Java

Scanner

- **Initialize a scanner:** Scanner in = new Scanner(System.in);
- **For Integers:** in.nextInt();
- **For String:** in.next();
- **Close scanner:** in.close();

String

- **Initializing new string:** String StringName = new String(“”);
- **Length of String:** StringName.length();
- **Character at Position in String:** StringName.charAt();
- **Delete a char:** StringName.replace(“o”,)
- **Find the Index of character in String:** StringName.indexof(“T”);
- **String Slicing from startIndex:** StringName.substring(3);
- **String Slicing from startIndex to endIndex:** StringName.substring(3,5) // It does not include whatever is there at index 5;
- **Check if two strings are same:** StringName1.equals(StringName2);
- **Check if Character is uppercase:** Character.isUpperCase(char);
- **Check if Character is lowercase:** Character.isLowerCase(char);
- **Check if Character is digit:** Character.isDigit(char);
- **Convert Character to uppercase:** Character.toUpperCase(char);
- **Convert Character to lowercase:** Character.toLowerCase(char);
- **Convert Character to uppercase:** Character.toUpperCase(char);
- **Convert Character to Integer:** Character.getNumericValue(‘5’);
- **Convert String to Integer:** Integer.parseInt(“5”);
- **Convert String to uppercase:** StringName.toUpperCase();
- **Convert String to lowercase:** StringName.toLowerCase();
- **Convert Integer/Float to String:** Integer.toString(int), Float.toString(float);
- **Get the ASCII value of character:** int ASCII = str.charAt(i);
- **Split a string:** String[] parts = StringName.split(“-“, 0); if number is +tive then split n-1 times.
- **Sort a string:**
  - String StringName1 = new String("hello");
  - char[] chars = StringName1.toCharArray();
  - Arrays.sort(chars);
  - String StringName2 = new String(chars);
  - System.out.println(StringName2);
- **To test how many time each character occurs, use an array:**
  - int[] char_set = new int[128];
  - Here 128 are the ASCII characters which are only 8 bits and hence 128
- **Traversing through a string and concatenating String:**
  - for (int i = 0; i < StringName.length(); i++) {
  - reversed += StringName.charAt(i);
  - }
- **Using StringBuilder to make string:** StringBuilder sb = new StringBuilder(“Hel”);
  - **This is a more efficient to make a string**
  - **You can use this to make strings from characters**
  - **Convert StringBuilder to String:** String str = sb.toString();
  - **To add to the StringBuilder:** sb.append(a);

- **Delete character from StringBuilder:** sb.deleteCharAt(i);
- **Length of StringBuilder:** sb.length();
- **Character at Position in StringBuilder:** sb.charAt();
- **Find the index of character in StringBuilder:** sb.indexof("T");

Miscellaneous

- **Import this for everything to work:** import java.util.*
- **Maximum value between 2 numbers:** Math.max(value1, value2);
- **Minimum value between 2 numbers:** Math.min(value1, value2);
- **Absolute value of a number:** Math.abs(value1);
- **Celling and flooring of a floating number:** Math.ceil(3.14), Math.floor(3.14);
- **Printing decimal:** System.out.printf("The answer is: %.7f\n", FloatValue);
- **To avoid integer overflow, use:** long longName = 5;
- **Minimum Integer:** Integer.MIN_VALUE;
- **Maximum Integer:** Integer.MAX_VALUE;

Arrays

- **Creating a 1D array:** int ArrayName[] = new int[6];
- **Length of 1D array:** ArrayName.length;
- **Creating a 2D array:** int ArrayName[][] = new int[6][6];
- **Length of 2D array:** ArrayName[1].length;
- **Creating an array of list:** List<Integer>[] ArrayName = new List[2];
- **Copy an array:** Arrays.copyOf(ArrayName);
- **Create new array of n elements & copy from another:** Arrays.copyOf(ArrayName, n);
- **Copy subarray from array:** Arrays.copyOfrange(ArrayName, StartIndex, EndIndex);
- **Check if two array are same:** Arrays.deepEquals(Array1, Array2);
- **Fill array with certain value:** Arrays.fill(ArrayName, integerValue);
- **Sort Array:** Arrays.sort(ArrayName), Arrays.sort(ArrayName, Collections.reverseOrder());
- **Convert Array to String:** Arrays.toString(ArrayName);
- **Another way of traversing through an array:**
  - for (int i : ArrayName) {
  - sum += i;
  - }

List/ArrayList

- **Creating String ArrayList:** ArrayList<String> ListName = new ArrayList<String>();
- **Creating Int ArrayList:** ArrayList<Integer> ListName = new ArrayList<Integer>();
- **Creating Object ArrayList:** ArrayList<Object> ListName = new ArrayList<Object>();
- **Add to ArrayList:** ListName.add("addthis");
- **Get the size of ArrayList:** ListName.size();
- **Get the value at specific position:** ListName.get(i);
- **Set an element of the list:** ListName.set(index, element);
- **Get the maximum value from an ArrayList:** Collections.max(ListName);
- **Get the minimum value from an ArrayList:** Collections.min(ListName);
- **Sort the list:** Collections.sort(ListName); Collections.sort(ArrayName, Collections.reverseOrder());
- **Reverse the list:** Collections.reverse(ListName);
- **Search in the list:** Collections.binarySearch(ListName, 10);

- **Swap two elements in ArrayList:** Collections.swap(ListName, IndexFrom, IndexTo);
- **Convert to Array:**
    - Integer[] arrayName = new Integer[AL.size()];
    - AL.toArray(arrayName);
- **Check if it contains:** ListName.contains(element);
- **Check if list is empty:** ListName.isEmpty();

Linked List

- **Create Node Class:**
  - class Node {
  - Node next;
  - int data;
  -
  - Node(int data) {
  - this.data = data;
  - next = null;
  - }
  - }
- **To access the data of a Node:** Node.data;
- **To access the next element:** Node.next;
- **Create new node:** Node newNode = new Node(5);
- **Using ArrayList to save node reference:** ArrayList<Node> temp = new ArrayList<Node>();
  - **Adding reference of node to ArrayList:** temp.add(newNode);
- **Referencing the next few nodes:** Node.next.next;

Linked List data structure in java

- **Creating LinkedList:** LinkedList LLName = new LinkedList();
- **Creating String LinkedList:** LinkedList<String> LLName = new LinkedList<String>();
- **Creating Int LinkedList:** LinkedList<Integer> LLName = new LinkedList<Integer>();
- **Creating Object LinkedList:** LinkedList<Object> LLName = new LinkedList<Object>();
- **Add an element to end of the List:** LLName.add("Hello");
- **Add an element to List in a specific position:** LLName.add(2, "Hello") ;
- **Add an element to the start of the list:** LLName.addFirst("Hello");
- **Add an element to the last of the list:** LLName.addLast("Hello");
- **Remove the first occurrence of an object:** LLName.remove("Hello");
- **Remove at a certain index:** LLName.remove(i);
- **Remove an element to the start of the list:** LLName.removeFirst("Hello");
- **Remove an element to the last of the list:** LLName.removeLast("Hello");
- **Remove all elements:** LLName.clear();
- **Return the object at a specific index:** LLName.get(i);
- **Replace the element at a specified index:** LLName.set(2, "Hello");
- **Find if an element is there:** LLName.contains("Hello");
- **Find index of an element:** LLName.indexOf("Hello");
- **Size of LinkedList:** LLName.size();
- **Convert to an array:** Int[] ArrayName = LLName.toArray();
- **Iterate all Elements of LinkedList:**
  - Iterator<Integer> itr= LLName.iterator();
  - while(itr.hasNext()) {
  - System.out.println(itr.next());
  - }

Stack

- **Creating a new Stack:** `Stack<Integer> StackName = new Stack<Integer>();`
- **Pushing to Stack:** `StackName.push(6);`
- **Pushing to Stack:** `StackName.push("hello");`
- **Check the top element on the Stack but don't pop:** `StackName.peek();`
  - **This returns an object so use:** `Integer number = (Integer)StackName.peek();`
- **Popping to Stack:** `StackName.pop();`
  - **This returns an object so use:** `Integer number = (Integer)StackName.pop();`
- **See if the stack is empty:** `StackName.empty();`
- **Return the position where the object is in the Stack:** `int pos = StackName.search(4);`
- **Iterate all Elements of Stack:**

```
Iterator itr=StackName.iterator();
while(itr.hasNext()) {
    System.out.println(itr.next());
}
```

Queue

- **Creating a new Queue:** `Queue<Integer> QueueName = new LinkedList();`
- **Adding to the Queue:** `QueueName.add(1)`
- **Retrieving and removing the head of the Queue:** `QueueName.poll()`
  - **Returns null if queue is empty.**
  - **This returns an object so use:** `Integer number = (Integer)myStack.poll();`
- **Retrieving the head of the Queue:** `QueueName.peek()`
- **Checking if Queue is empty:** `QueueName.peek();/QueueName.poll();` **will return Null**
- **Iterate all Elements of Queue:**

```
Iterator itr=QueueName.iterator();
while(itr.hasNext()) {
    System.out.println(itr.next());
}
```

Priority Queue/Heap

- **Creating a heap:** `PriorityQueue<Integer> HeapName = new PriorityQueue<Integer>();`
  - **The above heap is for integers**
  - **The top priority for above queue is the minimum value**
- **Creating String heap:** `PriorityQueue<String> queue=new PriorityQueue<String>();`
- **Adding to the heap:** `HeapName.add(5);`
- **Removing from the heap:** `HeapName.remove(5);`
- **For top priority removal:** `HeapName.poll()`
- **For top priority looking:** `HeapName.peek()`
- **Clear all elements:** `HeapName.clear();`
- **Returns the size of all elements in queue:** `HeapName.size();`
- **Converts the queue into an array:** `Object myArray[] = HeapName.toArray();`
  - **To convert the above Object array to Integer array:** `Integer myArray[] = HeapName.toArray(new Integer[0]);`
- **Creating a heap with maximum value as top priority:** `PriorityQueue<Integer> HeapName = new PriorityQueue<Integer>(size, Collections.reverseOrder());`
  - **Here the size is the total capacity of the heap which has to be specified at the start**
- **Iterate all Elements of Priority Queue:**

```
Iterator itr= HeapName.iterator();
while(itr.hasNext()) {
```

```
    System.out.println(itr.next());
}
```

Trees

- **Create Node Class for Tree:**

```
class Node {
    Node left;
    Node right;
    int data;

    Node(int data) {
        this.data = data;
        left = null;
        right = null;
    }
}
```

- **Preorder, Inorder and Postorder Transversal:**

```
void preOrder(Node root) {
    System.out.print("Preorder: " + root.data + " ");
    if (root.left != null) preOrder(root.left);
    System.out.print("Inorder: " + root.data + " ");
    if (root.right != null) preOrder(root.right);
    System.out.print("Postorder: " + root.data + " ");
}
```

- **Levelorder Transversal:**

```
void levelOrder(Node root) {
    Queue<Node> levelOrder = new LinkedList();
    levelOrder.add(root);
    while (levelOrder.peek() != null) {
        Node n = levelOrder.poll();
        System.out.print(n.data + " ");
        if (n.left != null) levelOrder.add(n.left);
        if (n.right != null) levelOrder.add(n.right);
    }
}
```

Hashset: This is used to determine if repetition is there in array or string or anything

- **Creating a Character hashset:** `HashSet<Character> HashSetName = new HashSet<Character>();`
- **Creating a String hashset:** `HashSet<String> HashSetName = new HashSet<String>();`
- **Creating a Node hashset:** `HashSet<Node> HashSetName = new HashSet<Node>();`
- **Add to hashset:** `HashSetName.add(Object);`
- **Remove from hashset:** `HashSetName.remove(Object);`
- **Check if hashset contains something:** `HashSetName.contains(Object);`
- **Find the size of hashset:** `HashSetName.size();`
- **Clearing a hashset:** `HashSetName.clear();`
- **Check if hashset is empty:** `HashSetName.isEmpty();`
- **Iterate all Elements of Hashset:**

```
Iterator itr= HashSetName.iterator();
while(itr.hasNext()) {
    System.out.println(itr.next());
}
```

HashTable

- **Creating a hashtable:** `Hashtable HashtableName = new Hashtable();`
- **Creating a hashtable:** `Hashtable<String, Double> HashtableName = new Hashtable<String, Double>();`
- **Put a key and value:** `HashtableName.put(key, value);`
- **Get the value:** `HashtableName.get(key);`
- **Remove a key and value, value is returned in this:** `HashtableName.remove(key);`
- **Reset and empty the hashtable:** `HashtableName.clear();`
- **Check if some key is contained:** `HashtableName.containsKey(Object key);`
- **Check if some value is contained:** `HashtableName.containsValue(Object value);`
- **Check if hashtable is empty:** `HashtableName.isEmpty();`
- **Return an enumeration of values:** `HashtableName.elements();`
- **Return an enumeration of keys:** `HashtableName.keys();`
- **Iterate through all keys:**
  - `Enumeration names = balance.keys();`
  - `while(names.hasMoreElements()) {`
  - `    str = (String) names.nextElement();`
  - `    System.out.println(str + ": " + balance.get(str));`
  - `}`

HashMap

- **Creating a HashMap:** `HashMap HMName = new HashMap();`
- **Creating a HashMap:** `HashMap<String, Double> HMName = new HashMap<String, Double>();`
- **Creating a HashMap with multiple values for every key:**
  - `HashMap<Integer,ArrayList<String>> HMN = new HashMap<Integer,ArrayList<String>>();`
  - `ArrayList<String> ListName = new ArrayList<String>();`
  - `ListName.add("abc");`
  - `ListName.add("xyz");`
  - `HMN.put(100,list);`
- **Creating a synchronized HashMap:** `Map HMSName = Collections.synchronizedMap(HMName);`
- **Put a key and value:** `HMName.put("Zara", new Double(3.33));`
- **Returns the value to which specific key is mapped:** `HMName.get("Zara")`
- **Remove the mappings for this key:** `HMName.remove("Zara");`
- **Replace a value for a key that already exists:** `HMName.put("Zara", 5.86);`
  - `HMName.put("Zara", HMName.get("Zara") + 1);`
- **Check the HashMap has no key-value mappings:** `HMName.isEmpty();`
- **Number of key value mappings:** `HMName.size();`
- **Tell if a certain key exists:** `HMName.containsKey("Zara");`
- **Tell if a certain value exists:** `HMName.containsValue(3.33);`
- **Returns a value of all the sets:** `Set set = HMName.keySet();`
- **Collection view of all the mappings:** `Set set = HMName.entrySet();`
- **Iterating all the elements:**
  - `for(Map.Entry m:HMName.entrySet()){`
  - `   System.out.println(m.getKey()+" "+m.getValue());`
  - `}`

Difference between Hashtable and HashMap

Hashmap	Hashtable
Not synchronized	Synchronized
Has null values	Does not have null values
O(1) for put, remove etc.	O(1) for put, remove etc.
Efficient	Obsolete

Graphs

- **Depth First Search:**
  - ```
    private static void dfTrav(Vertex v, Vertex parent) {
        System.out.println(v.id); // visit v
        v.done = true;
        v.parent = parent;
        Edge e = v.edges;
        while (e != null) {
            Vertex w = e.end;
            if (!w.done)
                dfTrav(w, v);
            e = e.next;
        }
    }
    ```
- **Breadth First Search:**
  - ```
    void search(Node root) {
        Queue<Node> queue = new LinkedList();
        queue.add(root);
        while (queue.peek() != null) {
            Node n = queue.poll();
            System.out.print(n.data + " ");
            Foreach (n in Node children) {
            if (n.left != null) queue.add(n);
        }
    }
    ```

Enums

- **Enums are like classes but they are used to declare constants.**
  - ```
    public enum fruit {
      apple,
      banana,
      grapes,
      carrot,
      tomato,
      pear; // 6 instances of fruits
    }
    ```
-
- **Each constant are objects and can have their own set of variables.**
- **Weird thing about enum is that all of the instances are defined inside the fruit enum.**
- **This means any property, variable or method of fruit enum will belong to each one of the instances.**
- **You don't use the new keyword with enums because you can only use the objects that are there in the enums. So, when you create a new object of enum you have to assign it as one of the objects in enums.**
    ```

```
public enum fruit {
 apple("Green"),
 banana("Yellow"),
 grapes("Green"),
 carrot("Orange"),
 tomato("Red"),
 pear("Brown"); // 6 instances of fruits

 public String color;

 fruit(String c) { // constructor
      color = c;
 }
}

public class mainClass {
 public static void main(String[] args) {
      fruit apple = fruit.apple;
      System.out.println("the " + apple + " and " + apple.color); // This will
print the apple and green.
 }
}
```

- **How to print every fruit in enum.**
- **values basically gets an array of every fruit value**

```
public class mainClass {
 public static void main(String[] args) {
      for (fruit f: fruit.values()) {
           System.out.println("the " + f + " and " + f.color);
      }
 }
}
```

- **Now if I want to print a range of fruits from let's say banana to tomato then do following:**

```
import java.util.Enumset;
public class mainClass {
 public static void main(String[] args) {
      for (fruit f: Enumset.range(fruit.banana, fruit.tomato)) {
           System.out.println("the " + f + " and " + f.color);
      }
 }
}
```

Multi-threading

- **Thread priority is an integer and is used to decide which thread should run. The priority is used to decide which thread will run and is called context switching. Context switching occurs in the following cases:**
  - **A thread relinquishing control by yielding or sleeping.**
  - **A thread can be preempted by a higher priority thread if the lower probability thread is not giving up the processor.**
- **Using thread class for getting information about threads using thread reference:**
  - **Obtain the thread's name:** `ThreadReference.getName();`
  - **Set the thread name:** `ThreadReference.setName();`
  - **Obtain the thread's priority:** `ThreadReference.getPriority();`
  - **Determine if a thread is still running:** `ThreadReference.isAlive();`
  - **Wait for a thread to terminate:** `ThreadReference.join();`
  - **Entry point for the thread:** `ThreadReference.run();`
  - **Suspend a thread for a period of time:** `ThreadReference.sleep();`
  - **Start a thread by calling its run method:** `ThreadReference.start();`
- **The above methods are involked on a particular thread. The following methods in the thread class are static and run on the current thread:**
  - **Make the thread sleep:** `Thread.sleep(1000);`
  - **Get the reference of current thread:** `Thread.currentThread();`
- **Main thread is the thread that is created right when the java program starts.**
- **The reference of this thread can be obtained using the currentThread() method.**

```
// Controlling the main Thread.
class CurrentThreadDemo {
    public static void main(String args[]) {
        Thread t = Thread.currentThread();

        System.out.println("Current thread: " + t);

        // change the name of the thread
        t.setName("My Thread");
        System.out.println("After name change: " + t);

        try {
            for(int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
}
```

  - **Object is created using:** `Thread t = Thread.currentThread();`
  - **Name is changed using:** `t.setName();`
- **Creating a thread using runnable is done by implementing the runnable interface**

```
// Create a second thread.
class NewThread implements Runnable {
    Thread t;
    NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
    }

    // This is the entry point for the second thread.
```

```
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ThreadDemo {
    public static void main(String args[]) {
        new NewThread(); // create a new thread
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

- **When creating the new thread, this is used so that it can be indicated that the run() method is called on this thread.**
- **T.start() starts the thread where it starts implementing whatever is inside the run() method.**

- **Creating a thread using extending the Thread class**

```
// Create a second thread by extending Thread
class NewThread extends Thread {

    NewThread() {
        // Create a new, second thread
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
    }

    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // create a new thread
```

```
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

- **In the above code, super() is calling the Thread constructor which is used for thread name.**
- `Public Thread(String threadName);`

- **The isAlive() method returns true if the thread upon which it is called is still running. It returns false otherwise.**
  - `threadReference.isAlive();`
- **The join() method waits on which it is called terminates and then it throws an exception. You can also specify the time that you want to wait for the specified thread to terminate.**

```
try{
    threadReference.join();
} catch (InterruptedException e) {
    System.out.println("Main thread Interrupted");
}
```

- **Priority is used to determine which thread will be given preference using context switching**
  - **Set priority of a thread:** `threadReference.setPriority(5);`
  - **Get priority of a thread:** `threadReference.getPriority(5);`
  - **Maximum priority possible:** `Thread.MAX_PRIORITY;`
  - **Minimum priority possible:** `Thread.MIN_PRIORITY;`
  - **Normal priority which is not very high or low:** `Thread.NORM_PRIORITY;`
- **Volatile: This saves the data in main memory and not in some sort of cache. This helps if two threads are accessing the same variable. If both threads are incrementing the variable then every thread will make sure if it has already been incremented by the other thread.**
  - `volatile int counter = 0;`
- **Synchronization**
  - **Synchronization is implemented using a concept of the monitor(semaphore). A monitor is an object that is used as a mutually exclusive lock or mutex. Only one thread can own a monitor at a given time. When a thread acquires the lock it is said to have entered the monitor. The other threads will be waiting for the monitor.**
  - **There are two ways to use synchronization:**
    - **Synchronized Methods: If a thread is inside a synchronized method, all other threads that are calling the method have to wait.**

```
synchrosized void call() {
    System.out.println("Hello");
}
```

    - **Synchronized Statements: Lets say you are using a class that was not made by you and hence none of the methods are synchronized. In this case you use a synchronized block.**

```
synchronized(object) {
    // Statements to be synchronized
}
```

- **Wait(), Notify(), NotifyAll()**

- wait(): **tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).**
- notify(): **wakes up a thread that called wait( ) on the same object.**
- notifyAll(): **wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.**

```
class Q {
    int n;
    boolean valueSet = false;

    synchronized int get() {
        while(!valueSet)
        try {
            wait();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
        }

        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
    }

    synchronized void put(int n) {
        while(valueSet)
        try {
            wait();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
    }
}
```

- **DeadLock**
  - **A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects.**
  - **For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.**

Java Concepts

- **The four fundamental OOP concepts are Encapsulation, Inheritance, Polymorphism, and abstraction.**
- **toString()**
  - **When any object is asked to print, it looks for the toString() method and you dont even need to write the toString() method. It knows it on its own that it has to implement the toString() method.**
    - `System.out.printf(objectName);`
  - **Due to this, it is also important that when you make a new object, you create a toString() method so that when the object is asked to print then it goes to the toString() method that you created.**
    - `System.out.println(objectCreatedName) // This will look for the toString() method that you created`
  - **This is why when you print Integer, it print easily because even though its an object, it must have a toString() method that is implemented.**
- **This:**
  - **It's used when a local variable and instance variable has the same name**
  - **If they have the same name then local variable can hide instance variable**
  - **It is mostly used in constructors but can also be used in other cases where the name of local variable and constructor variable is same**
  - **This is used to denote the instance variable and not the local variable**
  - **The following example shows a constructor using this**

```
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

  - **It can also be used to call another overloaded constructor**

```
class JBT {
    JBT() {
        this("JBT");//Here it is checking for a constructor that accepts string
        System.out.println("Inside Constructor without parameter");
    }

    JBT(String str) {
        System.out.println("Inside Constructor with String as " + str);
    }

    public static void main(String[] args) {
        JBT obj = new JBT();
    }
}
```

  - **It can also be used to call a method from same class**

```
class JBT {

    public static void main(String[] args) {
        JBT obj = new JBT();
        obj.methodTwo();
    }
    void methodOne(){
        System.out.println("Inside Method ONE");
    }
    void methodTwo(){
        System.out.println("Inside Method TWO");
        this.methodOne();// same as calling methodOne()
    }
```

    - }
  - **If the parameters of method has same name as instance variables then this can be used to refer the instance variable**
    - ```
      public class time {
          private int hour = 1;
          prinvate int minute = 2;

          public void setTime(int hour, int minute, int second) {
              this.hour = 4; // the instance variable hour is changed to 4
              this.minute = 3;
          }
      }
      ```
  - **Basically this carries the reference to the current object of the class. Hence this.variable is same as currentObjectofClass.variable and this.method is same as currentObjectofClass.method.**
- **Garbage Collection**
  - **When no references to an object exists then garbage collection works**
  - **It occurs at irregular intervals and cannot be called**
- **What is finalize method**
  - **When you add a finalizer to a class, it is implemented right before the garbage collection reclaims it**
  - **It is important to note that it is only before garbage collection and not before when the object goes out of scope.**
  - ```
    protected void finalize() {
        // finalization code here
    }
    ```
- **Overloading Methods**
  - **This occurs when two methods in the same class have the same name but parameter declarations are different**
  - **Java uses different parameters to determine which method is being called**
  - **Return types does not play any role in overloading**
  - ```
    void test() {
        System.out.print(“Hello”);
    }
    void test(int a) {
        System.out.print(“Hello”);
    }
    ```
  - **If in some case the method requires int and there are no overloaded methods with int input parameter but one overloaded method with double input parameter then java will convert the int to double on its own**
- **Overloading Constructors**
  - **This is very common where more than one type of constructor is there**
  - ```
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    Box(double len) {
        width = height = depth = len;
    }
    ```
  - **Java will decide which constructor to choose depending on the parameters input when creating the instance of class**
- **Static**
  - **This is basically used when you don’t want to create an object for it and want to use it anyhow. For example when Math.max() is used, we don’t need to create an object of Math, we just call Math.max(), this is because max would be a static method and hence can be called without creating an object for Math.**

- Box mybox = new Box(); // This is not needed to be declared for static methods in Box class.
- Box.volume(); // This can be directly called without creating the above object if volume is a static method.

- **An example of this is the main() which always has to be static because it is always called before any object exists.**
- **Instance variables declared as static are global variables. When objects of its class are declared, no copy of static variable is made. Instead, all instances of the class share the same static variable.**
- classname.method(); // To call a static method from a different class
- classname.variable(); // To call a static variable from a different class
- **Static variables are shared by every object of that class as shown below:**

```
public class girls {
    private String first; // This would only refer to the specific object
    private String last;
    private static int count = 0; // This is common among all objects of class

    public girls(String first, String last) {
          this.first = first;
          this.last = last;
          count++;
          System.out.print(count);
    }
}

public class mainClass {
    public static void main(String[] args) {
          girls one = new girls("Megan, "Fox");
          girls two = new girls("Natalie, "Portman");
          girls one = new girls("Taylor, "Swift");
    }
}
```

- This will print: 1,2,3

- **Methods declared as static have following restrictions:**
  - **They can only call other static methods.**
  - **They must only access static data.**
  - **They cannot refer to this or super in anyway.**
- **As soon as a class is run, all the static statements are run first. If this class contains main then main is called only after all static statements are run.**

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
    static int a = 3;
    static int b;
    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }
    public static void main(String args[]) {
        meth(42);
    }
}
```

- **The above class prints the following**

- Static block initialized.
- x = 42
- a = 3
- b = 12
- **In the above class all the static statements are initialized first. That is why the static block prints first and then the main method called meth.**
- class StaticDemo {
-     static int a = 42;
-     static int b = 99;
-     static void callme() {
-         System.out.println("a = " + a);
-     }
- }
- class StaticByName {
-     public static void main(String args[]) {
-         StaticDemo.callme();
-         System.out.println("b = " + StaticDemo.b);
-     }
- }
- **Here is the output of this program:**
- a = 42
- b = 99
- **In the above program, no object for StaticDemo was created and the callme() method was called directly because it was static. 'B' was also called in the second function using the class name because the b created in the first class was static and hence was global.**

- **Final**
  - **This is basically like a constant.**
  - **A final variable can be assigned a value only once.**
  - private final int NUMBER = 2;
- **Private**
  - **In case of subclass and superclass, a subclass does not inherit the methods and variables that are private.**
  - **If the object is declared and a method is private then it cannot be accessed.**
  - **Final thing, if a method is private, it cannot be accessed from anywhere outside the class regardless of if it is the superclass or the class is initialized as an object in another class or it is a static class. It can never be used outside that specific class.**
- **Protected**
  - **This is like private but is also available to the subclasses.**
- **Package**
  - **Like Protected but is available to everyone declared inside the package.**
  - **This is the default in java**
- **Public**
  - **Available to everyone.**
- **Polymorphism**
  - **Polymorphism is a feature in which one feature can be performed in different ways.**
  - **Any Java object that can pass more than IS-A test is considered to be polymorphic.**
  - public interface Vegetarian{}
  - public class Animal{}
  - public class Deer extends Animal implements Vegetarian{}
  - **Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples -**
  - **A Deer IS-A Animal**
  - **A Deer IS-A Vegetarian**
  - **A Deer IS-A Deer**
  - **A Deer IS-A Object**
  - **The Real Polymorphism**

        - class Animal;
        - Dog extends Animal;
        - Cat extends Animal;
        - Deer extends Animal;
        - Dog rover = new Dog();
        - Animal spot = new Dog();
        - **If it's convenient for me to think of it as a dog object then I will do so.**
        - **If it's more convenient for me to think of it as an animal object then I will do so.**
        - **Now spot can anytime be changed to car or deer.**
        - **Animal[] kingdom = new Animal[];**
        - kingdom[0] = new Dog();
        - kingdom[1] = new Cat();
        - for (Animal a:kingdom) {
        -     a.sleep(); // if a is a dog then it will sleep like a dog.
        -     a.eat(); // if a is a cat then it will eat like a cat
        - }
- **Inheritance**
    - **There are two types of relationships in java IS-A and HAS-A.**
        - **When a class extends from another class then it has IS-A relationship**
            - class Calculator {
            - public int add(int a, int b) {return a + b;}
            - }
            - class CalculatorAdv extends Calculator {
            - public int sub(int a, int b) {return a - b;}
            - }
            - **Here CalculatorAdv is also a calculator so there is a IS-A relationship.**
        - **If in a class you are creating object of another class then it has a HAS-A relationship.**
            - class Calculator {
            - public int add(int a, int b) {return a + b;}
            - }
            - class mainClass {
            - public static void main(String[] args) {
            - Calculator calc = new Calculator();
            - }
            - }
            - **Here mainClass has calculator object and so there is a HAS-A relationship.**
    - **Class that is inherited from is called the superclass and the class that does the inheriting is called the subclass.**
    - **Subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.**
    - **To inherit a class, you simply incorporate the definition of one class into another by using extends.**
    - **When a sub class is called then first the constructor of superclass is called and then the constructor of subclass is called.**
    - // Create a superclass.
    - class A {
    -     int i, j;
    -     void showij() {
    -         System.out.println("i and j: " + i + " " + j);
    -     }
    - }
    - // Create a subclass by extending class A.
    - class B extends A {
    -     int k;

```
    void showk() {
        System.out.println("k: " + k);
    }
}
class SimpleInheritance {
    public static void main(String args[]) {
        A superOb = new A();
        B subOb = new B();

        // The superclass may be used by itself.
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();

        /* The subclass has access to all public members of
        its superclass. */
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.showij();
        subOb.showk();
    }
}
```

The output is:

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
```

- **As it can be seen, all the variables and methods of A can be used and edited in B.**
- **Class cannot be a superclass of itself**
- **More than one superclass cannot be used. You can use hierarchy in this where one superclass can be a subclass of another class.**
- **A subclass cannot access a variables or methods in superclass that are declared private.**
- **How to use super:**
  - **Super can be used to call the constructor of the superclass from the subclass.**
  - super(arg-list); // for example super(w, h, d);
  - **Super can also be used as 'this'. If variable name in both subclass and superclass are same then in the subclass, the local variable in subclass will always hide the variable in superclass. To access the variable of superclass, super term is used.**
  - super.member; // for example super.i

- **Overriding**
  - **When inheriting, the subclass can override the method in superclass provide both the methods have same name, same input argument and same output argument**

```
class Animal {
   public void move() {
      System.out.println("Animals can move");
   }
}

class Dog extends Animal {
   public void move() {
      System.out.println("Dogs can walk and run");
   }
}
```

```
public class TestDog {

   public static void main(String args[]) {
      Animal a = new Animal();   // Animal reference and object
      Animal b = new Dog();   // Animal reference but Dog object

      a.move();   // runs the method in Animal class
      b.move();   // runs the method in Dog class
   }
}
```

This will produce the following result

```
Animals can move
Dogs can walk and run
```

Another example is

```
class Animal {
   public void move() {
      System.out.println("Animals can move");
   }
}

class Dog extends Animal {
   public void move() {
      System.out.println("Dogs can walk and run");
   }
   public void bark() {
      System.out.println("Dogs can bark");
   }
}

public class TestDog {

   public static void main(String args[]) {
      Animal a = new Animal();   // Animal reference and object
      Animal b = new Dog();   // Animal reference but Dog object

      a.move();   // runs the method in Animal class
      b.move();   // runs the method in Dog class
      b.bark();
   }
}
```

In the above example b.bark() will give an error because b is of type Animal

- **Abstraction**
  - **Abstraction Classes**
    - **Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user.**
    - **An abstract class may or may not contain abstract methods.**
    - **If a class has one abstract method then the class must be declared abstract.**
    - **If a class is declared abstract it cannot be instantiated.**
    - **To use an abstract class, you have to inherit it from another class and provide implementations to the abstract methods in it.**
    - **If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.**
    - ```
      public abstract class Employee {
          private String name;
          private String address;
          private int number;
      
      ```
      ```

```
    public Employee(String name, String address, int number) {
        System.out.println("Constructing an Employee");
        this.name = name;
        this.address = address;
        this.number = number;
    }

    public double computePay() {
        System.out.println("Inside Employee computePay");
        return 0.0;
    }
}
```

An abstract class cannot be instantiated so the following line will give an error.

```
Employee e = new Employee("George W.", "Houston, TX", 43);
```

Because the abstract class cannot be instantiated it has to be inherited.

```
public class Salary extends Employee {
    private double salary;   // Annual salary

    public Salary(String name, String address, int number, double salary){
        super(name, address, number);
        setSalary(salary);
    }
    public double computePay() {
        System.out.println("Computing salary pay for " + getName());
        return salary/52;
    }
}
```

Now the salary class can be instantiated

```
public class AbstractDemo {
    public static void main(String [] args) {
        Salary s = new Salary("Mohd", "Ambehta, UP", 3, 3600.00);
        Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
        System.out.println("Call mailCheck using Salary reference --");
        s.computePay();
        System.out.println("\n Call mailCheck using Employee reference");
        e.computePay();
    }
}
```

- **Abstraction Methods**
  - **In this case, the class has a particular method but the actual implementation of that method is determined by child classes. The method is declared in the parent class as abstract.**
  - **Abstract keyword is used to declare the method as abstract**
  - **Abstract method has no body**
  - **Instead of curly braces, an abstract method have a semi colon.**
  - **Implementation of all abstract methods has to be provided.**

```
public abstract class Employee {
    private String name;
    private String address;
    private int number;

    public abstract double computePay();
    // Remainder of class definition
}
public class Salary extends Employee {
    private double salary;   // Annual salary
```

```
    public double computePay() {
        System.out.println("Computing salary pay for " + getName());
        return salary/52;
    }
    // Remainder of class definition
}
```

- **Encapsulation**
  - **It is also known as data hiding**
  - **In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their class.**
  - **For encapsulation the following has to be done:**
    - **Declare the variables of a class as private.**
    - **Provide public setter and getter methods to modify and view the variable values.**

```
public class EncapTest {
    private String name;
    private String idNum;
    private int age;

    public int getAge() {
        return age;
    }

    public String getName() {
        return name;
    }

    public String getIdNum() {
        return idNum;
    }

    public void setAge( int newAge) {
        age = newAge;
    }

    public void setName(String newName) {
        name = newName;
    }

    public void setIdNum( String newId) {
        idNum = newId;
    }
}
```

  - **The public setXXX() and getXXX() methods are the access points of the instance variables of the EncapTest class. Normally, these methods are referred as getters and setters. Therefore, any class that wants to access the variables should access them through these getters and setters.**

```
public class RunEncap {

    public static void main(String args[]) {
        EncapTest encap = new EncapTest();
        encap.setName("James");
        encap.setAge(20);
        encap.setIdNum("12343ms");
```

```
        System.out.print("Name : " + encap.getName() + " Age : " +
encap.getAge());
    }
}
```

- **Interface**
  - **An interface is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. A class describes the attributes of an object but an interface describes the attributes of a class.**
  - **An interface is similar to class in the following way:**
    - **An interface can contain any number of methods.**
    - **An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.**
    - **The byte code of an interface appears in a .class file.**
    - **Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.**
  - **An interface is different from class in the following way:**
    - **Interface cannot be instantiated.**
    - **An interface does not contain any constructors.**
    - **All of the methods in an interface are abstract.**
    - **An interface is not extended by a class, it is implemented by a class.**
    - **An interface can extend multiple interfaces**
  - **Properties of interface is:**
    - **An interface is implicitly abstract. You do not need to use the abstract keyword while declaring an interface.**
    - **Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.**
    - **Methods in an interface are implicitly public.**

```
interface Animal {
    public void eat();
    public void travel();
}
public class MammalInt implements Animal {

    public void eat() {
        System.out.println("Mammal eats");
    }

    public void travel() {
        System.out.println("Mammal travels");
    }

    public int noOfLegs() {
        return 0;
    }

    public static void main(String args[]) {
        MammalInt m = new MammalInt();
        m.eat();
        m.travel();
    }
}
```

  - **A class can extend one class but can implement many interfaces**
    - public class A implements C,D {...} // A is a class and C and D are interfaces
  - **Multiple inheritances between interfaces is also allowed**
    - public interface A extends C,D {...} // A, C and D, all are interfaces

- **Interface can extend another interface in the same way that a class can extend another class**

- **Difference between interface and abstraction**
  - **Interface**
    - **An interface is an empty shell. There are only the signatures of the methods, which implies that the methods do not have a body. The interface can't do anything. It's just a pattern.**
    - **This is an interface not a class while abstract is a class**
    - **Interface supports multiple inheritance**

```
// I say all motor vehicles should look like this:
interface MotorVehicle
{
    void run();

    int getFuel();
}

// My team mate complies and writes vehicle looking that way
class Car implements MotorVehicle
{
    int fuel;

    void run()
    {
        print("Wrroooooooom");
    }


    int getFuel()
    {
        return this.fuel;
    }
}
```

  - **Abstraction**
    - **Abstract classes look a lot like interfaces, but they have something more: You can define a behavior for them. It's more about a guy saying, "these classes should look like that, and they have that in common, so fill in the blanks!".**
    - **This is basically a class.**
    - **Abstract class does not support multiple inheritance.**
    - **Implementation of all abstract methods has to be provided.**

```
// I say all motor vehicles should look like this:
abstract class MotorVehicle
{
    int fuel;

    // They ALL have fuel, so lets implement this for everybody.
    int getFuel()
    {
         return this.fuel;
    }

    // That can be very different, force them to provide their
    // own implementation.
    abstract void run();
}

// My teammate complies and writes vehicle looking that way
```

```
class Car extends MotorVehicle
{
    void run()
    {
        print("Wrroooooooom");
    }
}
```

- **Difference between Abstraction and Inheritance**
  - **Basically any class can be inherited but abstract class are specifically made to be inherited.**