Core Java

Topics to be discussed

Operations on arrays

Constructors

Inheritance

Interface - super type reference getting subclass instances

- dynamic polymorphism

Overriding equals & hashCode

Exception handling

Comparable & Comparator

File Handling

Multithreading - synchronization

javac - compilation

java - execution

javap - it is to disassemble the class files

Constructor:

* It is used to create object
* It initializes the property of the object
* In a class you can overload the constructors, i.e., you can have one or more constructors
* In a class if no constructors are defined then compiler only creates the constructor which is called as default constructor
* In a class if any constructor is defined, then compiler retains the same constructors that are defined

Coding Standards

1. class names: Begin with capital letter and follow camel case, if more than one word then first letter must be in uppercase

ex: HelloWorld, StringBuilder, Scanner, LocalDate, FileWriter, FileOutputStream

1. variables & methods name: Begin with smaller letter and follow came case

ex: storeUser(), toUpperCase(), toLowerCase(), readLine()

1. constructor names: will be same as class names

Arrays:

It is a container to store multiple elements of same type.

int[] items = { 5, 4, 1, 2} ;

String[] items = {“mobile”, “laptop”, “bag”};

int[] items = new int[5]; // 5 memory blocks are allocated to store int, by default each block will have 0 in it

items[0] = 4;

items[1] = 30;

items[2] = 40;

You can use for loop to iterate over the arrays

for(int index = 0; index < items.length; index++) {   
 items[index]  
}

Enhanced for loop

for(int item : items) { // item = items[index]  
  
}

Object array

User[] users = new User[5]; // 5 blocks will be allocated to store 5 user objects, but initially it will have null in each block

users[0] = new User(…..);

users[1] = new User(…..);

String methods: equals(..), concat(..), toUpperCase(), toLowerCase();

Activity:

Using Scanner ask how many User object you want to store in an array, based on the input initialize the array and iterate the array to ask for user details from the scanner, on each iteration create an User object and store in the array, once the iteration completes, you need to again iterate the same array by calling getDetails() on each iteration so that it must print the user details that are stored in the array.

this keyword:

It is used to access instance members, it can differentiate between local variable names & instance variable names, it can also be used to invoke a constructor from another constructor.

Note: a call to this() must be in the very first line of the constructor

class A {   
 A(String a, int b, long c) {   
 // statements  
 }  
 A(int a, int b, int c) {   
 // statements  
 }  
 A(String a, int b, long c, float d) {   
 this(a, b, c); // A(String, int, long)  
 // rest of the statements  
 }  
}

Static members:

When you want to access certain members without creating object you can make them static.

Static variables: These are single copy which can be shared across all the objects of a class, but you can modify the static variable, which will be reflected to all the objects

class A {

static int x = 20;  
}

A.x = 30; // Works

System.out.println(A.x);

Day 2 topics

OOPS concepts

* Encapsulation
* Inheritance
* Polymorphism
* Abstraction

Encapsulation:

Hiding the data and accessing them through public methods (setters & getters)

class User {   
 private String name;  
 int age;  
 String gender;  
 long phone;  
   
   
}

Java beans: These are reusable classes which will have private data & public methods

Inheritance

acquiring the properties & behaviors of an object from another object

extends is the keyword you will use to achieve the inheritance

class A { }   
class B extends A { }

List of members that are inherited to the Employe class

* private members & constructors will not be inherited
* all the setters, getters & public methods will be inherited

Method Overriding

* Keeping the method name & its signature same as of super class but having different logics in the subclass
* You can achieve runtime polymorphism

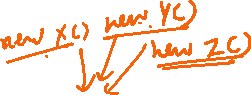
Runtime polymorphism / Dynamic polymorphism

Polymorphism is a method where a single method can have multiple forms i.e., a single method which can have different operations.

Ex: Power buttons have two forms on/off

In Programming language a single method can give multiple results but it is based on method overriding in multiple subclasses whose objects are passed to the super class reference

class X {   
 void demo() { prints X }  
}  
class Y extends X {   
 void demo() { prints Y }   
}  
class Z extends Y {   
 void demo() { prints Z }  
}



X obj;



obj = new X();  
obj.demo(); // prints X



obj = new Y();  
obj.demo(); // prints Y



obj = new Z();  
obj.demo(); // prints Z



Compile time polymorphism

Method overloading is the way to achieve compile time polymorphism

Activity

Create a program that will have classes which is having encapsulation, inheritance and polymorphism

Note: at-least 4 to 5 different types of classes, use has a relationship also if possible

class A {   
 void testA() { }  
}

class B extends A {   
 void testB() { }   
}  
  
A a1 = new B();  
a1.testA(); // works  
a1.testB(); // error

Note: Super class references can’t access sub-class members except the overridden methods  
class A0 {   
 int x = 10;  
}

class A extends A0 {   
 int x = 20;  
}

class B extends A {   
 int x = 30;  
 B(int x) {   
 System.out.println(x); // 40  
 System.out.println(this.x); // 30  
 System.out.println(super.x); // 20  
 }  
}  
B b1 = new B(40);

Storing all the objects in a single variable

An array of super type reference can store the objects of all of its subclasses

User[] users = new User[10];

users[0] = new User(….);  
users[1] = new Student(….);  
users[2] = new Employee(…);

Create a class UserManager that will have 4 methods

1. store(User user): takes any user object like User, Employee or Student and stores in an array of User
2. getUsers(): returns only those users present in the array instead of returning the entire array because some of the index might be having null
3. getEmployeeById(int id): must accept employee id and return the employee matching to the id else return null
4. getStudentByRollNo(int rollNo): must accept student roll no. and return the student matching to the roll no. else return null

Create a Main class that will have a repeating loop until you exit by some input, the loop must ask values at runtime as  
option 1: Store User  
 sub option a: User : Must ask user data like name, age, phone, gender, then store the user object in the UserManager array  
 sub option b: Employee: Must ask employee data like id, name, age, phone, gender, salary, then store the employee object in the UserManager array  
 sub option c: Student: Must ask student data like rollNo, name, age, phone, gender, grade, then store the student object in the UserManager array  
option 2: Show All users : This must call getUsers from the UserManager and print all the user details using getDetails() method  
option 3: Display employee by id: Ask id and print the employee matching to the id  
option 4: Display student by rollNo: Ask rollNo and print the student matching the rollNo  
option -1: Exit the program  
Note: Repeat the loop until you want to exit

Day 3 agenda

* Interface & the implementation
* Loosely coupled program
* Factory design patterns
* Exception handling

Interface:

It is a like a contract between two programs, so that when any changes happen at one end you don’t have to modify at the other end

interface I { 3 methods }

X implements I { should implement 3 methods }

Client code

I i1 = new X();   
X x1 = new X();  
I i1 = FactoryPattern.getInstanceOfImplementation();  
X x1 = FactoryPattern.getInstanceOfImplementation();

Important points on the interface

1. Interface is used to achieve complete abstraction so that you can make your code loosely coupled
2. All the methods of interfaces are by default abstract & public
3. You can have variables inside interface but they are by default public, static & final

i.e., interface I { int x = 10; } >> interface I { public static final int x = 10; }

1. A class can implement one or more interfaces i.e., class A implements I1, I2 { … }
2. You can achieve multiple inheritance in interface by extending one or more interfaces i.e., interface I1 extends I1, I2 { }
3. You can use interface to restrict access of operations based on the user

Factory pattern: it is a design pattern that abstracts object creation from the client

Singleton pattern: it is a design pattern that abstracts object creation from the client, but it creates only one object of the class

How to create a singleton pattern

You must use a counter to track how many times it is executed/incremented and based on that you can reuse the object which is already created

Abstraction: It can be achieved in 2 ways

1. interface
2. abstract class

interface can have only abstract methods however abstract class can have both abstract methods & methods with body and it can have even constructors also.

Note: You can’t create object of either interface or abstract class (though it has constructor it doesn’t allow you to create object, abstract keyword restricts object creation)

Note: Though you can have constructors in abstract class you can’t create object of it, but the use of constructor is to initialize the properties, the constructor will be called by the sub-class constructors using super keyword

How to write abstract classes

abstract class A {   
 void m1() { …. }  
abstract void m2();  
}

Note: abstract methods are implemented by sub-classes in their own way.

Abstract class & Interface

|  |  |
| --- | --- |
| Abstract class | Interface |
| We can have constructors | We can’t have constructors - not even default constructor is created |
| Abstract keyword is mandatory for the class & methods | Abstract keyword is optional for the methods & interface i.e., abstract interface I { } |
| You can’t create object of an abstract class | You can’t create object of an interface |
| You can have an abstract class without any abstract methods | You can have an interface without any abstract methods |
| Variables are not public, static & final by default | Variables are public, static & final |
| Methods are not public & abstract by default | Methods are public & abstract |

What all the things we can write inside the class

1. variables
2. methods
3. constructors
4. nested classes / inner classes
5. initializer blocks - static & instance initializer blocks

Layered architecture: This is to divide the application logic into multiple layers, so that you can create a loosely coupled application & maintain it easily

1. View layer: Presentation logic
2. Controller layer: request & response handling - calls the service layer
3. Service layer: business logic - calls the dao layer
4. DAO layer: Database logic - runs the SQL queries

Employee Case Study

1. Employee: id, name, salary, dob (LocalDate)
2. dao.EmployeeDao : save(Employee), findAll(), findById(int)
3. dao.EmployeeArrayImpl implements EmployeeDao
4. FactoryPattern
5. Main class - View & Controller

Note: create the packages for different layers, atleast two-levels package like com.hsbc, com.org and etc

ex: com.hsbc.beans.Employee

Create a factory pattern method that returns the instance of EmployeeDao implementation, currently you have one implementation(EmployeeArrayDao) but in future you may need to have multiple implementations like EmployeeListDao, EmployeeSetDao and etc, hence the factory pattern method must accept a string argument and return the instance accordingly.

Note: Make these instances singleton.

Day 4 Agenda

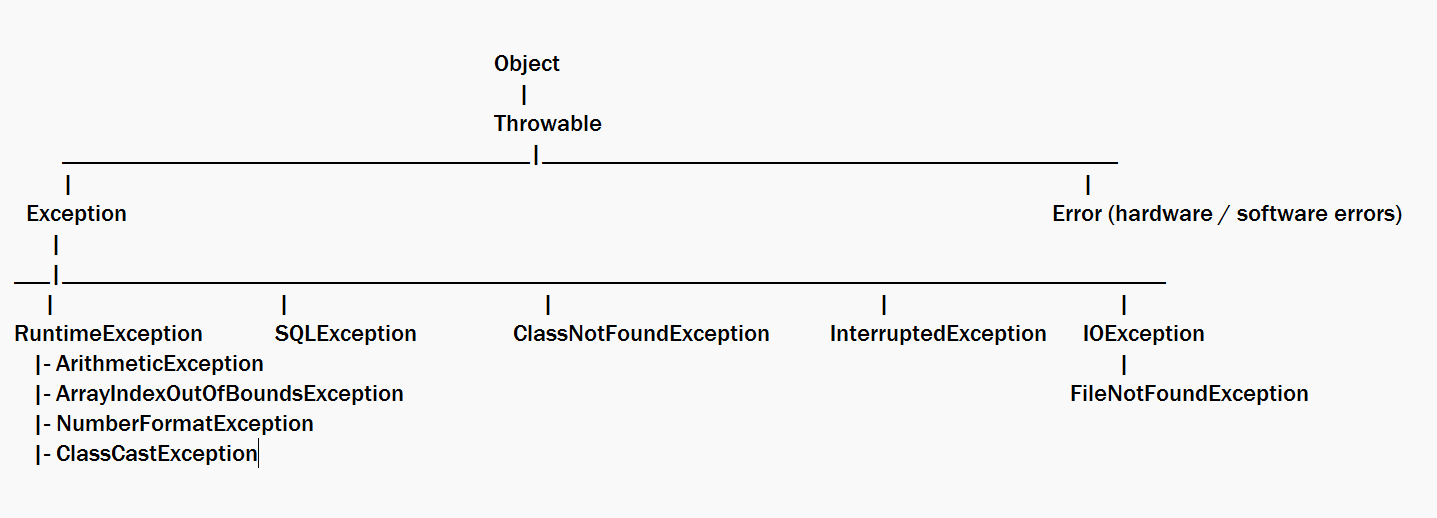
* Exception Handling
* Collection Framework
* Functional interface
* Lambda expression

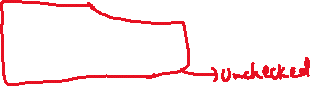
Exception Handling - try, catch, finally, throw & throws

try block - those code which generate exception

catch block - this handles the exception

finally block - it is definitely executed no matter exception is handled or not





How to write multiple catches after the try

you must handle all the sub-class exceptions and then the super-class exceptions, i.e., it must be most specific to most generic

throws: It is to propagate the exceptions to the caller, so that caller should handle the exceptions

public String find(int id) throws SQLException { …. }

Note: If there’s an SQLException in find then it doesn’t handle instead it throws to the caller

public void searchById() {   
 …  
 try { find(102); }  
 catch( SQLException ex) { … }  
 // some more statements   
}

Note: throws can propagate one or more exceptions as well

public int findAge(int id) throws SQLException, IOException { }

Now caller must handle two exceptions i.e., SQLException, IOException

Note: Compiler asks you to handle only checked exceptions and ignores unchecked exceptions if any method throws them

Legal combinations of try-catch-finally

1. try-catch-catch-catch
2. try-catch-catch-catch-finally
3. try-catch-finally
4. try-finally
5. try-catch

Illegal combinations of try-catch-finally

1. catch-finally
2. try-catch-finally-finally

Custom Exceptions / User-defined exceptions

You can create your own exception classes by extending exception classes, you can extend Exception to create checked exception or you can extend RuntimeException to create unchecked exception

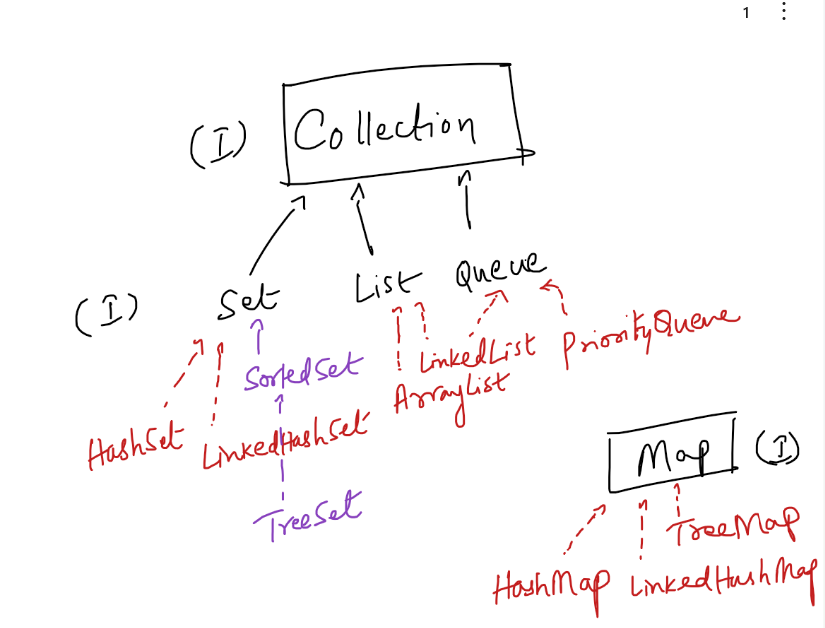
Note: Exception & its sub-class except the RuntimeException will allow you to create checked exceptions

Overriding rules when propagating exceptions

The rule is applied only for checked exception

1. If super class method is not throwing any checked exception, then overridden method shouldn’t throw any checked exception
2. If super class method is throwing any checked exception, then overridden method must retain the same checked exception or its subclass exception or ignore throws

Collection Framework



Method signatures: return type, parameters

Collection<T> methods:

public boolean add(T t): returns true if added else returns false

public int size(): returns size of collection in int

public void clear(): to clear the collection

public Iterator<T> iterator(): to iterate the collection

public boolean remove(Object obj): to remove the element from the collection.

public boolean isEmpty(): to check if the collection is empty, returns true if yes

Collection had 3 sub-types

1. Set: stores only unique elements, if its complex type you must have overridden equals & hashCode to maintain the uniqueness
2. List: stores duplicates, elements are indexed - so that you can add, remove & get by index
3. Queue: stores duplicates, removes elements in FIFO or Sorted order based on the implementation

Set has 3 implementations which maintain the elements in 3 different ways

1. HashSet: Random order
2. TreeSet: Sorted order
3. LinkedHashSet: Sequential order / Insertion order

Note: Set recognizes duplicates of simple values automatically, however for complex types Set recognizes duplicates by calling equals & hashCode, hence these two methods must be overridden.

* Create a Student class with rollNo, name & dob - generate 2 constructors, getters, setters, toString
* Add duplicate Student objects in the HashSet and check whether it is recognizing duplicates or not.

Note: LocalDate.parse(“2002-10-15”) or LocalDate.of(2002, 10, 15) creates LocalDate

Object class methods

public String toString()  
public int hashCode()  
public boolean equals(Object obj)

Note: If Set has to determine duplicates in the complex objects, then the complex objects must have overridden hashCode & equals both

TreeSet: It sorts the simple types automatically like Strings, Integers, Doubles, Floats and so on, however it sort the complex types only if it implements Comparable interface

Comparable interface: It provides sorting logic by comparing the properties, it returns -ve or 0 or +ve while comparting 2 properties to sort, based on these return values TreeSet arranges the object.

Comparable interface is for natural ordering i.e., it will have a default sorting logic, but if you want to sort based on other properties then you can use Comparator<T>

Comparator<T>: It has one abstract method compare(T t1, T t2), it can be used when you want to have multiple sorting logics on a complex object

i.e., sorting based on the name, sorting based on the dob, ascending order, descending order and so on.

Set<Student> set1 = new TreeSet<>(); // it sorts using Comparable

Set<Student> set2 = new TreeSet<>( comparator ); // it sorts using Comparator

Functional interface: It is an interface with only one abstract method, it enables you to directly pass function as a parameter instead of passing object

Note: The function you pass as a parameter is going to be lambda expression

interface A {   
 int demo();   
}  
Lambda expression for A  
A a = () -> 25; // it means demo() returns 25  
a.demo() // returns 25

interface B {   
 String greet(String name);  
}

B b = (n) -> “Welcome “+n;  
b.greet(“Raj”); // returns Welcome Raj

Comparator<T> is also a functional interface it has only one abstract method called compare(T t1, T t2) that returns int

i.e., public int compare(T t1, T t2)

Comparator<Student> comparator = (s1, s2) -> s1.getRollNo() - s2.getRollNo()   
// sorts by roll no

Comparator<Student> comparator2 = (s1, s2) -> s1.getName().compareTo(s2.getName())   
// sorts by name

Set<Student> set1 = new TreeSet<>( comparator2 ); // sorts based on name  
Set<Student> set2 = new TreeSet<>( comparator ); // sorts based on roll no  
Set<Student> set3 = new TreeSet<>(); // sorts by using Comparable of Student