Introduction to Classes and Objects

In this chapter 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 declare methods in a class to implement the class's behaviors.
- How to declare instance variables in a class to implement the class's attributes.
- How to call an object's method to make that method perform its task.
- How to use a constructor to ensure that an object's data is initialized when the object is created.

Classes, Objects, Methods and Instance Variables

- Class provides one or more methods
- Method represents task in a program
  - Describes the mechanisms that actually perform its tasks
  - Hides from its user the complex tasks that it performs
  - Method call tells method to perform its task
- Classes contain one or more attributes
  - Specified by instance variables
  - Carried with the object as it is used

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

- Each class declaration that begins with keyword **public** must be stored in a file that has the same name as the class and ends with the .java file-name extension.

Class `GradeBook`

- keyword `public` is an access modifier
- Class declarations include:
  - Access modifier
  - Keyword `class`
  - Pair of left and right braces

- Method declarations
  - Keyword `public` indicates method is available to public
  - Keyword `void` indicates no return type
  - Access modifier, return type, name of method and parentheses comprise method header

```java
// 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
```

Class `GradeBookTest`

- Java is extensible
  - Programmers can create new classes
- Class instance creation expression
  - Keyword `new`
  - Then name of class to create and parentheses
- Calling a method
  - Object name, then dot separator (`.`)
  - Then method name and parentheses

```java
// Fig. 3.2: GradeBookTest.java
// Create a GradeBook object and call its displayMessage method.

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

      // call myGradeBook's displayMessage metho
      myGradeBook.displayMessage();
   } // end main

} // end class GradeBookTest
```

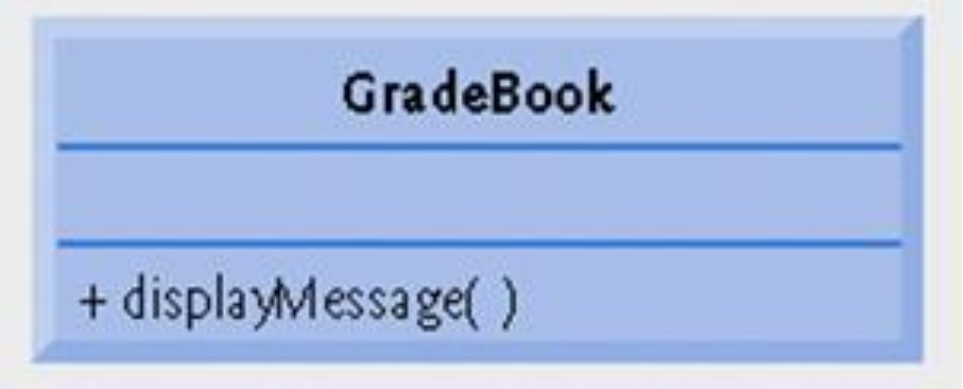
Use class instance creation expression to create object of class GradeBook

Call method displayMessage using GradeBook object

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

Unified Modeling Language (UML) Class Diagram for Class GradeBook

- UML class diagrams
  - Top compartment contains name of the class
  - Middle compartment contains class's attributes or instance variables
  - Bottom compartment contains class's operations or methods
    - Plus sign indicates `public` methods



Declaring a Method with a Parameter

- Method parameters
  - Additional information passed to a method
  - Supplied in the method call with arguments
- Scanner methods
  - nextLine reads next line of input
  - next reads next word of input

```java
// Fig. 3.4: GradeBook.java
// Class declaration with a 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
```

```java
// Fig. 3.5: GradeBookTest.java
// Create GradeBook object and pass a String to
// its displayMessage method.
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();

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

      // call myGradeBook's displayMessage method
      // and pass nameOfCourse as an argument
      myGradeBook.displayMessage( nameOfCourse );
   } // end main

} // end class GradeBookTest
```

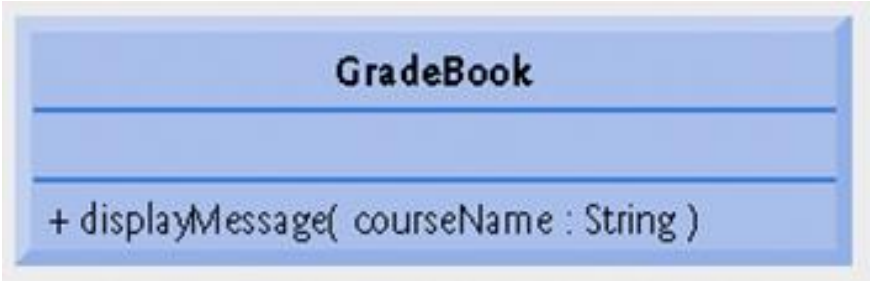
```
Please enter the course name:
CS101 Introduction to Java Programming

Welcome to the grade book for
CS101 Introduction to Java Programming!
```

Software Engineering Observation

- Normally, objects are created with new.
- Parameters specified in method's parameter list
  - Part of method header
  - Uses a comma-separated list
- A compilation error occurs if the number of arguments in a method call does not match the number of parameters in the method declaration.
- 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.

Updated UML Class Diagram for Class `GradeBook`

- UML class diagram
  - Parameters specified by parameter name followed by a colon and parameter type



Notes on `Import` Declarations

- `java.lang` is implicitly imported into every program
- Default package
  - Contains classes compiled in the same directory
  - Implicitly imported into source code of other files in directory
- Imports unnecessary if fully-qualified names are used

- The Java compiler does not require `import` declarations in a Java source code file if the fully qualified class name is specified every time a class name is used in the source code. But most Java programmers consider using fully qualified names to be cumbersome, and instead prefer to use `import` declarations.

Instance Variables, *set* Methods and *get* Methods

- Variables declared in the body of method
  - Called local variables
  - Can only be used within that method
- Variables declared in a class declaration
  - Called fields or instance variables
  - Each object of the class has a separate instance of the variable

```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()
   {
      // 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
```

Instance variable `courseName`

set method for `courseName`

get method for `courseName`

Call *get* method

Access Modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods, and constructors.

The four access levels are –

- Visible to the package, the default. No modifiers are needed.
- Visible to the world (public).
- Visible to the class only (private).
- Visible to the package and all subclasses (protected).

- **Default Access Modifier - No Keyword**
  - Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.
  - A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

- **Private Access Modifier - Private**
  - Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.
  - Private access modifier is the most restrictive access level. Class and interfaces cannot be private.
  - Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.
  - Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

- **Public Access Modifier - Public**
  - A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.
  - However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.
- **Protected Access Modifier - Protected**
  - Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.
  - The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.
  - Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Access Modifiers `public` and `private`

- private keyword
  - Used for most instance variables
  - `private` variables and methods are accessible only to methods of the class in which they are declared
  - Declaring instance variables `private` is known as data hiding
- return type
  - Indicates item returned by method
  - Declared in method header

Software Engineering Observation

- Precede every field and method declaration with an access modifier. As a rule of thumb, instance variables should be declared `private` and methods should be declared `public`. (We will see that it is appropriate to declare certain methods `private`, if they will be accessed only by other methods of the class.)

- 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. It is possible to list the class's fields anywhere in the class outside its method declarations, but scattering them tends to lead to hard-to-read code.
- Place a blank line between method declarations to separate the methods and enhance program readability.

`GradeBookTest` Class That Demonstrates Class `GradeBook`

- Default initial value
  - Provided for all fields not initialized
  - Equal to `null` for `String`s

set and *get* methods

- `private` instance variables
  - Cannot be accessed directly by clients of the object
  - Use *set* methods to alter the value
  - Use *get* methods to retrieve the value

```java
// Fig. 3.8: GradeBookTest.java
// Create and manipulate 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 messag
      myGradeBook.displayMessage();
   } // end main

} // end class GradeBookTest
```

Call *get* method for `courseName`

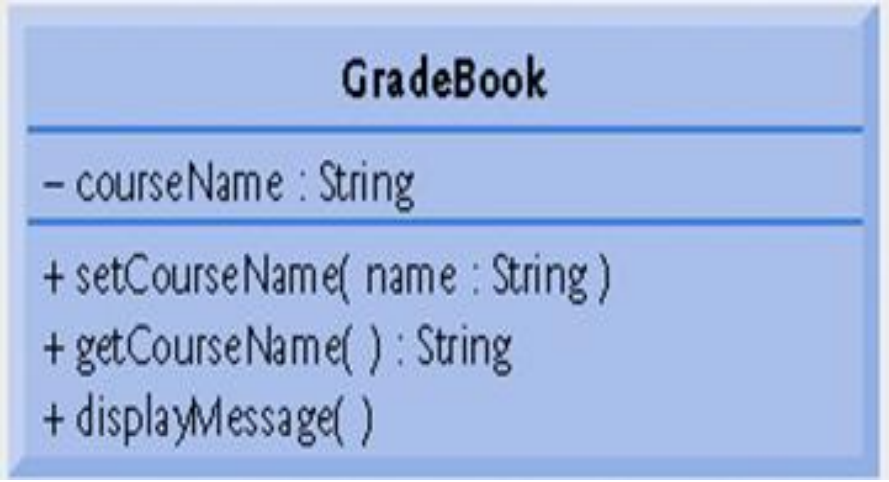
Call *set* method for `courseName`

Call `displayMessage`

```
Initial course name is: null

Please enter the course name:
CS101 Introduction to Java Programming

Welcome to the grade book for
CS101 Introduction to Java Programming!
```

`GradeBook`'s UML Class Diagram with an Instance Variable and *set* and *get* Methods

- Attributes
  - Listed in middle compartment
  - Attribute name followed by colon followed by attribute type
- Return type of a method
  - Indicated with a colon and return type after the parentheses after the operation name



Primitive Types vs. Reference Types

- Types in Java
  - Primitive
    - `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`
  - Reference (sometimes called nonprimitive types)
    - Objects
    - Default value of `null`
    - Used to invoke an object's methods

- A variable's declared type (e.g., `int`, `double` or `GradeBook`) indicates whether the variable is of a primitive or a reference type. If a variable's type is not one of the eight primitive types, then it is a reference type. For example, `Account account1` indicates that `account1` is a reference to an `Account` object).

Initializing Objects with Constructors

- Constructors
  - Initialize an object of a class
  - Java requires a constructor for every class
  - Java will provide a default no-argument constructor if none is provided
  - Called when keyword `new` is followed by the class name and parentheses

```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 supplied as argument
   public GradeBook( String 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
```

Constructor to initialize `courseName` variable

```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
```

Call constructor to create first grade book object

Create second grade book object

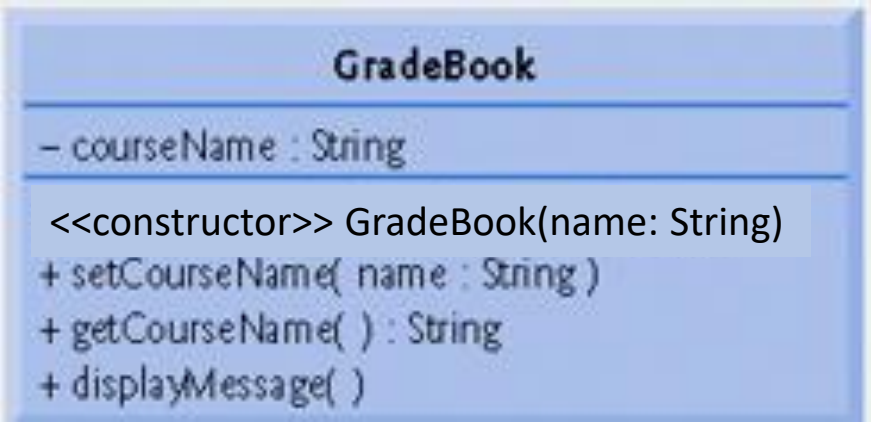
```
gradeBook1 course name is: CS101 Introduction to Java Programming
gradeBook2 course name is: CS102 Data Structures in Java
```

Software Engineering Observation:

- 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 when each new object of your class is created.

Adding the Constructor to Class `GradeBook`'s UML Class Diagram

- UML class diagram
  - Constructors go in third compartment
  - Place "<<constructor>>" before constructor name
  - By convention, place constructors first in their compartment



An example of Account Class

```java
// Fig. 3.13: Account.java
// Account class with a constructor to
// initialize instance variable balance.

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

   // constructor
   public Account( double initialBalance )
   {
      // validate that initialBalance is greater than 0.0;
      // if it is not, balance is initialized to the default value 0.0
      if ( initialBalance > 0.0 )
         balance = initialBalance;
   } // end Account constructor

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

   // return the account balance
   public double getBalance()
   {
      return balance; // gives the value of balance to the calling method
   } // end method getBalance

} // end class Account
```

AccountTest Class to use Class Account

```java
// Fig. 3.14: AccountTest.java
// Create and manipulate an Account object.
import java.util.Scanner;

public class AccountTest
{
   // main method begins execution of Java application
   public static void main( String args[] )
   {
      Account account1 = new Account( 50.00 ); // create Account object
      Account account2 = new Account( -7.53 ); // create Account object

      // display initial balance of each object
      System.out.printf( "account1 balance: $%.2f\n",
         account1.getBalance() );
      System.out.printf( "account2 balance: $%.2f\n\n",
         account2.getBalance() );

      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );
      double depositAmount; // deposit amount read from user

      System.out.print( "Enter deposit amount for account1: " ); // prompt
      depositAmount = input.nextDouble(); // obtain user input
      System.out.printf( "\nadding %.2f to account1 balance\n\n",
         depositAmount );
      account1.credit( depositAmount ); // add to account1 balance

      // display balances
      System.out.printf( "account1 balance: $%.2f\n",
         account1.getBalance() );
      System.out.printf( "account2 balance: $%.2f\n\n",
         account2.getBalance() );

      System.out.print( "Enter deposit amount for account2: " ); // prompt
      depositAmount = input.nextDouble(); // obtain user input
      System.out.printf( "\nadding %.2f to account2 balance\n\n",
         depositAmount );
      account2.credit( depositAmount ); // add to account2 balance

```

```java
      // display balances
      System.out.printf( "account1 balance: $%.2f\n",
         account1.getBalance() );
      System.out.printf( "account2 balance: $%.2f\n",
         account2.getBalance() );
   } // end main

} // end class AccountTest
```

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

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

account1 balance: $75.53
account2 balance: $0.00

Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

account1 balance: $75.53
account2 balance: $123.45
```

Output

UML class diagram indicating that class `Account` has a `private` `balance` attribute of UML type `Double`, a constructor (with a parameter of UML type `Double`) and two `public` `operations`– `credit` (with an `amount` parameter of UML type `Double`) and `getBalance` (returns UML type `Double`).