HST 952

Computing for Biomedical Scientists

Lecture 5

Outline

- Recursion and iteration
- Imperative and declarative programming revisited (the factorial problem)
- Encapsulation/information hiding and Abstract Data Types (ADTs)
- Classes, objects, and methods continued
- Arrays

Recursion and Iteration

- Some methods have a “fixed” number of steps (e.g. a method that uses multiplication to square a number n will use one multiplication no matter how large n is)
- Other methods have steps whose size depend on the value of their parameters
- Recursion strategy: do nearly all the work first; then there will only be a little left to do

  e.g.: to find 52!, have to multiply 53 numbers together, which requires 52 multiplications. First find 51! - multiply 52 numbers - and there is only one multiplication left to be done

Recursion

- Involves choosing a sub-problem that is of the same form as the main problem
- To avoid an infinite number of steps with a recursive strategy, must define a *base case*
- A base case is one for which there is no sub-problem, no further work needs to be done for a base case

  Note: may have more than one base case as with the Fibonacci numbers problem

  (in the factorial example, the base case is finding the factorial of 0 -- simply "return" the value 1)
- Basis for recursion is *mathematical induction*

Iteration

- Iteration strategy: By doing a little bit of work first, transform your problem into a smaller one with the same solution. Then solve this smaller problem

  e.g.: to find 52!, first multiply two of the 53 numbers to be multiplied and store their product. Now there are only 52 numbers to multiply (including the stored product) and 51 multiplications left to do. Multiply the stored product with another number from the remaining 52. Now there are only 51 numbers to multiply. Repeat this process of multiplying the product to a number until there is only one number remaining (the factorial) and there are zero multiplications left to do.

Factorial example revisited (iteration)

```
/* This class implements an imperative programming solution to the
   factorial problem using iteration
   factorial(n) = n * factorial(n - 1) and factorial(0) = 1 */

public class IterativeFactorial{
   public static void main(String[] args)
   {
         int factorial = 1;  // set initial value of factorial to factorial(0)
         int iterator = 1;   // set initial value of loop iterator to 1
         int n = 5;          // number whose factorial we are finding
```

Example continued on next slide

Factorial example revisited (iteration)

```
    while (iterator <= n) {
        /* set the new value of the variable factorial to its current
           value times the value of the iterator */
        factorial = factorial *  iterator++;
     } // end of while ()
    // print out the value of n factorial
    System.out.println("The factorial of " + n  + " is " + factorial);
  } // end of main
} // end of class IterativeFactorial
```

iterator incremented after its old value has been used

Factorial example revisited (iteration)

Trace of solution for the number 5
(factorial and iterator are initially 1):

factorial	iterator	
1	2	After first pass through while loop
2	3	Second time through while loop
6	4	
24	5	
120	6	Fifth time through while loop

```
while (iterator <= n)
{
   factorial = factorial *  iterator++;
}
```

Factorial example revisited (iteration)

```
public class IterativeFactorial2{
   public static void main(String[] args)
   {
        int n = 5;              // number whose factorial we are finding
        int factorial = n;  // set initial value of factorial to n
        int iterator = n;    // set initial value of loop iterator to n
        while (iterator > 1) {
        /* set the new value of factorial to its current value
            times the value of the iterator */
           factorial = factorial * --iterator;
        } // end of while ()
      System.out.println("The factorial of " + n + " is " + factorial);
   } // end of main
} // end of class IterativeFactorial2
```

iterator decremented before its old value has been used

Factorial example revisited (iteration)

Trace of solution for the number 5

(factorial and iterator are initially 5):

factorial	iterator
20	4
60	3
120	2
120	1

After first pass through while loop

Third time through while loop

Fourth time through while loop

Since we know that factorial(1) = 1, we could set the boolean expression in the while loop to (iterator >=2) and save two multiplications over the previous example

```
while (iterator > 1)
{
   factorial =  factorial *  --iterator;
}
```

Factorial example revisited (recursion)

```
/* This class implements an imperative programming solution to the
   factorial problem using recursion.
   factorial(n) = n * factorial(n - 1) and  factorial(0) = 1 */
public class RecursiveFactorial{
   private int factorial = 1;
   public int findFactorial(int number)
   { // this method returns the value of the factorial of number
     if (number == 0) {
       // the factorial of 0 is 1 (base case)
       return factorial;
     }
```

Example continued on next slide

Factorial example revisited (recursion)

```
    else {
        // the factorial of n is n * factorial(n - 1)
        factorial = number * findFactorial(number - 1);
         return(factorial);
    }
  } // end of  findFactorial method
 public static void main(String[] args) {
     int n = 5;      // number for which we are finding the factorial
    /* Since main is a static method, it cannot call findFactorial() directly
        Create a new RecursiveFactorial object in order to call findFactorial() */
     RecursiveFactorial fact = new RecursiveFactorial();
    System.out.println("The factorial of " + n + " is "  + fact.findFactorial(n));
   }
} // end of class RecursiveFactorial
```

Factorial example revisited

- Both examples shown above are imperative programming approaches to solving the factorial problem (using recursion and iteration)
- The declarative approach to the problem given in lecture 1 follows (uses recursion)

Factorial example revisited (recursion)

Declarative approach (definition using Scheme):

```
(define (factorial n)
    (if (= n 0) 1
    (* n (factorial (- n 1)))))

; applying factorial to a particular number:
 (factorial 5)
 ;Value:  120
```

Encapsulation/Information Hiding

One of the 3 cornerstones of object-oriented programming

- use classes and objects for programming
- objects include both data items and methods to act on the data

- protect data inside an object (do not allow direct access)
- use `private` modifier for instance variable declarations
- use `public` methods to access data

Formalized Abstraction: ADTs

ADT: Abstract data type

- A data type (in Java, a class) that is written using good information hiding/encapsulation techniques
- This Object-Oriented approach is used by several languages
- An ADT provides a public *user interface* so the user knows how to use the class
  - user interface has descriptions, parameters, and names of the class's methods
- Changes to a class implementation should not affect code that uses the class

Formalized Abstraction: ADTs

- Method definitions are usually public but specific implementation details are always hidden from the user
- The user cannot see or change the implementation
- The user only sees the interface, also called the *application programmer interface* (API)
- Prevents programming errors (e.g., user inadvertently changing the value of a public instance variable by using = instead of == for a comparison)
- Sharing API spares an end user from having to read through source code to understand how a class's methods work

Formalized Abstraction: ADTs

To create ADTs in Java that are used by others via an API:

- Use the `private` modifier when declaring instance variables
- Do *not* give the user the class definition (.java) file
- *Do* give the user the interface - a file with just the class and method descriptions and headings
  - the headings give the names and parameters of the methods
  - it tells the user how to use the class and its methods
  - it is all the user needs to know
  - the Java utility *javadoc* can be used to create the interface description file

Programming Tips for Writing Methods

- Use `public` and `private` modifiers judiciously
  - If a user will need the method, make it part of the interface by declaring it `public`
  - If the method is used only within the class definition (i.e., it is a *helper* method, then declare it `private`)
- Create a `main` method with diagnostic (test) code within a class's definition
  - run just the class to execute the diagnostic program (to make sure that it works the way it ought to)
  - when the class is used by another program the class's `main` is ignored

Testing a Method

- Test programs are sometimes called *driver* programs
- Keep it simple: test only one new method at a time
  - driver program should have only one untested method
- If method A uses method B, there are two approaches:
- *Bottom up*
  - test method B fully before testing A

Testing a Method

- *Top down*
  - test method A and use a *stub* for method B
  - A *stub* is a method that stands in for the final version and does little actual work. It usually does something as trivial as printing a message or returning a fixed value. The idea is to have it so simple you are nearly certain it will work.

Method Overloading

- The same method name has more than one definition *within the same class*
- Each definition must have a different signature
  - different argument types, a different number of arguments, or a different ordering of argument types
  - The return type is **not** part of the signature and **cannot** be used to distinguish between two methods with the same name and parameter types

Signature

- The combination of method name and number and types of arguments, in order
- `equals(Person)` has a different signature than `equals(String)`
  - same method name, *different argument types*
- *`myMethod(1)`* has a different signature than *`myMethod(1, 2)`*
  - same method name, *different number of arguments*

Signature

- *`myMethod(10, 1.2)`* has a different signature than *`myMethod(1.2, 10)`*
  - same method name and number of arguments, *but different order of argument types*:

    `(int, double) vs. (double, int)`

Overloading and Argument Type

- Accidentally using the wrong datatype as an argument can invoke a different method than desired
- For example, say we have defined a `Pet` class (there’s one defined in the Java text)
  - `set(int)` sets the pet's age in whole years
  - `set(double)` sets the pet's weight in pounds
  - `set(String, int, double)` sets the pet's name, age, and weight

Overloading and Argument Type

- You want to set the pet's weight to 6 pounds:
  - `set(6.0)` works as you want because the argument is type `double`
  - `set(6)` will set the *age* to 6, not the weight, since the argument is type `int`
- If Java does not find a signature match, it attempts some automatic type conversions, e.g. `int` to `double`
- An unwanted version of the method may execute

Overloading and Argument Type

Still using the Pet example of overloading:

- What you want: name "Scamp", weight 2, and age 3
- But you make two mistakes:
    1. you reverse the age and weight numbers, and
    2. you fail to make the weight a type double
- Remember: `set(String, int, double)` sets the pet's name, age, and weight
- `set("Scamp", 2, 3)` does not do what you want

Overloading and Argument Type

- `set(String, int, double)` sets the pet's name, age, and weight
- `set("Scamp", 2, 3)`
  - it sets the pet's age = 2 and the weight = 3.0
- Why?
  - `set` has no definition with the argument types `String`, `int`, `int`
  - However, it does have a definition with `String`, `int`, `double`,
    so it promotes the last number, 3, to 3.0 and executes the method with that signature

Constructors

- A *constructor* is a special method designed to initialize instance variables
- Automatically called when an object is created using `new`
- Has the same name as the class
- Often overloaded (more than one constructor for the same class definition)
  - different versions to initialize all, some, or none of the instance variables
  - each constructor has a different signature (a different number or sequence of argument types)

Defining Constructors

- Constructor headings do not include the word `void`
- In fact, constructor headings do not include a return type
- A constructor with no parameters is called a *default constructor*
- If no constructor is provided by the class creator, Java automatically creates a default constructor
  - If *any* constructor is provided, then *no* constructors are created automatically

Defining Constructors

Programming Tip

- Include a constructor that initializes *all* instance variables
- Include a constructor that has no parameters (that is, include your own *default constructor*)

Constructor Example

```
public class Person
{
   private String firstName;
   private String lastName;
   private GregorianCalendar dateOfBirth;
. . .
   public Person(String fName)
   {
      firstName = fName;
      lastName = null;
      dateOfBirth = null;
   }
```

Null is a special constant that be assigned to any variable of any class type. It is a place-holder for an object's address

Sample use:

```
Person person1 = new Person("Eric");
```

Constructor Example

```
public Person(String fName, String lName)
   {
      firstName = fName;
      lastName = lName;
      dateOfBirth = null;
   }
```

Sample use:

```
Person person2 = new Person("Eric","LeRouge");
```

```
public Person(String fName, String lName,
                 GregorianCalendar dob)
   {
      firstName = fName;
      lastName = lName;
      dateOfBirth = dob;
   }
```

Sample use:

```
Person person3 = new Person("Eric","LeRouge",
new GregorianCalendar(1965, 8, 21, 0, 0, 0));
```

Constructor Example

```
public Person()
    {
        firstName = null;
        lastName = null;
        dateOfBirth = null;
    }
```

Sample use:

```
Person person4 = new Person();
```

This is the *default constructor*

Using Constructors

- The keyword `new` must precede a call to a constructor
- If you want to change values of instance variables after you have created an object using a constructor, you must use other methods (e.g. defined set methods) for the object
  - you cannot call a constructor for an object after it is created
  - `set` methods should be provided for the purpose of changing values of instance variables

Arrays

- An array: a single name for a collection of data values, all of the same data type

  (it is a collection of variables that have the same type)
- Arrays are a carryover from earlier programming languages (e.g. C, C++)
- Array: more than a primitive type, less than an object
  - they work like objects when used as method arguments and return types (i.e., arrays are reference types)
  - they do not have or use inheritance
  - they are sort of like a Java class that is not fully implemented
- Arrays are a natural fit for loops, especially `for` loops

Creating Arrays

- General syntax for declaring an array:

  ```
  Base_Type[] Array_Name = new
  Base_Type[Length];
  ```

- Examples:

  80-element array with base type `char`:

  ```
  char[] symbol = new char[80];
  ```

  100-element array of `doubles`:

  ```
  double[] realNums = new double[100];
  ```

  90-element array of type `Person`:

  ```
  Person[] people = new Person[90];
  ```

Three Ways to Use [ ] (Brackets) with an Array Name

1. To create a type name, e.g. `int[] pressure;` creates a name `pressure` with the type "`int` array"
   - note that the types `int` and `int` array are different
   - `int` array is the type of the name `pressure`
   - the type of the data that can be stored in `pressure` is `int`

2. To create a new array, e.g. `pressure = new int[100];`

3. To name a specific element in the array
   - also called *an indexed variable*, e.g.

```
pressure[3] = 55;
System.out.println("You entered" +
pressure[3]);
```

Some Array Terminology

```
temperature[n + 2]
```

Array name

```
temperature[n + 2]
```

Index - also called a *subscript*
- must be an `int`,
- or an expression that evaluates to an `int`

```
temperature[n + 2]
```

Indexed variable - also called an *element* or *subscripted variable*

Value of the indexed variable
- also called an element of the array

```
temperature[n + 2] = 32;
```

Note that "element" may refer to either a single indexed variable in the array or the *value* of a single indexed variable.

Array Length

- The length of an array is specified by the number in brackets when it is created with `new`
  - it determines the amount of memory allocated for the array elements (values)
  - it determines the *maximum* number of elements the array can hold
    - storage is allocated whether or not the elements are assigned values

Array Length

- The array length can be read with the instance variable `length`, e.g. the following code displays the number 20 (the *size*, or *length* of the `Person` array, `morePeople`):

```
Person[] morePeople = new Person[20];
System.out.println(morePeople.length);
```

- The length attribute is established in the declaration and cannot be changed unless the array is redeclared

Initializing an Array's Values in Its Declaration

- Array elements can be initialized in the declaration statement by putting a comma-separated list in braces
- Uninitialized elements will be assigned some default value, e.g. 0 for `int` arrays
- The length of an array is automatically determined when the values are explicitly initialized in the declaration
- For example:

```
double[] realNums = {5.1, 3.02, 9.65};
System.out.println(realNums.length);
```

- displays `3`, the length of the array `realNums`

Subscript Range

- Array subscripts use zero-numbering
  - the first element has subscript 0
  - the second element has subscript 1
  - etc. - the nth element has subscript n-1
  - the last element has subscript `length-1`
- For example:

```
int[] scores = {97, 86, 92, 71};
```

Subscript:	0	1	2	3
Value:	97	86	92	71

Subscript out of Range Error

- Using a subscript larger than `length-1` causes a *run time* (not a compiler) error
  - an `ArrayOutOfBoundsException` is thrown
    - you do not need to catch this exception
    - you need to fix the problem and recompile your code
- Some programming languages, e.g. C and C++, do not even cause a run time error
  - one of the most dangerous characteristics of these languages is that they allow out of bounds array indexes

Initializing Array Elements in a Loop

- Array processing is easily done in a loop
- A `for` loop is commonly used to initialize array elements
- Example:

```
int i;//loop counter/array index
int[] a = new int[10];
for(i = 0; i < a.length; i++)
    a[i] = 0;
```

  - note that the loop counter/array index goes from `0` to `length - 1`
  - it counts through `length = 10` iterations/elements using the zero-numbering of the array index

Arrays, Classes, and Methods

An array of a class can be declared and the class's methods applied to the elements of the array.

create an array of `Person`s

each array element is a `Person` instance variable

use the `readInfo` method of `Person`

use the `writeInfo` method of `Person`

```
public void printFirstNames()
 {
    int numberOfPeople = 50;
    Person[] people = new Person[numberOfPeople];
    for (int i = 0; i < numberOfPeople; i++)
    {
       people[i] = new Person();
       //readInfo allows us to set the attributes of a person
       people[i].readInfo();
       //writeInfo allows us to print the attributes of a person
       System.out.println(people[i].writeInfo());
    }
 }
```

Arrays and Array Elements as Method Arguments

Arrays and array elements can be used with classes and methods just like other objects

- both an indexed element and an array name can be an argument in a method
- methods can return an array value or an array name

Indexed Variables as Method Arguments

```
public static void main(String arg[])
{
   String scoreString =
            JOptionPane.showInputDialog(ÒEnter your grade\nÓ);
   int firstScore = Integer.parseInt(scoreString);
   int[ ] nextScore = new int[3];
   int i;
   double possibleAverage;
   for (i = 0; i < nextScore.length; i++)
       nextScore[i] = 80 + 10*i;
   for (i = 0; i < nextScore.length; i++)
   {
       possibleAverage = average(firstScore, nextScore[i]);
       System.out.println("If your score on exam 2 is "
                          + nextScore[i]);
       System.out.println("your average will be "
                          + possibleAverage);
   }
}
public static double average(int n1, int n2)
{
  return (n1 + n2)/2.0;
}
```

`nextScore` is an array of `int`s

an element of `nextScore` is used as an argument of method `average`

`average` method definition

Modification of ArgumentDemo program in text

When Can a Method Change an Indexed Variable Argument?

Remember:

- primitive types are call-by-value
  - only a copy of the value is passed as an argument in a method call
  - so the method *cannot* change the value of the indexed variable
- class types are reference types; they pass the address of the object when they are an argument in a method call
  - the corresponding argument in the method definition becomes another means of accessing the object's contents (another "key" for the same mailbox)
  - this means the method has access to the object's contents
  - so the method *can* change the values associated with the indexed variable if it is a class (and not a primitive) type

Array Names as Method Arguments

When using an entire array as an argument to a method:

- use just the array name and no brackets (this passes the memory address of the array's first element)
- as described in the previous slide, the method has access to the original array contents and can change the value of its elements
- the length of the array passed can be different for each call
  - when you define the function you do not know the length of the array that will be passed
  - so use the `length` attribute inside the method to avoid `ArrayIndexOutOfBoundsExceptions`

Example: An Array as an Argument in a Method Call

```
public static void
   showArray(char[] a)
{
   int i;
   for(i = 0; i < a.length; i++)
         System.out.println(a[i]);
}
```

the method's argument is the name of an array of characters

uses the `length` attribute to control the loop
allows different size arrays and avoids index-out-of-bounds exceptions

Arguments for the Method main

- The heading for the `main` method shows a parameter that is an array of `Strings`:

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

- When you run a program from the command line, all words after the class name will be passed to the main method in the args array:

```
java TestProgram Josephine Student
```

Arguments for the Method main

- The following `main` method in the class `TestProgram` will print out the first two arguments it receives:

```
public static void main(String[] args)
{
   System.out.println("Hello " + args[0] + " " + args[1]);
}
```

- In this example, the output from the command line above will be:

```
Hello Josephine Student
```

Using = with Array Names: Remember They Are Reference Types

```
int[] a = new int[3];
int[] b = new int[3];
for(int i=0; i < a.length; i++)
   a[i] = i;
b = a;
System.out.println(a[2] + " " + b[2]);
a[2] = 10;
System.out.println(a[2] + " " + b[2]);
```

This does not create a copy of array `a`; it makes `b` another way of accessing the values associated with array `a`.

The output for this code will be:

```
2 2
10 10
```

If we change a stored value using `a`, we retrieve the changed value when we use `b` for access

Using == with Array Names: Remember They Are Reference Types

```
int i;
int[] a = new int[3];
int[] b = new int[3];
for(i=0; i < a.length; i++)
   a[i] = 0;
for(i=0; i < b.length; i++)
   b[i] = 0;
if(b == a)
   System.out.println("a equals b");
else
 System.out.println("a does not equal b");
```

`a` and `b` are both 3-element arrays of `int`s

all elements of `a` and `b` are assigned the value `0`

tests if the *addresses* of `a` and `b` are equal, not if the array values are equal

The output for this code will be "`a does not equal b`" because the *addresses* referenced by the arrays are not the same.

Testing Two Arrays for Equality

- To test two arrays for equality you need to define an `equals` method that returns true if and only if the arrays have the same length and all corresponding values are equal
- This code shows an example of such an `equals` method.

```
public static boolean equals(int[] a, int[] b)
 {
    boolean match;
    if (a.length != b.length)
        match = false;
    else
    {
        match = true; //tentatively
        int i = 0;
        while (match && (i < a.length))
        {
            if (a[i] != b[i])
                match = false;
            i++;
        }
    }
    return match;
 }
```

Methods that Return an Array

- Yet another example of passing a reference
- Actually, it is the address of the array that is returned
- The local array name within **main** provides another way of accessing the contents of the original array

```
public class returnArrayDemo
 {
    public static void main(String arg[])
    {
       char[] c;
       c = vowels();
       for(int i = 0; i < c.length; i++)
         System.out.println(c[i]);
    }
    public static char[] vowels()
    {
       char[] newArray = new char[5];
       newArray[0] = 'a';
       newArray[1] = 'e';
       newArray[2] = 'i';
       newArray[3] = 'o';
       newArray[4] = 'u';
       return newArray;
    }
 }
```

`c`, `newArray`, and the return type of `vowels` are all the same type: `char array`

Read

- Chapter 5 -- sections 5.1 - 5.7
- Chapter 6 -- sections 6.1 - 6.2