* Exception Handling
* Multithreading
* Collections
* Java 8
* Spring Core
* Spring MVC
* Spring Boot
* Spring Microservices
* Angular

Pre-requisites

* create git-hub account
* install git from : <https://git-scm.com/downloads>
* Eclipse IDE
* Java 8

Why Java?

Java is a platform independent & Object oriented language, using which you can develop any kind applications like standalone, distributed, mobile applications, you can write programs to IC’s, Cards and etc

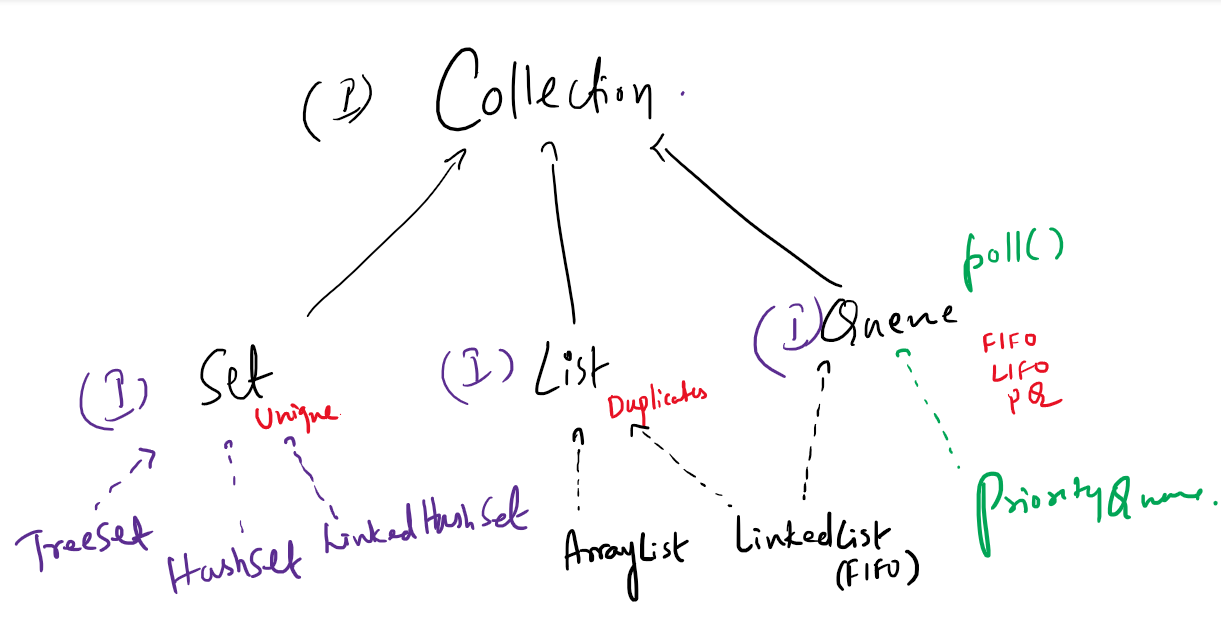
OOPs concepts:

1. Encapsulation: private variables & public setters & getters
2. Inheritance: extends keyword
3. Polymorphism: overriding and overloading
4. Abstraction: abstract class & interface

Exception Handling:

try, catch, finally, throw & throws

Collection Framework:



Collection has methods:

add, remove, clear, size, isEmpty, iterator,...

Set: Allows only unique elements

TreeSet: Maintains the elements in sorted order

HashSet: Maintains the elements in random order, but retrieval is faster

LinkedHashSet: Maintains the elements in insertion order

List: Allows duplicates and elements have index

ArrayList: Maintains elements in contiguous memory address, retrieval is faster but adding/removing of the elements are little slower

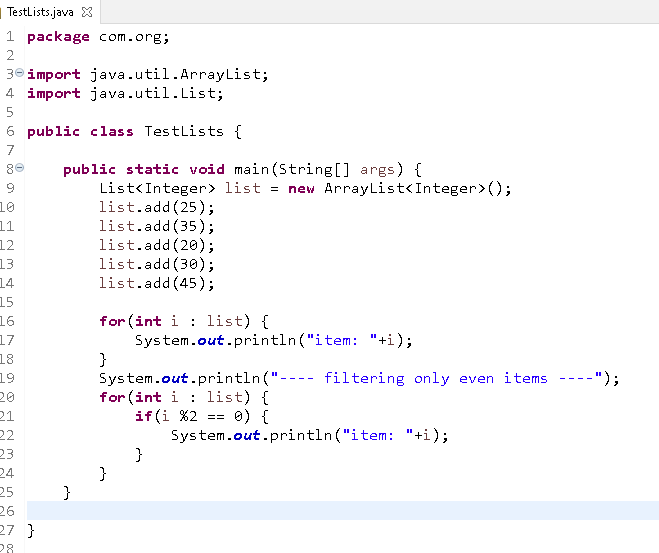
LinkedList: Maintains elements in non-contiguous memory address, retrieval is little slower but adding/removing of the elements are faster

Some useful git commands:

git clone <<url>>  
git pull  
git push -u origin master  
git add .  
git commit -m ‘message’  
git status

Collection:

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



Customer, Employee, Student and so on....

Exercise:

Create a user class which will have id, name and password, generate setters & getters

Add 5 user objects inside List<User> with different id, name & password

Iterate the List<User> such that it will only display the names whose length is more than 5

Object class: toString(), equals() & hashCode():

Set internally calls equals() & hashCode() of Object class, but you can override them in your userdefined class to identify the duplicates

**package** com.org;

**public** **class** User {

**private** **int** id;

**private** String name;

**private** String password;

**public** User(**int** id, String name, String password) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.password = password;

}

**public** User() {

**super**();

// **TODO** Auto-generated constructor stub

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getPassword() {

**return** password;

}

**public** **void** setPassword(String password) {

**this**.password = password;

}

@Override

**public** String toString() {

**return** "User [id=" + id + ", name=" + name + ", password=" + password + "]";

}

@Override

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + id;

**return** result;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

User other = (User) obj;

**if** (id != other.id)

**return** **false**;

**return** **true**;

}

}

TestSet.java

**package** com.org;

**import** java.util.HashSet;

**import** java.util.Set;

**public** **class** TestSets {

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

Set<User> set = **new** HashSet<User>();

set.add(**new** User(100, "Alex", "alex123"));

set.add(**new** User(103, "Alex", "alex123"));

set.add(**new** User(100, "Alex", "alex123"));

set.add(**new** User(100, "Alex", "alex123"));

System.***out***.println("Size: "+set.size());

}

}

Output:

Size: 2

Map: It maintains elements in key value pairs, it uses put() method to store & get() method to retrieve, here key should be unique.

Map is an interface and it has 4 implementations

1. HashMap: Random order, it is a newer class and supports null, not thread safe
2. Hashtable: Random order, it is an older class and doesn’t support null, it is threadsafe
3. TreeMap: Sorted order
4. LinkedHashMap: Insertion order

Exercise:

Create a map that stores multiple values for a single key

|  |  |
| --- | --- |
| Key(String) | Value (List<Student> |
| CS | Student1, Student2, .. |
| EC | Student3, Student4,.... |

And display the list of students based on the key

i.e, EC -> Student3, Student4,...

CS -> Student1, Student2,...

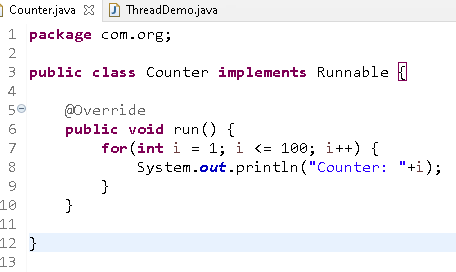
Multithreading

It makes a single program to perform multiple tasks concurrently.

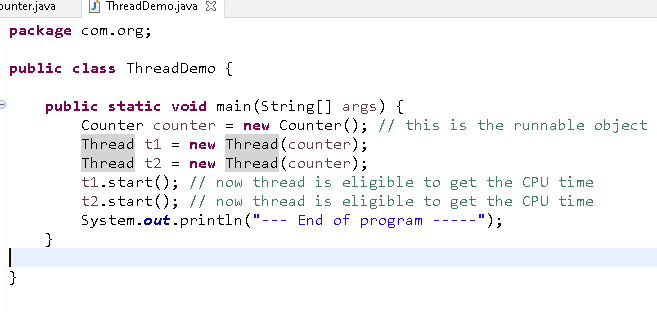
In java we use two API’s to achieve multithreading

1. Thread class: It creates thread and maintains threads
2. Runnable: it has run() method executed by threads

Counter.java



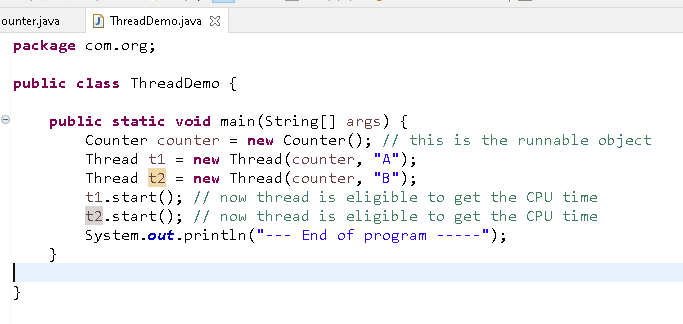
ThreadDemo.java



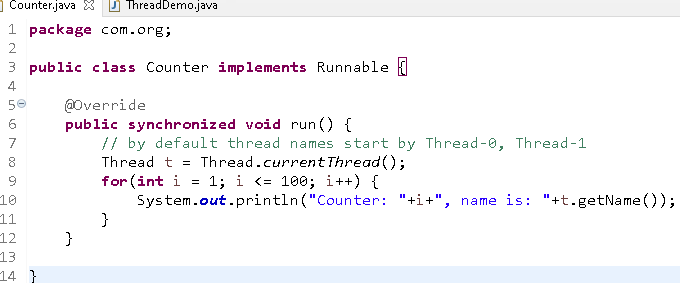
Thread Synchronization:

It is a process of locking the threads to enter inside a method that is synchronized so that only one thread can enter and perform the task.

ThreadDemo.java



Counter.java

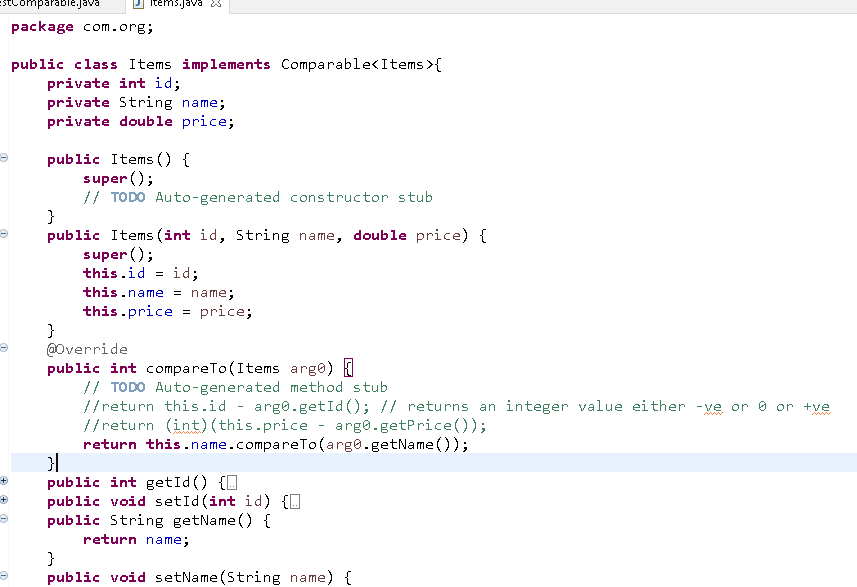


Comparable & Comparator interface

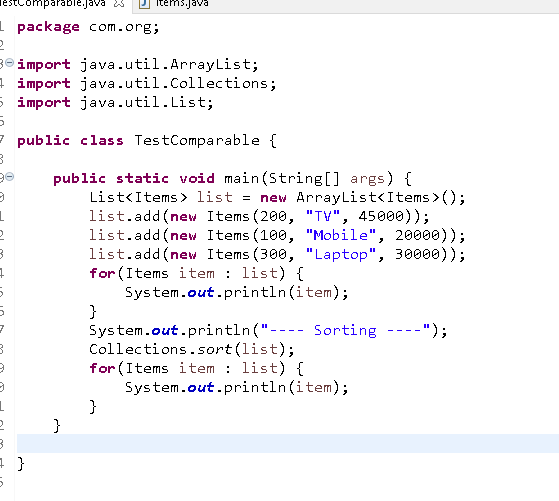
It is used to compare objects to perform sorting, by default many primitives have default sorting, however if you have user defined objects then they must either use Comparable or Comparator

Comparable provides you the natural sorting or default sorting, but Comparator gives you custom sorting, i.e., multiple way of sorting

Items.java



TestComparable.java



Comparable gives natural ordering and you can sort only in one way, however if you need multiple way of sorting you must use Comparator

Comparable is implemented by many classes like Integer, Double, LocalDate, String, and so on, they give default sorting.

Comparator: It gives us a provision where you can implement it to provide many sorting operations, because Comparator is implemented in different class and its instance is passed to the sort method, However Comparable is implemented in the same class.

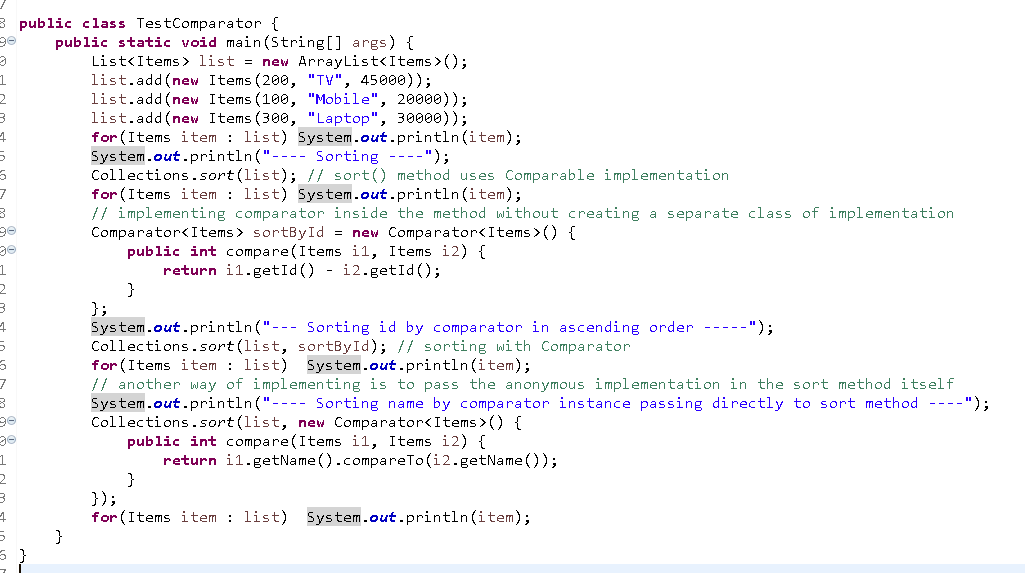
Comparator has compare() method that takes 2 arguments

Comparable example:

class Employee implements Comparable<Employee> { }   
class Item implements Comparable<Item> { }

Comparator example:

class Sort1 implements Comparator<Employee>{ }   
class Sort2 implements Comparator<Employee> {}



Exercise: Using Comparator with anonymous implementation and sort other properties of Items in both ascending and descending order

Java 8 Features:

Functional Interface: Which will have only one methods to enable functional programming, functional programming allows you to directly pass the function instead of object as a parameter.

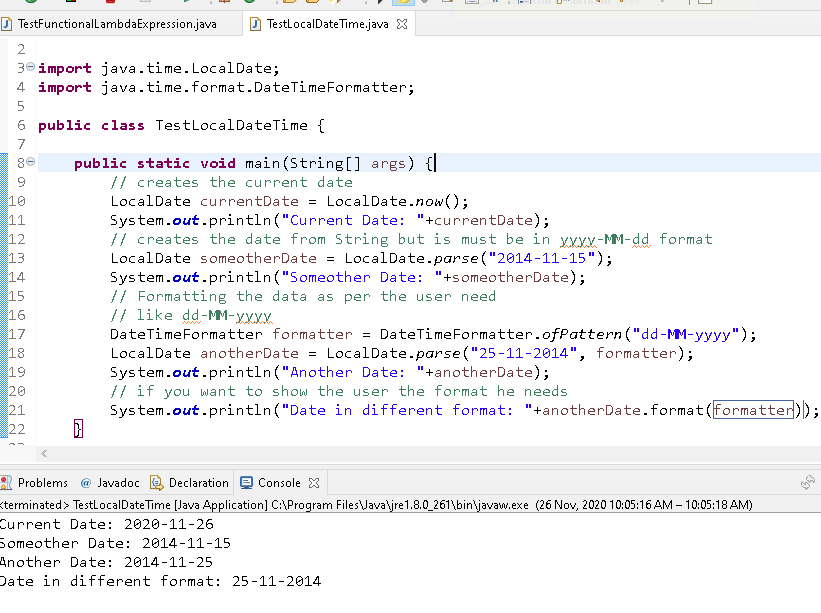
Lambda Expression: It is a simplified form anonymous class, which simplifies implementation of functional interface in functional programming style, i.e., without using classes & objects

New Date & Time Api’s: LocalDate, LocalTime, LocalDateTime, Period, Duration

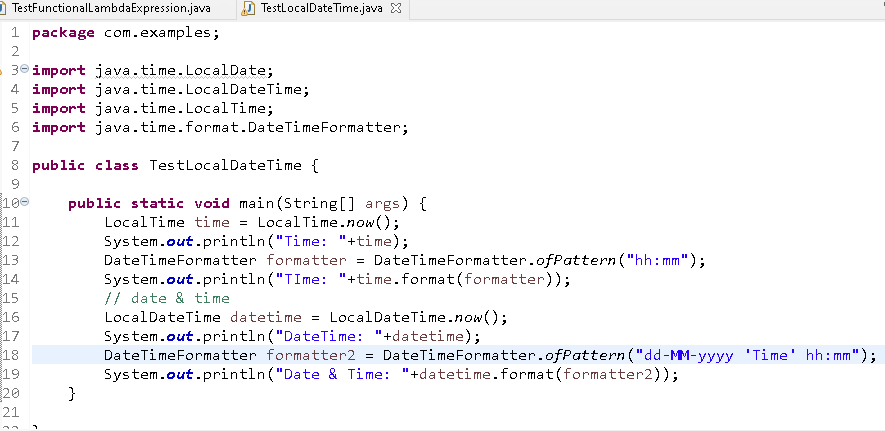
forEach method in Collection interface to simplify iteration using Lambda expression

Stream API’s: To work on bulk data while processing the collection

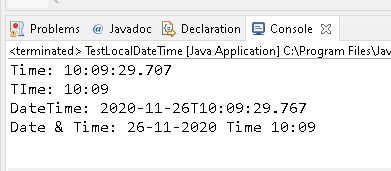
LocalDateDemo



DateTimeDemo

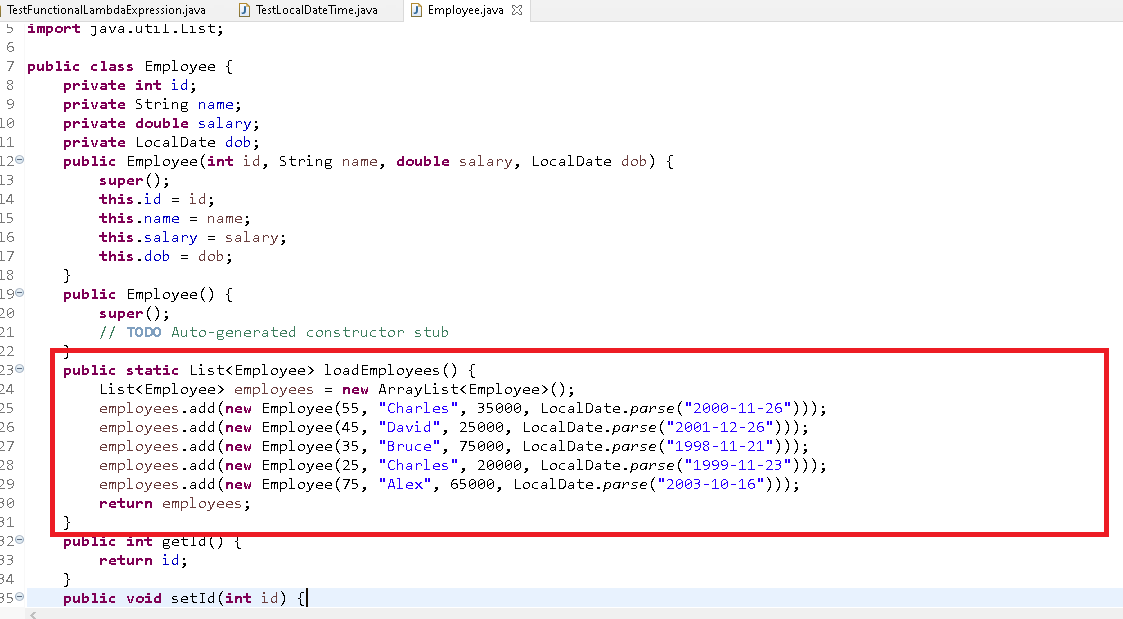


Output:



Functional Interface & Lambda Expression

Employee.java



TestFunctionalLambdaExpression.java



Exercise:

Create 3 classes

1. Main class
2. Employee class for object
3. Service class to maintain the employee objects in a List<Employee>

Main class will have a main method that shows a menu in a loop which keeps iterating until you exit, the menu must be

1. Add 2: Display Id 3: Display All 4: Sort by id 5: Sort by dob 6: Sort by salary 7: Sort by name 8: Exit

Note: Take all the inputs from the Scanner, you must create Scanner object only in the main class not in the Service class and also you must print the output only in the main class not in the Service class

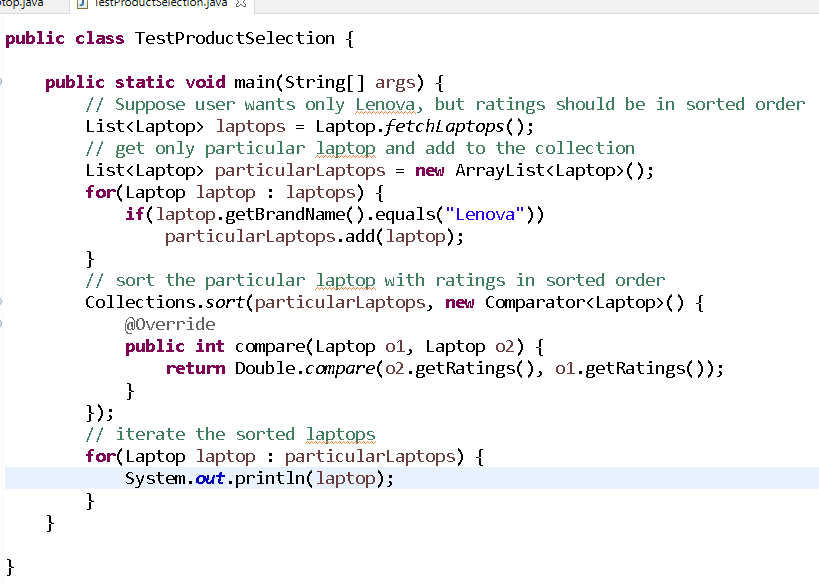
Service class will have the List<Employee> that will have methods to perform based on the menu you select like:   
 add(Employee e), findEmployee(id), getEmployees(), getSortedEmployeesById(), getSortedEmployeesBySalary(),... and so on

Stream API’s

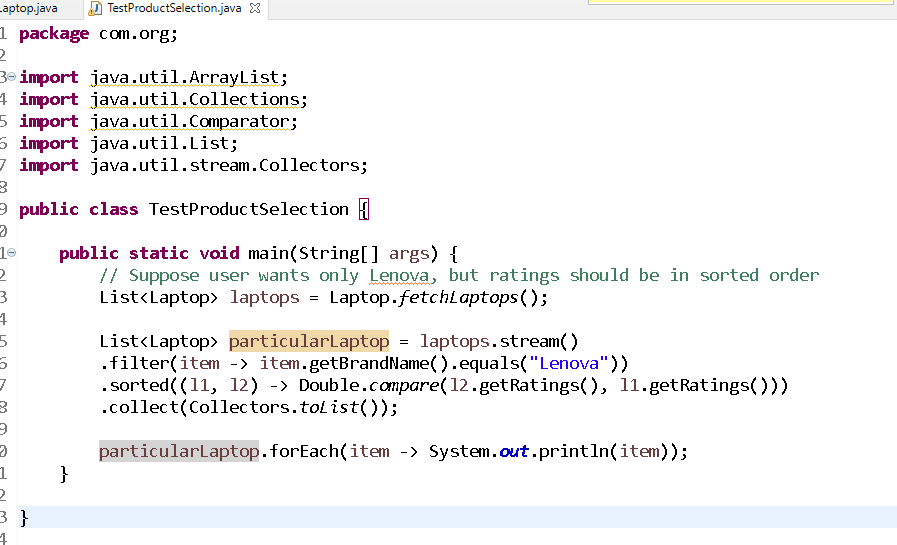
Java 8 introduced a new feature where you can process the collection of data in a declarative way any complex operations on collection can be done completely through lambda expressions to simplify the work.

Without declarative way approach developers had to write too much code

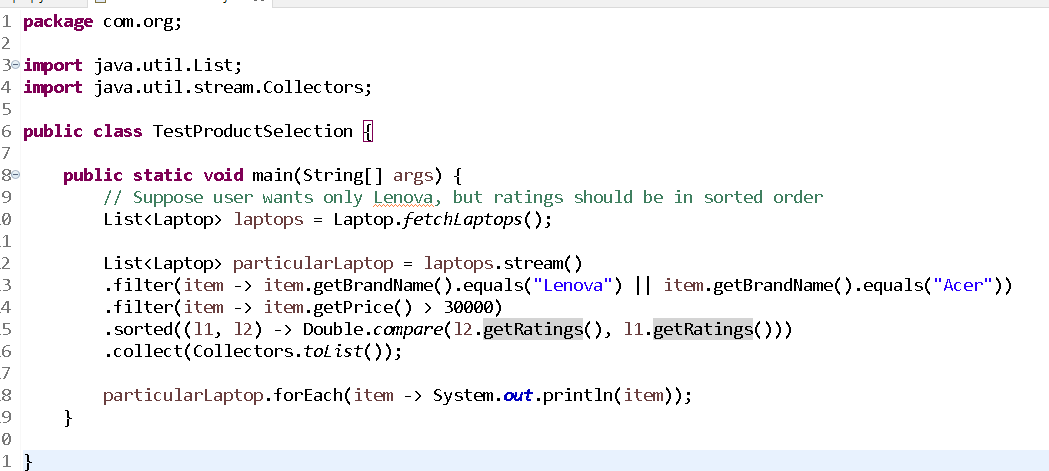
Suppose you want to select only a particular laptops from the collection of different brands and sort according to the ratings below is the code without Stream API & without Lambda expression.



Streams: it allows you to split all the items in a collection & process as independent items and pass to another stream for further processing

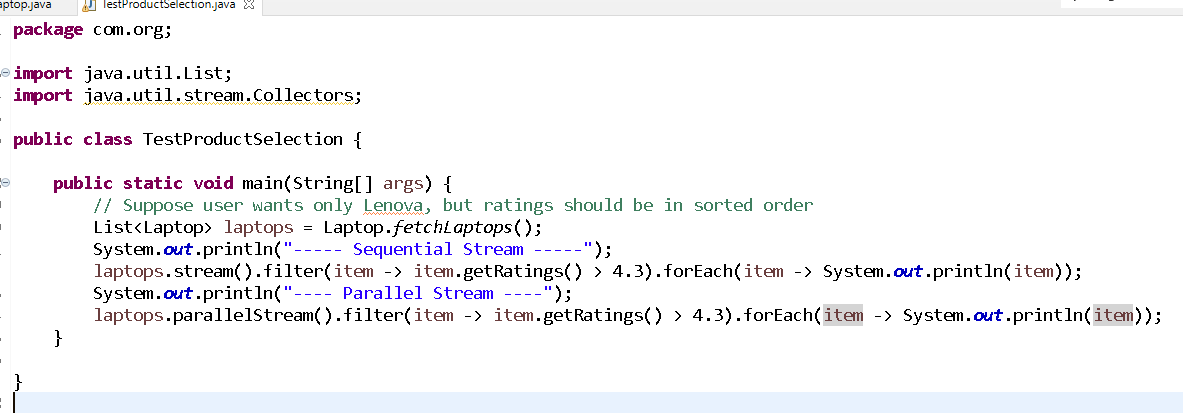


You can apply multiple conditions, suppose you need both Acer & Lenova



We have 2 types of Streams

1. stream: sequential way of processing
2. parallelStream: parallel processing



We have two types of operations in Stream

1. Intermediate: done in between like
   1. filter
   2. sorted
   3. map
   4. distinct
2. Terminal: done at the end
   1. forEach
   2. count
   3. collect

