Software Requirements

* JDK 17 or later
* Eclipse IDE or command prompt and notepad

Java: Platform independent and object oriented programming language

Object Oriented programming language has 2 building blocks

1. class: Template of an object
2. object: Instance of a class

class Customer {   
 // properties & behaviors  
 // properties – variables  
 // behaviors - methods  
}  
Customer c1 = new Customer();

JDK: Java Development Kit

JRE: Java Runtime Environment

JVM: Java Virtual Machine

Java Fundamentals: Base for Java programmers

Datatypes

Operators

Conditions

Loops

Arrays

Datatypes: These are set of keywords which are used to create variables that can store some value, there are two types

1. primitive types – size is fixed
2. derived types (combination of primitives) – size varies

Primitive types

|  |  |
| --- | --- |
| Type | Size in bytes |
| byte | 1 (-128 to +127) |
| short | 2 |
| int | 4 |
| long | 8 |
| float | 4 |
| double | 8 |
| char | 2 (‘M’, ‘F’) |
| boolean | 1 (true, false) |

Derived types

1. class
2. arrays

Operators

++, --, <, >, <=, >=, ==, !=, +, -, \*, /

Conditional Statements

1. simple if
2. if & else
3. if, else if, else if …. else
4. switch

if(conditions) {   
  
}  
else if (conditions) {   
  
}  
else {  
  
}

Loops: It is to iterate the statements until some condition is true, there are 3 types of loops

1. for
2. while
3. do while

Activity:

1. Enter 3 digits numbers & print their digit with words  
   ex: 890 must print Eight Nine Zero
2. Enter 3 digits and add the highest digit and lowest digit and display the result  
   ex: 759 must add 9 + 5 and print 14
3. Create an array of some elements and display the maximum, minimum & sum of the array, using only one loop perform all the operations  
   ex: items = {7, 1, -1, 9, 10, 15, 8}, then maximum = 15, minimum = -1, sum = 49

Classes & Objects

Inside class you write variables, methods & constructors

variables: They store data of an object like id, name, salary, phone, email and etc

methods: They will have logics like display(), debit(), credit() and etc

constructors: They will have initialization logics

Command line arguments: It is used to pass input to the program before launching the program

java HelloWorld arg1 arg2 arg3

arg1, arg2, arg3 are stored in args of main method

Day 2 Agenda

Overloading

Static

OOPS concepts

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

Naming Conventions

1. Class Names: Begin with uppercase & follow camel case when there are more than one word ex: HelloWorld, StringBuffer, RuntimeException
2. Variables & Methods: Begin with lower case & follow camel case ex: nextInt(), nextFloat(), charAt()
3. Project Names(Optional): You can begin with lower case and separate by hyphen when there are more than one word

Constructor Overloading:

When multiple constructors are written in the class, it will be constructor overloading

class Person {   
 // name, gender, email & phone  
 Person(String name, String gender) { }  
 Person(String name, String gender, String email) { }   
 Person(String name, String gender, String email, long phone) { }  
 Person(String name, String gender, long phone) { }  
}

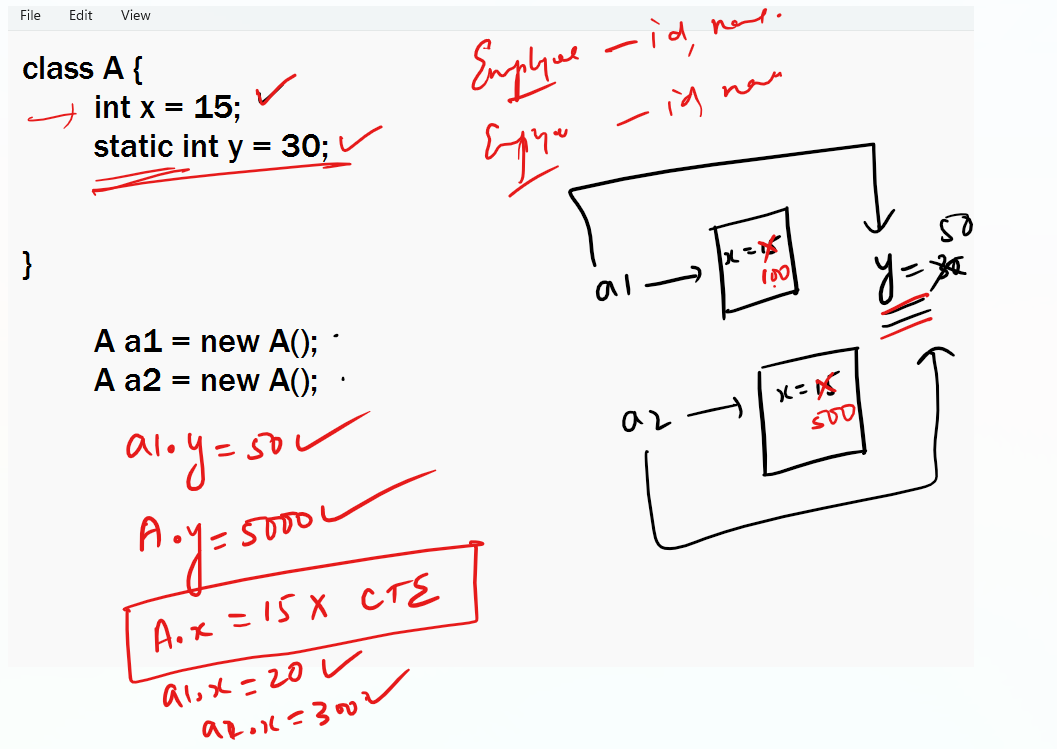
Method Overloading

A class with multiple methods having same name but different signature(parameters, parameters type)

class Calculator {   
 int add(int x, int y) { }   
 float add(float x, float y) { }  
 String add(String x, String y) { }   
}

Static members:

Static variables & static methods can be accessed without creating objects

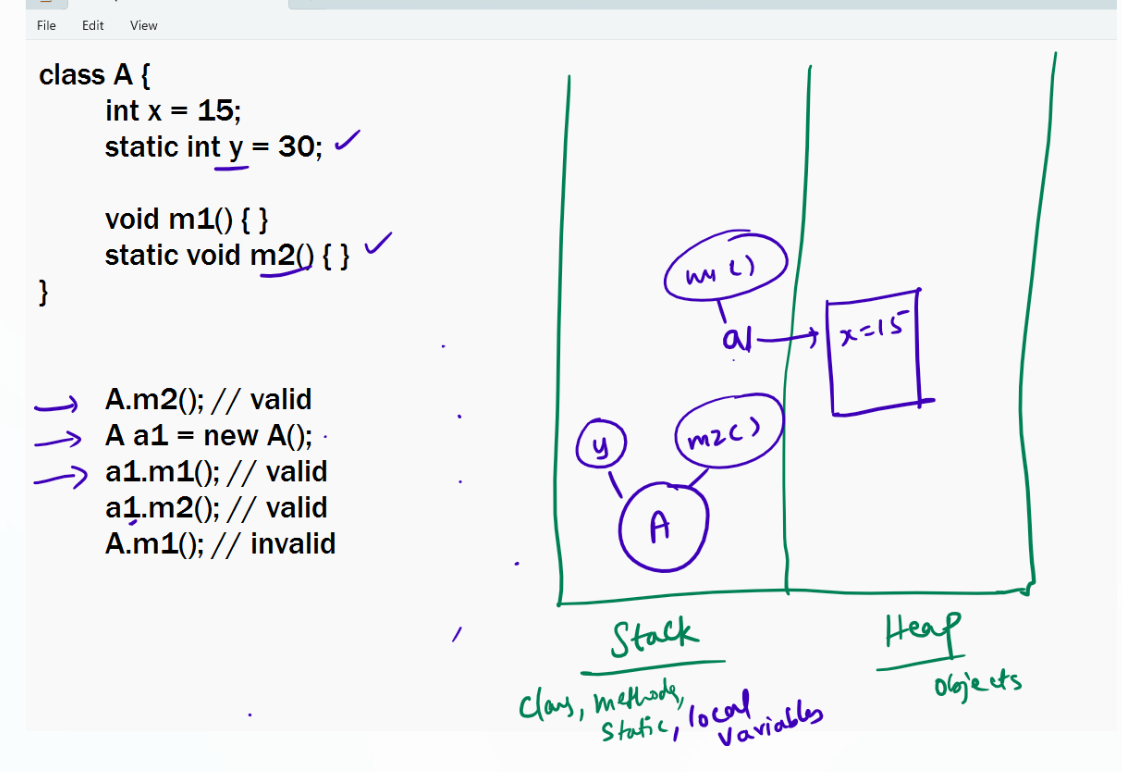


You can access static members using class name or reference variable,

instance variable: every object will have their own copy of instance variable, ex: id, name, salary, phone can be instance variable of Employee

static variable: all the objects share single copy of static variable, ex: company name of Employee

Memory allocation of static & instance members



OOPs principles

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

Encapsulation: Hiding the data(variables) and accessing them only through public methods, so that you will have control over the data, it is achieved by making variables private & methods public

public methods: These are setters & getters which are used to modify the data & read the data

Inheritance: It is used to acquire properties & behaviors of a class from another class so that you don’t have to rewrite properties & behaviors

In Java you use extends keyword to inherit

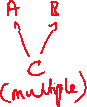
Inheritance types



1. Single level

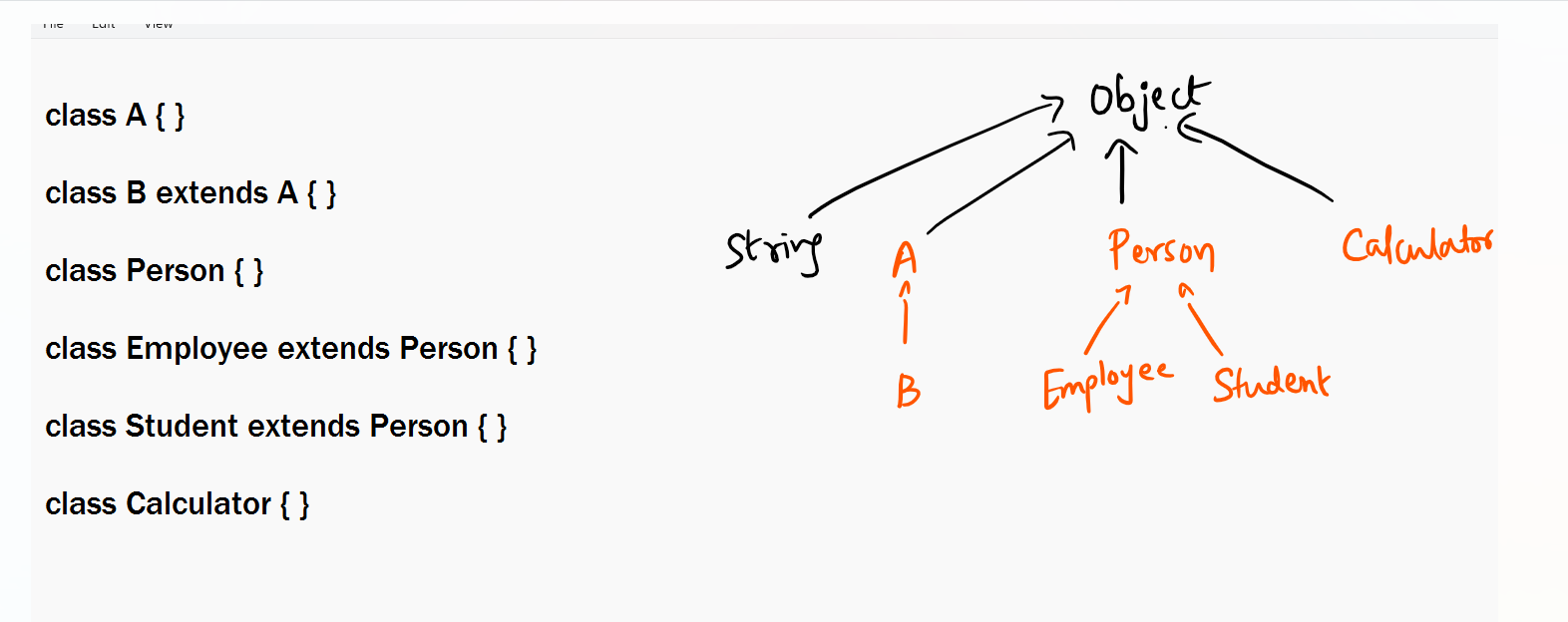


1. Multi level



1. Hierarchical
2. Multiple (No supported in java classes, but supported in java interface)

Object: It is the root class in Java, every class either directly or indirectly inherits Object



Polymorphism: Poly means many, morphism means forms, it allows a method to have many forms

ex: power button acts like on / off

Similarly in OOPs you can create a method with many forms, this can be achieved in 2 ways

1. Method Overloading: Same method name but different signature in the same class
2. Method Overriding: Same method & same signature but different logics in the subclass

Activity:

1. Create Student & Customer class with some properties & provide getters, setters, constructors as per the properties, both of them must inherit Person, override display method in both the classes & print their respective details, in main method create object of Student & Customer and pass them to the test method & observe how the display() method of test method prints student & customer details.

Access specifiers: These are used to specify the visibility for class, variables, methods & constructors, there are 4 access specifiers in Java

1. private: visible within the class
2. public: visible to all
3. protected: visible within the package & outside the package only to the subclass
4. doesn’t have any keyword – its called default scope: visible only within the package

final keyword: It is used on a class, variable and method

final variable: It is constant you can’t modify

ex: final int x = 20; // you can’t modify x

final method: Overriding is restricted

ex: final void display() { } // you can’t override

final class: You can’t make a subclass

Abstraction: Hiding the implementation and showing only the necessary details to the end user, it is achieved using interfaces & abstract classes

interfaces: It provides only method signatures without method body

interface X {   
 void store(); // abstract methods  
 void delete(); // abstract methods   
}

class Imp1 implements X {   
 // they must override all the abstract methods  
}

class Imp2 implements X {   
 // this must override all the abstract methods  
}

Advantages of abstraction

1. Client code doesn’t need to be modified if the implementation changes
2. You can hide the methods at the client side by restricting him to access only few methods instead of accessing all the methods

Interface summary

1. Interface methods are by default abstract, it can have variables which are by default static & final
2. Interface cannot be instantiated, i.e., you can’t create object of interface, however you can create object of the implementation and assign to the interface reference
3. Interface cannot have constructors, not even default constructor
4. By default interface members are public
5. Interface is always used at the client side to create a loosely coupled code (no need to make changes at the client side, if the implementation changes), and also the implementation must be completely abstracted

Tightly coupled code

interface A { void store(); }  
class X implements A { …. }  
class Y implements A { … }  
class Z implements A { … }

Developer1:   
A a = new X(); // implementation is not abstracted  
a.store();  
  
Developer2:  
A a = new X(); // implementation is not abstracted  
a.store();

We must use some design patterns to abstract the implementation i.e., factory design pattern

Factory pattern: It is a design pattern that hides the object creation from the client

class Factory {   
 public static A getInstance() {   
 return new Z();  
 }  
}  
Tightly coupled code

Developer1:

A a = new Z();  
a.store();  
Loosely coupled code

Developer2:  
A a = Factory.getInstance();  
a.store();

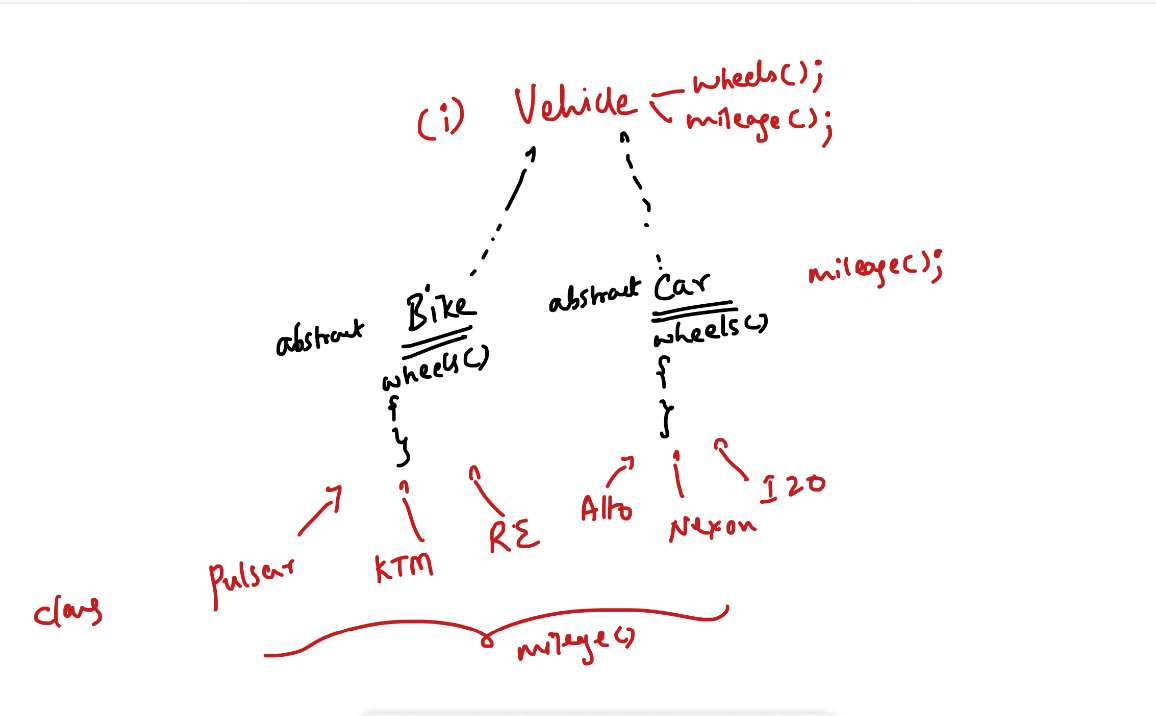
Abstract class:

It is class which can have both abstract methods & methods with body, it can also have constructors to initialize the properties, but you can’t create object of an abstract class

Go for abstract class when you don’t know logic of all the methods, but knows logic of a few methods

Go for interface to create a loosely coupled code at the client side.

abstract class A { // if a class has abstract methods then it must be abstract  
 abstract void m1(); // abstract method, keyword abstract is mandatory in abstract class  
 void m2() { // method with body  
  
 }  
}



Exception Handling

Exceptions are runtime errors which might cause application to stop running if not handled, they occur in many operations like IO operations, DB operations, Remote operations and etc.

In exception handling there are 5 keywords you can use for various use case

1. try { }
2. catch (…) { }
3. finally { }
4. throw
5. throws

try: Keep codes that might cause exceptions like db access, io operations

catch: This is an handler if a try generates an exception catch will handle that exception, you can have more than one catch block after try, because a try can generate one or more exceptions so that if one catch can’t handle another catch will be there to handle

finally: This block is obsoletely executed regardless of exception handled or not, you can write closing resource logics here

throw: This is used when application wants to manually generate an exception/user-defined exception when some conditions are not met

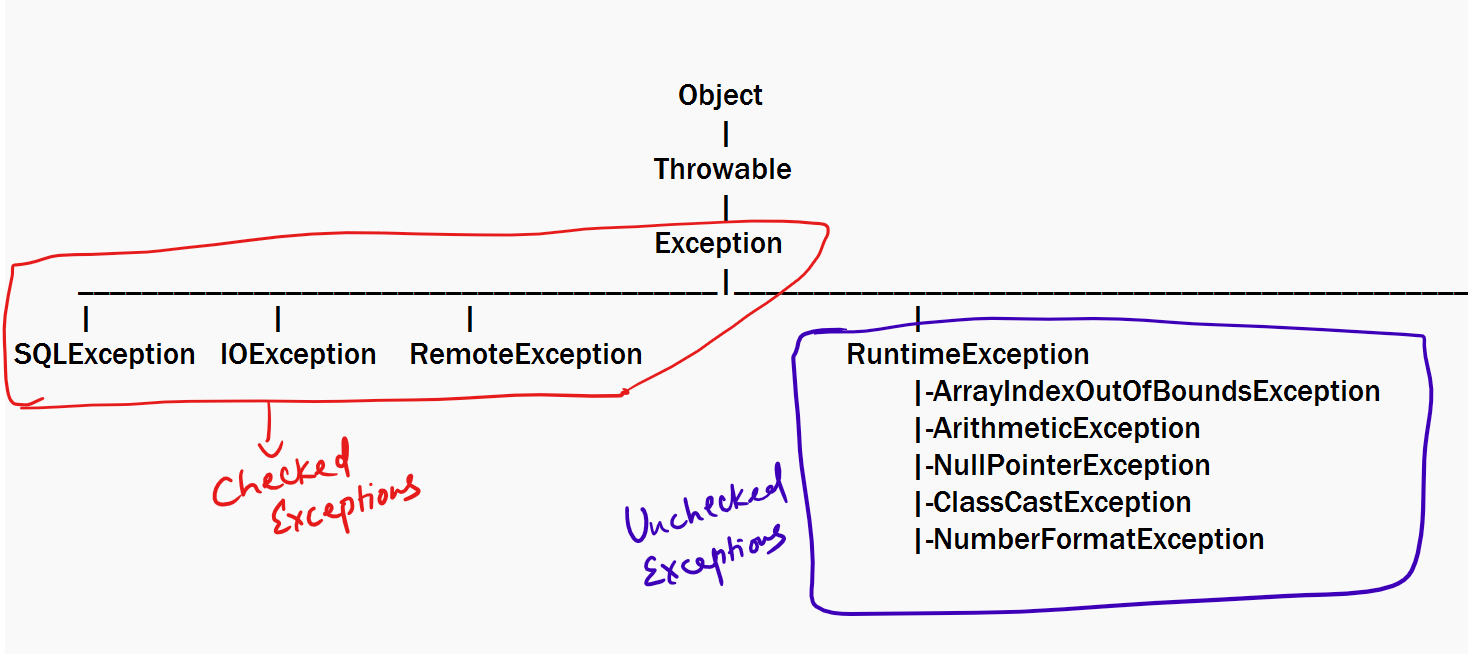
if(condition) {   
 throw new ExceptionName();  
}

throws: This can propagate the exceptions to the caller so that caller can handle it  
public int transfer(double amount) throws ExceptionName {  
 if(condition) throw new ExceptionName();  
}

caller

try {  
 transfer(amount);  
} catch(ExceptionName) { …. }

Exception Hierarchy



Checked Exceptions: These are critical exceptions that must be handled at the compilation time itself, else you will get compilation errors, these will not be in programmers control because they use resources outside the application

Unchecked Exceptions: These are not critical exceptions hence compilers don’t force you to handle, because they can be controlled in the program itself

Custom Exception/User defined Exception: These are the exceptions which you can create by extending any one of the Exception classes, it is necessary when business needs our own exception classes

class InsufficientBalanceException1 extends Exception {   
 // This is a checked exception – recommended because compiler can detect them  
}

class InsufficientBalanceException2 extends RuntimeException {   
 // This is an unchecked exception – compiler ignores this  
}

throw new InsufficientBalanceException1(); // compiler forces you to handle

throw new InsufficientBalanceException2(); // compiler ignores to notify you to handle

Handling exceptions with super types

You can use Exception in the catch to handle all the exceptions, then you can use RuntimeException in the catch to handle all the unchecked exceptions

try {   
 AIOBE  
} catch(ArrayIndexOutOfBoundsException e) { }   
catch(RuntimeException e) { }   
catch(Exception e) { }

Activity

1. Create a Product class with name, price and quantity, make all the properties private, and generate setters, getters & constructor to initialize the name, price & quantity
2. Create Main class that will have main method and invoice method, invoice method takes Product array as an argument

i.e., public static void invoice(Product[] products) { .. }

invoice method must iterate all the products in the array and calculate the total price by calculating price of each product with quantity and add 18% GST to the total price and print the total price of all the products purchased

main method must ask user an input of how many products to add in the Product[] array, then it must ask each product name, price and quantity as an input as per the size of the Product[] array, each product must be added into the Product[], once all the product is added pass that Product[] to the invoice which prints the total price

Ex:   
Product1 : name: Breads, price : 20, quantity: 10  
Product2: name: Chocolate: price:50, quantity: 5

Total Price = (20 \* 10) + (50 \* 5) + 18% tax of total price

Hint:

class Product {   
 // 3 properties : name, price and quantity  
 // constructor with 3 arguments   
// setters & getters  
   
}  
class TestProduct {   
 main() {   
 // create scanner object  
 // take size of an array to initialize like 5   
 // create product array: Product[] products = new Product[size];  
 // iterate Product[] to store product object in each index  
 for(int index = 0; index < products.length; index++) {   
 // take input for name, price & quantity  
 products[index] = new Product(name, price, quantity);  
 }  
 // call invoice by passing Product[] i.e., invoice(products);  
 }   
 public static void invoice(Product[] products) {   
 // create a total variable: double total = 0;  
 // iterate the products to get Product object from each index  
 for(…) {   
 total = (products[index].getPrice() \* products[index].getQuantity()) + total;  
 }  
 // apply 18% GST at the end, i.e., after for  
 total = total + (total \* 0.18);  
 //print total  
 }  
}

Object class: It is a root class, it has various methods in it that will be helpful in the subclass

1. public String toString(): returns the object information in string form
2. public boolean equals(Object obj): compares two objects and returns true/false – its mainly used in Datastructures / Collection Framework
3. public int hashCode(): It returns the object hash code – its mainly used in Datastructures / Collection Framework

toString(): Whenever you print object automatically toString() is called to represent the object in string format

String classes

String creates an immutable object, where the content can’t be modified once created

String s1 = “hello”;  
String s2 = s1.concat(“123”); // hello123, but s1 content wouldn’t be modified

char c = s1.charAt(0); // returns ‘h’;

String s3 = “123;Raj;35000”;  
String[] s4 = s3.split(“;”); // s4 = {123, Raj, 35000}

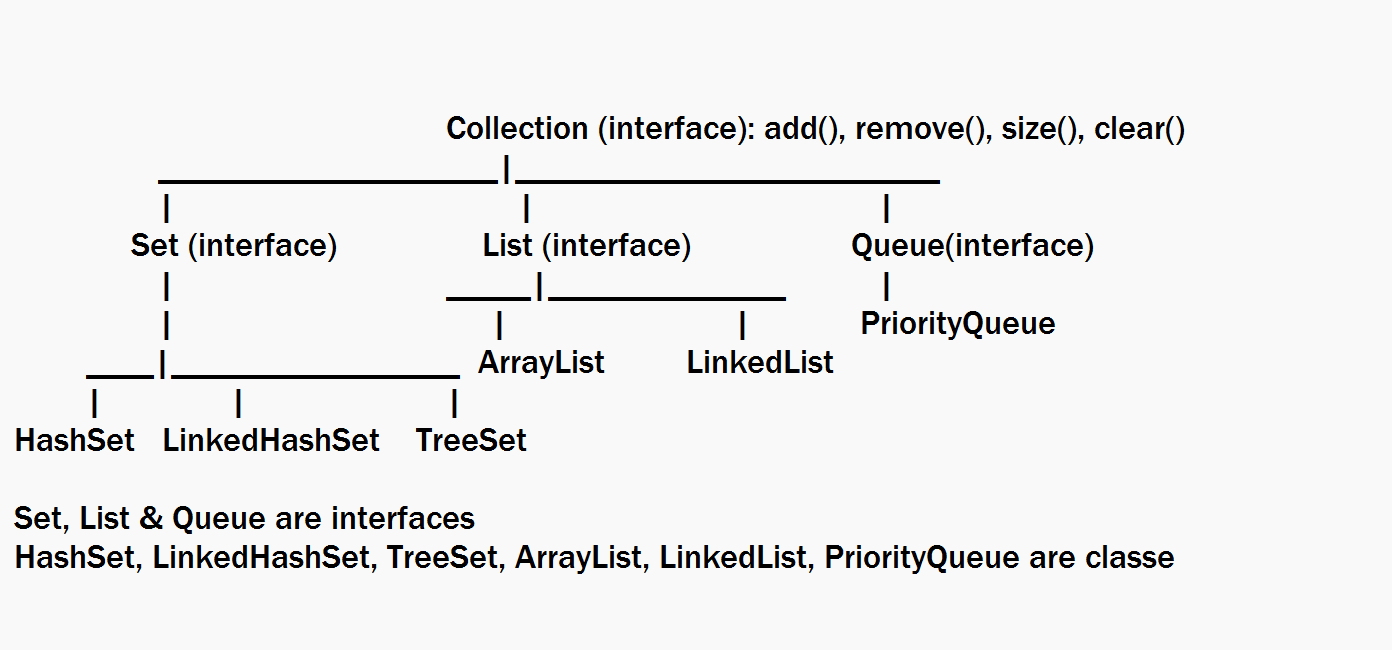
String Buffer: It creates mutable object, where the content can be modified

StringBuffer sb1 = new StringBuffer(“abc”);  
sb1.append(“123”); // s1 = abc123

Datastructures / Collection Framework

It is used to maintain the data in various formats like index based, sorted order, random order, unique data

In Java we have inbuilt classes to maintain the data, all these inbuilt classes have common methods derived from an interface Collection.





Set: It stores only unique elements   
HashSet: Doesn’t maintain the order, but searches the element faster  
LinkedHashSet: Stores the elements in insertion order  
TreeSet: Stores the elements in sorted order

List: It maintains the elements in insertion order & elements are indexed, it allows duplicate, since it is indexed you can add, remove, retrieve based on index

ArrayList: It maintains the elements in contiguous memory address, its retrieval is faster, whereas adding & removing the elements are slower

LinkedList: It maintains the elements in non-contiguous memory address with links, its retrieval is slower but adding & removing the elements are faster

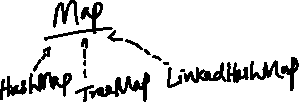
Note: LinkedList implements Queue as well, it can also remove elements in FIFO order

Queue: It is mainly used to process the elements after removing

PriorityQueue: It removes the elements in sorted order

All these APIs have common methods: add(), remove(), clear(), iterator(), size(), clear()

Map: It is used to maintain the elements in key & value pairs, it is also an interface, but its not part of Collection



Map is similar to set, it doesn’t allow duplicates(i.e., key will be unique), Map has methods like  
put(key, value): to add an element  
get(key): to retrieve an element using a key  
remove(key): to remove an element using a key

HashMap: Doesn’t maintain the order, but retrieval is faster  
TreeMap: Maintains the element in sorted order  
LinkedHashMap: Maintains the element in insertion order

ex:   
Map map = new HashMap();   
map.put(“username”, “system”);  
map.put(“password”, “root@123”);  
Note: Collection & Map related APIs are present in java.util package, we must import this

equals & hashCode:

Whenever you add complex types you must override equals & hashCode, because Set internally calls equals & hashCode while storing the object, these two methods help Set to search the element and also to identify the duplicates

hashCode: Helps in searching the element

equals: Helps in identifying the duplicates

Sorting mechanism

By default simple types are automatically sorted, however complex types are not sorted by default.

There are 2 interfaces you can use to implement sorting technique

1. Comparable: This is implemented within the class which provides natural sorting
2. Comparator: This is implemented outside the class which is to customize the natural sorting, this can sort multiple properties

Comparable & Comparator are interfaces it has only one abstract method that return int.

Comparable: public int compareTo(Object obj)

Comparator: public int compare(Object obj1, Object obj2)

compareTo or compare must return an int value, which can be either 0, -ve or +ve, based on these int value element can be sorted internally

0: pivot element

+ve: > pivot element

-ve: < pivot element

Internally these numbers are used to arrange the elements

Limitations of Comparable: you can sort only one property or only one way, to over come this drawback we ban use Comparator

Comparator: It can sort in multiple ways, i.e., id in asc/desc, then name in asc/desc, then dob in asc/desc

We can implement Comparator within the method using anonymous class instead of creating separate classes for each sorting technique

class A implements Comparator { this can sort id in desc }  
class B implements Comparator { this can sort id in asc }   
class C implements Comparator { this can sort name in asc }

Instead of creating separate classes for multiple sorting technique we can implement Comparator without creating class i.e., using anonymous class.

Anonymous class

// descending order id  
Comparator<User> descId = new Comparator<User>() {   
 public int compare(User x, User y) { return Integer.compare(y.getId(), x.getId()); }  
}  
//ascending order id  
Comparator<User> ascId = new Comparator<User>() {   
 public int compare(User x, User y) { return Integer.compare(x.getId(), y.getId()); }  
}  
// ascending order name : String has compareTo method internally to return int for 2 strings

Comparator<User> ascName = new Comparator<User>() {   
 public int compare(User x, User y) { return x.getName().compareTo(y.getName()); }  
}

// ascending order dob: LocalDate has compareTo method to compare 2 dates

Comparator<User> ascDob = new Comparator<User>() {   
 public int compare(User x, User y) { return x.getDob().compareTo(y.getDob()); }  
}

You need to pass this reference to the TreeSet as below  
Set<User> users = new TreeSet( ascDob ); // compares using Comparator  
Set<User> users = new TreeSet(); // compares using Comparable

Future topics

1. Spring Boot
2. Spring Microservices
3. Docker & Kubernetes