Advanced Programming

Defining Classes II
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.

- **private** methods – can only be used in the definition of another method of the same class.
public class DateFirstTry
{
    public String month;
    public int day; //a four digit number.
    public int year;

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

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

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

DateSecondTry date = new DateSecondTry();
date.month = "January";
date.day = 1;
date.year = 2006;
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`
Updated “Date” class

```java
import java.util.Scanner;

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

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

    public void readInput() {
        // Note that this version of read enter the month as an int string. In this class, a month user, but is a string inside t.
    }
```
boolean tryAgain = true;
Scanner keyboard = new Scanner(System.in);
while (tryAgain)
{
    System.out.println("Enter month, day, and year");
    System.out.println("as three integers:");
    System.out.println("do not use commas or other punctuations.");
    int monthInput = keyboard.nextInt();
    int dayInput = keyboard.nextInt();
    int yearInput = keyboard.nextInt();
    if (dateOK(monthInput, dayInput, yearInput))
    {
        setDate(monthInput, dayInput, yearInput);
        tryAgain = false;
    }
    else
    {
        System.out.println("Illegal date. Reenter input.");
    }
}
public void setDate(int month, int day, int year) {
    if (dateOK(month, day, year)) {
        this.month = monthString(month);
        this.day = day;
        this.year = year;
    } else {
        System.out.println("Fatal Error");
        System.exit(0);
    }
}

public void setMonth(int monthNumber) {
    if ((monthNumber <= 0) || (monthNumber > 12)) {
        System.out.println("Fatal Error");
        System.exit(0);
    } else {
        month = monthString(monthNumber);
    }
}
public void setDay(int day) {
    if ((day <= 0) || (day > 31)) {
        System.out.println("Fatal Error");
        System.exit(0);
    } else
        this.day = day;
}

public void setYear(int year) {
    if ( (year < 1000) || (year > 9999) ) {
        System.out.println("Fatal Error");
        System.exit(0);
    } else
        this.year = year;
}
public boolean equals(DateFifthTry otherDate) {
    return (month.equalsIgnoreCase(otherDate.month)) && (day == otherDate.day) && (year == otherDate.year);
}

Within the definition of DateFifthTry, you can directly access private instance variables of any object of type DateFifthTry.

public boolean precedes(DateFifthTry otherDate) {
    return (year < otherDate.year) || (year == otherDate.year && getMonth() < otherDate.getMonth()) || (year == otherDate.year && month.equals(otherDate.month) && day < otherDate.day);
}

Within the definition of DateFifthTry, you can directly access private instance variables of any object of type DateFifthTry.

The definitions of the following methods are the same as in Display 4.2 and Display 4.7: getMonth, getDay, getYear, and toString.

private boolean dateOK(int monthInt, int dayInt, int yearInt) {
    return (monthInt >= 1) && (monthInt <= 12) && (dayInt >= 1) && (dayInt <= 31) && (yearInt >= 1000) && (yearInt <= 9999);
private String monthString(int monthNumber)
{
    switch (monthNumber)
    {
    case 1:
        return "January";
    case 2:
        return "February";
    case 3:
        return "March";
    case 4:
        return "April";
    case 5:
        return "May";
    case 6:
        return "June";
    case 7:
        return "July";
    case 8:
        return "August";
    case 9:
        return "September";
    case 10:
        return "October";
    case 11:
        return "November";
    case 12:
        return "December";
    default:
        System.out.println("Fatal Error");
        System.exit(0);
        return "Error"; //to keep the compiler happy
    }
}
An encapsulated class

Implementation details hidden in the capsule:
- Private instance variables
- Private constants
- Private methods
- Bodies of public and private method definitions

Interface available to a programmer using the class:
- Comments
- Headings of public accessor, mutator, and other methods
- Public defined constants

A class definition should have no public instance variables.
A Class Has Access to Private Members of All Objects of the Class

```java
public boolean equals(DateFifthTry otherDate)
{
    return (month.equalsIgnoreCase(otherDate.month))
    && (day == otherDate.day) && (year == otherDate.year);
}
```
Overloading (1)

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

• 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 list of its parameter types
  
  – Differing signatures must have different numbers and/or types of parameters
Overloading (2)

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

setDate(int, int, int)
setDate(String, int, int)
setDate(int)
```
Overloading Example

```java
import java.util.Scanner;

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

    public void setDate(int monthInt, int day, int year) {
        if (dateOK(monthInt, day, year)) {
            this.month = monthString(monthInt);
            this.day = day;
            this.year = year;
        } else {
            System.out.println("Fatal Error");
            System.exit(0);
        }
    }
}
```

There are three different methods named `setDate`
```java
public void setDate(String monthString, int day, int year)
{
    if (dateOK(monthString, day, year))
    {
        this.month = monthString;
        this.day = day;
        this.year = year;
    }
    else
    {
        System.out.println("Fatal Error");
        System.exit(0);
    }
}

public void setDate(int year)
{
    setDate(1, 1, year);
}
```

Two different methods named `setDate`. 

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
public class SampleClass
{
    public void doSomething(double n1, int n2)
        .
        .
    public void doSomething(int n1, double n2)
        .
        .

    aSampleObject.doSomething(5, 10);
    aSampleObject.doSomething(5.0, 10);
    aSampleObject.doSomething(5, 10.0);
public class OverloadingDemo
{

    public static void main(String[] args)
    {
        DateSixthTry date1 = new DateSixthTry(),
        date2 = new DateSixthTry(),
        date3 = new DateSixthTry();

        date1.setDate(1, 2, 2008);
        date2.setDate("February", 2, 2008);
        date3.setDate(2008);

        System.out.println(date1);
        System.out.println(date2);
        System.out.println(date3);
    }
}
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

```java
public class SampleClass2 {
    public int computeSomething(int n)
        .
        .
        .
    public double computeSomething(int n)
        .
        .
        .

double answer = anObject.computeSomething(10);
```
You Can Not Overload Operators in Java

• Although many programming languages, such as C++, allow you to overload operators (+, -, etc.), Java does not permit this
Constructors

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

```java
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
  • Authomatic
Constructors: Example

```java
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);
    }

    private void setDate(int monthInt, int day, int year) {
        // Method to set the date
    }
}
```

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

No-argument constructor

You can invoke another method inside a constructor definition.
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;
}

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 (cont.)

• 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

```
birthday.Date("January", 27, 1756); //Illegal!
```

• If a constructor is invoked again (using `new`), the first object is discarded and an entirely new object is created
  – If you need to change the values of instance variables of the object, use mutator methods instead
Constructors (cont.)

```javascript
Date birthday = new Date("December", 16, 1770);
    :
    :
    birthday = new Date("January", 27, 1756);
    
    birthday.setDate("January", 27, 1756);
```
public class ConstructorsDemo
{
    public static void main(String[] args)
    {
        Date date1 = new Date("December", 16, 1770),
        date2 = new Date(1, 27, 1756),
        date3 = new Date(1882),
        date4 = new Date();

        System.out.println("Whose birthday is " + date1 + "?");
        System.out.println("Whose birthday is " + date2 + "?");
        System.out.println("Whose birthday is " + date3 + "?");
        System.out.println("The default date is " + date4 + ".");
    }
}
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
A Constructor Has a **this** Parameter

- Like any ordinary method, every constructor has a **this** parameter.
- The **this** parameter can be used explicitly, but is more often understood to be there than written down.
- The first action taken by a constructor is to automatically create an object with instance variables.
- Then within the definition of a constructor, the **this** parameter refers to the object created by the constructor.
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
• If you include even one constructor in your class, Java will not provide this default constructor
• 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
The **StringTokenizer** Class

- The **StringTokenizer** class is used to recover the words or *tokens* in a multi-word **String**
  - You can use whitespace characters to separate each token, or you can specify the characters you wish to use as separators
  - In order to use the **StringTokenizer** class, be sure to include the following at the start of the file:
    ```java
    import java.util.StringTokenizer;
    ```
Some Methods in the `StringTokenizer` Class (Part 1 of 2)

**Display 4.17 Some Methods in the Class StringTokenizer**

The class `StringTokenizer` is in the `java.util` package.

```java
public StringTokenizer(String theString)
Constructor for a tokenizer that will use whitespace characters as separators when finding tokens in theString.

public StringTokenizer(String theString, String delimiters)
Constructor for a tokenizer that will use the characters in the string delimiters as separators when finding tokens in theString.

public boolean hasMoreTokens()
Tests whether there are more tokens available from this tokenizer’s string. When used in conjunction with `nextToken`, it returns `true` as long as `nextToken` has not yet returned all the tokens in the string; returns `false` otherwise.

(continued)
Some Methods in the **StringTokenizer** Class (Part 2 of 2)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public String nextToken()</code></td>
<td>Returns the next token from this tokenizer’s string. (Throws <code>NoSuchElementException</code> if there are no more tokens to return.)&lt;sup&gt;5&lt;/sup&gt;</td>
</tr>
<tr>
<td><code>public String nextToken(String delimiters)</code></td>
<td>First changes the delimiter characters to those in the string <code>delimiters</code>. Then returns the next token from this tokenizer’s string. After the invocation is completed, the delimiter characters are those in the string <code>delimiters</code>. (Throws <code>NoSuchElementException</code> if there are no more tokens to return. Throws <code>NullPointerException</code> if <code>delimiters</code> is null.)&lt;sup&gt;5&lt;/sup&gt;</td>
</tr>
<tr>
<td><code>public int countTokens()</code></td>
<td>Returns the number of tokens remaining to be returned by <code>nextToken</code>.</td>
</tr>
</tbody>
</table>
Material Covered

• Chapter 4: pages 233 – 272
  – Absolute Java, 4\textsuperscript{th} edition, Savitch