

## An Introduction to Objects and Classes

### I. Using and Constructing Objects & Terminology

Recall: Data Types - **int**, **char**, **double**, **float** ex.) **int** number;

- 1) Data Type - a collection of values together with a set of basic operations defined on these values.
- 2) Encapsulation - the separation of a concepts logical properties from its actual implementation (coding) i.e. hiding the data
- 3) Object - a "special" variable created from a *class* not from a built-in data type
- 4) Class - a programmer-defined data type out of which objects associated with methods and/or member (field/instance) variables can be created
  - a) member variables - the concrete data variables, usually defined to be **private**

Note: **private** means that the data can not be manipulated from outside the file except by the member functions as described below.

b) member methods - methods associated with a class that can only be accessed (called) by the objects also in that class i.e. the operations allowed on the data in the class usually defined to be **public**

1) three basic categories of operations:

- (a) constructors - creates a new instance of the data type
- (b) mutator - makes a new value from a previous value or values
- (c) accessor - observes a value without changing it

Note: (b) and (c) are discussed in Chapter 7

Note: **public** means that the user has access to these functions and can use them in the file that the user is using called the client file.

*Dot Operator* syntax- the period (dot) used in the call to a member method that connects the object to the member method. It starts with the class object (variable), then a dot, then the member method name, then the argument.

ex.) **c.print(x)**

c is an object of the Console class and `println( )` is a member method of the Console class

//Example of a programmer defined class:

//Only a portion of the application and implementation files for the Example class

```
import hsa.Console; // or import hsa.*;
public class Example
{
    static Console c;
    public static void main(String[ ] args)
    {
        c = new Console( );
        Circle circleA, circleB;
        circleA = new Circle( );
        circleB = new Circle( );
        circleA.output( );
        circleB.output( );
        circleA.area( );
        circleB.area( );
    }
    public class Circle
    {
        public void output( )
        {
            c.println("Center is " + x + "," + y);
            c.println("Radius is " + r);
        }
        public void area( )
        {
            double pi = 3.14;
            c.println("The area is " + pi*r*r);
        }
    }
}
```

Name the following from the example above:

- 1) the two classes (i.e. the *type* like **int** or **double**) -
- 2) the two objects (i.e. the *variables* of the class) -
- 3) the two class member methods associated with the objects -

### II. Void Methods with No Parameters

ex.1):

```
public class Example1Test
{
    public static void main(String[ ] args)
    {
        Example1 song = new Example1( );
        song.refrain1( );           // 4 method calls
        song.refrain2( );
        song.refrain3( );
        System.out.print("Dog ");
        song.refrain2( );
    }
}

public class Example1           // class heading
{
    public void refrain1( )      // first method signature - no semicolon
    {                             // or also called heading
        System.out.print("Old MacDonald had a farm.\n");
    }

    public void refrain2( )      // second method signature - no semicolon
    {
        System.out.print("E I E I O\n");
    }

    public void refrain3( )      // third method signature - no semicolon
    {
        System.out.print("and on his farm he had a \n");
    }
}
```

ex.2)

```
public class Example2Test
{
    public static void main(String[ ] args)
    {
        Example2 payroll = new Example2( );
        int HOURS = 40;
        int payPerHr = 8;
        payroll.inputData( );           // 3 method calls:
        payroll.calcMethod(HOURS, 7.5); // Actual parameters
        payroll.calcMethod(HOURS + 2, payPerHr); // Multiple call #1
    }
}

public class Example2 // class heading
{
    public void inputData( ) // method Definition:
    { // method Signature
        int id; // or method Heading
        id = 12345; // method Body
        System.out.println("ID is " + id);
    }

    public void printMethod(double salary)
    {
        System.out.println("Your pay is " + salary);
        return; // Optional return
    }

    public void calcMethod(int hrs, double payPerHr,) // 2 Formal Parameters:
    {
        int overtimeHrs = 5; // local variable of overtimeHrs
        grossPay = hrs * payPerHr + payPerHr * 1.5 * overtimeHr;
        System.out.println("Your gross pay is " + grossPay);
    }
}
```

### *Introduction to Method Terminology*

- A) Flow of control - see example
- B) Call to a Method - using the method name and parameter list in the program
  - ex.1) `inputData();`
  - ex.2) `calcMethod(HOURS, payPerHr);`
- C) Multiple Calls to a Method - calling the same method more than once
- D) Actual Parameters - those in the call to the method
  - Can consist of: literal values, constant values, variables, variable or numeric expressions
- E) Formal Parameters - those in the method heading
  - Note 1: They must be preceded by the data type.
  - Note 2: The actual and formal variables must match in number and should match in type (or type coercion will occur).
  - Note 3: The formal parameters are also local variables (see next page) where they are already declared in the heading so do not declare them again!
- F) Method Definition - consists of two parts:
  - 1) the method heading (must list the variables)
  - 2) the body (block of code) for the method between the set of braces { }
- G) Local Variables - those declared in a block and are not accessible outside of that block
- H) **return** Statement - *optional* in void methods, but if used, then the syntax is:  
**return;**
- I) Value Parameter - a formal parameter that receives a copy of the contents of the corresponding actual parameter

ex.3)

```
public class Example3Test
{
    public static void main(String[ ] args)
    {
        Example3 song = new Example3( );
        song.refrain1( );
        song.refrain3( );
        System.out.print("Dog ");
        song.refrain2( );
    }
}
```

```
public class Example1           // class heading
{
    public void refrain1( )      // first method signature - no semicolon
    {
        System.out.print("Old MacDonald had a farm.\n");
        this.refrain2( );
    }

    public void refrain2( )     // second method signature - no semicolon
    {
        System.out.print("E I E I O\n");
    }

    public void refrain3( )     // third method signature - no semicolon
    {
        System.out.print("and on his farm he had a \n");
    }
}
```

Note: A Java method can not contain another method definition (unlike Pascal).

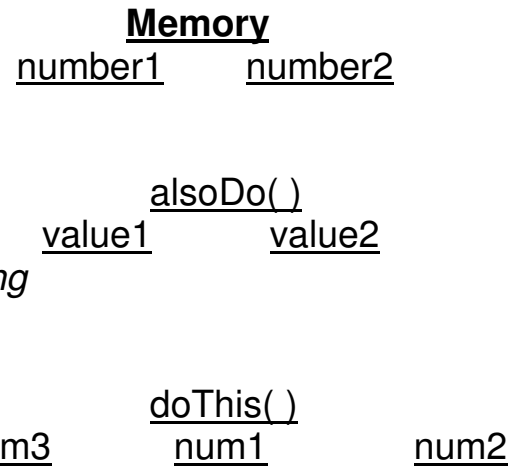
Note: A Java method can call another method.

# AP Programming - Chapter 2 Lecture

## II. Void Methods with Value Parameters

ex.4)

```
Ex4 test = new Ex4( );
int number1 = 1;
float number2 = 2.5;
test.doThis(number1, number2); // method call
test.alsoDo(number1, number2) // method call
System.out.println(number1 + " " + number2);
public class Ex4 // class heading
{
    public void alsoDo(int value1, float value2)
    {
        System.out.println(value1 + value2);
    }
    public void doThis(int num1, float num2)
    {
        int num3 = 3;
        num1 = num1 * 2;
        System.out.println(num1 + num2 + num3);
    }
}
```

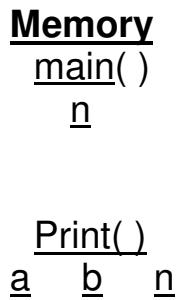


**Output:**

---

ex.5)

```
Ex5 test = new Ex5( );
int n;
n = 2;
test.print(3, n);
test.print(n, n);
test.print(n+n, 12);
System.out.println("Bye");
public class Ex5
{
    public void print(int a, int b)
    {
        int n;
        n = a + b;
        System.out.println(a + " " + b + " " + n);
    }
}
```



**Output:**

---

## III. Value-Returning Methods, Scope and Lifetime of Identifiers

### I. Value-Returning Methods

A value returning method is one that returns a single value to be used directly in an expression.

A) Math class methods - Methods such as **sqrt** and **abs** are value returning methods that are built in to the Math class.

ex.1) `root = Math.sqrt(9.0);`

B) Programmer-Defined Methods

Syntax of Value-Returning Method Definition:

```
DataType MethodName(FormalParameterList)    // method header
{
    statement(s);                            // method body
    return value;                            // or return(value)
}
```

Note: The call to a value-returning method must be part of an expression. It can not be a stand alone statement such as a **void** method. If it is a stand alone expression then the call is not an error. The method is still done but the **return** in the method body is ignored by the computer.

Note: **return;** is not allowed in a value-returning method only in a **void** method is **return;** allowed

Note: Any statements after the **return** statement will be ignored since the **return** statement signals the end of the method.

Note: If the value of the expression in the **return** does not match the data type in the method to be returned then a type coercion takes place.

### III. Value-Returning Methods continued

ex.1) Pass to a method the pizza diameter and the cost of the pizza and return the cost per square inch, i.e. cost per area.

```
public class Example1Test
{
    public static void main(String[ ] args) throws IOException
    {
        Ex1 pizza = new Ex1( );
        int theDiameter, pricePerArea;
        double cost;
        BufferedReader keyboard = new BufferedReader(new
        InputStreamReader(System.in));
        System.out.println("Type in the diameter then <Enter>");
        input = keyboard.readLine( );
        theDiameter = Integer.parseInt(input);
        System.out.println("Type in the cost then <Enter>");
        cost = keyboard.readLine( );
        pricePerArea = pizza.unitprice(theDiameter, cost);           // method call
        System.out.println("The cost per square inch is " + pricePerArea,2);
    }
}

public class Ex4
{
    public double unitprice(int diameter, double price)           // method header
    {
        final double PI = 3.14159;
        double radius, area, costPerArea;                          // local variables
        radius = diameter/2;                                       // This is wrong! Why?????
        area = PI * radius * radius;                               // answer: needs to be 2.0
        costPerArea = price/area;
        return (costPerArea);                                     // optional parenthesis
    }                                                               // the return could also have been
                                                                    // return (price/area);
}
}
```

### Memory Simulation:

<u>theDiameter</u>	<u>cost</u>	<u>pricePerArea</u>
10	5.25	14.95993238

<u>diameter</u>	<u>price</u>	<u>radius</u>	<u>area</u>	<u>costPerArea</u>
10	5.25	5	78.53975	14.95993238

**Output:** The cost per square inch is 14.96

Note: The method call above could have been :  
System.out.println("The cost per square inch is " +  
Unitprice(theDiameter, cost));

### III. Value-Returning Methods continued

ex.2) A method call nested inside another method call:

```
System.out.println(test.callEx(5,test.callEx(7,9));
```

```
int callEx(int a, int b)      a      b
{
    return a + b;
}
```

**Output**

ex.3) Layered methods:

```
{
    float a;
    a = this.callFirstOne(1.5);
    System.out.println(a);
    System.out.println("Pop");
}

public float test.callFirstOne(float m)
{
    System.out.println("Snap");
    m = m + 2.0;
    m = this.callNextOne(m);
    return m;
}

public float callNextOne(float n)
{
    System.out.println("Crackle");
    n = n + 5.0;
    return n;
}
```

**Memory**

a

m

n

**Output**

IV. Scope - the region of program code where it is legal to reference (use) an identifier

- A) Local Scope - the scope of an identifier declared inside a block extends from the point of declaration to the end of the block
- B) Global Scope - the scope of an identifier declared outside all methods extends from the point of declaration to the end of the entire program.
- C) Name Precedence (name hiding) - when one declares a local variable with the same name as a global variable, the local variable takes precedence within the method.

ex.1)

```
public class Example1
{
    int alpha = 2;
    int beta = 3;

    public static void main(String[ ] args)
    {
        callMethod( );
        int alpha = 5;
    }

    public static void callMethod( )
    {
        int alpha = 4;
        int delta;
        delta = alpha + beta;
        System.out.println(delta);
    }
}
```

- D) Lifetime - The period of time during program execution when the variable has memory allocated to it. A variable has memory allocated to it when it is declared and defined. This storage location (memory) is destroyed (deallocated) when the method where that variable is declared is exited.

### V. Creating a Program

A) *Implementation* file - consists of the class member function definitions (specification / heading & body) and field instance variables

```
public class Mycircle
{
    public Mycircle( )                // default constructor (initializer)
    {
        xcenter = 0;
        ycenter = 0;
        radius = 1;
    }
    public Mycircle(int x, int y, int r)    // constructor (initializer)
    {
        xcenter = x;
        ycenter = y;
        radius = r;
    }
    public double area( )
    {
        return (3.14*radius*radius)
    }
    public double perimeter( )
    {
        return (2*3.14*radius);
    }
    public void changeTheRadius(int r)
    {
        radius = r ;                // or this.radius r is also called the explicit parameter
    }                                // this is also called the implicit variable or the object
    public void output( )           // that called the method
    {
        System.out.println("center: " + xcenter + "," + ycenter + " radius: " + radius);
    }
    private int xcenter;
    private int ycenter;
    private int radius;
}
```

1) Explaining the parts of the implementation code:

a) Constructors - they initialize the member values and have the same name as the class

ex.1) **public** Mycircle( )

ex.2) **public** Mycircle(**int** x, **int** y, **int** r)

Note: A constructor method is automatically called when the class object is created (declared called instantiated). The constructor that is called is the one that matches with the number of actual parameters.

Note: A class can have many constructors but only one default constructor and if there is more than one constructor one of them *must* be the default constructor.

Note: A constructor does not have a return type.  
A constructor does not have **void** or a data type in front of it.

\*Using the same name more than once in a program is called **overloading**.

\*The process of “hiding” the data and providing methods for data access is called **encapsulation**.

B) Test file (sometimes called the application file, client file, or driver) - software that declares and manipulates objects of a particular class, i.e. the file that contains the **main( )** method that actually uses the defined member functions and variables

of a class by declaring variables that are of a class type.

*object* (instance) - A variable of a class type.

Note: When an object is declared we do not use the word declare but the word *instantiate*

ex.1)

```
public class MyCircleTest
{
    public static void main(String[ ] args)
    {
        Mycircle circle1, circle2;
        circle1 = new MyCircle(3, 4, 10); // instantiation i.e. declaration of Mycircle objects
        circle2 = new MyCircle( );
        circle1.output( );
        circle2.output( );
        System.out.println(circle1.area( ));
        System.out.println(circle2.area( ));
        System.out.println(circle1.perimeter( ));
        System.out.println(circle2.perimeter( ));
        circle1.changeTheRadius(5);
        circle1.output( );
        circle2.output( );
        circle1.change_the_radius(6);
        circle1.output( );
        circle2.output( );
    }
}
```

### C) Constructors

Calls from the test file to the constructor where the class name is TheClass:

ex.1)

```
private class TheClass
{
    public TheClass(int NewInfo, char moreInfo); { . . . }
    public TheClass( ); { . . . }
    private int info;
    private char moreInfo;
}
```

Legal Calls:

```
TheClass object1 = new TheClass(50, 'R');
```

```
TheClass object2;
object2 = new TheClass( );
```

```
TheClass object3 = new TheClass( );
```

```
TheClass object4;
object4 = TheClass(10, 'a');
```

Illegal Calls:

```
TheClass object( );
```

```
TheClass object = new TheClass('A', 20);
```

```
object = TheClass;
```

```
object.TheClass(10,'B');
```

```
object.TheClass( );
```

```
object.TheClass;
```

```
TheClass(10,'r');
```

```
TheClass( );
```

## AP Programming - Chapter 2 Lecture

Extra Topic needed for the Unit 2 program: String data

A string like your name, Chris Doe, can be stored in a String variable. A String is not a primitive data type like **int** or **double** but is an object and it has a shortcut built into it.

Recall: Car mustang = **new** Car( );

Now for strings: String name = **new** name("Chris Doe");

Now for the shortcut: String name = "Chris Doe";

### Terminology

Encapsulation – “Hiding” code from the client (user)

Overloading – method name or operator use more than once with different results

Implicit Parameter – ex.) obj.get(x) the obj i.e. **this** is the implicit parameter

Explicit Parameter – ex.) obj.get(x) the x is the explicit parameter

Logic Error – occurs when run, not caught at compiled time

Syntax Error – caught at compiled time

Library – C++ term; a location where frequently used code can be accessed quickly

Package – file that can contain multiple Classes

Class – blueprints or factories for objects; it contains the code that specifies the data and the actions/operations (methods) that can use the data

Object – a specific instance of a class; it has data and methods associated with it

Method – a block of code that performs some action

Instance Field – the data an object has stored in it

Instantiation – declaration of an object

Constructor – method like code that assigns values to the instance fields

Access Specifier – **public, private, protected**

Method Header – the whole line including access specifier, return type (including void), method name, and parameter list

Method Signature - method name and parameter list

Java Interpreter

Java Bytecode (virtual machine)

Java Compiler