has to build a car from its engineering drawings before you can actually drive it, you must build an **object** of a class before you can perform the tasks the class describes.
- You send **messages** to an object by making **method call**s.

4.2 Classes, Objects, Methods, Properties and Instance Variables (Cont.)

- A car also has many **attributes**, such as its color, the number of doors, the amount of gas in its tank, its current speed and its total miles driven.
- These attributes are represented in its engineering diagrams, but every car maintains its own attributes.
- Attributes are specified by the class's **instance variables**.

4.2 Classes, Objects, Methods, Properties and Instance Variables (Cont.)

- Attributes are not necessarily accessible directly.
- Customers talk to a bank teller or check personalized online bank accounts to obtain their account balance.
- Similarly, you can use **`get` accessors** and **`set` accessors** to manipulate attributes.

- Select **File > New Project...** and create a `GradeBook` **Console Application**.
- The `GradeBook` **class declaration** (Fig. 4.1) contains a `DisplayMessage` method that displays a message on the screen.

`GradeBook.cs`

```
1  // Fig. 4.1: GradeBook.cs
2  // Class declaration with one method.
3  using System;
4
5  public class GradeBook
6  {
7     // display a welcome message to the GradeBook user
8     public void DisplayMessage()
9     {
10       Console.WriteLine( "Welcome to the Grade Book!" );
11    } // end method DisplayMessage
12 } // end class GradeBook
```

Line 8 is commonly referred to as the **method header**.

Fig. 4.1 | Class declaration with one method.

4.3 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- Keyword `public` is an **access modifier**.
  - Access modifiers determine the accessibility of properties and methods.
- The class's body is enclosed in a pair of left and right braces (`{` and `}`).

4.3 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- The method declaration begins with `public` to indicate that the method can be called from outside the class declaration’s body.
- Keyword `void`—known as the method’s return type—indicates that this method will not return information to its calling method.
- When a method specifies a return type other than `void`, the method returns a result to its calling method.

```
int result = Square( 2 );
```

- The body of a method contains statement(s) that perform the method’s task.

4.3 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- To add a class, right click the project name in the **Solution Explorer** and select **Add > New Item…**.
- In the **Add New Item** dialog, select **Code File** and enter the name of your new file.

- The `GradeBookTest` class declaration (Fig. 4.2) contains the `Main` method that controls our application’s execution.

GradeBookTest.cs

```
1  // Fig. 4.2: GradeBookTest.cs
2  // Create a GradeBook object and call its DisplayMessage method.
3  public class GradeBookTest
4  {
5     // Main method begins program execution
6     public static void Main( string[] args )
7     {
8        // create a GradeBook object and assign it to myGradeBook
9        GradeBook myGradeBook = new GradeBook();
10
11       // call myGradeBook's DisplayMessage method
12       myGradeBook.DisplayMessage();
13    } // end Main
14 } // end class GradeBookTest
```

Object creation expression (constructor).

Using the object created in line 9.

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

Fig. 4.2 | Create a `GradeBook` object and call its `DisplayMessage` method.

4.3 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- Any class that contains a `Main` method can be used to execute an application.
- A `static` method can be called without creating an object of the class.

4.3 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- Figure 4.3 presents a **UML class diagram** for class `GradeBook`.
- Classes are modeled as a rectangle with three compartments.
  - The top compartment contains the name of the class.
  - The middle compartment contains the class's attributes.
  - The bottom compartment contains the class's operations.
- The plus sign (+) indicates that `DisplayMessage` is a public operation.

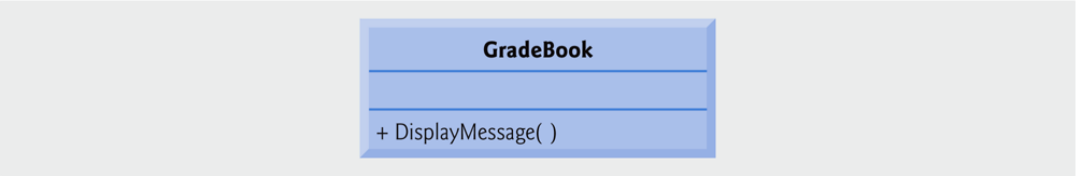


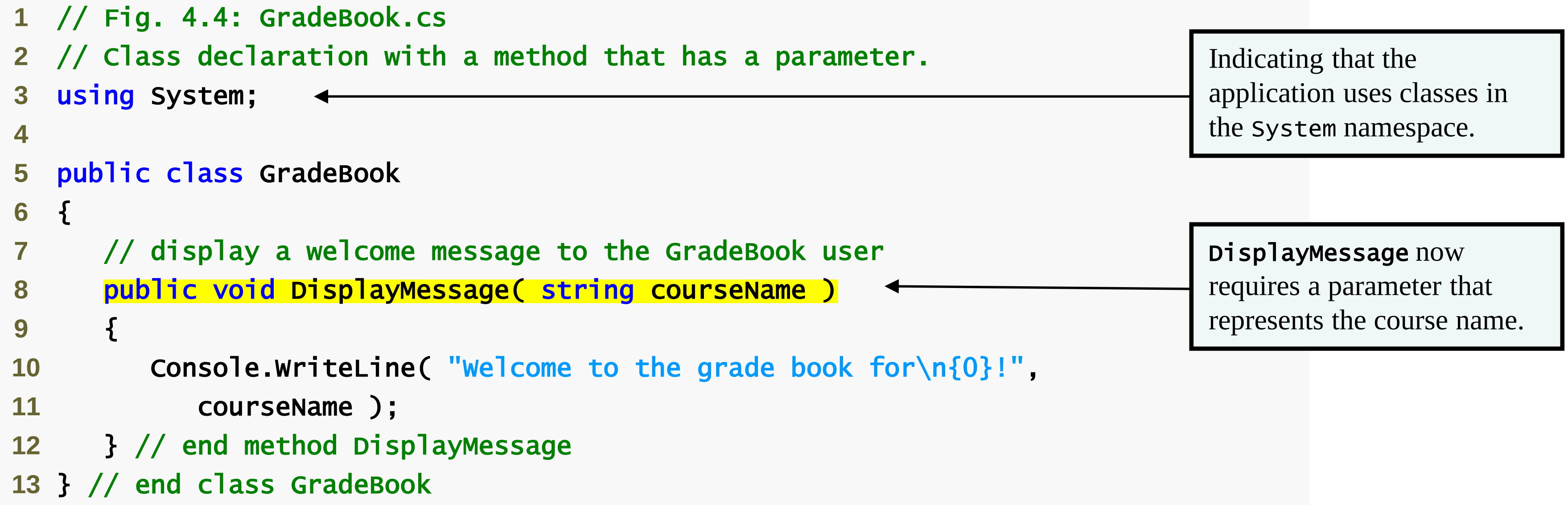
Fig. 4.3 | UML class diagram indicating that class `GradeBook` has a `public` `DisplayMessage` operation.

4.4 Declaring a Method with a Parameter

- A method can specify parameters, additional information required to perform its task.
- A method call supplies values—called arguments—for each of the method’s parameters.
- For example, the `Console.WriteLine` method requires an argument that specifies the data to be displayed in a console window.

- Class `GradeBook` (Fig. 4.4) with a `DisplayMessage` method that displays the course name as part of the welcome message.

`GradeBook.cs`



```
// Fig. 4.4: GradeBook.cs
// Class declaration with a method that has a parameter.
using System;

public class GradeBook
{
   // display a welcome message to the GradeBook user
   public void DisplayMessage( string courseName )
   {
      Console.WriteLine( "Welcome to the grade book for\n{0}!",
         courseName );
   } // end method DisplayMessage
} // end class GradeBook
```

Indicating that the application uses classes in the `System` namespace.

`DisplayMessage` now requires a parameter that represents the course name.

Fig. 4.4 | Class declaration with a method that has a parameter.

- The new class is used from the `Main` method of class `GradeBookTest` (Fig. 4.5).

GradeBookTest.cs

(1 of 2)

```
// Fig. 4.5: GradeBookTest.cs
// Create a GradeBook object and pass a string to
// its DisplayMessage method.
using System;

public class GradeBookTest
{
   // Main method begins program execution
   public static void Main( string[] args )
   {
      // create a GradeBook object and assign it to myGradeBook
      GradeBook myGradeBook = new GradeBook();

      // prompt for and input course name
      Console.WriteLine( "Please enter the course name:" );
      string nameOfCourse = Console.ReadLine(); // read a line of text
      Console.WriteLine(); // output a blank line
```

Creating an object of class `GradeBook` and assigns it to variable `myGradeBook`.

Prompting the user to enter a course name.

Reading the name from the user.

Fig. 4.5 | Create `GradeBook` object and pass a string to its `DisplayMessage` method. (Part 1 of 2).

GradeBookTest.cs

(2 of 2)

```

      // call myGradeBook's DisplayMessage method
      // and pass nameOfCourse as an argument
      myGradeBook.DisplayMessage( nameOfCourse );
   } // end Main
} // end class GradeBookTest
```

Calling `myGradeBook`’s `DisplayMessage` method and passing `nameOfCourse` to the method.

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

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

Fig. 4.5 | Create `GradeBook` object and pass a string to its `DisplayMessage` method. (Part 2 of 2).

4.4 Declaring a Method with a Parameter (Cont.)

Software Engineering Observation 4.1

Normally, objects are created with new. One exception is a string literal that is contained in quotes, such as `"hello"`*.* `String` *literals are references to string objects that are implicitly created by C#.*

- The method's **parameter list** is located in the parentheses that follow the method name.
- Empty parentheses indicate that a method does not require any parameters.
- The argument value in the call is assigned to the corresponding parameter in the method header.

4.4 Declaring a Method with a Parameter (Cont.)

Common Programming Error 4.1

A compilation error occurs if the number of arguments in a method call does not match the number of parameters in the method declaration.

Common Programming Error 4.2

A compilation error occurs if the types of the arguments in a method call are not consistent with the types of the corresponding parameters in the method declaration.

4.4 Declaring a Method with a Parameter (Cont.)

- The UML class diagram of Fig. 4.6 models class `GradeBook`.
- The UML models `DisplayMessage`'s parameter by listing the parameter name and type.

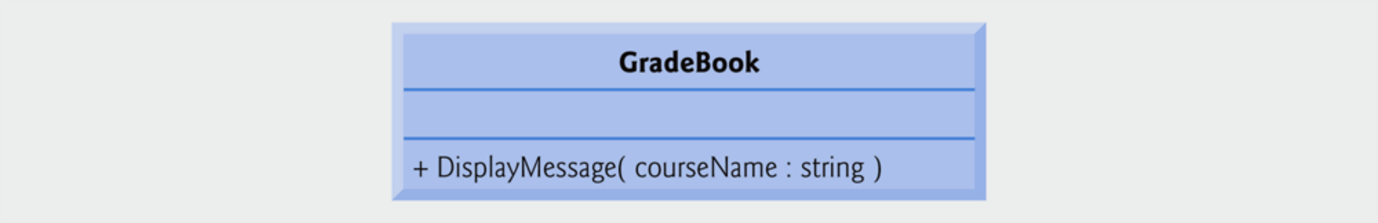


Fig. 4.6 | UML class diagram indicating that class `GradeBook` has a public `DisplayMessage` operation with a `courseName` parameter of type `string`.

4.4 Declaring a Method with a Parameter (Cont.)

- Classes in the same project are considered to be in the same namespace.
- `using` indicates that the application uses classes in another namespace.
- Without `using`, we would write the **fully qualified class name**:

```
System.Console.WriteLine( "Please enter the course
  name:" );
```

4.5 Instance Variables and Properties

- Variables declared in the body of a method are known as **local variables**.
- When a method terminates, the values of its local variables are lost.
- Attributes are represented as variables in a class declaration.
- When each object of a class maintains its own copy of an attribute, the field is known as an instance variable.

- Class `GradeBook` (Fig. 4.7) maintains the course name as an instance variable so that it can be used or modified.

`GradeBook.cs`

(1 of 2)

```
// Fig. 4.7: GradeBook.cs
// GradeBook class that contains a courseName instance variable,
// and a property to get and set its value.
using System;

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

   // property to get and set the course name
```

Declaring `courseName` as an instance variable.

Fig. 4.7 | `GradeBook` class that contains a `private` instance variable, `courseName` and a `public` property to `get` and `set` its value. (Part 1 of 2).

```
   public string CourseName
   {
      get
      {
         return courseName;
      } // end get
      set
      {
         courseName = value;
      } // end set
   } // end property CourseName

   // display a welcome message to the GradeBook user
   public void DisplayMessage()
   {
      // use property CourseName to get the
      // name of the course that this GradeBook represents
      Console.WriteLine( "Welcome to the grade book for\n{0}!",
         CourseName ); // display property CourseName
   } // end method DisplayMessage
} // end class GradeBook
```

GradeBook.cs

(2 of 2)

A public property declaration.

Fig. 4.7 | GradeBook class that contains a private instance variable, courseName and a public property to get and set its value. (Part 2 of 2).

4.5 Instance Variables and Properties (Cont.)

- Variables, properties or methods declared with access modifier `private` are accessible only within the class in which they are declared.

- Declaring instance variables with access modifier `private` is known as **information hiding**.

4.5 Instance Variables and Properties (Cont.)

Software Engineering Observation 4.2

Precede every field and method declaration with an access modifier. Generally, instance variables should be declared `private` and methods and properties should be declared `public`. If the access modifier is omitted before a member of a class, the member is implicitly declared `private`.

Software Engineering Observation 4.3

Declaring the instance variables of a class as `private` and The methods of the class as `public` facilitates debugging, because problems with data manipulations are localized to the class's methods and properties.

4.5 Instance Variables and Properties (Cont.)

Good Programming Practice 4.1

We prefer to list the fields of a class first, so that, as you read the code, you see the names and types of the variables before you see them used in the methods of the class.

Good Programming Practice 4.2

Placing a blank line between method and property declarations enhances code readability.

4.5 Instance Variables and Properties (Cont.)

- We need to provide controlled ways for programmers to “get” and “set” the value of an instance variable.
- Properties contain `get` and `set` **accessors** that handle the details of returning and modifying data.
- After defining a property, you can use it like a variable in your code.

4.5 Instance Variables and Properties (Cont.)

- The `get` accessor begins with the identifier `get` and is delimited by braces.
  - The expression’s value is returned to the client code that uses the property.

```
string theCourseName = gradeBook.CourseName;
```

- `gradeBook.CourseName` implicitly executes the `get` accessor, which returns its value.

4.5 Instance Variables and Properties (Cont.)

- The `set` accessor begins with the identifier `set` and is delimited by braces.

```
gradeBook.CourseName = "CS100 Introduction to Computers";
```

- The text `"CS100 Introduction to Computers"` is assigned to the `set` accessor’s keyword named `value` and the `set` accessor executes.
- A `set` accessor does not return any data.

- Class `GradeBookTest` (Fig. 4.8) creates a `GradeBook` object and demonstrates property `CourseName`.

GradeBookTest.cs

(1 of 2)

```
// Fig. 4.8: GradeBookTest.cs
// Create and manipulate a GradeBook object.
using System;

public class GradeBookTest
{
   // Main method begins program execution
   public static void Main( string[] args )
   {
      // create a GradeBook object and assign it to myGradeBook
      GradeBook myGradeBook = new GradeBook();

      // display initial value of CourseName
      Console.WriteLine( "Initial course name is: '{0}'\n",
         myGradeBook.CourseName );

```

Creating a `GradeBook` object and assigning it to local variable `myGradeBook`.

A `public` property declaration.

Fig. 4.8 | Create and manipulate a `GradeBook` object. (Part 1 of 2).

Outline

GradeBookTest.cs

(2 of 2)

```
      // prompt for and read course name
      Console.WriteLine( "Please enter the course name:" );
      myGradeBook.CourseName = Console.ReadLine(); // set CourseName
      Console.WriteLine(); // output a blank line

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

Assigns the input course name to myGradeBook's `CourseName` property.

Calling DisplayMessage for a welcome message.

```
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!
```

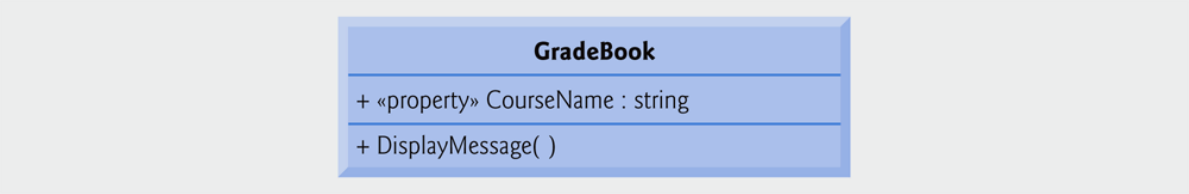
Fig. 4.8 | Create and manipulate a `GradeBook` object. (Part 2 of 2).

4.5 Instance Variables and Properties (Cont.)

- Unlike local variables, every instance variable has a **default initial value**.
- The default value for an instance variable of type `string` is `null`.
- When you display a `string` variable that contains the value `null`, no text is displayed.

4.6 UML Class Diagram with a Property

- Figure 4.9 contains an updated UML class diagram for the version of class `GradeBook`.
- We model properties in the UML as attributes preceded by the word “property” in **guillemets** (« and »).
- To indicate that an attribute is `private`, a class diagram would list the **private visibility symbol**—a minus sign (–)—before the attribute’s name.



GradeBook

+ «property» CourseName : string

+ DisplayMessage( )

Fig. 4.9 | UML class diagram indicating that class `GradeBook` has a public `CourseName` property of type `string` and one public method.

4.7 Software Engineering with Properties and set and get Accessors

- Properties allow the class to control how the data is set or returned.
- For example, `get` and `set` accessors can translate between the format used by the client and the format stored in the `private` instance variable.
- Properties of a class should also be used by the class's own methods.

Software Engineering Observation 4.4

Accessing `private` data through `set` and `get` accessors not only protects the instance variables from receiving invalid values, but also hides the internal representation of the instance variables from that class's clients. Thus, if representation of the data changes, only the properties' implementations need to change.

4.8 Auto-implemented Properties

- Notice that `CourseName`’s `get` accessor simply returns `courseName`’s value and the `set` accessor simply assigns a value to the instance variable.
- For such cases, C# now provides **automatically implemented properties**.
- If you later decide to implement other logic in the `get` or `set` accessors, you can simply reimplement the property.

```
//Read Only Property
public string Name { get; private set; }
//Write Only Property
public string Job { private get; set; }
```

Outline

- Figure 4.10 redefines class `GradeBook` with an auto-implemented `CourseName` property.

GradeBook.cs

```
// Fig. 4.10: GradeBook.cs
// GradeBook class with an auto-implemented property.
using System;

public class GradeBook
{
   // auto-implemented property CourseName implicitly creates
   // an instance variable for this GradeBook's course name
   public string CourseName { get; set; }

   // display a welcome message to the GradeBook user
   public void DisplayMessage()
   {
      // use auto-implemented property CourseName to get the
      // name of the course that this GradeBook represents
      Console.WriteLine( "Welcome to the grade book for\n{0}!",
         CourseName ); // display auto-implemented property CourseName
   } // end method DisplayMessage
} // end class GradeBook
```

Declaring the auto-implemented property.

Implicitly obtaining the property's value.

Fig. 4.10 | GradeBook class with an auto-implemented property.

- The unchanged test program (Fig. 4.11) shows that the auto-implemented property works identically.

`GradeBookTest.cs`

(1 of 2)

```
// Fig. 4.11: GradeBookTest.cs
// Create and manipulate a GradeBook object.
using System;

public class GradeBookTest
{
   // Main method begins program execution
   public static void Main( string[] args )
   {
      // create a GradeBook object and assign it to myGradeBook
      GradeBook myGradeBook = new GradeBook();

      // display initial value of CourseName
      Console.WriteLine( "Initial course name is: '{0}'\n",
         myGradeBook.CourseName );

```

Fig. 4.11 | Create and manipulate a `GradeBook` object. (Part 1 of 2).

GradeBookTest.cs

(2 of 2)

```
      // prompt for and read course name
      Console.WriteLine( "Please enter the course name:" );
      myGradeBook.CourseName = Console.ReadLine(); // set CourseName
      Console.WriteLine(); // output a blank line

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

```
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!
```

Fig. 4.11 | Create and manipulate a `GradeBook` object. (Part 2 of 2).

4.8a Auto-Property Initializers

It is often required to create read-only properties in order to **make the type immutable** (as .NET standards suggest it should be). The following syntax is required for this purpose:

- A **read-only-defined backing field**
- Initialization of **the backing field from within the constructor**
- **Explicit implementation of the property** (rather than using an auto-property)
- **An explicit getter implementation** that returns the backing field

4.8a Auto-Property Initializers

All of this is just to **“properly” implement an immutable property**. This behavior is then repeated for all properties on the type. So doing the right thing requires significantly more effort than the brittle approach.

C# 6.0 comes to the rescue with a new feature called **auto-property initializers**. The **auto-property initializer allows assignment of properties directly within their declaration**.

For read-only properties, it takes care of all the ceremony required to **ensure** the **property is immutable.**

4.8a Auto-Property Initializers

- As the code shows, property initializers allow for **assigning the property an initial value as part of the property declaration**. The property **can be read-only** (only a getter) or **read/write** (both setter and getter).
- When it’s read-only, **the underlying backing field is automatically declared with the read-only modifier**. This ensures that it’s immutable following initialization.
- **Initializers can be any expression**. For example, by leveraging the conditional operator, you can default the initialization value.

```
public class FingerPrint
  {
     public DateTime TimeStamp { get; } = DateTime.UtcNow;
     public string User { get; }     =
                               System.Security.Claims.ClaimsPrincipal.Current.Identity.Name;
     public string Process { get; } =
                               System.Diagnostics.Process.GetCurrentProcess().ProcessName;
  }
```

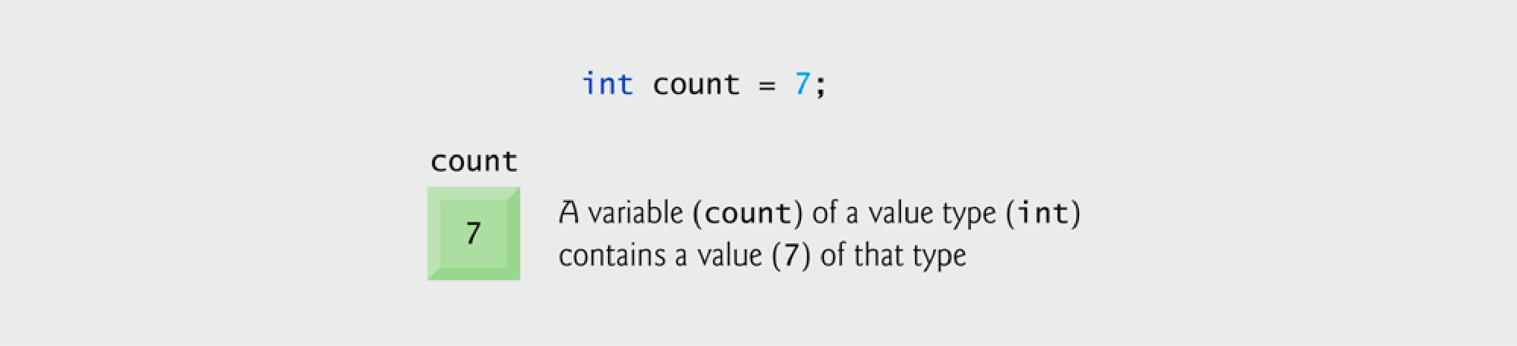
4.8a Auto-Property Initializers

- **Initializers can be any expression**. For example, by leveraging the conditional operator, you can default the initialization value.

```
public class FingerPrint
  {
     public DateTime TimeStamp { get; } = DateTime.UtcNow;
     public string User { get; }    =
                                        System.Security.Claims.ClaimsPrincipal.Current.Identity.Name;
     public string Process { get; } =
                                        System.Diagnostics.Process.GetCurrentProcess().ProcessName;
  }
```

4.9 Value Types vs. Reference Types

- A variable of a value type (such as `int`) simply contains a value of that type (Fig. 4.12).



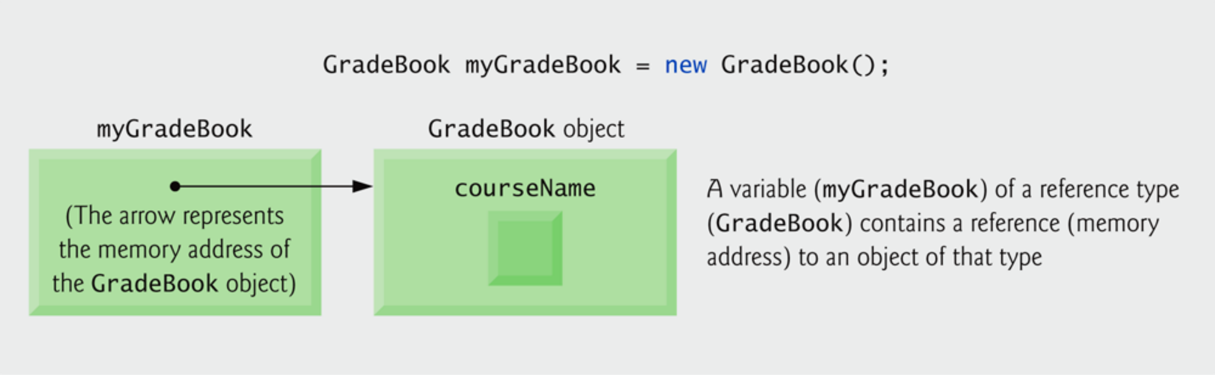
```
int count = 7;
```

A variable (`count`) of a value type (`int`) contains a value (7) of that type

Fig. 4.12 | Value-type variable.

4.9 Value Types vs. Reference Types (Cont.)

- A variable of a **reference type** contains **the address of a location** in memory where its data is stored (Fig. 4.13).
- Reference-type instance variables are initialized by default to the value `null`.
- A variable that refers to an object is used to **invoke** (i.e., call) the object's methods and access the object's properties.



```
GradeBook myGradeBook = new GradeBook();
```

(The arrow represents the memory address of the GradeBook object)

A variable (myGradeBook) of a reference type (GradeBook) contains a reference (memory address) to an object of that type

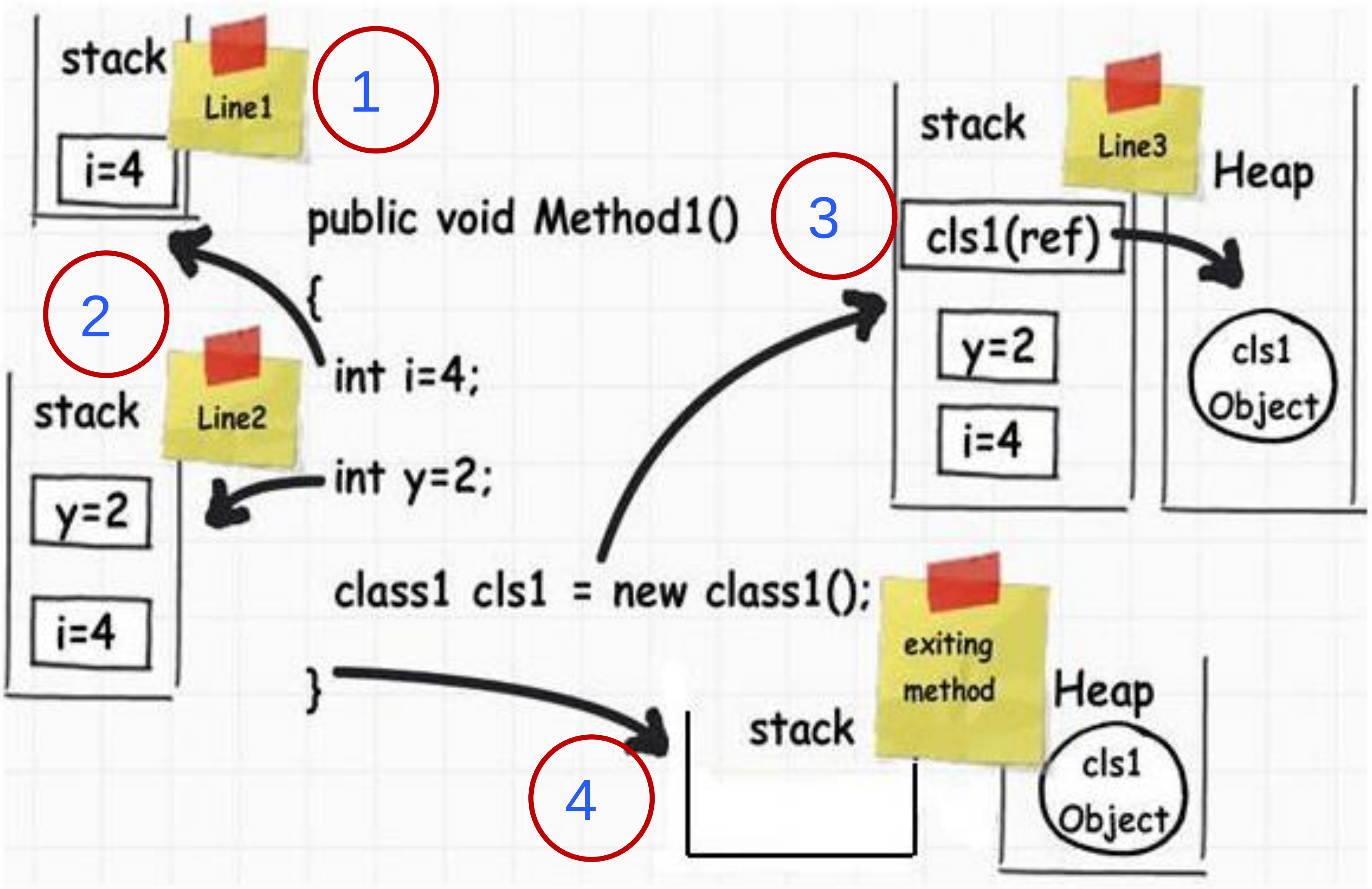
Fig. 4.13 | Reference-type variable.

4.9 Value Types vs. Reference Types

The **heap** is a large block of memory **reserved for instances of reference types**. This block of memory can have “holes” – some of the memory is associated with “live” objects, and some of the memory is free for use by newly created objects.

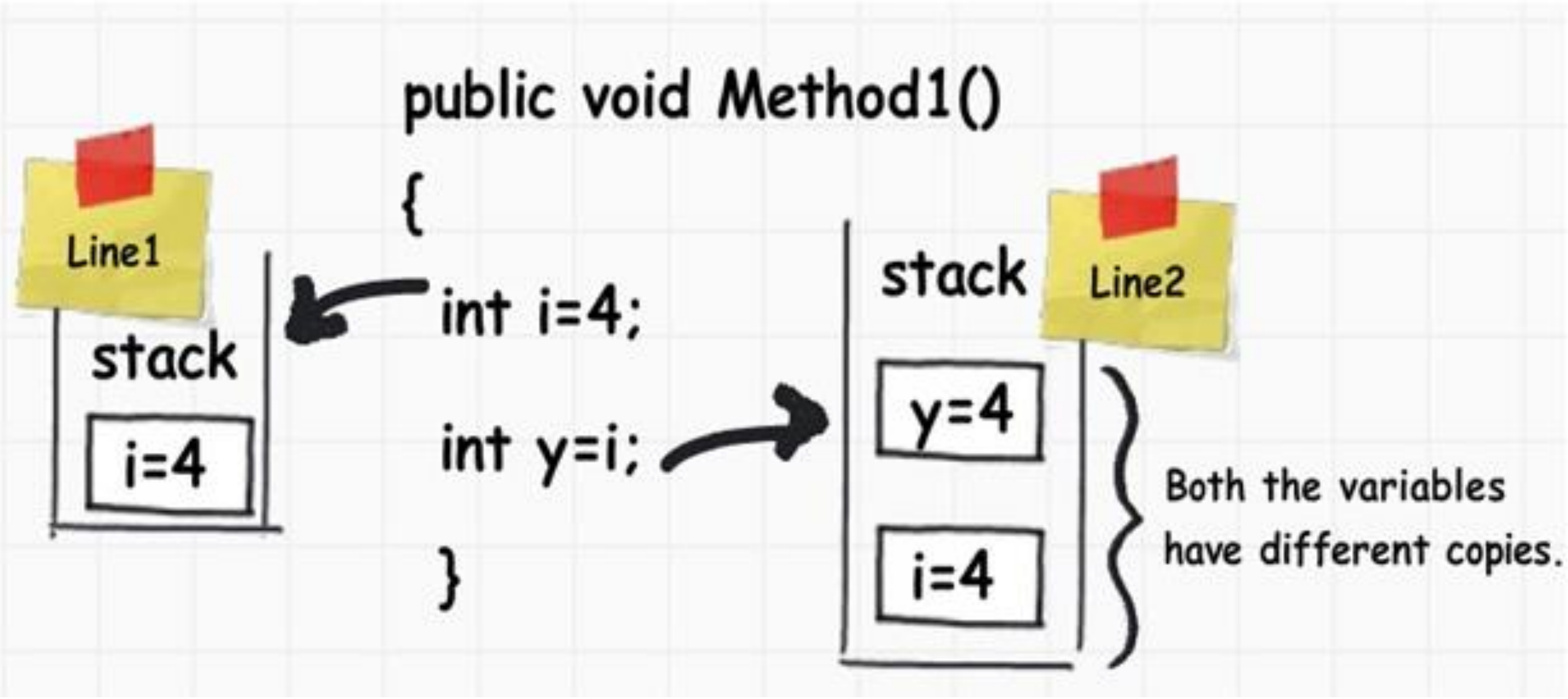
The **stack** is a large block of memory **reserved for local variables of both value or reference type.** It is directly accessible by the processor and it is strictly ordered. Each variable is pushed on the top of the stack and popped out from the top of the stack. The variables that are going to be removed first are on the top, the variables that are going to removed last are on the bottom. Therefore the stack will never have holes, and therefore will not need compacting.

4.9 Value Types vs. Reference Types (Cont.)



4.9 Value Types vs. Reference Types (Cont.)

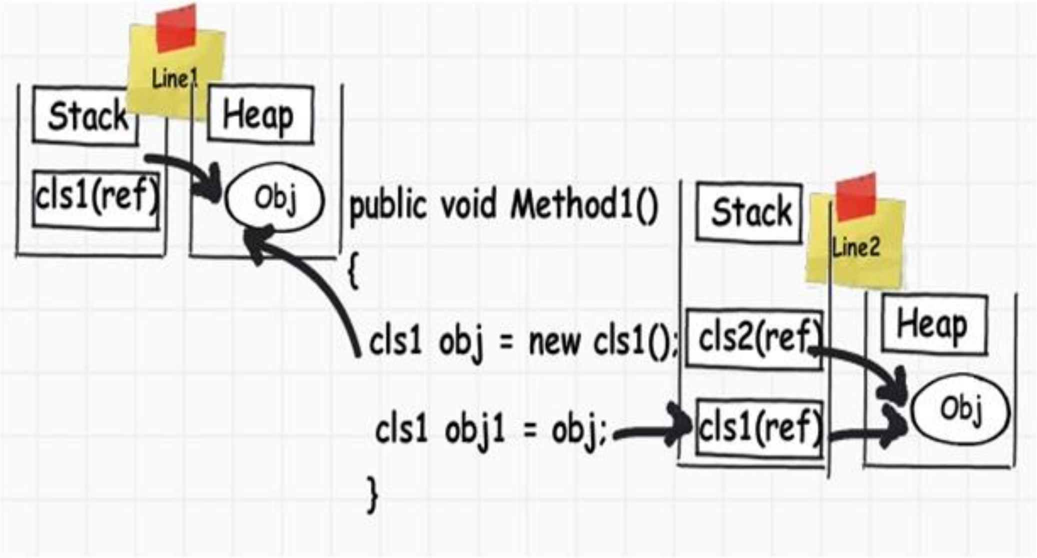
The *by-design semantic meaning* of “value type” is *that they are always copied “by value”.*



4.9 Value Types vs. Reference Types (Cont.)

When we create an object and when we assign an object to another object, they both point to the same memory location as shown in the below code snippet. So when we assign **`obj`** to **`obj1`**, they both point to the same memory location.

In other words if we change one of them, the other object is also affected; this is termed as ‘Reference types’



```
public void Method1()
{
cls1 obj = new cls1();
cls1 obj1 = obj;
}
```

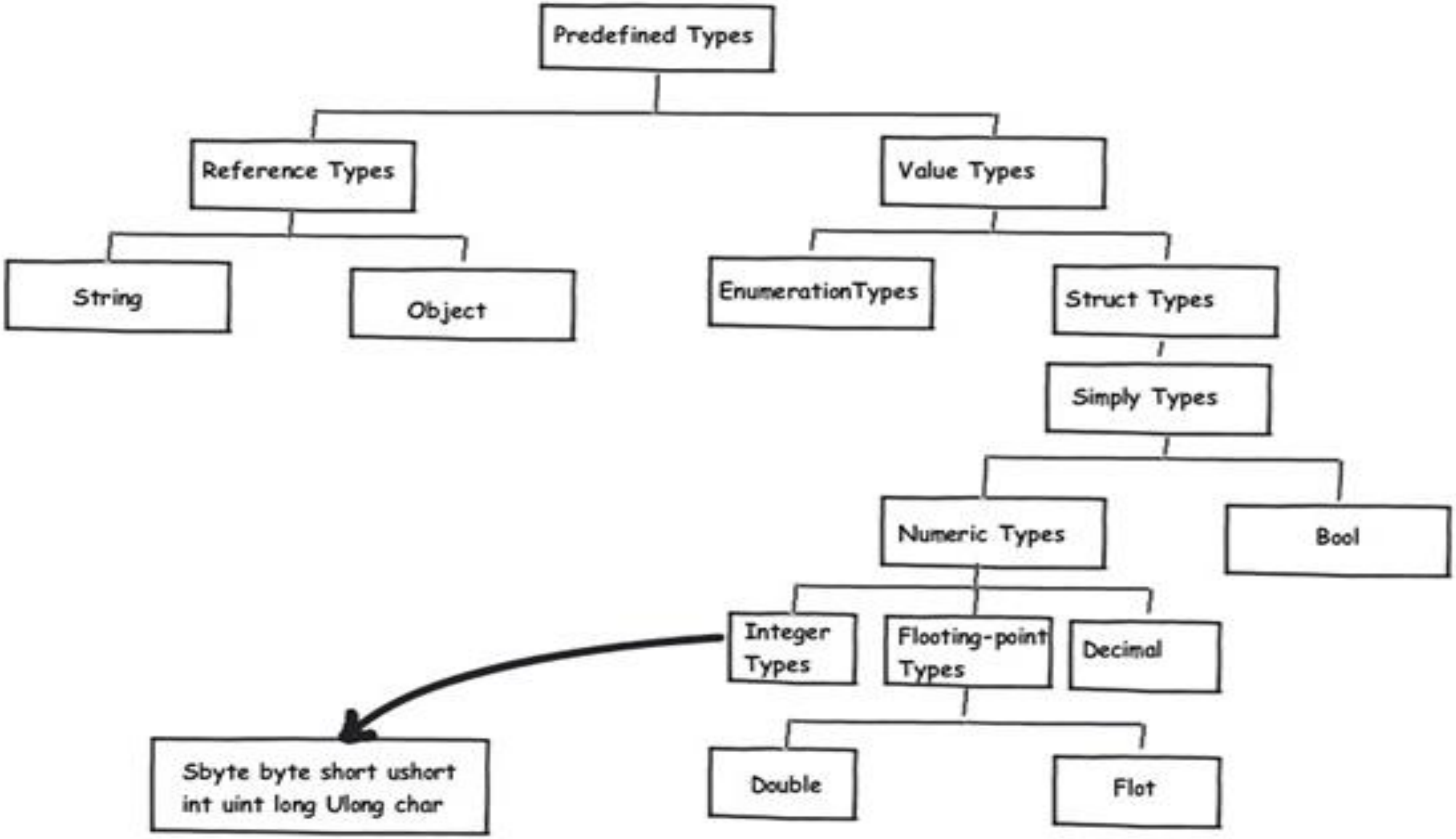
4.9 Value Types vs. Reference Types (Cont.)

Software Engineering Observation 4.5

A variable's declared type indicates whether the variable is of a value or a reference type. If a variable's type is not one of the thirteen simple types, `enum` or a `struct` type, then it is a reference type.

Always make the choice of value type vs reference type based on whether the type is *semantically* representing a value or *semantically* a reference to something.

4.9 Value Types vs. Reference Types (Cont.)



4.10 Initializing Objects with Constructors

- Each class can provide a **constructor** to initialize an object of a class when the object is created.
- The `new` operator calls the class's constructor to perform the initialization.
- The compiler provides a **`public` default constructor** with no parameters, so *every* class has a constructor.

4.10 Initializing Objects with Constructors (Cont.)

- When you declare a class, you can provide your own constructor to specify custom initialization:

```
GradeBook myGradeBook =
    new GradeBook( "CS101 Introduction to C#
                   Programming" );
```

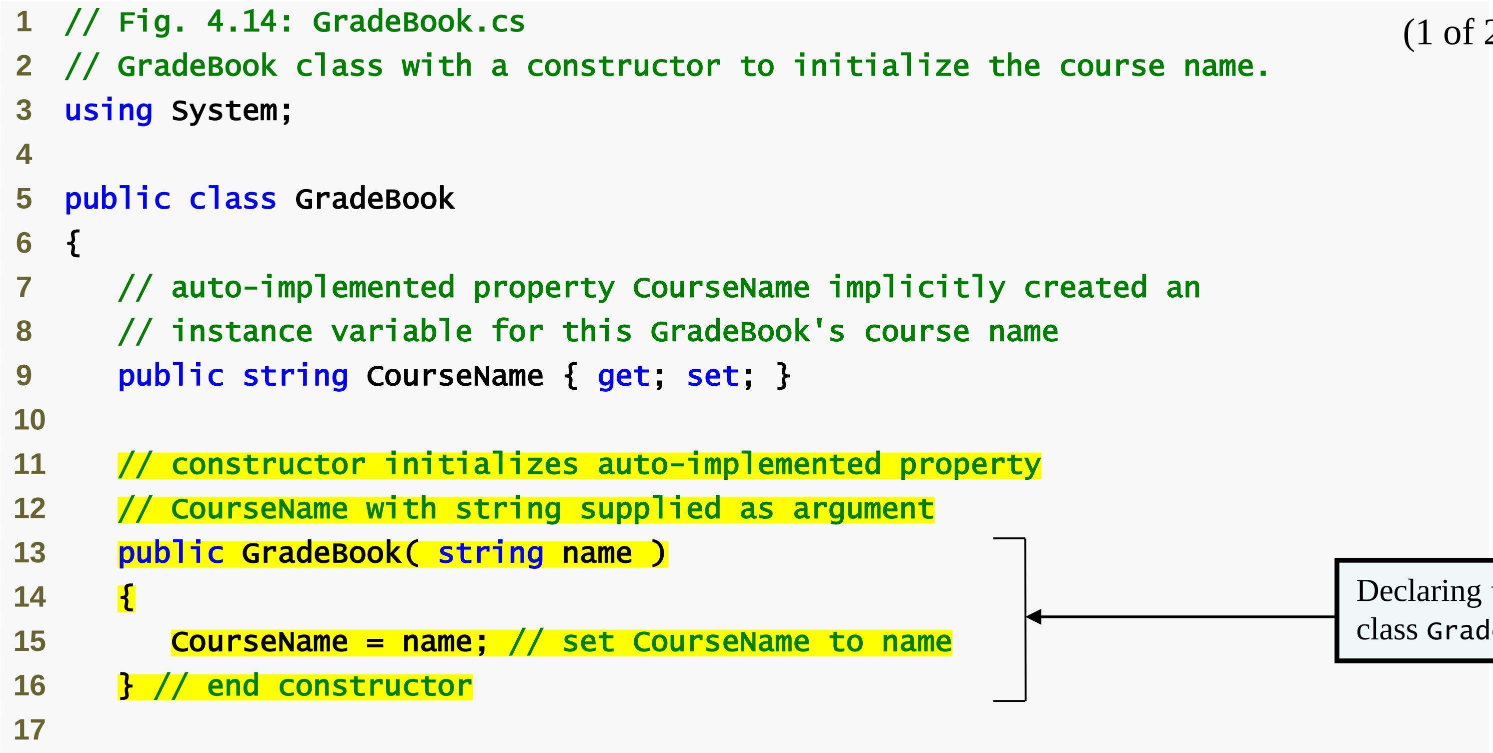
- `"CS101 Introduction to C# Programming"` is passed to the constructor.

Outline

- Figure 4.14 contains a modified `GradeBook` class with a custom constructor.

GradeBook.cs

(1 of 2)



```
// Fig. 4.14: GradeBook.cs
// GradeBook class with a constructor to initialize the course name.
using System;

public class GradeBook
{
   // auto-implemented property CourseName implicitly created an
   // instance variable for this GradeBook's course name
   public string CourseName { get; set; }

   // constructor initializes auto-implemented property
   // CourseName with string supplied as argument
   public GradeBook( string name )
   {
      CourseName = name; // set CourseName to name
   } // end constructor

```

Declaring the constructor for class `GradeBook`.

Fig. 4.14 | `GradeBook` class with a constructor to initialize the course name. (Part 1 of 2).

Outline

GradeBook.cs

(2 of 2)

```
   // display a welcome message to the GradeBook user
   public void DisplayMessage()
   {
      // use auto-implemented property CourseName to get the
      // name of the course that this GradeBook represents
      Console.WriteLine( "Welcome to the grade book for\n{0}!",
         CourseName );
   } // end method DisplayMessage
} // end class GradeBook
```

Fig. 4.14 | `GradeBook` class with a constructor to initialize the course name. (Part 2 of 2).

4.10 Initializing Objects with Constructors (Cont.)

- A constructor must have the same name as its class.
- Like a method, a constructor has a parameter list.

Outline

- Figure 4.15 demonstrates initializing `GradeBook` objects using the constructor.

GradeBookTest.cs

(1 of 2)

```
// Fig. 4.15: GradeBookTest.cs
// GradeBook constructor used to specify the course name at the
// time each GradeBook object is created.
using System;

public class GradeBookTest
{
   // Main method begins program execution
   public static void Main( string[] args )
   {
      // create GradeBook object
      GradeBook gradeBook1 = new GradeBook( // invokes constructor
         "CS101 Introduction to C# Programming" );
      GradeBook gradeBook2 = new GradeBook( // invokes constructor
         "CS102 Data Structures in C#" );

```

Creating and initializing `GradeBook` objects.

Fig. 4.15 | GradeBook constructor used to specify the course name at the time each GradeBook object is created. (Part 1 of 2).

GradeBookTest.cs

(2 of 2)

```
      // display initial value of courseName for each GradeBook
      Console.WriteLine( "gradeBook1 course name is: {0}",
         gradeBook1.CourseName );
      Console.WriteLine( "gradeBook2 course name is: {0}",
         gradeBook2.CourseName );
   } // end Main
} // end class GradeBookTest
```

```
gradeBook1 course name is: CS101 Introduction to C# Programming
gradeBook2 course name is: CS102 Data Structures in C#
```

Fig. 4.15 | GradeBook constructor used to specify the course name at the time each GradeBook object is created. (Part 2 of 2).

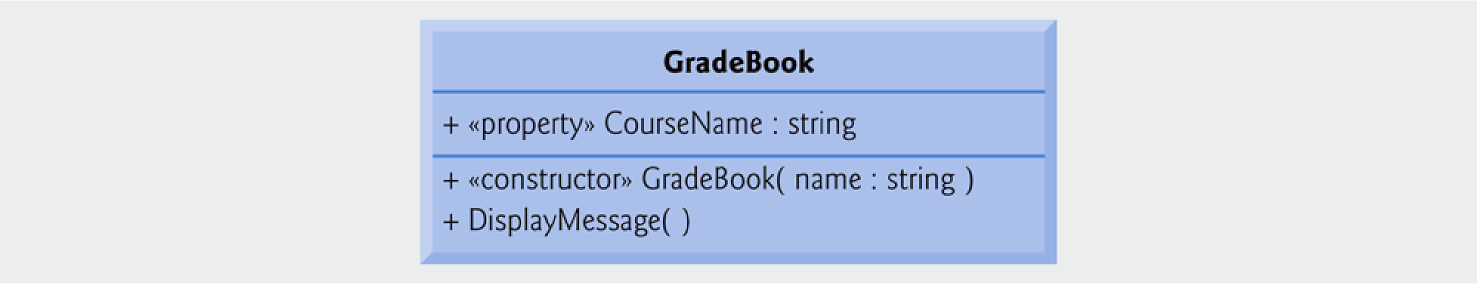
4.10 Initializing Objects with Constructors (Cont.)

Error-Prevention Tip 4.1

Unless default initialization of your class’s instance variables is acceptable, provide a constructor to ensure that your class’s instance variables are properly initialized with meaningful values.

4.10 Initializing Objects with Constructors (Cont.)

- The UML class diagram of Fig. 4.16 models class `GradeBook`.
- To distinguish a constructor from other operations, the UML places the word “constructor” between guillemets (« and »).



GradeBook

+ «property» CourseName : string

+ «constructor» GradeBook(name : string)
+ DisplayMessage( )

Fig. 4.16 | UML class diagram indicating that class `GradeBook` has a constructor with a `name` parameter of type `string`.

4.11 Floating-Point Numbers and Type decimal

- Types `float` and `double` are called **floating-point** types.
- C# treats all real numbers you type in an application’s source code (such as `7.33` and `0.0975`) as `double` values.
- `decimal` variables store a limited range of real numbers, but are more precise and better suited for monetary amounts.
- To type a **`decimal` literal**, you must type the letter “`M`” or “`m`” at the end of a real number.

Common Programming Error 4.3

Using floating-point numbers in a manner that assumes they are represented precisely can lead to logic errors.

- A class named `Account` (Fig. 4.17) maintains the balance of a bank account.

Account.cs

(1 of 2)

```
// Fig. 4.17: Account.cs
// Account class with a constructor to
// initialize instance variable balance.

public class Account
{
   private decimal balance; // instance variable that stores the balance

   // constructor
   public Account( decimal initialBalance )
   {
      Balance = initialBalance; // set balance using property
   } // end Account constructor

   // credit (add) an amount to the account
   public void Credit( decimal amount )
   {
      Balance = Balance + amount; // add amount to balance
   } // end method Credit
```

An instance variable represents each `Account`'s own `balance`.

The constructor receives a parameter that represents the account's starting balance.

Method `Credit` receives one parameter named `amount` that is added to the property `Balance`.

Fig. 4.17 | Account class with a constructor to initialize instance variable balance. (Part 1 of 2).

Outline

Account.cs

(2 of 2)

```

    // a property to get and set the account balance
    public decimal Balance
    {
       get
       {
          return balance;
       } // end get
       set
       {
          // validate that value is greater than or equal to 0;
          // if it is not, balance is left unchanged
          if ( value >= 0 )
             balance = value;
       } // end set
    } // end property Balance
 } // end class Account
```

`Balance`'s `get` accessor returns the value of the `Account`'s `balance`.

`Balance`'s `set` accessor performs validation to ensure that `value` is nonnegative.

Fig. 4.17 | Account class with a constructor to initialize instance variable balance. (Part 2 of 2).

- `AccountTest` (Fig. 4.18) creates two `Account` objects and initializes them with `50.00M` and `-7.53M` (decimal literals).

`AccountTest.cs`

(1 of 3)

```
// Fig. 4.18: AccountTest.cs
// Create and manipulate Account objects.
using System;

public class AccountTest
{
   // Main method begins execution of C# application
   public static void Main( string[] args )
   {
      Account account1 = new Account( 50.00M ); // create Account object
      Account account2 = new Account( -7.53M ); // create Account object

      // display initial balance of each object using a property
      Console.WriteLine( "account1 balance: {0:C}",
         account1.Balance ); // display Balance property
      Console.WriteLine( "account2 balance: {0:C}\n",
         account2.Balance ); // display Balance property

```

Passing an initial balance which will be invalidated by `Balance`'s `set` accessor.

Outputting the `Balance` property of each `Account`.

Fig. 4.18 | Create and manipulate an Account object. (Part 1 of 3).

AccountTest.cs

(2 of 3)

```
      decimal depositAmount; // deposit amount read from user

      // prompt and obtain user input
      Console.Write( "Enter deposit amount for account1: " );
      depositAmount = Convert.ToDecimal( Console.ReadLine() );
      Console.WriteLine( "adding {0:C} to account1 balance\n",
         depositAmount );
      account1.Credit( depositAmount ); // add to account1 balance

      // display balances
      Console.WriteLine( "account1 balance: {0:C}",
         account1.Balance );
      Console.WriteLine( "account2 balance: {0:C}\n",
         account2.Balance );

      // prompt and obtain user input
      Console.Write( "Enter deposit amount for account2: " );
      depositAmount = Convert.ToDecimal( Console.ReadLine() );
```

Local variable `depositAmount` is *not* initialized to `0` but will be set by the user’s input.

Obtaining input from the user.

Obtaining the deposit value from the user.

Fig. 4.18 | Create and manipulate an Account object. (Part 2 of 3).

AccountTest.cs

(3 of 3)

```
      Console.WriteLine( "adding {0:C} to account2 balance\n",
         depositAmount );
      account2.Credit( depositAmount ); // add to account2 balance

      // display balances
      Console.WriteLine( "account1 balance: {0:C}", account1.Balance );
      Console.WriteLine( "account2 balance: {0:C}", account2.Balance );
   } // end Main
} // end class AccountTest
```

Outputting the balances of both Accounts.

```
account1 balance: $50.00
account2 balance: $0.00

Enter deposit amount for account1: 49.99
adding $49.99 to account1 balance
```

```
account1 balance: $99.99
account2 balance: $0.00

Enter deposit amount for account2: 123.21
adding $123.21 to account2 balance

account1 balance: $99.99
account2 balance: $123.21
```

Fig. 4.18 | Create and manipulate an Account object. (Part 3 of 3).

4.11 Floating-Point Numbers and Type decimal (Cont.)

- A value output with the format item `{0:C}` appears as a monetary amount.
- The `:` indicates that the next character represents a **format specifier**.

4.11 Floating-Point Numbers and Type decimal (Cont.)

Format specifier	Description
`C` or `c`	Formats the string as currency.
`D` or `d`	Formats the string as a decimal.
`N` or `n`	Formats the string with a thousands separator and two decimal places.
`E` or `e`	Formats the number using scientific notation.
`F` or `f`	Formats the string with a fixed number of decimal places.
`G` or `g`	Default setting. Formats the number with decimal places or using scientific notation, depending on context.
`X` or `x`	Formats the string as hexadecimal.

Fig. 4.19 | `string` format specifiers.

4.11 Floating-Point Numbers and Type decimal (Cont.)

- It is possible to declare the `get` and `set` accessors with different access modifiers.
- One of the accessors must implicitly have the same access as the property and the other must be declared with a more restrictive access modifier.

Error-Prevention Tip 4.2

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

Error-Prevention Tip 4.3

`set` accessors that set the values of `private` data should verify that the intended new values are proper; if they are not, the `set` accessors should leave the instance variables unchanged and indicate an error.

4.11 Floating-Point Numbers and Type decimal (Cont.)

- The UML class diagram in Fig. 4.20 models class `Account`.

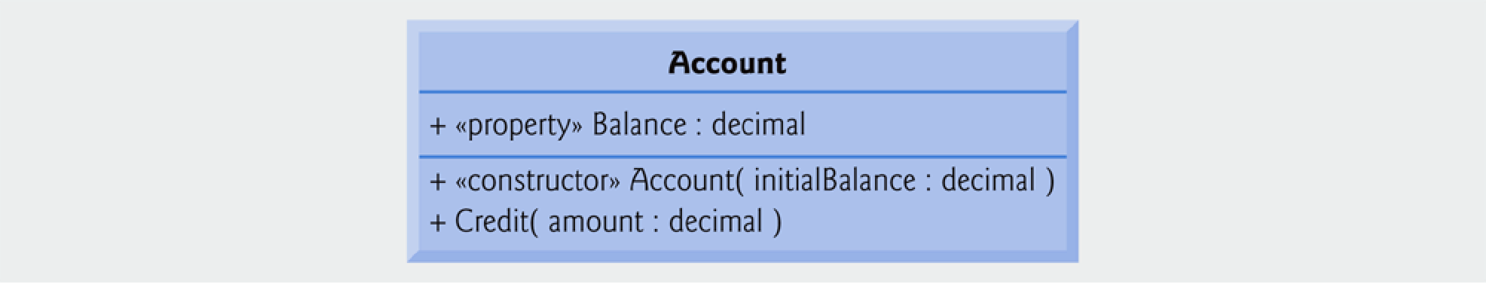

Account
+ «property» Balance : decimal
+ «constructor» Account(initialBalance : decimal)
+ Credit(amount : decimal)

Fig. 4.20 | UML class diagram indicating that class `Account` has a `public` `Balance` property of type `decimal`, a constructor and a method.

4.12 Software Engineering Case Study: Identifying the Classes in the ATM Requirements Document

- We create classes only for the nouns and noun phrases in the ATM system (Fig. 4.21).
- We do not need to model some nouns such as “bank” which are not part of the ATM operations.

Nouns and noun phrases in the requirements document		
bank	money / funds	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. 4.21 | Nouns and noun phrases in the requirements document.

4.12 Identifying the Classes in the ATM Requirements Document (Cont.)

- UML **class diagrams** model the classes in the ATM system and their interrelationships (Fig. 4.22).
  - The top compartment contains the name of the class.
  - The middle compartment contains the class's attributes.
  - The bottom compartment contains the class's operations.

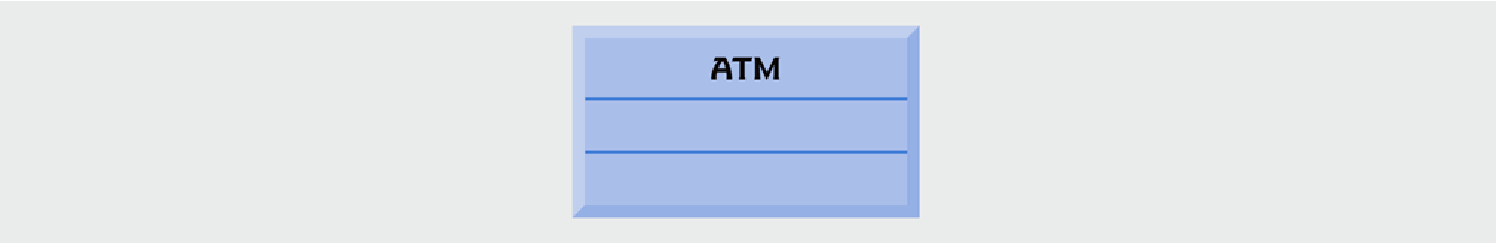


Fig. 4.22 | Representing a class in the UML using a class diagram.

4.12 Identifying the Classes in the ATM Requirements Document (Cont.)

- Figure 4.23 shows how our classes `ATM` and `Withdrawal` relate to one another.
  - The line that connects the two classes represents an **association**.
  - **Multiplicity** values indicate how many objects of each class participate in the association.
  - One `ATM` object participates in an association with either zero or one `Withdrawal` objects.
- `currentTransaction` is a **role name**, which identifies the role the `Withdrawal` object plays.

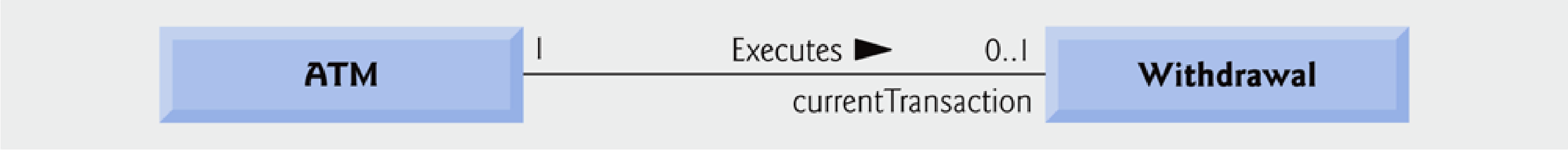


Fig. 4.23 | Class diagram showing an association among classes.

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

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. 4.24 | Multiplicity types.

4.12 Identifying the Classes in the ATM Requirements Document (Cont.)

- In Fig. 4.25, the **solid diamonds** indicate that class `ATM` has a **composition** relationship with classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`.
- Composition implies a whole/part relationship—the ATM “has a” screen, a keypad, a cash dispenser and a deposit slot.
- The **has-a relationship** defines composition.

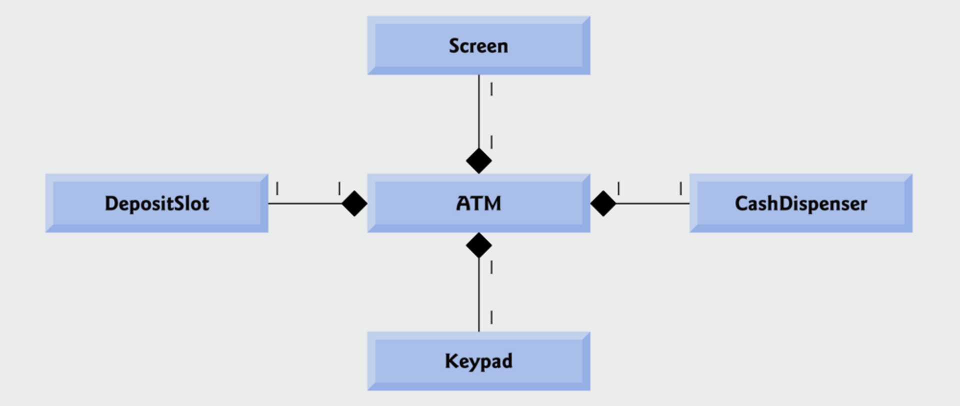


Fig. 4.25 | Class diagram showing composition relationships.

4.12 Identifying the Classes in the ATM Requirements Document (Cont.)

- Composition relationships have the following properties:
  - Only one class in the relationship can represent the whole.
  - The parts in the composition relationship exist only as long as the whole.
  - A part may belong to only one whole at a time.
- If a “has-a” relationship does not satisfy one or more of these criteria, hollow diamonds are used to indicate **aggregation**.

4.12 Identifying the Classes in the ATM Requirements Document (Cont.)

- Figure 4.26 shows a class diagram for the ATM system.
- The class diagram shows that class `ATM` has a **one-to-one relationship** with class `BankDatabase`.
- We also model that one object of class `BankDatabase` participates in a composition relationship with zero or more objects of class `Account`.

4.12 Identifying the Classes in the ATM Requirements Document (Cont.)

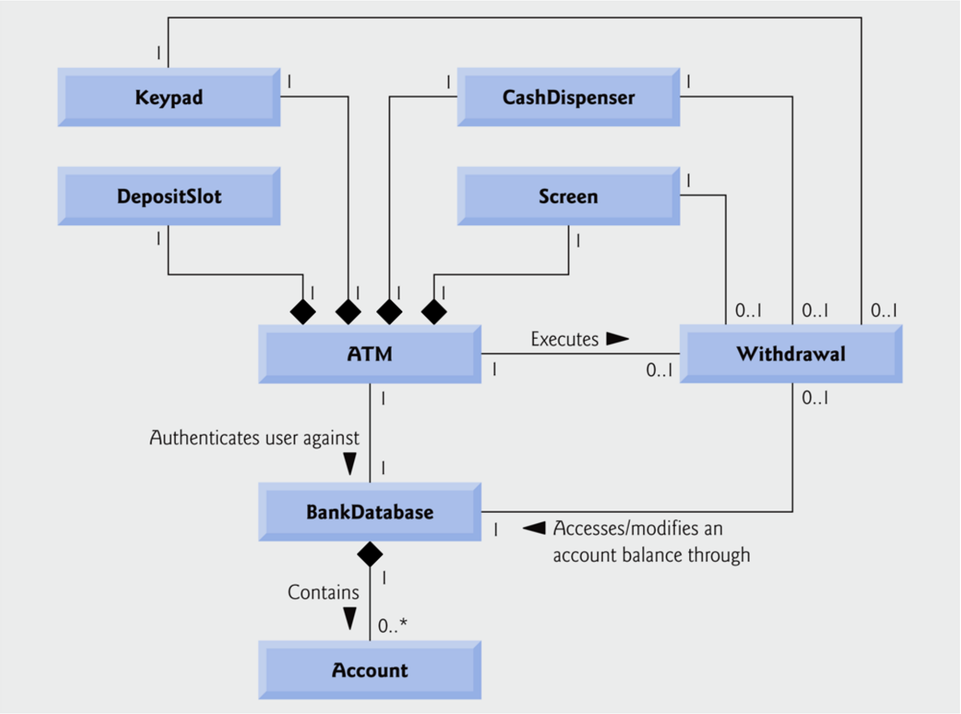


Fig. 4.26 | Class diagram for the ATM system model.