BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
CS F213
LAB 4

AGENDA **TIME: 02 Hrs**

- Dynamic Binding
- Arrays
- Strings

4.1 Dynamic Binding

(OOPChef is a slight variation on the chef to avoid copyright infringement and is here to guide you on your journey to learning object-oriented programming)

OOPChef has already learned about method overriding and overloading. He is already building complex classes with many overridden methods. He has built the following class structure and is now very confused.

Program 4.1.a **:** *Object-Oriented Bakery*

```
public class bread{
   static void leaveUnattended(){
      System.out.println("mould has formed");
   }
   final void cook(){
      System.out.println("cooking in secret oven on 180 degrees celsius");
   }
   private void knead(){
      System.out.println("adding secret dough and water");
   }
   public void eat(){
      System.out.println("eating the bread");
   }
}

public class cake extends bread{
   public void eat(){
      System.out.println("eating the cake");
   }

   public void decorate() {
      System.out.println("decorating the cake");
```

```
    }
}

public class pastry extends cake {
    public void eat(){
        System.out.println("eating the pastry");
    }

    public void eatFrosting() {
        System.out.println("eating frosting on pastry");
    }
}

public class driver{
    public static void main(String[] args){
        bread b1 = new bread();
        bread b2 = new cake();
        bread b3 = new pastry();
        cake c1 = new cake();
        cake c2 = new pastry();
        pastry p1 = new pastry();

        //various functions of b1
        System.out.println("b1");
        b1.leaveUnattended();
        b1.cook();
        //b1.knead();
        b1.eat();
        //various functions of b2
        System.out.println("b2");
        b2.leaveUnattended();
        b2.cook();
        b2.eat();
        //b2.decorate();
        //various functions of b3
        System.out.println("b3");
        b3.leaveUnattended();
        b3.cook();
        b3.eat();
        //b3.decorate();
        //b3.eatFrosting();
        //various functions of c1
        System.out.println("c1");
        c1.leaveUnattended();
        c1.cook();
        c1.eat();
        c1.decorate();
        //various functions of c2
```

```
        System.out.println("c2");
        c2.leaveUnattended();
        c2.cook();
        c2.eat();
        c2.decorate();
        //c2.eatFrosting();
        //various functions of p1
        System.out.println("p1");
        p1.leaveUnattended();
        p1.cook();
        p1.eat();
        p1.decorate();
        p1.eatFrosting();
    }
}

//questions to think about
//why haven't we overridden the functions knead(), cook() and leaveUnattended()
//what does the final method cook tell you about the bakery
//what does the static keyword tell you about the nature of bread
//can we call knead() on any object other than b1?
//what do you learn from the various calls of eat()
//how do you find out which implementation of eat() is called
```

4.1.1 Reference Type and Object Type

OOPChef ran the code and was utterly confused by the various outputs. Before trying to understand what is going on, you must have a little understanding of two very important, but often overlooked concepts: Reference Type, and Object Type. Let's look at some examples to understand the difference:

```
Bread b1 = new Bread();
// Reference Type: Bread
// Object Type: Bread

Bread b2 = new Cake();
// Reference Type: Bread
// Object Type: Cake

// In general: ReferenceType <reference_name> = new ObjectType()
```

Now, you must be wondering, why is it even possible to have different object and reference types in the first place. This is because of the inheritance support provided by Java. An object is not only an instance of a class, but also an instance of it’s ancestor classes.

```
Cake c1 = new Pastry();
```

On executing the above statement, the memory looks something like this:

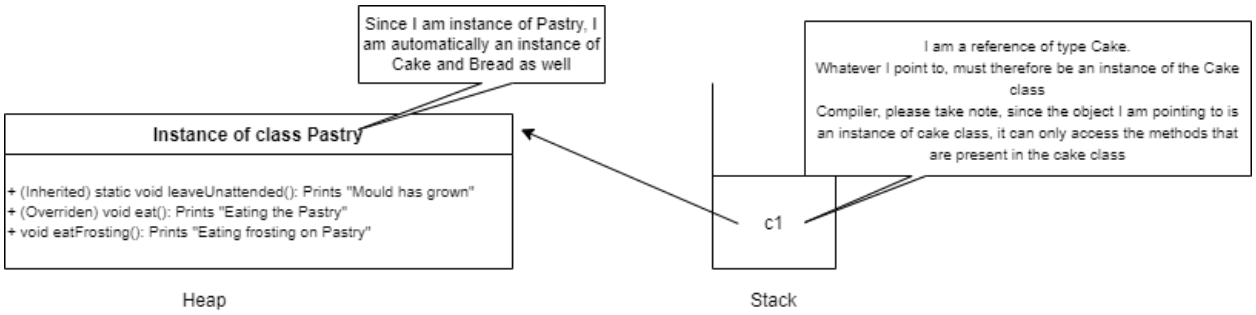

Image 4.1.a

A reference (pointer) is created in the stack, with the type ReferenceType. At the same time, a new object of type ObjectType is created in the heap, which actually contains all the information about the object.

4.1.2 Compiler's point of view

Now, since the object stored in the heap could be interpreted as an instance of its class, or of any of its ancestor classes, the reference type actually helps the compiler in determining how to interpret the object. Hence, the reference type is used to determine what all methods and members can be accessed. However, during runtime, the object in the heap refers to the information about the class it is an object of, and hence that determines which method would be executed.

4.1.3 Understanding the output

Now with this clear, let us try to make sense of the output of the above program step by step.

- The outputs from c1, b1, and p1 should be pretty straightforward, because the reference type is the same as the object type.
- The outputs for the leaveUnattended, and the cook method are the same everywhere because they have not been overridden by the child classes.
- Which eat method would be called actually depends on the object type, and not on the reference type, because of dynamic method dispatch. Hence, b2 and c1 call the eat method of the cake class, while b3 and c2 call the eat method of the pastry class.

4.1.4 Understanding the Errors

Uncomment each of the commented lines in the program 4.1.a and observe the various errors that you get.

- b1.knead() results in an error because knead is a private method of the bread class.
- b2.decorate() , b3.decorate() and b3.eatFrosting() result in an error because the reference type is bread, and bread doesn't have the decorate() method nor the eatFrosting() method.
- c2.decoreate() doesn't result in an error because the reference of c2 is cake, and the cake class has a decorate method.
- c2.eatFrosting() gives an error as the cake class doesn't have the eatFrosting method.

4.1.5 Exercises

1. Build 5 classes A, B, C, D and E, each with one public method speak() which prints the name of the class along with a small message. B and C extend A, D extends B, and E extends C
2. Do the following, or determine that it is impossible to do:
    - Run the implementation of speak() in class D with a reference of type A
    - Run the implementation of speak() in class A with a reference of type D
    - Run the implementation of of speak() in class C with a reference of type A
    - Run the implementation of` of speak() in class C with a reference of type E

4.2 Arrays

4.2.1 Introduction to Arrays

We have seen how to create individual variables either of primitive or non-primitive types and got habituated with their working. Now, there could be a scenario where we have to group the data about the entities and work with them. A more specific case could be we have to group similar types of data.

Like, we want to gather the details of Students who got enrolled in the OOP course. How do we do that?

Right! We start with creating a Student Class, Okay! Next?

Wait, one question, Why do we have to group them?
Think and state your reasons!!

To have ease in management and accessing the data, we usually group similar entities.
An array is one of the available data structures which will group similar items. In the Java programming language, arrays are considered as objects, which is not the case with C and C++ paradigms.

Few points about arrays in java:

- As arrays are viewed as objects in java, we can find the size of the array using a member-variable "length".
- The size of an array needs to be specified beforehand and can't be changed once it is fixed.
- A java array variable can be created by appending "[ ]" to the variable name.

4.2.2 Array Declaration and Initializations.

```
//Declaration
type variable_name[]
type []variable_name
```

- "**type**" specifies either primitive data types like (int, float, char, byte) or non-primitive types like (String, Student, Employee).
- ''**[ ]**", this informs the compiler that an array reference is going to be created. It also gives the details about the "no of dimensions" we are interested in.
- "**variable_name**", becomes a reference to the array where we are going to store the data of type "type".

```
int []intArr;  //line1
byte byteArr[];
Student [] studArray;
```

All the above declarations are one-dimensional in nature.

When an array is declared, only a reference variable is created, but to assign actual memory to the array, we need to use the "new" keyword to instantiate the array.

```
intArr = new int[5]; //line2
studArray = new Student[150];
```

Here an array of size '5' is created inside the memory and is being referenced by the 'intArr'variable. We can always combine both line1 and line2 in one step.

```
int []intArr = new int[5];
```

Note:

The elements in the array allocated by new will automatically be initialized to **0** (for numeric types), or **null** (for reference types).

Program 4.2.a

```
public class Main {
   public static void main(String[] args) {
      int[] intArr = new int[5];
      for(int i=0; i<5;i++){
         System.out.println(intArr[i]);
      }
   } }
```

Here are a few ways of initializing the array with elements.
Program 4.2.b

```
public class Main {
   public static void main(String[] args) {
      int[] a1 = new int[]{1,2,3,4,5};
      byte[] a2 = {5,2,4};
      System.out.println("Int Array1");
      for(int i=0; i<5;i++){
         System.out.println(a1[i]);
      }
       System.out.println("Byte Array2");
      for(int j=0; j< a2.length;j++){
         System.out.println(a2[j]);
      }
      int n = 4; //line1
      float[] a3 = new float[n];
      a3[1]= 2;
      a3[2] = 3;
      a3[3] = 4;
     // a3[4] = 5; //line2
       System.out.println("Float Array3");
       for(int k=0; k<n;k++){
         System.out.println(a3[k]);
      }
   }
}
```

```
/*
You can try this! Change the datatype of 'n' in line1 from 'int' to 'float', Observe what's happening!
After that! Try to comment out the line2 statement, run it, and state your reasons.
*/
```

Program 4.2.c

```
class Student
{
   public int id;
   public String name;
   Student(int id, String name)
   {
      this.id = id;
      this.name = name;
   }
}
public class Main
{
   public static void main (String[] args)
   {
      Student[] arr;
      arr = new Student[4];
      System.out.println("Before Initialization");
      for(int i=0;i<arr.length;i++){
         System.out.println(arr[i]);
      }
      arr[0] = new Student(234,"surya");
      arr[1] = new Student(345,"ravi");
      arr[2] = new Student(456,"aman");
      arr[3] = new Student(567,"amit");
      System.out.println("Students Details!");

      for(Student stud: arr){
        System.out.println(stud.id +" "+ stud.name);
      }
   }
}
```

4.2.3 Types of Arrays

We can categorize the arrays mainly on the basis of:

- No of Dimensions
    - One Dimensional
    - Multi-Dimensional
- Size of the array
    - Fixed sized (Normal arrays)
    - Dynamic sized (ArrayList etc.,)

One-dimensional arrays are the ones that we have already worked on in the above sections. As we discussed '[ ]' - it also refers to the no. of dimensions. In this section, we will look into the 2D arrays in particular in multi-dimensional arrays.

4.2.3.1 Two-Dimensional Arrays

We can visualize data in rows by columns in 2D arrays. It can be declared and instantiated in the following way:

```
int[][] intArray = new int[2][3];
int[] a = {1,2,3};
int[] b = {4,5,6};
```

In a 1D array, all the elements are stored in contiguous locations. But, when coming to 2D arrays, the elements are the array objects. We can see a 2D array as "an array of arrays".

The above statement can be easily comprehended if we read it,

```
int[]  []   intArray;
/*-1--  -2-  ---3---
 from right to left!
In the above statement, we are saying to the compiler,
1) Hello! I want to create a 'reference variable' -> "intArray" (--1--)
2) of type 'array' -> "[]" (-2-)
3) which refers to the elements of type
   'array objects of int type' -> "int[ ]" (--3--)
*/
```

Program 4.2.d

Observe the below program, which explains the 2D array working,

```
public class Main {
   public static void main(String[] args) {

      int[][] intArray = new int[2][3];
```

```
        int[] a = {1,2,3};
        int[] b = {4,5,6};

        intArray[0] = a;
        intArray[1] = b;

        for(int i =0;i<intArray[0].length;i++){
         System.out.print(intArray[0][i]+" ");
        }
        System.out.print("\n");
         for(int i =0;i<intArray[1].length;i++){
         System.out.print(intArray[1][i]+" ");
        }

    }
}
```

In order to gather the " array objects 'a' and 'b' " together (which might be different locations in the memory), we shall be having contiguous int[ ] array references

```
intArray[0], intArray[1]
```

ready in "intArray" 2D-array to refer to those array objects.

Program 4.2.e

The below program illustrates the passing of array as a parameter to a method

```
public class Main {
    public static void printArray(int[][] arr){
        for(int[] row: arr){
          for(int i=0; i< row.length;i++){
             System.out.print(row[i]+" ");
          }
           System.out.print("\n");
        }
    }
    public static void main(String[] args) {
        int[][] arr1 = {{1,2,3},{4,5,6}};
        System.out.println("Array1 Elements");
        printArray(arr1);

        int[][] arr2 = arr1; //line1
        int m=2,n=2;
        arr1 = new int[m][n];
```

```
        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                arr1[i][j]+=7;
            }
        }
        System.out.println("Array2 Elements");
         printArray(arr2);
        System.out.println("Array1 Elements after the change");
         printArray(arr1);
    }
}
```

Hey JavaBeans! Some **questions** for you!

1) What are your thoughts about the *//line1* statement?
2) Try to check if the (array object) parameter passing mechanism is 'pass by reference' or 'pass by value'?

4.2.4 Exercises

Exercise 4.2.f

Given a one-dimensional array **'arr'**, you should be cloning any sub-array of '**arr**' or whole array, upon the user requirement without using any inbuilt functions. You have to create a method with the name "**cloneArray**" inside the "**Exercise1**" class itself.

Note: A Subarray is a contiguous part of the given array 'arr'.

Use cases of cloneArray method:

1) Cloning full array:
    a) Arguments user should pass:
        i) Array **'arr'**
    b) Return a whole new array object with an exact clone.

2) Cloning sub-array of the given array:
    a) Arguments user should pass: -- Type 2.1
        i) Array **'arr'** and length **'len'** as arguments
        ii) You should consider cloning and returning the subarray of 'arr' which starts at index '**0**' and of length 'len'.
    b) Arguments user should pass: -- Type 2.2

i) Array **'arr'**, start index **'start'**, and length **'len'** as arguments.
ii) You should clone a new array which is the subarray of 'arr' which is starting at index 'start' and of length 'len'.

Consider having the same method name "**cloneArray**' for implementing the above functionalities.

Quick questions:
1) How do you implement the above three methods with the same name?
2) What is the nature of the method, is it a normal method or static type? State your reasons.

Use this Boilerplate code for the exercise!

```
public class Exercise1 {
  /*Implement all the required functionalities here*/


   /*Prefer to have a method for printing array elements with name
   "printArray" */

   public static void main(String[] args) {
      /*
      declare and initialize an array 'arr'
      Try to clone 'arr' in the above-mentioned ways by calling
      the above methods.
      */
   }
}

/*

Example:
arr is {1,2,3,4,5,6,7,8,9},
cloning a sub-array with length '4' gives out a new array with elements {1,2,3,4}, if you print it.
cloning a sub-array with start index '3' and length '5' gives out {4,5,6,7,8} new array.

*/
```

Exercise 4.2.g

Given a 2D array '**arr**', return a new array '**newArray**', where the **index** '*i*' element in the new array indicates the **sum of the elements in column** '*i*' of the given 2D array 'arr'.

Implement a method inside the provided '**Exercise2**' class with any name of your likes.

```
public class Exercise2 {
   /*
   Implement a method that takes 2D array 'arr' as input
   and outputs the' newArray' 1D array.
   */
   public static void main(String[] args) {
     //declare and initialize 2D array 'arr' here for
       simplicity purposes!


     //print the newArray contents here!
   }

}

/*
Example:
Input array 'arr' contents :[ [1 2 3],
                              [4 5 6],
                              [7 8 9] ]

'newArray' contents : [ 12 15 18 ]

newArray[i] = sum of elements of column 'i' of 'arr' 2D array.

*/
```

4.3 Strings

In Java, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.

We use **double quotes** to represent a string in Java. For example,

```
// create a string
String type = "Java programming";
```

Program 4.3.a : *Create a String in Java*

```
class Main {
  public static void main(String[] args) {

    // create strings
    String first = "Java";
    String second = "Python";
    String third = "JavaScript";

    // print strings
    System.out.println(first);   // print Java
    System.out.println(second);  // print Python
    System.out.println(third);   // print JavaScript
  }
}
```

In the above example, we have created three strings named first, second, and third. Here, we are directly creating strings like primitive types. However, there is another way of creating Java strings (using the new keyword).

4.3.1 Creating strings using the new keyword

So far we have created strings like primitive types in Java. Since strings in Java are objects, we can create strings using the new keyword as well. For example,

Program 4.3.b

```
class Main {
  public static void main(String[] args) {


    // create a string using new
    String name = new String("Java String");


    System.out.println(name);  // print Java String
  }
}
```

4.3.2 Create String using literals vs new keyword

Now that we know how strings are created using string literals and the new keyword, let's see what is the major difference between them.

In Java, the JVM maintains a **string pool** to store all of its strings inside the memory. The string pool helps in reusing the strings.

1. While creating strings using string literals,

```
String example = "Java";
```

Here, we are directly providing the value of the string (Java). Hence, the compiler first checks the string pool to see if the string already exists.

- **If the string already exists**, the new string is not created. Instead, the new reference example points to the already existing string (Java).
- **If the string doesn't exist**, the new string (Java) is created.

2. While creating strings using the new keyword,

```
String example = new String("Java");
```

Here, the value of the string is not directly provided. Hence, a new "Java" string is created even though "Java" is already present inside the memory pool.

The string can also be declared using a new operator i.e. dynamically allocated. In case String is dynamically allocated they are assigned a new memory location in heap. This string will not be added to the String constant pool.

Interfaces and Classes in Strings in Java

- **CharBuffer**: This class implements the CharSequence interface. This class is used to allow character buffers to be used in place of CharSequences. An example of such usage is the regular-expression package java.util.regex.

- **String**: String is a sequence of characters. In java, objects of String are immutable which means constant and cannot be changed once created.

Some important String Methods :-

Program 4.3.c

```
public class Main {
  public static void main(String[] args) {


    String name = new String("Java String");
    System.out.println(name);
    System.out.println(name.length());//prints length of String name
    System.out.println(name.charAt(5));// prints the char at index 5
    char ch[]=name.toCharArray();//convert name into a char array
    String name1 = new String("New String");
    Boolean result = name1.equals(name);
    System.out.println("result = " + result);
    int p=name1.compareTo(name);
    System.out.println("p="+p);

   //Substring method
   String s1 = new String("Hello World");//returns the substring from the ith index
   String s2=s1.substring(4);//
   System.out.println(s2);
  }
}
```

4.3.3 StringBuffer:

StringBuffer is a peer class of **String** that provides much of the functionality of strings. The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.

Program 4.3.d

```
class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
}
}
```

String and StringBuffer classes are declared final, so there cannot be subclasses of these classes.

4.3.4 StringBuilder:

The **StringBuilder** in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters.

Program 4.3.e:

```
class Example{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

Exercise 4.3.f:

```
//How will you print out the characters in ch[]?
//Can you change ch[] and convert it back into string?
//Now try these methods with String
• int compareToIgnoreCase( String anotherString)
  //Compares two strings lexicographically, ignoring case considerations.
• String toLowerCase()
  //Converts all characters to lowercase
• String toUpperCase()
  //Converts all characters to uppercase
• String trim()
   //Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the
middle.
• String replace (char oldChar, char newChar)
  //Returns new string by replacing all occurrences of oldCharwith newChar
// Now  try the methods of String with StringBuilder and StringBuffer
```