CMR INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | | |

Internal Assesment Test - I

Sub:	OBJECT ORIENTED CONCEPTS					Code:	15CS45
Date:	30/03/2017 (8.30 TO 10 AM)	Duration:	90 mins	Max Marks:	50	Sem: IV	Branch: ISE

	Answer Any FIVE FULL Questions	Marks	OBE CO	OBE RBT
1(a)	List and explain the java buzzwords. **(Any 7 buzzwords 1*7)**	[07]	CO1	L2
1(b)	How "compile once and run anywhere" is implemented in java, explain.	[03]		
2(a)	Define Type Conversion. What are its Types? Explain with an example **Definition 1, 2 types with eg 2 each**	[05]	CO1	L2
2(b)	What are classes and object? How is reference object different from an object? **Class -2, object -2, reference object -1**	[05]	CO1	L3
3(a)	What is a constructor? How is it different from methods? What are the types of constructors available in Java? **Constructor -2 , difference – 2 , types 3 each**	[10]	CO1	L3
4(a)	Define function Overloading. Write a C++ program with three overloaded function area() to find area of rectangle, area of rectangular box, area of circle. **Definition – 2 program -8**	[10]	CO3	L3
5(a)	Explain the different uses of final keyword. Give example for each**. 3 types 2 marks each**	[06]	CO2	L2
5(b)	EXPLAIN the following Operators: a) >>> b)short circuit logical operators **2 marks each**	[04]		
6(a)	Explain the keywords "super" and "this" in multi-level inheritance with example. **Super definition 2 program 3, this definition 2 program 3**	[10]	CO3	L2
7(a)	What is array in Java? Explain the operation of Enhanced For on a matrix. **Array – 2, for each syntax -2 , program - 6**	[10]	CO3	L3

INTERNAL ASSESMENT TEST – I SOLUTION
15CS45 - OBJECT ORIENTED CONCEPTS

1a. List and explain the java buzzwords(Java Features) [7]

- **Simple**

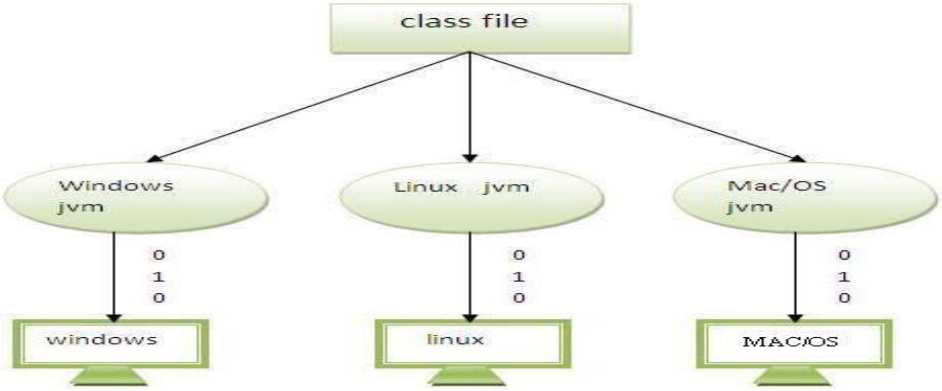
Syntax is based on C++ (so easier for programmers to learn it after C++). Java removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc. No need to remove unreferenced objects because there is Automatic Garbage Collection in java.

- **Object-oriented**

Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior. Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules. Basic concepts of OOPs are: Object, Class, Inheritance, Polymorphism, Abstraction, Encapsulation

- **Platform Independent**

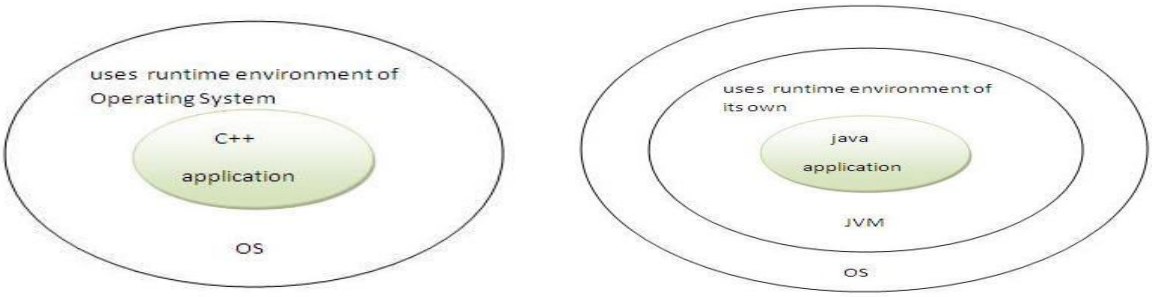
A platform is the hardware or software environment in which a program runs. There are two types of platforms software-based and hardware-based. Java provides software-based platform. The Java platform differs from most other platforms in the sense that it's a software-based platform that runs on top of other hardware-based platforms.It has two components:
Runtime Environment
API(Application Programming Interface)



Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere (WORA).

- **Secured**

Java is secured because: No explicit pointer, Programs run inside virtual machine sandbox.



Classloader- adds security by separating the package for the classes of the local file system from those that are imported from network sources.
Bytecode Verifier- checks the code fragments for illegal code that can violate access right to objects.
Security Manager- determines what resources a class can access such as reading and writing to the local disk.

These security are provided by java language. Some security can also be provided by application developer through SSL, JAAS, cryptography etc.

- **Robust**

Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points make java robust.

- **Architecture-neutral**

There are no implementation dependent features e.g. size of primitive types is set.

- **Portable**

We may carry the java bytecode to any platform.

- **High-performance**

Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

- **Distributed**

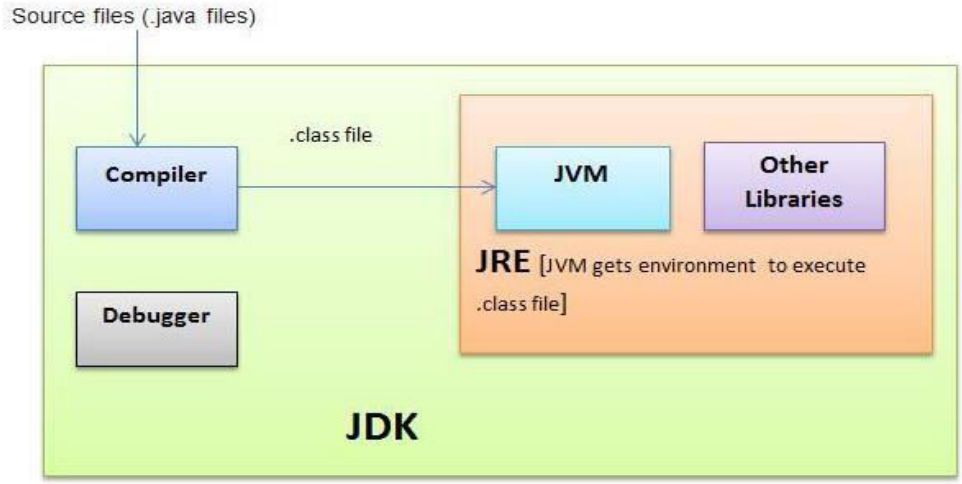
We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

- **Multi-threaded**

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the same memory. Threads are important for multi-media, Web applications etc.

1b. How "compile once and run anywhere" is implemented in java, explain. [3]

- Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine.
- Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.
- **JVM (Java Virtual Machine):** As we all aware when we compile a Java file, output is not an "_.exe" but it's a " _.class" file. The " _.class" file consists of Java byte codes which are understandable by JVM. Java Virtual Machine interprets the byte code into the machine code depending upon the underlying operating system and hardware combination. It is responsible for all the things like garbage collection, array bounds checking, etc… JVM is platform dependent. The There are different JVM implementations for different OS or machines.



2a. Define Type Conversion. What are its types? Explain with example. [5]

When you assign value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly. A conversion among different primitive types i.e. char, byte, short, int, long, float, and double is allowed. Booleans cannot be converted to other types. For the other primitive types (char, byte, short, int, long, float, and double), there are two kinds of conversion: **implicit** and **explicit**.

Implicit conversions (Widening):
An implicit conversion means that a value of one type is changed to a value of another type without any special directive from the programmer. A char can be implicitly converted to an int, a long, a float, or a double. For example, the following will compile without error:

```
char c = 'a';
int k = c;
long x = c;
float y = c;
double d = c;
```

For the other (numeric) primitive types, the basic rule is that implicit conversions can be done from one type to another if the range of values of the first type is a subset of the range of values of the second type. For example, a byte can be converted to a short, int, long or float; a short can be converted to an int, long, float, or double, etc.

Explicit conversions (Narrowing):
Explicit conversions are done via **casting**. The name of the type to which you want a value converted is given, in parentheses, in front of the value. For example, the following code casts a value of type double to a value of type int, and a value of type double to a value of type short:

```
double d = 5.6;
int k = (int)d;
short s = (short)(d * 2.0);
```

Casting can be used to convert among any of the primitive types except boolean. Please note that casting can lose information; for example, floating-point values are truncated when they are cast to integers (e.g. the value of k in the code fragment given above is 5).

2b. What are class and object? How is reference object different from object? [5]

In object-oriented programming technique, we design a program using objects and classes. Object is the physical as well as logical entity whereas class is the logical entity only. An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tangible and intangible). The example of intangible object is banking system. An object has three characteristics:

- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.**

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.
Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

Object Definitions:
Object is a real world entity.
Object is a run time entity.
Object is an entity which has state and behavior.
Object is an instance of a class.
A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
A class in Java can contain:

- fields
- methods
- constructors
- blocks
- nested class and interface

A variable which is created inside the class but outside the method, is known as instance variable. Instance variable doesn't get memory at compile time. It gets memory at run time when object(instance) is created. That is why, it is known as instance variable.

```
class Student
{
        int id;//field or data member or instance variable
        String name;
        void inputid(int n)
        {
                id=n;
        }
        public static void main(String args[])
        {
                Student s1=new Student();     //creating an object/instance of Student
                s1.name = “aditi”;       //accessing instance member through object
                s1.inputid(10);            //accessing method through object
                System.out.println(s1.id);
                System.out.println(s1.name);
        }
}
```

Object and Object Reference:
Consider the the example below where Employee is a class.

```
Employee e;
Employee emp = new Employee();
```

These two statements have different effect memory-wise since they are reference types. The first statement just creates a reference, (pointer) to an instance of type Employee. This essentially means, the statement tells the compiler **"e" is going to point (refer) to an Employee object, but right now is pointing to nothing (null)**. The interesting part is, the reference itself, which is "e" is stored. So, here you create just the reference.

The second statement however, does more than this. "emp" is allotted memory as a reference as in the previous case, but the use of new keyword creates an object and allots memory to it, at runtime, i.e dynamically. This statement tells the compiler **"emp" is going to refer to the Employee object that will be created as a result of the new keyword**. So, here you create a reference and an object that the reference variable refers to.

3a. What is a constructor? How is it different from methods? What are the types of Constructors available in Java? [10]

A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides, this is because the call for the constructor is made automatically at the time of object creation using the class name. Constructors are syntactically similar to a method but look a little strange because they have no return type, not even void. This is because the implicit return type of a class constructor is the class type itself (That means the object is created and the address is assigned, thus another assignment is not possible). It is the constructor‘s job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.
The different types of constructors are
Default Constructor - constructor without arguments/parameters
Parameterized Constructor - constructor with arguments/parameters

/* In the program below, the Box classes uses constructors (default and parameterized constructor to initialize the dimensions of a box. Depending on the parameters during object creation any one of the constructors will be called */

```
class Box
{
        double width,  height,  depth;
        // Below  is the default constructor for Box i.e., constructor without arguments/parameters
        Box()
        {       System.out.println("Constructing Box");
                width = 10;
                height = 10;
                depth = 10;
        }
        //Below is the parameterized constructor for Box i.e., this constructor requires 3 parameters
        Box(double w, double h, double d)
        {       width = w;
                height = h;
                depth = d;
        }

        // compute and return volume
        double volume()
        {       return width * height * depth;
        }
}
class BoxDemo
{       public static void main(String args[])
        {       // declare, allocate, and initialize Box objects
                Box mybox1 = new Box(); // calls the default constructor
                Box mybox2 = new Box(10, 20, 15);  // calls the parameterized constructor
                double vol;
                // get volume of first box
                vol = mybox1.volume();
                System.out.println("Volume is " + vol);
                // get volume of second box
                vol = mybox2.volume();
                System.out.println("Volume is " + vol);
        }
}
```

4a.Define function Overloading. Write C++ program with 3 overloaded function area() to find area of rectangle, area of rectangular box, area of circle. [10]

Overloading occurs when two or more methods in one class have the same method name but different parameters. Return type of method does not matter in case of method overloading, it can be same or different. Overloading happens at compile-time. In the example below in a class called Sum

we have 4 add() methods with different signature (parameter list is different). At the time of the method call depending on the parameter list, one of the four methods is executed.

```
class Area
{
  void area(int n1, int n2)
   {
      System.out.println("Area of rectangle = "+ (n1*n2));
   }
   void area(int n1, int n2, int n3)
   {
      System.out.println("Area of rectangle box = "+ (n1*n2*n3));
   }
   void area(int n1)
   {
      System.out.println("Area of circle = "+  (3.142*n1*n1));
   }

   public static void main(String args[])
   {
         Area obj = new Area();
         obj.area(20, 21);
         obj.area(20, 21, 22);
         obj.area(20);
   }
}
```

Output:
Area of rectangle = 420
Area of rectangle box = 9240
Area of circle = 1256.8

5a. Explain the different uses of final keyword. Give example for each. [6]

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

- Variable – stop value change
- Method – stop method overriding
- Class - stop inheritance

final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant). In the example below there is a final variable speedlimit, we are going to change the value of this variable, but it can't be changed because final variable once assigned a value can never be changed. Thus we get a compile time error.

```
class Bike9
{
 final int speedlimit=90;//final variable
 void run()
{
                speedlimit=400;   //trying to change the value of a final variable leads to compile time
error
```

```
        }
        public static void main(String args[])
{        Bike9 obj=new  Bike9();
 obj.run();
        }
}//end of class
```

final method

If you declare a method as final, you cannot override it anymore. The definition of the method becomes the last version of the method. In the example below there is a class called Bike in which a method run() is declared as final and the definition provided becomes the last version of the method. The class Honda which inherits or extends the Bike class tries to override (give new definition to an existing function) the method run().This is not allowed because the run() was declared as final. Thus a compile time error occurs.

```
class Bike
{
        final void run()
        {
                System.out.println("running");
        }

}
class Honda extends Bike
{
        void run() //overriding is not allowed because run() method is declared as final in Bike class
        {
                System.out.println("running safely with 100kmph");
        }

        public static void main(String args[])
        {
                Honda honda= new Honda();
                honda.run();
        }
}
```

final class

If you declare a class as final, you cannot extend or inherit it (the class cannot behave as super class anymore). In the example below Bike class is declared as final. So the Honda class cannot be inherit or extend the Bike class. Thus a compile time error occurs.

```
final class Bike
{
         void display()
        {
                System.out.println("running");
        }
}

class Honda extends Bike
{
        void run()
        {
                System.out.println("running safely with 100kmph");
        }
        public static void main(String args[])
```

```
        {
                Honda honda= new Honda();
                honda.run();
        }
}
```

5b. Explain the following Operators: a) >>> [2]

>>> (Unsigned right shift) In Java, the operator '>>>' is unsigned right shift operator. It always fills 0 irrespective of the sign of the number.

```
  class Test
  {         public static void main(String args[])
            {
                // x is stored using 32 bit 2's complement form.
                // Binary representation of -1 is all 1s (111..1)
                    int x = -1;
                    System.out.println(x>>>29);  // The value of 'x>>>29' is 00...0111
                    System.out.println(x>>>30);  // The value of 'x>>>30' is 00...0011
                    System.out.println(x>>>31);  // The value of 'x>>>31' is 00...0001
            }
  }
```

Output:

```
  7
  3
  1
```

5b. Explain the following Operators: b) short circuit logical operators [2]

The && and || operators are "short-circuit" logical operaors, meaning they don't evaluate the right hand side if it isn't necessary.
The & and | operators, when used as logical operators, always evaluate both sides.
There is only one case of short-circuiting for each operator, and they are:

- false && ... - it is not necessary to know what the right hand side is, the result must be false
- true || ... - it is not necessary to know what the right hand side is, the result must be true

6a. Explain the keywords "super" and "this" in multi-level inheritance with example [10]

Super Keyword

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword `super`. You can also use `super` to refer to a hidden field (although hiding fields is discouraged). Consider this class, `Superclass`:

```
public class Superclass {

   public void printMethod() {
      System.out.println("Printed in Superclass.");
   }
}
//Here is a subclass, called Subclass, that overrides printMethod():
public class Subclass extends Superclass {

   // overrides printMethod in Superclass
   public void printMethod() {
      super.printMethod(); //calls the printMethod() present in the Superclass
      System.out.println("Printed in Subclass");
   }
   public static void main(String[] args) {
```

```
        Subclass s = new Subclass();
        s.printMethod();    // calls the printMethod() present in the Subclass
    }
}
```

Within `Subclass`, the simple name `printMethod()` refers to the one declared in `Subclass`, which overrides the one in `Superclass`. So, to refer to `printMethod()` inherited from `Superclass`, `Subclass` must use a qualified name, using `super` as shown. Compiling and executing `Subclass` prints the following:

Printed in Superclass.
Printed in Subclass

Subclass Constructors

The following example illustrates how to use the `super` keyword to invoke a superclass's constructor. Invocation of a superclass constructor must be the first line in the subclass constructor. The syntax for calling a superclass constructor is

super(); // for default constructor

or:

super(parameter list); // for parameterized constructors

With `super()`, the superclass no-argument constructor is called. With `super(parameter list)`, the superclass constructor with a matching parameter list is called.

Note: If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor (default constructor) of the superclass. If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that there will be a whole chain of constructors called. It is called *constructor chaining*, and you need to be aware of it when there is a long line of class descent.

```
public class Superclass
{
   Superclass()
   {
            System.out.println(“Inside Superclass constructor”);
   }
    public void printMethod()
   {
       System.out.println("Printed in Superclass.");
    }
}
//Here is a subclass, called Subclass, that overrides printMethod():
public class Subclass1 extends Superclass
{
   Superclass()
   {
            super();
            System.out.println(“Inside Subclass1 constructor”);
   }
    // overrides printMethod in Superclass
    public void printMethod() {
       super.printMethod(); //calls the printMethod() present in the Superclass
       System.out.println("Printed in Subclass");
    }
public class Subclass2 extends Subclass1
{
   Superclass()
   {
            super();
            System.out.println(“Inside Subclass2 constructor”);
   }
}
```

```
    public static void main(String[] args) {
        Subclass2 s = new Subclass2();
        s.printMethod();    // calls the printMethod() present in the Subclass1
    }
}
```

Output:

```
Inside Superclass constructor
Inside Subclass1 constructor
Inside Subclass2 constructor
Printed in Subclass
```

this keyword

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

Usage of java this keyword

Here is given the 6 usage of java this keyword.

- this can be used to refer current class instance variable.
- this can be used to invoke current class method (implicitly)
- this() can be used to invoke current class constructor.
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this can be used to return the current class instance from the method.

The most common reason for using the this keyword is because a field is shadowed by a method or constructor parameter. The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

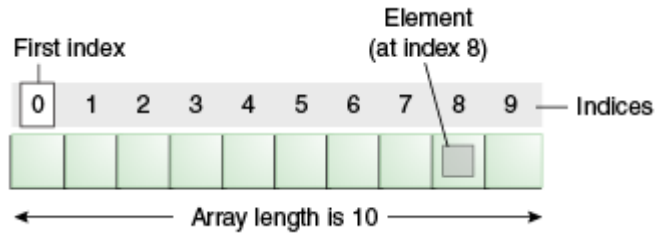
```
class Student
{
        int rollno;
        String name;
        float fee;
        Student(int rollno,String name,float fee)
        {
                this.rollno=rollno;
                this.name=name;
                this.fee=fee;
        }
        void display()
        {
                System.out.println(rollno+" "+name+" "+fee);
        }
}

class TestThis2
{
        public static void main(String args[])
        {
                Student s1=new Student(111,"ankit",5000f);
                Student s2=new Student(112,"sumit",6000f);
                s1.display();
                s2.display();
```

```
        }
}
```

7a. What is array in Java? Explain the operation of Enhanced For (For Each Loop) on a matrix. [10]

Normally, array is a collection of similar type of elements that have contiguous memory location. **Java array** is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array. Array in java is index based, first element of the array is stored at 0 index.



There are two types of array - Single Dimensional Array & Multidimensional Array. Declaration Syntax: dataType[] arr; (or) dataType []arr; (or) dataType arr[]; //Single dimension array
dataType[][] arr; (or) dataType [][]arr; (or) dataType arr[][]; //Multi dimension array
For- Each Loop (Enhanced For Loop)
This loop is exclusively used only in arrays and is designed to cycle through the collection in strictly sequential fashion, from start to finish. The general syntax of this loop is
for(type itr-var : collection)
{ statement –block;
}
Here type specifies the type (it has to be same as the collection) and itr-var specifies the name of an iteration variable that will receive the elements from a collection, one at a time from beginning to end. The collection being cycled through is specified by collection.

```
class ForEachExample
{
    public static void main(String [] args)
    {
         int sum = 0;
         int num[] [] = new int [3][5];
        for(int i = 0; i <3 ; i++)
          for(int j=0; j<5; j++)
              num[i][j] = (i+1) * (j+1);
        for (int x[] : num)
          for(int y : x)
             sum +=y;
       System.out.println("Summation :" +sum);
    }
}
```

Output :

Summation : 90

Notice how x is declared, it is a reference to a one dimensional array of integers. This is necessary because each iteration of the for obtains the next array in num, beginning with the array specified ny num[0]. The inner for loop then cycles through each og these arrays displaying the value of each element.