

Introduction to Classes and Objects

Classes, Objects, Methods and Instance Variables

- Analogy to help you understand classes and their contents.
 - Suppose you want to drive a car and make it go faster by pressing down on its accelerator pedal.
 - Before you can drive a car, someone has to design it.
 - A car typically begins as engineering drawings, similar to the blueprints used to design a house.
 - These include the design for an accelerator pedal to make the car go faster.

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

- Analogy to help you understand classes and their contents.
 - The pedal “hides” from the driver the complex mechanisms that actually make the car go faster, just as the brake pedal “hides” the mechanisms that slow the car and the steering wheel “hides” the mechanisms that turn the car.
 - This enables people with little or no knowledge of how engines work to drive a car easily.
 - Before you can drive a car, it must be built from the engineering drawings that describe it.
 - A completed car has an actual accelerator pedal to make the car go faster, but even that’s not enough—the car won’t accelerate on its own, so the driver must press the accelerator pedal.

3

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

- Performing a task in a program requires a method.
 - The method describes the mechanisms that actually perform its tasks.
 - Hides from its user the complex tasks that it performs, just as the accelerator pedal of a car hides from the driver the complex mechanisms of making the car go faster.
- In Java, a class houses a method, just as a car’s engineering drawings house the design of an accelerator pedal.
- In a class, you provide one or more methods that are designed to perform the class’s tasks.

4

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

- You must build an object of a class before a program can perform the tasks the class describes how to do.
 - That is one reason Java is known as an object-oriented programming language.
- When you drive a car, pressing its gas pedal sends a message to the car to perform a task—make the car go faster.
- You **send messages** to an object—each message is implemented as a **method call** that tells a method of the object to perform its task.

5

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

- A car has many attributes
 - Color, the number of doors, the amount of gas in its tank, its current speed and its total miles driven.
- Attributes are represented as part of a car's design in its engineering diagrams.
- Every car maintains its own attributes.
 - Each car knows how much gas is in its own gas tank, but not how much is in the tanks of other cars.

6

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

- An object has attributes that are carried with the object as it's used in a program.
 - Specified as part of the object's class.
 - A bank account object has a balance attribute that represents the amount of money in the account.
 - Each bank account object knows the balance in the account it represents, but not the balances of the other accounts in the bank.
- Attributes are specified by the class's **instance variables**.

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

- Create a new class (`GradeBook`)
- Use it to create an object.
- 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.
- Keyword `public` is an **access modifier**.
 - Indicates that the class is “available to the public”



Common Programming Error 3.1

Declaring more than one `public` class in the same file is a compilation error.

9

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

- The `main` method is called automatically by the Java Virtual Machine (JVM) when you execute an application.
- Normally, you must call methods explicitly to tell them to perform their tasks.
- A `public` is “available to the public”
 - It can be called from methods of other classes.
- The `return type` specifies the type of data the method returns after performing its task.
- Return type `void` indicates that a method will perform a task but will *not* return (i.e., give back) any information to its `calling method` when it completes its task.

10

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

- Method name follows the return type.
- By convention, method names begin with a lowercase first letter and subsequent words in the name begin with a capital letter.
- Empty parentheses after the method name indicate that the method does not require additional information to perform its task.
- Together, everything in the first line of the method is typically called the **Method header**
- Every method's body is delimited by left and right braces.
- The method body contains one or more statements that perform the method's task.

11

```
1 // Fig. 3.1: GradeBook.java
2 // Class declaration with one method.
3
4 public class GradeBook
5 {
6     // display a welcome message to the GradeBook user
7     public void displayMessage()
8     {
9         System.out.println("Welcome to the Grade Book!");
10    } // end method displayMessage
11 } // end class GradeBook
```

Performs the task of displaying a message on the screen; method `displayMessage` must be called to perform this task

Fig. 3.1 | Class declaration with one method.

12

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

- Use class `GradeBook` in an application.
- Class `GradeBook` is not an application because it does not contain `main`.
- Can't execute `GradeBook`; will receive an error message like:
 - Exception in thread "main"
java.lang.NoSuchMethodError: main
- Must either declare a separate class that contains a `main` method or place a `main` method in class `GradeBook`.
- To help you prepare for the larger programs, use a separate class containing method `main` to test each new class.
- Some programmers refer to such a class as a driver class.

13

```
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
```

Creates a `GradeBook` object and assigns it to variable `myGradeBook`

Invokes method `displayMessage` on the `GradeBook` object that was assigned to variable `myGradeBook`

Welcome to the Grade Book!

Fig. 3.2 | Creating a `GradeBook` object and calling its `displayMessage` method.

14

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

- A `static` method (such as `main`) is special
 - It can be called without first creating an object of the class in which the method is declared.
- Typically, you cannot call a method that belongs to another class until you create an object of that class.
- Declare a variable of the class type.
 - Each new class you create becomes a new type that can be used to declare variables and create objects.
 - You can declare new class types as needed; this is one reason why Java is known as an **extensible language**.

15

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

- **Class instance creation expression**
 - Keyword `new` creates a new object of the class specified to the right of the keyword.
 - Used to initialize a variable of a class type.
 - The parentheses to the right of the class name are required.
 - Parentheses in combination with a class name represent a call to a **constructor**, which is similar to a method but is used only at the time an object is created to initialize the object's data.

16

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

- Call a method via the class-type variable
 - Variable name followed by a **dot separator (.)**, the method name and parentheses.
 - Call causes the method to perform its task.
- Any class can contain a `main` method.
 - The JVM invokes the `main` method only in the class used to execute the application.
 - If multiple classes that contain `main`, then one that is invoked is the one in the class named in the `java` command.

17

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

- Compiling an Application with Multiple Classes
 - Compile the classes in Fig. 3.1 and Fig. 3.2 before executing.
 - Type the command

```
javac GradeBook.java
GradeBookTest.java
```
 - If the directory containing the application includes only this application's files, you can compile all the classes in the directory with the command

```
javac *.java
```

18

Declaring a Method with a Parameter

- Car analogy
 - Pressing a car's gas pedal sends a message to the car to perform a task—make the car go faster.
 - The farther down you press the pedal, the faster the car accelerates.
 - Message to the car includes the task to perform and additional information that helps the car perform the task.
- **Parameter**: Additional information a method needs to perform its task.

19

Declaring a Method with a Parameter (Cont.)

- A method can require one or more parameters that represent additional information it needs to perform its task.
 - Defined in a comma-separated **parameter list**
 - Located in the parentheses that follow the method name
 - Each parameter must specify a type and an identifier.
- A method call supplies values—called arguments—for each of the method's parameters.

20

```

1 // Fig. 3.4: GradeBook.java
2 // Class declaration with a method that has a parameter.
3
4 public class GradeBook
5 {
6     // display a welcome message to the GradeBook user
7     public void displayMessage( String courseName )
8     {
9         System.out.printf( "Welcome to the grade book for\n%s!\n",
10             courseName );
11     }
12 } // end class GradeBook

```

Parameter `courseName` provides the additional information that the method requires to perform its task

Parameter `courseName`'s value is displayed as part of the output

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

21

```

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 }

```

Reads a String from the user

Fig. 3.5 | Creating a `GradeBook` object and passing a `String` to its `displayMessage` method. (Part 1 of 2.)

22

```

22     // call myGradeBook's displayMessage method
23     // and pass nameOfCourse as an argument
24     myGradeBook.displayMessage( nameOfCourse );
25 } // end main
26 } // end class GradeBookTest

```

Passes the value of nameOfCourse as the argument to method displayMessage

```

Please enter the course name:
CS101 Introduction to Java Programming

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

```

Fig. 3.5 | Creating a GradeBook object and passing a String to its displayMessage method. (Part 2 of 2.)

23

Declaring a Method with a Parameter (Cont.)

- Scanner method `nextLine`
 - Reads characters typed by the user until the newline character is encountered
 - Returns a `String` containing the characters up to, but not including, the newline
 - Press *Enter* to submit the string to the program.
 - Pressing *Enter* inserts a newline character at the end of the characters the user typed.
 - The newline character is discarded by `nextLine`.
- Scanner method `next`
 - Reads individual words
 - Reads characters until a white-space character is encountered, then returns a `String` (the white-space character is discarded).
 - Information after the first white-space character can be read by other statements that call the `Scanner`'s methods later in the program.

24

Declaring a Method with a Parameter (Cont.)

- More on Arguments and Parameters
 - The number of arguments in a method call must match the number of parameters in the parameter list of the method's declaration.
 - The argument types in the method call must be "consistent with" the types of the corresponding parameters in the method's declaration.

25

Declaring a Method with a Parameter (Cont.)

- Notes on import Declarations
 - Classes `System` and `String` are in package `java.lang`
 - Implicitly imported into every Java program
 - Can use the `java.lang` classes without explicitly importing them
 - Most classes you'll use in Java programs must be imported explicitly.
 - Classes that are compiled in the same directory on disk are in the same package—known as the [default package](#).
 - Classes in the same package are implicitly imported into the source-code files of other classes in the same package.
 - An `import` declaration is not required if you always refer to a class via its [fully qualified class name](#)
 - Package name followed by a dot (`.`) and the class name.

26

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

- Local variables
 - Variables declared in the body of a particular method.
 - When a method terminates, the values of its local variables are lost.
 - An object has attributes that are carried with the object as it's used in a program. Such attributes exist before a method is called on an object and after the method completes execution.

27

Instance Variables, *set Methods* and *get Methods (Cont.)*

- A class normally consists of one or more methods that manipulate the attributes that belong to a particular object of the class.
 - Attributes are represented as variables in a class declaration.
 - Called **fields**.
 - Declared inside a class declaration but outside the bodies of the class's method declarations.
- Instance variable
 - When each object of a class maintains its own copy of an attribute, the field is an instance variable
 - Each object (instance) of the class has a separate instance of the variable in memory.

28

```

1 // Fig. 3.7: GradeBook.java
2 // GradeBook class that contains a courseName instance variable
3 // and methods to set and get its value.
4
5 public class GradeBook
6 {
7     private String courseName; // course name for this GradeBook
8
9     // method to set the course name
10    public void setCourseName( String name )
11    {
12        courseName = name; // store the course name
13    } // end method setCourseName
14
15    // method to retrieve the course name
16    public String getCourseName()
17    {
18        return courseName;
19    } // end method getCourseName
20

```

Each GradeBook object maintains its own copy of instance variable courseName

Method allows client code to change the courseName

Method allows client code to obtain the courseName

Fig. 3.7 | GradeBook class that contains a courseName instance variable and methods to set and get its value. (Part 1 of 2.)

29

```

21 // display a welcome message to the GradeBook user
22 public void displayMessage()
23 {
24     // calls getCourseName to get the name of
25     // the course this GradeBook represents
26     System.out.printf( "Welcome to the grade book for\n%s!\n",
27         getCourseName() );
28 } // end method displayMessage
29 } // end class GradeBook

```

No parameter required: all methods in this class already know about instance variable courseName and the class's other methods

Good practice to access your instance variables via set or get methods

Fig. 3.7 | GradeBook class that contains a courseName instance variable and methods to set and get its value. (Part 2 of 2.)

30

Instance Variables, *set Methods* and *get Methods* (Cont.)

- Every instance (i.e., object) of a class contains one copy of each instance variable.
- Instance variables typically declared `private`.
 - `private` is an access modifier.
 - `private` variables and methods are accessible only to methods of the class in which they are declared.
- Declaring instance `private` is known as **data hiding** or **information hiding**.
- `private` variables are encapsulated (hidden) in the object and can be accessed only by methods of the object's class.
 - Prevents instance variables from being modified accidentally by a class in another part of the program.
 - *Set* and *get* methods used to access instance variables.

31

Instance Variables, *set Methods* and *get Methods* (Cont.)

- When a method that specifies a return type other than `void` completes its task, the method returns a result to its calling method.
- Method `setCourseName` and `getCourseName` each use variable `courseName` even though it was not declared in any of the methods.
 - Can use an instance variable of the class in each of the classes methods.
 - Exception to this is `static` methods (Chapter 8)
- The order in which methods are declared in a class does not determine when they are called at execution time.
- One method of a class can call another method of the same class by using just the method name.

32

Instance Variables, *set Methods* and *get Methods* (Cont.)

- Unlike local variables, which are not automatically initialized, every field has a **default initial value**—a value provided by Java when you do not specify the field's initial value.
- Fields are not required to be explicitly initialized before they are used in a program—unless they must be initialized to values other than their default values.
- The default value for a field of type `String` is `null`

33

```
1 // Fig. 3.8: GradeBookTest.java
2 // Creating and manipulating a GradeBook object.
3 import java.util.Scanner; // program uses Scanner
4
5 public class GradeBookTest
6 {
7     // main method begins program execution
8     public static void main( String[] args )
9     {
10         // create Scanner to obtain input from command window
11         Scanner input = new Scanner( System.in );
12
13         // create a GradeBook object and assign it to myGradeBook
14         GradeBook myGradeBook = new GradeBook();
15
16         // display initial value of courseName
17         System.out.printf( "Initial course name is: %s\n\n",
18             myGradeBook.getCourseName() );
19
20         // prompt for and read course name
21         System.out.println( "Please enter the course name:" );
22         String theName = input.nextLine(); // read a line of text
23         myGradeBook.setCourseName( theName ); // set the course name
```

Gets the value of the myGradeBook object's courseName instance variable

Sets the value of the courseName instance variable

Fig. 3.8 | Creating and manipulating a GradeBook object. (Part 1 of 2.)

34

```

24     System.out.println(); // outputs a blank line
25
26     // display welcome message after specifying course name
27     myGradeBook.displayMessage();
28 } // end main
29 } // end class GradeBookTest

```

Displays the GradeBook's message, including the value of the `courseName` instance variable

```

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!

```

Fig. 3.8 | Creating and manipulating a GradeBook object. (Part 2 of 2.)

35

Instance Variables, *set Methods* and *get Methods* (Cont.)

- *set* and *get* methods
 - A class's `private` fields can be manipulated only by the class's methods.
 - A **client of an object** calls the class's `public` methods to manipulate the `private` fields of an object of the class.
 - Classes often provide `public` methods to allow clients to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) `private` instance variables.
 - The names of these methods need not begin with *set* or *get*, but this naming convention is recommended.

36

Primitive Types vs. Reference Types

- Types are divided into primitive types and **reference types**.
- The primitive types are `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`.
- All nonprimitive types are reference types.
- A primitive-type variable can store exactly one value of its declared type at a time.
- Primitive-type instance variables are initialized by default—variables of types `byte`, `char`, `short`, `int`, `long`, `float` and `double` are initialized to 0, and variables of type `boolean` are initialized to `false`.
- You can specify your own initial value for a primitive-type variable by assigning the variable a value in its declaration.

37

Primitive Types vs. Reference Types

- Programs use variables of reference types (normally called **references**) to store the locations of objects in the computer's memory.
 - Such a variable is said to **refer to an object** in the program.
- Objects that are referenced may each contain many instance variables and methods.
- Reference-type instance variables are initialized by default to the value `null`
 - A reserved word that represents a “reference to nothing.”
- When using an object of another class, a reference to the object is required to **invoke** (i.e., call) its methods.
 - Also known as sending messages to an object.

38

Initializing Objects with Constructors

- When an object of a class is created, its instance variables are initialized by default.
- Each class can provide a constructor that initializes an object of a class when the object is created.
- Java requires a constructor call for *every* object that is created.
- Keyword `new` requests memory from the system to store an object, then calls the corresponding class's constructor to initialize the object.
- A constructor *must* have the same name as the class.

39

Initializing Objects with Constructors (Cont.)

- By default, the compiler provides a **default constructor** with no parameters in any class that does not explicitly include a constructor.
 - Instance variables are initialized to their default values.
- Can provide your own constructor to specify custom initialization for objects of your class.
- A constructor's parameter list specifies the data it requires to perform its task.
- Constructors cannot return values, so they cannot specify a return type.
- Normally, constructors are declared `public`.
- *If you declare any constructors for a class, the Java compiler will not create a default constructor for that class.*

40

```

1 // Fig. 3.10: GradeBook.java
2 // GradeBook class with a constructor to initialize the course name.
3
4 public class GradeBook
5 {
6     private String courseName; // course name for this GradeBook
7
8     // constructor initializes courseName with String argument
9     public GradeBook( String name )
10    {
11        courseName = name; // initializes courseName
12    } // end constructor
13
14    // method to set the course name
15    public void setCourseName( String name )
16    {
17        courseName = name; // store the course name
18    } // end method setCourseName
19

```

Constructor that initializes
courseName to the specified value

Fig. 3.10 | GradeBook class with a constructor to initialize the course name. (Part 1 of 2.)

41

```

20 // method to retrieve the course name
21 public String getCourseName()
22 {
23     return courseName;
24 } // end method getCourseName
25
26 // display a welcome message to the GradeBook user
27 public void displayMessage()
28 {
29     // this statement calls getCourseName to get the
30     // name of the course this GradeBook represents
31     System.out.printf( "Welcome to the grade book for\n%s!\n",
32         getCourseName() );
33 } // end method displayMessage
34 } // end class GradeBook

```

Fig. 3.10 | GradeBook class with a constructor to initialize the course name. (Part 2 of 2.)

42

```

1 // Fig. 3.11: GradeBookTest.java
2 // GradeBook constructor used to specify the course name at the
3 // time each GradeBook object is created.
4
5 public class GradeBookTest
6 {
7     // main method begins program execution
8     public static void main( String[] args )
9     {
10         // create GradeBook object
11         GradeBook gradeBook1 = new GradeBook(
12             "CS101 Introduction to Java Programming" );
13         GradeBook gradeBook2 = new GradeBook(
14             "CS102 Data Structures in Java" );
15
16         // display initial value of courseName for each GradeBook
17         System.out.printf( "gradeBook1 course name is: %s\n",
18             gradeBook1.getCourseName() );
19         System.out.printf( "gradeBook2 course name is: %s\n",
20             gradeBook2.getCourseName() );
21     } // end main
22 } // end class GradeBookTest

```

Class instance creation expression initializes the `GradeBook` and returns a reference that is assigned to variable `gradeBook1`

Class instance creation expression initializes the `GradeBook` and returns a reference that is assigned to variable `gradeBook1`

Fig. 3.11 | `GradeBook` constructor used to specify the course name at the time each `GradeBook` object is created. (Part 1 of 2.)

43

```

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

```

Fig. 3.11 | `GradeBook` constructor used to specify the course name at the time each `GradeBook` object is created. (Part 2 of 2.)

44