HST 952

Computing for Biomedical Scientists

Lecture 6
Designing Methods: Top-Down Design

• In pseudocode, write a list of subtasks that the method must perform
• If you can easily write Java statements for a subtask, you are finished with that subtask
• If you cannot easily write Java statements for a subtask, treat it as a new problem and break it up into a list of subtasks
• Eventually, all of the subtasks will be small enough to easily design and code
• Solutions to subtasks might be implemented as private helper methods
• Top-down design is also known as divide-and-conquer or stepwise refinement
Designing Methods: Top-Down Design

- Person class has attributes of type String and GregorianCalendar corresponding to a person’s first name, last name, and date of birth: firstName, lastName, and dateOfBirth
- Create a new method:
  ```java
double ageOfPerson()
```
for the Person class that returns the approximate age (with respect to year and month of birth) of a person. Approximate in this case means that if a person was born in September of 1965 and the current month and year are September 2002, the age returned should be 37.0 (the actual day of the month on which the person was born is ignored).

What tasks should this method perform?
Designing Methods: Top-Down Design

Some tasks this method should perform:

– find out the current year
– find out the current month
– find out the birth year
– find out the birth month
– find out the age using these values

- subtract the birth year and month from the current year and month
- return the value obtained as the age
The `ageOfPerson()` method

```java
public double ageOfPerson()
{
    // The GregorianCalendar class default constructor creates
    // a new date and time corresponding to the date and time
    // the program in which it is called is executed
    GregorianCalendar today = new GregorianCalendar();
    // Calendar is a parent class to GregorianCalendar
    // YEAR is a static named constant of the Calendar class
    int thisYear = today.get(Calendar.YEAR);
    int birthYear = dateOfBirth.get(Calendar.YEAR);
    // Java Gregorian Calendar month is zero based -- Jan==0
    int thisMonth = today.get(Calendar.MONTH);
    int birthMonth = dateOfBirth.get(Calendar.MONTH);
    double age = (thisYear - birthYear) + ((thisMonth - birthMonth)/12.0);
    return age;
}
```
public double ageOfPerson()
{
    // The GregorianCalendar class default constructor creates
    // a new date and time corresponding to the date and time
    // the program in which it is called is executed
    GregorianCalendar today = new GregorianCalendar();
    // Calendar is a parent class to GregorianCalendar
    // YEAR is a static named constant of the Calendar class
    // Java Gregorian Calendar month is zero based -- Jan==0
    int thisYear = today.get(Calendar.YEAR);
    int birthYear = dateOfBirth.get(Calendar.YEAR);
    int thisMonth = today.get(Calendar.MONTH);
    int birthMonth = dateOfBirth.get(Calendar.MONTH);
    double age = (thisYear - birthYear) +
                 ((thisMonth - birthMonth)/12.0);
    return(age);
}
**Wrapper Classes**

- Used to wrap primitive types in a class structure
- All primitive types have an equivalent class
- The class includes useful constants and static methods, including one to convert back to the primitive type

<table>
<thead>
<tr>
<th>Primitive type</th>
<th>Class type</th>
<th>Method to convert back</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer</td>
<td>intValue()</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
<td>longValue()</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
<td>floatValue()</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
<td>doubleValue()</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
<td>charValue()</td>
</tr>
</tbody>
</table>
Wrapper class example: Integer

- Declare an `Integer` class variable:
  ```java
  Integer n = new Integer();
  ```

- Convert the value of an `Integer` variable to its primitive type, `int`:
  ```java
  int i = n.intValue();
  //method intValue() returns an int
  ```

- Some useful `Integer` constants:
  - `Integer.MAX_VALUE` - the maximum integer value the computer can represent
  - `Integer.MIN_VALUE` - the smallest integer value the computer can represent
Wrapper class example: Integer

• Some useful `Integer` methods:
  – `Integer.parseInt("123")` to convert a string of numerals to an integer
  – `Integer.toString(123)` to convert an `Integer` to a `String`

• The other wrapper classes have similar constants and functions
Wrapper classes

There are some important differences in the code to use wrapper classes and that for the primitive types.

**Wrapper Class**
- variables contain the *address* of the object
- variable declaration example:
  ```java
  Integer n;
  ```
- variable declaration & init:
  ```java
  Integer n = new Integer(0);
  ```
- assignment:
  ```java
  n = new Integer(5);
  ```

**Primitive Type**
- variables contain a value
- variable declaration example:
  ```java
  int n;
  ```
- variable declaration & init.:
  ```java
  int n = 0;
  ```
- assignment:
  ```java
  n = 99;
  ```
Outline

- Arrays continued
- Packages
- Inheritance
Partially Filled Arrays

- Sometimes only part of an array has been filled with data.
- Array elements always contain something, whether you have written to them or not.
  - Elements which have not been written to/filled contain unknown (garbage) data so you should avoid reading them.
- There is no automatic mechanism to detect how many elements have been filled - you need to keep track...
Example of a Partially Filled Array

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Anne</td>
<td>Stephen</td>
<td>Rahul</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `countOfEntries` has a value of **3**.
- `entry.length` has a value of **5**.
- `countOfEntries - 1` has a value of **2**.
- Garbage values are indicated.
Multidimensional Arrays

- Arrays with more than one index
  - number of dimensions = number of indexes
- Arrays with more than two dimensions are a simple extension of two-dimensional (2-D) arrays
- A 2-D array corresponds to a table or grid
  - one dimension is the row
  - the other dimension is the column
  - cell: an intersection of a row and column
  - an array element corresponds to a cell in the table
Multidimensional Arrays

Example of usage:

Store the different possible ending balances corresponding to $1000 saved at 6 different interest rates over a period of 10 years
### Table as a 2-D Array

<table>
<thead>
<tr>
<th>Indexes</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$1050</td>
<td>$1055</td>
<td>$1060</td>
<td>$1065</td>
<td>$1070</td>
<td>$1075</td>
</tr>
<tr>
<td>1</td>
<td>$1103</td>
<td>$1113</td>
<td>$1124</td>
<td>$1134</td>
<td>$1145</td>
<td>$1156</td>
</tr>
<tr>
<td>2</td>
<td>$1158</td>
<td>$1174</td>
<td>$1191</td>
<td>$1208</td>
<td>$1225</td>
<td>$1242</td>
</tr>
<tr>
<td>3</td>
<td>$1216</td>
<td>$1239</td>
<td>$1262</td>
<td>$1286</td>
<td>$1311</td>
<td>$1335</td>
</tr>
<tr>
<td>4</td>
<td>$1276</td>
<td>$1307</td>
<td>$1338</td>
<td>$1370</td>
<td>$1403</td>
<td>$1436</td>
</tr>
</tbody>
</table>

- **Generalizing to two indexes:** [row][column]
- **First dimension:** row index
- **Second dimension:** column index
- **Cell contains balance for the year/row and percentage/column**
- **All indexes use zero-numbering**
  - Balance[3][4] = cell in 4th row (year = 4) and 5th column (7.00%)
  - Balance[3][4] = $1311 (shown in yellow)
Java Code to Create a 2-D Array

• Syntax for 2-D arrays is similar to 1-D arrays
• Declare a 2-D array of ints named table
  – the array table should have ten rows and six columns
  ```java
  int[][][] table = new int[10][6];
  ```
Calculating the Cell Values

Each array element corresponds to the balance for a specific number of years and a specific interest rate (assuming a starting balance of $1000):

\[
balance(\text{start-balance, years, rate}) = (\text{start-balance}) \times (1 + rate)^{\text{years}}
\]

The repeated multiplication by \((1 + rate)\) can be done in a `for` loop that repeats `years` times.

```java
public static int balance(double startBalance, int years, double rate) {
    double runningBalance = startBalance;
    int count;
    for (count = 0; count < years; count++)
        runningBalance = runningBalance*(1 + rate/100);
    return (int) (Math.round(runningBalance));
}
```
Processing a 2-D Array: for Loops Nested 2-Deep

• Arrays and for loops are a natural fit
• To process all elements of an $n$-dimensional array nest $n$ for loops
  – each loop has its own counter that corresponds to an index
Processing a 2-D Array: for Loops Nested 2-Deep

• For example: calculate and enter balances in interest table (10 rows and 6 columns)
  – inner loop repeats 6 times (six rates) for every outer loop iteration
  – the outer loop repeats 10 times (10 different values of years)
  – so the inner repeats 10 x 6 = 60 times = # cells in table

```java
int[][] table = new int[10][6];
int row, column;
for (row = 0; row < 10; row++)
    for (column = 0; column < 6; column++)
        table[row][column] = balance(1000.00, row + 1, (5 + 0.5*column));
```
Multidimensional Array Parameters and Returned Values

- Methods may have multi-dimensional array parameters.
- Methods may return a multi-dimensional array as the value returned.
- The situation is similar to 1-D arrays, but with more brackets.
- Example: a 2-D int array as a method argument.
Multidimensional Array Parameters and Returned Values

Number of rows of a 2D array is: `nameOfArray.length`
Number of columns for each row is: `nameOfArray[row-index].length`

```java
public static void showTable(int[][] displayArray) {
    int row, column;
    for (row = 0; row < displayArray.length; row++) {
        System.out.print((row + 1) + "   ");
        for (column = 0; column < displayArray[row].length; column++)
            System.out.print("$" + displayArray[row][column] + "  ");
        System.out.println();
    }
}
```

Notice how the number of rows is obtained
Notice how the number of columns is obtained
Ragged Arrays

• Ragged arrays have rows of unequal length
  – each row has a different number of columns, or entries
• Ragged arrays are allowed in Java
• Example: create a 2-D int array named b with 5 elements in the first row, 7 in the second row, and 4 in the third row:

```java
int[][][] b;
b = new int[3][];
b[0] = new int[5];
b[1] = new int[7];
b[2] = new int[4];
```
Packages

• A way of grouping and naming a collection of related classes
  – the classes in a package serve as a *library* of classes
  – they do not have to be in the same directory as the code for your program

• The **first line** of each class in the package must be the keyword `package` followed by the name of the package
Packages

Example -- a group of related classes that represent shapes and methods for drawing them:

```java
package graphics;
public class Circle extends Graphic {
    ...
} // in Circle.java

package graphics;
public class Rectangle extends Graphic {
    ...
} // in Rectangle.java

package graphics;
public class Ellipse extends Graphic {
    ...
} // in Ellipse.java
```
Packages

• To use classes from a package in program source code, can put an import statement at the start of the file, e.g.:
  
  ```java
  import graphics.*;
  ```

  – note the ".*" notation, "*" is a wild-card that matches all class names in the graphics package; in our example, it is shorthand for graphics.Circle, graphics.Rectangle, and graphics.Ellipse

• Class descriptions with no package statement are automatically placed in a default package (a package with no name)
Packages

• Use lowercase letters for the package name
• By using packages if we write a new class description that has the same name as a built-in Java class, we can avoid problems
• java.awt has a Rectangle class
  – to refer to it by its full name: java.awt.Rectangle
• graphics package has a Rectangle class
  – to refer to it by its full name: graphics.Rectangle
• To use java.awt and graphics Rectangle packages in the same code, can use their full names (which includes their package name)
Packages

- In directory `c:jdk\lib\examples\graphics` have

```java
package graphics;

public class Rectangle {
    private double length=5.5;
    private double width=4.0;

    public double getArea()
    {
        return length*width;
    }
} // Rectangle.java
```

```java
package graphics;

public class Circle {
    private double radius=5;

    public double getArea()
    {
        return Math.PI * radius * radius;
    }
} // Circle.java
```
Packages

- In directory `c:jdk\lib\examples\test` have

```java
package test;
import graphics.*; // import graphics.Rectangle and graphics.Circle

public class TestGraphics
{
    public static void main (String[] args) {
        Rectangle r1 = new Rectangle();
        System.out.println("Rectangle area is " + r1.getArea());
        Circle c1 = new Circle();
        System.out.println("Circle area is " + c1.getArea());
    } // end of main ()
}
```
Packages

- Pathnames are usually relative and use the CLASSPATH environment variable.

**DOS**

- If: `CLASSPATH=c:\jdk\lib\examples`, and the classes in your graphics package are in `c:\jdk\lib\examples\graphics\`, and your test program is in package `test` in `c:\jdk\lib\examples\test\TestGraphics.java`.

From the DOS command line in `c:\jdk\lib\examples`, can type `javac test\TestGraphics.java` to compile and `java test.TestGraphics` to run.

Output:

- Rectangle area is 22.0
- Circle area is 78.53981633974483
Packages

Unix/Linux

- If: `CLASSPATH=/name/lib/examples`, and the classes in your graphics package are in `/name/lib/examples/graphics/`, and your test program is in package `test` in `/name/lib/examples/test/TestGraphics.java`

From the unix/linux command line in `/name/lib/examples`, you can type `javac test/TestGraphics.java` to compile and `java test.TestGraphics` to run

Output:

Rectangle area is 22.0
Circle area is 78.53981633974483
Inheritance

• OOP is one paradigm that facilitates managing the complexity of programs
• OOP applies principles of abstraction to simplify the tasks of writing, testing, maintaining and understanding complex programs
• OOP aims to increase code reuse
  – reuse classes developed for one application in other applications instead of writing new programs from scratch ("Why reinvent the wheel?")
• Inheritance is a major technique for realizing these objectives
Inheritance Overview

• Inheritance allows you to define a very general class then later define more specialized classes by adding new detail
  – the general class is called the base or parent class

• The specialized classes inherit all the properties of the general class
  – specialized classes are derived from the base class
  – they are called derived or child classes
Inheritance Overview

• After the general class is developed you only have to write the "difference" or "specialization" code for each derived class

• A class hierarchy: classes can be derived from derived classes (child classes can be parent classes)
  – any class higher in the hierarchy is an ancestor class
  – any class lower in the hierarchy is a descendant class
An Example of Inheritance: a Person Class

The base class:

• Constructors:
  – a default constructor
  – three others that initialize the firstName, lastName, and dateOfBirth attributes (instance variables)

• Accessor methods:
  – setFirstName to change the value of the firstName attribute
  – getFirstName to read the value of the firstName attribute
  – same for lastName
An Example of Inheritance: a Person Class

Accessor methods contd.:
- `setDateOfBirth` to change the value of the `dateOfBirth` attribute
- `getDateOfBirth` to read the value of the `dateOfBirth` attribute
- `writeOutput` to display the values of the `firstName`, `lastName` attributes

• One other class method:
  - `sameName` to compare the values of the `firstName` and `lastName` attributes for objects of the class

• Note: the methods are `public` and the attributes `private`
The base class can be used to implement specialized classes
  - For example: student, employee, faculty, and staff
Classes can be derived from the classes derived from the base class, etc., resulting in a *class hierarchy*
Example of Adding Constructor in a Derived Class: Student

```java
public class Student extends Person {
    private int studentNumber;
    public Student()
    {
        super();
        studentNumber = 0;
    }
    ...
}
```

- **Keyword** `extends` in first line
  - creates derived class from base class
  - this is inheritance

- Four new constructors (one on next slide)
  - default initializes attribute `studentNumber` to 0
- `super()` must be first action in a constructor definition
  - Included automatically by Java if it is not there
  - `super()` calls the parent default constructor
Example of Adding Constructor in a Derived Class: Student

- Passes parameter fName to constructor of parent class
- Uses second parameter to initialize instance variable that is not in parent class.

```java
public class Student extends Person {

  ...

  public Student(String fName, int newStudentNumber) {
    super(fName);
    studentNumber = newStudentNumber;
  }

  ...
```
More about Constructors in a Derived Class

- Constructors can call other constructors
- Use `super` to invoke a constructor that is defined in the parent class
  - as shown on the previous slide
- Use `this` to invoke a constructor that is defined within the derived class itself
  - shown on the next slide
Example of a constructor using this

**Student** class has a constructor with three parameters: **String** for the **firstName** and **lastName** attributes and **int** for the **studentNumber** attribute

```java
public Student(String fName, String lName,
               int newStudentNumber)
{
    super(fName, lName);
    studentNumber = newStudentNumber;
}
```

Another constructor within **Student** takes two **String** arguments and initializes the **studentNumber** attribute to a value of **0**:

- calls the constructor with three arguments, **fName**, **lName** (**String**) and **0** (**int**), within the same class

```java
public Student(String first, String last)
{
    this(first, last, 0);
}
```
Example of Adding an Attribute in a Derived Class: Student

A line from the Student class:

```java
private int studentNumber;
```

• Note that an attribute for the student number has been added
  - Student has this attribute in addition to firstName, lastName, and dateOfBirth, which are inherited from Person
Example of Overriding a Method in a Derived Class: Student

• Both parent and derived classes have a `writeOutput` method
• Both methods have the same parameters (none)
  – they have the same *signature*
• The method from the derived class *overrides* (replaces) the parent's
• It will not override the parent if the parameters are different (since they would have different signatures)
• This is *overriding*, not overloading

```java
public void writeOutput()
{
    System.out.println("Name: " + getFirstName() + " " + 
                       getLastName());
    System.out.println("Student Number : " + 
                       studentNumber);
}
```
Call to an Overridden Method

- Use `super` to call a method in the parent class that was overridden (redefined) in the derived class
- Example: `Student` redefined the method `writeOutput` of its parent class, `Person`
- Could use `super.writeOutput()` to invoke the overridden (parent) method

```java
public void writeOutput()
{
    super.writeOutput(); // prints first and last name
    System.out.println("Student Number : "+
        studentNumber);
}
```
<table>
<thead>
<tr>
<th>Overriding</th>
<th>Overloading</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Same method name</td>
<td>• Same method name</td>
</tr>
<tr>
<td>• Same signature</td>
<td>• Different signature</td>
</tr>
<tr>
<td>• One method in ancestor, one in descendant</td>
<td>• Both methods can be in same class</td>
</tr>
</tbody>
</table>
The final Modifier

• Specifies that a method definition cannot be overridden with a new definition in a derived class
• Example:

```java
public final void specialMethod()
{
    ...
}
```

• Used in specification of some methods in standard libraries
• Allows the compiler to generate more efficient code
• An entire class can be declared final, which means it cannot be used as a base class to derive another class
private & public
Instance Variables and Methods

- **private** instance variables from the parent class are not available by name in derived classes
  - "Information Hiding" says they should not be
  - use accessor methods to change them, e.g. can call parent’s `setFirstName` method for a Student object to change the `firstName` attribute

- **private** methods are **not** inherited!
  - use `public` to allow methods to be inherited
  - only helper methods should be declared `private`
What is the "Type" of a Derived class?

• Derived classes have more than one type
• They have the type of the derived class (the class they define)
• They also have the type of every ancestor class
  – all the way to the top of the class hierarchy
• All classes derive from the original, predefined Java class Object
• That is, Object is the original ancestor class for all other Java classes (including user-defined ones)
Assignment Compatibility

- **Can** assign an object of a derived class to a variable of any ancestor type

  ```java
  Person josephine;
  Employee boss = new Employee();
  josephine = boss;  // OK
  ```

- **Can not** assign an object of an ancestor class to a variable of a derived class type

  ```java
  Person josephine = new Person();
  Employee boss;
  boss = josephine;  // Not allowed
  ```

An employee is a person but a person is not necessarily an employee.
Character Graphics Example

- **Inherited**
- **Overrides**
- **Static**

**Figure**
- **Instance variables:**
  - offset
  - height
  - width
- **Methods:**
  - setOffset
  - getOffset
  - drawAt
  - drawHere

**Box**
- **Instance variables:**
  - offset
  - height
  - width
- **Methods:**
  - setOffset
  - getOffset
  - drawAt
  - drawHere
  - reset
  - drawHorizontalLine
  - drawSides
  - drawOneLineOfSides
  - spaces

**Triangle**
- **Instance variables:**
  - offset
  - base
- **Methods:**
  - setOffset
  - getOffset
  - drawAt
  - drawHere
  - reset
  - drawBase
  - drawTop
  - spaces
Java program execution order

• Programs normally execute in sequence

• Non-sequential execution occurs with:
  – selection (if/if-else/switch) and repetition (while/do-while/for)
    (depending on the test it may not go in sequence)
  – method calls, which jump to the location in memory that contains the method's instructions and returns to the calling program when the method is finished executing

• One job of the compiler is to try to figure out the memory addresses for these jumps

• The compiler cannot always know the address
  – sometimes it needs to be determined at run time
Static and Dynamic Binding

- **Binding**: determining the memory addresses for jumps (calls to class methods, etc.)
- **Static**: done at compile time
  - also called *offline*
- **Dynamic**: done at run time
- Compilation is done *offline*
  - it is a separate operation done before running a program
- Binding done at compile time is, therefore, static
- Binding done at run time is dynamic
  - also called *late binding*
Example of Dynamic Binding: General Description

- A derived class calls a method in its parent class which calls a method that is overridden (defined) in the derived class
  - the parent class is compiled separately; in some cases before the derived class is even written
  - the compiler cannot possibly know which address to use
  - therefore the address must be determined (bound) at run time
Dynamic Binding: Specific Example

**Parent class:** Figure
- Defines methods: drawAt and drawHere
- drawAt calls drawHere

**Derived class:** Box extends Figure
- Inherits drawAt
- redefines (overrides) drawHere
- Calls drawAt
  - uses the parent's drawAt method
  - which must call the derived class's, drawHere method
- Figure is compiled before Box is even written, so the address of drawHere (in the derived class Box) cannot be known then
  - it must be determined during run time, i.e. dynamically
Polymorphism revisited

- Using the process of dynamic binding to allow different objects to use different method actions for the same method name
- Method overloading is an example of polymorphism
- However, the term polymorphism is most often used in reference to dynamic binding
Summary

- A derived class inherits the instance variables & methods of the base class
- A derived class can create additional instance variables and methods
- The first thing a constructor in a derived class normally does is call a constructor in the base class
- If a derived class redefines a method defined in the base class, the version in the derived class overrides that in the base class
- Private instance variables and methods of a base class cannot be accessed directly in the derived class
- If A is a derived class of class B, then an instance of A (object) is both a member of classes A and B
  - the type of the object is both A and B
Read

• Sections 6.3 - 6.5