ALGONQUIN COLLEGE

CST8130 – Data Structures

September 12, 2017

Professor : Dr. Anu Thomas
Email: thomasa@algonquincollege.com
Office: T314

REVIEW of :
Arrays
Inheritance
Polymorphism
Files
Exception Handling

Data Types

Primitive – boolean, byte, char, short, int, long, float, double

Reference – everything else

- objects, arrays etc

How are they different – in memory and in how you write code (hint: think about what needs a **new** operation)

Constructors

- Methods with the same name as the class
- Execute when you instantiate an object

ClassA classAObj = new ClassA();

If you have reference (non-primitive) data members in the class, you need to think about what memory to allocate in the constructor

Array Basics

What is an array?

An **array** is (in native language) a construct that holds a ***fixed size set*** (in a block) of variables of the ***same datatype***.

How do we declare an array?

Syntax: dataType [] arrayName;

This declares a *reference* variable called arrayName which can hold an array of dataType

Examples: int [] nums;
Shape [] shapes;

nums can hold a set of integer values.

shapes can hold a set of Shape objects.

Array basics

We then need to allocate the actual block of memory to hold the array elements

Syntax:

arrayName = new dataType[size];

Examples:

nums = new int[5]; // a block of 5 ints

shapes = new Shape[20]; // a block of 20 references to Shape objects

Array basics

- Can declare and allocate at same time
- Can use variable for size – but must have a value at the time array is allocated
- Access to each element is through index

Examples:

```
int numElements = 5;
float grades = new float[numElements];
for (int i=0; i < numElements; i++)
        grades[i]= 0.0f;
```

Inheritance, Encapsulation, Polymorphism

Inheritance basics

“has a” relationship:

we can say that one class **has a relationship with** another class…..this type of relationship implies that the class contains one (or more) objects of the other class.

“is a” relationship:

we can say that one class **is** another class…..this type of relationship implies that the class contains all of the other class…plus more. We will implement this type of relationship with inheritance.

Inheritance basics

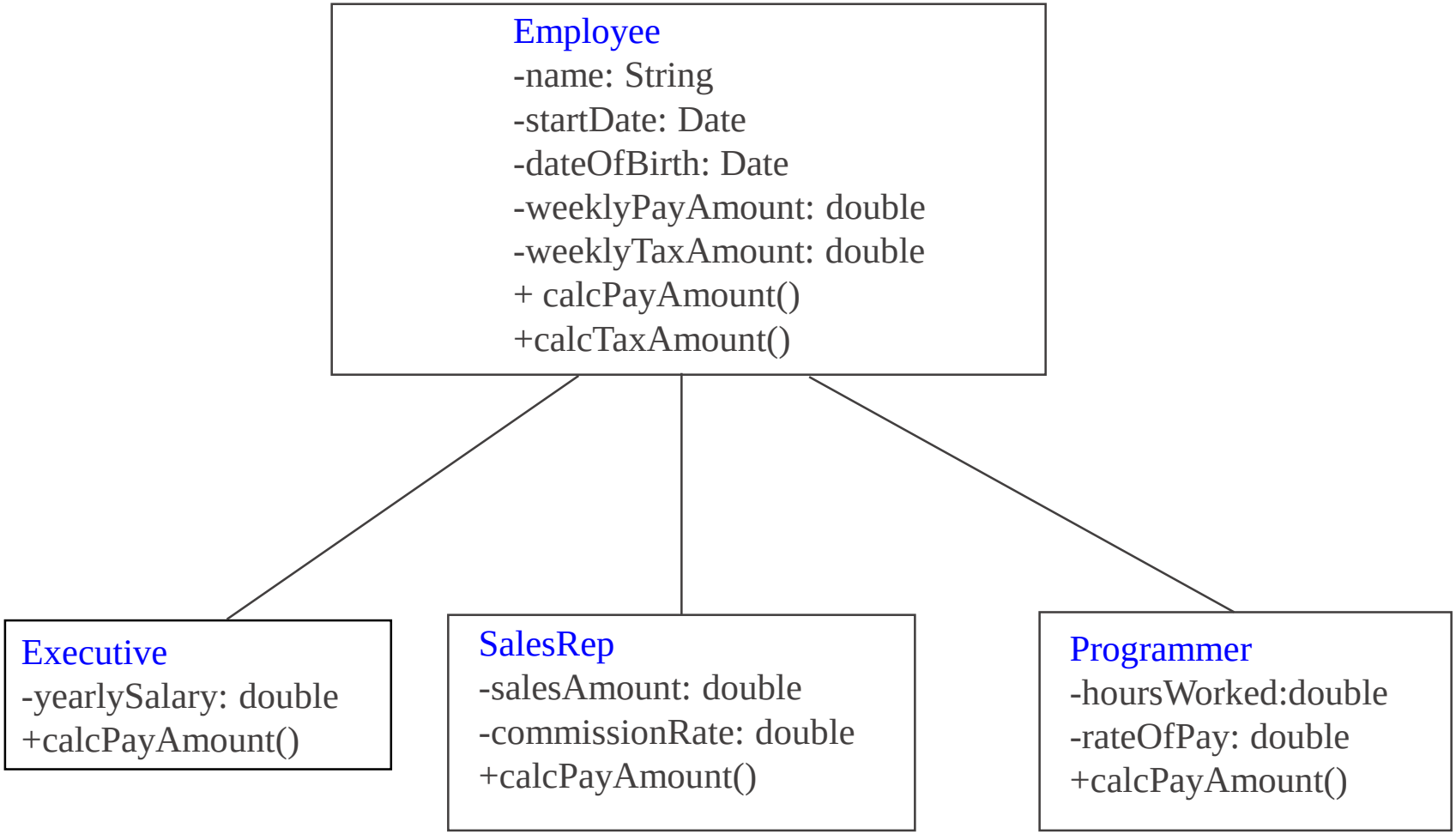
Example: three kinds of **Employees**

- **Executives** that get paid yearly,
- **Sales Reps** that get paid by commission
- **Programmers** that get paid hourly.

They all are employees – so they all have an identification number, a start date, a pay amount, income tax deductions, etc.

With "is a" relationships - we put the common elements (fields and processing) in the superclass (also called Base class in other languages) (Employee in this case)…. and derive classes from it **using the extends keyword** for each of the different types of employees (called subclasses) – (Executive, SalesRep and Programmer) with the differences put into these subclasses.

Inheritance Diagrams



Private, Public, Protected labels

public:

- any member that is declared public can be seen inside and outside the class
- by convention, we use this for method members

private:

- any member that is declared private can only be seen inside the class
- by convention, we use this for data members

protected:

- any member that is declared protected can be seen inside the class and inside any inherited class. So this is a cross between public and private.

Access in class and in main

```
class ClassA  {    // note we can access protected inside the class
    private      int privA = 4;
    protected   int protA = 6;
    public       int pubA = 10;

     public String toString () {
          return "privA:"  + privA + "  protA:" +protA +"  pubA: "+pubA ;
    }
     public void setPrivA(int a){
     privA = a;
     }
}

public static void main (String args[])  {
          ClassA objA = new ClassA();
       //  objA.privA = 5;          // can't access private outside of class -
            objA.protA = 10;     // can access protected outside of class if in same package
            objA.pubA = 15;     // CAN access public outside of class
            System.out.println (objA.toString());   // CAN access public outside of class, will display  privA:   protA: 10   pubA:15

            objA.setPrivA(20);
            System.out.println (objA.toString());  // CAN access public outside of class, will display  privA:20   protA: 10   pubA:15
 }
```

Access in inherited class

```
class ClassB extends ClassA {
    protected  int protB = 1;
    public       int pubB = 2;
    private      int privB = 3;


        @Override  // helps detect errors at compile time
        public String toString() {
                // statement below calls the toString  method in ClassA
                return super.toString() + “protB: ”+ protB + “privB: “ + privB + “pubB: “ +
pubB;
        }
        public int  calcTotal () {
                return protA + protB + privB + pubA + pubB; // note – privA not included – it
can’t be accessed in this class
        }
```

Now to use methods…..

```
public static void main (String [] args){
        ClassB   objB1 = new ClassB();
        ClassA   objB2  = new ClassB();  // can do this…..polymorphism!
        ClassA  objA = new ClassA();


        System.out.println ("objA " + objA.toString());  //  objA  privA:4  protA:6 pubA: 10
        System.out.println ("objB1 " + objB1.toString());  // objB1 protB: 1 privB:  3 pubB: 2
        System.out.println ("objB2" + objB2.toString());  // objB2 protB: 1 privB:  3 pubB: 2


        //  System.out.println ("objA " + objA.calcTotal());
        System.out.println ("objB1 " + objB1.calcTotal());  // objB1 22
    // System.out.println ("objB2 " + objB2.calcTotal());
}
```

abstract keyword – with a class

When we use inheritance, we can use the keyword abstract to refer to a class or a method inside a class.

abstract class ClassA {…..etc }

When used with a class definition abstract means that you can never instantiate an object of type ClassA

(you CANNOT declare ClassA objA = new ClassA();)

class ClassB extends ClassA { …etc }

You can instantiate objects of ClassB……so you can declare ClassA objB = new ClassB();

abstract keyword … with methods

```
class ClassA {
        abstract  public void method1() {    etc….  }
```

Declaring a method abstract means that it MUST be overloaded in all derived classes

```
Class ClassB extends ClassA {
 must have….
      public void method1 () {….code here….}
```

Application of abstract

In our Employee example:

- it is realistic for us to declare the calcWeeklyPayAmount method as abstract – which would force every subclass of Employee to implement this method
- It could also be realistic for us to declare the Employee class as abstract to force that you can't create an Employee object – you can only create an Executive, Programmer or SalesRep.

final keyword

In general, final means “cannot be changed”

final int TAXRATE = 0.30; // often used to document constant values in program – convention is to use all capital letters for constant name

// cannot execute this

// TAXRATE = 0.40;

final (contd.)

a method that is declared final in a superclass cannot be overridden in a subclass

- Methods declared private are implicitly final
- Example – in our Employee class – it would be reasonable to declare the calcWeeklyTaxAmount method as final so that no subclasses redefine this method – because the logic is that TaxAmount is not based on type of pay – only on amount of pay.

A class that is declared final cannot be used as a superclass

Static – as variable or data member modifier

Static means “only one is created and is shared”

Examples: variable bank account number…

USE OF THIS IS EXCEPTIONALLY RARE…you shouldn’t need to do this

Static – as method modifier

Static means do not need object to execute

USE OF THIS IS EXCEPTIONALLY RARE…you should only need to use this for methods in class that contains your main method

Object Oriented Concepts - Classes

A **class** is a description of set of objects that share common attributes and a common behavior (operations). There can be different type of classes

- fully implemented or concrete classes;
- classes that are partially implemented (abstract classes);
- classes which only specify some behavior but do not provide implementation at all (pure abstract classes or interfaces).

Once a class is defined in a program it becomes a new data type and it can be used to create (instantiate) objects of that data type.

That is why the class definitions are called abstract data types.

Object Oriented Concepts - Objects

- An **object** has a state (attributes), behavior (operations), and identity.
- The structure and behavior of similar objects are defined in their originating class.
- An object is an instance of a class. The terms object and instance are interchangeable.
- The concept of a class and object are very tightly interconnected. We cannot speak about an object without referring in some way to its class. However, there are important differences between these two terms.
- Whereas an object is a concrete entity that exists in time and space in a program, a class represents only an abstraction, the model of a set of similar objects.

Object Oriented Concepts - Encapsulation

- Most real-world objects are defined in terms of characteristics (or attributes) that describe the object and behaviors (or actions) the object can perform.
- The object-oriented technique that allows us to model real-world objects and provides some protection mechanisms in our programs is called **encapsulation**.
- Protection is obtained through the private/protected/public declaration of data fields and methods

Object Oriented Concepts - Inheritance

- The real-world objects like people, animals, and cars are often understood by grouping them in a set of related classes and objects that all share common attributes and behavior.
- The object oriented technique that allows us to organize classes and objects in hierarchical relationships is called **inheritance**.

Object Oriented Concepts - Polymorphism

- The last major contribution of object-oriented programming is its mimicking of how real-world objects describe the actions they perform.
- Real-world objects often perform the same type of action yet perform it in their own slightly different way. A person runs, a dog runs, a stream runs, a car runs. A person, a dog, a stream can run in many different ways. All these real-world objects use the verb "run" to describe one of their actions.
- The object-oriented technique that allows us to use one name for many different implementations of an action is called **polymorphism**.

What does this mean????

- Use a name for classes that accurately describes what the class is modelling
- Put logically into the base class what is “common”
- Put into each inherited class the “differences” – can be data or methods
- Write your code so that only the creation of the initial object needs to be unique code…all the rest uses polymorphism – one set of instructions for all objects

Files

Sequential Files

- Sequential files can be read from or printed to from top line to bottom, left to right (the same way we read).
- We can either read from a file ……..or print to a file….with sequential access you cannot do both at same time to same file.
- Files can be of any size……the end of the file is denoted by a special sequence of codes called the “end of file” marker.
- Our files will need to be DOS files…..maximum 8 chars in name followed with a . and a 3 char extension (usually .dat or .txt)

Output to file logic

OPEN FILE for output – connects the hard-drive file to an object in your program (creates file if it doesn't exist)

```
IF open not successful
        end of program
ELSE
    WHILE  there is data to put to file
        PUT data to file
    ENDWHILE
    Close file
ENDIF
```

Input from file logic

OPEN FILE for input – connects the hard-drive file to an object in your program (file must exist)

IF open not successful

    end of program

ELSE

    WHILE there is data to be read from file (ie not EOF)

        READ data from file

    ENDWHILE

    Close file

ENDIF

java.io Package

FileInputStream – for byte-based input from a file (inherits from **InputStream**)

FileOutputStream – for byte-based output to a file (inherits from **OutputStream**)

FileReader – for character-based input from a file (inherits from **Reader**)

FileWriter – for character-based output to a file (inherits from **Writer**)

BufferedInputStream A BufferedInputStream adds the ability to buffer the input and to support the mark and reset methods.

BufferedOutputStream The class implements a buffered output stream.

BufferedReader Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

BufferedWriter Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.

InputStream

Most of the Input/Output classes extend the InputStream and OutputStream correspondingly.

InputStream Methods

- int read()
- int read(byte[])
- int read(byte[],int,int)

These methods return a byte read from the stream or –1, which indicates EOF

- void close()
- int available()
- long skip(long)

All input stream methods throw IOException. Additionally, some throw NullPointerException.

OutputStream

OutputStream Methods

- void write()
- void write(byte[])
- void write(byte[],int,int)
- void close()
- void flush()

Sample Binary IO File

```
/** A class used to demonstrate binary file input and output */
import java.io.*;
public class DataIO {

   public static void main( String[] args ) throws IOException {

      DataInputStream  istream = null;
      DataOutputStream ostream = null;
      File f = new File("numbers.dat" );
      try{
            ostream = new DataOutputStream( new FileOutputStream( f ));
            for(int i = 100; i <= 110;++i)
                  ostream.writeInt(i);
            ostream.close();
```

```
 if(f.exists()){
        istream = new DataInputStream( new FileInputStream( f ));
        for(int i = 0; i <= 10;++i)
         System.out.println("Number: " + istream.readInt());
      }
     }catch( IOException ioe ) {
         System.out.println( ioe );
      }
      finally {
         istream.close();
         ostream.close();
      }
   }
}
```

Results …

```
/*File: numbers.dat
  d  e  f  g  h  i  j  k  l  m  n
*/
/*Output
Number: 100
Number: 101
Number: 102
Number: 103
Number: 104
Number: 105
Number: 106
Number: 107
Number: 108
Number: 109
Number: 110
*/
```

Use of Scanner to read “mixed” data

```
String fileName = new String();
Scanner inFile = null;

System.out.print("\n\nEnter name of file to process: ");
fileName = keyboard.next();
File file = new File(fileName);
try {
        if (file.exists()) {
                inFile = new Scanner(file);
                        while (inFile.hasNext())
                                ……..
        }
} catch (IOException e) {
        System.out.println("Could not open file...." + fileName + "exiting");
}
```

….then you can use ….. inFile.nextInt(), etc…..

Exception Handling

Exception Handling

The process of dealing with the exceptions in a program is called ***exception handling***. In Java, exception handling is implemented by a ***try block*** construct. The try block has the following syntax and semantics.

Syntax:

```
try{
   // statement(s) or method call(s) that can throw exceptions and for that
   // reason they must be monitored
}
catch(Exception_class_name     object_reference ){
 // statements that handle exceptions
}
finally{
 // statements that always will be executed at the end of try block
}
```

Semantics of Exception Handling

- At least one statement that throws an exception should exist in a **try** block.
- The **try** block includes zero, one or more than one ***catch*** clauses and an optional ***finally*** clause.
- The **finally** clause is not optional if the **try** block does not have a **catch** clause. If the **finally** clause is present, its statements are always executed regardless of whether an exception is thrown and caught or no exception is thrown at all. The finally clause will be executed even there is a ***return*** or ***break*** statement in the try or catch clauses. Only ***System.exit()*** call can prevent the finally clause from execution.

Simple Example

```
boolean cont = true;
do {
        System.out.println("Please enter a start date:");
        System.out.println("\nPlease enter a month:");
        int month =0;
        try {
                Scanner keyboard = new Scanner(System.in);
                 String input = keyboard.nextLine();
                month = Integer.parseInt(input);
                if (month <= 0 || month > 12) {
                        System.out.println("\nPlease enter a valid month  1-12:");
                        input = keyboard.nextLine();
                        month = Integer.parseInt(input);
                }
                cont = false;
              } catch (IOException e) {
               System.out.println("Input by user could not be read….restarting");
               // any code that would be needed to recover could be put here…none in this case
              } catch (NumberFormatException e) {
                 System.out.println("Input by user was not a number…..restarting");
                // any code that would be needed to recover could be put here…none in this case
              }
}  while (condition);
```

Memory Allocation example

```
try {
        size = 10;   // default to size of 10
        numbers = new int[size];
        // if allocation doesn't work - an OutOfMemoryError Exception will  be thrown
} catch (OutOfMemoryError e) {
        System.out.println("Not enough memory for array allocation");
        size = 0;
        numbers = new int[size];
}
```

Questions?