STUDENT OUTLINE

Lesson 7 – More About Methods

INTRODUCTION: Programs of any significant size are broken down into logical pieces called methods. It was recognized long ago that programming is best done in small sections that are connected in very specific and formal ways. Java provides the construct of a function, allowing the programmer to develop new functions not provided in the original Java libraries. Breaking down a program into blocks or sections leads to another programming issue regarding identifier scope. Also, functions need to communicate with other parts of a program that requires the mechanics of parameter lists and a return value.

The key topics for this lesson are:

A. Writing Methods in Java
B. Value Parameters and Returning Values
C. The Signature of a Method
D. Lifetime, Initialization, and Scope of Variables

VOCABULARY:

METHOD DECLARATION
METHOD DEFINITION
PARAMETERS
ACTUAL PARAMETERS
VALUE PARAMETERS
FORMAL PARAMETERS
SIGNATURE
SCOPE
GLOBAL SCOPE
LOCAL SCOPE
BLOCK
FUNCTION
STATIC VARIABLE

DISCUSSION:

A. Writing Methods in Java

1. A method is like a box that takes data in, solves a problem, and usually returns a value. The standard math methods follow this pattern:

```
Math.sqrt (2)  -->  1.414
Math.sin (30)  -->  -0.988 (Note: computation is in radians!)
```

2. There are times when the built-in methods of Java will not get the job done. We will often need to write customized methods that solve a problem using the basic tools of a programming language.

3. For example, suppose we need a program that converts gallons into liters. We could solve the problem within the *main* method, as shown in Program 7-1.

Program 7-1

```
import chn.util.*;

class GallonsToLiters
{
  public static void main (String[] args)
  {
    ConsoleIO console = new ConsoleIO();

    System.out.println("Enter an amount of gallons --> ");
    double gallons = console.readDouble();
    double liters = gallons * 3.785;
    System.out.println("Amount in liters = " + liters);
  }
}
```

This works fine, but the mathematics of the conversion is buried inside the `main` method. The conversion tool is not available for general use. We are not following the software engineering principle of writing code that can be recycled in other programs.

4. Here is the same routine coded as a reusable method, which would allow for conversion of gallons to units other than liters:

Program7-2

```
import chn.util.*;

class FluidConverter
{
  public double toLiters(double amount)
  {
    return amount * 3.785;
  }
}
public class TestConverter
{
  public static void main(String[] args)
  {
    ConsoleIO console = new ConsoleIO();
    FluidConverter convert = new FluidConverter();

    System.out.print("Enter an amount of gallons --> ");
    double gallons = console.readDouble();
    System.out.println("Amount in liters = " +
      convert.toLiters(gallons));
   }
}
```

Sample run output:

```
Enter an amount of gallons --> 10
Amount in liters = 37.85
```

5. Here is the sequence of events in Program 7-2.
   a. Execution begins in the method named `main` with the user prompt and the input of an amount of gallons.
   b. The `toLiters` method of the `convert` object is called and the number of gallons is passed as an argument to the `toLiters` method.
   c. Program execution moves to `toLiters`, which does the computation and returns the answer to the calling statement.
   d. The answer is displayed.

6. The general syntax of a method declaration is

```
modifiers return_type method_name ( parameters )
{
    method_body
}
```

   Example (from program 7-2):

```
public double toLiters(double gallons)
```

   a. The ***modifiers*** refers to a sequence of terms designating different kinds of methods. These will be discussed gradually in later lessons.
   b. The ***return_type*** refers to the type of data a method returns. The data type can be one of the predefined types (integer, double, char) or a user-defined type.
   c. *method_name* is the name of the method. It must be a valid identifier. In Program 7-2, the names of the methods are `main` and `toLiters`.
   d. The *parameters* list will allow us to send values to a method. The parameter list consists of one or more type-identifier pairs (example: **double** amount). The parameters in the method instantiation are called the *formal parameters*.
   e. The *method_body* contains statements to accomplish the work of the method. In the `toLiters` method there is one line in the body.

7. The last line of the main method contains the following reference to a method: `convert.toLiters(gallons)`. This causes the value represented by the variable `gallons` to be sent (passed) to `toLiters`, where a computation is done, and a value is returned. In the `toLiters` method the value that is passed is referred to as `amount`. In the next section we will talk in detail about passing values in this manner.

8. This idea of a method taking in a value, using the value in a computation, and returning the result of a computation is similar to the mathematical idea of a *function*, which is why a method that returns a value is often referred to as a function.

B. Value Parameters and Returning Values

1. The word *parameter* is used to describe variables that pass information within a program. The simple process in Program 7-2 of passing a number (of gallons) to a method that will compute something (number of liters) is representative of a common occurrence in Java programming – passing values with parameters. Sometimes the word "argument" is used in place of parameter.

2. We mentioned above that the parameter `gallons` is a formal parameter because it is named in the instantiation of the method. Another kind of parameter is a *value parameter*, which is used in a computation. In the computation `amount * 3.785`, `amount` is a value parameter.

3. A value parameter has the following characteristics:

   a. This value parameter is a local variable. This means that it is valid only inside the block (method) in which it is declared.
   b. It receives a copy of the argument that was passed to the method. The value of 10 stored in `gallons` (inside `main`) is passed to the parameter `amount` (inside `toLiters`).
   c. The value parameter is a variable that can be modified within the method.

4. In order for a method to return a value, there must be a **`return`** statement somewhere in the body of the method.

5. If a method returns no value the term **`void`** should be used. For example:

```
public void printHello( )
{
   System.out.println("Hello world");
}
```

6. A function (method) can have multiple parameters in its parameter list. For example:

```
public double doMath(int a, double x)
{
   ... code ...
   return doubleVal;
}
```

   When this method is called, the arguments fed to the `doMath` method must be of an appropriate type. The first argument must be an integer. The second argument can be an integer because it will be promoted to a double.

```
double dbl = doMath(2, 3.5);      //  this is okay
double dbl = doMath(2, 3);        //  this is okay
double dbl = doMath(1.05, 6.37);  //  this will not compile
```

7. Value parameters are often described as one-way parameters. The information flows into a function but no information is passed back through the value parameters. A single value can be passed back using the return statement, but the formal parameters in the function remain unchanged.

8. The formal parameters used to supply values for the value parameters can be either literal values `(2, 3.5)` or variables `(a, x)`.

```
double dbl = doMath(a, x);    // example using variables
```

C. The Signature of a Method

1. In order to call a method legally, you need to know its name, you need to know how many formal parameters it has, and you need to know the type of each parameter. This information is called the method's *signature*. The signature of the method `doMath` can be expressed as as: `doMath(int, double)`. Note that the signature does not include the names of the parameters; in fact, if you just want to use the method, you don't even need to know what the formal parameter names are, so the names are not part of the signature.

2. Java allows two different methods in the same class to have the same name, provided that their signatures are different. We say that the name of the method is *overloaded* because it has several different meanings. The computer doesn't get the methods mixed up. It can tell which one you want to call by the number and types of the actual parameters that you provide in the subroutine call statement. You have already seen overloading used in the `System.out` class. This class includes many different methods named `println`, for example. These methods all have different signatures, such as:

```
println(int)     println(double)     println(String)
println(char)    println(boolean)    println()
```

3. The signature does not include the method's return type. It is illegal to have two methods in the same class that have the same signature but that have different return types. For example, it would be a syntax error for a class to contain two methods defined as:

```
double dbl = doMath(int, double);
int dbl = doMath(int, double);
```

D. Lifetime, Initialization, and Scope of Variables

1. Three categories of Java variables have been explained thus far in this curriculum guide.
   - Instance variables
   - Local variables
   - Parameter variables

2. The lifetime of a variable defines the portion of run time during which the variable exists.

   a. When an object is constructed, all its instance variables are created. As long as any part of the program can access the object, it stays alive.
   b. A local variable is created when the program enters the statement that defines it. It stays alive until the block that encloses the variable definition is exited.
   c. When a method is called, its parameter variables are created. They stay alive until the method returns to the caller.

3. The initial state of a variable is also determined by its type.

   a. Instance variables (associated with a particular object) and static variables (associated with a particular class) are automatically initialized with a default value (0 for numbers, **`false`** for **`boolean`**, **`null`** for objects) unless you specify another parameter.
   b. Parameter variables are initialized with copies of the formal parameters.
   c. Local variables are not initialized by default. An initial value must be supplied. The compiler will generate an error if an attempt is made to use a local variable that has never been initialized.

4. Scope refers to the area of a program in which an identifier is valid and has meaning.

   a. Instance variables of a class are usually declared **`private`**, and have class scope. Class scope begins at the opening left brace, `{`, of the class definition and terminates at the closing brace, `}`, of the class definition. Class scope enables methods of a class to directly access all instance variables defined in the class.
   b. The scope of a local variable extends from the point of its definition to the end of the enclosing block
   c. The scope of a parameter variable is the entire body of its method.

5. An example of the scope of a variable is given in Program 7-3. The class `ScopeTest` is created with four methods:

   - `printLocalTest`
   - `printInstanceTest`
   - `printParamTest`

- `main`

6. The subclass `st` is created as "a kind of" `ScopeTest`, so it contains the same methods. Each of these methods contains a variable named `test.`

7. The statement `st.printLocalTest()` calls the method `printLocalTest` , and in a similar way each method is called.

8. The results show the following about the scope of the variable `test`:

   a. Within the scope of `main`, the value of test is 10, the value assigned within the `main` method.
   b. Within the scope of `printLocalTest`, the value of test is 20, the value assigned within the `printLocalTest` method
   c. Within the scope of `printInstanceTest`, the value of test is 30, the private value assigned within `ScopeTest`, because there is no value given to `test` within the `printInstanceTest` method
   d. Within the scope of `printParamTest`, the value of test is 40, the value sent to the `printParamTest` method

Program 7-3

```
public class ScopeTest
{
  private int test = 30;

  public void printLocalTest()
  {
    int test = 20;
    System.out.println("printLocalTest: test = " + test);
  }

  public void printInstanceTest()
  {
    System.out.println("printInstanceTest: test = " + test);
  }

  public void printParamTest(int test)
  {
    System.out.println("printParamTest: test = " + test);
  }

  public static void main (String[ ] args)
  {
    int test = 10;

    ScopeTest st = new ScopeTest();
    System.out.println("main: test = " + test);

    st.printLocalTest();
    st.printInstanceTest();
    st.printParamTest(40);
  }
}
```

Run output:

```
main: test = 10
printLocalTest: test = 20
printInstanceTest: test = 30
printParamTest: test = 40
```

SUMMARY/ REVIEW: Your programs will grow in size and complexity. Initially you will not use all the tools presented in this lesson and other lessons regarding methods. However, you need to see and understand all the method-writing tools in Java since eventually you will need them in your own work and to help you read another programmer's code.

ASSIGNMENT:
Lab Exercise, L.A.7.1, *Fun*
Lab Exercise, L.A.7.2, *Polygon*
Lab Exercise, L.A.7.3, *RectangleMethods*

LAB EXERCISE

Fun

Assignment:

1. The two temperature scales used in the United States are Celsius and Fahrenheit. The mathematical relationship between the two scales is:

$$C = \frac{5}{9}(F - 32)$$

   Write two methods, `fToC` and `cToF`, which convert temperatures from one scale to another. For example, a call of `fToC` will return the equivalent Celsius temperature for a given Fahrenheit temperature.

```
Celsius = fToC(100);     // Celsius now stores 37.8
```

2. Write a method which takes in the radius of a sphere and returns its volume. The formula is:

$$V = \frac{4}{3} p\, r^3$$

   You are encouraged to use a constant for the value of p .

3. Write a method which returns the hypotenuse of a right triangle given the input of the two smaller sides. Use the Pythagorean theorem:

$$a^2 + b^2 = c^2$$

Instructions:

1. Format all floating point values to two decimal places (0.01).

2. You can use the following format of an output statement to test your program:

```
System.out.println("212 F --> " + Format.left(fToC(212), 10, 2));
```

   the `main` method could be written using only 11 methods calls.

3. Use the following values to test your methods:

   Fahrenheit to Celsius: 212ºF, 98.6ºF, 10ºF
   Celsius to Fahrenheit: -15ºC, 0ºC, 70ºC
   Volume of a sphere, radius of: 1.0, 2.25, 7.50
   Hypotenuse calculations: sides of 3.0 and 4.0, sides of 6.75 and 12.31

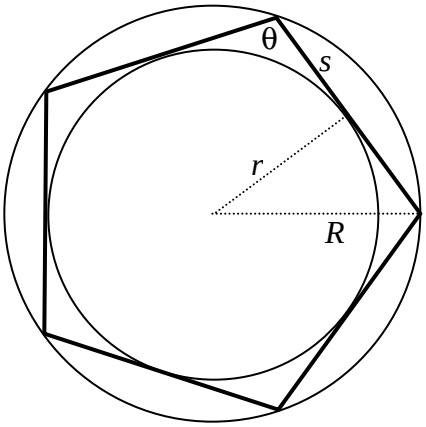
LAB EXERCISE

Polygon

Background:

Polygons are two-dimensional shapes formed by line segments. The segments are edges that meet in pairs at corners called *vertices*. A polygon is *regular* if all its sides are equal and all its angles are equal.

For an *n*-sided regular polygon of side *s*, the angle at any vertex is *q*, and the radii of the inscribed and circumscribed circles are *r* and *R* respectively. A 5-sided regular polygon (pentagon) would be represented as follows:



Assignment:

1. Create a `RegularPolygon` class to model any regular polygon. Use the following declarations as a starting point for your lab work.

```
class RegularPolygon
{

   private int myNumSides;        // # of sides
   private double mySideLength;   // length of side
   private double myR;            // radius of circumscribed circle
   private double myr;            // radius of inscribed circle

   // constructors
   public RegularPolygon()
   {
   }

   public RegularPolygon(int numSides, double sideLength)
   {
   }
```

```
// private methods
private void calcr()
{
}
```

```
    private void calcR()
    {
    }

    // public methods
    public double vertexAngle()
    {
    }

    public double Perimeter()
    {
    }

    public double Area()
    {
    }

    public double getNumside()
    {
    }

    public double getSideLength()
    {
    }

    public double getR()
    {
    }

    public double getr()
    {
    }
}
```

2. Write two constructor methods. The default constructor creates a 3-sided polygon (triangle). The other constructor takes an integer value representing the number of sides and a double value representing the length of side, and constructs the corresponding regular polygon.

3. Write a method that calculates the vertex angle, q. This angle can be determined as follows:

$$q = \left(\frac{n-2}{n} \right) \times 180^\circ$$

where n represents the number of sides.

4. Write a method that calculates the perimeter. This value is determined simply as the number of sides, n, times the length of a side, s:

$$perimeter = ns$$

5. Write a method that calculates the radius of the inscribed circle, *r*. The inscribed circle is the circle that can be drawn inside of the regular polygon such that it is tangent to every side of the polygon, for example, the smaller circle in the diagram above. It can be calculated as:

$$r = \frac{1}{2}\, s\, \cot\left(\frac{p}{n}\right)$$

where *n* represents the number of sides, *s* represents the length of a side, and `cot()` is the trigonometric function, cotangent. We use the value p instead of 180 in the formula because the Java math functions assume that angles are given in radians instead of degrees. Note: the built-in Java method `math.PI` produces the value of p. An alternative is to replace p with 180 in all the formulas here and use following method from the Math Class to convert from degrees to radians:

```
Math.toRadians(double angdeg)
```

The cotangent function is not part of the Java Math library, however, the cotangent of an angle can be calculated as the reciprocal of the tangent as follows:

$$\cot(q) = \frac{1}{\tan(q)}$$

6. Write a method that calculates the radius of the circumscribed circle, *R*. The circumscribed circle is the circle that intersects each vertex of the polygon, for example the larger circle in the diagram above. *R* can be calculated as:

$$R = \frac{1}{2} s\, \csc\left(\frac{p}{n}\right)$$

where *n* represents the number of sides, *s* represents the length of a side and `csc()` is the trigonometric function, cosecant. The cosecant function is not part of the Java Math Class, however, the cosecant of an angle can be calculated as the reciprocal of the sine as follows:

$$\csc(q) = \frac{1}{\sin(q)}$$

7. Write a method that calculates the area of the regular polygon . It can be calculated as:

$$area = \frac{1}{2} n R^2 \sin\left(\frac{2p}{n}\right)$$

where *n* represents the number of sides and *R* represents the radius of the circumscribed circle.

8. All trigonometric function in the Java Math Class take radians as the parameter. To convert an angle measured in degrees to the equivalent angle measured in radians use the following method from the Math Class:

```
public static double toRadians(double angdeg)
```

9. Write a testing class with a `main()` that constructs a `RegularPolygon` and calls each method. A sample run of the program for a polygon with 4 sides of length 10 would give:

```
number of sides = 4
length of side = 10.00
radius of circumscribed circle = 7.07
radius of inscribed circle = 5.00
vertex angle = 90.0
perimeter = 40.00
area = 100.00
```

Instructions:

1. Format all floating point values to two decimal places (0.01).

2. Test the default constructor by constructing a regular polygon with no parameters as follows:

```
RegularPolygon poly = new RegularPolygon();
```

3. Use the following values to test your functions:

   Square: number of sides = 4, length of side = 10
   Octagon: number of sides = 8, length of side = 5.75
   Enneadecagon: number of sides = 19, length of side = 2
   Enneacontakaihenagon: number of sides = 91, length of side = 0.5

   Answers:

	n	s	? (degrees)	r	R	Perimeter	Area
Square	4	10	90.00	5.00	7.07	40.00	100.00
Octagon	8	5.75	135.00	6.94	7.51	46.00	159.64
Enneadecagon	19	2	161.05	5.99	6.08	38.00	113.86
Enneacontakaihenagon	91	0.5	176.04	7.24	7.24	45.50	164.68

LAB EXERCISE

RectangleMethods

Background:

1. In this lab exercise, you will continue your work on the `Rectangle` class created in *L.A.6.2 - Rectangle* by adding additional attributes and behaviors. These enhancements will include the ability to construct a `Rectangle` from and existing one, get and set the x coordinate, y coordinate, width and height parameter, set the orientation in which the rectangle will be drawn, and display textual information on the drawing surface.

2. The specifications of a class that models a rectangular shape would be:

   Variables

```
private double myX;      // the x coordinate of the rectangle
private double myY;      // the y coordinate of the rectangle
private double myWidth;  // the width of the rectangle
private double myHeight; // the height of the rectangle

// saves the direction the pen is pointing
// (0 = horizontal, pointing right)
private double myDirection;

// Creates a 500 x 500 SketchPad with a DrawingTool, pen, that is used
// to display Rectangle objects. The Drawingtool is declared static
// so that multiple Rectangle objects can be drawn on the Sketchpad
// at the same time.
private static DrawingTool pen =
   new DrawingTool(new SketchPad(500, 500));
```

   Constructors

```
// Creates a default instance of a Rectangle object with all dimensions
// set to zero.
Rectangle()

// Creates a new instance of a Rectangle object with the left and right
// edges of the rectangle at x and x + width. The top and bottom edges
// are at y and y + height.
Rectangle(double x, double y, double width, double height)

// Creates a new instance of a Rectangle object that is a copy of an
// existing Rectangle object.
Rectangle(Rectangle rect)
```

   Methods

```
// Sets the x coordinate of this Rectangle
public void setXPos(double x)
```

```
// Sets the y coordinate of this Rectangle
public void setYPos(double Y)
```

```
// Sets the width this Rectangle
public void setWidth(double width)

// Sets the height of this Rectangle
public void setHeight(double height)

// Sets the direction the DrawingTool is pointing
// 0 = horizontal to the right
public void setDirection(double direction)


// Returns the x coordinate of this Rectangle
public double getXPos()

// Returns the y coordinate of this Rectangle
public double getYPos()

// Returns the width of this Rectangle
public double getWidth()

// Returns the height of this Rectangle
public double getHeight()

// Returns the direction the DrawingTool is pointing
// 0 = horizontal to the right
public double getDirection()

// calculates and returns the perimeter of the rectangle
public double getPerimeter()

// Calculates and returns the area of the rectangle.
public double getArea()


// Draws String str at the specified x and y coordinates
public void drawString(String str, double x, double y)

// Draws a new instance of a Rectangle object with the left and right
// edges of the rectangle at x and x + width. The top and bottom edges
// are at y and y + height.
public void draw()
```

Assignment:

1. Implement a `Rectangle` class with the following properties.

    a. A default `Rectangle` object is specified in the constructor with the `x`, `y`, `width` and `height` set to 0.

    b. A `Rectangle` object is specified in the constructor with the left and right edges of the rectangle at `x` and `x + width`. The top and bottom edges are at `y` and `y + height`.

c. A `Rectangle` object is specified that is a copy of an existing `Rectangle`.

d. Methods `getXPos`, `getYPos`, `getWidth`, and `getHeight`, return the `x`, `y`, `height` and `width` of the `Rectangle` respectively.

e. A method, `getDirection`, returns the current orientation of the `DrawingTool`.

f. Methods `setXPos`, `setYPos`, `setWidth`, and `setHeight`, sets the `x`, `y`, `height` and `width` of the `Rectangle` respectively to the value of each methods **double** parameter.

g. A method `setDirection`, sets the current orientation of the `DrawingTool`.

h. A method `getPerimeter` calculates and returns the perimeter of the `Rectangle`.

i. A method `getArea` calculates and returns the area of the `Rectangle`.

j. A method `draw` displays a new instance of a `Rectangle` object.

k. A method `drawString` displays String at the specified x and y coordinates of the drawing area.

2. The methods `draw`, `drawString`, and `setDirection` make use of existing `DrawingTool` methods. Refer to handout, *H.A.1.1 – DrawingTool*, for details on the `DrawingTool` methods.

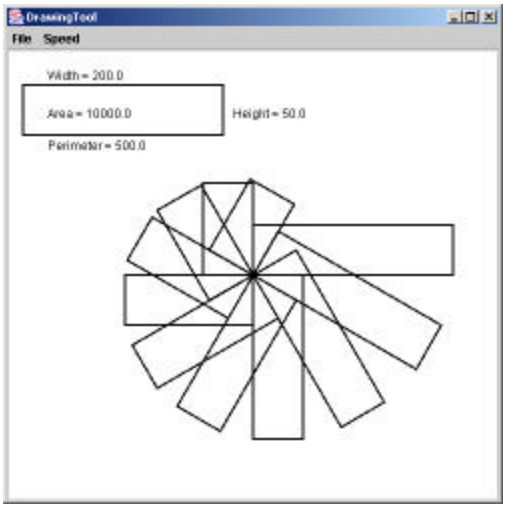
3. Write a testing class with a `main` method that constructs a `Rectangle`, `rectA`, and calls `setDirection`, `setWidth`, and `draw` for each `Rectangle` created. It is recommended that the changes in orientation and width of each successive rectangle in the spiral be calculated using the `getDirection` and `getWidth` methods. For example, if the increment for each turn is given by `turnInc` and the decrease in size of rectangle is given by `widthDec`, then successive calls to the following:

```
rectA.setDirection(rectA.getDirection() - turnInc);
rectA.setWidth(rectA.getWidth() - widthDec);
rectA.draw();
```

would draw each "spoke" of the rectangular spiral:

Construct another `Rectangle`, `rectB` that is a copy of the original `rectA`. Draw the rectangle in the upper left corner of drawing area. Label the rectangle with its width, height, perimeter and area.

The resulting image would be similar to the one shown below:



4. Turn in the source code with the run output attached. It is recommended that the `Rectangle` class and the testing class be combined in one source file (*RectangleMethods.java*).