Java Programming

Java is a high level, general purpose programming language that produces software for multiple platforms. It was developed by James Gasling in 1991 and released by Sun Microsystems in 1996 and is currently owned by oracle.

- Java is a programming language as well as a Platform
- It is open source

Java Installation:

- Check if you have java installed: CMD >> java –version
- Download JDK 1.8 from oracle website
  - Java Development Kit
  - Inside JDK, we have JRE (Java Runtime Environment). It gives an environment to execute your code
  - It is platform independent.
- Eclipse: Java Editor— used to write and run the code. (other IDEs: Netbeans, intelliJ)
  - For JDK 32 bit you have to download Eclipse 32 bit and vice versa.

How to get the author Before public class:

```
/**
 * @author Ramsey
 * This class is to understand data types in java
 */
```

General tips

- **Declaration**: Declaration is when you declare a variable with a name, and a variable can be declared only once. Example: int x; String myName; Boolean myCondition;
- **Initialization**: is when we put a value in a variable, this happens while we declare a variable. Example: int x = 7; String myName = "Emi"; Boolean myCondition = false;
- **Assignment**: Assignment is when we already declared or initialized a variable, and we are changing the value. You can change value of the variable as many time you want or you need.
- **Class**: A class can be defined as a collection of objects. It is the **blueprint or template** that **describes the state and behavior of an object.**
- **Local variable**: is a variable which we declare inside a Method. A method will often store its temporary state in local variables.
- **class variable**: **Instance variable** is a variable which is declared inside a Class but outside a Method. They can be both static and non static. If they are common among all methods, you can store it as static. You cannot create a static variable inside only one method.
- Class Name starts with Capital Letter
  - Number at the beginning is not allowed;
- Package name starts with lower letter.
- All keywords in Java start with lower case.
- Duplicate variables are NOT allowed in java.
- 1 byte (Ba'ayt)= 8 bits (Beets)

Primitive Data Type (Java Memory Management)

Type	Size	Range
byte	8	-128..127
short	16	-32,768..32,767
int	32	-2,147,483,648..2,147,483,647
long	64	9,223,372,036,854,775,808..9,223..
float	32	3.4e-0.38..3.4e+0.38
double	64	1.7e-308..1.7e+308
char	16	Complete Unicode Character Set
Boolean	1	True, False

- A few points in Data Type
  - *float f1 = (float) 12.33;*
  - *float f2 = 12.45f;*
  - *char c1 = 'a';*
  - true and false are keywords in java

String Concatenation (Merging)

- Concatenation Operator = +
- Execution is done from left to right.
- int age = 25;
  System.out.println("age of Tom is "+age);
- you can do any arithmetic operation inside the Sysout
  System.out.println(100 - 20);
- First inside parentheses are computed
  System.out.println(x + z + (a + b));
- **Exception**: only for byte (it automatically converts to int)
  byte ba = 100;
  byte bb = 50;
  byte diff = (byte) (ba - bb);
- Concatenate char: each char on the keyboard is assigned a value based on ASCII table. Then, when concatenated, instead they will give value in return.
  char c11 = 'a';
  char c12 = 'b';
  System.out.println(c11); //return a
  System.out.println(c11 + c12); //return 195

Java Operators

- Assignment operator =
- Comparison operator = =
- Not equal to != !
- Short circuit operator &&

Type	Operators
Arithmetic	+, – , *, ? , %
Assignment	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=
Bitwise	^, &, |
Logical	&&, ||
Relational	<, >, <=, >=, == , !=
Shift	<<, >>, >>>
Ternary	? :
Unary	++x, –x, x++, x–, +x, –x, !, ~

Data Type Conversion

- Widening (byte<short<int<long<float<double)
  int i = 10; int ——>long
  long l = i; automatic type conversion
- Narrowing
  double d = 10.02;
  long l =(long) d; explicit type casting

- Numeric Values to String
  String str = String.valueof(value);
- String to Numeric Values
  int intValue= Integer.parseInt(str);
  double doubleValue = Double.parseDouble(str);

  String A = "100k"; >>cannot be converted and it throws: NumberFormatException

Decisive Statements

- If statement:
  ```
  if(condition){
    Statement(s);   }
  ```

- Nested if statement: When there is an if statement inside another if statement
  ```
  if(condition_1) {
    Statement1(s);
    if(condition_2) {
      Statement2(s); }}
  ```

- If-else statement
  ```
  if(condition) {
    Statement(s);}
  else {
    Statement(s); }
  ```

- If-else-if statement:
- Switch statement is used when we have number of options (or choices) and we may need to perform a different task for each choice.
  - Case doesn't always need to have order 1, 2, 3 and so on
  - You can also use characters in switch case
  - The expression given inside switch should result in a constant value otherwise it would not be valid
  - Break statements are used when you want your program-flow to come out of the switch body. **Break only works for loop and switch case NOT if statement.**

  ```
  switch (variable or an integer expression){
    case constant:
    //Java code;
     break;
    case constant:
    //Java code;
    break;
    default:
    //Java code, acts as else condition in if statement;  }
  ```

Iterative Statements: Loop is used to avoid repetitive tasks.

- While loop
  While (condition) {expression}
  ```
  int i = 0; // initialization
  while (i <= 10) { // conditional
  System.out.println(i); // logic
  i = i + 1; // Incremental
  ```

- Do while loop
  ```
  public static void main(String args[]){
    int i=10;
   do{
    System.out.println(i);
    i--;
    }while(i>1);   }
  ```

- For loop—the initialization, condition and increment all are written in one line
  For (condition) {expression}
  ```
  for (int p = 1; p <= 10; p++) { // Starting point, Ending Point, Increment
  System.out.println("Loop " + p);        }
  ```

  - for - if - break

  ```
  for (int number = 0; number <= 10; number++) {
  System.out.println(number);
  if (number == 5) {
  System.out.println("Ali is not here");
  break; }      }
  ```

  - **Enhanced for loop:** Enhanced for loop is useful when you want to iterate Array/Collections, it is easy to write and understand.

  ```
  public static void main(String args[]){
    int arr[]={2,11,45,9};
    for (int num : arr) {
    System.out.println(num); } }
  ```

Incremental & Decremental

```
int a = 1;
int b = a++;
```
post increment: give value of a to b, then increment a by one. returns 2 and 1

```
int p = 1;
int q = ++p;
```
pre increment, returns 2 and 2

```
int m = 2;
int n = m--;
```
post decrement, returns 1 and 2

```
int g = 2;
int h = --g;
```
pre decrement, returns 1 and 1

```
int c = -2;
int d = --c;
```
pre decrement, returns –3 and –3

Arrays in Java

- **Arrays**: stores multiple values in a variable.
- There are two types of Array**: One dimensional Array & Multidimensional Array**

Static Array:

- Two Major Limitations of Static Array:
  - **Static Array**: size is fixed. you have to manually increase the size of array. To solve this issue we use -- ArrayList (dynamic Array)
  - **cannot store different data types**. to solve this, use (ArrayList or Object Static Array)
- Declare and initialize one dimensional Array
  - Syntax for default values: *int[] num = new int[5]; num[0]= 1; ….*
    - "ArrayIndexOutOfBoundsException"
  - Syntax with values given: *int[] num = {1,2,3,4,5};*
- **size of array**: (*num.length*)it is NOT a method.
- **Sort array in ascending order**: Arrays.sort(num); //sort() method is a java.util.Arrays class
- how to print all the values of array: for loop
- Object Static Array : it only allows to store different data types but still static.
  ```
  Object empData[] = new Object[3];
  empData[0] = "Tom";
  empData[1] = 25;
  empData[2] = 'm';
  ```

Dynamic Array (ArrayList) :

- Can store any valued with different data types
- Create the object of ArralyList (it is a class in Java) *ArrayList ar = new ArrayList();*
  - add() to add value :
  - size() size of array: No length
  - remove() removes **on the basis of index** //if removed, the size will automatically adjusted.
  - get() gets the value of index
- How ArrayList work internally?
  - The moment the object is created, virtually it is divided into **10** parts. **(Default Capacity)**
  - **Physical/actual size**: the moment you add values in object. the remaining is **virtual size**.
  - .size() gives you the actual size of the object.
  - The moment the virtual size is 0, memory automatically increase it by 10. this cycle continues.
  - if you want to reach the virtual memory in the print console, it gives you "**IndexOutOfBoundsException**"
- to print all the values from ArrayList: for Loop
- Generics in ArrayList. if not specified, you can store values of different type and it gives you warning. to remover the warning, we insert the data type Class
  ```
  ArrayList<Integer> marksList = new ArrayList<Integer>();
  ArrayList<Double> bmiList = new ArrayList<Double>();
  ArrayList<String> studentNameList = new ArrayList<String>();
  ```
- Difference between Array and ArrayList?
  - Array
    - Array is static
    - Size of the array should be given at the time of array declaration. We cannot change the size of array after creating it
    - Array can contain both primitive data types as well as objects
    - Arrays are multidimensional
  - ArrayList
    - ArrayList is dynamic
    - Size of the array may not be required. It changes the size dynamically. Capacity of ArrayList increases automatically whenever we add elements to an ArrayList
    - ArrayList cannot contain primitive data types. It contains only objects
    - ArrayList is always single dimension

Functions = methods

public void nameOfFunction( ){ }

- The purpose is usability
- return **variable**;
- void **does not** return anything
- psvm is a function
- We cannot create a function inside the function but we can call them
- Duplicate functions are not allowed
- Types of Functions:
  - System defined functions (arraylist.size())
  - User defined functions
    - No input No return
      ```
      public void test(){
      System.out.println("test method");
      }
      ```
    - No input SOME return
      ```
      public ArrayList<String> getPlayersName(){
      System.out.println("getPlayersName");
      ArrayList<String> ar = new ArrayList<String>();
      ar.add("Virat");
      ar.add("Sachin");
      ar.add("Sunil");
      return ar; }
      ```
    - Some input SOME return
      ```
      public String getCountryCapital(String countryName){
      System.out.println("get Country Capital");
      if(countryName.equals("India")){
      return "New Delhi"; //return inside if statement
      }
      else if(countryName.equals("USA")){
      return "Washington DC";
      }
      else if(countryName.equals("Russia")){
      return "Moscow";
      }
      return null;}      returns Nothing—mostly in If statement
      ```

      Another example:
      ```
      public boolean isUSCitizen(String personName){
      System.out.println("isUSCitizen method");
      if(personName.equals("Naveen")){
      return false;
      }
      else if(personName.equals("Elena")){
      return true;
      }
      return false;
      ```

- How to call a function—**create the object of the class : use the new keyword.** The right hand side of = is **object**. obj is only **object reference name**. when created, all the methods created inside the class will be **copied in object**.
  ```
  L06FunctionsConcept obj = new L06FunctionsConcept();
  obj.test();
  ```

Method Overloading

- Method Overloading is a feature that **allows a class** to have more than one method having **the same name**, if their argument lists are different
- Three ways to overload a method—login() is a good example here.
  - Number of Parameters
    ```
    add(int, int)
    add(int, int, int)
    ```
  - Data Type of Parameters
    ```
    add(int, int)
    add(int, float)
    ```
  - Sequence of data type parameters
    ```
    add(int, float)
    add(float, int)
    ```
- If two methods have same name, same parameters and have different return type, then this is not a valid method overloading
  ```
  int add(int, int)
  float add(int, int)
  ```
- Method overloading is an example of compile time polymorphism (many-forms)

Static Block, Methods and Variables

- Static members are common for all the instances (objects) of the class but non-static members are separate for each instance of class.
- Memory is divided into two parts: **Heap & Stack**. When you create the object, it will be created in Heap section. Static methods will be stored in a section in Stack memory section and is called "**Common Memory Allocation**"

Static Block

- Static block is used for initializing the static variables
- A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.
  ```
  class JavaExample{
    static int num;
    static String mystr;
    static{
      num = 97;
      mystr = "Static keyword in Java";  }
    public static void main(String args[])  {
      System.out.println("Value of num: "+num);
      System.out.println("Value of mystr: "+mystr);  }                }
  ```

Static Variables

- A static variable is common to all the instances (or objects) of the class because it is a class level variable
- Static variables are also known as Class Variables.
- Unlike **non-static variables**, such variables can be accessed directly in static and non-static methods.
  ```
  class JavaExample3{
    static int var1;
    static String var2;
  ```

Static Methods

- Static Methods can be access without using object (instance) of the class, however non-static methods and non-static variables can only be accessed using objects.
- How to call **static methods**: (**No need to create the object of the class**)
  ```
  public static void getSchoolName( ){   }
  ```
  - Call by class name *className.getSchoolName();*
  - Call directly *getSchoolName();*
- How to call **non-static methods**
  ```
  className obj = new className();
  obj.getSchoolName();
  ```
- Can you call static methods using object of the class? Yes, but IDE gives a warning bcz the object created occupies unnecessary space.
- **Can we override/overload main method? Explain the reason? Can you override static method?**
  - We cannot override static method, so we cannot override main method.
  - However, you can overload main method in Java. But the program doesn't execute the overloaded

String

In java, a String is an object that represents a sequence of characters. java.lang.String class is used to create String object. String is a collection of multiple characters including spaces.

Creating a String:

- String str1 = "Welcome"; //using literal String
- Str2 = new String ("Welcome") //using new keyword

Methods of Strings:

String str = "I love java but I do not like javascript";

- str.length() //returns the number of characters presents in the string . Starting from 1 not 0. Here 37
- str.charAt(37) //extract i'th character. Here "t"
- str.charAt(38) //StringIndexOutOfBoundsException — if you go beyond str length
- str.indexOf("i") //pass the character and it gives the index number. Here 0
- str.indexOf("j", str.indexOf("j")+1)); //2nd j
- str.indexOf("java") //7 it gives you from where java started
- str.indexOf("Ramsey") //No exception. **Returns -1**
- str.trim() //removes spaces from corners
- str.replace("I", "You") //search and replace
- str.replace(" ", "") //removes the spaces between words
- str.toUpperCase() //returns string in ALL CAPS
- str.toLowerCase() //returns string in ALL LOWERCASE
- str.equals("I love java but I do not like javascript") //comparing two strings
- Str.compareTo("I love java but I do not like javascript") //comparing two strings
- Str.equalsIgnoreCase("i Love java BUT I do not like javascript")
- Str.contains("java") //check for values
- Sub String: it returns the substring of String
  String s2 = "your total amount is 1500 USD";
  System.out.println(str.substring(21, s2.indexOf("USD")-1)); /give beg and end index
  System.out.println(str.substring(0)); /from beginning till end
- Split a string on basis of something. **Converting String to Array**
  String lang = "Java-Python-JavaScript-Ruby"; //it can be also based on space
  String language[] = lang.split("-"); //stores in array.
  System.out.println(language[0]);
- Converting Array to String: Array.toString();
- **Difference between String and StringBuffer?** The most important difference between String and StringBuffer in java is that String object is immutable whereas StringBuffer object is mutable.The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed. By immutable, we mean that the value stored in the String object cannot be changed.

Object Oriented Programming in Java

- Java is an Object Oriented Programming language that produces software for multiple platforms. An object-based application in Java is concerned with declaring classes, creating objects from them and interacting between these objects.
- **OOPS** stands for Object Oriented Programming System. It includes Abstraction, Encapsulation, Inheritance, Polymorphism, interface and..

Java Object

- An object is an instance of a class. Objects have state (variables) and behavior (methods). Example: A dog is an object of Animal class. The dog has its states such as color, name, breed, and behaviors such as barking, eating, wagging her tail.
- Declaring and initializing an object

```
Test t = new Test();
```

Constructor

- Constructor is a block of code that initializes the newly created object. it doesn't have a return type. People often refer constructor as special type of method in Java. Constructor has same name as the class and looks like this in a java code.

```
public MyClass() {
System.out.println("default const");}
```

- Every class has a constructor whether it's a normal class or a abstract class.
- The **new keyword** creates the object of class and invokes the constructor to initialize this newly created object.
- Methods like constructors method can also have name same as class name, but still they have return type, though which we can identify them that they are methods not constructors.
- Constructor overloading is possible but overriding is not possible.
- Constructors can not be inherited.
- Types of constructors:
  - **Default**: If you do not implement any constructor in your class, Java compiler inserts a default constructor into your code on your behalf. You would not find it in your source code (the java file) as it would be inserted into the code during compilation and exists in .class file. If you implement any constructor then you no longer receive a default constructor from Java compiler.
  - **No-arg**: The signature is same as default constructor, however body can have any code unlike default constructor where the body of the constructor is empty.

```
public Demo() {
System.out.println("This is a no argument constructor"); }
```

  - **Parameterized**: Constructor with arguments (or you can say parameters) is known as Parameterized constructor.

```
Employee(int id, String name){
this.empId = id;
this.empName = name; }
```

- Difference between Constructor and Method
  - The purpose of constructor is to initialize the object of a class while the purpose of a method is to perform a task by executing java code.
  - Constructors cannot be abstract, final, static and synchronised while methods can be.
  - Constructors do not have return types while methods do.

Encapsulation

- Encapsulation is also known as "**data Hiding**".
- If a data member is **private** it means it can only be accessed **within the same class**. No outside class can access private data member (class variables and Class methods) of other class.

```
package JavaSessions;
public class Artist {
private String name;

//getter method
public String getName() {return name;                }

//setter method
public void setNmae (String name) {this.name = name;}}

package JavaSessions;
public class Show {
public static void main(String[] args) {
// creating instance of the encapsulated class
Artist s = new Artist();

//setting value in the name member
s.setNmae("BTS");

//getting value of the name member
System.out.println(s.getName());          } }
```

- Advantages of Encapsulation:
  - It improves maintainability and flexibility and re-usability:
  - The fields can be made read-only (If we don't define setter methods in the class) or write-only (If we don't define the getter methods in the class). For e.g. If we have a field (or variable) that we don't want to be changed so we simply define the variable as private and instead of set and get both we just need to define the get method for that variable. Since the set method is not present there is no way an outside class can modify the value of that field.
  - User would not be knowing what is going on behind the scene. They would only be knowing that to update a field call set method and to read a field call get method but what these set and get methods are doing is purely hidden from them.

Inheritance

- The process by which one class acquires the properties (data members) and functionalities (methods) of another class is called **inheritance**
- The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.
- **Child class**: The class that extends the features of another class is known as *child class, sub class or derived class*.
- **Parent Class**: The class whose properties and functionalities are used (inherited) by another class is known as *parent class, super class or Base class*
- Child can inherit from parent not vice versa.
- Sibling to sibling inheritance is not allowed.
- **Types of Inheritance:**
  - **Single Inheritance:** refers to a child and parent class relationship where a class extends the another class

```
class A {//your parent class code}
class B extends A {//your child class code}
```

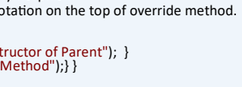
  - **Multilevel inheritance:** refers to a child and parent class relationship where a class extends the child class. For example class C extends class B and class B extends class A.

```
class A {//your parent class code}
class B extends A {//your code}
class C extends B {//your code}
```

  - **Hierarchical inheritance:** refers to a child and parent class relationship where more than one classes extends the same class. For example, classes B, C & D extends the same class A.

```
class A {//your parent class code}
class B extends A {//your child class code}
class C extends A {//your child class code}
```

  - **Hybrid Inheritance**: Combination of more than one types of inheritance in a single program



- The derived class inherits all the members and methods that are declared as public or protected. If the members or methods of super class are declared as private then the derived class cannot use them directly. The private members can be accessed only in its own class. Such private members can only be accessed using public or protected getter and setter methods of super class as shown in the example below.

```
class Teacher {
  private String designation = "Teacher";
  private String collegeName = "Beginnersbook";

  public String getDesignation() {
          return designation; }
  protected void setDesignation(String designation) {
          this.designation = designation; }
  protected String getCollegeName() {
          return collegeName; }
  protected void setCollegeName(String collegeName) {
          this.collegeName = collegeName;}
  void does(){
          System.out.println("Teaching"); }      }

public class JavaExample extends Teacher{
  String mainSubject = "Physics";
  public static void main(String args[]){
          JavaExample obj = new JavaExample();
          /* Note: we are not accessing the data members directly we are using
          public getter method to access the private members of parent class*/
          System.out.println(obj.getCollegeName());
          System.out.println(obj.getDesignation());
          System.out.println(obj.mainSubject);
          obj.does(); }    }
```

- **Up Casting**: Child class object can be referred by parent class reference variable *parent p = new child(); — it is like putting a small box into a larger box.*
  - *Reference Type concept:* in this object created, you can access methods and data members of parent, overridden but NOT child class because of mentioned rule.
- **Down casting**: *BMW b1 = (BMW) new car(); - we never use*
  - It is fine in compile time but it gives you Exception during run time: ClassCastException

Method Overriding

- When we declare the same method in child class which is already present in the parent class, this is called method overriding. In this case when we call the method from child class object, the child class version of the method is called. However we can call the parent class method using super keyword.
- In this case the method in parent class is called overridden method and the method in child class is called overriding method
- The main advantage of method overriding is that the class can give its own specific implementation to a inherited method **without even modifying the parent class code**.
- Method Overriding is an example of runtime polymorphism
- It is a good practice to write "@Override" annotation on the top of override method.

```
class ParentClass{
  //Parent class constructor
  ParentClass(){System.out.println("Constructor of Parent"); }
  void disp(){ System.out.println("Parent Method");} }

class JavaExample extends ParentClass{
  JavaExample(){System.out.println("Constructor of Child");  }
  void disp(){System.out.println("Child Method");  //Calling the disp() method of parent class
          super.disp(); }
  public static void main(String args[]){
          //Creating the object of child class
          JavaExample obj = new JavaExample();
          obj.disp();  } }
```

Output is:

```
Constructor of Parent
Constructor of Child
Child Method
Parent Method
```

- **Overloading vs Overriding in Java**:
  - Overloading happens at **compile-time** while Overriding happens at **runtime**
  - **Performance**: Overloading gives better performance compared to overriding. The reason is that the binding of overridden methods is being done at runtime
  - Overloading is being done in the same class while for overriding base and child classes are required.
  - Argument list should be different while doing method overloading. Argument list should be same in method Overriding.
  - private and final methods can be overloaded but they cannot be overridden. It means a class can have more than one private/final methods of same name but a child class cannot override the private/final methods of their base class.
  - Static binding is being used for overloaded methods and dynamic binding is being used for overridden/overriding methods.

Interface

- Interface looks like a class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body). Also, the variables declared in an interface are public**, static & final by default**
- **Use of Interface in java?**
  - They are used for full abstraction. Since methods in interfaces do not have body, they have to be implemented by the class before you can access them. The class that implements interface must implement all the methods of that interface.
  - In java, multiple inheritance is not allowed, however you can use interface to make use of it as you can implement more than one interface.
- class *implements* interface but an interface *extends* another interface.
- We **can't** instantiate an interface in java. That means we cannot create the object of an interface
- Interface provides full abstraction as none of its methods have body.
- Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.
- Interface cannot be declared as private, protected or transient.
- All the interface methods are by default **abstract and public**.
- Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.
- A **class** can implement any **number of interfaces**.
- A class cannot implement two interfaces that have methods with same name but different return type

```
interface MyInterface{
  /* compiler will treat them as:
   * public abstract void method1();
   * public abstract void method2(); */
  public void method1();
  public void method2(); }
class Demo implements MyInterface {
  /* This class must have to implement both the abstract methods
   * else you will get compilation error*/
  public void method1()  { System.out.println("implementation of method1");}
  public void method2()  {System.out.println("implementation of method2"); }

  public static void main(String arg[]) {
          MyInterface obj = new Demo();
          obj.method1();  }          }
```

What is the difference between interface and a class? Example from your framework?

- Class:
  - Class will contain concrete methods
  - Class is extended
  - We can create an Object of the class
  - Class can inherit only one Class and can implement many interfaces
- Interface
  - Interface will have Interface keyword.
  - Interface will contain only abstract methods
  - We cannot create object of interface
  - Interface needs to be implemented
  - Class can extends many interfaces
  - We need to provide implementation to all methods when we implement interface to the class
- Practical Example: Basic statement we all know in Selenium is

```
WebDriver driver = new FirefoxDriver();
```

WebDriver itself is an Interface. We are initializing Firefox browser using Selenium WebDriver. It means we are creating a reference variable (driver) of the interface (WebDriver) and creating an Object.
Here WebDriver is an Interface and FirefoxDriver is a class.

Abstract

- Abstraction is a process where you show only "relevant" data and "hide" unnecessary details of an object from the user. Example: Mobile Phone.
  - A layman who is using mobile phone doesn't know how it works internally but he can make phone calls.
- A class that is declared using "**abstract**" keyword is known as abstract class. It can have **abstract methods** (methods without body) as well as non abstract methods AKA **concrete** methods (regular methods with body)
- **Abstract Method**: An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon).
- In order to use an abstract method, you need to override that method in sub class.
- An abstract class has no use until unless it is extended by some other class.
- A normal class (non-abstract class) cannot have abstract methods.
- A class which is not abstract is referred as **Concrete class**
- An abstract class can not be **instantiated**, which means you are not allowed to create an **object** of it
- Abstract classes may or may not include abstract methods
- Up casting is allowed
- The class that extends the abstract class, has to implement all the abstract methods of it, else you have to declare that class abstract as well.
- Since abstract class allows concrete methods as well, it does not provide 100% abstraction. You can say that it provides partial abstraction.
- If you declare an **abstract method** in a class then you must declare the class abstract as well. you can't have abstract method in a concrete class. It's vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
- An abstract class must be extended and in a same way abstract method must be overridden. If concrete methods use "final" keyword, they wont be overridden.
  - Static methods can not be overridden
- Why abstract class?
  - we force all the sub classes to implement this method(otherwise you will get compilation error) - set the rules

```
abstract class Animal{ //abstract parent class

  public abstract void sound(); //abstract method
  public void smell();  //concrete method with body }

public class Dog extends Animal{ //Dog class extends Animal class
  public void sound(){ System.out.println("Woof");  }
  public void smell(); { System.out.println("Monkey"); }

public static void main(String args[]){
          Animal obj = new Dog();
          obj.sound();
          obj.smell();
  }          }
```

Difference between Abstract Class and Interface?

- ABSTRACT CLASS
  - To declare Abstract class we have to use abstract keyword
  - In an Abstract class keyword abstract is mandatory to declare a method as an abstract
  - An abstract class contains both abstract methods and concrete methods (method with body)
  - An abstract class provides partial abstraction
  - An abstract class can have public and protected abstract methods
  - An abstract class can have static, final or static final variables with any access modifiers
  - An abstract class can extend one class or one abstract class
  - Abstract class doesn't support multiple inheritance
- INTERFACE
  - To declare Interface we have to use interface keyword
  - In an Interface keyword abstract is optional to declare a method as an abstract. Compiler treats all the methods as abstract by default
  - An interface can have only abstract methods
  - An interface provides fully abstraction
  - An interface can have only public abstract methods
  - An interface can have only public static final variables
  - An interface can extend any number of interfaces
  - Interface supports multiple inheritance

When to use abstract class and interface in Java?

- An abstract class is good if you think you will plan on using inheritance since it provides a common base class implementation to derived classes.
- An abstract class is also good if you want to be able to declare non-public members. In an interface, all methods must be public.
- If you think you will need to add methods in the future, then an abstract class is a better choice. Because if you add new method headings to an interface, then all of the classes that already implement that interface will have to be changed to implement the new methods. That can be quite a hassle.

final Keyword

- **final variable**: Once a variable is declared as final, then the value of the variable could not be changed. It is like a constant. *final int AGE= 12;*
  - It is considered as a good practice to have constant names in UPPER CASE (CAPS).
- **final method**: A final method cannot be overridden. Which means even though a sub class can call the final method of parent class without any issues but it cannot override it.
- **final class**: We cannot extend a final class . No class can extend the final class.
- A few more points:
  - A constructor cannot be declared as final
  - All variables declared in an interface are by default final

final, finally and finalize

- Final
  - It is keyword
  - Used to apply restrictions on class, methods and variables
    - Final class cannot be inherited
    - Final method cannot be overridden
    - Final variable cannot be changed
- Finally
  - Is a block
  - Used to place important code
  - It will be executed whether the exception is handled or not
- Finalize
  - Finalize is a method
  - Used to perform clean up processing just before the object is garbage collected.

Polymorphism

- Polymorphism allows us to perform a task in multiple ways. Let's break the word Polymorphism and see it, 'Poly' means 'Many' and 'Morphos' means 'Shapes'.
- Assume we have four students and we asked them to draw a shape. All the four may draw different shapes like Circle, Triangle, and Rectangle.
- We can perform polymorphism by 'Method Overloading' and 'Method Overriding'
- There are two types of Polymorphism in Java
  - **Compile time polymorphism (Static binding)** – Method overloading
  - **Runtime polymorphism (Dynamic binding)** – Method overriding

Non-Access Modifiers in Java

Java provides a number of non-access modifiers to achieve many other functionalities.

- The ***static*** modifier for creating class, methods and variables.
- The ***final*** modifier for finalizing the implementations of classes, methods, and variables.
- The ***abstract*** modifier for creating abstract classes and methods.
- The ***synchronized*** and *volatile* modifiers, which are used for threads.

Access Modifiers in Java

An access modifier restricts the access of a class, constructor, data member and method in another class

- **default**: The scope of default access modifier is limited to the package only. If we do not mention any access modifier, then it acts like a default access modifier.
- **private**: The scope of private access modifier is only within the classes.
  - Note: Class or Interface cannot be declared as private
- **protected**: The scope of protected access modifier is within a package and also outside the package through inheritance only.
  - Note: Class cannot be declared as protected
- **public**: The scope of public access modifier is everywhere. It has no restrictions. Data members, methods and classes that declared public can be accessed from anywhere.

	Class	Package	Subclass (same package)	Subclass (diff package)	Outside Class
public	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	Yes	No
default	Yes	Yes	Yes	No	No
private	Yes	No	No	No	No

Java Exception Handling Cheat Sheet

- In Java, an exceptions is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime. Exception Handling is a mechanism to handle runtime errors.
- When an exception occurs program execution gets terminated.
- The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.
- System generated message is not user friendly so a user will not be able to understand what went wrong. In order to let them know the reason in simple language, we handle exceptions
- Advantage of Exception handling:
    - Exception handling ensures that the flow of the program doesn't break when an exception occurs.
    - We can identify the problem by using catch declaration

Error vs Exception

- Error
    - Impossible to recover from error
    - Errors are of type unchecked exception
    - All errors are of type java.lang.error
    - They are known to complier.
    - They happen at run time
    - Errors are caused by the environment in which the application is running.
- Exception
    - Possible to recover from Exception
    - Exceptions can be checked type or unchecked type.
    - All exceptions are of type java.lang.exception
    - Checked exceptions are known to complier where as unchecked exceptions are not known to compiler
    - Exceptions are caused by the application.

Types of Exception:

- Checked exceptions:
    - All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not
    - Checked exception (compile time) force you to handle them, if you don't handle them then the program will not compile.
- Unchecked exceptions:
    - Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it's the responsibility of the programmer to handle these exceptions and provide a safe exit
    - unchecked exception (Runtime) doesn't get checked during compilation

Exception Hierarchy

⇒ Object
- ⇒ Throwable
    - ⇒ Exceptions
        - ⇒ RuntimeException
            - ⇒ ArithmeticException
            - ⇒ ClassCastException
            - ⇒ ArrayndexOutOfBoundsException
            - ⇒ NullPointerException
            - ⇒ .....
        - ⇒ NoSuchMethodException
        - ⇒ SQLException
        - ⇒ ClassNotFoundException
        - ⇒ IOException
    - ⇒ Error
        - ⇒ Virtual Machine Error
        - ⇒ Assertion Error
        - ⇒ .....

Different ways to handle Exceptions:

- **Using try/catch**: A risky code is surrounded by try block. If an exception occurs, then it is caught by the catch block which is followed by the try block.
- **By declaring throws keyword**: At the end of the method, we can declare the exception using throws keyword.

Basic Exception

```
try{
 //code that may cause an exception
}
catch (exception(type) e(object))
{
//error handling code
}
//rest of the program
finally {
//statements to be executed
}
```

- **Try block**: Risky code is surrounded by a try block. An exception occurring in the try block is caught by a catch block. Try can be followed either by catch (or) finally (or) both. But any one of the blocks is mandatory.
- **Catch block**: A catch block is where you handle the exceptions, this block must follow the try block.
    - A single try block can have several catch blocks associated with it.
    - You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes
    - you should place the catch blocks in such a way that the generic exception handler catch block is at the last
    - If no exception occurs in try block then the catch blocks are completely ignored.
    - If you are wondering why we need other catch handlers when we have a generic that can handle all. This is because in generic exception handler you can display a message but you are not sure for which type of exception it may trigger so it will display the same message for all the exceptions and user may not be able to understand which exception occurred. Thats the reason you should place it at the end of all the specific exception catch blocks
- **Finally Block**: contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing a connection, stream etc.
    - A finally block must be associated with a try block, you cannot use finally without a try block
    - Finally block is optional, however if you place a finally block then it will always run after the execution of try block.
    - In normal case when there is no exception in try block then the finally block is executed after try block. However if an exception occurs then the catch block is executed before finally block
    - The statements present in the **finally block** execute even if the try block contains control transfer statements like return, break or continue.
    - generally clean up codes are provided here.

```
public static int myMethod() {
try {
return 112;   }
finally {
System.out.println("This is Finally block");
System.out.println("Finally block ran even after return statement");  }
```

Output of above program:

```
This is Finally block
Finally block ran even after return statement
112
```

- Flow of Control in try/catch/finally blocks
    1) If exception occurs in try block's body then control immediately transferred (**skipping rest of the statements in try block**) to the catch block. Once catch block finished execution then finally block and after that rest of the program.
    2) If there is no exception occurred in the code which is present in try block then first, the try block gets executed completely and then control gets transferred to finally block (**skipping catch blocks**).
    3) If a return statement is encountered either in try or catch block. In this case **finally block runs**. Control first jumps to finally and then it returned back to **return statement.**
- **Nested try catch block:** When a try catch block is present in another try block then it is called the nested try catch block. If neither catch block nor parent catch block handles exception then the system generated message would be shown for the exception

```
//Main try block
try {
  statement 1;
  statement 2;
  //try-catch block inside another try block
  try {
    statement 3;
    statement 4;
    //try-catch block inside nested try block
    try {
      statement 5;
      statement 6;
    }
    catch(Exception e2) {
      //Exception Message
    }
  }
  catch(Exception e1) {
    //Exception Message
  }
}
//Catch of Main(parent) try block
catch(Exception e3) {
    //Exception Message
}
```

throws clause

- The "throws" keyword is used to declare exceptions.
- syntax

```
public void myMethod() throws ArithmeticException, NullPointerException
{
  // Statements that might throw an exception
}
```

- What is the need of having throws keyword when you can handle exception using try-catch?
    - The throws does the same thing that try-catch does but there are some cases where you would prefer throws over try-catch. For example:
      Lets say we have a method myMethod() that has statements that can throw either ArithmeticException or NullPointerException, in this case you can use try-catch. But suppose you have several such methods that can cause exceptions, in that case it would be tedious to write these try-catch for each method. The code will become unnecessary long and will be less-readable. One way to overcome this problem is by using throws. declare the exceptions in the method signature using throws and handle the exceptions where you are calling this method by using try-catch.
    - Another advantage of using this approach is that you will be forced to handle the exception when you call this method, all the exceptions that are declared using throws, must be handled where you are calling this method else you will get compilation error.

```
public void myMethod() throws ArithmeticException, NullPointerException
{
  // Statements that might throw an exception
}
public static void main(String args[]) {
  try {
    myMethod();
  }
  catch (ArithmeticException e) {
    // Exception handling statements
  }
  catch (NullPointerException e) {
    // Exception handling statements
  } }
```

throw keyword

- The "throw' keyword is used to throw an exception.
- We can define our own set of conditions or rules and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException when we divide number by 5, or any other numbers, what we need to do is just set the condition and throw any exception using throw keyword.

```
public void a(){
throw new exception_class("error message"); }
```

- Example:

In this program we are checking the Student age.if the student age<12 and weight <40 then our program should return that the student is not eligible for registration.

```
public class ThrowExample {
  static void checkEligibilty(int stuage, int stuweight){
    if(stuage<12 && stuweight<40) {
      throw new ArithmeticException("Student is not eligible for registration");
    }
    else {
      System.out.println("Student Entry is Valid!!");
    }
  }

  public static void main(String args[]){
    System.out.println("Welcome to the Registration process!!");
    checkEligibilty(10, 39);
    System.out.println("Have a nice day..");
  } }
```

Difference between throw and throws in java

1) **Throws clause** is used to declare an exception, which means it works similar to the try-catch block. On the other hand **throw** keyword is used to throw an exception explicitly.
2) If we see syntax wise than **throw** is followed by an instance of Exception class and **throws** is followed by exception class names.
3) Throw keyword is used in the method body to throw an exception, while throws is used in method signature to declare the exceptions that can occur in the statements present in the method.
4) You can throw one exception at a time but you can handle multiple exceptions by declaring them using throws keyword.

Common Scenarios

- **NullPointerException** – When you try to use a reference that points to null.
  String b = null;
  int i = Integer.parseInt(b);
- **ArithmeticException** – When bad data is provided by user.
  int a = 10/0;
- **ArrayIndexOutOfBoundsException** – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.
  int c[]= new int [5];
  c[10]= 50;
- **NumberFormatException**:
  String b = null;
  int i = Integer.parseInt(b);
- **StringIndexOutOfBoundsException**
- **RuntimeException**
- **NoSuchMethodException**
- **NoSuchFieldException**
- **InterruptedException**

Garbage Collection

- Garbage means unreferenced objects
- It is a process to destroy the unused objects.
- Advantages of garbage collection:
    - It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
    - It is **automatically done** by the garbage collector (a part of JVM) so we don't need to make extra efforts.
- How can an object be unreferenced?
    - By nulling the reference

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

    - By assigning a reference to another

```
Employee e1=new Employee();
Employee e2=new Employee();
      e1=e2;
```

    - By anonymous object etc.

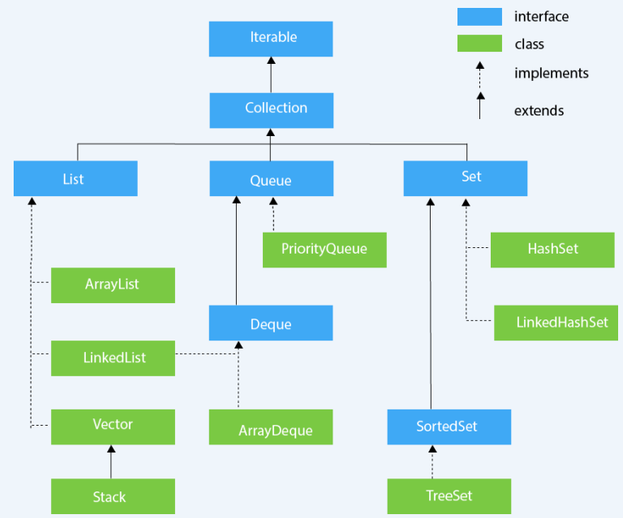
```
new Employee();
```

- The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as

```
protected void finalize(){}
```

- The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
System.gc();
```



Java Programs

How to print “Hello” without main method?

Static block has the highest priority in java. So, any thing that is written in static block is executed first.

```
static{         System.out.println("hello");                }
```

How to find whether given number is odd number?

```
int a=22;

if (a%2!=0) {
System.out.println(a+" is an odd number");
}else {
System.out.println(a+" is not an odd number"); }
```

Finding the max number (given three numbers)

```
int x = 600;
int y = 700;
int z = 300;
if(x>y && x>z) {System.out.println("x is the highest number");   }
else if(y>z){  System.out.println("y is the highest number");      }
else{          System.out.println("z is the highest number");      }
```

How to reverse a String

- Using for loop

```
String str = "I like Java but I hate Selenium";
String reverse = "";
for (int i = str.length() - 1; i >= 0; i--) {
reverse = reverse + str.charAt(i);          }
System.out.println(reverse);                }
```

- Using StringBuffer class

```
String str = "I like Java but I hate Selenium";
StringBuffer sf = new StringBuffer(str);
System.out.println(sf.reverse());
```

How to reverse an Integer

- Using algorithm

```
int num = 12345;
int rev = 0;
while (num != 0) {
rev = rev * 10 + num % 10;
num = num / 10;}
System.out.println(rev);
```

- Using StringBuffer

```
int a = 123456;
String str = String.valueOf(a);
StringBuffer sb = new StringBuffer (str);
System.out.println(sb.reverse());
```

Remove junk/special chars in a String

- Regular Expression: [^a-zA-Z0-9]. Meaning, remove all except those in bracket.

```
String str ="#2. Baidu.com 百度";
String a = str.replaceAll("[^a-zA-Z0-9]", "");
System.out.println(a);
```

How to find min and max in an integer array

```
int[] numbers = { 2, 4, 6, 8, -10, 12 };
int largest = numbers[0];
int smallest = numbers[0];
for (int i = 1; i < numbers.length; i++) {
        if (numbers[i] > largest) {
                largest = numbers[i];
        } else if (numbers[i] < smallest) {
                smallest = numbers[i];
        }                                       }
System.out.println("given array is " + Arrays.toString(numbers));
System.out.println("largest is :: " + largest);
System.out.println("smallest is :: " + smallest);
```

How to swap two integers

```
int x = 5;
int y  = 10;
```

- Using third variable

```
int t;
t=x;
x = y;
y =t;
```

- Using (+ -) operators

```
x = x + y; //frist get the total and give to a variable
y = x-y;  //now deduct from
x = x-y;
```

- Using (*/)operators

```
x = x * y;
y = x/y;
x = x/y;
```

How to swap two Strings

```
String x="Hello";
String y="TekSchool";

x = x+y;
y=x.substring(0, x.length()-y.length());
x= x.substring(y.length());
```

How to Find Duplicates Elements in Java Array?

```
String[] str = { "A", "B", "C", "D", "E", "B" };
```

- First method - good only for small arrays

```
for (int i = 0; i < str.length; i++) {
for (int j = i + 1; j < str.length; j++) {
        if (str[i].equals(str[j])) {
        System.out.println("Duplicate value is:: " + str[i]);}}}
```

- Using HashSet class. It does NOT store duplicate values.

```
HashSet<String> store = new HashSet<String>();
for(String names: str) {
        if(store.add(names) == false) {
        System.out.println("Duplicate Value is:: " + names);}}
```

How to remove all duplicates from an ArrayList?

```
List<String> al = new ArrayList<String>();
        al.add("Ajay");
        al.add("Becky");
        al.add("Chaitanya");
        al.add("Ajay");
        al.add("Rock");
        al.add("Becky");

HashSet<String> hs=new HashSet<String>();
        for (int i=0; i<al.size(); i++) {
        hs.add(al.get(i));}
        System.out.println(hs);}
```

Find 2nd Largest Number in Array using Array

```
import java.util.Arrays;
public class SecondLargestInArrayExample1{

public static int getSecondLargest(int[] a, int total){
Arrays.sort(a);
return a[total-2];
}

public static void main(String args[]){
int a[]={1,2,5,6,3,2};
int b[]={44,66,99,77,33,22,55};
System.out.println("Second Largest: "+getSecondLargest(a,6));
System.out.println("Second Largest: "+getSecondLargest(b,7));
}}
```

Find 2nd Largest Number in Array using Collections

```
import java.util.*;
public class SecondLargestInArrayExample2{

public static int getSecondLargest(Integer[] a, int total){
List<Integer> list=Arrays.asList(a);
Collections.sort(list);
int element=list.get(total-2);
return element;
}
public static void main(String args[]){
Integer a[]={1,2,5,6,3,2};
Integer b[]={44,66,99,77,33,22,55};
System.out.println("Second Largest: "+getSecondLargest(a,6));
System.out.println("Second Largest: "+getSecondLargest(b,7));
}}
```