New Features of Java 7 and Java 8

* Diamond Operator
* Improvements in Switch
* Improvements in Exception Handling
* Improvements in Resource Management
* Streaming API’s
* Functional Interface
* Lambda Expression
* Method Reference

Spring Framework

Spring Web MVC

Spring REST

Spring Boot

Spring Microservices

Javascript technologies

* Node.js
* ES6

Angular Framework

Features of Java 7

1. Diamond operator
2. Strings in switch
3. Improvements in Exception handling and resource management
   1. multi-catch statements
   2. auto closing of the resources
   3. auto flushing the output streams

Diamond Operator

It simplifies the instantiation of generics used in collections

Earlier:

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

Set<Employee> set = new HashSet<Employee>();

From Java 7 onwards you can use diamond operator at the right side

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

Set<Employee> set = new HashSet<>();

switch(option) {   
 case “test” : … break;

case “demo” : .. break;  
}

Multi-catch statements

A single catch that can handle multiple exceptions

catch(IOException | SQLException e) { … }

Auto closing & flushing of the resources

Earlier before java 7 programmers has to take care of closing and flushing the streams

FileWriter fw = new FileWriter(“abc.txt”)

// after the write operation

fw.flush();

fw.close()

From Java 7 onwards programmers need not to flush or close the stream, these are automatically taken care by try-with resource block

try (FileWriter fw = new FileWriter(“abc.txt”) {   
 // once the program exits from the try block streams are flushed and close automatically  
}

You can also avoid closing database resources if you use try with resource closing feature

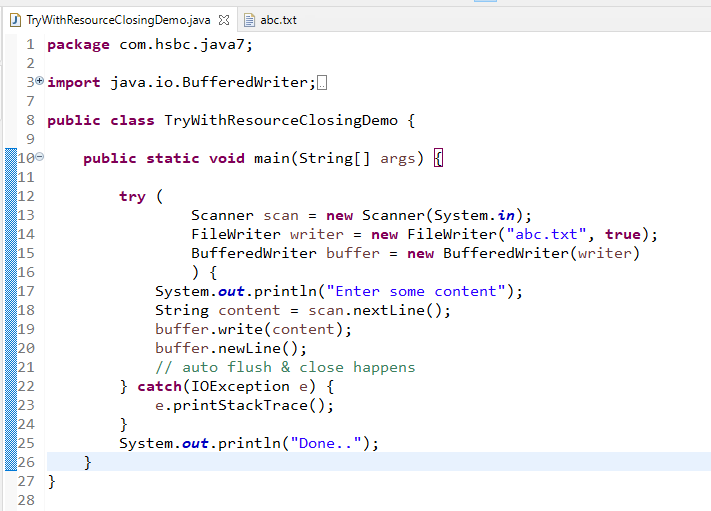
try (Connection con = DriverManager.getConnection(“”,””,””);  
 Statement stmt = con.createStatement()) {   
  
}

IN the above statement you don’t have close stmt & con

Try with resource

It is used only on those classes that are of type Closable or Flushable

ex: FileReader, FileWriter, BufferedWriter, Scanner, Connection, Statement, PreparedStatement, ResultSet and etc.



You can also use the try with resource in JDBC program



Earlier: Developers had to write close() on Connection, Statement and ResultSet in the finally block but close() throws checked exception so in finally again the close() should be wrapped in the try catch

Try with Resource closing feature: This eliminates developers writing lot of such codes

Summary on Java 7 features

1. Diamond operator
2. Strings in switch
3. Improvements in Exception Handling like Multi-catch statements, Try with resource closing feature
4. Automatically handling the resources with the help of try-with resource closing feature

Java 8 features

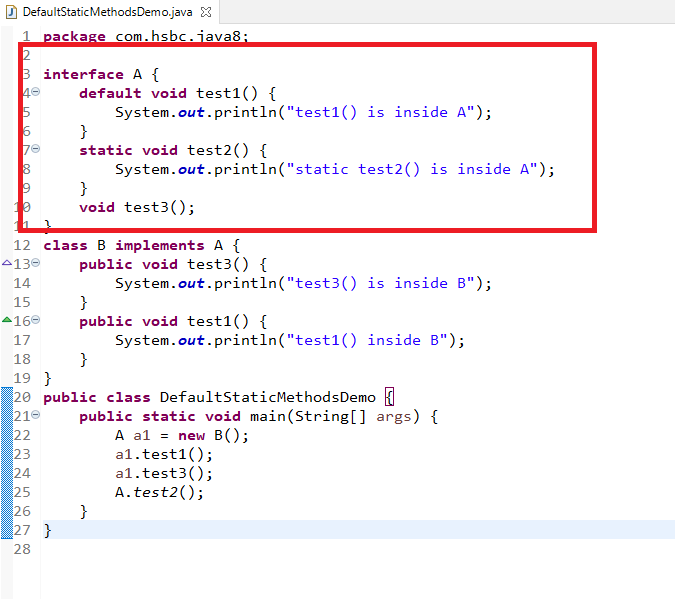
* Static methods inside interface
* Default methods inside interface
* Functional interface
* Lambda Expression
* Method Reference
* New Date & Time api’s: LocalDate, LocalTime, LocalDateTime
* Streams: to process huge data in the collection easily

Static & Default methods inside Interface

These are the interfaces which will have some default implementations for the methods and also you can have static methods

Default methods: They will have some default implementations in the interface if in case required to change you can change in the sub-class, these methods need not to override mandatorily

Static methods: These are the methods which are common for all the classes but you can’t override as it wouldn’t be polymorphic



Comparator: It is also used to provide sorting logic for a specific property of the object, but it is written in separate class unlike Comparable which is implemented in the same class

|  |  |
| --- | --- |
| Comparable | Comparator |
| method is compareTo(Object o) | method is compare(Object o, Object o) |
| It is implemented in the same class that are added in the list | It is implemented separately |
| It can only sort one property atmost | It can sort any number of properties as its written in separate class, you can create separate class for sorting each property |
| It can be used for natural sorting or default sort | It can be used to customize the natural sorting |

Anonymous inner classes

These classes don’t have names, but are implemented inside the method which would be useful only inside that method.

for Ex: sorting technique is useful only at the time of displaying the data to the end user, so sorting logic can be implemented in anonymous class instead of creating separate class for each sorting technique

Anonymous class will not have any names, it is more widely used when you have an interface with single methods and anonymous class instance is passed as a parameter as well.

Comparator is an interface with single abstract method you can write anonymous class for this without creating separate java file, Since Comparator can customize sorting for multiple properties of the same class, you don’t have to implemented multiple java classes that implements Comparator instead you can use anonymous inner class which doesn’t need separate class to implement the Comparator.

Ex:

Collections.*sort*(employees, **new** Comparator<Employee>() {

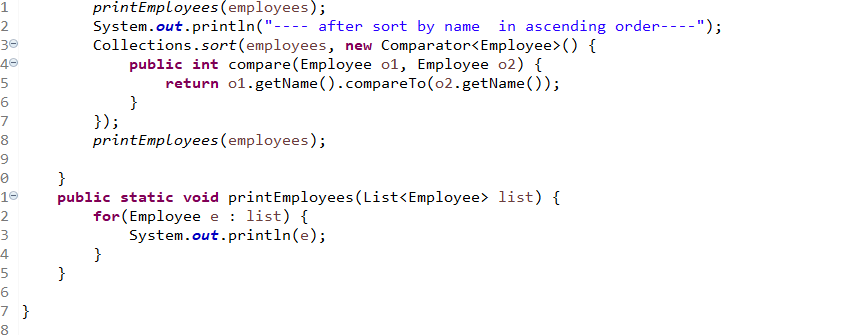
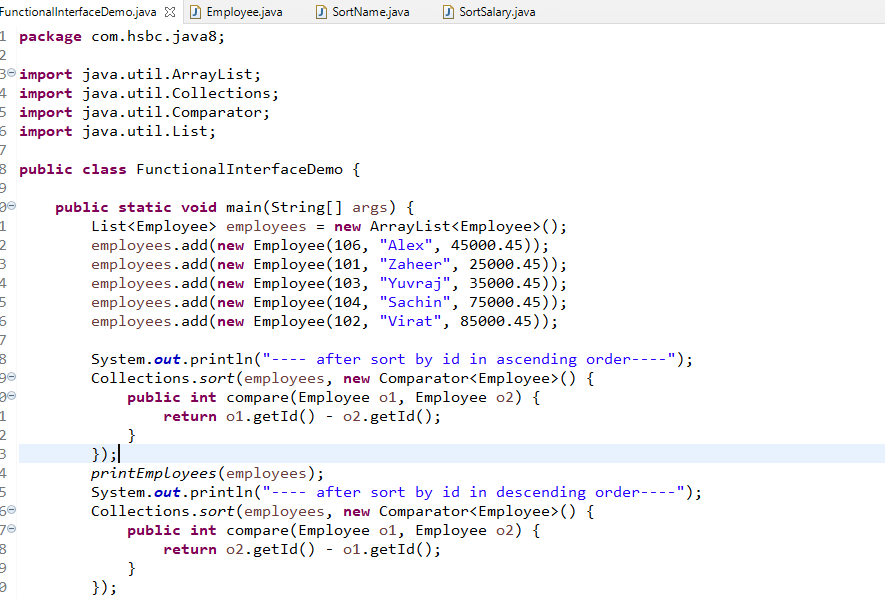
**public** **int** compare(Employee o1, Employee o2) {

**return** o2.getId() - o1.getId();

}

});

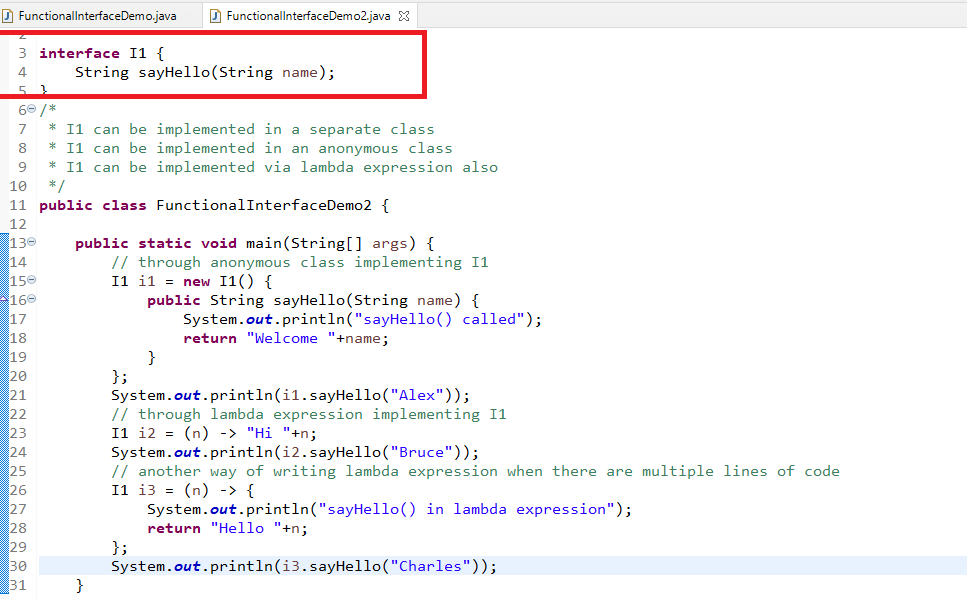
Here sort() method takes 2 parameter 1st is the list & 2nd is the Comparator reference, where we had passed comparator implementation through anonymous class, the sort() method calls compare()(2nd parameter) method to sort the items in the list(1st parameter)



The above code is using anonymous class for Comparator in the sort() method.

Functional Interface:  
These are interfaces with single abstract method and will have @FuntionalInterface annotation on top of the interface, it enables us to pass code as a parameter which is nothing but function as a parameter through lambda expression, but lambda expression is applied on to the function interface i.e., should have only one method.

Lambda Expression: It simplifies writing the implementation for the functional interface with a simple expression instead of writing the anonymous class.



You can also pass lambda expression in the parameter which is like passing the function itself as parameter, this you can do only in functional interface



Lambda expression can be applied only on the functional interface i.e., an interface having only one method, which is why you don’t have specify the method name return type, parameter type while creating lambda expression as the compiler knows the signature of the method when pass lambda expression to the parameter,

i.e., Collections.sort(List, Comparator), takes 2nd parameter as a lambda expression for the Comparator interface compare method only.

Valid Lambda Expressions

() -> “Hello “;

() -> 20;

() -> return 20; // invalid

() -> return “hello”; // invalid

() -> { return 20; } // valid

(x, y) -> x + y; // valid

New Date & Time Api’s

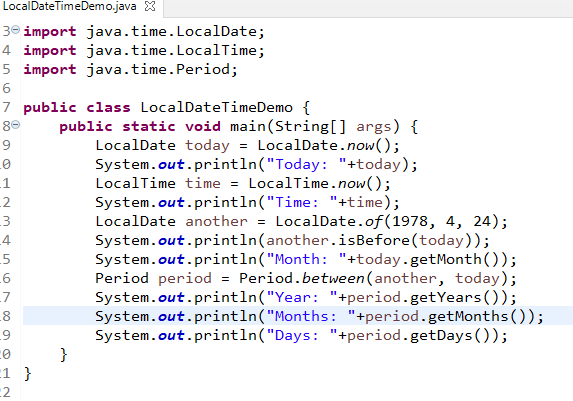
Old Date & Calendar classes had lot of issues like months were starting from 0 instead of 1, trying to add or subtract date were not give the expected output, Date was showing date & time both, hence Java 8 introduced new classes for Date & Time like:

LocalDate, LocalDateTime, LocalTime

LocalDate.now(), LocalDateTime.now(), LocalTime.now(): all these methods returns corresponding instance of current date/time

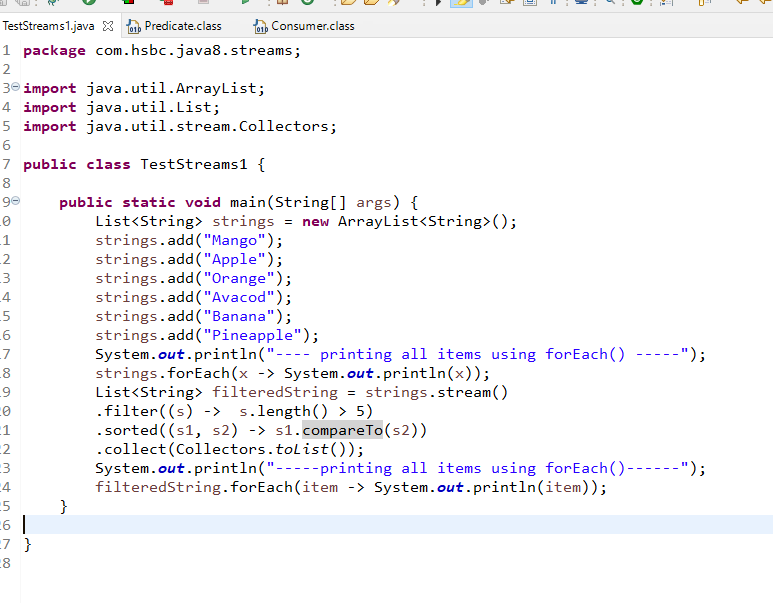
It also have methods to convert string to date & time

ex: LocalDate dob = LocalDate.parse(“2010-10-22”)



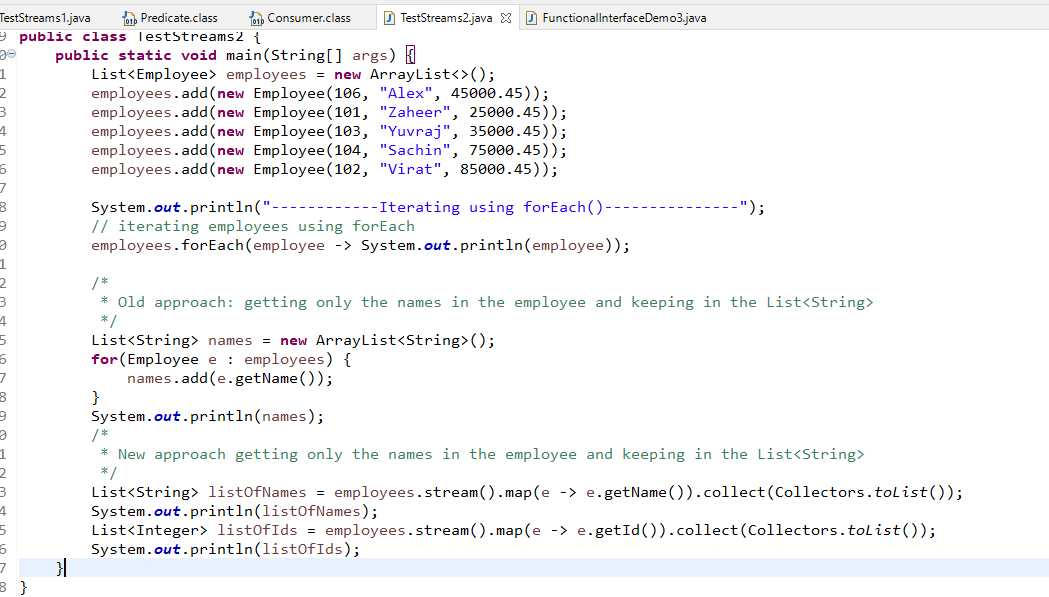
Java 8 Streams

It is used to process the bulk data to perform operations on collections, like filtering, iterating, sorting, collecting, mapping and so on.



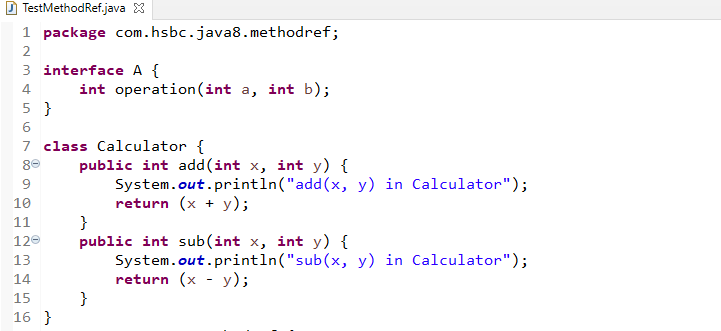
With Streams lot of complex operations can be done in a simpler way without using any secondary variable

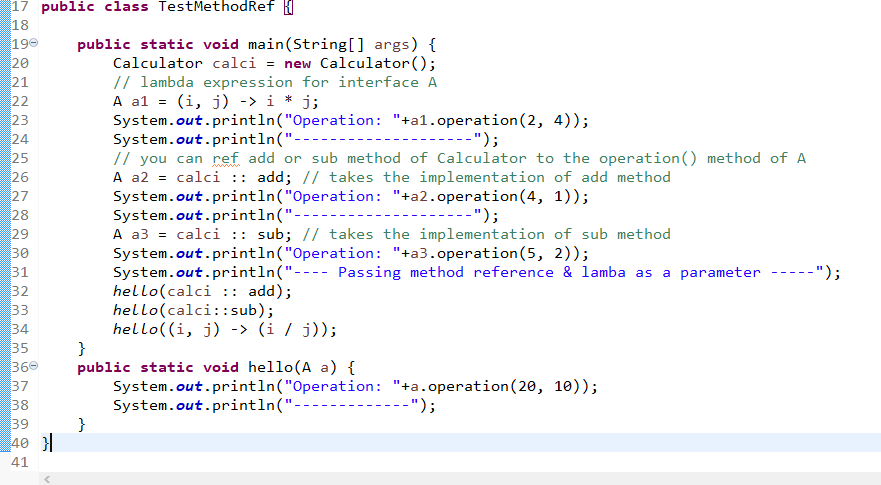
ex: Assume you have products with id, name, price and you want to keep all the product id’s in one list, all the product names in another list and all the product price in another list, then you need to iterate 3 items and apply conditions 3 times and instantiate the list 3 times and segregate all the types of data in separate datastructures without streams, but in streams you have a map() to convert from one type to another



Method Reference:

It allows you to reuse the existing method logic for the functional interface instead of implementing the functional interface via lambda expression or through some classes





Output:

