**DAY 1 - BASICS OF JAVA - COURSE**

**TABLE OF CONTENTS**

* Java Tokens, Variables, Keywords and Data Types
* New, Class, Object
* Access modifiers
* Constructors
* Inheritance
* Polymorphism
* Abstract Class and Abstract Method
* Interfaces
* Garbage Collection
* Package and Import
* Stack Trace
* Exception Handling
* Reading from Excel File
* Date Object to String Conversion
* String Builders
* Exception Handling
* Assignment Problems
* Collections – Integer Array, ArrayList, linked list, hash set, hash map
* Data structures – Linear Search, Binary search
* Sorts – Bubble and Merge Sort
* Serialization and de-serialization

**JAVA TOKENS – Identifiers, Keywords, Literals, Operators and Separators**

A token is the smallest element in a program that is meaningful to the compiler.

Five types of token exist as seen in the table below -

|  |  |  |
| --- | --- | --- |
| A | Identifiers | Identifiers are names provided by you. These can be assigned to variables, methods, functions, classes etc. to uniquely identify them to the compiler.   1. Example Code: 2. public class Test 3. { int a; 4. public void setValueA() { 5. this.a = 22; } 6. } 7. In the above java code, we have ‘3’ identifiers namely : 8. Test - class name. 9. setValueA - method name. 10. a - variable name. |
| B | Keywords | Keywords are reserved words that have a specific meaning for the compiler. They cannot be used as identifiers. Java has a rich set of keywords. Some examples are: boolean, char, if, protected, new, this, try, catch, null, threadsafe etc. |

|  |  |  |
| --- | --- | --- |
| C | Literals | Literals are variables whose values remain constant throughout the program. They are also called Constants. Literals can be of four types. They are:  a. String Literals  b. Character Literals  c. Boolean Literals  d. Numeric Literals   1. Example Code: 2. String a = "This is a string"; 3. char b = '\u041D'; 4. boolean c = true; 5. int d = 11; 6. float e = 10.5; 7. System.out.println("a = "+a); 8. System.out.println("b = "+b); 9. System.out.println("c =  "+c); 10. System.out.println("d = "+d); 11. System.out.println("e = "+e); |
| D | Operators | An operator is a symbol that operates on one or more operands to produce a result.   1. Example Code: 2. class OperatorExample{ 3. public static void main(String args[]){ 4. int x=10; 5. System.out.println(x++);//10 (11) 6. System.out.println(++x);//12 7. System.out.println(x--);//12 (11) 8. System.out.println(--x);//10 9. boolean c=true; 10. boolean d=false; 11. System.out.println(!c);//false (opposite of boolean value) 12. System.out.println(!d);//true 13. int a=10; 14. int b=5; 15. System.out.println(a+b);//15 16. System.out.println(a-b);//5 17. System.out.println(a\*b);//50 18. System.out.println(a/b);//2 19. System.out.println(a%b);//0 20. System.out.println(10\*10/5+3-1\*4/2); 21. System.out.println(15<<4);//15\*2^4=15\*16=240 22. }} |
| E | Separators | Separators are symbols that indicate the division and arrangement of groups of code.   1. Example Code: 2. /\* Java Program Example - Java Separators 3. \* This program contains some Separators 4. \*/ 5. class JavaProgram 6. { 7. public static void main(String args[]) 8. { 10. int i, j=10, k=11; // uses comma or , separator 12. for(i=0; i<j; i++) // uses parentheses or () separator 13. { // uses braces or {} separator 14. System.out.println("I am " + i); 15. } 16. System.out.println(k + " is biggest here."); 18. } 19. } |

**JAVA – VARIABLES**

|  |  |
| --- | --- |
| Instance Variables | Objects store their individual states in “non-static fields”, that is,  fields declared without the static keyword   1. Example Code: 2. class A{ 3. int data=50;//instance variable 4. static int m=100;//static variable 5. void method(){ 6. int n=90;//local variable 7. } 8. }//end of class |
| Class Variables | Exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. E.g. static int numGears = 5;   1. Example Code: 2. class A{ 3. int data=50;//instance variable 4. static int m=100;//static variable 5. void method(){ 6. int n=90;//local variable 7. } 8. }//end of class |
| Local Variables | A method stores its temporary state in local variables.   1. Example Code: 2. class A{ 3. int data=50;//instance variable 4. static int m=100;//static variable 5. void method(){ 6. int n=90;//local variable 7. } 8. }//end of class |
| Parameters | They are the variables that are passed to the methods of a class.   |  | | --- | | 1. Example Code : 2. public class Example { 4. public static int multiply(int a, int b) 5. { 6. return a + b; 7. } 9. public static void main(String[] args) 10. { 11. int x = 2; 12. int y = 5; 14. // the variables x and y are arguments 15. int sum = multiply(x, y); 17. System.out.println("SUM IS: " + sum); 18. } 19. } |   Output:  SUM IS: 7 |

**1) New operator, Class and Object:-**

|  |
| --- |
| New:-  The new operator dynamically allocates memory for an object during run time.  Advantages:-   * Your program can create as many or as few objects as it needs during the evaluation of your program.   Dis-Advantages:-   * Since memory is finite, it is possible that new will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur. |

|  |
| --- |
| 1. class Box 2. { 3. double width; 4. double height; 5. double depth; 6. } 8. // Driver class 9. public class Test 10. { 11. // Driver method 12. public static void main(String[] args) 13. { 14. // Creating box object 15. Box b1 = new Box(); 16. b1.height = 1; 18. // Assigning b2 to b1 19. Box b2 = b1; 21. // Height via b1 and b2 22. System.out.println(b1.height); 23. System.out.println(b2.height); 25. // Changing height via b2 26. b2.height = 22; 28. // Height via b1 and b2 29. // After modification through b2 30. System.out.println(b1.height); 31. System.out.println(b2.height); 32. } 34. }   Output:  1  1  22  22 |

|  |
| --- |
| Class:-  A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members.  A class is a blueprint containing members/data types and methods. |

|  |
| --- |
| Object:-  Object is an instance of that class. Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.) |

**2) Assembly Class Access Modifiers**

Access modifiers are those which **set access levels** for classes, attributes, methods and constructors.

|  |
| --- |
| Assembly Class Java Access Modifiers for Class Level   * Default = Accessed only within the package. * Public = Accessed outside the package. * Final = Final class level it cannot be sub classed but can be instantiated. * Abstract = Abstract has to be sub classed and not instantiated. |

|  |
| --- |
| Assembly Class Access Modifiers for Attributes Level   * Public = Attribute can be accessed by class/objects in any package * Private = Attribute can be accessed from only within the class. * Protected = Attributes can be accessed from within the same package and the sub classes outside the package. * Default = Attributes can be accessed by same class or other classes/objects within the same package. * Final = The value of the attribute cannot be changed. Only one value can be assigned * Static = Only one value of the attribute per class. Remains the same in all the objects |

|  |
| --- |
| Assembly Class Access Modifiers = Method Level   * Public = The method can be accessed by a class or object in any package * Private = Method can be accessed from only within the class * Protected = Method can be accessed by other classes or objects within the same package and the class/objects sub-classing this methods’ class outside the package * Default = Method can be accessed only be the other classes/objects within the same package and the class/objects sub-classing this methods’ class outside the package. * Final = Method cannot be overridden * Abstract = Only provides the method declaration not the definition. It can be inherited and can be overridden by sub classes. * Synchronized = Only one thread can access the method at a time   Static = They are those class level methods which don’t belong to any particular instance. In a static method there cannot be an instance variable; A static method is not allowed to read or write the non-static methods of its class. |

Private Method Example:

class A{

private int data=40;

private void msg(){System.out.println("Hello java");}

}

public class Simple{

 public static void main(String args[]){

   A obj=new A();

   System.out.println(obj.data);//Compile Time Error

   obj.msg();//Compile Time Error

   }

}

Default Method Example:

//save by A.java

package pack;

class A{

  void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.\*;

class B{

  public static void main(String args[]){

   A obj = new A();//Compile Time Error

   obj.msg();//Compile Time Error

  }

}

Protected Method Example:

//save by A.java

package pack;

public class A{

protected void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.\*;

class B extends A{

  public static void main(String args[]){

   B obj = new B();

   obj.msg();

  }

}

Public Method Example:

//save by A.java

package pack;

public class A{

public void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.\*;

class B{

  public static void main(String args[]){

   A obj = new A();

   obj.msg();

  }

}

Final variable -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.

Example Code:

class Bike9{

 final int speedlimit=90;//final variable

 void run(){

  speedlimit=400;

 }

 public static void main(String args[]){

 Bike9 obj=new  Bike9();

 obj.run();

 }

}//end of class

Output:Compile Time Error

Java final method - If you make any method as final, you can inherit it but you cannot override it.

Example Code:

class Bike{

  final void run(){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();

   }

}

Output:Compile Time Error

Java final class - If you make any class as final, you cannot extend it.

final class Bike{}

class Honda1 extends Bike{

  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){

  Honda1 honda= new Honda1();

  honda.run();

  }

}

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=Honda1)

Output: Compile Time Error

|  |
| --- |
| Assembly Class Final  Final Class = Cannot be extended within and outside the package  Final Function = Cannot be overridden in the sub class.  Final Variable = Value is like constant; New value cannot be assigned or modified anywhere. |

|  |
| --- |
| Class Diagram for a Class Attributes and Methods of Washing Machine and ER Diagram  Washing Machine color: make: size: model: spin() rinse() model() |

|  |
| --- |
| Explain “static” access modifier as applied to attributes, methods and classes.   * Static attribute – It is like a constant. Value of the variable cannot change inside the class or inside the package. Only one value can be assigned to it. And it remains constant throughout the class or package. * Static method – can be called directly without creation of object. E.g. public static void main. Can be called directly without requirement of object creation. * Static class – Static is applied to inner classes present inside an outer class. Only nested classes can be static. Outer class cannot be static. Nested classes no need reference of Outer class but inner classes do need reference of outer class. Nested classes can directly access static members but not non static members of outer class. |

|  |
| --- |
| Example code for static method and variables:  class Student  {      String name;      int rollNo;        // static variable      static String cllgName;        // static counter to set unique roll no      static int counter = 0;          public Student(String name)      {          this.name = name;            this.rollNo = setRollNo();      }        // getting unique rollNo      // through static variable(counter)      static int setRollNo()      {          counter++;          return counter;      }        // static method      static void setCllg(String name){          cllgName = name ;      }        // instance method      void getStudentInfo(){          System.out.println("name : " + this.name);          System.out.println("rollNo : " + this.rollNo);            // accessing static variable          System.out.println("cllgName : " + cllgName);      }  }    //Driver class  public class StaticDemo  {      public static void main(String[] args)      {          // calling static method          // without instantiating Student class          Student.setCllg("XYZ");            Student s1 = new Student("Alice");          Student s2 = new Student("Bob");            s1.getStudentInfo();          s2.getStudentInfo();        }  } |

Output:

name : Alice

rollNo : 1

cllgName : XYZ

name : Bob

rollNo : 2

cllgName : XYZ

|  |
| --- |
| 3) Constructors   * A constructor initializes attributes after object is created. * It has same name as that of the class in which it resides, Once defined the constructor is automatically called immediately after the object creation. * No return type * Constructor’s job is to initialize the inner state of an object so that the code creating the instance of the class will have a fully initialized usable object immediately. * Two Types * Default Constructor – When you do not explicitly define a constructor for a class, then java creates a constructor for the class * Parameterized Constructor – When the object is created, the user can send the values to the constructor to initialize the attributes. |

**Example:-**

|  |
| --- |
| public class Person {  int age;  short weight;  Person(int a, short w) {  age = a;  weight = w; } }  When the object is created, it is like>> Person Sunil = new Person(25,75); |

**Copy Constructor**

* Frequently we need to use the same values used in the different object to initialize the current object. In that case we need to pass the object itself as the parameter.

|  |
| --- |
| public CubePrime {  double w; double h; double l;  Cube(double a,double b, double c)  { w=a; h=b; l=c; }  Cube (Cube obj) { w = obj.w; h= obj.h; l= obj.l; }  public static void main (String args) {  Cube obj1 = new Cube(10,20,30);  Cube obj2 = new Cube(obj1); } } |

**Static Methods**

* Static methods are class level methods which don’t belong to one particular instance. Instance method and Instance variable versus Static method and static variable;

**4) Inheritance**

* A class has data and methods. Inheritance is how a class can get data and methods present in another class. Using extends keyword.

|  |
| --- |
| public class A {  int i,j;  void showij() {  System.out.println(i+""+ j); } }    public class B extends A {  int k;  void showk() { System.out.println(k);}  void sum() { int sum = i+j+k; System.out.println(sum); } }  public static void main(String args[]) {  A superOb = new A();  B subOb = new B();  // Below can be done since B inherits the members and methods of A  subOb.i = 7; subOb.j = 8; subOb.k= 9;  subOb.sum();// Outputs 24 as result } } |

Java does not support multiple Inheritance.

**Polymorphism**

|  |
| --- |
| Overloading Methods is Compile Time Polymorphism   * When we define two or more methods within the same class that share the same name but their parameter declaration are different the methods are said to be overloaded and the process is said to be method overloading. * In own words suppose there are three methods namely add(double a, double b) and add ()   And add (int a, int b). All three have same name but different parameters. When the object of the class is defined and called as obj.add() obj.add(10.2,10.4) or obj (1,3) the respective methods are called which have no parameters, double parameters and int parameters respectively |

**Overloading Constructors is also Polymorphism**

* Constructors can also be overloaded;

**Example**:-

|  |
| --- |
| public class Cube {  double width; double height; double length;  // Three overloaded constructors  Cube() { width = -1; height =-1; length = -1; }  Cube (double x) {width = x; height = x; length = x;}  Cube(double w,double h,double l) {width = w; height=h; length = l;}  public static void main (String args[]) { // All three are valid  Cube c1 = new Cube();  Cube c2 = new Cube(3.2);  Cube c3= new Cube (3.1, 2.4, 4.5); } } |

**Method Overriding**

|  |
| --- |
| * When the method with same name is present in the super class and same name and signature is created and defined in the sub class. Then the method of the sub class overrides the method of the super class. i.e. when the object of sub class calls this method, the method of the sub class is called instead of the super class. |

|  |
| --- |
| class A {  int i, int j;  A(int a, int b) {i=a;j=b;}  void show() {System.out.println(i+” ”+j);} }  class B extends A { int k;  B (int a, int b, int c) { super(a,b); k =c; }  void show() {System.out.println(k);}  public static void main(String args[]) {  B subOb = new B (1,2,3);  subOb.show(); // Calls the subOb’s show method which was overridden  }} |

**Run Time Polymorphism and Dynamic Dispatch**

|  |
| --- |
| Call to an overridden method is resolved at run time   * In own words, suppose a class has members and method. The sub class inheriting from the super class has the same method (overridden method). Now we define a super class object and a sub class object. Now when we create a reference variable of super class and then assign the object of the sub class, then at run time the compiler determines at run time that the sub class method needs to be called.   Concept = Reference variable of a super class can be assigned the object of a sub class. This causes run time polymorphism. |

Example

|  |
| --- |
| class A {  void callme() {  System.out.println(“Inside A’s callme method”);}}  class B extends A {// Call me is overridden herevoid callme() { System.out.println(“Inside B’s callme method”);}}  class C extends A {// Callme is overridden here toovoid callme() {System.out.println(“Inside C’s callme method”);  public static void main (String args[]) {A r1 = new A(); r1.show();  // A’s method is called  A r2 = new B();r2.sow(); // B’s method is called  // Run time polymorphism  A r3 = new C();  r3.show(); // C ‘s method is called// Run time polymorphism  }} |

**Abstract Class and Abstract Method**

|  |
| --- |
| * Abstract class cannot be instantiated but can be sub classed. * Many situations where a super class has only the generalized form and the sub classes define the methods. Example Figure class containing parameters as l, b, h and area function and triangle extending Figure class and defining the area. * If we require a method be overridden by the sub classes, then we add abstract key word to it. * If more than one abstract method is present in a class then the class need to have keyword called abstract * An abstract class cannot be instantiated. |

**Example:**

|  |
| --- |
| abstract class Figure {  abstract area(); }  public class Triangle extends Figure {  int l,b,h;  void area(){  System.out.println(l\*b\*h); }  public static void main(String args[]) {  Triangle tri1 = new Triangle();  Tri1.area(); // This prints area of triangle overridden from Figure class. } |

**Interfaces**

|  |
| --- |
| 1) Interface cannot be instantiated.  2) Interface is a list of public method declarations.  3) When the classes implement the interface, it is class’s responsibility to implement all the methods  4) None of the methods in the interface are defined.  5) A sub-class is allowed to implement multiple interfaces.  5.1) Interface has no state. It means that variables are treated as final static. The implementing class need not have the same variables. If required, then switch to abstract class. |

**Example**

|  |
| --- |
| public interface Figure {  public void area();  public void perimeter(); }  public class Rectangle implements Figure {  int l, int b;  public void area() { int area = l \*b;}  public void perimeter() {int peri = 2\*(l+b); } }  public class Square implements Figure {  int s; public void area () {int area = s \* s;}  public void perimeter() {int perimeter = 4\*s} } |

**Garbage Collection**

|  |
| --- |
| * The problem is that the programs create objects which use a sizeable memory space and other resources. Now when the objects are no longer in use they fill up the space and create memory and other resource constraints. * Garbage collector works this way. It remembers all the variables belonging to the program and the objects which are pointed by these variables. These are called Reachable objects. Unreachable objects are those that are not pointed by any of the current programs. * The garbage collector keeps track of the unreachable objects and deletes them and clears the memory space. * The GC acts in the background and cannot be controlled. * Package and Import * Package is like a directory. It contains group of classes and sub directories or sub packages. * It can be used to group classes with common purpose together and we can impart access modifiers to the package so that the classes and objects outside the package cannot access classes inside the package. * Import = If we need to use a public class present in another package we need to import the contents of the package into the current class and then call the respective class for inheritance or other purposes. * Stack Trace * The JVM stores the functions in LIFO format as stack at run time. When the main method is executed then main becomes bottom of stack and then any object called, the constructor takes the position and then any other method called takes the position above the stack. * Now, when any method shows abnormal behavior, then error report is created known as stack trace. This is useful for debugging of the code. * Exception Handling * Java exception Handling is managed via five key words = try, catch, finally, throw and throws. * Exception handling means error handling. Error due to programmer or due to the system happening at the run time. * Exception Process - When the exception happens, an object representing that exception is created and thrown in the method that caused the error. The method may catch it or throw back. At some point exception is caught and then processed. |

**Example**

|  |
| --- |
| class Exc0 {  public static void main(String args[]) {  int d=0; int a = 42/d;  // We are attempting to divide by zero. }}  When the program is run we get the following output:-   * lang.ArithmeticException: Divide by zero   at Exc0.main(Exc0.java:4) |

|  |
| --- |
| * Reading the compiler message, it is a Arithmetic Exception; When this exception is found by the compiler interrupting the normal behavior or flow of process, then object of exception is created and at once handled by the Default Handler provided by Java Run Time system which displays a string displaying the type of exception and the line of the code where the exception had occurred |

.

**Using try and Catch**

|  |
| --- |
| class Exc0 {  public static void main (String args[]) {  int d,a;  try { // Monitoring the block of code for Arithmetic Exception d =0; a = 42/d;  System.out.println(“This line is not printed.”); }  catch (ArithmeticException e) {  System.out.println(“Division by Zero”); } }} |

**Multiple Catch Clauses**

|  |
| --- |
| class MultiCatch {  public static void main (String args[]) {  try { // Manually monitoring the block for  int a = Integer.parseInt(args[0]);  int b = 42/a; }  catch (ArithmeticException ae ) {  System.out.println(“Divide by zero”); }  catch(ArrayOutOBoundsException e) {  System.out.println(“No Command line argument present.”); }  catch (NumberFormatException e) {  System.out.println(“Command Line argument is not a number.”); } }} |

**Finally Keyword:-**

|  |
| --- |
| You can attach a finally-clause to a try-catch block. The code inside the finally clause will always be executed, even if an exception is thrown from within the try or catch block. If your code has a return statement inside the try or catch block, the code inside the finally-block will get executed before returning from the method. Here is how a finally clause looks: |

|  |
| --- |
| public void openFile(){  FileReader reader = null;  try {  reader = new FileReader("someFile");  int i = reader.read(); }  } catch (IOException e) {  System.out.println(“IOException. Reader not reading.”);  } finally {  if(reader != null){  try {  reader.close(); }  catch (IOException e) {  System.out.println(“IO Exception. File not closing.”); } }  System.out.println("--- File End ---"); } } |

|  |
| --- |
|  |

1. **Reading from Excel File:-**

|  |
| --- |
| public class ReadExcelFile {  public void readExcel() throws BiffException, IOException {  String FilePath = "D:\\sampledoc.xls";  FileInputStream fs = new FileInputStream(FilePath);  Workbook wb = Workbook.getWorkbook(fs);// TO get the access to the sheet  Sheet sh = wb.getSheet("Sheet1");// To get the number of rows present in sheet  int totalNoOfRows = sh.getRows();// To get the number of columns present in sheet  int totalNoOfCols = sh.getColumns();  for (int row = 0; row < totalNoOfRows; row++) {  for (int col = 0; col < totalNoOfCols; col++) {  System.out.print(sh.getCell(col, row).getContents() + "\t"); }  System.out.println(); } }  public static void main(String args[]) throws BiffException, IOException {  ReadExcelFile DT = new ReadExcelFile(); DT.readExcel(); } }  Output  Username password testuser1 testpassword1 testuser2 testpassword2 testuser3 testpassword3 testuser4 testpassword4  Output:system oracle |

**Enum Example**

|  |
| --- |
| class EnumExample1{  public enum Season { WINTER, SPRING, SUMMER, FALL }  public static void main(String[] args) {  for (Season s : Season.values())  System.out.println(s); }}  Output:WINTERSPRINGSUMMERFALL |

**Enum with specific values inside a class:-**

|  |
| --- |
| class EnumExample4{  enum Season{ WINTER(5), SPRING(10), SUMMER(15), FALL(20);  private int value;  private Season(int value){ this.value=value; } }  public static void main(String args[]){  for (Season s : Season.values()) System.out.println(s+" "+s.value); }}  Output:WINTER 5SPRING 10SUMMER 15FALL 20 |

**Date Object to String conversion**

|  |  |  |
| --- | --- | --- |
| Letter | Description | Examples |
| y | Year | 2013 |
| M | Month in year | July, 07, 7 |
| d | Day in month | 1-31 |
| E | Day name in week | Friday, Sunday |
| a | Am/pm marker | AM, PM |
| H | Hour in day | 0-23 |
| H | Hour in am/pm | 1-12 |
| m | Minute in hour | 0-60 |
| s | Second in minute | 0-60 |

**Note**  
 **For complete date and time patterns, please refer to this** [java.text.SimpleDateFormat JavaDoc](http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html)

1. **Date Example**

If ‘M’ is 3 or more, then the month is interpreted as text, else number

1. **Date = 7-Jun-2013**

|  |
| --- |
| SimpleDateFormat formatter = new SimpleDateFormat("dd-MMM-yyyy"); String dateInString = "7-Jun-2013";  try { Date date = formatter.parse(dateInString); System.out.println(date);  System.out.println(formatter.format(date));  } catch (ParseException e) { e.printStackTrace(); }  OutputFri Jun 07 00:00:00 MYT 201307-Jun-2013 |

1. **Date = 07/06/2013**

|  |
| --- |
| SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy"); String dateInString = "07/06/2013"; try { Date date = formatter.parse(dateInString); System.out.println(date); System.out.println(formatter.format(date)); } catch (ParseException e) { e.printStackTrace(); }OutputFri Jun 07 00:00:00 MYT 201307/06/2013 |

1. **Date = Jun 7, 2013**

|  |
| --- |
| SimpleDateFormat formatter = new SimpleDateFormat("MMM dd, yyyy"); String dateInString = "Jun 7, 2013"; try { Date date = formatter.parse(dateInString); System.out.println(date); System.out.println(formatter.format(date)); } catch (ParseException e) { e.printStackTrace(); }OutputFri Jun 07 00:00:00 MYT 2013Jun 07, 2013 |

1. **Date = Fri, June 7 2013**

|  |
| --- |
| SimpleDateFormat formatter = new SimpleDateFormat("E, MMM dd yyyy"); String dateInString = "Fri, June 7 2013"; try { Date date = formatter.parse(dateInString); System.out.println(date); System.out.println(formatter.format(date)); } catch (ParseException e) { e.printStackTrace(); }OutputFri Jun 07 00:00:00 MYT 2013Fri, Jun 07 2013 |

1. **Date and Time Example**
2. Date and Time = **Friday, Jun 7, 2013 12:10:56 PM**

|  |
| --- |
| SimpleDateFormat formatter = new SimpleDateFormat("EEEE, MMM dd, yyyy HH:mm:ss a"); String dateInString = "Friday, Jun 7, 2013 12:10:56 PM"; try { Date date = formatter.parse(dateInString); System.out.println(date); System.out.println(formatter.format(date)); } catch (ParseException e) { e.printStackTrace(); }OutputFri Jun 07 12:10:56 MYT 2013Friday, Jun 07, 2013 12:10:56 PM |

**String Builder**

|  |
| --- |
| public class StringBuilderClass {  final static String filename = "C:\\Users\\user\\Dropbox\\JOB INTERVIEW FOLDER\\JAVA" + "[\\entire-java-code-repo\\FileReaderTextFile.txt](file:///\\entire-java-code-repo\\FileReaderTextFile.txt)";  Logger logger = Logger.getLogger(StringBuilderClass.class);  public static void main(String[] args) {  // StringBuilder with 16 empty elements  StringBuilder sb = new StringBuilder();  sb.append("Hello from JCG");  System.out.println("sb appends a string: " + sb);  // append a character  char c = '!';  sb.append(c);  System.out.println("sb after appending a char: " + sb);  sb.insert(6, "everyone ");  System.out.println("sb after insert: " + sb);  // StringBulder with a initialized capacity  StringBuilder sbnew = new StringBuilder(15);  sbnew.append(123456789);  System.out.println("sb with length " + sbnew.length() + " and capacity " + sbnew.capacity() + " appends an int: " + sbnew);  // delete 234  sbnew.delete(1, 4);  System.out.println("sb after delete: " + sbnew);  // read from a file and append into a StringBuilder every new line try { BufferedReader br = new BufferedReader(new FileReader(filename)); StringBuilder sbFile = new StringBuilder(); String line = br.readLine(); while (line != null) { // append the line of the file sbFile.append(line); // separate the line with a '@' sbFile.append('@'); // read the next line of the file line = br.readLine(); } // this string contains the character sequence String readFile = sbFile.toString(); br.close(); System.out.println("from file: " + readFile); } catch (FileNotFoundException e) { e.printStackTrace(); } catch (IOException e) { e.printStackTrace(); } }}  Output  sb appends a string: Hello from JCG  sb after appending a char: Hello from JCG!  sb after insert: Hello everyone from JCG!  sb with length 9 and capacity 15 appends an int: 1234567  sb after delete: 1567from file: first line; @second line; @third line; @fourth line; @fifth line;@sixth line; @seventh line; @ |

**Sample Page Object method**

|  |
| --- |
| @Override  public void waitToBeDisplayed(long waitTime) throws RequirementsException |

**COLLECTIONS**

**Integer Array Declaration and Sizing**

|  |
| --- |
| int check[] = {1,1,2,3,5,8};  int n = check.length // checking the size.  int fibo[] = new int[10];  fibo[0] = 1; // filling values  fibo[1] = 1;  int m = fibo.length; // checking the size. |

**ArrayList Methods - Ordered List**

|  |
| --- |
| ArrayList<String> arraylist = new ArrayList<String>();  arrayList.add(“JAVA”);  arrayList.add(2,”C++”);  arrayList.isEmpty() // verify if arrayList is empty  arrayList.indexOf(“C++”) // return index of element  arrayList.size() // Size of arrayListsArrayList can be read using IteratorIterator<String> arrayIterator = arrayList.iterator();  ArrayList<String> cloneArrayList = (ArrayList<String>)  arrayList.clone(); // Clone arraylist  arryaList1.addAll(arrayList2); // Add elements of array2 into array1  subList1 = arrayList1.subList(2,4); // Sub list of array list.  //Iterator for loop  for (String stringElement: arrayList)System.out.println(stringElement); |

**Iterator**

|  |
| --- |
| List<String> myList = new ArrayList<String>();  myList.add(“Java”);  myList.add(“Unix”);  Iterator<String> itr = myList.iterator();  while(itr.hasNext()) {  System.out.println(itr.next());}  Iterator itr = list1.iterator();itr.hasNext() – Navigation  itr.next() – return element  itr.remove() – remove element |

**LinkedList - Unordered list- Sunil Kumar Gunasekaran**

|  |
| --- |
| LinkedList<String> L1 = new LinkedList<String>();  L1.add(“Orange”);L1.add(“Apple”);  //Methods  L1.size();L1.isEmpty();  L1.contains(“element”) – Read elements via iterator.  linkedList.addAll(arrayList); - Add elements of arraylist to Linked ListlinkedList.clear() – delete all elements in linked listlinkedListIterator.remove() - Individual delete |

**HashSet (Unordered, Unique list)**

|  |
| --- |
| HashSet<String> HS1 = new HashSet<String>();  HS1.add(“first”);HS1.remove(“first”);Use iterator to navigate through hashSetIterator <String> HSItr1= HS1.iterator();while (HSItr1.hasNext()) {HSItr1.next();} HS1.addAll(HS2); // Add all elements from another hash setHS1.clear() // Clear all elements of hash set |

**HashMap - (unsorted key value pair)**

|  |
| --- |
| HashMap<String, String> hashMap = new HashMap<String, String>();  hashMap.put(“first”,”FIRST INSERTED”);  hashMap.put(“second”, “SECOND INSERTED”);  hashMap.put(“third”, “THIRD INSERTED”);  hashMap.get(“first”); // get value of keyhashMap.size(); - size of hashMaphashMap.isEmpty(); // boolean of emtpy array |

* **DATA STRUCTURES**

**Linear Search Return location**

|  |
| --- |
| O(n)Speed of iteration grows  public static int linearSearch(int[] arrayList,int searchValue) {  int size = arrayList.length;for (int i=0; i< size; i++) { if(arrayList[i]==searchValue)return i;} |

**Binary Search Return location for sorted array**

**log(n) Speed of iteration**

|  |
| --- |
| public int binarySearch(ArrayList<Integer> arrayList, searchValue)  {{int start =0;int end = arrayList.size() -1;  while (start<=end) {int mid = (start+ end)/2;  if (searchValue == arrayList.get(mid))return mid;  if (searchValue > arrayList.get(mid))start = mid + 1;  if (searchValue < arrayList.get(mid)) end = mid -1; }return -1;} |

**Bubble Sort n^2 – Speed of computation**

|  |
| --- |
| public static void bubble\_srt( int a[], int n ){ int i, j,t=0;  for (i = 0; i < n; i++) {  // since highest value is put at the last in first iteration  for (j = 1; j < n-i; j++) {  if(a[j-1] > a[j]) {  t = a[j-1];  a[j-1]=a[j];  a[j]=t; } } } }// end of bubble\_srt() |

**Merge Sort n log n – Fastest computation speed Gravity Sort**

|  |
| --- |
| TopDownMergeSort(A[], B[], n){  CopyArray(A, 0, n, B);  TopDownSplitMerge(B, 0, n, A);  }  TopDownSplitMerge(B[], begin, end, A[]){  if(end - begin < 2) return;  middle = (end + begin) / 2;  TopDownSplitMerge(A, begin, middle, B);  TopDownSplitMerge(A, middle, end, B);  TopDownMerge(B, begin, middle, end, A);}TopDownMerge(A[], begin, middle, end, B[]) { i = begin, j = middle; for (k = begin; k < end; k++) { if (i < middle && (j >= end || A[i] <= A[j])) { B[k] = A[i]; i = i + 1; } else { B[k] = A[j]; j = j + 1; }}  }CopyArray(A[], begin, end, B[]){ for(k = begin; k < end; k++) B[k] = A[k];} |

**JAVA - SERIALIZATION**

|  |
| --- |
| Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory. |

|  |
| --- |
| Serializing an Object The ObjectOutputStream class is used to serialize an Object. T he following SerializeDemo program instantiatesan Employee object and serializes it to a file.When the prog ram is done executing , a file named employee.ser is created. T he prog ram does not g enerateany output, but study the code and try to determine what the prog ram is doing .  Note: When serializing an object to a file, the standard convention in Java is to g ive the file a .ser extension. |

|  |
| --- |
| import java.io.\*;  public class SerializeDemo{  public static void main(String [] args){  Employee e = new Employee();  e.name = "ABD";  e.address = "ABD, ABD";  e.SSN = 112255;  e.number = 101;  try{FileOutputStream fileOut =new FileOutputStream("/tmp/employee.ser");  ObjectOutputStream out = new ObjectOutputStream(fileOut);  out.writeObject(e);  out.close();fileOut.close();  System.out.printf("Serialized data is saved in /tmp/employee.ser");}catch(IOException i){i.printStackTrace();}}} |

**Deserializing an Object**

The following DeserializeDemo prog ram deserializes the Employee object created in the SerializeDemoprog ram. Study the prog ram and try to determine its output:

|  |
| --- |
| import java.io.\*;  public class DeserializeDemo{  public static void main(String [] args){  Employee e = null;  try{FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");  ObjectInputStream in = new ObjectInputStream(fileIn);  e = (Employee) in.readObject();  in.close();  fileIn.close();}  catch(IOException i){  i.printStackTrace();  return;}  catch(ClassNotFoundException c)  {System.out.println("Employee class not found");  c.printStackTrace();  return;}  System.out.println("Deserialized Employee...");  System.out.println("Name: " + e.name);  System.out.println("Address: " + e.address);  System.out.println("SSN: " + e.SSN);  System.out.println("Number: " + e.number);}}  This would produce the following result:  Deserialized Employee...Name: ABCDAddress:ABCD, ABCDSSN: 0Number:101Here are following important points to be noted:T he try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject()method. For a JVM to be able to deserialize an object, it must be able to find the |

References:

1. Java Identifiers - <https://www.geeksforgeeks.org/java-identifiers/>
2. Underscores in Numeric Literals Example -<https://www.javatpoint.com/java-numeric-literals-with-underscore>
3. Operators in Java - <https://www.javatpoint.com/operators-in-java>
4. Java Separators - <https://codescracker.com/java/java-separators.htm>
5. Java Variables - <https://www.javatpoint.com/java-variables>
6. Java Argument vs parameter - <https://www.geeksforgeeks.org/argument-vs-parameter-in-java/>
7. Java New Operator - <https://www.geeksforgeeks.org/new-operator-java/>
8. Java Access Modifiers - <https://www.javatpoint.com/access-modifiers>
9. Access modifiers in Java - <https://www.geeksforgeeks.org/access-modifiers-java/>
10. Java Final Keyword - <https://www.javatpoint.com/final-keyword>

