Access Modifier

---------------------

1.Public - access from anywhere

2.private - with in the class

3.protected - with in base class and subclass

4.default - with in the package

comments

--------------

1. single line

// single line comment

2. multi line

/\*\*

\* multi line comment

\* multi line comment

\*/

**Variable**

Variables - its an identifier , use to store value

---------

1. Static variable

* Class Variable
* Store static data/common data
* Inside the class and outside the method
* We should use class name dot variable name to invoke the static variable
* Final - it's a keyword for constant value
* We cannot create 2 variable with same name, variable name should be unique
* When you create static variable, it will create the space in class area

**public static** String *ORG\_NAME*=**"TCS"**;

2. Instance variable

* Store unique data
* Inside the class and outside the method
* We should create object to invoke instance variable
* Variable name should be unique
* When you create the object it will create the memory space in Heap area
* We can create n number of objects

**public** int *empId*=**10**;

**Private** String *empName*=**”Ajay”**;

***//invoke instance variable***

**Employee obj = new Employee();**

**System.*out*.println(obj.empId);**

**System.*out*.println(obj.empName);**

3. local variable

* If you create a variable inside the function or method that's called local variable
* Local variable scope is only with in the function
* We cannot access the local variable from outside the method
* No need access modifier for local variable

**Function/Method**

1. Static method

* To return the static data
* We can invoke static method using class name
* We should use static keyword for static method

**public static void** getOrgDetails(){

System.***out***.println(**"Static Method"**);

}

// invoke static method

Method.*getOrgDetails*();

1. Instance method

* To return unique data
* No need static keyword for instance method
* We should create object to invoke the instance method

**public void** getEmpDetails(){

System.***out***.println(**"Instance method"**);

}

Method obj = **new** Method();

obj.getEmpDetails();

**Constructor**

* To initialize the variable
* When you create the object, if you want to assign the value to any variable then we can go for constructor
* Two types of constructor
  + Default constructor
  + Parameterized constructor
* We can create n number of constructor
* Constructor name should be same as class name
* There no return type for constructor
* Manually we don't need to call the constructor, when you create the object JVM internally will invoke the corresponding constructor based on the number of constructor
* Constructor is mandatory, if there is no constructor in a class jvm will add the default constructor in the compile time

***// instance variable***

**private int empId ;**

**public Employee(int id){**

**this.empId =id;**

**}**

**public Employee() {**

**}**

**public static void main(String[] args) {**

**Employee emp1= new Employee(1);**

**System.*out*.println(emp1.empId);**

**Employee emp2= new Employee(2);**

**System.*out*.println(emp2.empId);**

**}**

**Primitive Data Types**

- byte (8 bit)

- short (16 bit)

- int (32 bit)

- long (64 bit)

- float (32 bit)

- double (64 bit)

- boolean (1 bit)

- char (16 bit)

public long creditCardNo = 10000l;

Private float val= 10.0f;

**Wrapper Class**

* Byte
* Short
* Integer
* Long
* Float
* Double
* Boolean
* Character

**AutoBoxing and Unboxing**

* Convert primitive data type to **corresponding** wrapper class is autoboxing
* Convert wrapper class to corresponding primitive data type is called unboxing
* Manually we don't need to do the conversion, JVM itself will take care the conversion

**String**

* String is a Class
* String is Immutable, if you modify the string , it will create a new string
* String is not a primitive data type
* We can create String 2 ways
  + String literal
  + Using new operator
* String literal
  + String str = “Hello”
  + When we create string using string literal, it will create the memory space in String pool area
  + If the string is already exist, it will not create a new string, it will return the memory address of the existing string
  + If the string is not exist in the string pool then it will create the new string
* Using new Operator
  + String str = new String(“Hello”)
  + When we create a new string using new operator it will create the space in heap memory
  + Everytime it will create the space in heap memory

**String Comparison**

* We can compare the string using 3 ways
  + ==
  + Using equals() method
  + Using compareTo() method
* ==
  + It will compare the reference of the string

String str1 = “hello”;

String str2 = new String(“hello”);

If (str1 == str2){

System.out.println(“True”);

} else {

System.out.println(“False”);

}

* Using equals() method
  + It will compare the content of the string
  + Return type of equals method is boolean
  + It is case sensitive
  + Use equalsIgnoreCase() to ignore the case

String str1 = “hello”;

String str2 = new String(“hello”);

If (str1.equals(str2)){

System.out.println(“True”);

} else {

System.out.println(“False”);

}

* Using compareTo() method
  + Its similar to equals method, it will compare the content of the string
  + Return type is 0 , + ve, -ve integer
  + Its case sensitive
  + For case insensitive use compareToIgnoreCase() method

String str1 = “hello”;

String str2 = new String(“hello”);

If (str1.compareTo(str2) == 0){

System.out.println(“True”);

} else {

System.out.println(“False”);

}

**Custom Immutable Class**

1) Final class - Make class final so that it cannot be inherited

2) private member variable -> Making member variables private ensures that fields cannot be accessed outside class.

3) final member variable -> Make member variables final so that they can be assigned only once.

4) Constructor -> Initialize all fields in constructor.

5) Don't provide setter methods in class/ provide only getter methods.

6) object of immutable class - Any change made to object of immutable class produces new object.

- Integer, String are immutable class,

any changes made to object of these classes produces new object.

so return reference variable of Integer.

- HashMap is mutable class,

any changes made to HashMap object won't produce new HashMap object.

so return copy/clone of object, not reference variable of HashMap.

Example

----------

/\*\* ImmutableClass

\* 1) Final class - Make class final so that it cannot be inherited

\*/

public final class ImmutableClass{

/\*\*

\* 2) private member variable -> Making fields private ensures that fields

cannot be accessed outside class.

\* 3) final member variable -> Make field final

so that they can be assigned only once.

\*/

private final Integer id; //Immutable member variable

private final String name; //Immutable member variable

private final HashMap<Integer,String> map; //mutable member variable

private final Date date; //mutable member variable

/\*\* 4) Constructor -> Initialize all fields in constructor.

\* assign all mutable member variable using new keyword.

\*/

public ImmutableClass(Integer id, String name, HashMap<Integer,String> map, Date d){

this.id=id;

this.name=name;

//assign all mutable member variable using new keyword.

this.map=new HashMap<Integer, String>(map);

this.date = new Date(d)

}

/\* getter method for id.\*/

public Integer getId() {

/\*\* 5a) Integer is immutable class,

\* any changes made to Integer object produces new Integer object.

\* so return reference variable of Integer.

\*/

return id;

}

/\* getter method for name.\*/

public String getName() {

/\*\* 5b) String is immutable class,

\* any changes made to String object produces new String object.

\* so return reference variable of String.

\*/

return name;

}

/\* Method returns clone of HashMap. \*/

public HashMap<Integer, String> getMap() {

/\*\* 5c) HashMap is mutable class,

\* any changes made to HashMap object won't produce new HashMap object.

\* so return copy/clone of object, not reference variable of HashMap.\*/

return (HashMap<Integer, String>) map.clone();

}

/\*\* 6) Don't provide setter methods in class \*/

}

**Mutable String**

* We can create mutable string using 2 ways
  + StringBuilder
  + StringBuffer
* The difference between StringBuilder and StringBuffer is, StringBuilder is not thread safe and StringBuffer is thread safe
* Both are use to create the mutable string

StringBuilder builder = new StringBuilder();

builder.append(“hi”);

builder.append(“hellow”);

builder.append(“how”);

builder.append(“are”);

builder.append(“you”);

* To convert the StringBuilder to String

builder.toString();

**OOPS**

1. **What's the use of interface?**

* Common functionality but immentaion is different in that case we can go for interface
* Example - Car is an interface and start and stop are common functionality
* If we create any function inside the interface by default abstract, we don't need to use abstract keyword
* Abstract method means only method declaration and no method body
* By default all the variables in interface are public static final
* From JDK 1.7 onwards we can have default method inside the interface
* Default method - we can have method definition
* We should use the default keyword for default method
* We can only one default method inside the interface
* We cannot create constructor in interface
* We cannot create object for interface
* We can create sub interface

**2. What's the use of Abstract class?**

* If we have some common functionality and common implementation then we can go for abstract class
* We should use abstract keyword for abstract methods
* We can create the constructor in abstract class
* We can have instance and static variable inside the abstract class
* We cannot create the object for abstract class
* We can extend only one abstract class in another class

**3. What's the use of encapsulation?**

* Bind the code the data together
* Mainly for enhancement
* Provide the getter and setter methods

**4. What's the use of inheritance?**

* If we want to inherit another class properties or methods then we can go for inheritance
* We are going to reuse the another class properties and methods
* We have base class and subclass in inheritance. We can inherit base class properties or methods into subclass
* We should use extends keyword
* Java supports only single level inheritance
* To avoid ambiguity, java will not support multiple inheritance
* Using subclass object, we can call the base class properties and methods, we don't need to create the object for base class
* Super keyword use to invoke the base class constructor, we can have only one super statement inside the constructor, super statement should be the first statement
* **Method Overloading** : - Inside the class, if we have more than one method with same name but number of argument is different or type of the argument is different
* **Method Overriding**: - same Method name and number of arguments and type of the arguments are same in the base class and sub-class then its called method overriding.
* What’s the use of method overriding?
  + If we want give some specific implementation for the sub class then we can go for method overriding.

**5. What's the use of polymorphism?**

* Polymorphism means more than one form
* In OOP, polymorphism means a type can point to different object at different time.
* In other words, the actual object to which a reference type refers, can be determined at runtime.
  + **Reusability**: the teach() method can be re-used for different kinds of objects as long as they are sub types of the Animal interface.
  + **Flexibility**: the actual object can be determined at runtime which allows the code run more flexibly.
  + **Extensibility**: when we want to add a new kind of Animal, e.g. Snake, we just pass an object of Snake into the teach() method without any modification

public interface Animal {

public void move();

}

public class Dog implements Animal {

public void move() {

System.out.print("Running...");

}

}

public class Bird implements Animal {

public void move() {

System.out.print("Flying...");

}

}

public class Fish implements Animal {

public void move() {

System.out.print("Swimming...");

}

}

public class Trainer {

public void teach(Animal animal) {

animal.move();

}

}

Trainer trainer = new Trainer();

Animal dog = new Dog();

Animal bird = new Bird();

Animal fish = new Fish();

trainer.teach(dog);

trainer.teach(bird);

trainer.teach(fish);

Output:

Running…

Flying…

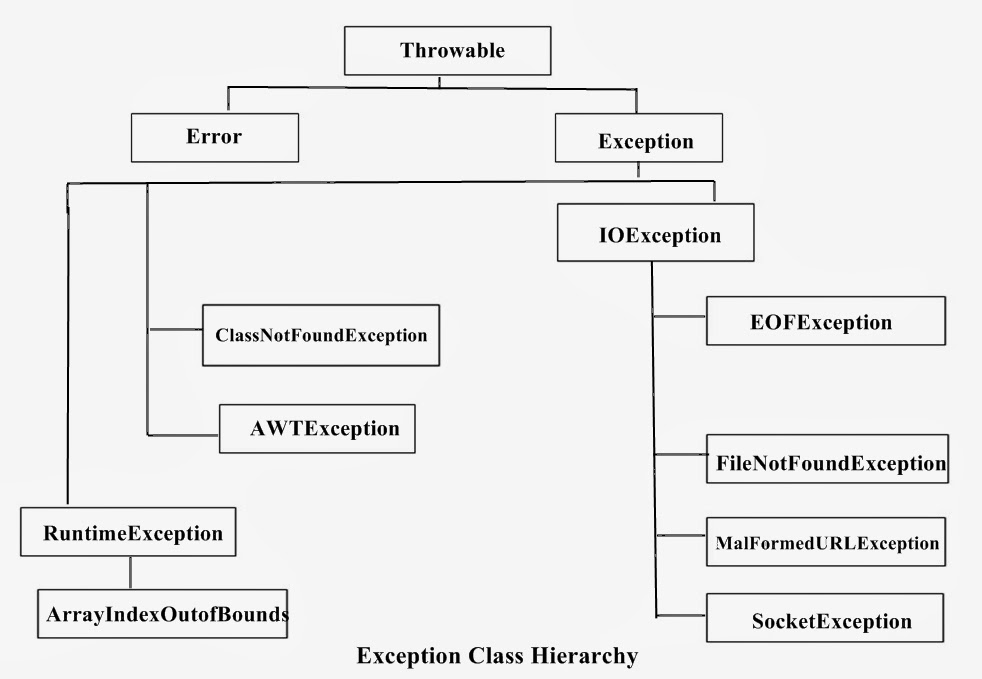
Swimming…

**Exception Handling**

1. Whats Exception?

* Exception is an event, it will disturb the normal execution of the program
* We can handle the exception, if we handle the exception then rest of the code will execute else it will not execute
* There 2 types of exception
  + Checked Exception/Compile time Exception
    - Its user exception
    - It will check at compile time
    - IOException, FileNotFoundException
    - Checked exceptions are sub class of Exception class
  + Unchecked Exception/Runtime Exception
    - Its programmer exception
    - It will check at run time
    - NULLPointerException, ArithmeticException and ArrayIndexOutofBoundException

1. Exception hierarchy?



1. What are the different types of exception?
   1. Checked Exception
   2. Unchecked Exception
2. How to handle the exception?

- we can handle the exception using catch block

- we can have more then one catch block

- From JDK1.7 we can handle all the exception in the same catch block

- if we have multiple catch block then subclass exception should be the first catch

Block

try{

} catch(Exception e) {

e.printStackTrace();

}

5. What is the use of finally block?

* Finally block will always execute in normal execution and if there is any exception
* We can have only one finally block
* Use case - write DB connection close or to close the file

Try{

}catch (Exception e ) {

} finally {

}

6. What’s the use of throw keyword?

* Manually if we want to throw any exception we can use throw keyword
* Mostly we will be using throw keyword to throw the custom exception
* We can throw only one exception at a time

**throw new** ArithmeticException();

7. What’s the use of throws keyword?

* Throw the exception outside of the method
* We can throw n number of exception
* Who ever call the that function they should handle the exception

**public int** divide(**int** input1, **int** input2) **throws** ArithmeticException, NullPointerException{

**int** output = input1/input2;

**return** output;

}

**public static void** main(String[] args){

ExceptionExample obj = **new** ExceptionExample();

**try** {

**int** response = obj.divide(10,0);

System.***out***.println(response);

} **catch** (Exception ex){

ex.printStackTrace();

}

}

7. How to create custom exception?

* If we want to create our own exception then we can go for custom exception
* Create a subclass for Exception class
* Inside the constructor, we should call base class constructor using super() keyword

**public class** CustomException **extends** Exception {

**public** CustomException(){

**super**();

}

**public** CustomException(String msg){

**super**(msg);

}

}

**throw new** CustomException(**"Invalid Input"**);

**Collection**

1. What’s collection?

* It's like a containers
* We can store n number of data
* Array size is static by collection size is dynamic
* Types of collection
  + List
  + Set
  + Queue

1. What’s List?

* List is an interface
* It will allow duplicate data also
* List is not thread safe
* List is an ordered collection
* List is mutable
* Implementation for List
  + ArrayList
  + LinkedList

3.What’s ArrayList?

* ArrayList is implementation for List interface
* ArrayList is similar to Array
* Its index based
* Array size is fixed but ArrayList size is dynamic
* Default size of ArrayList is 16
* Once it reaches 50% it will double the size

List<String> list = new ArrayList<>();

list.add(“Chennai”);

list.add(“Bangalore”);

list.add(“mumbai”);

list.add(“Bangalore”);

list.get(index);

For(String data: list) {

System.out.println(data);

}

list.forEach(data -> { System.out.println(data);

}

4. What is Linked list?

* Linked list is the implementation for the list.
* It is also used to store data and perform some operations
* Syntax :

List<EmployeeDetails> details = new LinkedList<>();

* To perform deletion or insertion ,we can prefer linked list than ArrayList as we need to perform reindexing in Arraylist.
* Linked List is a orderderd collection.It is not thread safe.
* For read operation, we can prefer ArrayList than linked list ,because we can read the element in that particular index directly.whereas in Linkedlist we need to perform iteration from beginning .

4. Difference between iterator and listiterator?

* Iterator use to iterate the elements in the forward direction
* Listiterator use to iterate the elements in the forward or backward direction
* Listiterator we can use only in List