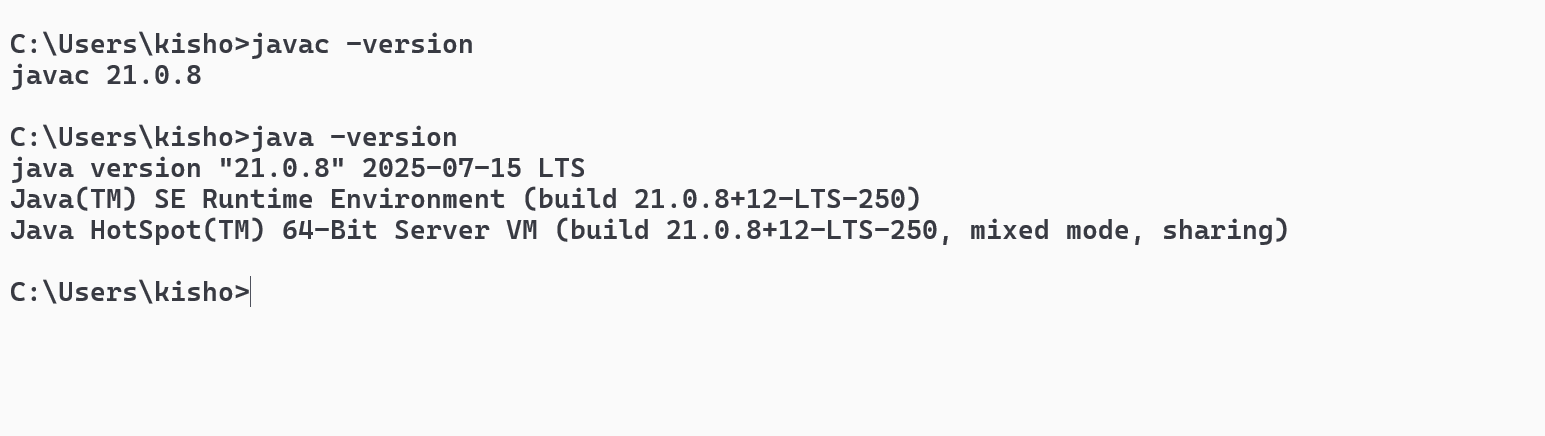
Java, Spring Boot and React.js

Software’s requirements

1. JDK 21
2. Eclipse IDE Enterprise
3. VS Code

Verify Java



Java is a platform independent and object oriented programming language

JVM - Java Virtual Machine

Object oriented - real world entities applications

Objects: Properties & Behavior

Bank Application - Customer, Account, Employee, Loan

Class - Blueprint of an object / template

Packages in Java - these are folders to categorize Java classes

example: com.mahindra [or] com.birstelstone

Fundamentals of java

1. Variables & Datatypes
2. Operators
3. Branching Statements
4. Looping constructs
5. Arrays
6. Variables & Methods

Datatypes

Primitive data types

1. byte - 1 byte (-128 to +127)
2. short - 2 bytes
3. int - 4 bytes
4. long - 8 bytes
5. float - 4 bytes
6. double - 8 bytes
7. char - 2 bytes
8. boolean - 1 byte

byte, short, int & long represents integer values

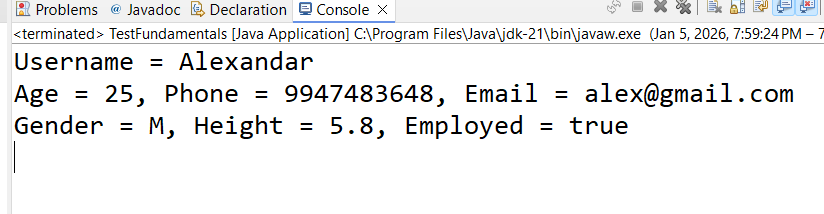
Non-primitive data types - class, arrays, interface

String, LocalDate, LocalTime, Employee, Customer

TestFundamentals.java

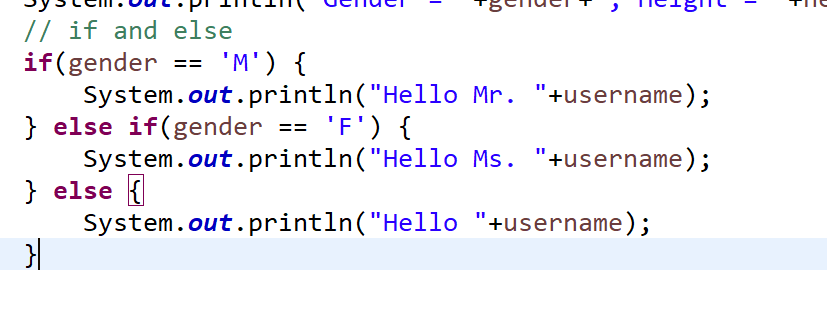


Output:



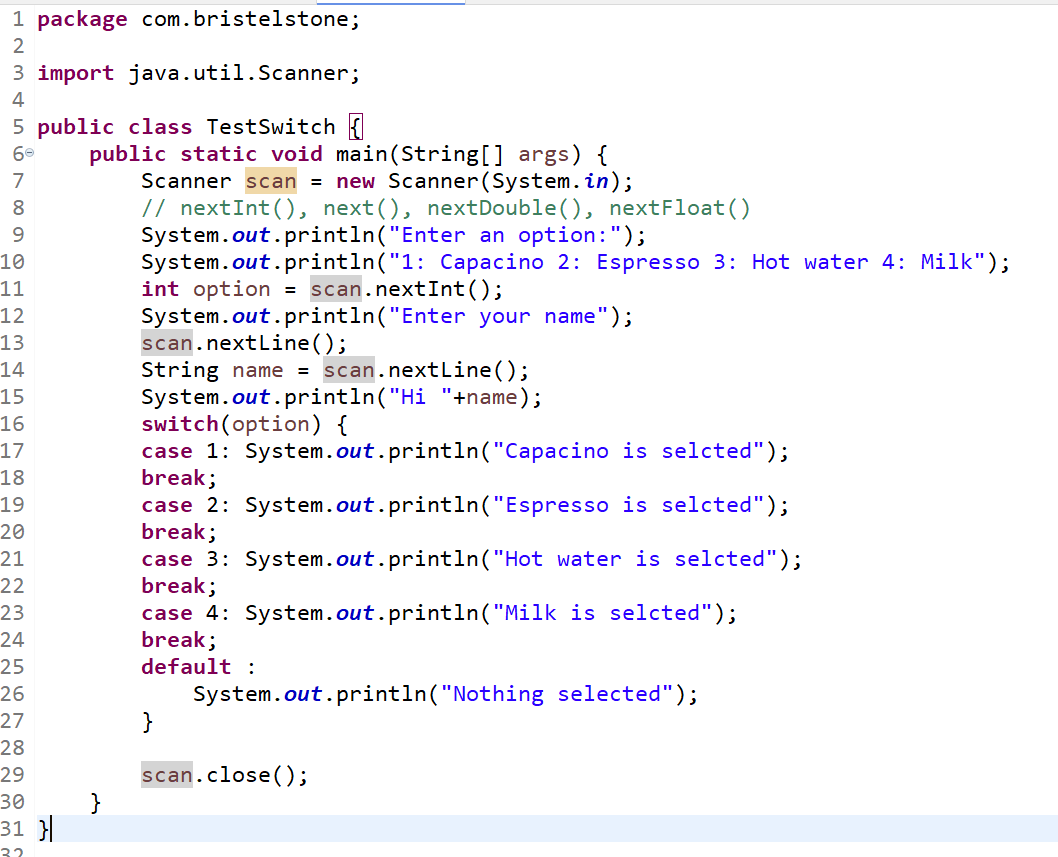
Branching Statements

* if
* if else
* if else if else if else
* switch

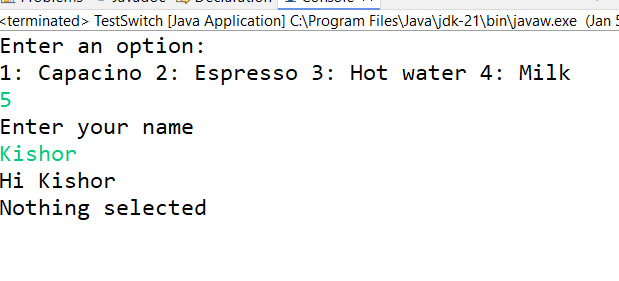


Switch statements

These are going to evaluate on list of conditions with case label



Output:



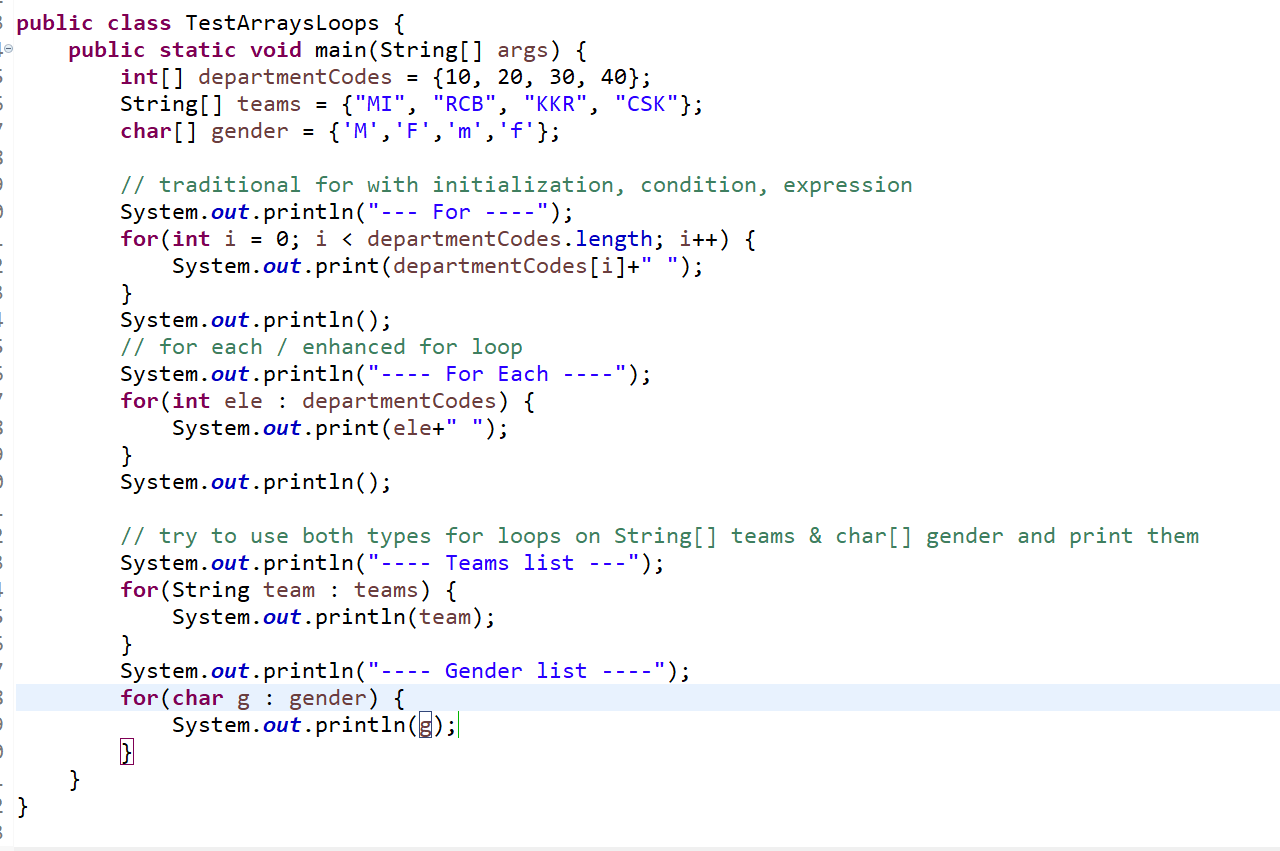
Arrays & Loops

Arrays are containers that can store multiple values of same datatypes

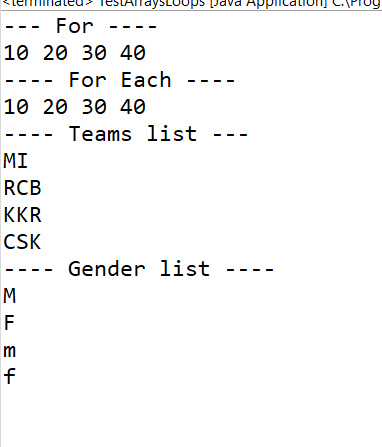
int[] departmentCodes = { 10, 20, 30, 40} ;

String[] teams = {“KKR”, “RCB”, “MI”, “CSK”} ;

char[] gender = {‘M’, ‘F’, ‘m’, ‘f’}



Output:

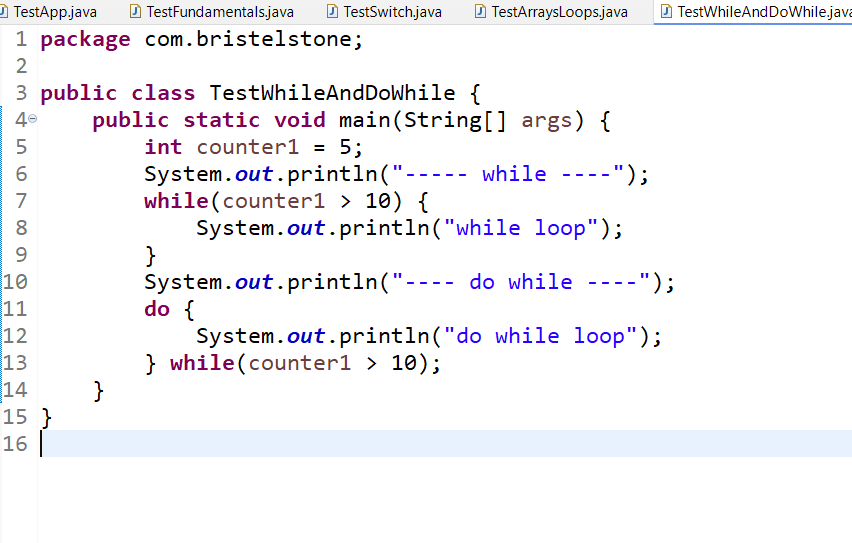


While & Do While loop

It runs the statements repeatedly until some condition becomes false

while - checks the condition first & then runs the loop

do while - similar to while loop, but runs the loop first & then checks the condition, it is executed atleast once



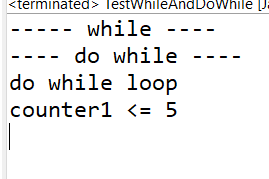
Operators in Java

Operators perform operations on the variables

++, --, ==, =, +, -, \*, /, %, <=, >=, <, >, !=, ?:



Output:



Classes & Objects

Classes are the blueprint of an object and objects are the instances of the classes.

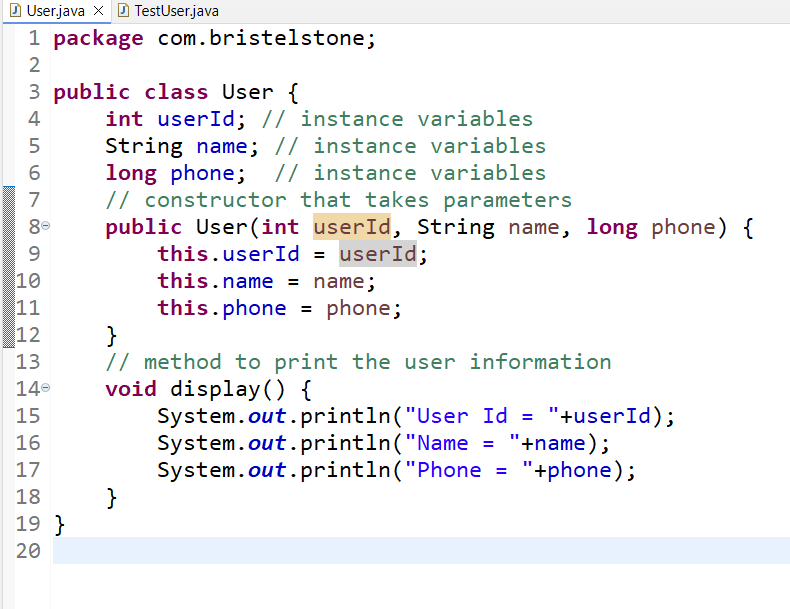
What all the members we can write inside the class

* variables - static & instance
* constructors
* methods

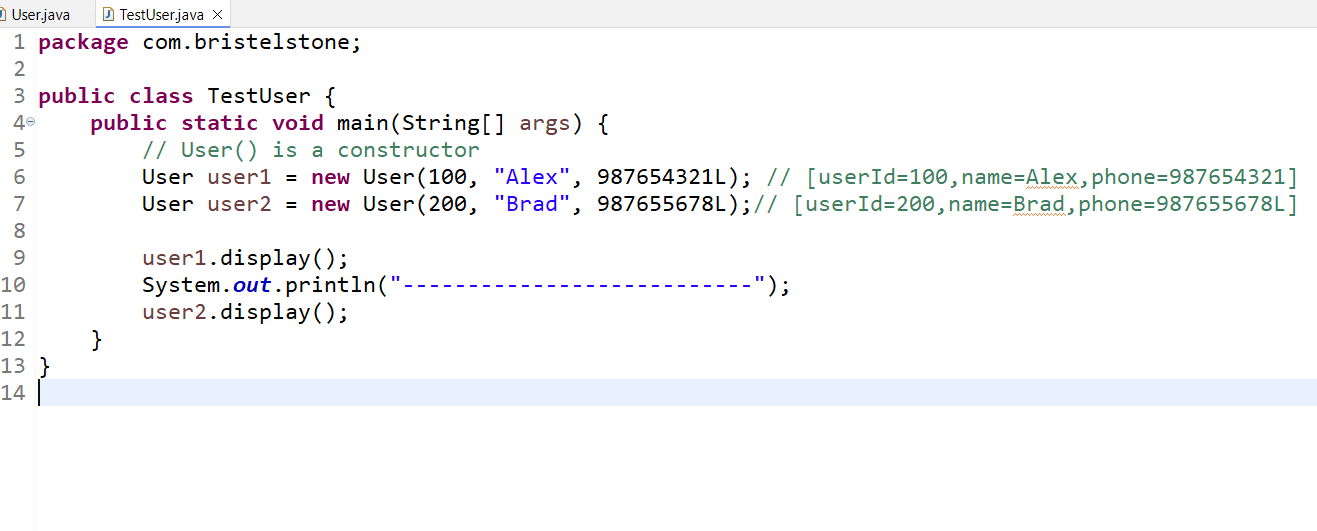
Constructors: These are just like methods but their names will be same as the class name and doesn’t have return types

1. Constructors are called when you create objects
2. You can write more than one constructor inside the class - it means you can overload it
3. By default a class gets a default constructor if there’s no constructor inside the class, however if you provide the constructor then default constructor is not created

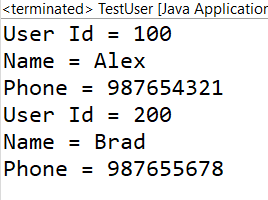
User.java



TestUser.java



Output:

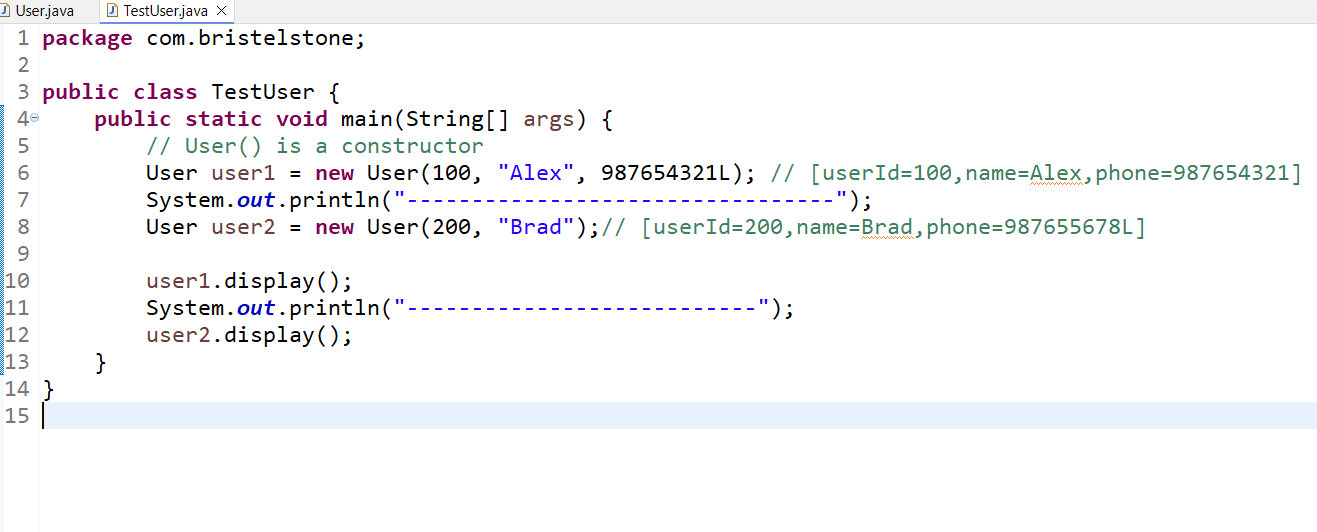


this(): It is used to invoke a constructor from another constructor, you must always write them in the first line of the constructor

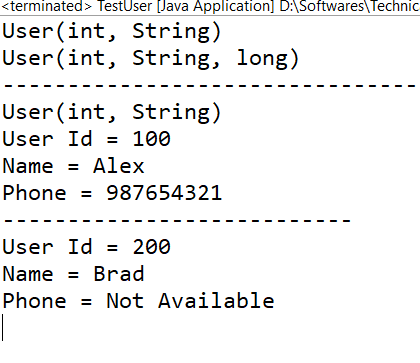
User.java



TestUser.java



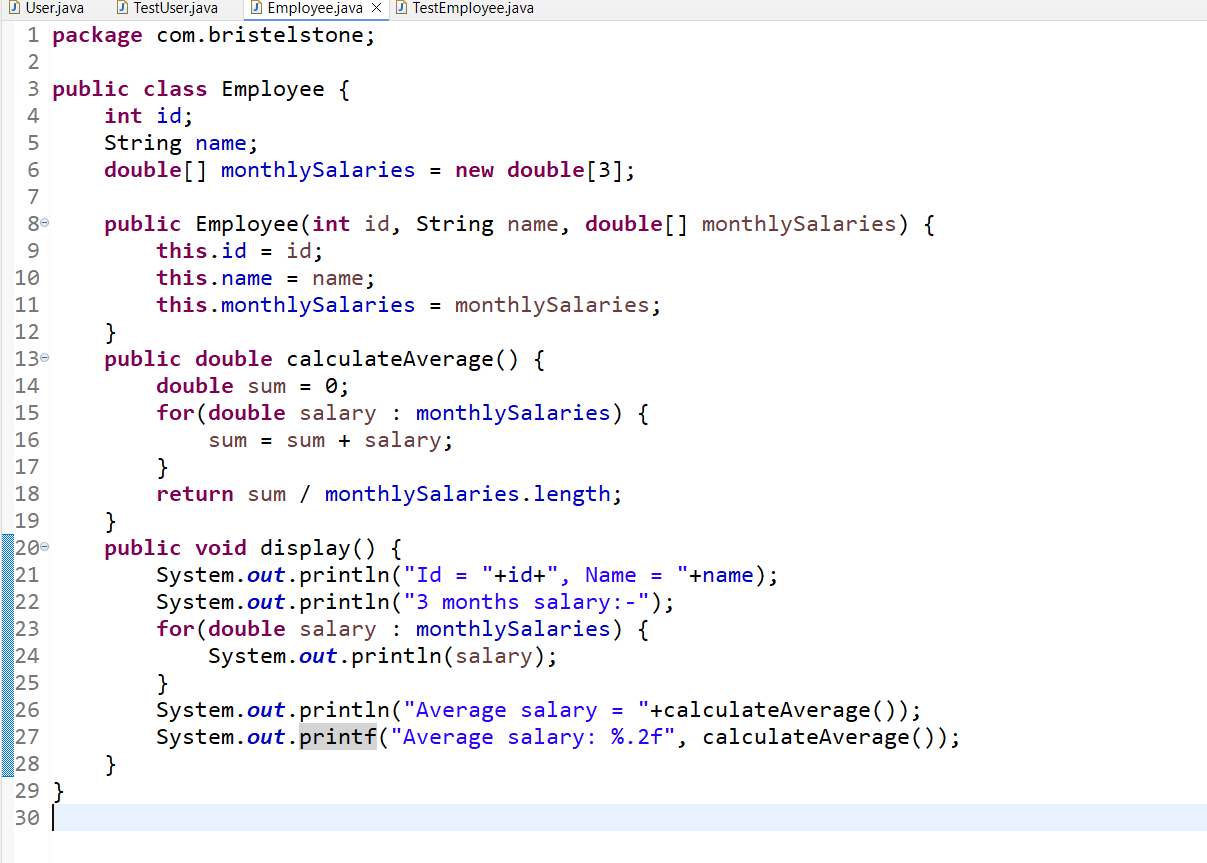
Output:

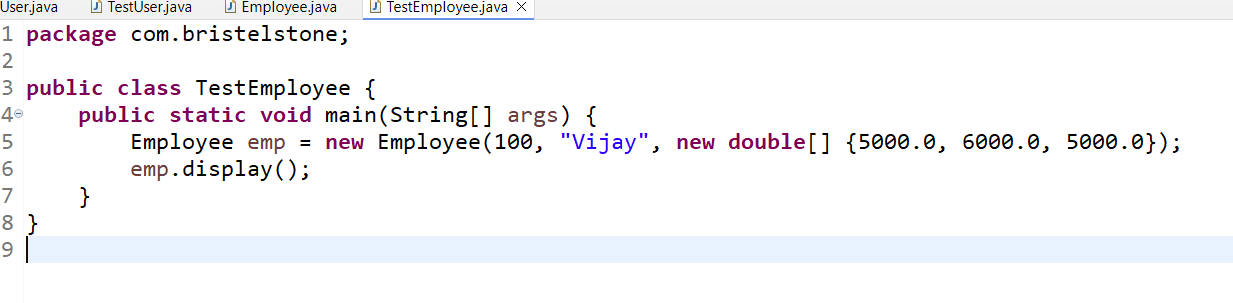


Activity:

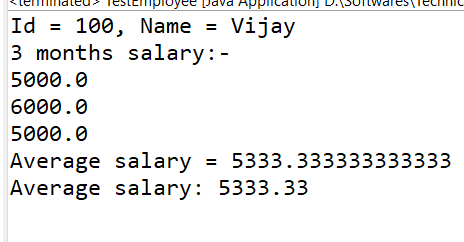
Create a Employee class with id, name, monthlySalaries[] array, it must be 3 months salary, create calculateAverage() method that returns average salary of 3 months, then create a display method that prints id, name, 3 months salary and average salary by calling calculateAverage() method

Create a main method and create 2 employee objects and invoke display() on each object.





Output:



static members:

static variables are those variables where multiple objects share single copy of it, we can modify the static variables

static methods: These methods can have some logics that are common to all the objects

OOPs principles

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

Encapsulation:

Hiding the data and accessing them only through public methods like setters / getters

setters -> to set the value

getters -> to read the value

class Employee {   
 int id;  
 String name;  
}

Employee emp = new Employee(100, “Raj”);

emp.id = 1234;  
emp.name = “123Xyz”;

class Employee {   
 private int id;  
 private String name;  
   
 public Employee(int id, String name) { //initialization }  
 public void setName(String name) { if (condition) this.name = name; }   
 public int getId() { return id; }  
 public String getName() { return name; }  
}

Employee.java



Right Click -> Source -> Generate Setters & Getters

Inheritance

It is a process of acquiring properties & behaviors of an object from another object.

Assume you have some classes to design for your application like

* Employee - id, name, gender, phone, salary, dob, desig
* Student - rollNo, name, gender, phone, marks, dob
* Customer - customer\_id, name, gender, phone, dob, accountNumber

In the above classes you have some common properties like name, gender, phone, dob and some properties which are specific to the classes like Employee has id, salary, Student has rollNo, marks, Customer has customer\_id, accountNumber

extends keyword is used to inherit properties & behaviors

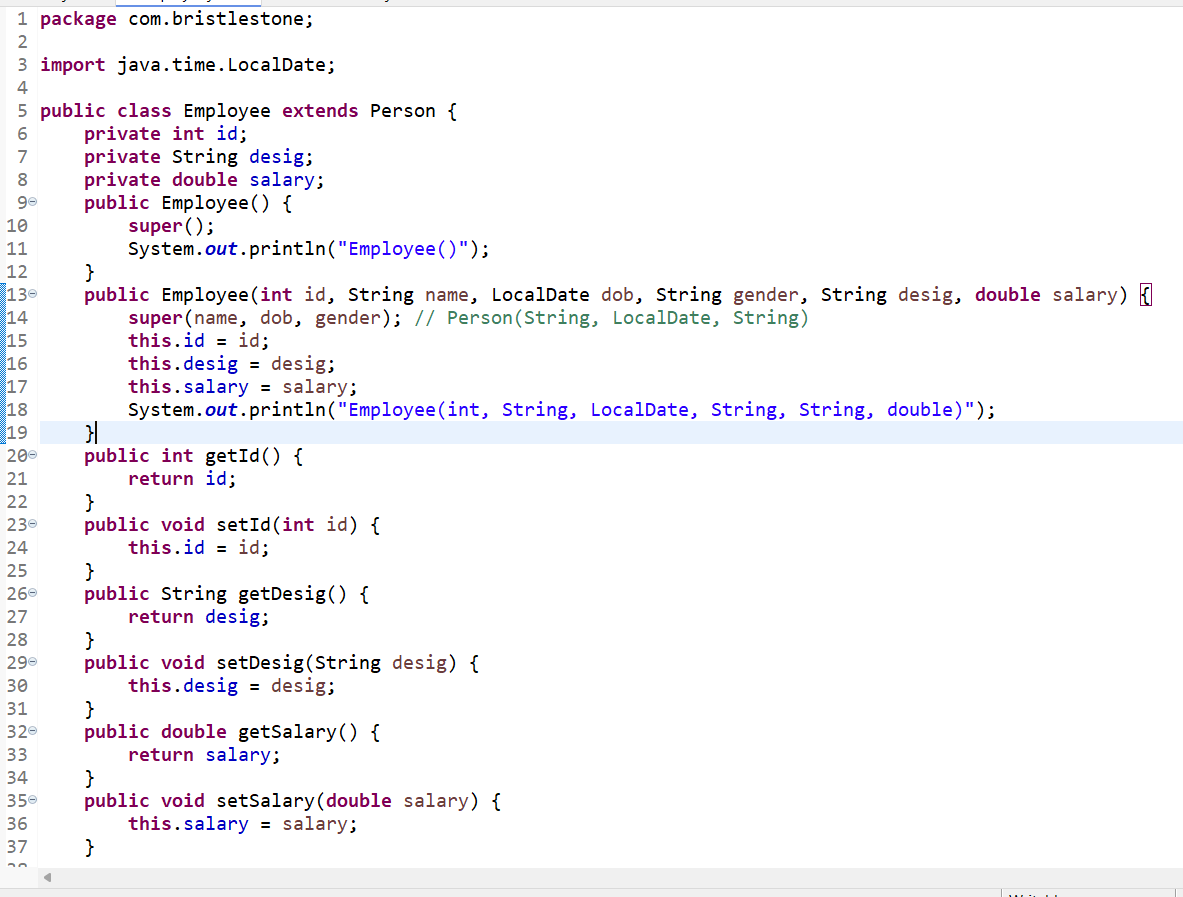
Note: private members & constructors wouldn’t be inherited to the subclass

* Person : name, gender, phone, dob
* Employee extends Person - id, salary
* Student extends Person - rollNo, marks[], departmentName
* Every subclass invokes super class default constructor automatically with super() statement, but you can use super(args, args,..) to invoke the parameterized constructor of the super class
* If the super class doesn’t have default constructor then subclass must explicitly call super(args, args,..) to call the parameterized constructor of the super class

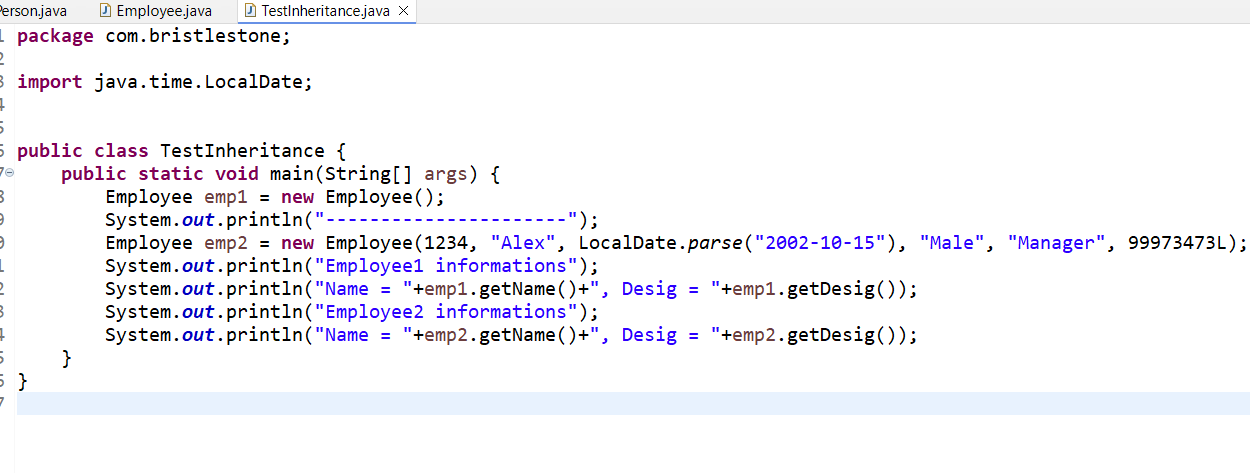
Person.java



Employee.java



TestInheritance.java



Activity:

Create a Student class that will have rollNo, name, gender, dob, phone, marks[] array (pass 3 marks in int format)

Create an Account class that will have accountNo, balance, then create a Customer class that will have customer\_id, name, gender, dob, phone, account (Account as a parameter), Customer will have a constructor that accepts all the above properties

Using Person class inherit Student & Customer class and from main create objects of these 2 classes

Is - a relationship and Has - a relationship

Is - a relationship: Inheritance

class Employee extends Person { }

Has - a relationship: Composition

class Customer extends Person { // extends - is-a relationship  
 Account account; // has-a relationship  
}

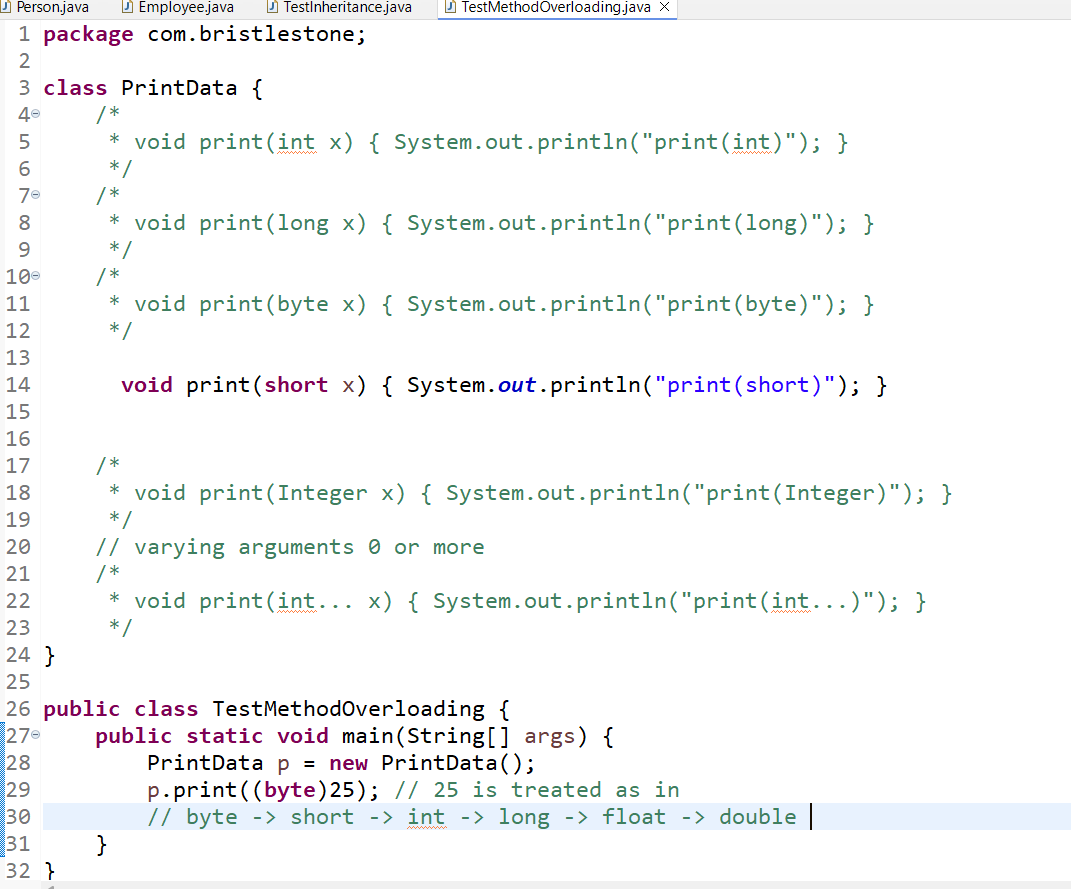
Polymorphism:

An object with many forms

1. Method overloading - you will create methods with same name but different signature (different types of parameters, order of parameters), compile time polymorphism - because method invocation is determined at compile time
2. Method overriding - you will have same method names and same signature but different logics in the subclass, runtime polymorphism - because method invocation is determined at runtime

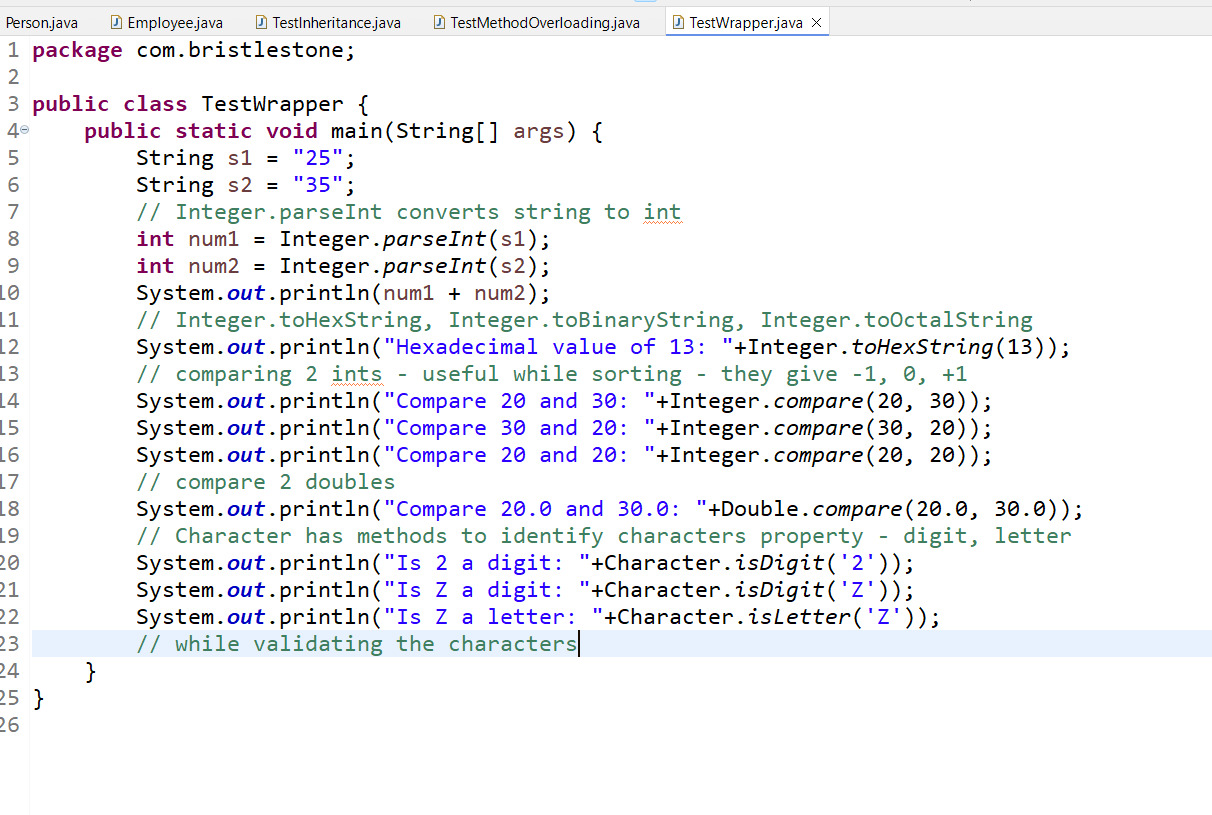
Wrapper classes - these are the classes provided for every primitive datatypes to perform some advanced operations on the primitives

int - Integer  
short - Short  
byte - Byte  
boolean - Boolean  
char - Character  
float - Float  
double - Double  
long - Long

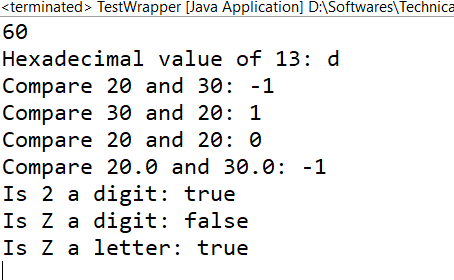


Wrapper classes

These are the classes provided to perform some complex operations which primitives can’t do



Output:



isLetter vs isAlphabetic in Character class

isLetter works for all the letters, isAlphabetic is super set of isLetter, all the letters are part of alphabets, isAlphabetic also used for other characters like roman numbers

Has a relationship

It can be achieved using Aggregation and Composition

Aggregation: Parent and Child objects are independent to each other: loose coupling

Composition: Child objects doesn’t exist without Parent object: tight coupling

// Aggregation

class UserService {   
 private UserRepository repo;  
 UserService(UserRepository repo) { this.repo = repo; } //   
}

UserRepository object you will supply outside the UserService, it doesn’t create the UserRepository itself, this is aggregation

// Composition

class Order {   
 private OrderItem item;

public Order() {   
 this.item = new OrderItem(); // composition  
 }  
}

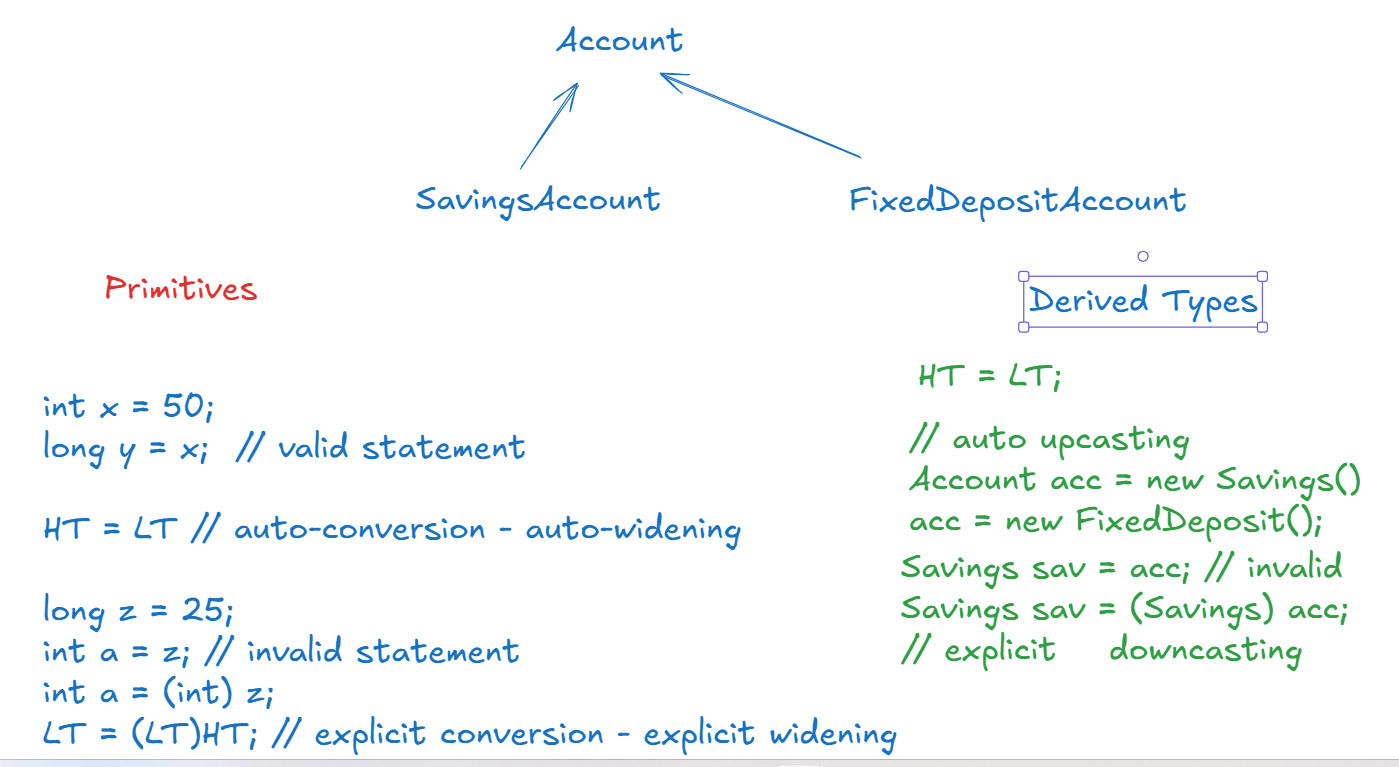
// another example  
class UserAccount {   
 private UserCredentials credentials;  
 public UserAccount() {   
 this.credentials = new UserCredentials(“…”, “…”);  
 }  
}

Method Overriding:

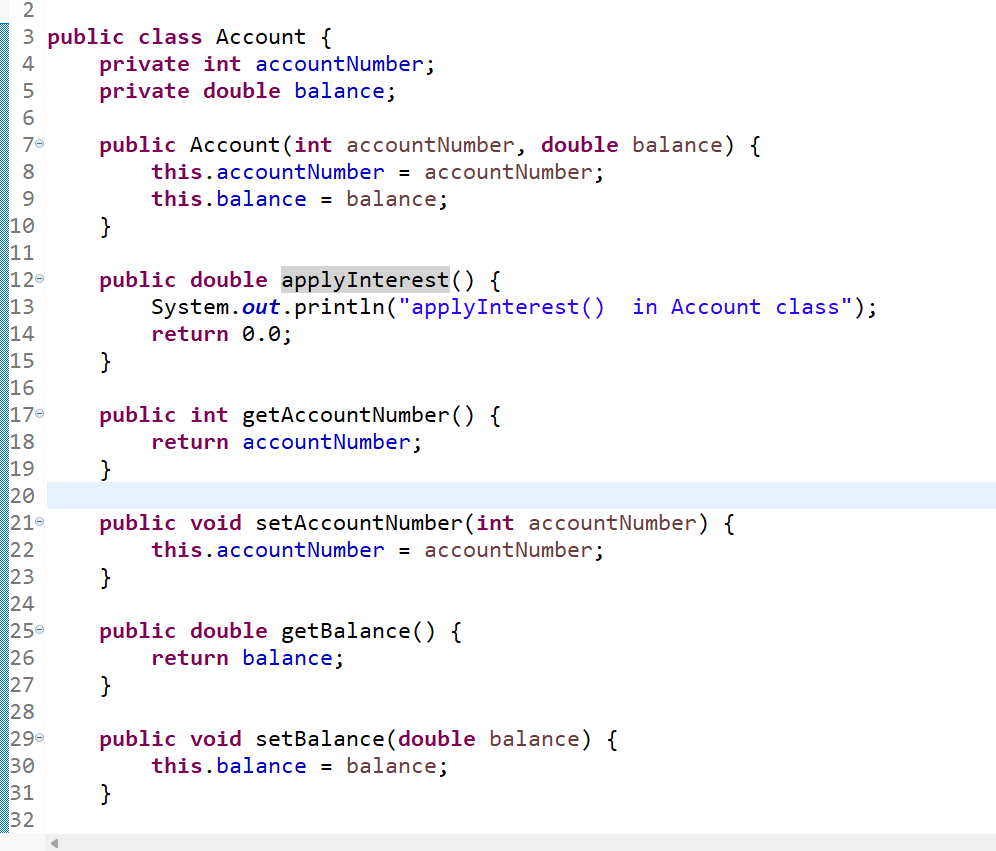
It is about having same method in the super class & subclass with same signature but different implementation logic

Person, Employee, Student, Customer

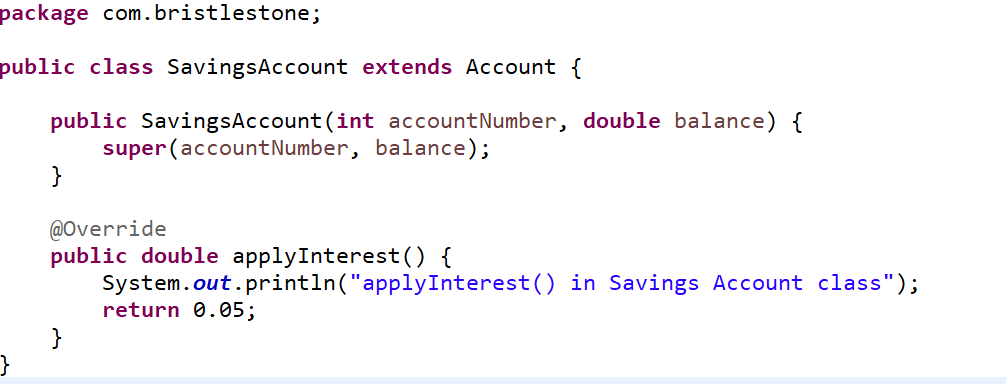
display() { .. }



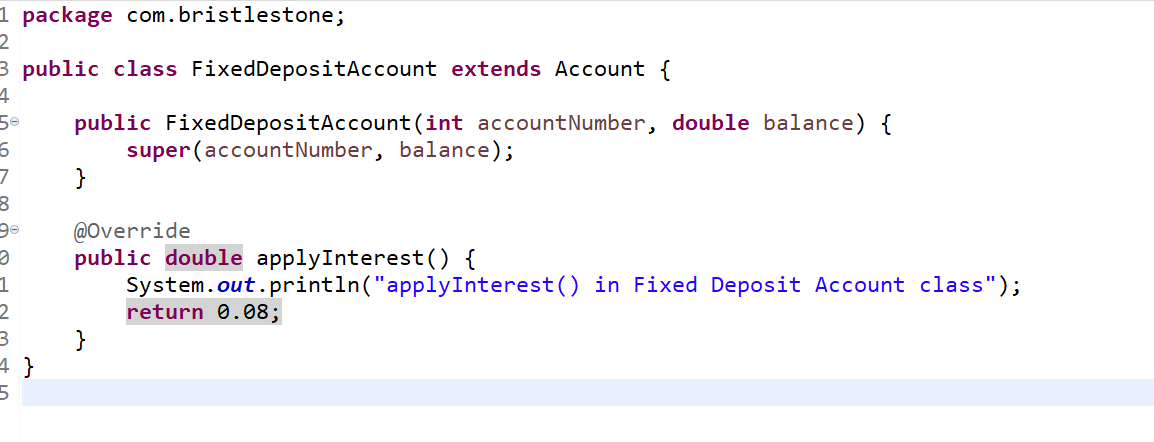
Account.java



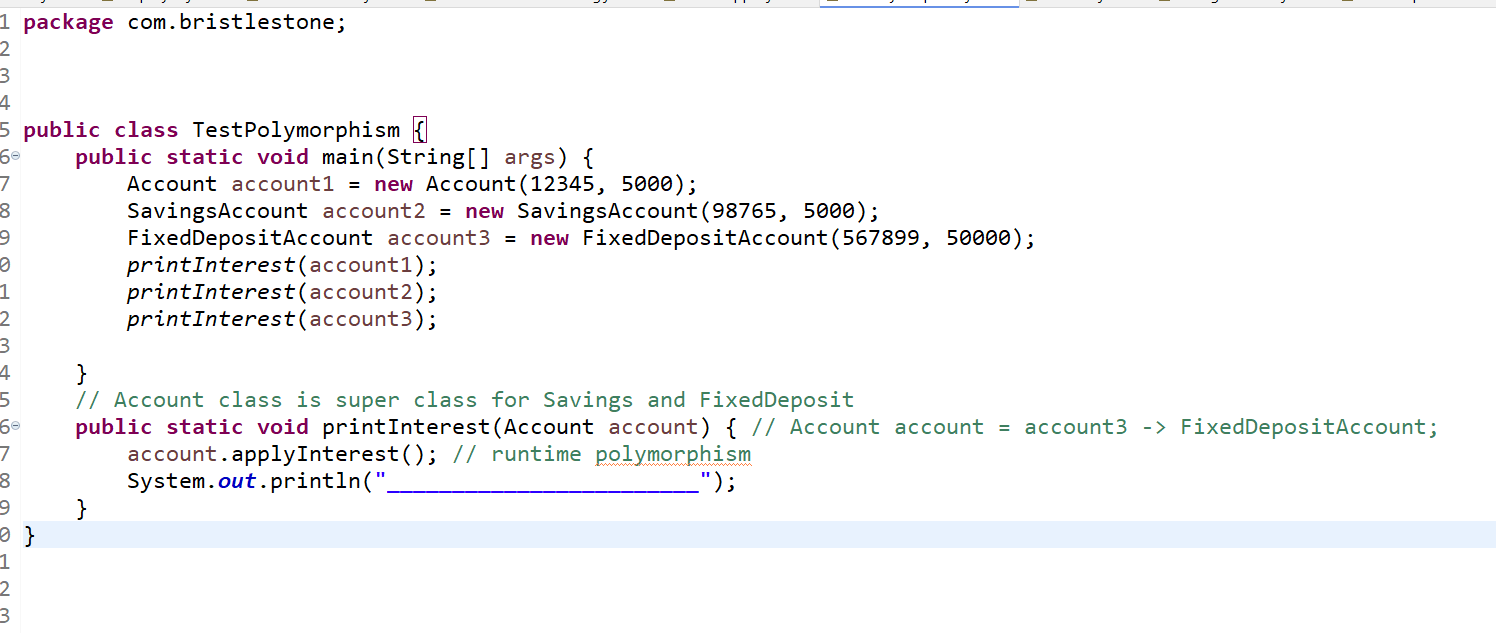
SavingsAccount.java



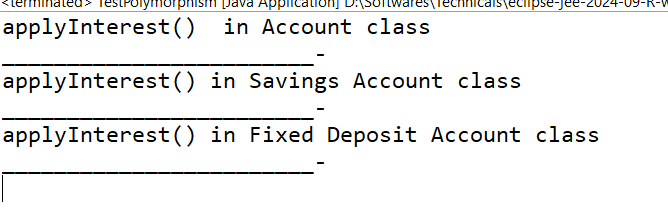
FixedDepositAccount.java



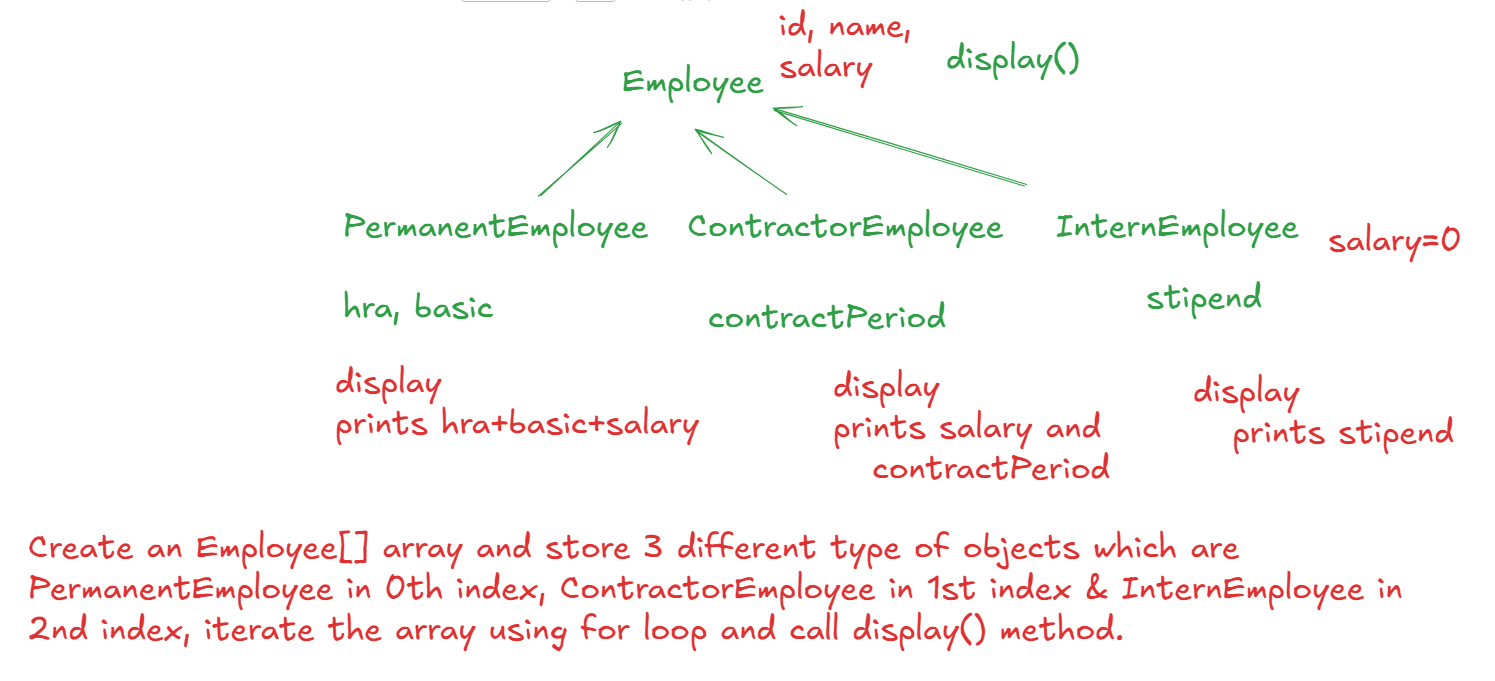
TestPolymorphism.java



Output:



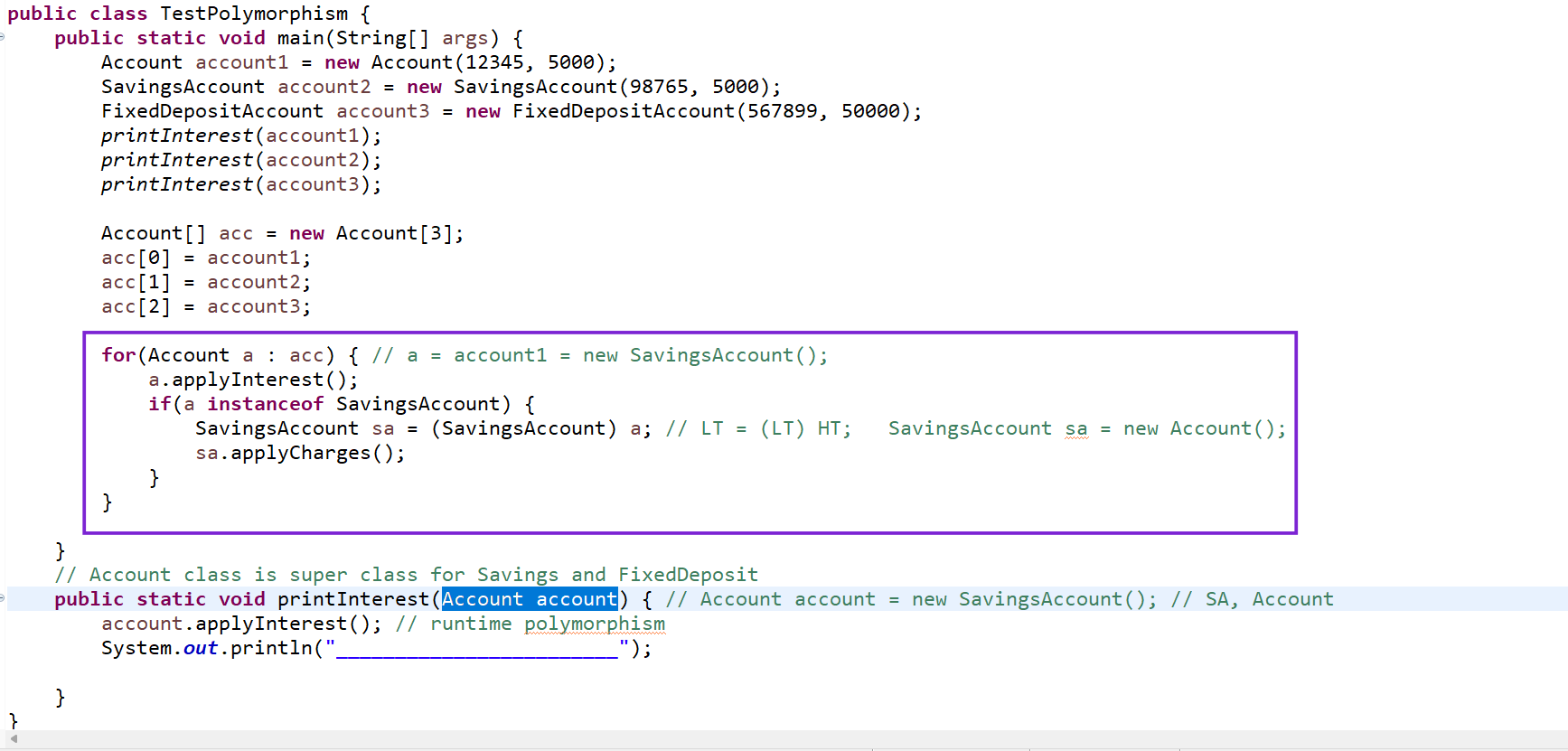
Activity



instanceof: It is a keyword used to check the type of object when a super class is handling various sub class type of objects, it avoid ClassCastException

Explicit Downcasting: In many cases, you will use super class reference variable to handle all the sub-types, but when you need to access members of sub-classes then you need a reference variable of subclass type

SavingsAccount sa = (SavingsAccount) a;



Root class in Java

class Person { }   
class Employee extends Person { }

If a class doesn’t extend any class then it automatically inherits Object class

Object class is the root class for all the classes

final keyword:

It is applied to variables, methods & classes

final variables: You can’t change, it can be applied on instance variable, static variable and even on local variables

Note: final variables can be initialized at the time of declarations or if its instance variable you can initialize in the constructor

final methods: These methods you can’t override, but you can inherit

final class: The class which you can’t extend

Abstraction:

It hides the complexity from the end user and shows only necessary details to the end user.

It helps end users to understand what methods do instead of knowing their complex implementations, this adds flexibility in the code so that the can use the methods without knowing its internal logic.

Abstraction is achieved in two ways

1. interface -> it will have only abstract methods and constants
2. abstract class -> It will have both abstract & concrete methods

What are abstract methods:

These are the methods without body or implementation

What are concrete methods

These are the methods with body or logics

Interface

It will have only abstract methods & constants

interface TicketBooking {   
 void bookTicket(); // public abstract void bookTicket()  
 void printTicket(); // public abstract void printTicket();  
}

Who will provide body: Classes implement the interface

class RailwayTicketBooking implements TicketBooking {   
 // it has to provide body for all the abstract methods mandatorily, else a class can be abstract  
}

class FlightTicketBooking implements TicketBooking {   
 // it has to implement all the methods of TicketBooking, else it can be made abstract   
}

Where exactly interfaces are useful in real time

Interfaces acts like a contract between two programs, so that both the programs would use same methods so that both knows the rules.

Abstract class:

It can have both abstract and concrete methods, it can be used when you know partial implementation of the class

abstract class Account {   
 void display() { … // prints account details }  
 abstract double applyCharages();  
}

class Savings extends Account {   
 // sub class must mandatorily implement abstract methods

double applyCharges() { return 0.02; }  
}  
class Current extends Account {  
 double applyCharages() { return 0.03; }  
}

Car showroom application

abstract class Car {   
 abstract void mileage();  
 abstract double getPrice();  
 void basicFeatures() { … }  
}  
class Creta extends Car {   
 // you must override mileage() & getPrice()  
}  
class I20 extends Car {   
 // you must override mileage() & getPrice();  
}

Abstract class vs Interfaces

|  |  |
| --- | --- |
| Interfaces | Abstract class |
| all the methods are abstract by default | you can have abstract & concrete methods both |
| You can’t create constructor | You can create constructor |
| all the variables are constants by default | variables are not constants by default |
| members are public by default | members are not public by default |

Common feature of abstract class & interface

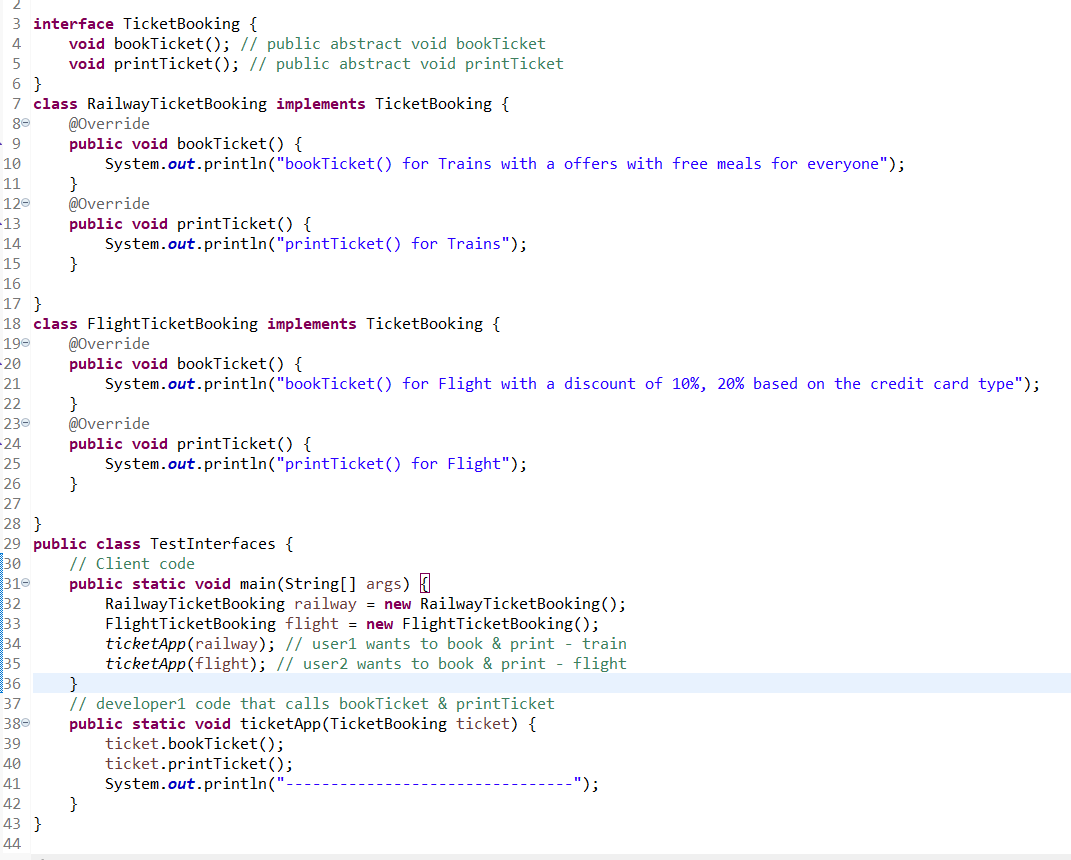
You can’t create object for abstract class or interfaces

Car c = new Creta(); // this is valid

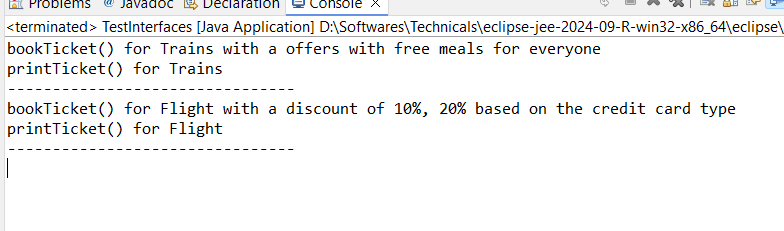
Car c = new Car(); // invalid, because Car is an abstract class

Car c; // it is not creating an object of Car, it is just a reference that can refer to all its subclass object

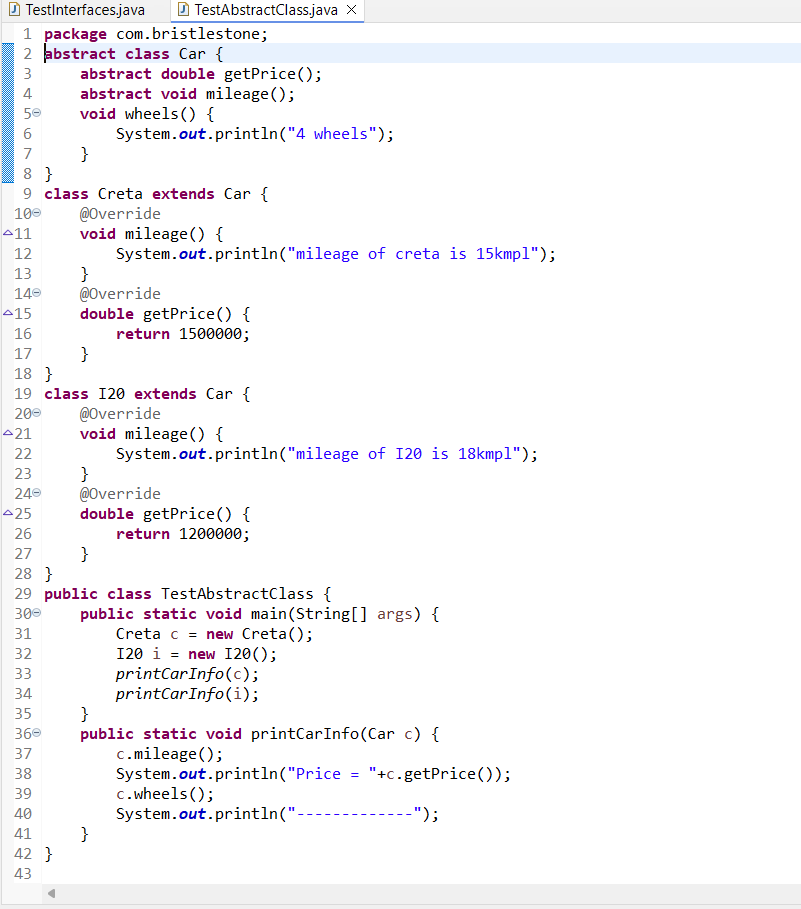
TestInterfaces.java



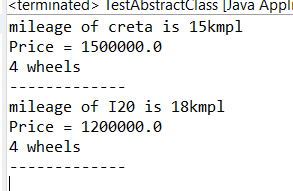
Output:



TestAbstractClass.java

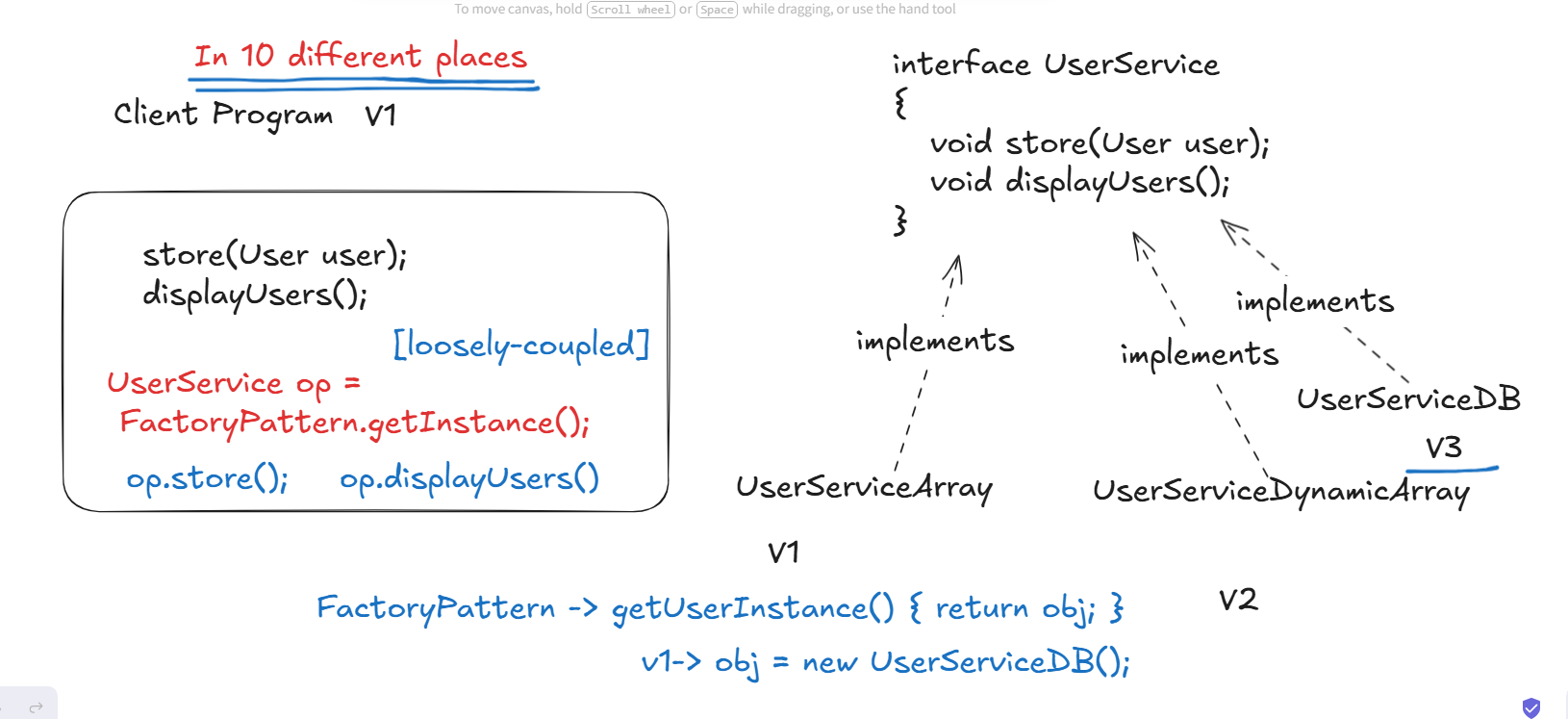


Output:



How interface can become loosely coupled at the client side

1. Interfaces are used at both client and implementations side
2. We must use design patterns that would hide the instance used by the interface at the client side i.e, using single-ton/factory design pattern
3. We must use the interface reference at the client side and get the object of the implementation
4. Since interfaces will not have any logics it’s a good practice to give interface to the client instead of giving abstract class



Exercise:

1. We will maintain the User objects in the array
2. Client -> main method -> must use the interface to invoke store & displayAll methods
3. Client -> main method -> must able to use the factory pattern to get the object of the interface implementation
4. Create an implementation for the interface and its object must be created in a factory pattern class, so that object creation is hidden

The advantage of this is client programs don’t need to change whenever the new implementations are provided for the interface, just the changes in the factory pattern will make the client program to use the new implementations, as the client program gets the object from the factory pattern class.

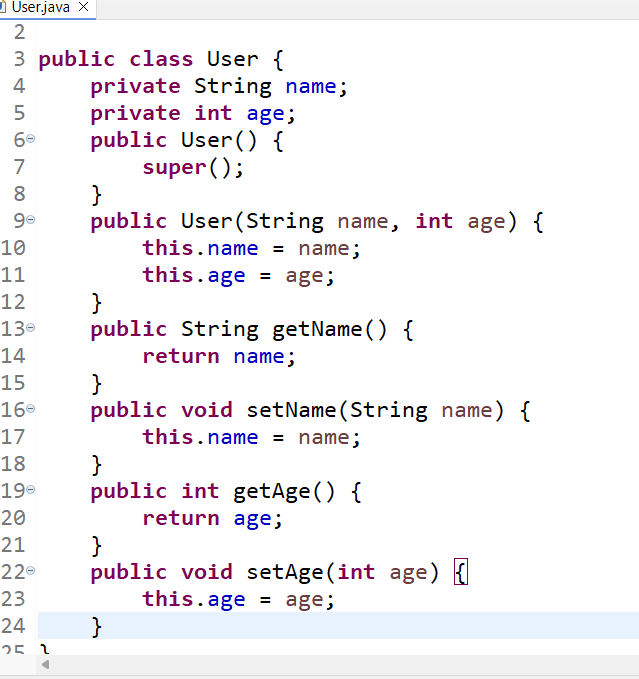
i.e.,

UserService op = new UserServiceArray(); // is tightly coupled code

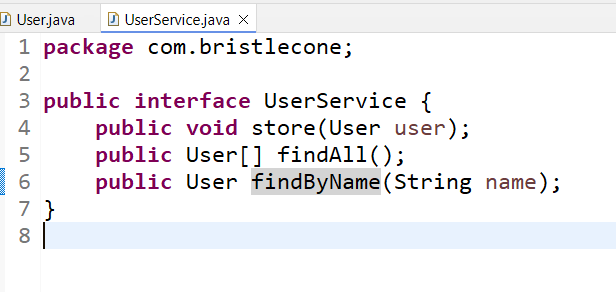
UserService op = FactoryPattern.getInstance(); // is loosely coupled code

Because FactoryPattern.getInstance() method will take care of creating the object of the interface implementation

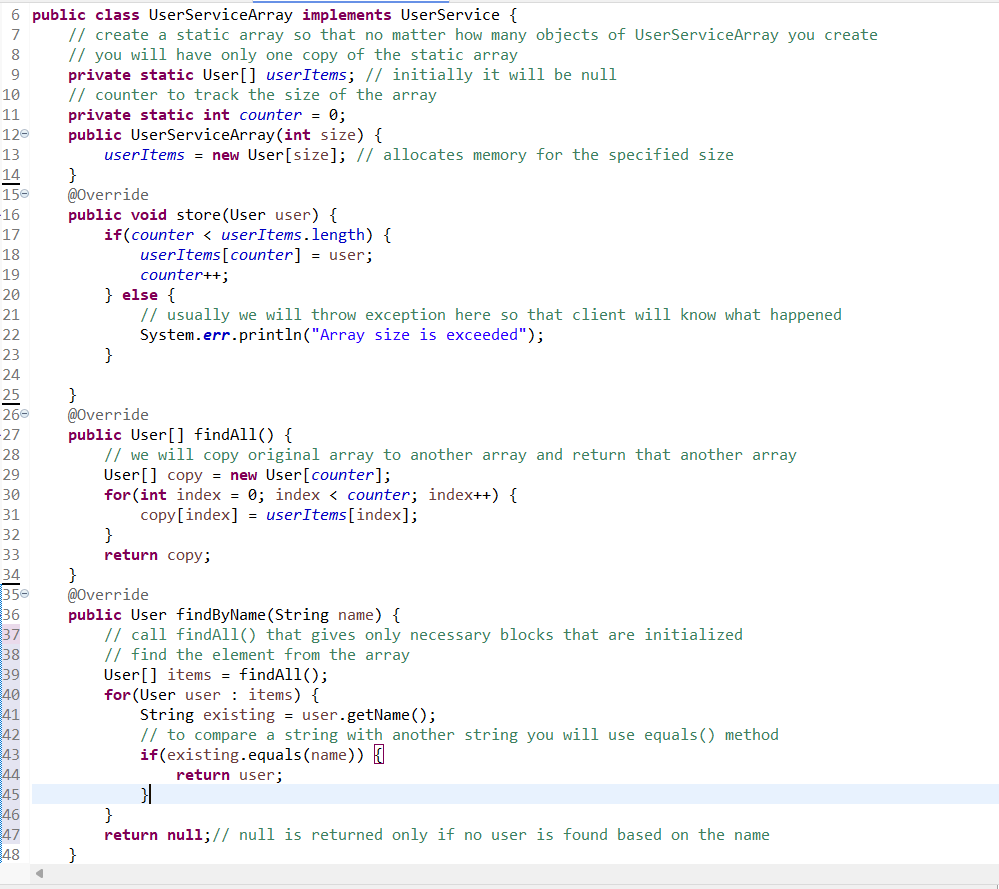
User.java



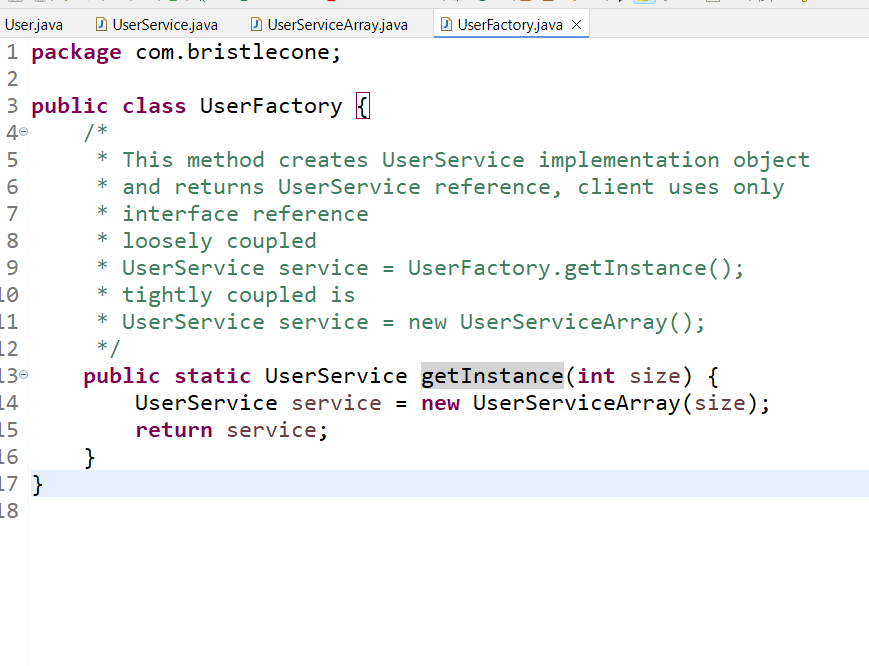
UserService.java



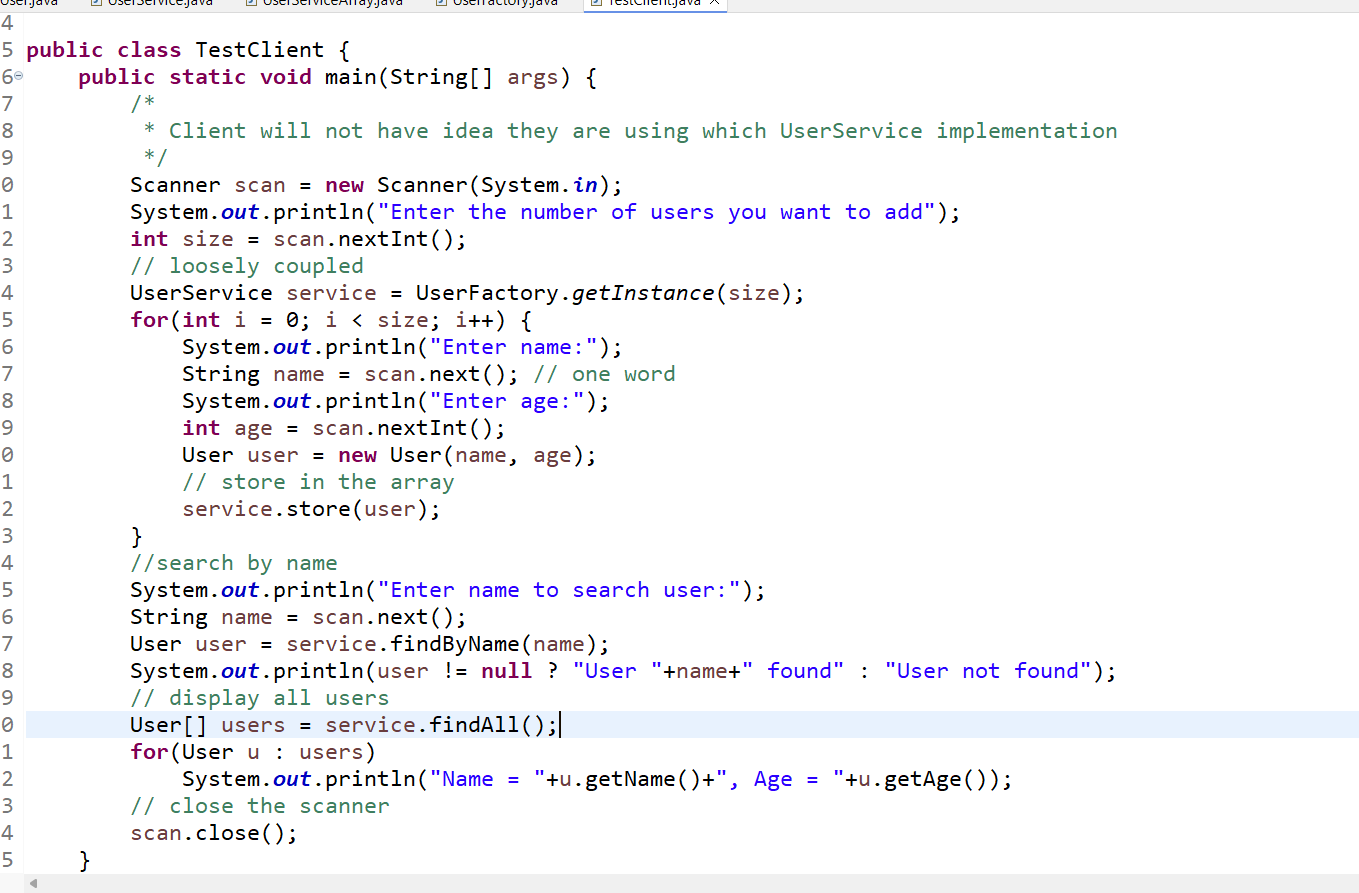
UserServiceArray.java



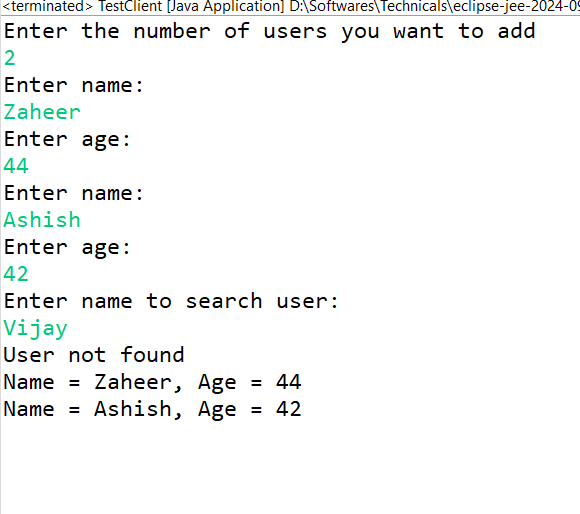
UserFactory.java



TestClient.java



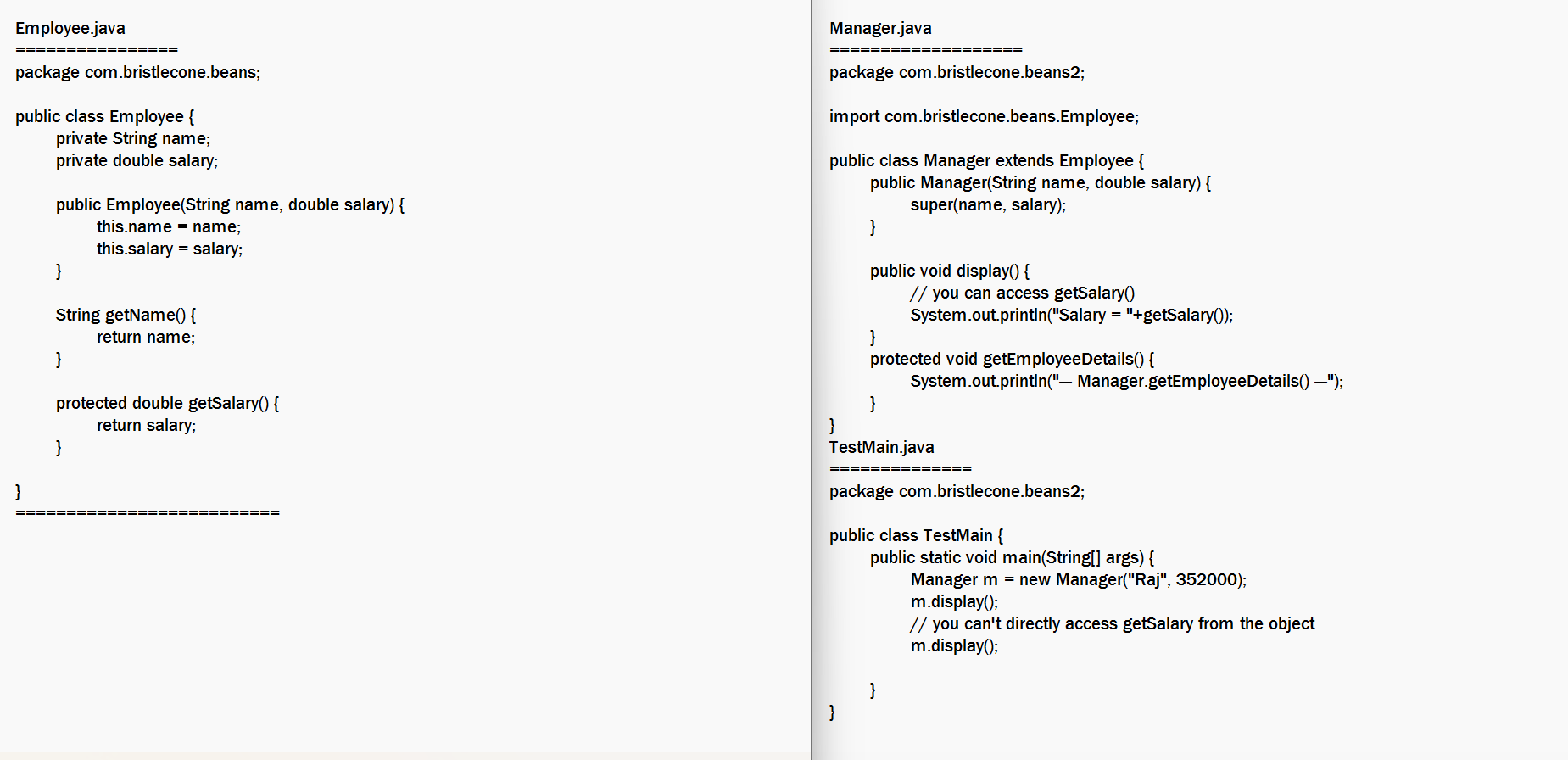
Output:



Access specifiers in Java

These are the keywords that specifies the visibility of the class members to the outsiders, there are 4 access specifier’s in java

1. private: visible within the class
2. package private (it is not a keyword): visible within the package
3. protected: visible within the package & to the subclass outside the package
4. public: visible to all the classes

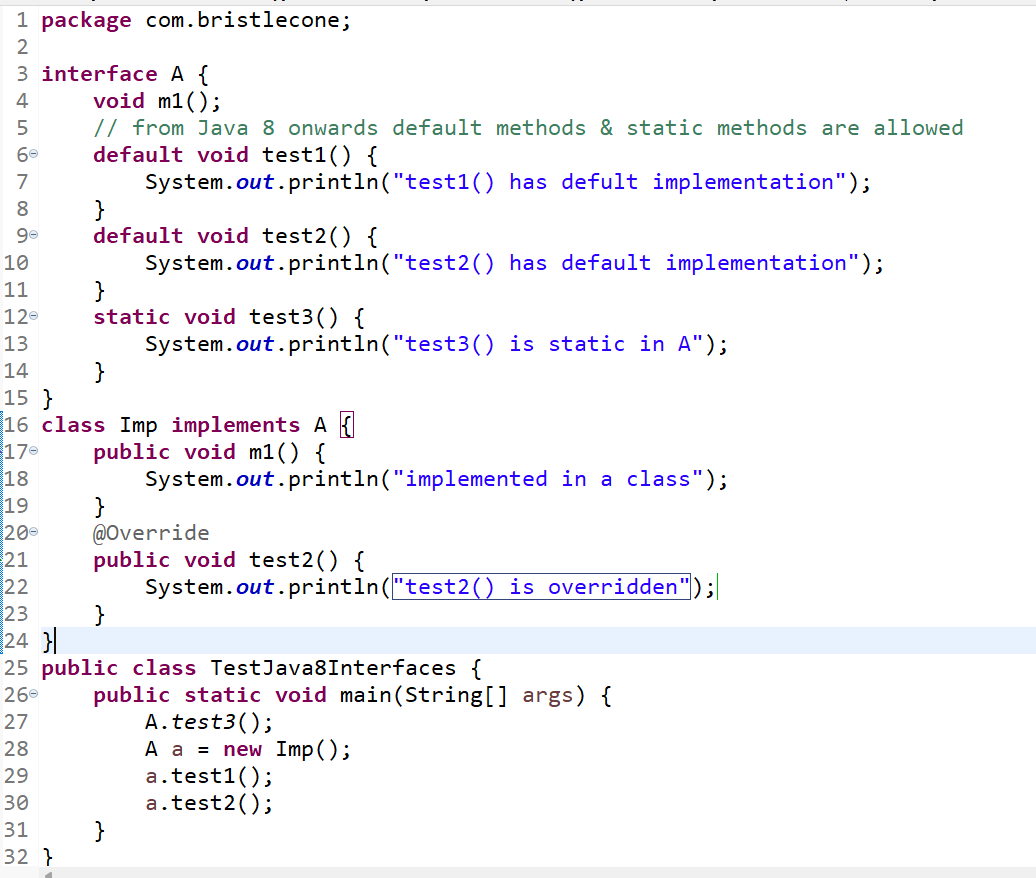


In the program, TestMain method can create Manager object, but it can’t access getSalary() because it is visible only to the subclass of Manager class, not to the TestMain which is a subclass.

Changes added to the interfaces from Java 8 version

Java added 2 features to the interface where you can have methods with body

1. default methods: These will have default implementations so that the class is not forced to override, but still class can override if it wants.
2. static methods: These will have some common utility logics, which you directly access with the interface alone, but static methods can’t be overridden.



Enums:

These are fixed set of constants which will be of enum type, they are useful to have pre-validation on the data

Instead of creating a gender variable of String and validating, its better to have fixed set of constants which are valid only if we use any one of the values in the set.

String gender = “Male”; // dynamic

if(gender.equals(“Male”) || gender.equals(“Female”) {   
 // manual validation   
}

String loanType = “CAR”;  
if(loanType.equals(“CAR”) || loanType.equals(“HOME”) || loanType.equals(“PERSONAL”)) {  
  
}

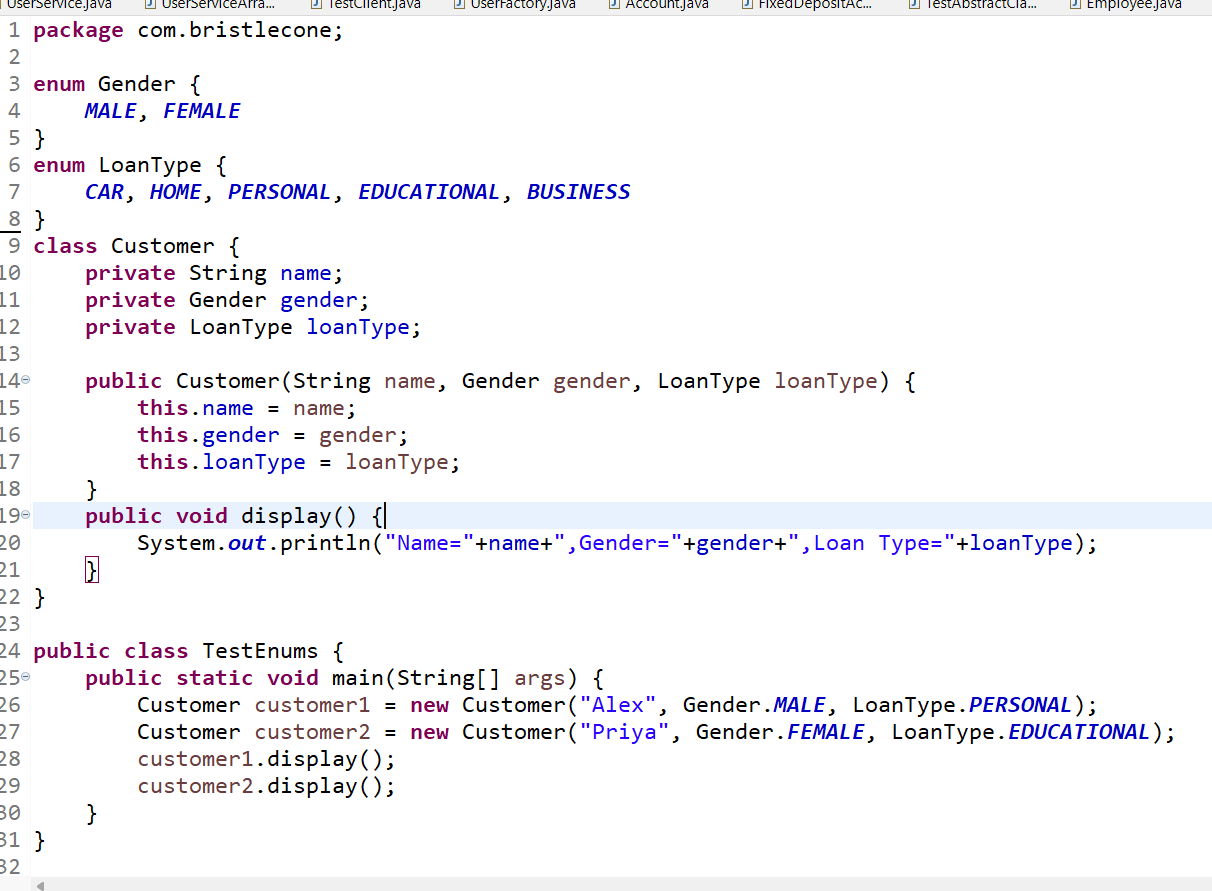
It’s a good idea to use fixed set of constants that are existing in the program itself through enums.

enum Gender {   
MALE, FEMALE  
}

enum LoanType {   
 CAR, HOME, PERSONAL, EDUCATIONAL, BUSINESS  
}

class Customer {   
 String name;   
 Gender gender; // gender = Gender.MALE or Gender.FEMALE  
 LoanType loan; // loan = LoanType.CAR or LoanType.HOME or …  
}

TestEnums.java



Static & Non Static members

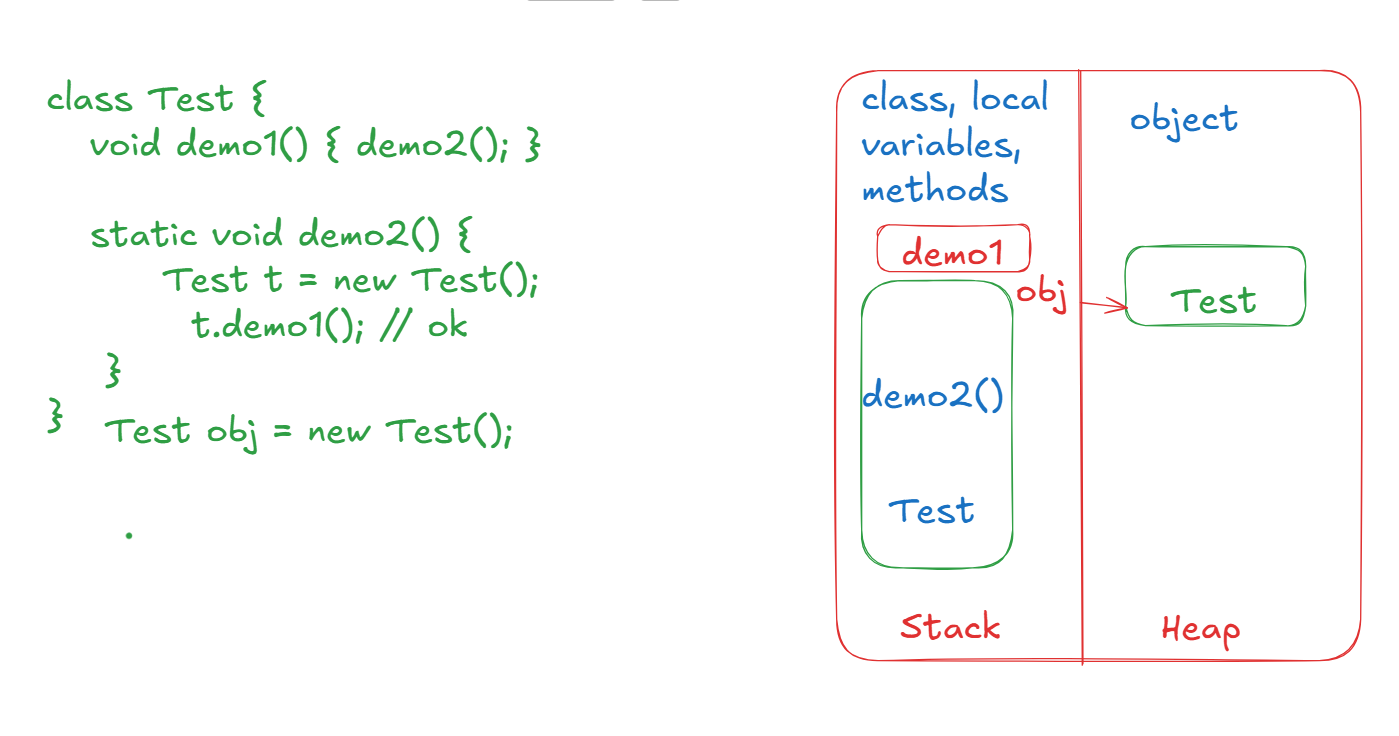
How do you access static - members?

You can use class name and access static members

How do you access non-static members?

You need to create object and access

Static methods can’t access non-static members directly, however non-static methods can access static members directly.



Can I use private, protected, public, static to the local variables?

class Demo {   
 public void test() {   
 private int count1 = 0;  
 public int count2 = 25;  
 public static int count3 = 35;  
 }

}

Answer: No, you can’t use private, protected, public, static to the local variables, because their scope is within the method.

19-02-2026

Built-in classes in Java

1. Object
2. String
3. StringBuilder
4. StringBuffer
5. Wrapper classes - Integer, Double, Character, …

Object class

It is a root class in Java it has common methods which every class must have, some of the important methods are

1. public String toString()
2. public boolean equals(Object obj)
3. public int hashCode()
4. public void wait()
5. public void notify()
6. public notifyAll()

toString(): It is automatically called when you print an object, it returns object information in String format, the default implementation of toString() is returning the hexadecimal value of the object’s hash code

hashCode(): It returns the memory address of an object in int format, this is useful when you add the objects in the HashSet or HashMap datastructure to identify the hash buckets

equals(Object obj): It compares two objects address by default and returns true if they are same else returns false, even equals() are useful when you maintain objects in HashSet or HashMap

Note: We can override all the above 3 methods as per our requirement.

Create Address class with state, city, pin and use Address object in the Employee class so that it will have id, name, salary and address(Address class type) in toString() of Employee you must return id, name, salary and address, but the S.o.p(emp1) & S.o.p(emp2) should also print id, name, salary, state, city and pin code.

Hint: In Employee class toString you must write

return "Id = "+id+", Name = "+name+", Salary = "+salary+”, Address = “+address;

Expected Output:

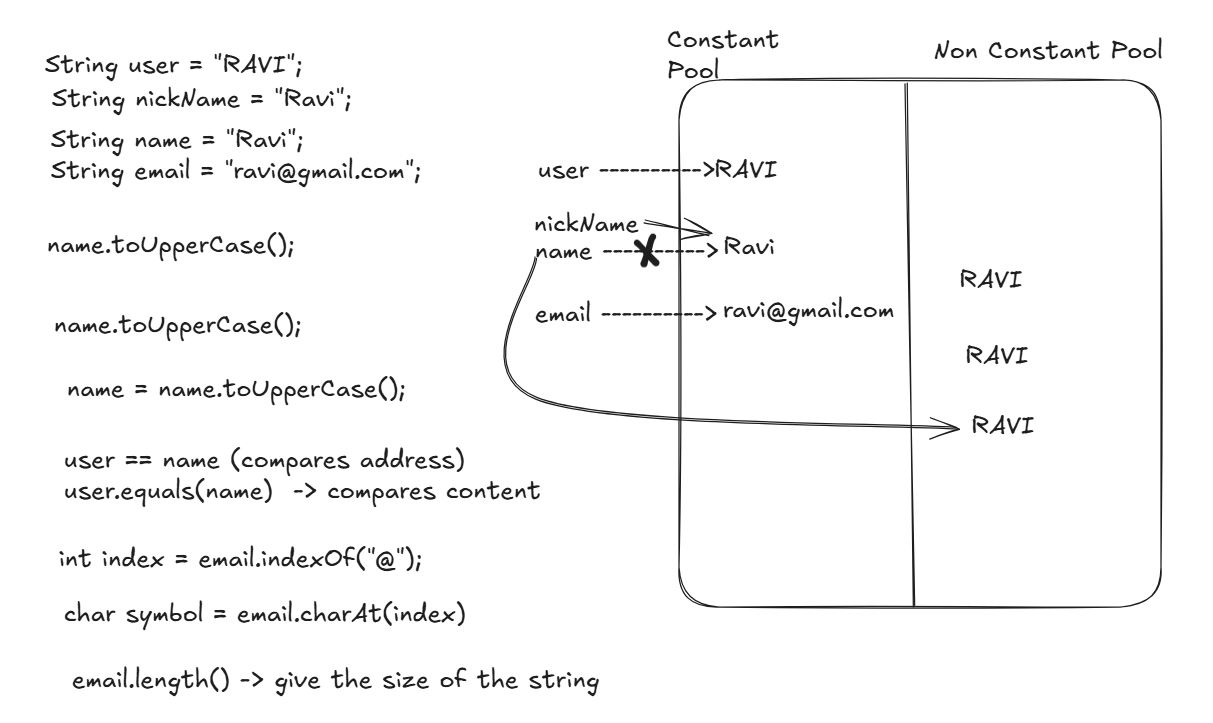
Id = 101, Name = Ramesh, Salary = 50000.0, Adress = State = KA, City = BLR, Pin = 560001

String class

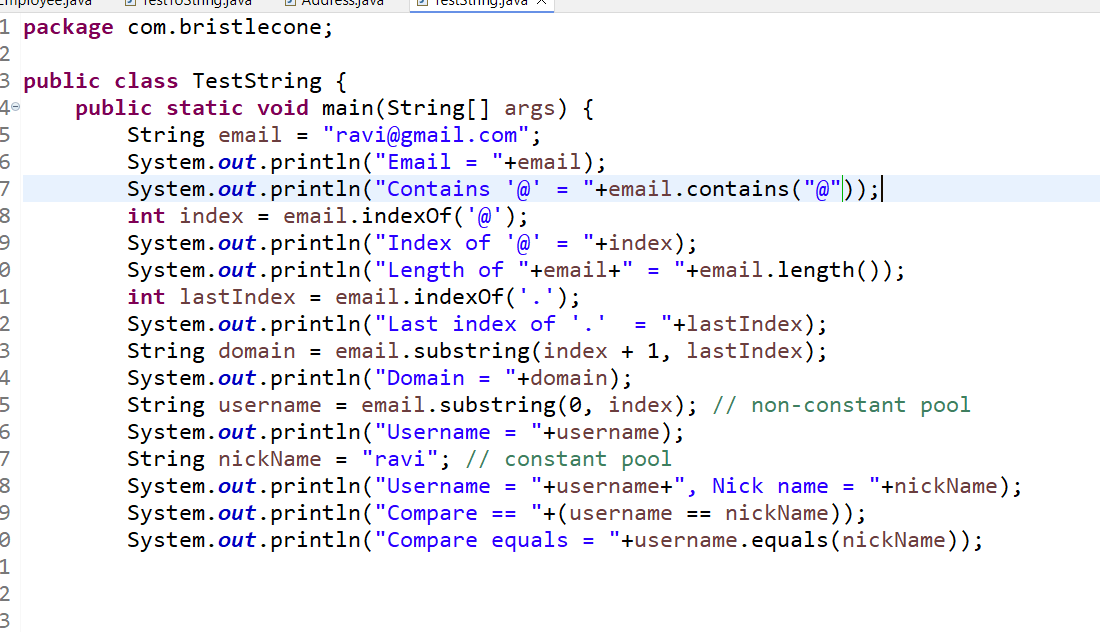
It creates an immutable string object, which can’t be modified once created, however when you modify it creates a new string instead of changing the existing string.

String uses two types of pool

1. String constant pool: Stores the strings that you create with = operator or String in some place where you directly pass without using any variable, these are created to reuse the string, in constant pool only one string object of that value can exist.
2. String non-constant pool: It can have duplicate string content, it will be used when you manipulate the strings.



TestString.java



Activity:

Use a text that will have some extra spaces in the beginning and end, using trim() you can remove leading & trailing spaces, add some text to the string and print the message

String message = “ Java is powerful and java is easy “;

1. Remove extra spaces
2. Convert to upper case
3. Check if message contains java
4. Find the index of first occurrence of java
5. Extract first word and append text “ - LEARNING MODE”
6. Print the message as - JAVA - LEARNING MODE

StringBuffer / String Builder: These creates a String that is mutable, both the classes have same methods but they have some differences in the feature

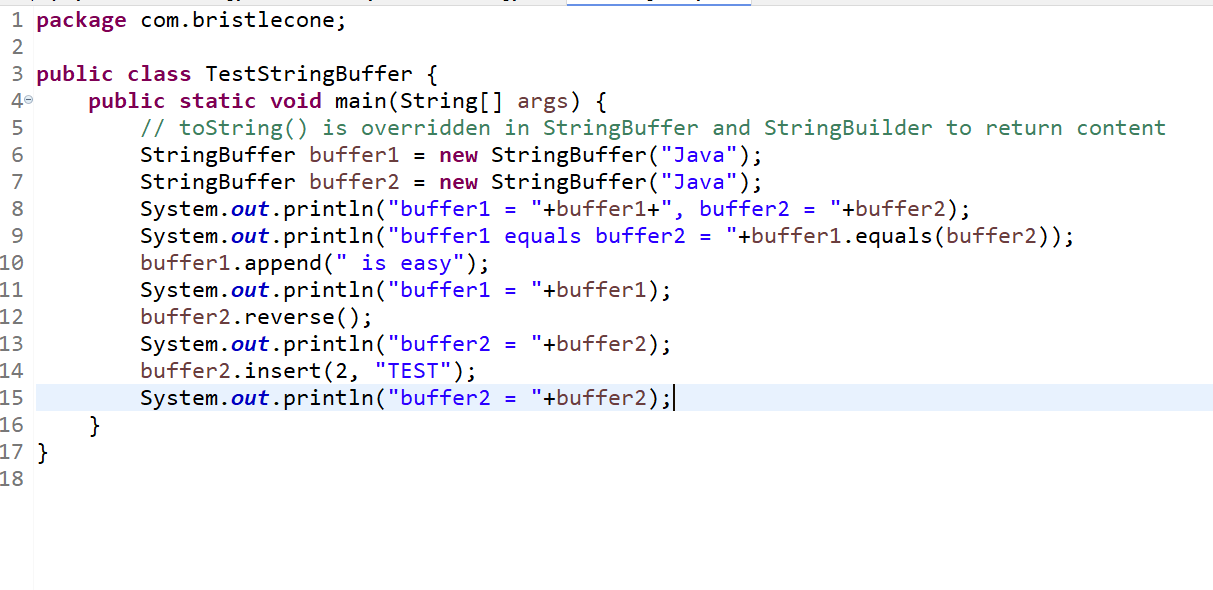
StringBuffer vs StringBuilder

|  |  |
| --- | --- |
| StringBuffer | StringBuilder |
| This is a legacy class | This is an improved version of StringBuffer and it released in Java 5 |
| This has only synchronized methods, which are thread-safe, when concurrent threads try to modify | This has non-synchronized methods, which are not thread-safe for concurrent modify operations |

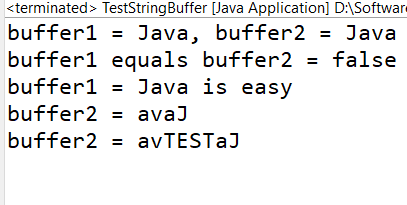
Both have same methods that help to perform different type of string manipulation, but they are mutable

* append(): To add a string to the existing string
* reverse(): To reverse a string
* delete(): To delete some characters from start to end
* insert(): To insert some characters from start to end

Note: You can’t use equals() method of StringBuffer / StringBuilder to compare the content, because they are not overridden from Object class to compare the content



Output:



Wrapper classes

These classes are provided for every primitive datatypes so that you can perform extra operations other than arithmetic operations like converting from string to number, comparing two values, finding the max or min values and so on.

Below are the wrapper classes defined for each primitives

|  |  |
| --- | --- |
| Primitive datatypes | Wrapper classes |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

Converting String to int, double, long and so on, there are some parse methods provided which can convert string to numbers

int num1 = Integer.parseInt(“25”);

double num2 = Double.parseDouble(“35.0”);

long num3 = Long.parseLong(“425”);

boolean bool = Boolean.parseBoolean(“true”);

Mobile app => if you enter 50000 then it will be treated as “50000” in text form, hence you need to convert that 50000 in string form to int, long or other types

We have static methods in each wrapper classes that perform some operations like

Integer.compare(20, 30);  
Double.compare(20.0, 30.0);

The compare() method takes 2 input and returns +ve or 0 or -ve value

These comparison can be used mainly in sorting algorithms to sort the numbers

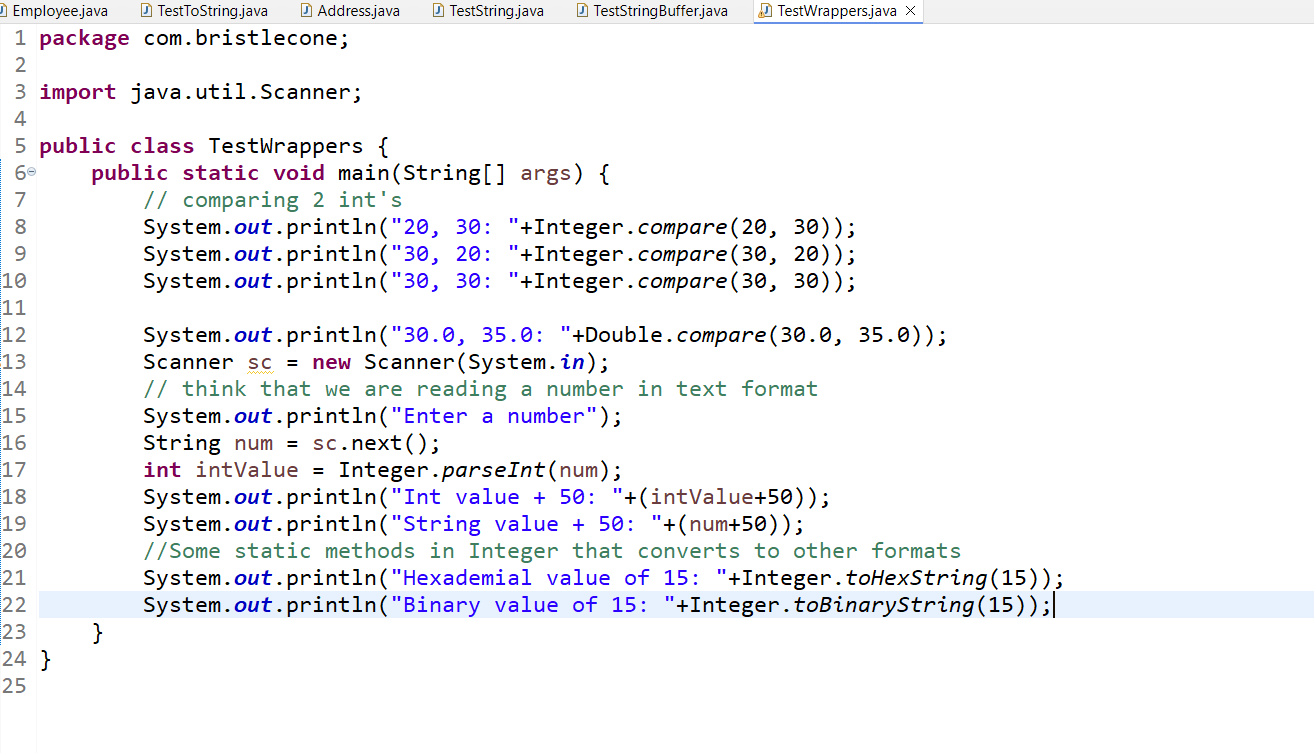
Ex: Integer.compare(20, 30): -ve

Ex: Integer.compare(20, 20): 0

Ex: Integer.compare(30, 20): +ve

Best example where these comparison is used is in Sorting collection of data

TestWrappers.java



Exception Handling

Exceptions are runtime errors, they will crash the applications if not handled, to safely terminate the program we must handle the exceptions.

There 5 keywords used in Exception Handling mechanism

1. try
2. catch
3. finally
4. throws
5. throw

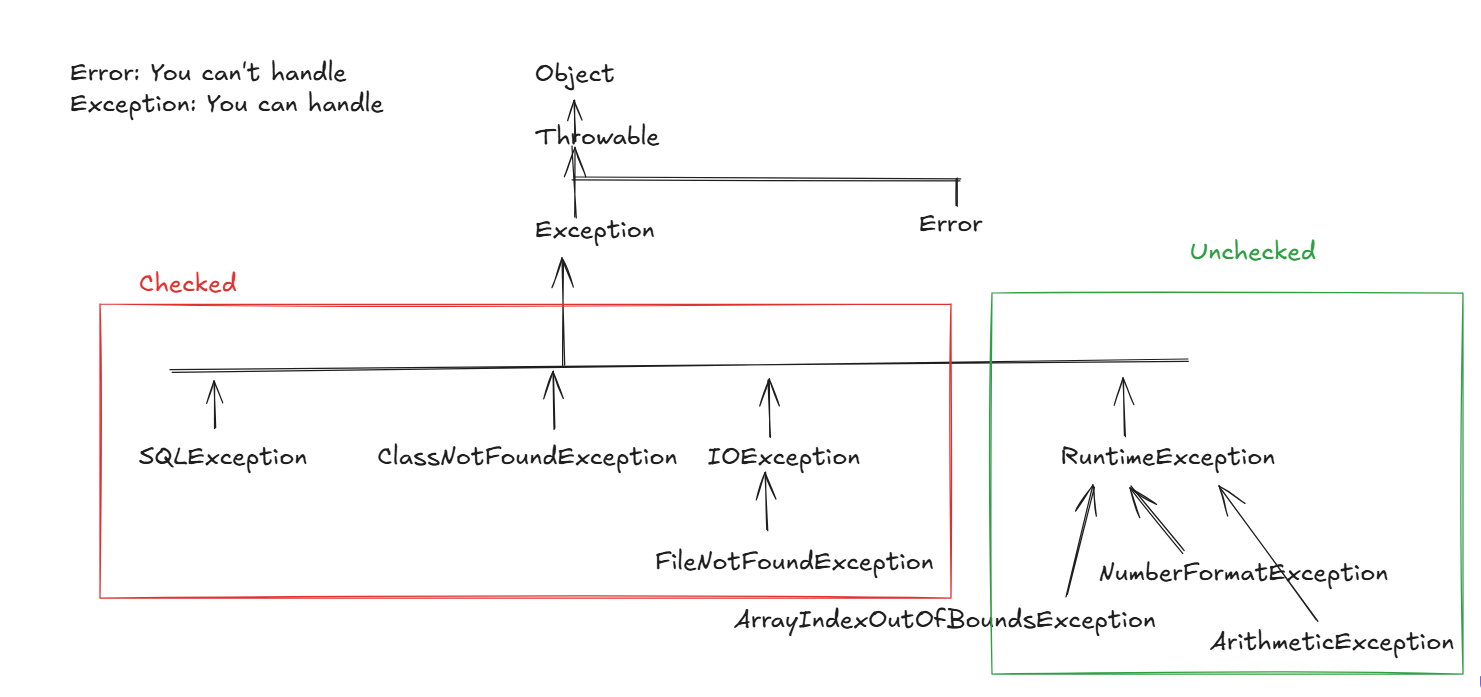
try block: Here we write those codes which may generate an exception, it can be performing DB operations, file io operations, accessing arrays and so on.

catch block: Here it acts like a handler, which must come after the try block, you can have any number of catch blocks after try

finally block: It is executed always whether exception occurs or doesn’t occur, you can have logics that you want to run anyways like closing resources

Note: finally block is an optional block, it can come after try or after all the catch blocks.

Built in exceptions



All the subclasses of Exception except RuntimException falls into checked exception, others fall into Unchecked

Checked: These exceptions you must be handled mandatorily, else compiler gives error, if there are exceptions you will always have an handler

Unchecked: These exceptions are not mandatory to handle, compiler doesn’t give error if you don’t handle, however if exception occurs still program terminates abnormally

Why compiler doesn’t force to handle unchecked exception?

Because these exceptions you can avoid within the application with the code itself

ex: ArithmeticException occurs when you divide any number by 0, we can avoid this with the code itself, ArrayIndexOutOfBoundsException occurs when you access an index which is not available, this also you can avoid with the code

But checked exceptions are not in application hand

It can’t be avoided in the code, because exceptions like SQLException, IOException, ClassNotFoundException occur when you access some external resource, these resource may not be available temporarily, in that case application can’t do anything.

throws: It is used to propagate the exceptions from the method to the caller, when a method doesn’t want to handle an exception or doesn’t know what to do when exception occurs, this is used when caller knows how to handle the exception, it is written in a method signature.

Backend code

void insertData(arg1, arg2, arg3) throws SQLException {   
 // SQLException occurs  
}

Client code - Caller

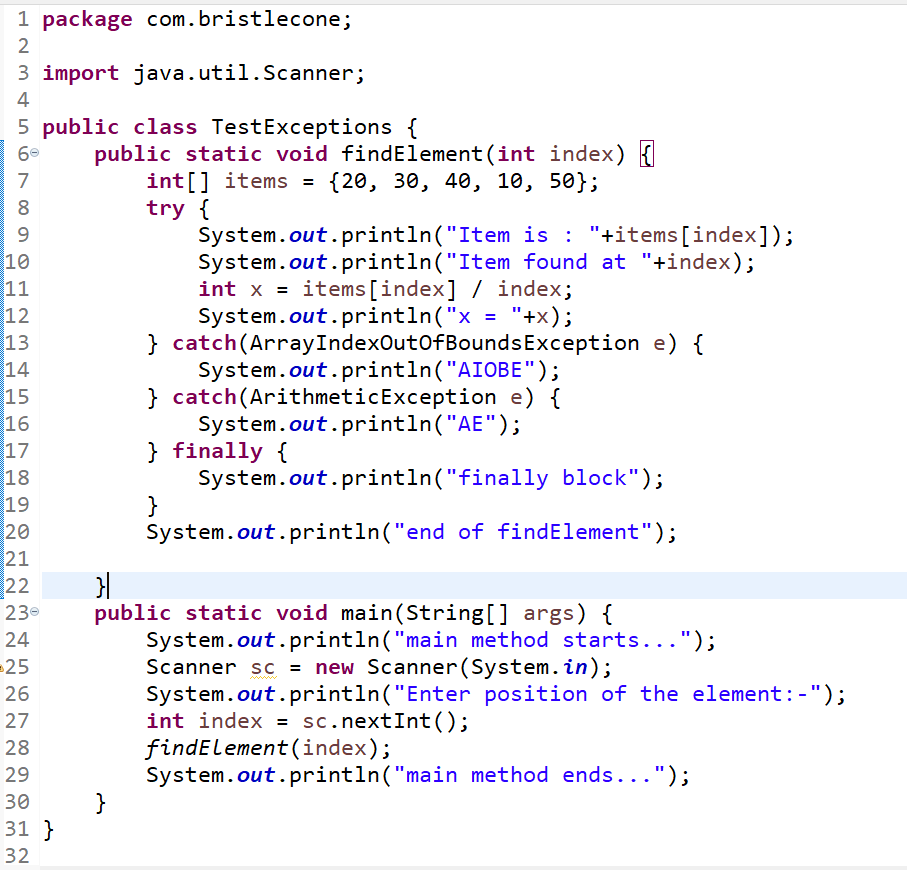
void createResource(TakingInputFromUser input) {   
 try { insertData(v1, v2, v2); } catch(SQLException e) { print err message to the user }  
}

throw: It is used to manually generate an exception in the application when certain condition is met and also you can create Custom Exception or User defined exception for your requirement and generate those custom exceptions

To create custom exception you must extend any one of the exception class like Exception or RuntimeException

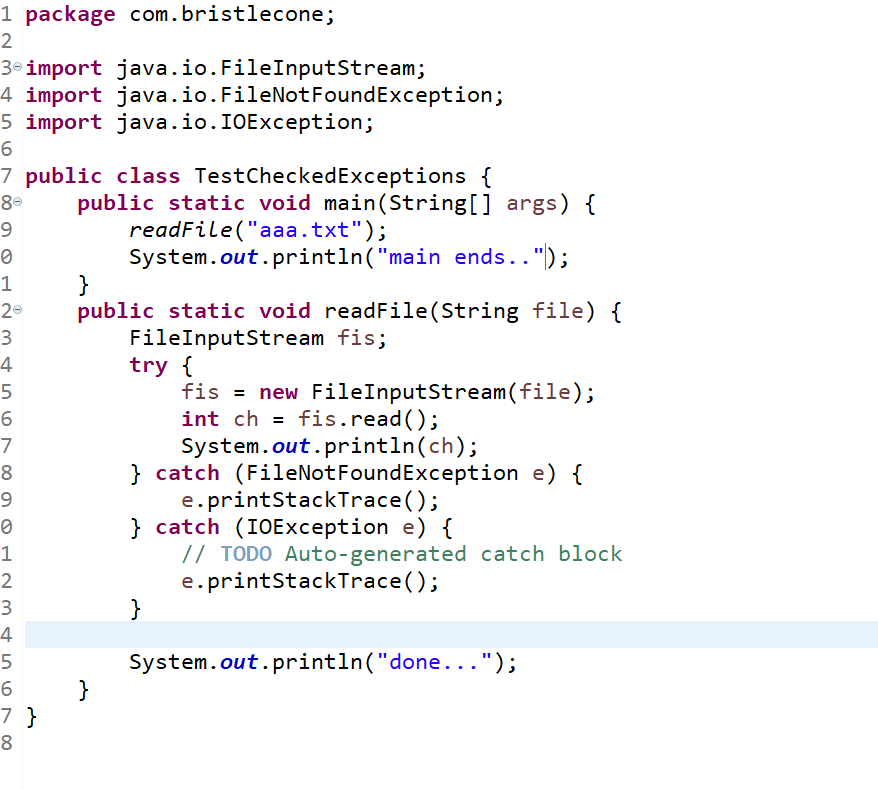
class AgeInvalidException extends Exception { … }

if(age < 18) { throw new AgeInvalidException(“age must be >= 18”); }

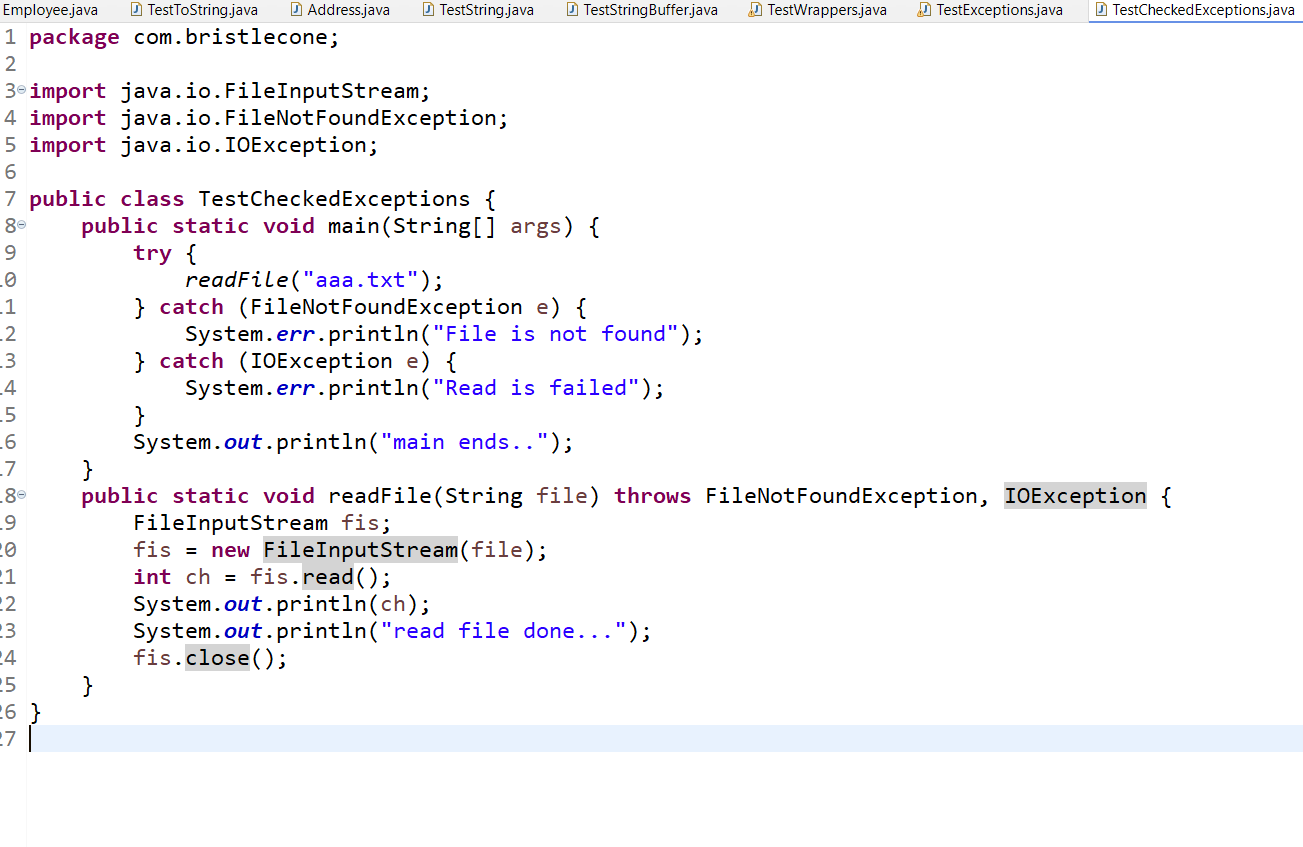


Checked & Unchecked exceptions

Unchecked exceptions are ignored by the compiler, but checked exceptions are forced to handle.



throws: It is used to propagate checked exceptions so that the caller will handle them.



21-01-2026

throws: It is used to propagate the exceptions so that a method will not handle instead it propagates to the caller so that they will know how to handle.

Note: any method that throws checked exceptions needs to be called within try-catch block, however if method throws unchecked exceptions then caller is not forced to use try-catch block

Custom Exceptions

These are the user defined exceptions which are created as per the application requirement.

How to create Custom exceptions

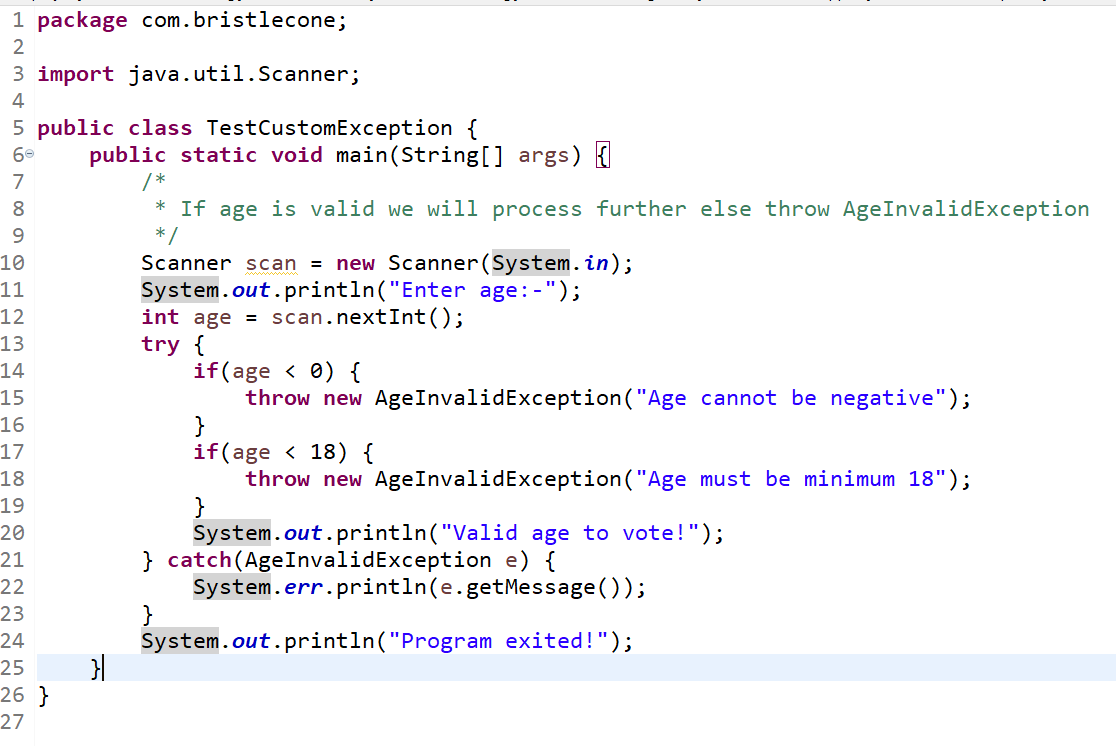
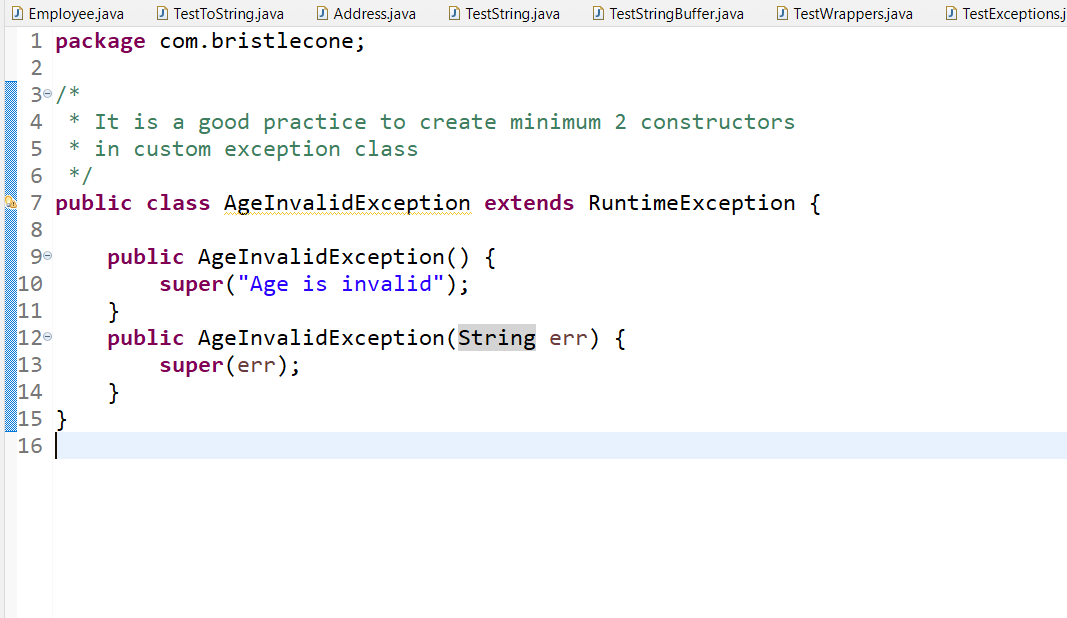
You either extend RuntimeException or Exception, if you extend RuntimeException it becomes unchecked, if you extend Exception, then it becomes checked exception.

All the Exception class has 2 constructors minimum, one is default constructor to initialize default message, another is a String argument constructor to customize the message.

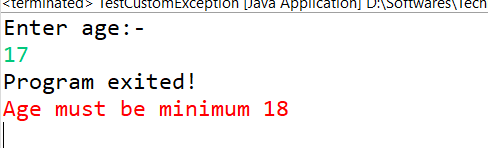
class AgeInvalidException extends RuntimeException {   
 public AgeInvalidException() { … }  
 public AgeInvalidException(String msg) { super(msg); }  
}

if(age < 0) {   
 throw new AgeInvalidException(“Age cannot be negative”);  
}

InvalidAgeException.java



Output:



Activity:

Create AccountLockedException class that is a custom exception thrown when user enters the incorrect pin 3 times continuously. If the user enters the correct PIN within three attempts, display a success message, if the user fails to enter the correct PIN after three attempts, the system should throw AccountLockedException with an appropriate message

Sample Input & Output

|  |  |
| --- | --- |
| Input | Output |
| Enter ATM PIN: 1234 | PIN verified, Access granted.  Thank you for using ATM |
| Enter ATM PIN: 1111  Enter ATM PIN: 2222  Enter ATM PIN: 1234 | Wrong PIN, attempts left: 2  Wrong PIN, attempts left: 1  PIN verified, Access granted.  Thank you for using ATM |
| Enter ATM PIN: 1111  Enter ATM PIN: 2222  Enter ATM PIN: 1142 | Wrong PIN, attempts left: 2  Wrong PIN, attempts left: 1  Wrong PIN, attempts left: 0  Account locked due to 3 wrong PIN attempts  Thank you for using ATM |

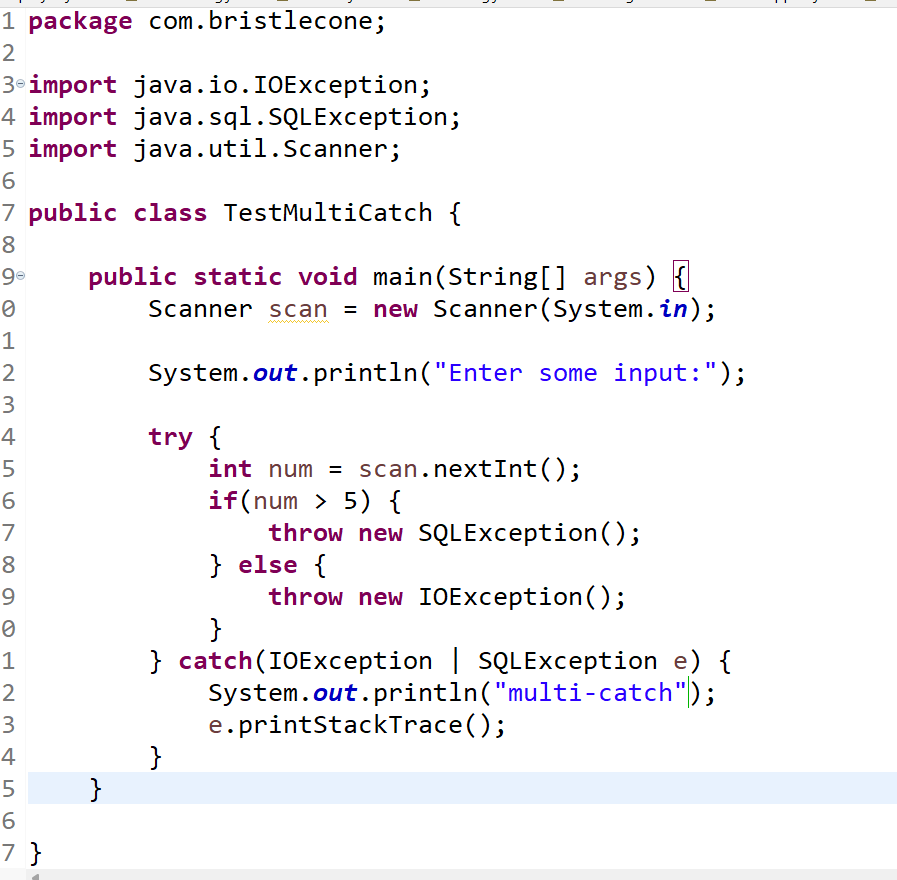
Rules of throws clause while overriding

* it is optional for an overridden method to use throws clause if super class method uses throws of checked exceptions
* An overridden method must not use throws clause of checked exception if super class method isn’t throwing any checked exception
* An overridden method must use throws of same exception class what super class method is using or overridden must use subclass of that exception in the throws

Multi-catch statements

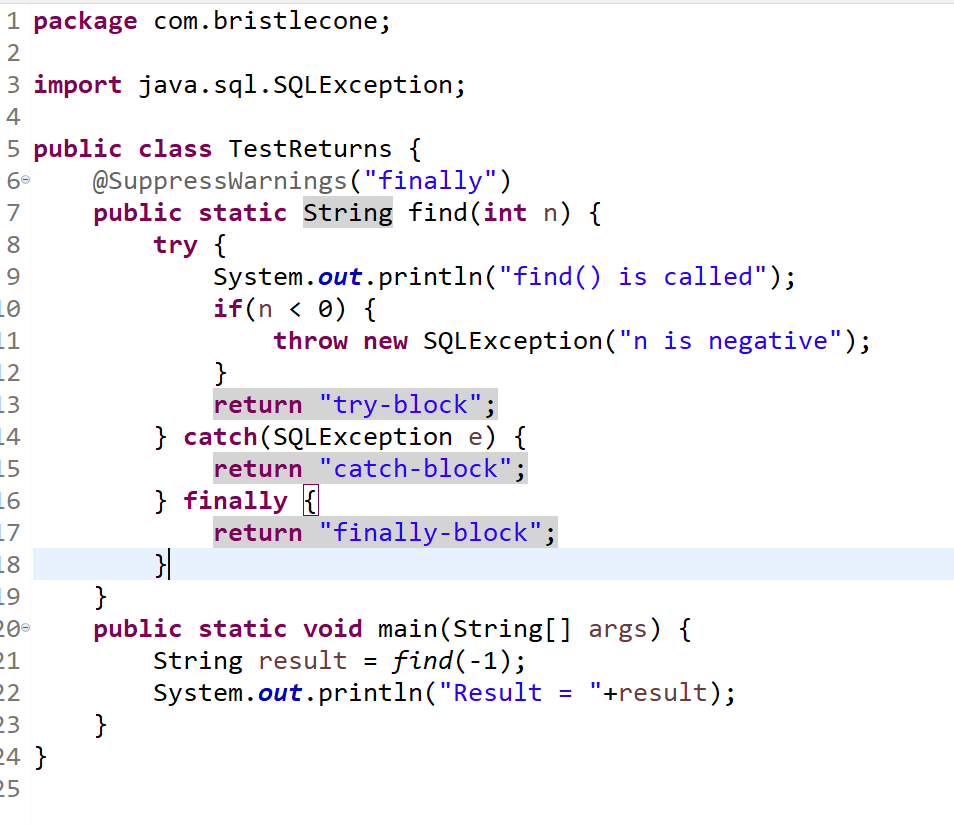
A single catch can handle one or more exceptions.

try {   
  
} catch(SQLException | IOException e) {   
 e.printStackTrace();  
}

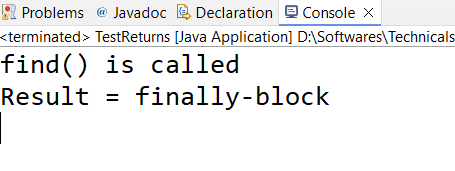


Valid try-catch-finally combinations

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



Output:



Modern syntax of try

Java has added a try with resource closing feature, where some resources created inside the is closed once the try block completes, this feature is added to avoid memory leaks.

ex:

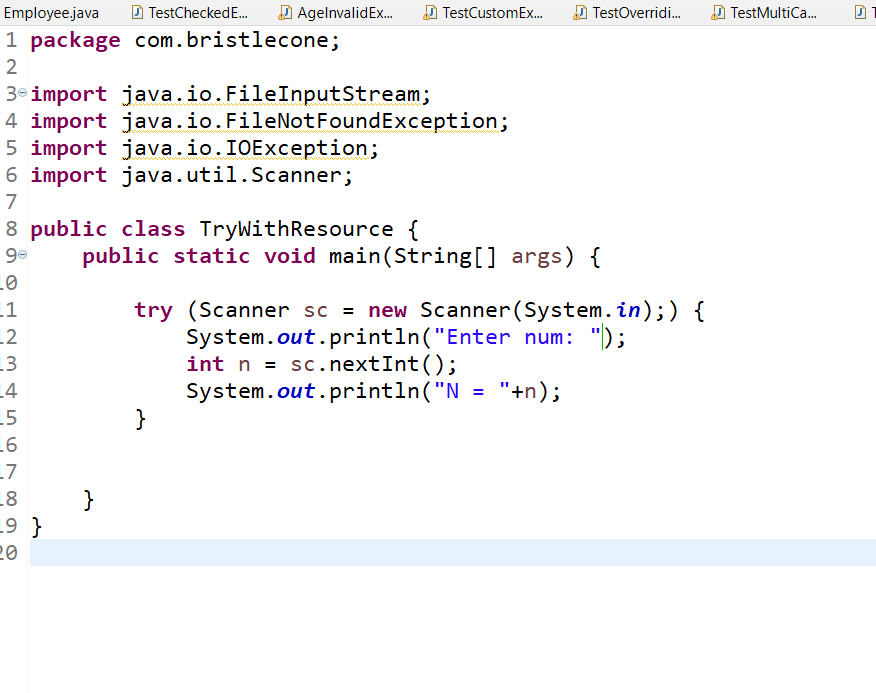
try {   
 Read a file  
} catch( … ) {   
 Handle an exception   
} finally {

closing resources  
}

The syntax for try-with resource close is

try ( Resource r = new Resource() ) {   
} catch(..) {   
}

Here you don’t have to close the resources in the finally, they are auto-closed, but Resources are related to the objects that access files, databases, system input, you can’t create objects or data inside the try ( ) resource statement



Collection Framework

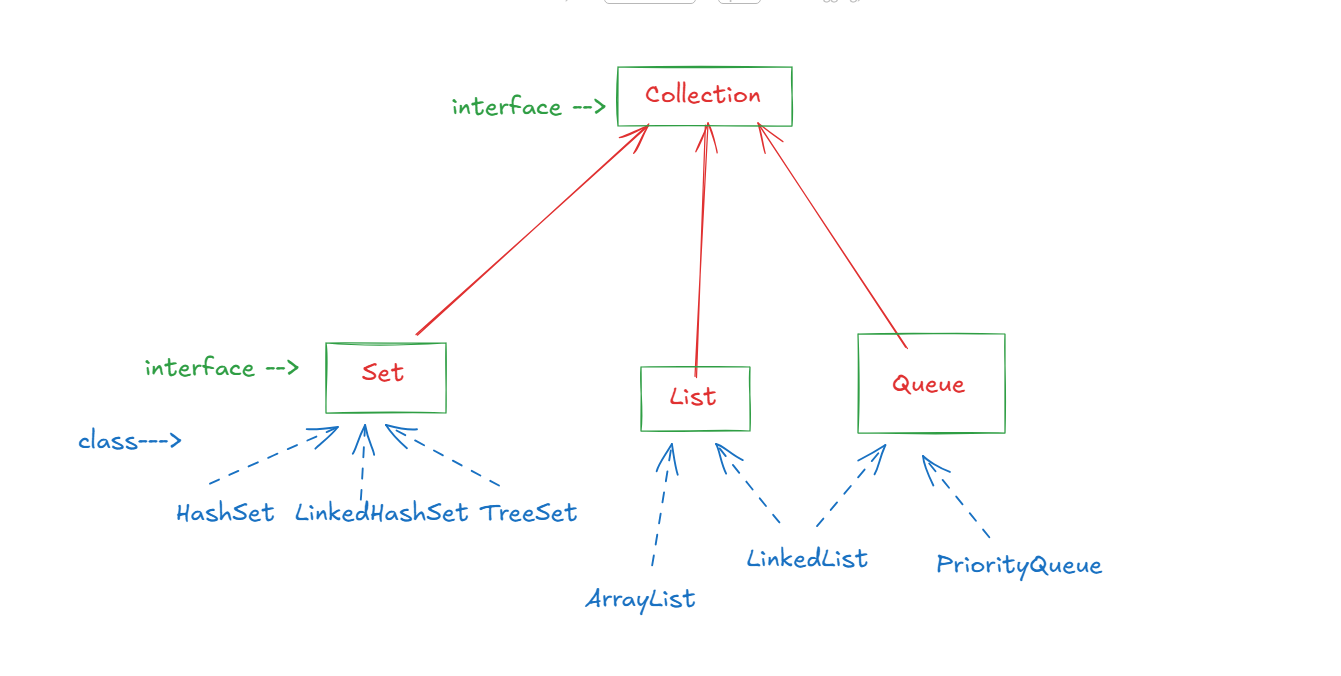
It provides various data structures to maintain the data & it is dynamic in nature, at runtime it can grow or shrink its size when you add or remove elements.

Disadvantages of arrays

1. You can store only fixed types of data
2. Array size is fixed
3. Array doesn’t have its own built-in methods to manipulate the elements which means searching the data, finding the array is empty or not, clearing all the data at one go, adding the elements / updating the elements using index / removing the elements using index
4. It always maintains the data in sequential order, but if we want to change the order we must write the logics

Collection Framework provides many interfaces & implementations to those interfaces with many built-in methods

Collection Framework has a root interface called Collection which is divided into 3 types.



Collection has methods that are available in all the implementations

add(T), remove(T), iterator(), isEmpty(), clear(), size(), contains()

Here `T` stands for any types like Integer, String, Double, Float, Custom types like Student, Bank, Employee and etc, in Collection everything is stored as an object, even primitives are converted to objects automatically

add(T): This is used to add the elements in the collection

remove(T): This is used to remove the elements from the collection

iterator(): This returns an Iterator object that helps to iterate over the collection with its method

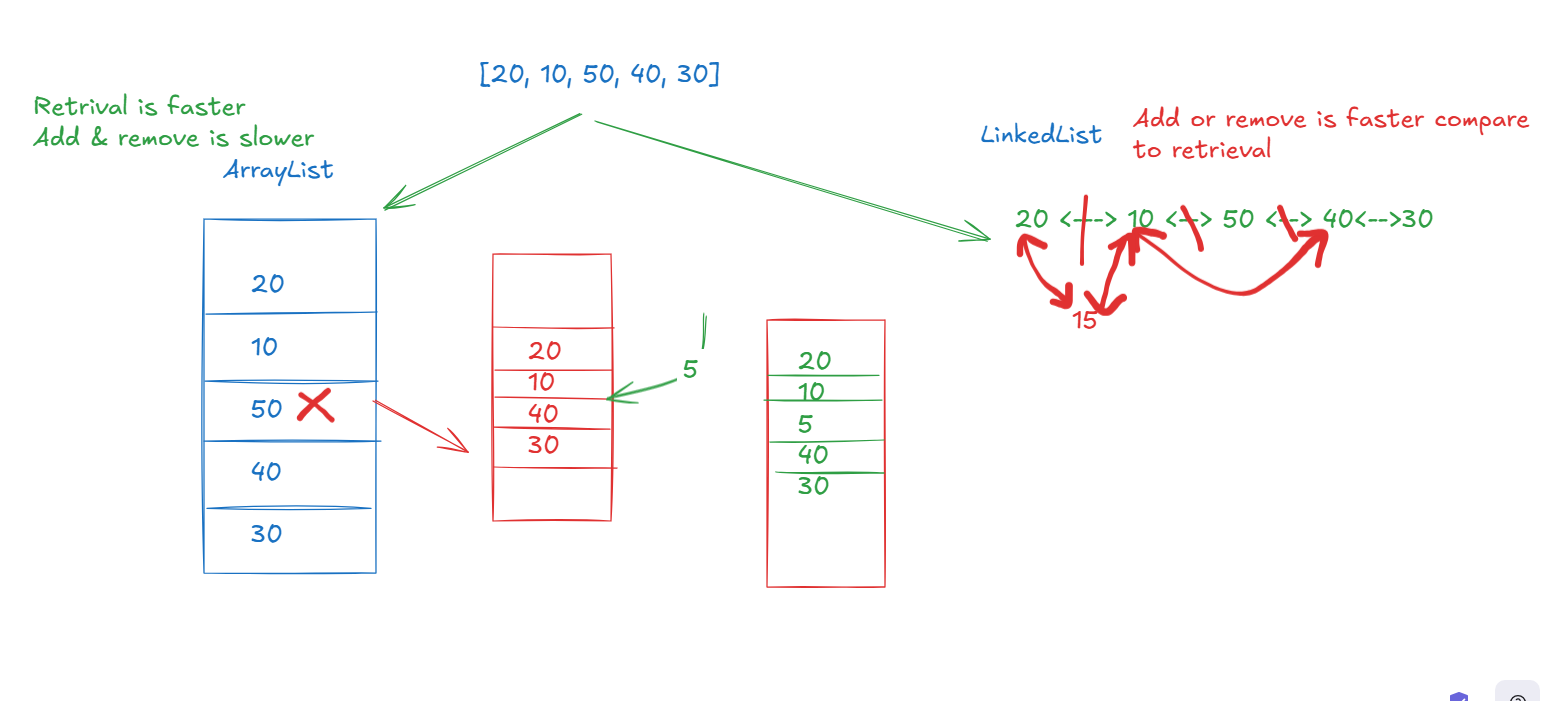
isEmpty(): returns true if collection is empty else false

clear(): It removes all the elements from the collection.

The three interfaces define how collection must maintain the data

1. Set: stores unique elements ( no duplicates )
   1. HashSet: It doesn’t maintain the order but uses hashing to read the data faster
   2. LinkedHashSet: It maintains the insertion order
   3. TreeSet: It maintains the sorted order
2. List: allows index-based access and preserves insertion order, it allows duplicates, it provides some extra methods to maintain the data based on index like: add(int, T), remove(int), get(int), set(int, T), it has 2 implementations like:-
   1. ArrayList: Stores the elements in contiguous memory address
   2. LinkedList: Stores the elements in non-contiguous memory address
3. Queue: Stores elements for processing, following FIFO order, where insertion & removal happens at different ends, it has some extra methods to process like poll(), peek()
   1. LinkedList: Processes elements using FIFO order, it can also remove the elements from different end like removeFirst(), removeLast(), addFirst(), addLast()
   2. PriorityQueue: Processes elements using sorted order, it removes the elements in sorting order

ArrayList vs LinkedList



How to create instances of these collections

Collection is used with Generics

What are generics

Generics add types to the Collection so that you will know what type of data collection will maintain, without generics Collection can store multiple types of data which is not recommended, because maintaining multiple type of elements slows down the retrieval

Collection<T>, List<T>, Set<T>, Queue<T>, LinkedList<T>, ArrayList<T>

Here the ‘T’ stands for class type, which could be any wrapper class, String, StringBuffer, custom class

ArrayList<Integer> arrays = new ArrayList<Integer>(); // now you can only store int / Integer type of data in the ArrayList

ArrayList<String> strings = new ArrayList<String>(); // only stores strings

LinkedList<Employee> emps = new LinkedList<Employee>(); // only stores employee objects

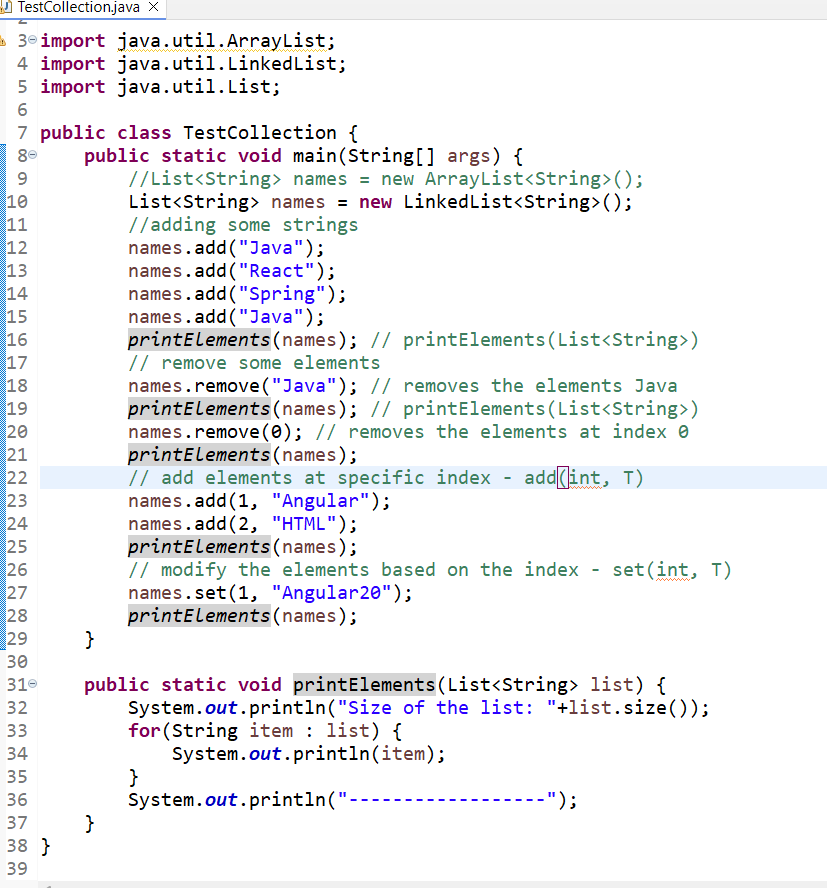
This is not recommended

ArrayList arrays = new ArrayList(); // now you can store any type of object

Following are the methods you can use to maintain the data in any collection

1. add(T)
2. remove(T)
3. iterator()
4. isEmpty()
5. contains()
6. clear()

Note: In ArrayList / LinkedList we have index based methods like add(int, T), remove(int), get(int), set(int, T), these will perform CRUD operations based on the index



Activity: Using ArrayList<String> create a TODO tasks, where you must repeatedly ask user to perform add, remove, update, delete and display tasks: Create a menu options where 1: Add tasks, 2: Remove tasks based on the index 3: Update the tasks as [COMPLETED] based on the index, 4: display all tasks, 5: Exit that will print thank you and terminate the program

Sample Input

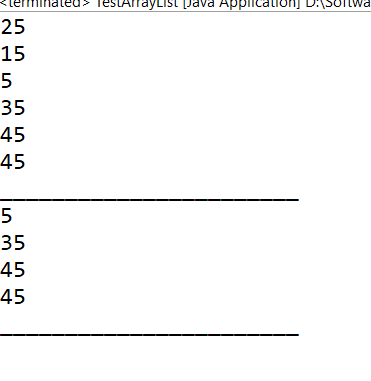
Enter tasks: 1: Add, 2: Remove 3: Update 4: Display 5: Exit  
1  
Meeting  
Enter tasks: 1: Add, 2: Remove 3: Update 4: Display 5: Exit  
1  
Sports  
Enter tasks: 1: Add, 2: Remove 3: Update 4: Display 5: Exit  
3  
Enter the index to update the tasks   
1   
-> print the message completed or error message if index is wrong

Enter tasks: 1: Add, 2: Remove 3: Update 4: Display 5: Exit  
4  
Tasks are:  
Meeting  
Sports [ COMPLETED]

Maintaining the Integer elements in the List, since List provides index based access the elements you maintain and the index will be of integer, we must be careful while doing any operations.



Output:



Iterator<E>: It is used to iterate collection, it can be applied on any Collection implementations, it has 3 methods

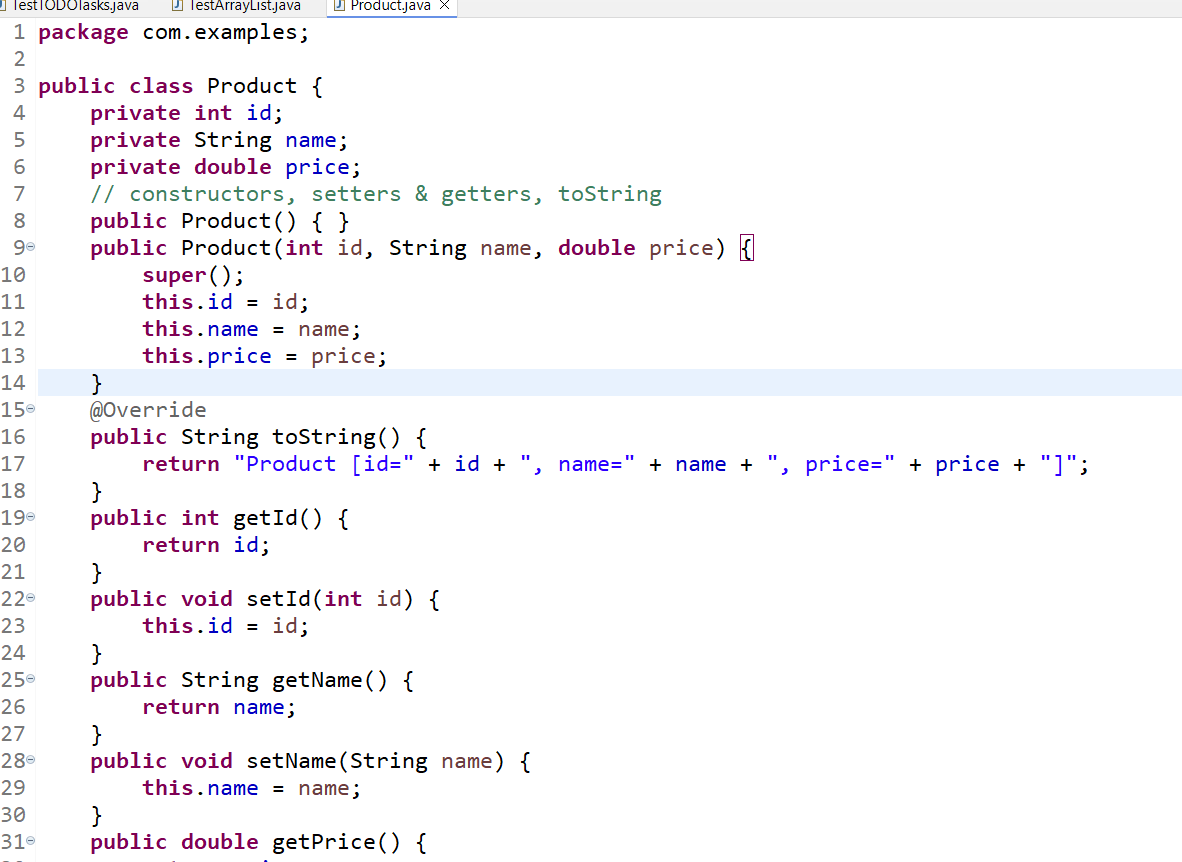
1. boolean hasNext()
2. E next()
3. void remove()

Adding complex types to the List

Product -> id, name, price

List<Product> -> add, remove, display, clear

Product.java

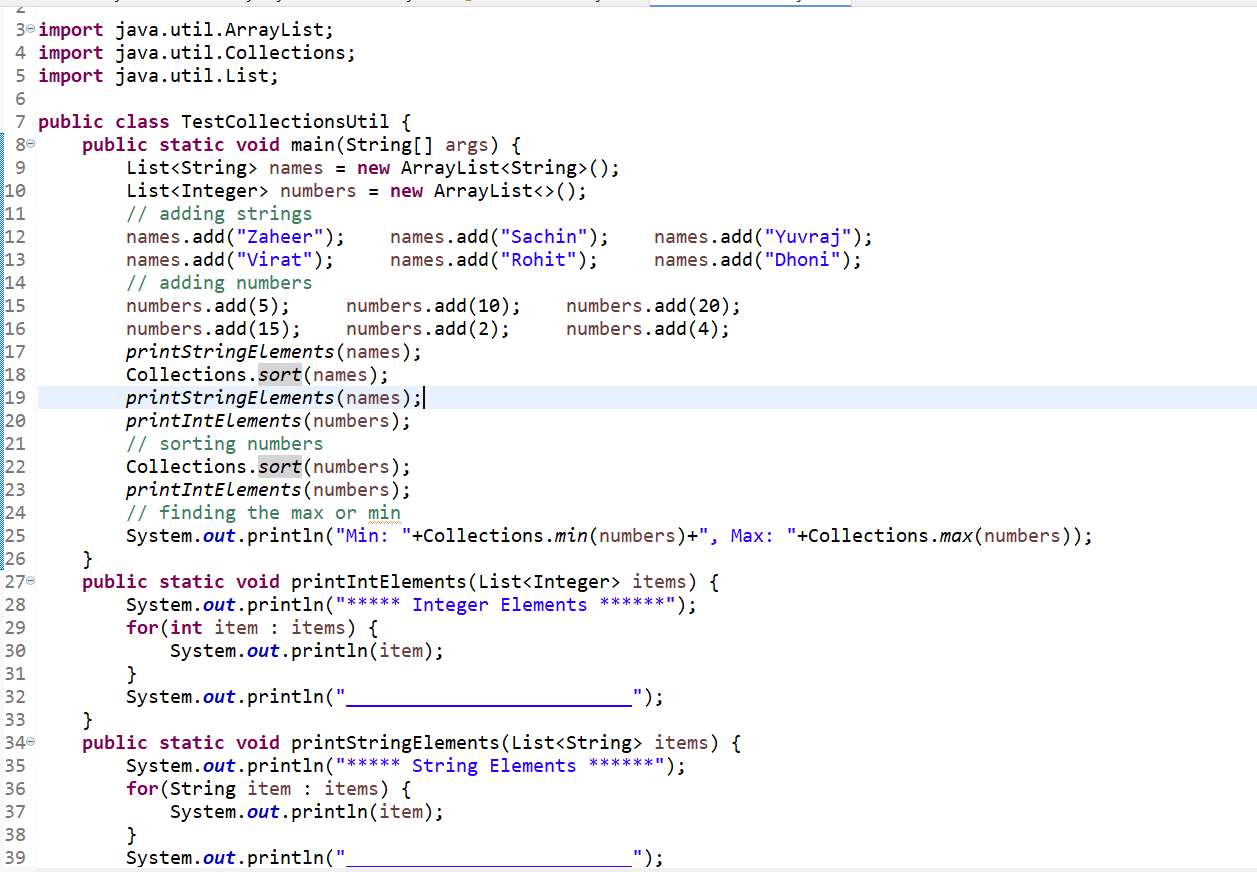


27-01-2026

Collections class

It is a utility class that can mainly perform operation on the List<T> instances, it provides utility methods like sort(), max(), min(), shuffle()

Note: Collection is an interface & Collections is a class



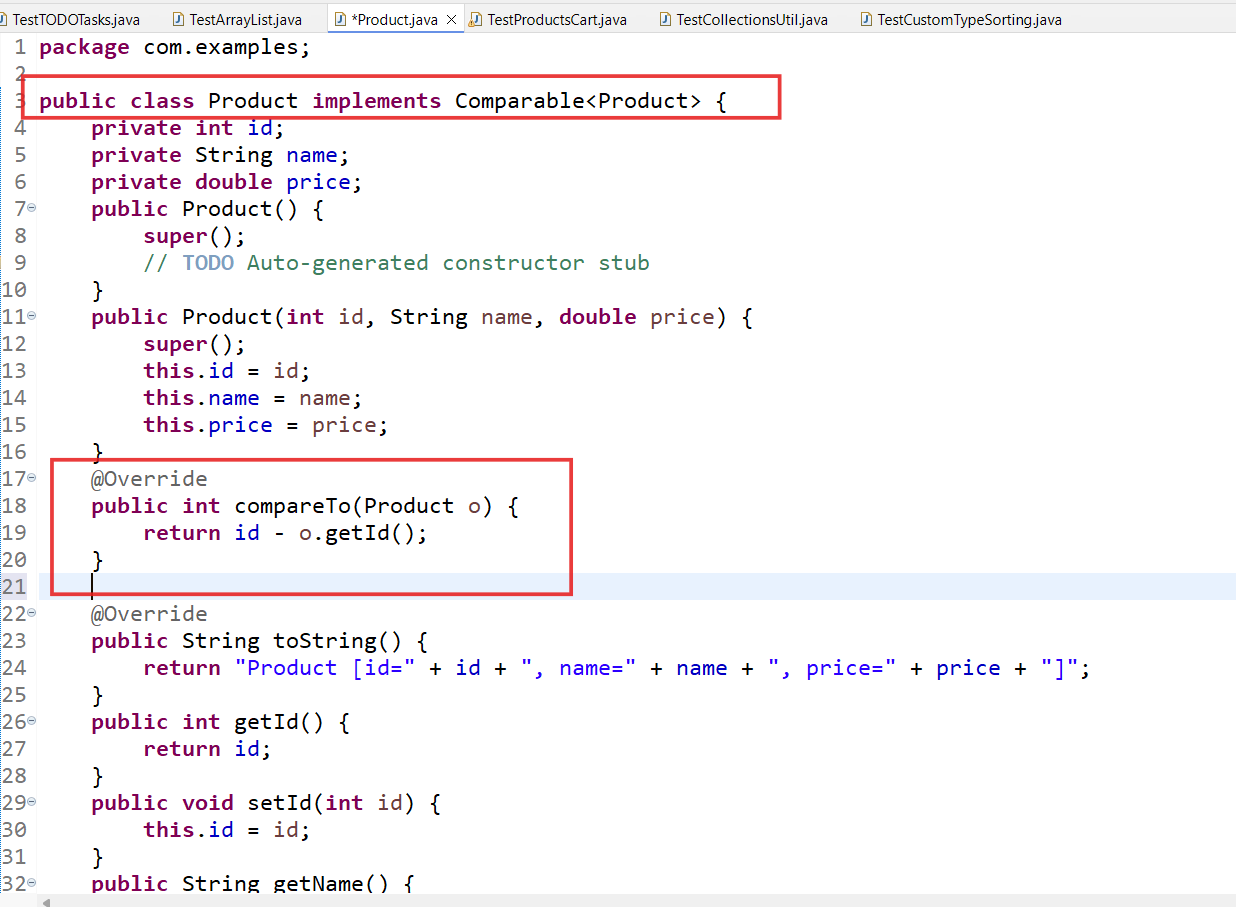
Collections.sort(List<T>): This can only sort if the List has elements which implements Comparable interface i.e., T implements Comparable<T>

Comparable<T>: This is an interface that has one abstract method called compareTo(T t) which returns an int value, the int value is used by the sort(..) method of Collections to arrange the elements based on the int value, the int value must be +ve, 0 or -ve, based on this it arranges the elements.

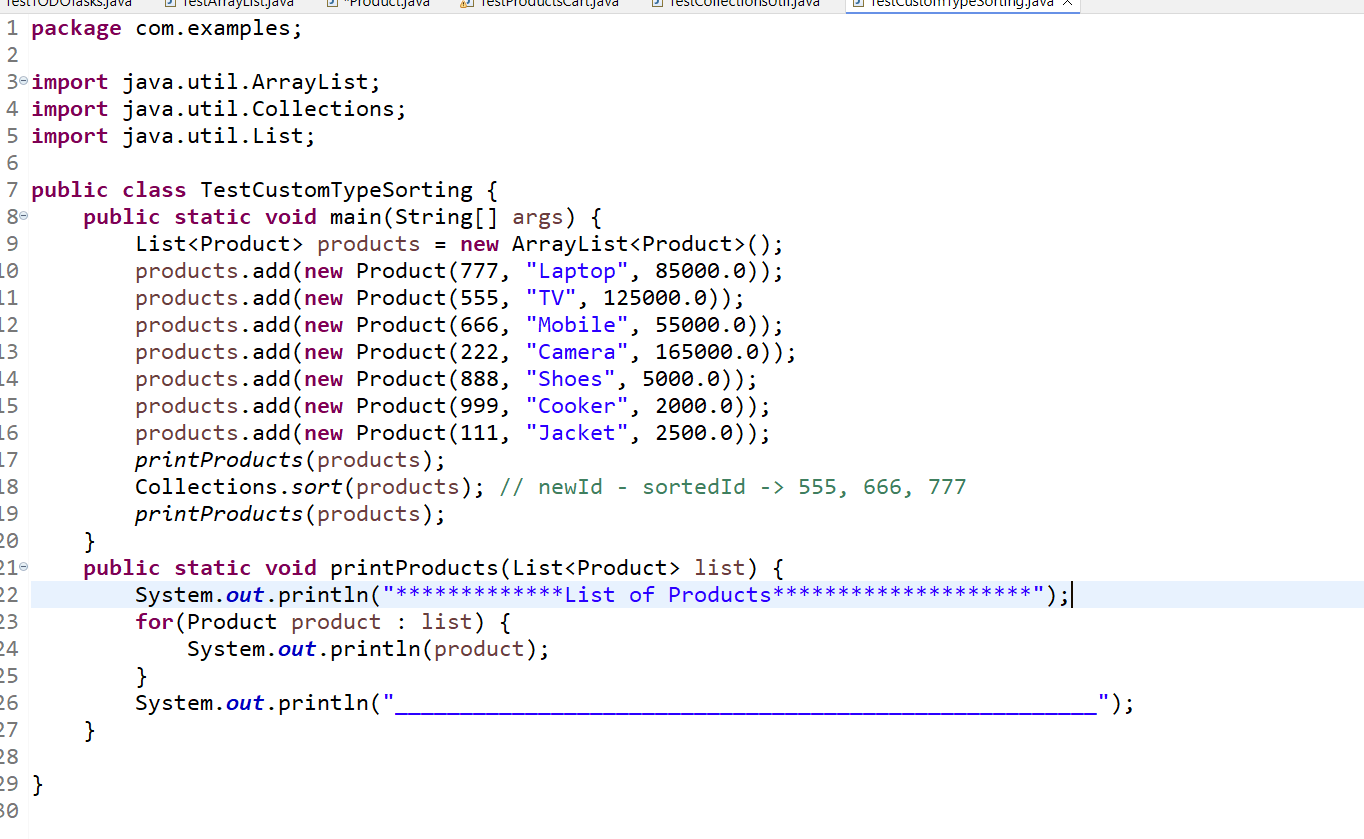
Custom classes must implement Comparable<T> and override compareTo method which specifies what properties to be compared

Note: compareTo doesn’t sort, but it tells the sorting algorithm based on which property it has to arrange the elements

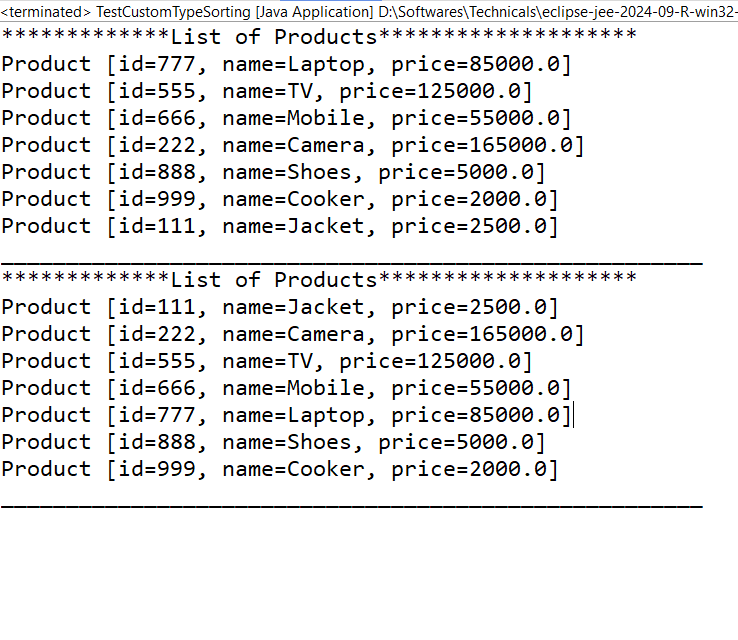
We must implement Comparable<T> in Product class



Adding the products to the List and sorting them



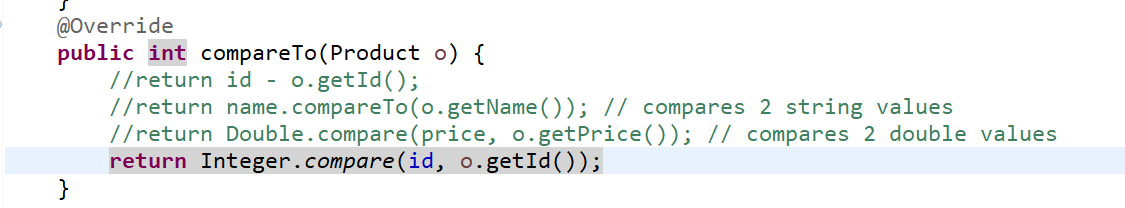
Output:



Activity:

Right now the comparison is done for id, modify the Product class so that it can compare name and run the main method to see the products list sorting based on name, similarly compare price and run the main method to see the products list sorting based on the price, previous comparison must be removed to apply new comparison on other properties

You can use Double.compare(x,y) and Integer.compare(x, y) to compare 2 values, however for string you must invoke the compareTo on string variable, just shown below with the comments.



29-01-2026

Comparator<T>:

It is also an interface with one abstract method called compare(T o1, T o2) returns int, you can implements this interface in any class and compare 2 data and return an int value, then you must pass this object to the sort() method of Collections class.

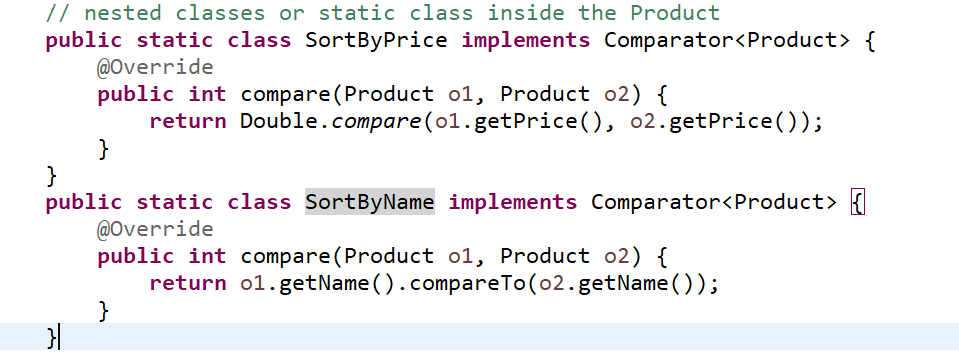
Because Collections has 2 sort methods

1. sort(List<T>): This always calls compareTo on the object that is added in the List
2. sort(List<T>, Comparator<T>): This always calls compare method of the Comparator by passing 2 elements to it to get the comparison value

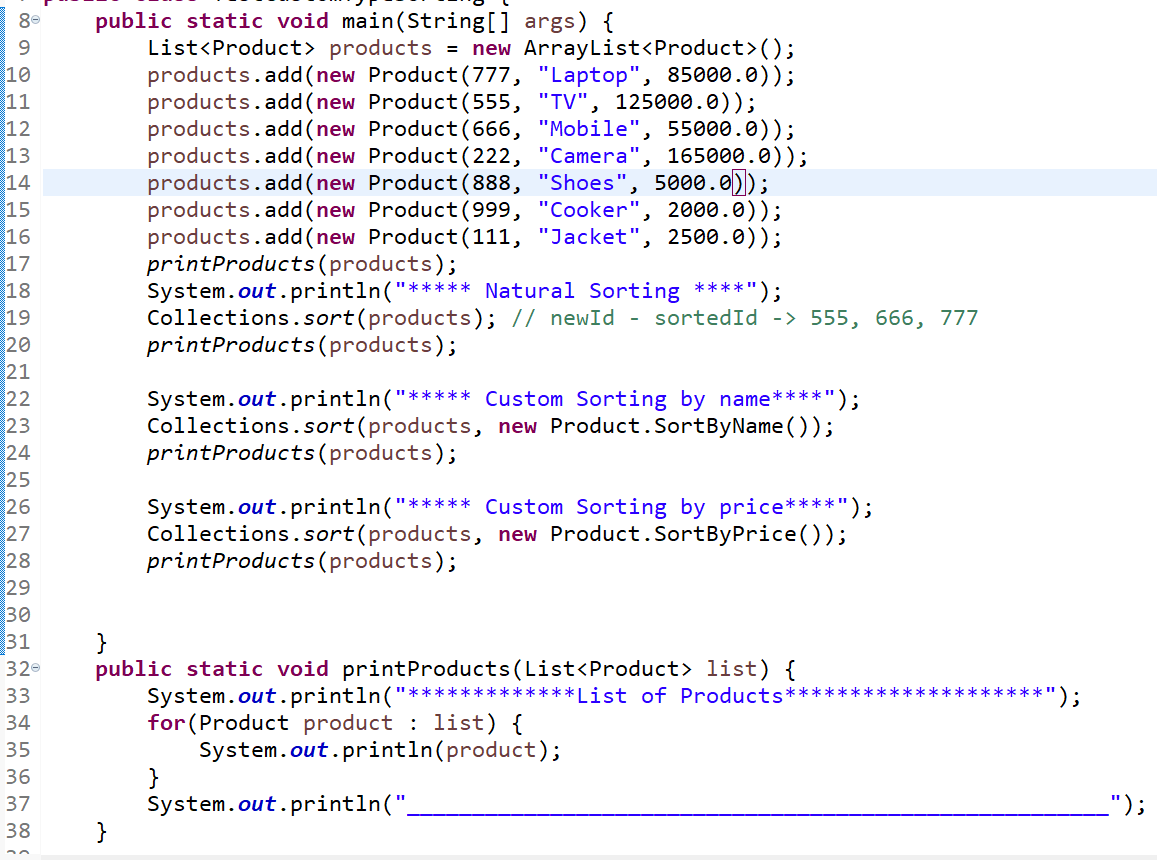
public class PriceSortingAsc implements Comparator<Product> {   
 public int compare(Product p1, Product p2) {   
 return Double.compare(p1.getPrice(), p2.getPrice());  
 }  
}  
public class NameSortingAsc implements Comparator<Product> { … }

by price -> Collections.sort(productsList, new PriceSortingAsc());   
by name -> Collections.sort(productsList, new NameSortingAsc());  
no selection -> Collections.sort(productList); // default sorting technique

Implementing the sorting technique in the existing class



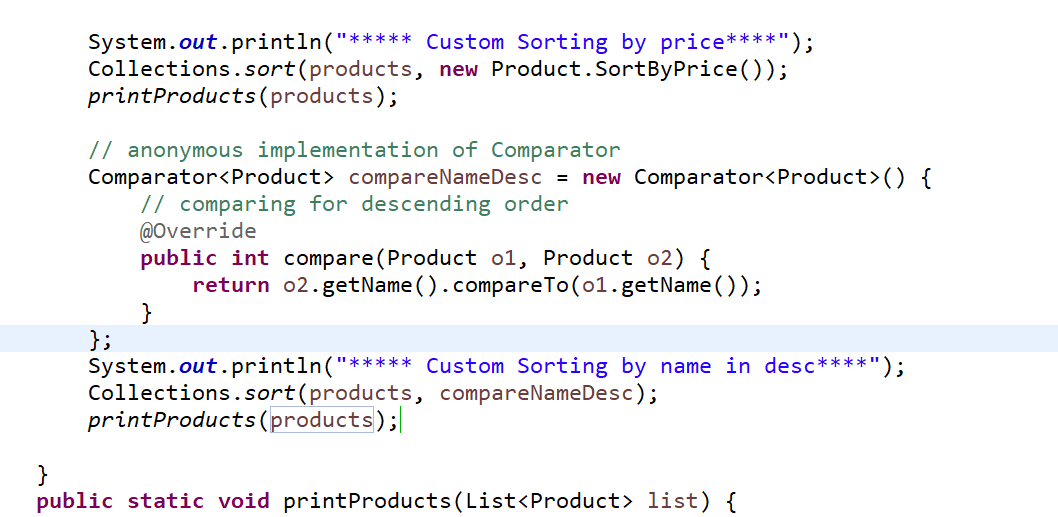
In Main method



Java supports another way of creating an inner class without naming the class - anonymous class which is useful sometimes when you don’t want to reuse the class implementation.

Comparator<Product> productCompare = new Comparator<>() {   
 // an anonymous implementation of Comparator  
 public int compare(…) { }   
}

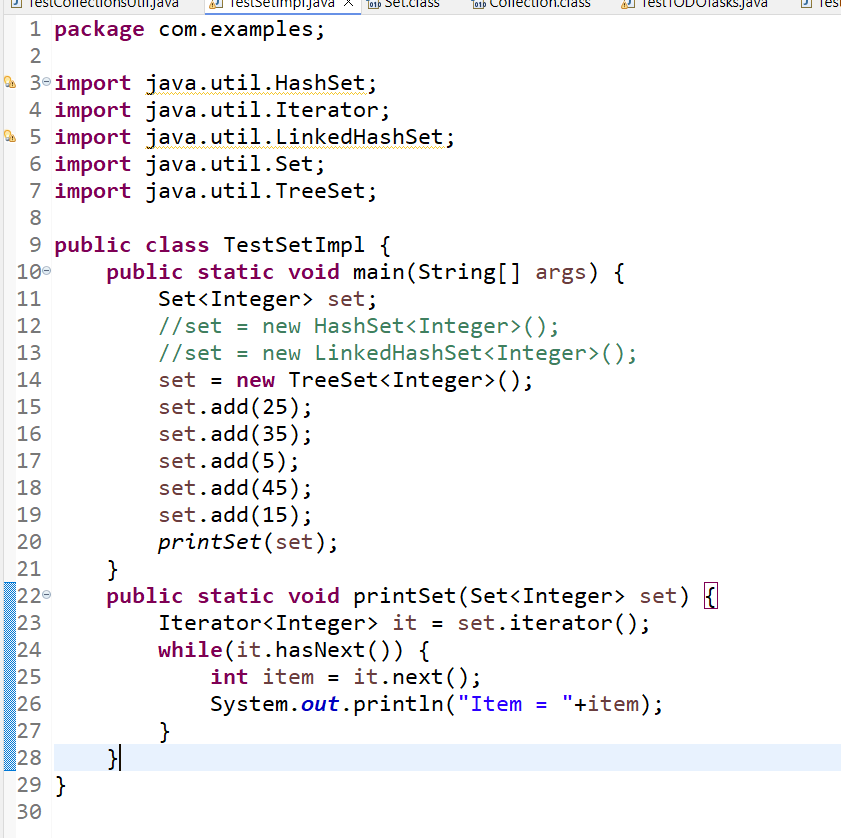
Collections.sort(productsList, productCompare);



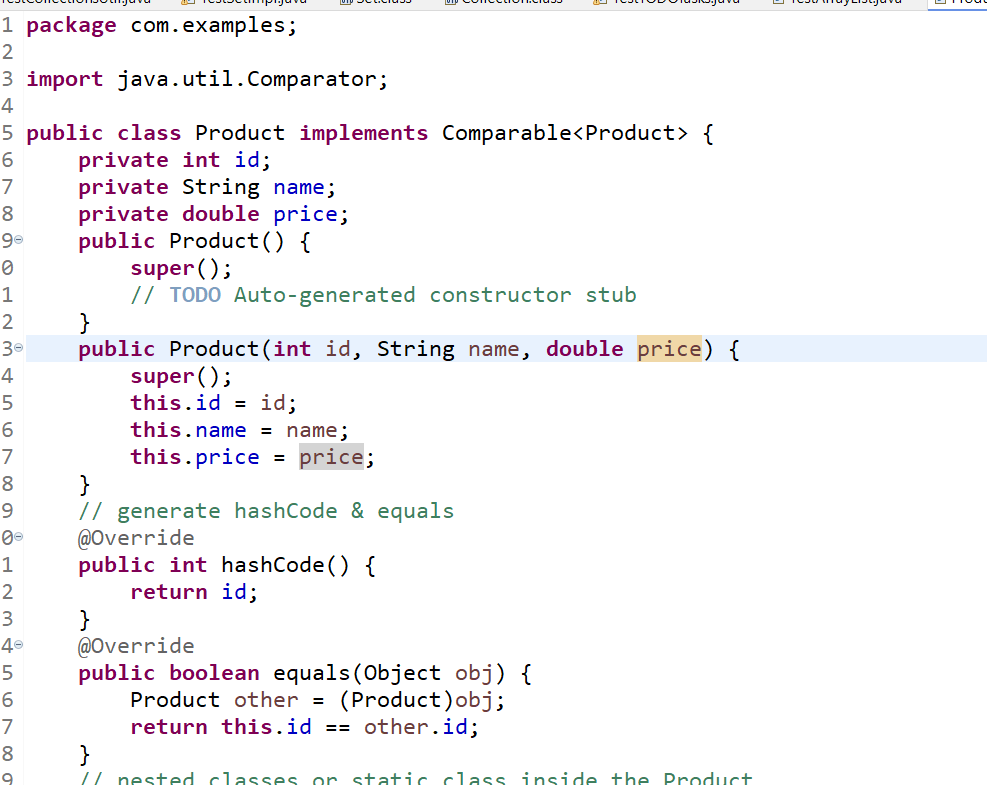
There are other implementations in Collection which are part of Set<T> interface

1. HashSet: random order, but retrieval is faster
2. LinkedHashSet: insertion order
3. TreeSet: sorted order

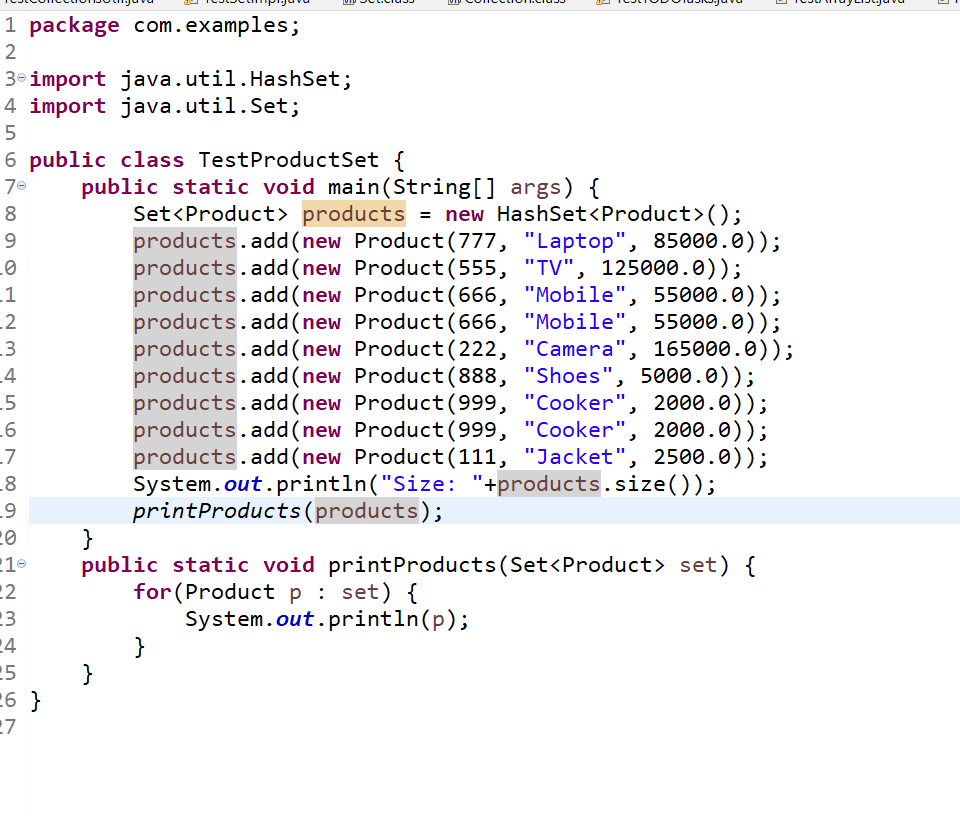
All these 3 implementations support only unique elements



overriding equals & hashCode



TestProductSet



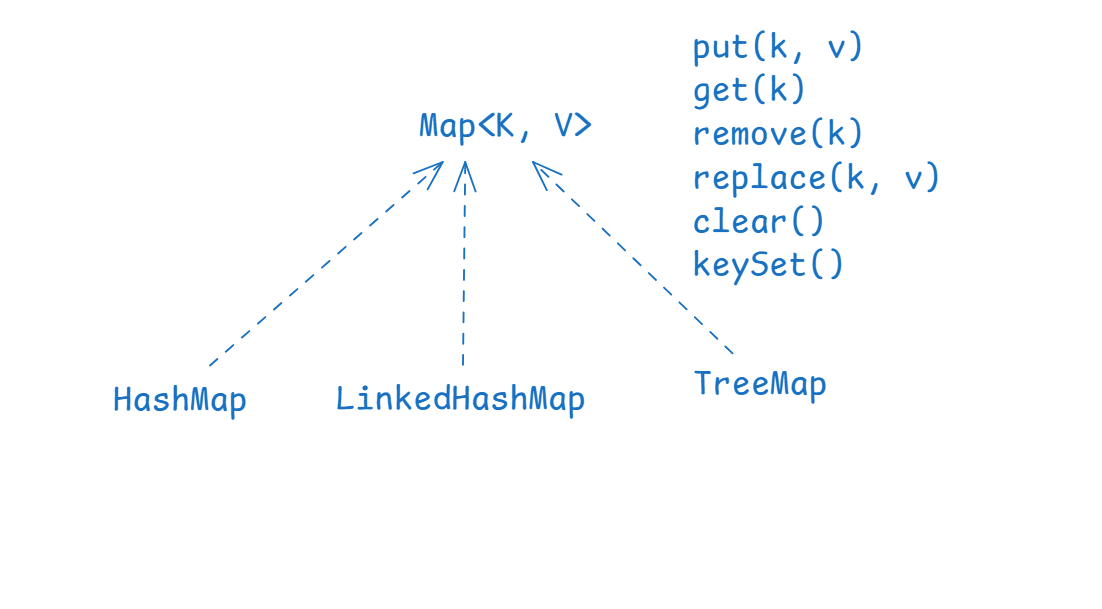
Output:



Activity:

Try to add Product objects to the TreeSet, it naturally sorts the product because of Comparable, then use TreeSet(Comparator<T>) to compare by name so that TreeSet arranges the elements by sorting name.

Map: It maintains the data in key value pairs, but Map is a separate interface that is not part of Collection hierarchy, it has its own methods

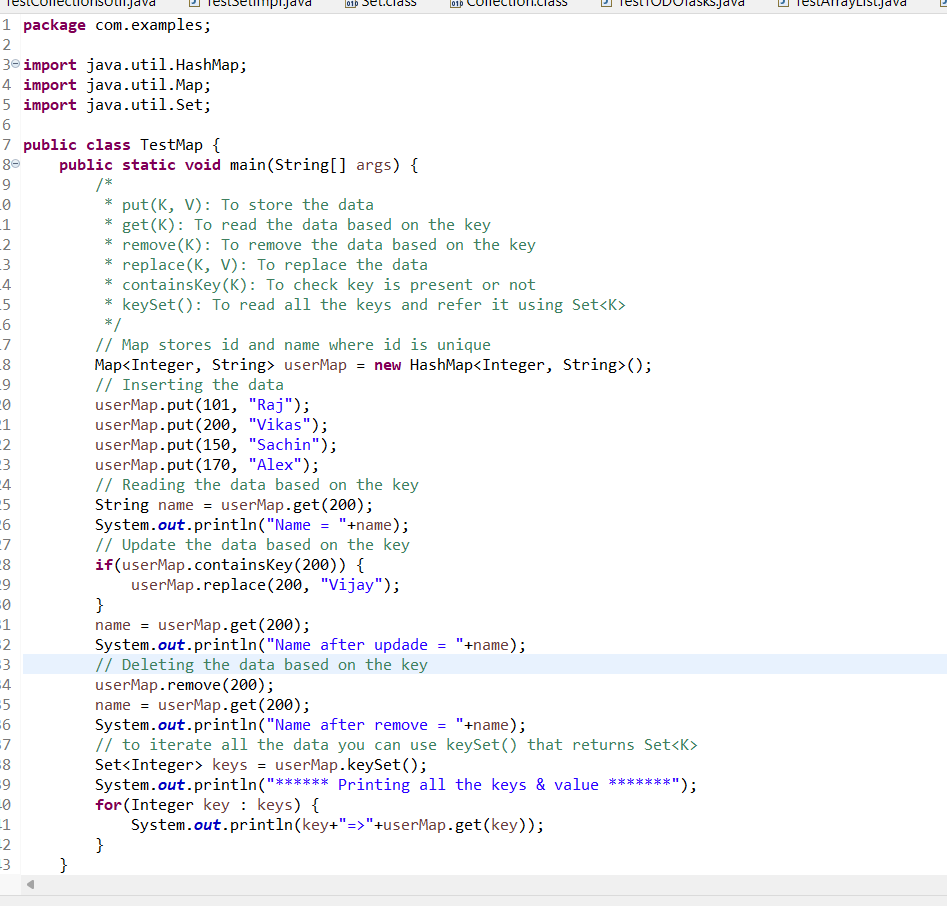


In Map Key must be unique, but value can be duplicate

HashMap: It maintains the key & value in random order, however you can retrieve the element faster because it uses hashing

LinkedHashMap: It maintains the key & value in insertion order

TreeMap: It maintains the key & value in sorted order

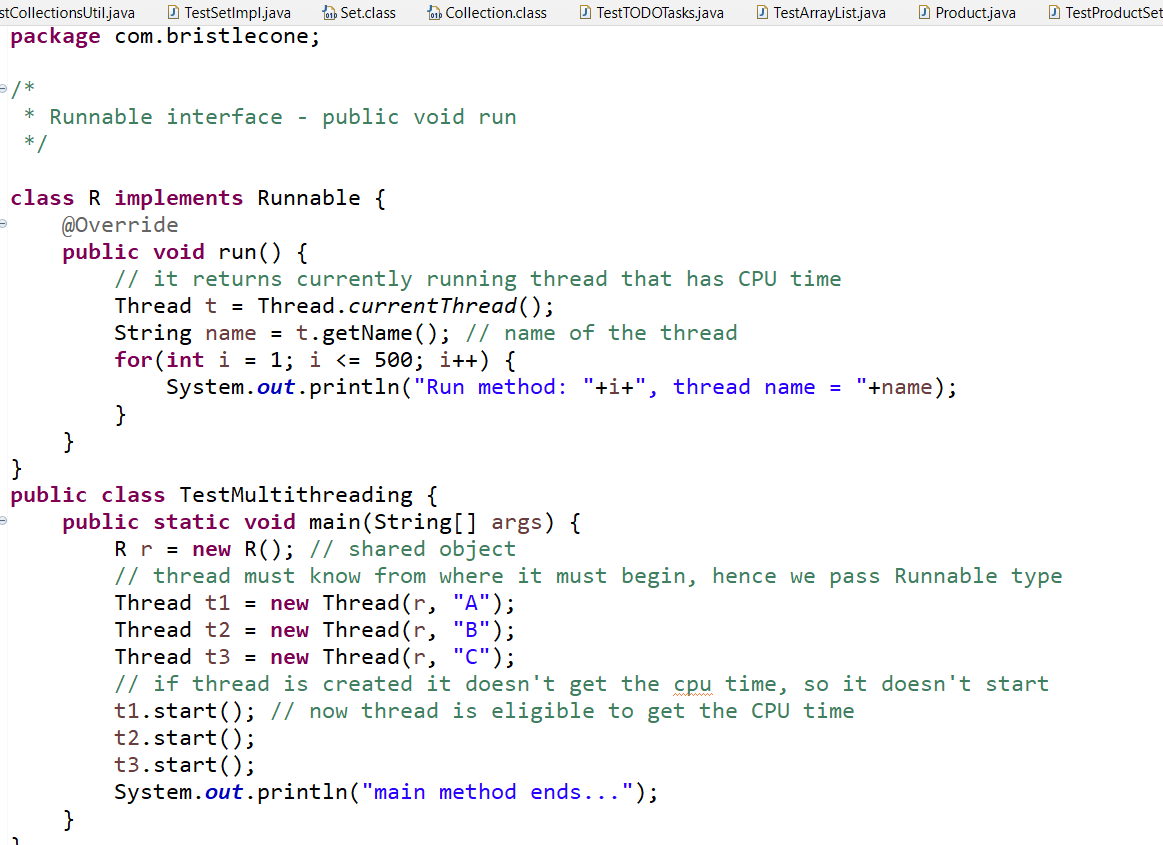


Multithreading:

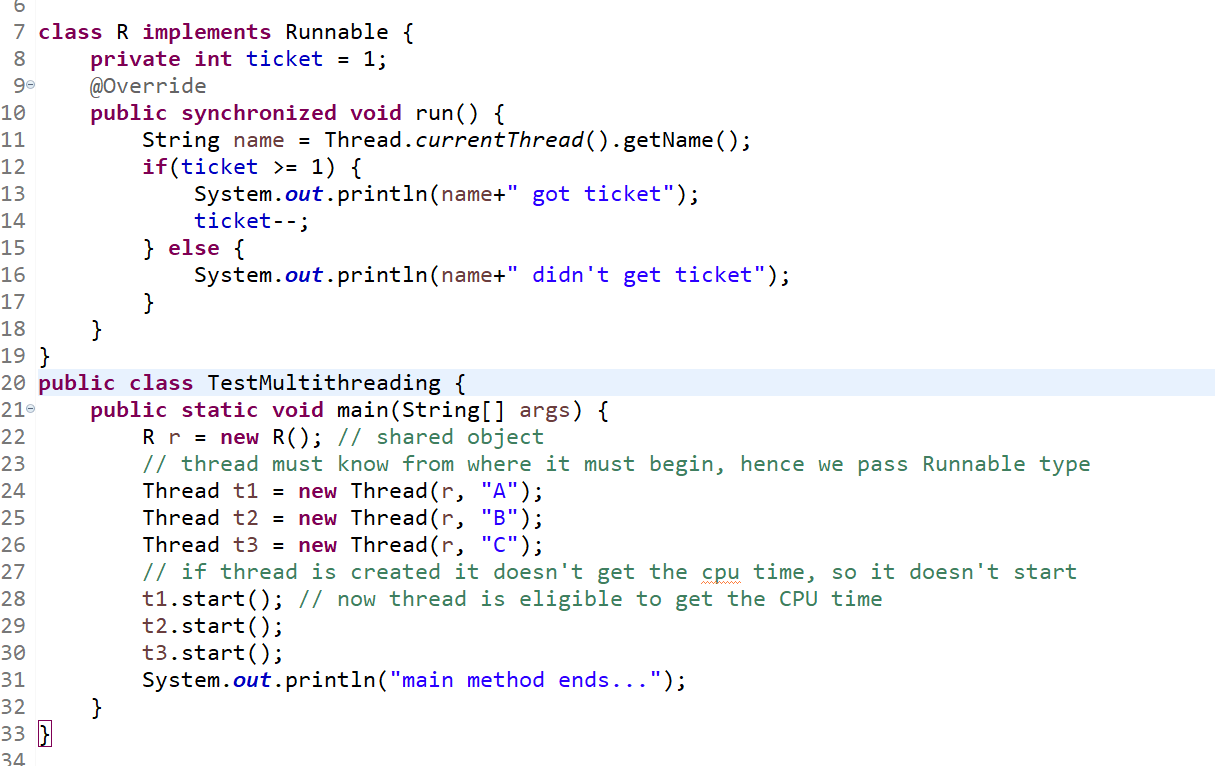
An application that can do more than one task at the same time

In Java you can implement multithreading feature with 2 API’s

1. Thread class -> To manage the thread objects
2. Runnable interface -> Entry point for the thread objects -> It has a run() method which is called by the threads when the thread starts



Thread synchronization: It is process which allows only one thread to enter inside a method so other threads must wait until the thread entered inside completes the task



File System

IO Streams: This provides set of classes to read or write files / memory

Streams are nothing but flow of data from source to destination, here source & destination can be files, input devices, output devices, memory

You can read/write data in 2 ways

1. Byte Streams - read/write binary data like image, video,
2. Character Streams - read/write character data like text data

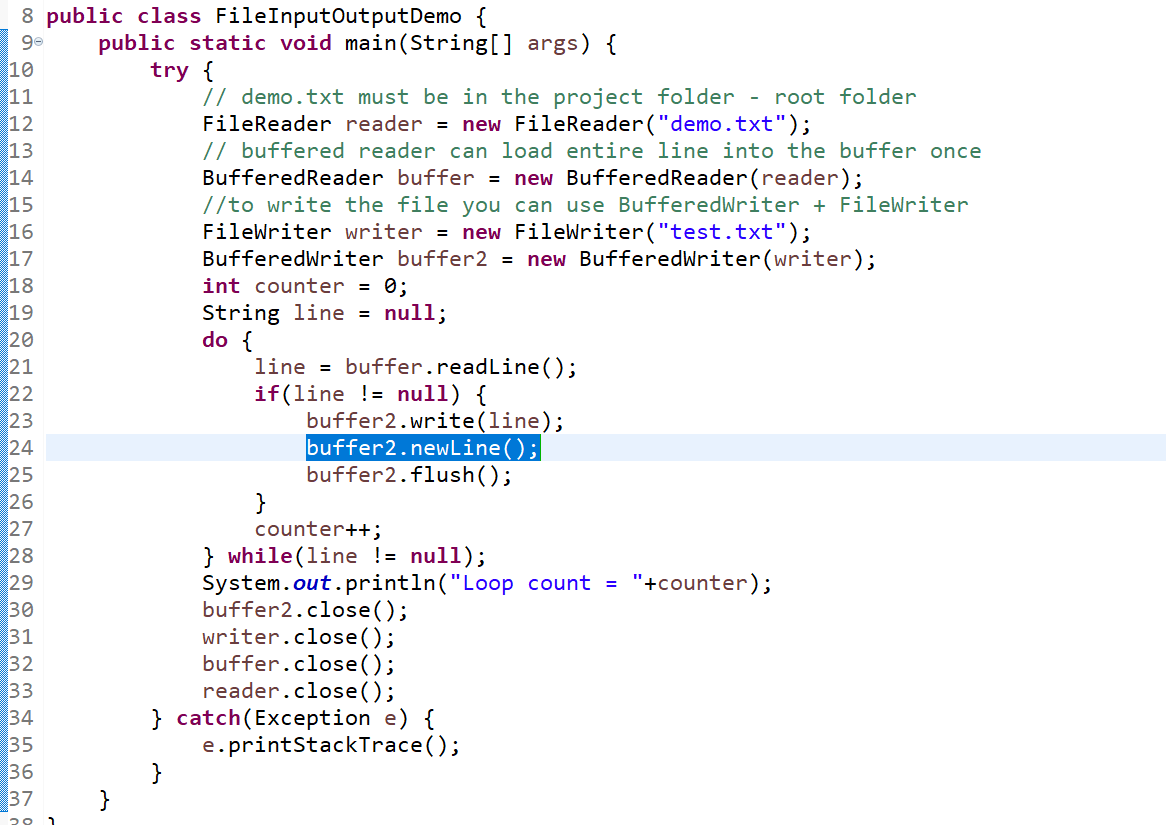
Java has provides set of classes to perform IO operations

Byte Stream classes

1. FileInputStream - read binary files
2. BufferedInputStream - read binary data from the memory
3. FileOuptutStream - write binary files
4. BufferedOutputStream - write binary data to the memory
5. ObjectInputStream - read object from source
6. ObjectOutputStream - write object to destination

Character stream classes

1. FileReader - read text files
2. FileWriter - write text files
3. BufferedReader - read text data from the memory
4. BufferedWriter - write text data to the memory.



Serialization: This is a process of writing an object into the file

Deserialization: This is a process of reading an object from the file

Note: When you write an object to the file, the file content will not be in a readable format

Any object you want to serialize must implement an interface Serializable, it is a marker interface, this interface doesn’t have any methods, it is just to mark the objects

public class Product implements Serializable { }  
public class Account { }

Only those objects that are of Serializable can be written to the file, here Product objects can be written to the file, but account objects can’t be written to the file

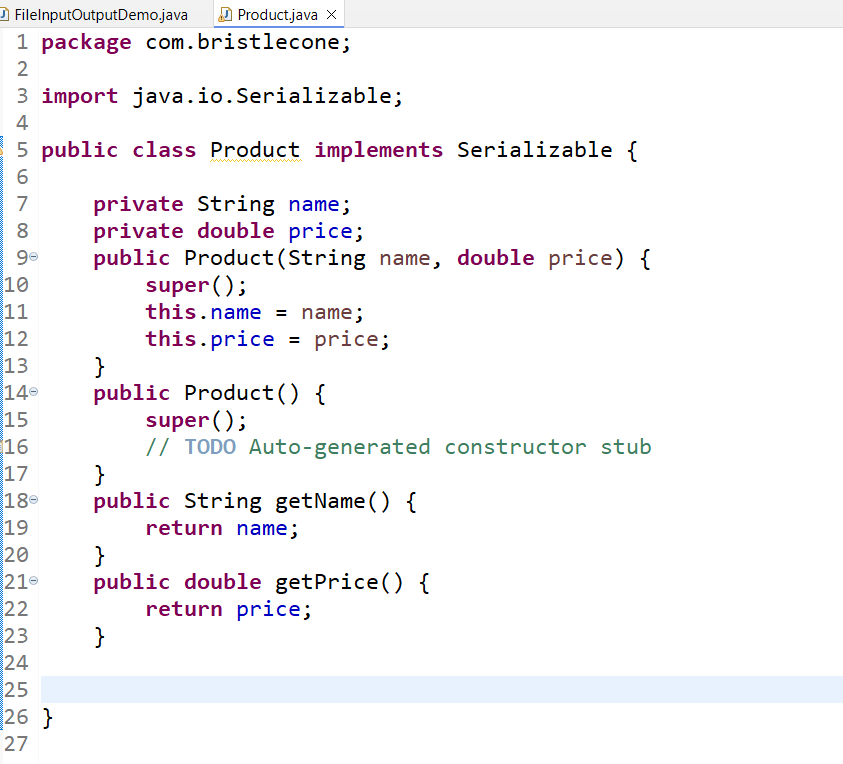
ObjectOutputStream -> writeObject(object): This writes the object you pass

ObjectInputStream -> readObject(): This reads & returns the object that is serialized

FileOutputStream fos = new FileOutpuStream(“products.ser”);  
ObjectOutputStream oos = new ObjectOutputStream(fos);   
oos.writeObject( product ); // if product implements Serializable it writes else throws error

FileInputStream fis = new FileInputStream(“products.ser”);  
ObjectInputStream ois = new ObjectInputStream(fis);  
Product p = (Product) ois.readObject();

Product.java





Activity:

List<Product> you can serialize the List<Product> directly because every collection is Serializable type, add some product to the List<Product> and serialize the List<Product> and then deserialize the List<Product> and iterate over the List<Product>

Functional interface & Lambda expression

Functional interfaces are those interface that will have only one abstract method, which helps you to implement them using lambda expression so that the code will be concise to write, it supplies function as an argument but behind the scene its like passing an object of the implementation.

Comparator<T> comp = new Comparator<T> {   
 public int compare(T t1, T t2) {   
 return intValue;  
 }  
}

Collections.sort(comp); // traditional approach

Comparator<T> comp2 = (t1, t2) -> { return intValue; }  
Comparator<T> comp3 = (t1, t2) -> intValue;

Collections.sort(comp3);

[or]

Collections.sort((t1, t2) -> intValue);

Predicate<Integer> predicate = new Predicate<Integer>() {   
 public boolean test(Integer t) { return t % 2 == 0; }  
}

Predicate<T> predicate = t => true;

Predicate<Integer> predicate = t => t % 2 == 0;

Consumer<Integer> consumer = t => statement;

Java new features

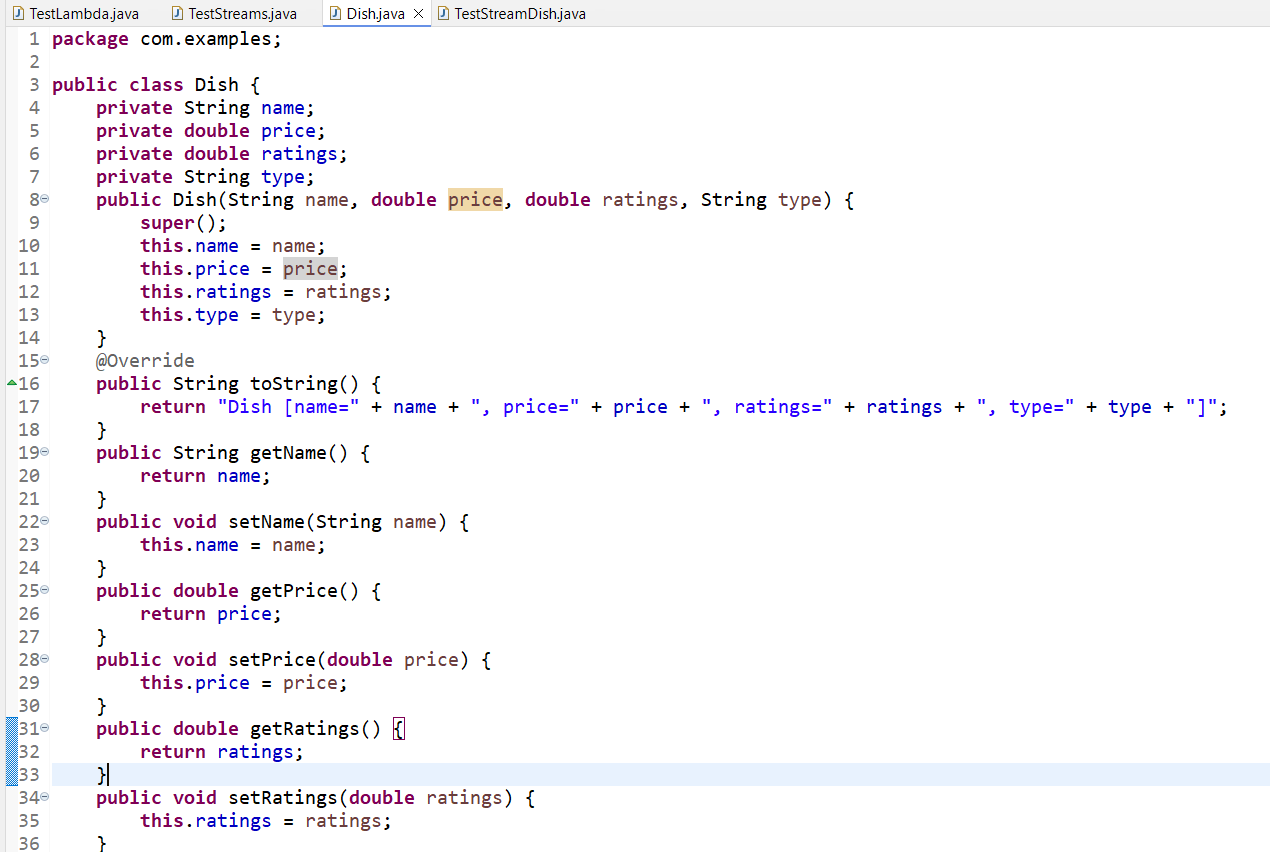
1. Functional interface
2. Lambda expression
3. LocalDate & LocalTime
4. Default methods & static methods inside the interface
5. Sealed classes & non-Sealed classes
6. Stream API’s

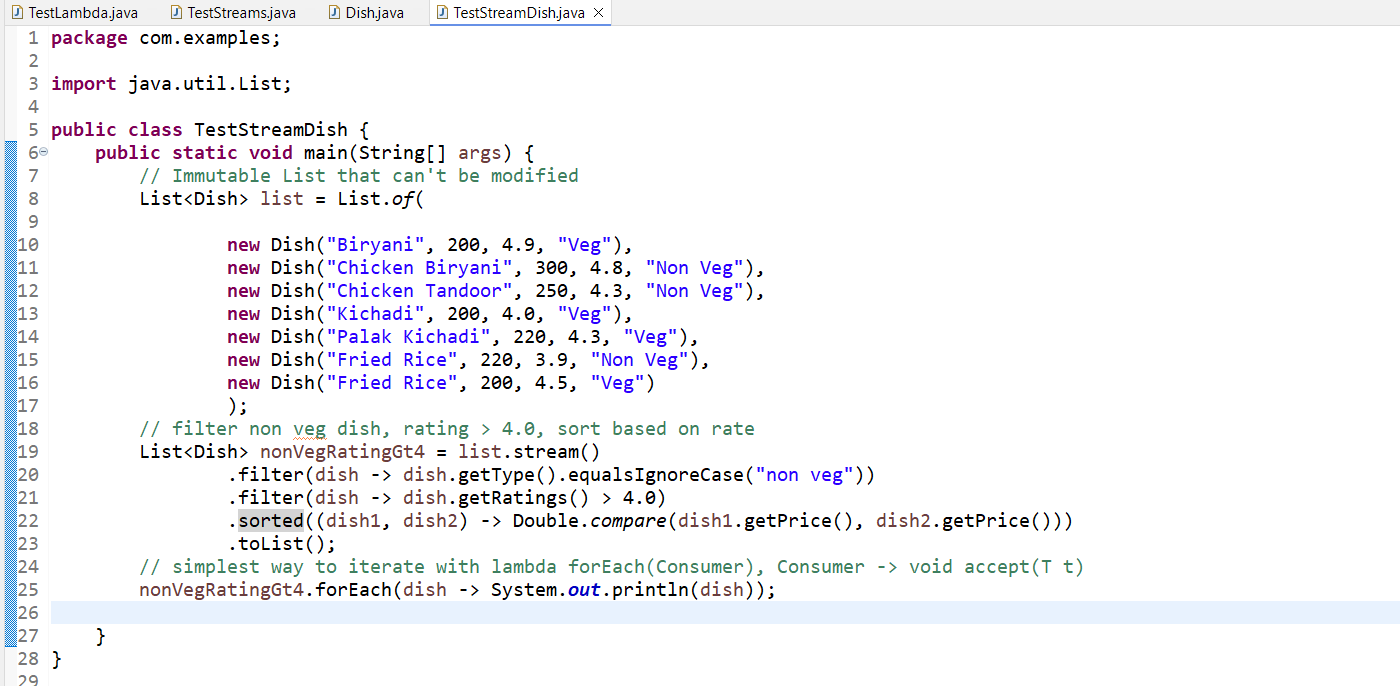


What is a Stream in Java

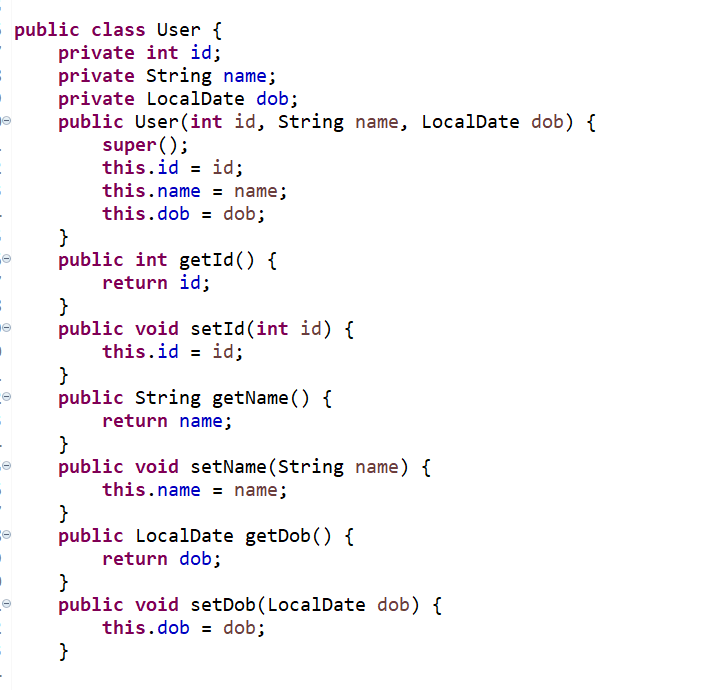
A stream in java is used to process collections of data in a functional way

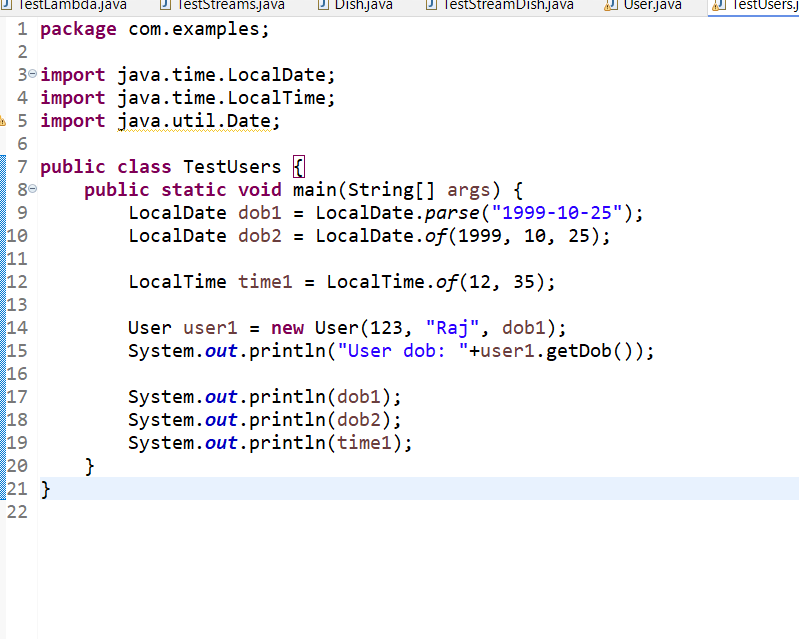
Data -> operations -> result





LocalDate & LocalTime





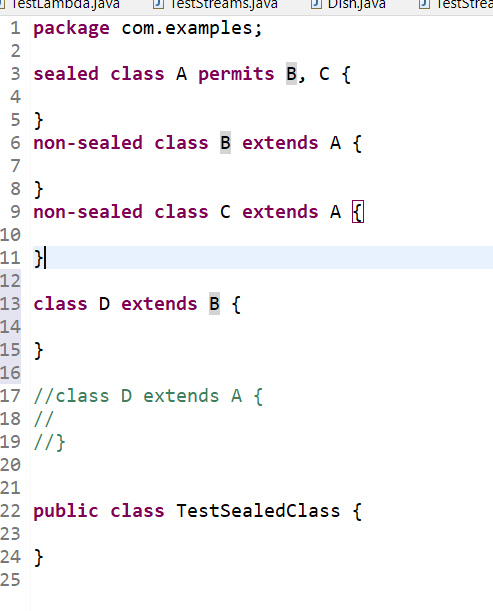
Earlier classes like java.util.Date gave wrong results, hence java provides classes to store date & time separately

LocalDate - stores date in ISO format (yyyy-MM-dd)

LocalTime - stores time in ISO format

LocalDateTime - stores date & time in ISO format

Sealed classes & Non-Sealed classes This prevents wrong classes inheriting the parent class



Block String: This allows you to write strings in multiple lines without breaking the string with + operator

