Chapter 4
Defining Classes I

Prof. Choonhwa Lee

Dept. of Computer Science and Engineering
Hanyang University

Introduction

- Classes are the most important language feature that makes *object-oriented programming* (*OOP*) possible
- Programming in Java consists of defining a number of classes
  - Every program is a class
  - All helping software consists of classes
  - All programmer-defined types are classes
- Classes are central to Java

Class Definitions

- You already know how to use classes and the objects created from them, and how to invoke their methods
  - For example, you have already been using the predefined `String` and `Scanner` classes
- Now you will learn how to define your own classes and their methods, and how to create your own objects from them

A Class Is a Type

- A class is a special kind of programmer-defined type, and variables can be declared of a class type
- A value of a class type is called an object or *an instance of the class*
  - If A is a class, then the phrases "bla is of type A," "bla is an object of the class A," and "bla is an instance of the class A" mean the same thing
- A class determines the types of data that an object can contain, as well as the actions it can perform

The Contents of a Class Definition

- A class definition specifies the data items and methods that all of its objects will have
- These data items and methods are sometimes called *members* of the object
- Data items are called *fields* or *instance variables*
- Instance variable declarations and method definitions can be placed in any order within the class definition

Display 4.1 A Simple Class

Display 4.1 A Simple Class

```
1  public class DateFirstTry
2  {
3      public String month;
4      public int day;
5      public int year; //a four digit number.

6      public void writeOutput()
7      {
8          System.out.println(month + " " + day + ", " + year);
9      }
10 }
```

This class definition goes in a file named **DateFirstTry.java.**

Later in this chapter, we will see that these three **public** *modifiers should be replaced with* **private.**

This class definition (program)
DateFirstTryDemo.java.

```
1  public class DateFirstTryDemo
2  {
3     public static void main(String[] args)
4     {
5         DateFirstTry date1, date2;
6         date1 = new DateFirstTry();
7         date2 = new DateFirstTry();
8         date1.month = "December";
9         date1.day = 31;
10        date1.year = 2007;
11        System.out.println("date1:");
12        date1.writeOutput();

13        date2.month = "July";
14        date2.day = 4;
15        date2.year = 1776;
16        System.out.println("date2:");
17        date2.writeOutput();
18    }
19 }
```

The **new** Operator

- An object of a class is named or declared by a variable of the class type:

```
ClassName  classVar;
```

- The **new** operator must then be used to create the object and associate it with its variable name:

```
classVar = new ClassName();
```

- These can be combined as follows:

```
ClassName classVar = new ClassName();
```

Instance Variables and Methods

- Instance variables can be defined as in the following two examples
  - Note the **`public`** modifier (for now):

```
public String  instanceVar1;
public int  instanceVar2;
```

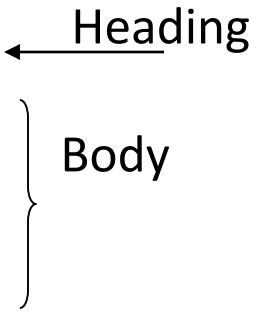
- In order to refer to a particular instance variable, preface it with its object name as follows:

```
objectName.instanceVar1
objectName.instanceVar2
```

Instance Variables and Methods

- Method definitions are divided into two parts: a *heading* and a *method body:*

```
public void myMethod()
{
    code to perform some action
    and/or compute a value
}
```



- Methods are invoked using the name of the calling object and the method name as follows:

```
classVar.myMethod();
```

- Invoking a method is equivalent to executing the method body

More About Methods

- There are two kinds of methods:
  - Methods that compute and return a value
  - Methods that perform an action
    - This type of method does not return a value, and is called a **`void`** method
- Each type of method differs slightly in how it is defined as well as how it is (usually) invoked

More About Methods

- A method that returns a value must specify the type of that value in its heading:

```
public typeReturned methodName(paramList)
```

- A **`void`** method uses the keyword **`void`** in its heading to show that it does not return a value :

```
public void methodName(paramList)
```

`main` is a `void` Method

- A program in Java is just a class that has a **`main`** method
- When you give a command to run a Java program, the run-time system invokes the method **`main`**
- Note that **`main`** is a **`void`** method, as indicated by its heading:

```
public static void main(String[] args)
```

Local Variables

- A variable declared within a method definition is called a *local variable*
  - All variables declared in the `main` method are local variables
  - All method parameters are local variables
- If two methods each have a local variable of the same name, they are still two entirely different variables

Global Variables

- Some programming languages include another kind of variable called a *global* variable
- The Java language does **not** have global variables

Blocks

- A *block* is another name for a compound statement, that is, a set of Java statements enclosed in braces, { }
- A variable declared within a block is local to that block, and cannot be used outside the block
- Once a variable has been declared within a block, its name cannot be used for anything else within the same method definition

Parameters of a Primitive Type

- When a method is invoked, the appropriate values must be passed to the method in the form of *arguments*
  - Arguments are also called *actual parameters*
- The number and order of the arguments must exactly match that of the **(formal) parameter** list
- The type of each argument must be compatible with the type of the corresponding parameter

```
int a=1,b=2,c=3;
double result = myMethod(a,b,c);

public double myMethod(int p1, int p2, double p3)
```

The `this` Parameter

- All instance variables are understood to have `<the calling object>.` in front of them
- If an explicit name for the calling object is needed, the keyword `this` can be used
  - `myInstanceVariable` always means and is always interchangeable with `this.myInstanceVariable`

The `this` Parameter

- In Display 4.1

```
public class DateFirstTry
{
    public String month;
    public int day;
    public int year;

    public void writeOutput()
    {
       System.out.println(month + " " + day + ", " + year);
    }
   …
}
```

➔ public void writeOuput()

```
{
    System.out.println(<the calling object>.month + " " + <the calling object>.day + ", " + <the calling
    object>.year);
}
```

➔ public void writeOuput()

```
{
    System.out.println(this.month + " " + this.day + ", " + this.year);
}
```

The `this` Parameter

- **`this`** ***must*** be used if a parameter or other local variable with the same name is used in the method
  - Otherwise, all instances of the variable name will be interpreted as local

```
int someVariable = this.someVariable
```

local **instance**

The `this` Parameter

```
public void setDate(int month, int day, int year)
{
    this.month = month;
    this.day = day;
    this.year = year;
}
```

The methods `equals` and `toString`

- Java expects certain methods, such as **`equals`** and **`toString`**, to be in all, or almost all, classes
- The purpose of **`equals`**, a **`boolean`** valued method, is to compare two objects of the class to see if they satisfy the notion of "being equal"
  - Note: You cannot use **`==`** to compare objects

```
public boolean equals(ClassName objectName)
```

- The purpose of the **`toString`** method is to return a **`String`** value that represents the data in the object

```
public String toString()
```

Information Hiding and Encapsulation

- *Information hiding* is the practice of separating how to use a class from the details of its implementation
  - *Abstraction* is another term used to express the concept of discarding details in order to avoid information overload
- *Encapsulation* means that the data and methods of a class are combined into a single unit (i.e., a class object), which hides the implementation details
  - Knowing the details is unnecessary because interaction with the object occurs via a well-defined and simple interface
  - In Java, hiding details is done by marking them **`private`**

A Couple of Important Acronyms: API and ADT

- The API or *application programming interface* for a class is a description of how to use the class
  - A programmer need only read the API in order to use a well designed class
- An ADT or *abstract data type* is a data type that is written using good information-hiding techniques

`public` and `private` Modifiers

- The modifier **`public`** means that there are no restrictions on where an instance variable or method can be used
- The modifier **`private`** means that an instance variable or method cannot be accessed by name outside of the class
- It is considered good programming practice to make **all** instance variables **`private`**
- Most methods are **`public`**, and thus provide controlled access to the object
- Usually, methods are **`private`** only if used as helping methods for other methods in the class

`public` and `private` Modifiers

- The following would produce a compiler error message if used with DateSecondTry() in Display 4.2

```
public class DemoOfDateSecondTry
{
        public static void main(String[] args)
        {
            DateSecondTry date = new DateSecondTry();
            date.month = "January";
            date.day = 1;
            date.year = 2006;
            …
         }
         …
}
```

Display 4.2 A Class with More Methods (part 1 of 2)

```
1   import java.util.Scanner;

2   public class DateSecondTry
3   {
4       private String month;
5       private int day;
6       private int year; //a four digit number.

7       public void writeOutput()
8       {
9           System.out.println(month + " " + day + ", " + year);
10      }
```

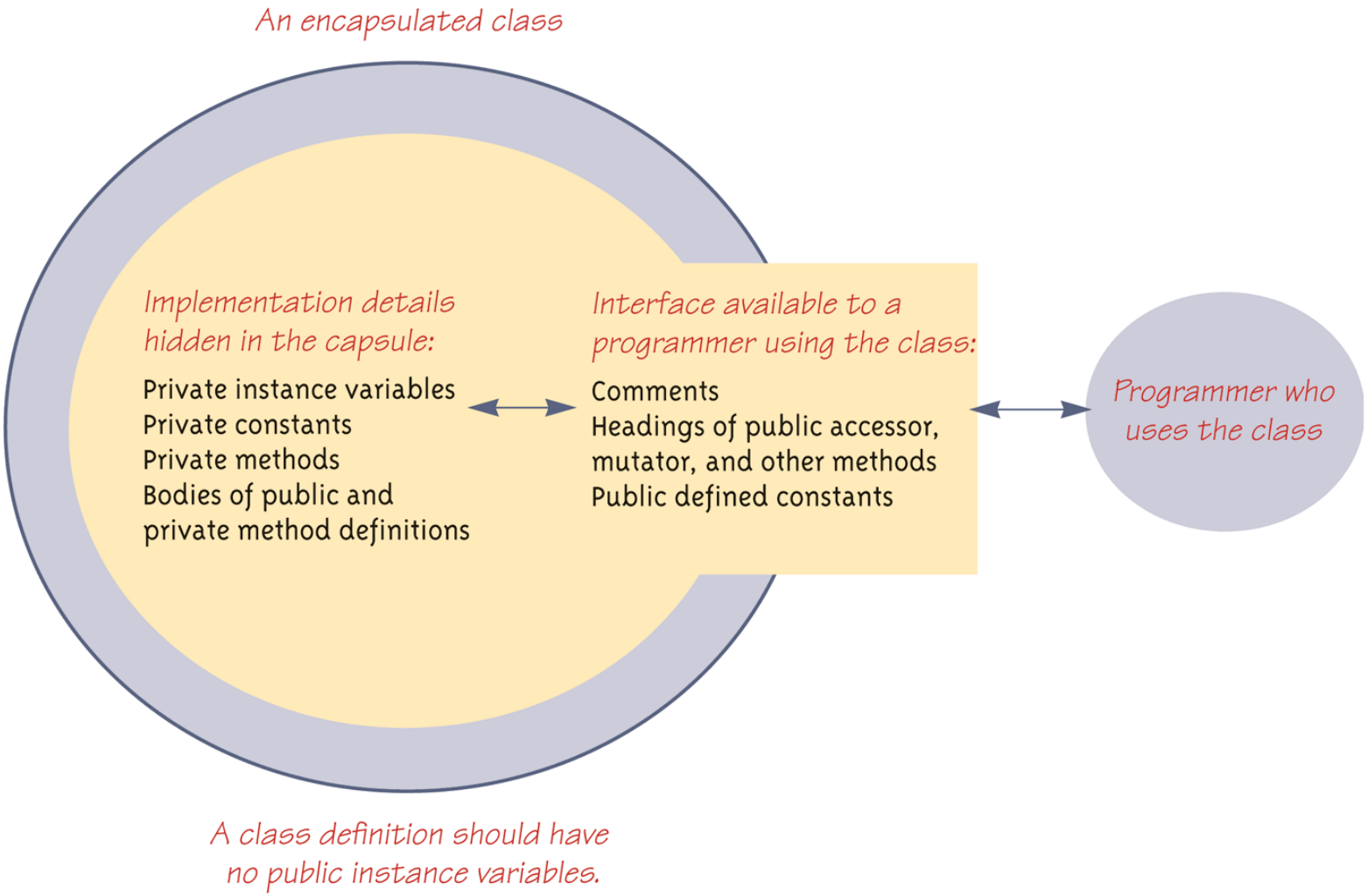
The significance of the modifier **private** *is discussed in the subsection "***public** *and* **private** *Modifiers" in Section 4.2 a bit later in this chapter.*

Accessor and Mutator Methods

- *Accessor* methods allow the programmer to obtain the value of an object's instance variables
  - The data can be accessed but not changed
  - The name of an accessor method typically starts with the word **`get`**
- *Mutator* methods allow the programmer to change the value of an object's instance variables in a controlled manner
  - Incoming data is typically tested and/or filtered
  - The name of a mutator method typically starts with the word **`set`**

Encapsulation

Display 4.10 Encapsulation



A Class Has Access to Private Members of All Objects of the Class

- Within the definition of a class, private members of **any** object of the class can be accessed, not just private members of the calling object

```
public  boolean equals(DateSecondTry otherDate)
{
        return ( (month.equalsIngoreCase(otherDate.month)
                && (day == otherDate.day) && (year == otherDate.year));
}
```

Display 4.2 A Class with More Methods (part 1 of 2)

```
1   import java.util.Scanner;

2   public class DateSecondTry
3   {
4       private String month;
5       private int day;
6       private int year; //a four digit number.

7       public void writeOutput()
8       {
9           System.out.println(month + " " + day + ", " + year);
10      }
```

The significance of the modifier **private** *is discussed in the subsection "***public** *and* **private** *Modifiers" in Section 4.2 a bit later in this chapter.*

Overloading

- *Overloading* is when two or more methods *in the same class* have the same method name

```
public void setDate(int month, int day, int year);
public void setDate(String month, int day, int year);
public void setDate(int year);
```

- To be valid, any two definitions of the method name must have different *signatures*
  - A signature consists of the name of a method together with its parameter list
  - Differing signatures must have different numbers and/or types of parameters

Overloading and Automatic Type Conversion

- If Java cannot find a method signature that exactly matches a method invocation, it will try to use automatic type conversion
- The interaction of overloading and automatic type conversion can have unintended results
- In some cases of overloading, because of automatic type conversion, a single method invocation can be resolved in multiple ways
  - Ambiguous method invocations will produce an error in Java

Pitfall: You Can Not Overload Based on the Type Returned

- The signature of a method only includes the method name and its parameter types
  - The signature does **not** include the type returned
- Java does not permit methods with the same name and different return types in the same class

Constructors

- A *constructor* is a special kind of method that is designed to initialize the instance variables for an object:

  ```
  public ClassName(anyParameters){code}
  ```

  – A constructor must have the same name as the class
  – A constructor has no type returned, not even `void`
  – Constructors are typically overloaded

Display 4.13 A Class with Constructor

```
import java.util.Scanner;

public class Date
{
    private String month;
    private int day;
    private int year; //a four digit number.

    public Date()
    {
        month = "January";
        day = 1;
        year = 1000;
    }

    public Date(int monthInt, int day, int year)
    {
        setDate(monthInt, day, year);
    }

    public Date(String monthString, int day, int year)
    {
        setDate(monthString, day, year);
    }

    public Date(int year)
    {
        setDate(1, 1, year);
    }

    public Date(Date aDate)
    {
        if (aDate == null) //Not a real date.
        {
            System.out.println("Fatal Error.");
            System.exit(0);
        }

        month = aDate.month;
        day = aDate.day;
        year = aDate.year;
    }
```

This is our final definition of a class whose objects are dates.

No-argument constructor

You can invoke another method inside a constructor definition.

A constructor usually initializes all instance variables, even if there is not a corresponding parameter.

We will have more to say about this constructor in Chapter 5. Although you have had enough material to use this constructor, you need not worry about it until Section 5.3 of Chapter 5.

Constructors

- A constructor is called when an object of the class is created using **`new`**

```
ClassName objectName = new ClassName(anyArgs);
```

  - The name of the constructor and its parenthesized list of arguments (if any) must follow the **`new`** operator
  - This is the **only** valid way to invoke a constructor: a constructor cannot be invoked like an ordinary method

You Can Invoke Another Method in a Constructor

- The first action taken by a constructor is to create an object with instance variables
- Therefore, it is legal to invoke another method within the definition of a constructor, since it has the newly created object as its calling object
  - For example, mutator methods can be used to set the values of the instance variables
  - It is even possible for one constructor to invoke another

Include a No-Argument Constructor

- If you do not include any constructors in your class, Java will automatically create a *default* or *no-argument* constructor that takes no arguments, performs no initializations, but allows the object to be created
  - So, the following is legal:

    ```
    MyClass myObject = new MyClass();
    ```

- If you include even one constructor in your class, Java will not provide this default constructor
  - So, the following is illegal:

    ```
    YourClass yourObject = new YourClass();
    ```

- If you include any constructors in your class, be sure to provide your own no-argument constructor as well

Default Variable Initializations

- Instance variables are automatically initialized in Java
  - `boolean` types are initialized to false
  - Other primitives are initialized to the zero of their type
  - Class types are initialized to `null`
- However, it is a better practice to explicitly initialize instance variables in a constructor
- Note: Local variables are not automatically initialized

this Program

```
public class ThisTestClass {
        public int i;

        ThisTestClass(int i)
        {
                this.i = i;
        }

        public void writeOutput(ThisTestClass that)
        {
                System.out.println("i is " + i + ", this i is " + this.i + ", and that i is " + that.i);
        }

        public static void main(String[] args)
        {
                ThisTestClass a = new ThisTestClass(1);
                ThisTestClass b = new ThisTestClass(10);

                a.writeOutput(b);
                b.writeOutput(a);
        }
}
```

```
i is 1, this i is 1, and that i is 10
i is 10, this i is 10, and that i is 1
```