Java, Spring Boot and React.js

Software's requirements

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

Verify Java

```
C:\Users\kisho>javac -version
javac 21.0.8

C:\Users\kisho>java -version
java version "21.0.8" 2025-07-15 LTS
Java(TM) SE Runtime Environment (build 21.0.8+12-LTS-250)
Java HotSpot(TM) 64-Bit Server VM (build 21.0.8+12-LTS-250, mixed mode, sharing)

C:\Users\kisho>
```

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

TestApp.java TestFundamentals.java

```java
package com.bristelstone;

public class TestFundamentals {
    public static void main(String[] args) {
        // Datatypes - primitives / non-primitives
        // String is a non-primitive datatypes
        String username = "Alexandar";
        int age = 25;
        long phone = 9947483648L;
        String email = "alex@gmail.com";
        char gender = 'M';
        double height = 5.8;
        boolean isEmployed = true;
        System.out.println("Username = "+username);
        System.out.println("Age = "+age+", Phone = "+phone+", Email = "+email);
        System.out.println("Gender = "+gender+", Height = "+height+", Employed = "+isEmployed);
    }
}

```

Output:

Problems Javadoc Declaration Console

<terminated> TestFundamentals [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Jan 5, 2026, 7:59:24 PM – 7

```
Username = Alexandar
Age = 25, Phone = 9947483648, Email = alex@gmail.com
Gender = M, Height = 5.8, Employed = true
```

Branching Statements

- if
- if else
- if else if else if else
- switch

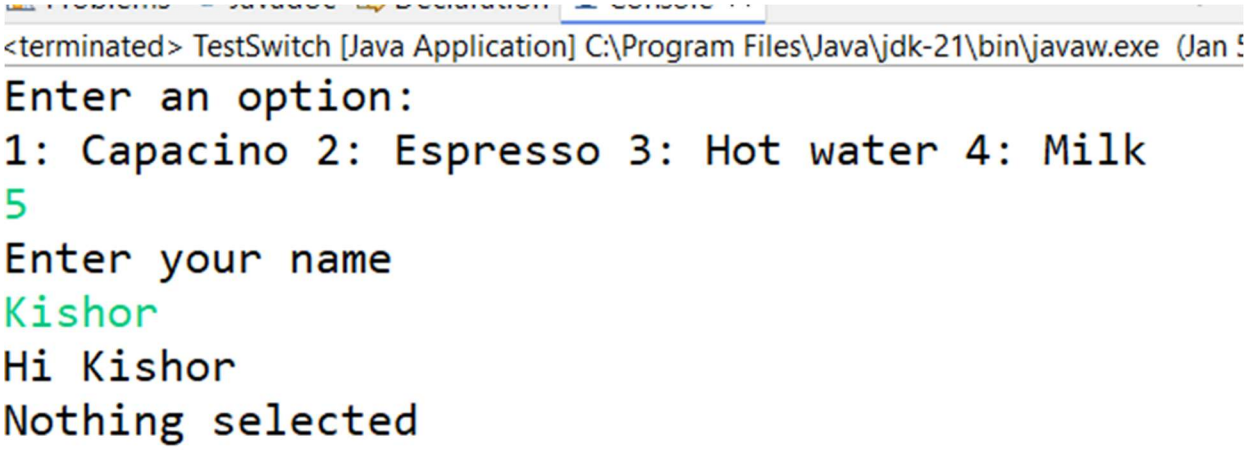
```
// if and else
if(gender == 'M') {
    System.out.println("Hello Mr. "+username);
} else if(gender == 'F') {
    System.out.println("Hello Ms. "+username);
} else {
    System.out.println("Hello "+username);
}
```

Switch statements

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

```
1  package com.bristelstone;
2
3  import java.util.Scanner;
4
5  public class TestSwitch {
6      public static void main(String[] args) {
7          Scanner scan = new Scanner(System.in);
8          // nextInt(), next(), nextDouble(), nextFloat()
9          System.out.println("Enter an option:");
10         System.out.println("1: Capacino 2: Espresso 3: Hot water 4: Milk");
11         int option = scan.nextInt();
12         System.out.println("Enter your name");
13         scan.nextLine();
14         String name = scan.nextLine();
15         System.out.println("Hi "+name);
16         switch(option) {
17         case 1: System.out.println("Capacino is selcted");
18         break;
19         case 2: System.out.println("Espresso is selcted");
20         break;
21         case 3: System.out.println("Hot water is selcted");
22         break;
23         case 4: System.out.println("Milk is selcted");
24         break;
25         default :
26             System.out.println("Nothing selected");
27         }
28
29         scan.close();
30     }
31 }
```

Output:



Arrays & Loops

Arrays are containers that can store multiple values of same datatypes

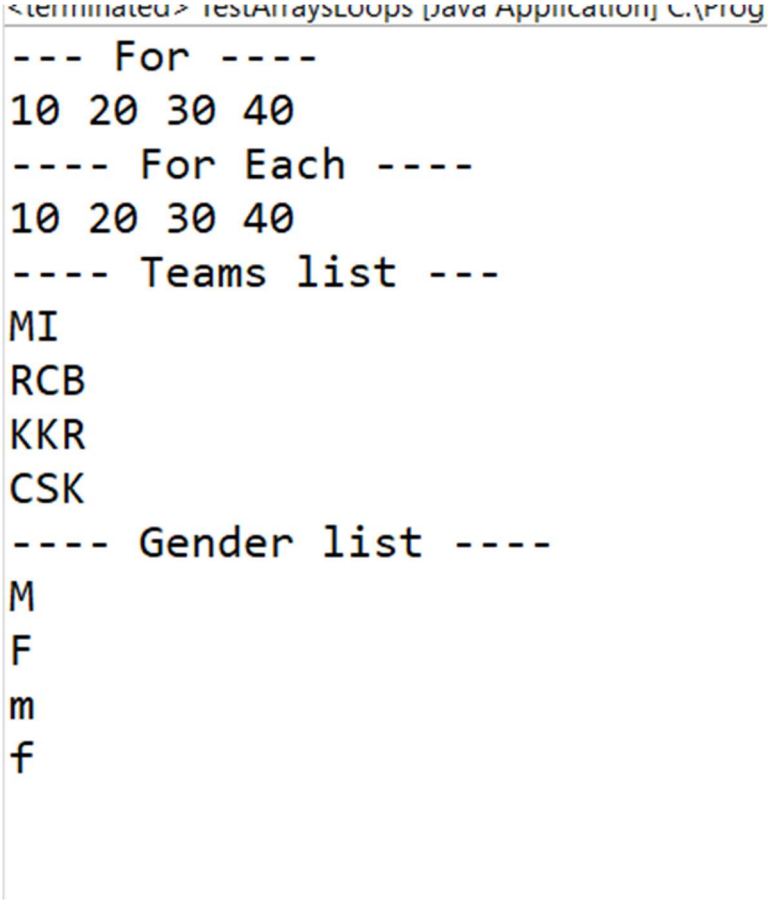
int[] departmentCodes = { 10, 20, 30, 40} ;

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

char[] gender = {‘M’, ‘F’, ‘m’, ‘f’}

```java
public class TestArraysLoops {
    public static void main(String[] args) {
        int[] departmentCodes = {10, 20, 30, 40};
        String[] teams = {"MI", "RCB", "KKR", "CSK"};
        char[] gender = {'M','F','m','f'};

        // traditional for with initialization, condition, expression
        System.out.println("--- For ----");
        for(int i = 0; i < departmentCodes.length; i++) {
            System.out.print(departmentCodes[i]+" ");
        }
        System.out.println();
        // for each / enhanced for loop
        System.out.println("---- For Each ----");
        for(int ele : departmentCodes) {
            System.out.print(ele+" ");
        }
        System.out.println();

        // try to use both types for loops on String[] teams & char[] gender and print them
        System.out.println("---- Teams list ---");
        for(String team : teams) {
            System.out.println(team);
        }
        System.out.println("---- Gender list ----");
        for(char g : gender) {
            System.out.println(g);
        }
    }
}
```

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

TestApp.java | TestFundamentals.java | TestSwitch.java | TestArraysLoops.java | TestWhileAndDoWhile.java

```java
package com.bristelstone;

public class TestWhileAndDoWhile {
    public static void main(String[] args) {
        int counter1 = 5;
        System.out.println("----- while ----");
        while(counter1 > 10) {
            System.out.println("while loop");
        }
        System.out.println("---- do while ----");
        do {
            System.out.println("do while loop");
        } while(counter1 > 10);
    }
}
```

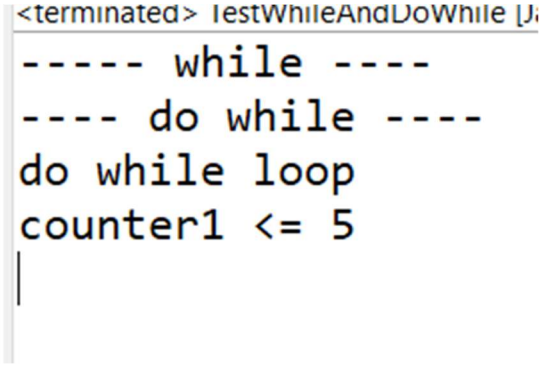
Operators in Java

Operators perform operations on the variables

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

```java
package com.bristelstone;

public class TestWhileAndDoWhile {
    public static void main(String[] args) {
        int counter1 = 5;
        System.out.println("----- while ----");
        while(counter1 > 10) {
            System.out.println("while loop");
        }
        System.out.println("---- do while ----");
        do {
            System.out.println("do while loop");
        } while(counter1 > 10);

        String result = (counter1 > 5) ? "counter1 > 5" : "counter1 <= 5";

        System.out.println(result);
    }
```

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

User.java TestUser.java

```java
package com.bristelstone;

public class User {
    int userId; // instance variables
    String name; // instance variables
    long phone;  // instance variables
    // constructor that takes parameters
    public User(int userId, String name, long phone) {
        this.userId = userId;
        this.name = name;
        this.phone = phone;
    }
    // method to print the user information
    void display() {
        System.out.println("User Id = "+userId);
        System.out.println("Name = "+name);
        System.out.println("Phone = "+phone);
    }
}
```

TestUser.java

User.java TestUser.java

```java
package com.bristelstone;

public class TestUser {
    public static void main(String[] args) {
        // User() is a constructor
        User user1 = new User(100, "Alex", 987654321L); // [userId=100,name=Alex,phone=987654321]
        User user2 = new User(200, "Brad", 987655678L);// [userId=200,name=Brad,phone=987655678L]

        user1.display();
        System.out.println("---------------------------");
        user2.display();
    }
}
```

Output:

```
<terminated> TestUser [Java Applicatior
User Id = 100
Name = Alex
Phone = 987654321
User Id = 200
Name = Brad
Phone = 987655678
```

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

User.java × TestUser.java

```java
package com.bristelstone;

public class User {
    int userId; // instance variables
    String name; // instance variables
    long phone;  // instance variables

    // constructor that takes parameters
    public User(int userId, String name) {
        this.userId = userId;
        this.name = name;
        System.out.println("User(int, String)");
    }

    public User(int userId, String name, long phone) {
        this(userId, name); // User(int, String)
        this.phone = phone;
        System.out.println("User(int, String, long)");
    }
    // method to print the user information
    void display() {
        System.out.println("User Id = "+userId);
        System.out.println("Name = "+name);
        System.out.println("Phone = "+((phone != 0) ? phone : "Not Available"));
    }
}
```

TestUser.java

User.java TestUser.java

```java
package com.bristelstone;

public class TestUser {
    public static void main(String[] args) {
        // User() is a constructor
        User user1 = new User(100, "Alex", 987654321L); // [userId=100,name=Alex,phone=987654321]
        System.out.println("--------------------------------");
        User user2 = new User(200, "Brad");// [userId=200,name=Brad,phone=987655678L]

        user1.display();
        System.out.println("----------------------------");
        user2.display();
    }
}
```

Output:

```
<terminated> TestUser [Java Application] D:\Softwares\Technic
User(int, String)
User(int, String, long)
--------------------------------
User(int, String)
User Id = 100
Name = Alex
Phone = 987654321
----------------------------
User Id = 200
Name = Brad
Phone = Not Available
```

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.

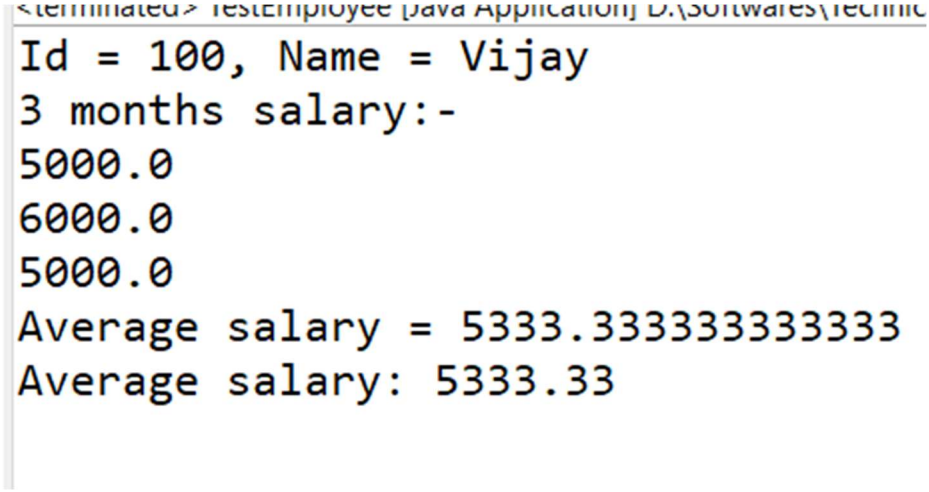
User.java TestUser.java Employee.java TestEmployee.java

```java
package com.bristelstone;

public class Employee {
    int id;
    String name;
    double[] monthlySalaries = new double[3];

    public Employee(int id, String name, double[] monthlySalaries) {
        this.id = id;
        this.name = name;
        this.monthlySalaries = monthlySalaries;
    }
    public double calculateAverage() {
        double sum = 0;
        for(double salary : monthlySalaries) {
            sum = sum + salary;
        }
        return sum / monthlySalaries.length;
    }
    public void display() {
        System.out.println("Id = "+id+", Name = "+name);
        System.out.println("3 months salary:-");
        for(double salary : monthlySalaries) {
            System.out.println(salary);
        }
        System.out.println("Average salary = "+calculateAverage());
        System.out.printf("Average salary: %.2f", calculateAverage());
    }
}
```

User.java TestUser.java Employee.java TestEmployee.java

```java
package com.bristelstone;

public class TestEmployee {
    public static void main(String[] args) {
        Employee emp = new Employee(100, "Vijay", new double[] {5000.0, 6000.0, 5000.0});
        emp.display();
    }
}
```

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

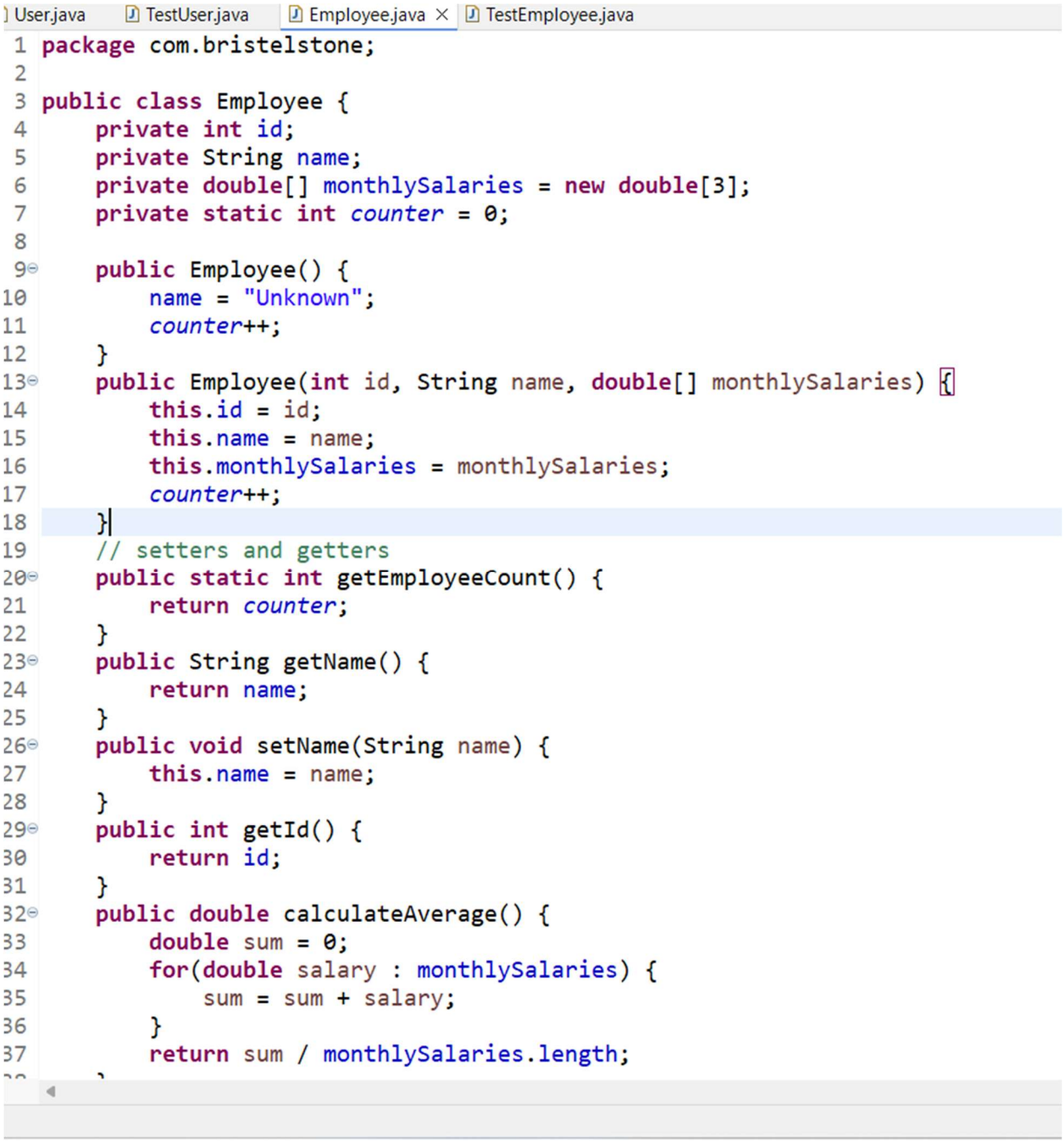
setters -> to set the value

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

```java
package com.bristlestone;

import java.time.LocalDate;

public class Person {
    private String name;
    private LocalDate dob; // ISO date format - yyyy/MM/dd
    private String gender;
    public Person() {
        System.out.println("Person() constructor");
    }
    public Person(String name, LocalDate dob, String gender) {
        this.name = name;
        this.dob = dob;
        this.gender = gender;
        System.out.println("Person(String, LocalDate, String)");
    }
    // setters and getters
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public LocalDate getDob() {
        return dob;
    }
    public void setDob(LocalDate dob) {
        this.dob = dob;
    }
    public String getGender() {
        return gender;
    }
    public void setGender(String gender) {
        this.gender = gender;
    }
}
```

Employee.java

```java
package com.bristlestone;

import java.time.LocalDate;

public class Employee extends Person {
    private int id;
    private String desig;
    private double salary;
    public Employee() {
        super();
        System.out.println("Employee()");
    }
    public Employee(int id, String name, LocalDate dob, String gender, String desig, double salary) {
        super(name, dob, gender); // Person(String, LocalDate, String)
        this.id = id;
        this.desig = desig;
        this.salary = salary;
        System.out.println("Employee(int, String, LocalDate, String, String, double)");
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getDesig() {
        return desig;
    }
    public void setDesig(String desig) {
        this.desig = desig;
    }
    public double getSalary() {
        return salary;
    }
    public void setSalary(double salary) {
        this.salary = salary;
    }
```

TestInheritance.java

erson.java | Employee.java | TestInheritance.java ×

```java
package com.bristlestone;

import java.time.LocalDate;


public class TestInheritance {
    public static void main(String[] args) {
        Employee emp1 = new Employee();
        System.out.println("----------------------");
        Employee emp2 = new Employee(1234, "Alex", LocalDate.parse("2002-10-15"), "Male", "Manager", 99973473L);
        System.out.println("Employee1 informations");
        System.out.println("Name = "+emp1.getName()+", Desig = "+emp1.getDesig());
        System.out.println("Employee2 informations");
        System.out.println("Name = "+emp2.getName()+", Desig = "+emp2.getDesig());
    }
}
```

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

Person.java Employee.java TestInheritance.java TestMethodOverloading.java

```java
package com.bristlestone;

class PrintData {
    /*
     * void print(int x) { System.out.println("print(int)"); }
     */
    /*
     * void print(long x) { System.out.println("print(long)"); }
     */
    /*
     * void print(byte x) { System.out.println("print(byte)"); }
     */

      void print(short x) { System.out.println("print(short)"); }


    /*
     * void print(Integer x) { System.out.println("print(Integer)"); }
     */
    // varying arguments 0 or more
    /*
     * void print(int... x) { System.out.println("print(int...)"); }
     */
}

public class TestMethodOverloading {
    public static void main(String[] args) {
        PrintData p = new PrintData();
        p.print((byte)25); // 25 is treated as in
        // byte -> short -> int -> long -> float -> double
    }
}
```

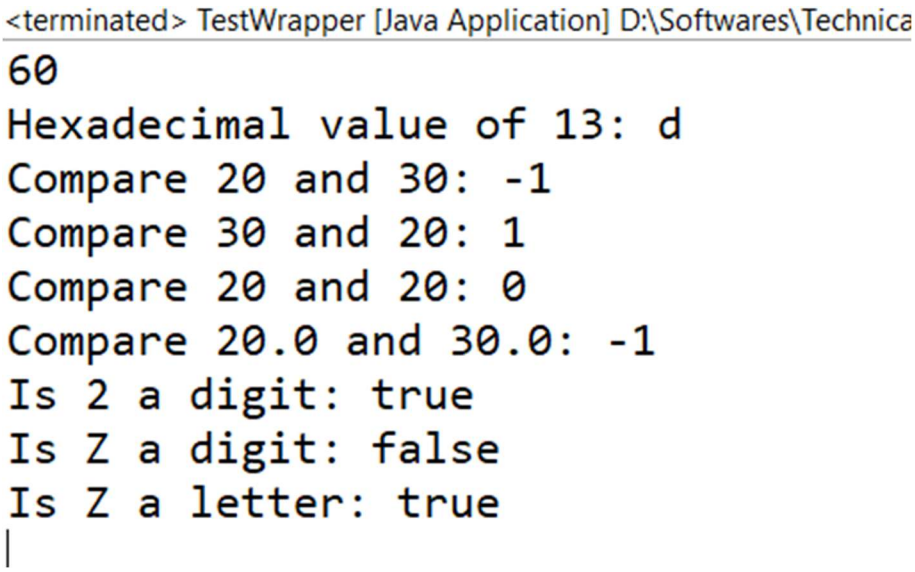
Wrapper classes

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

Person.java | Employee.java | TestInheritance.java | TestMethodOverloading.java | TestWrapper.java

```java
package com.bristlestone;

public class TestWrapper {
    public static void main(String[] args) {
        String s1 = "25";
        String s2 = "35";
        // Integer.parseInt converts string to int
        int num1 = Integer.parseInt(s1);
        int num2 = Integer.parseInt(s2);
        System.out.println(num1 + num2);
        // Integer.toHexString, Integer.toBinaryString, Integer.toOctalString
        System.out.println("Hexadecimal value of 13: "+Integer.toHexString(13));
        // comparing 2 ints - useful while sorting - they give -1, 0, +1
        System.out.println("Compare 20 and 30: "+Integer.compare(20, 30));
        System.out.println("Compare 30 and 20: "+Integer.compare(30, 20));
        System.out.println("Compare 20 and 20: "+Integer.compare(20, 20));
        // compare 2 doubles
        System.out.println("Compare 20.0 and 30.0: "+Double.compare(20.0, 30.0));
        // Character has methods to identify characters property - digit, letter
        System.out.println("Is 2 a digit: "+Character.isDigit('2'));
        System.out.println("Is Z a digit: "+Character.isDigit('Z'));
        System.out.println("Is Z a letter: "+Character.isLetter('Z'));
        // while validating the characters
    }
}
```

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

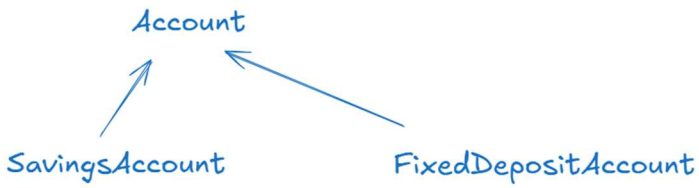
```
class Order {
  private OrderItem item;

   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() { .. }



Primitives

```
int x = 50;
long y = x;  // valid statement

HT = LT // auto-conversion - auto-widening

long z = 25;
int a = z; // invalid statement
int a = (int) z;
LT = (LT)HT; // explicit conversion - explicit widening
```

Derived Types

```
HT = LT;

// auto upcasting
Account acc = new Savings()
acc = new FixedDeposit();
Savings sav = acc; // invalid
Savings sav = (Savings) acc;
// explicit    downcasting
```

Account.java

```java

public class Account {
    private int accountNumber;
    private double balance;

    public Account(int accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }

    public double applyInterest() {
        System.out.println("applyInterest()  in Account class");
        return 0.0;
    }

    public int getAccountNumber() {
        return accountNumber;
    }

    public void setAccountNumber(int accountNumber) {
        this.accountNumber = accountNumber;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }

```

SavingsAccount.java

```java
package com.bristlestone;

public class SavingsAccount extends Account {

    public SavingsAccount(int accountNumber, double balance) {
        super(accountNumber, balance);
    }

    @Override
    public double applyInterest() {
        System.out.println("applyInterest() in Savings Account class");
        return 0.05;
    }
}
```

FixedDepositAccount.java

```java
package com.bristlestone;

public class FixedDepositAccount extends Account {

    public FixedDepositAccount(int accountNumber, double balance) {
        super(accountNumber, balance);
    }

    @Override
    public double applyInterest() {
        System.out.println("applyInterest() in Fixed Deposit Account class");
        return 0.08;
    }
}
```

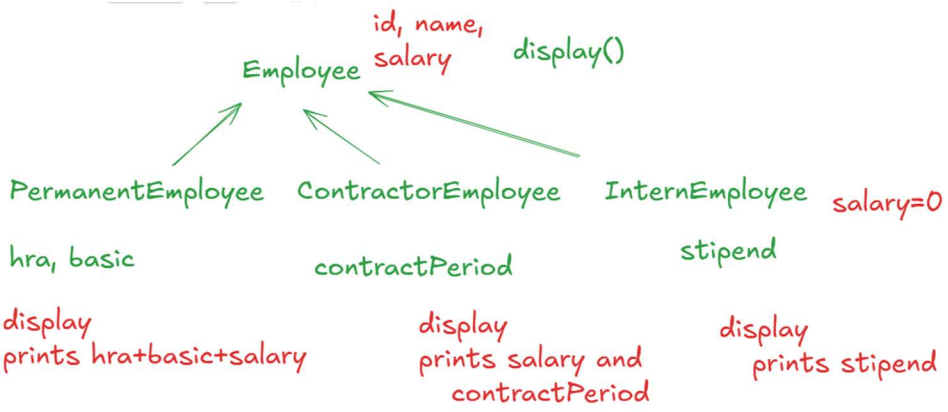
TestPolymorphism.java

```java
package com.bristlestone;



public class TestPolymorphism {
    public static void main(String[] args) {
        Account account1 = new Account(12345, 5000);
        SavingsAccount account2 = new SavingsAccount(98765, 5000);
        FixedDepositAccount account3 = new FixedDepositAccount(567899, 50000);
        printInterest(account1);
        printInterest(account2);
        printInterest(account3);

    }
    // Account class is super class for Savings and FixedDeposit
    public static void printInterest(Account account) { // Account account = account3 -> FixedDepositAccount;
        account.applyInterest(); // runtime polymorphism
        System.out.println("_____");
    }
}
```

Output:

```
<terminated> TestPolymorphism [Java Application] D:\Softwares\Technicals\eclipse-jee-2024-09-R-w
applyInterest()  in Account class
_____-
applyInterest() in Savings Account class
_____-
applyInterest() in Fixed Deposit Account class
_____-
```

Activity



Create an Employee[] array and store 3 different type of objects which are PermanentEmployee in 0th index, ContractorEmployee in 1st index & InternEmployee in 2nd index, iterate the array using for loop and call display() method.

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;

```
public class TestPolymorphism {
    public static void main(String[] args) {
        Account account1 = new Account(12345, 5000);
        SavingsAccount account2 = new SavingsAccount(98765, 5000);
        FixedDepositAccount account3 = new FixedDepositAccount(567899, 50000);
        printInterest(account1);
        printInterest(account2);
        printInterest(account3);

        Account[] acc = new Account[3];
        acc[0] = account1;
        acc[1] = account2;
        acc[2] = account3;

        for(Account a : acc) { // a = account1 = new SavingsAccount();
            a.applyInterest();
            if(a instanceof SavingsAccount) {
                SavingsAccount sa = (SavingsAccount) a; // LT = (LT) HT;   SavingsAccount sa = new Account();
                sa.applyCharges();
            }
        }

    }
    // Account class is super class for Savings and FixedDeposit
    public static void printInterest(Account account) { // Account account = new SavingsAccount(); // SA, Account
        account.applyInterest(); // runtime polymorphism
        System.out.println("_____");

    }
}
```

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
```

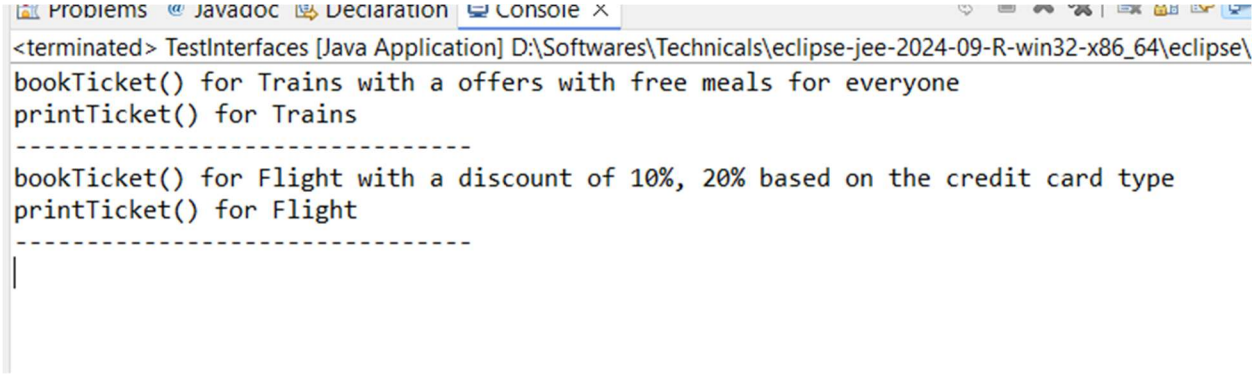
Car c = new Car(); // invalid, because Car is an abstract class

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

```
interface TicketBooking {
    void bookTicket(); // public abstract void bookTicket
    void printTicket(); // public abstract void printTicket
}
class RailwayTicketBooking implements TicketBooking {
    @Override
    public void bookTicket() {
        System.out.println("bookTicket() for Trains with a offers with free meals for everyone");
    }
    @Override
    public void printTicket() {
        System.out.println("printTicket() for Trains");
    }

}
class FlightTicketBooking implements TicketBooking {
    @Override
    public void bookTicket() {
        System.out.println("bookTicket() for Flight with a discount of 10%, 20% based on the credit card type");
    }
    @Override
    public void printTicket() {
        System.out.println("printTicket() for Flight");
    }

}
public class TestInterfaces {
    // Client code
    public static void main(String[] args) {
        RailwayTicketBooking railway = new RailwayTicketBooking();
        FlightTicketBooking flight = new FlightTicketBooking();
        ticketApp(railway); // user1 wants to book & print - train
        ticketApp(flight); // user2 wants to book & print - flight
    }
    // developer1 code that calls bookTicket & printTicket
    public static void ticketApp(TicketBooking ticket) {
        ticket.bookTicket();
        ticket.printTicket();
        System.out.println("--------------------------------");
    }
}

```

Output:



TestAbstractClass.java

TestInterfaces.java | TestAbstractClass.java

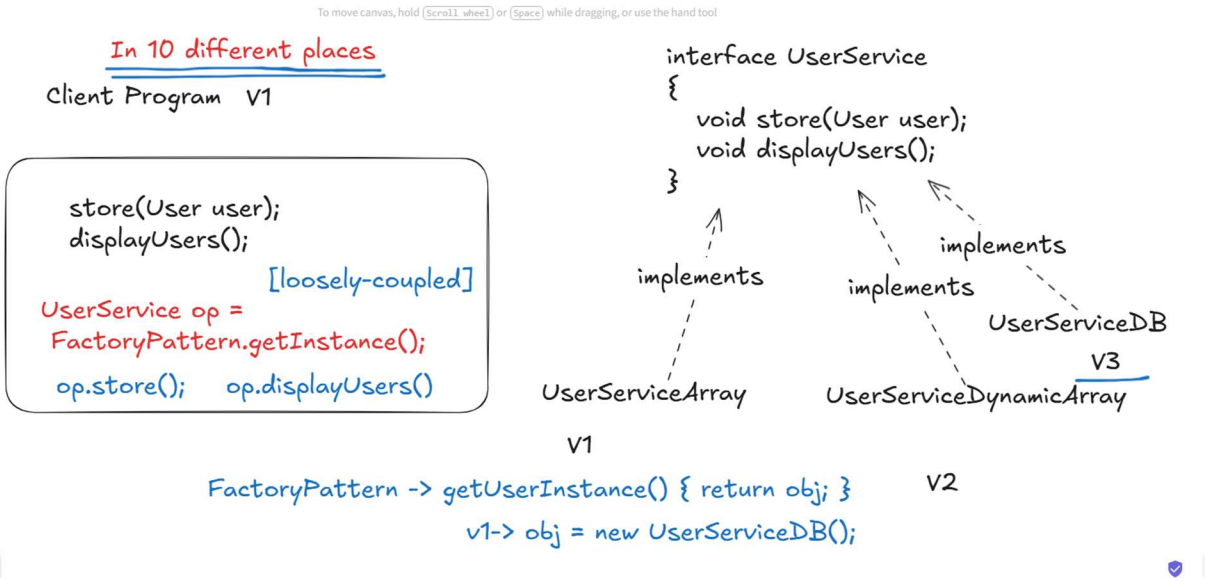
```java
package com.bristlestone;
abstract class Car {
    abstract double getPrice();
    abstract void mileage();
    void wheels() {
        System.out.println("4 wheels");
    }
}
class Creta extends Car {
    @Override
    void mileage() {
        System.out.println("mileage of creta is 15kmpl");
    }
    @Override
    double getPrice() {
        return 1500000;
    }
}
class I20 extends Car {
    @Override
    void mileage() {
        System.out.println("mileage of I20 is 18kmpl");
    }
    @Override
    double getPrice() {
        return 1200000;
    }
}
public class TestAbstractClass {
    public static void main(String[] args) {
        Creta c = new Creta();
        I20 i = new I20();
        printCarInfo(c);
        printCarInfo(i);
    }
    public static void printCarInfo(Car c) {
        c.mileage();
        System.out.println("Price = "+c.getPrice());
        c.wheels();
        System.out.println("-------------");
    }
}

```

Output:

```
<terminated> TestAbstractClass [Java Appl
mileage of creta is 15kmpl
Price = 1500000.0
4 wheels
-------------
mileage of I20 is 18kmpl
Price = 1200000.0
4 wheels
-------------
```

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

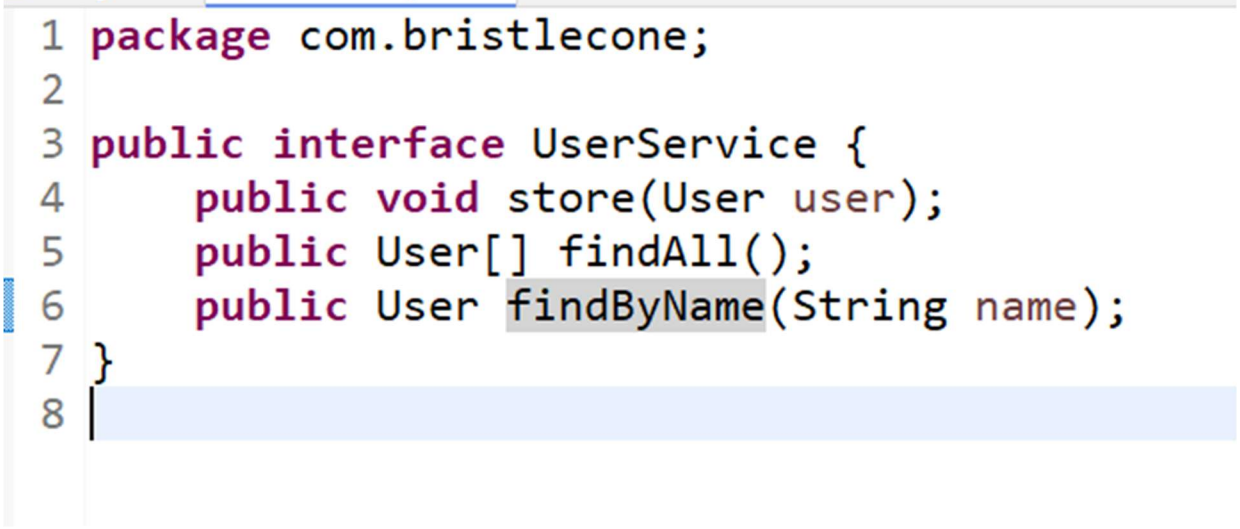
User.java

User.java

```java

public class User {
    private String name;
    private int age;
    public User() {
        super();
    }
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

UserService.java

User.java | UserService.java



```java
package com.bristlecone;

public interface UserService {
    public void store(User user);
    public User[] findAll();
    public User findByName(String name);
}
```

UserServiceArray.java

```java
public class UserServiceArray implements UserService {
    // create a static array so that no matter how many objects of UserServiceArray you create
    // you will have only one copy of the static array
    private static User[] userItems; // initially it will be null
    // counter to track the size of the array
    private static int counter = 0;
    public UserServiceArray(int size) {
        userItems = new User[size]; // allocates memory for the specified size
    }
    @Override
    public void store(User user) {
        if(counter < userItems.length) {
            userItems[counter] = user;
            counter++;
        } else {
            // usually we will throw exception here so that client will know what happened
            System.err.println("Array size is exceeded");
        }

    }
    @Override
    public User[] findAll() {
        // we will copy original array to another array and return that another array
        User[] copy = new User[counter];
        for(int index = 0; index < counter; index++) {
            copy[index] = userItems[index];
        }
        return copy;
    }
    @Override
    public User findByName(String name) {
        // call findAll() that gives only necessary blocks that are initialized
        // find the element from the array
        User[] items = findAll();
        for(User user : items) {
            String existing = user.getName();
            // to compare a string with another string you will use equals() method
            if(existing.equals(name)) {
                return user;
            }
        }
        return null;// null is returned only if no user is found based on the name
    }
```

UserFactory.java

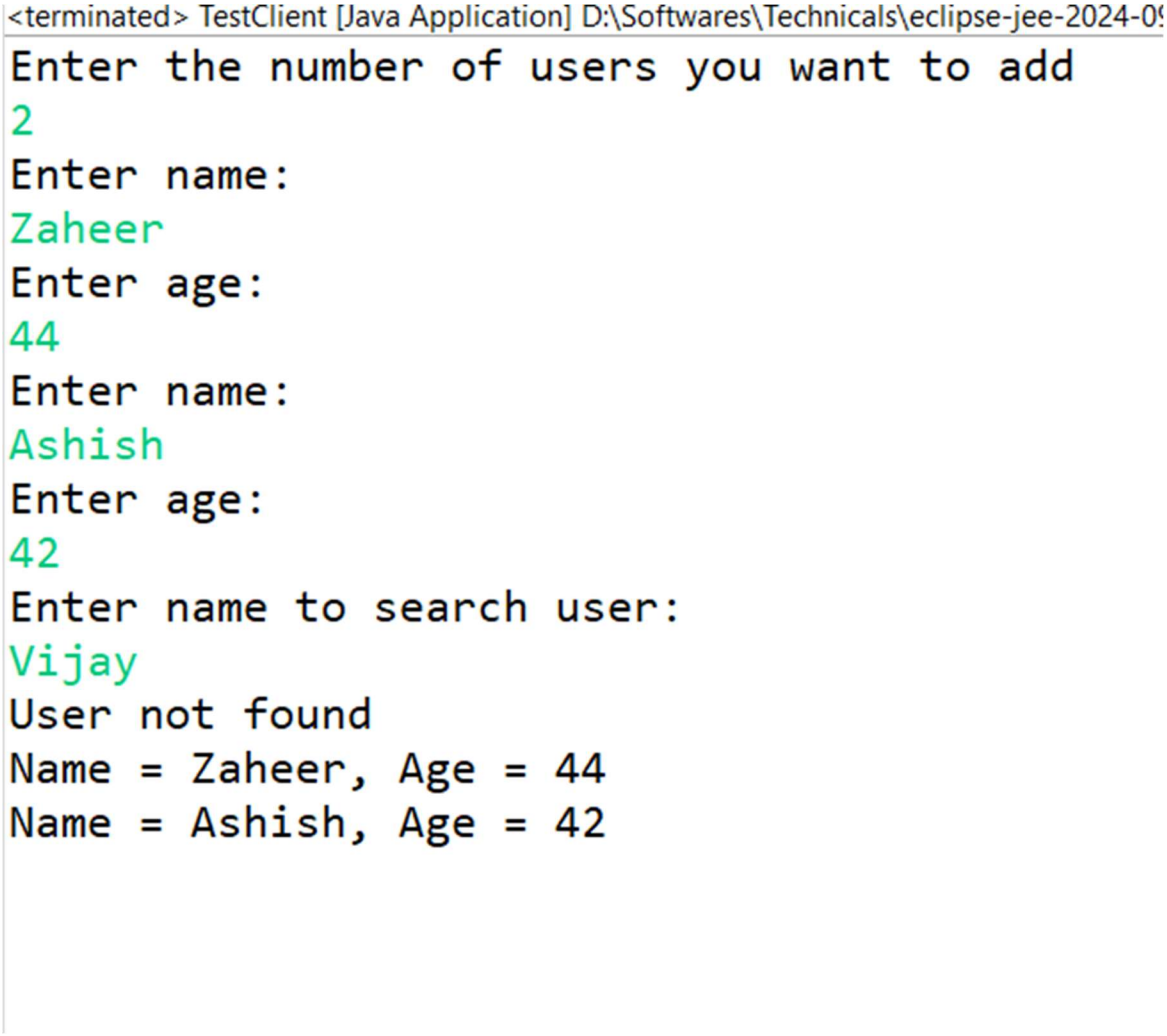
User.java | UserService.java | UserServiceArray.java | UserFactory.java

```java
package com.bristlecone;

public class UserFactory {
    /*
     * This method creates UserService implementation object
     * and returns UserService reference, client uses only
     * interface reference
     * loosely coupled
     * UserService service = UserFactory.getInstance();
     * tightly coupled is
     * UserService service = new UserServiceArray();
     */
    public static UserService getInstance(int size) {
        UserService service = new UserServiceArray(size);
        return service;
    }
}

```

TestClient.java

```java
public class TestClient {
    public static void main(String[] args) {
        /*
         * Client will not have idea they are using which UserService implementation
         */
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the number of users you want to add");
        int size = scan.nextInt();
        // loosely coupled
        UserService service = UserFactory.getInstance(size);
        for(int i = 0; i < size; i++) {
            System.out.println("Enter name:");
            String name = scan.next(); // one word
            System.out.println("Enter age:");
            int age = scan.nextInt();
            User user = new User(name, age);
            // store in the array
            service.store(user);
        }
        //search by name
        System.out.println("Enter name to search user:");
        String name = scan.next();
        User user = service.findByName(name);
        System.out.println(user != null ? "User "+name+" found" : "User not found");
        // display all users
        User[] users = service.findAll();
        for(User u : users)
            System.out.println("Name = "+u.getName()+", Age = "+u.getAge());
        // close the scanner
        scan.close();
    }
```

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

Employee.java
```
package com.bristlecone.beans;

public class Employee {
	private String name;
	private double salary;

	public Employee(String name, double salary) {
		this.name = name;
		this.salary = salary;
	}

	String getName() {
		return name;
	}

	protected double getSalary() {
		return salary;
	}

}
```

Manager.java
```
package com.bristlecone.beans2;

import com.bristlecone.beans.Employee;

public class Manager extends Employee {
	public Manager(String name, double salary) {
		super(name, salary);
	}

	public void display() {
		// you can access getSalary()
		System.out.println("Salary = "+getSalary());
	}
	protected void getEmployeeDetails() {
		System.out.println("— Manager.getEmployeeDetails() —");
	}
}
```

TestMain.java
```
package com.bristlecone.beans2;

public class TestMain {
	public static void main(String[] args) {
		Manager m = new Manager("Raj", 352000);
		m.display();
		// you can't directly access getSalary from the object
		m.display();

	}
}
```

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.

```java
package com.bristlecone;

interface A {
    void m1();
    // from Java 8 onwards default methods & static methods are allowed
    default void test1() {
        System.out.println("test1() has defult implementation");
    }
    default void test2() {
        System.out.println("test2() has default implementation");
    }
    static void test3() {
        System.out.println("test3() is static in A");
    }
}
class Imp implements A {
    public void m1() {
        System.out.println("implemented in a class");
    }
    @Override
    public void test2() {
        System.out.println("test2() is overridden");
    }
}
public class TestJava8Interfaces {
    public static void main(String[] args) {
        A.test3();
        A a = new Imp();
        a.test1();
        a.test2();
    }
}
```

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

UserService.java | UserServiceArra... | TestClient.java | UserFactory.java | Account.java | FixedDepositAc... | TestAbstractCla... | Employee.java

```java
package com.bristlecone;

enum Gender {
    MALE, FEMALE
}
enum LoanType {
    CAR, HOME, PERSONAL, EDUCATIONAL, BUSINESS
}
class Customer {
    private String name;
    private Gender gender;
    private LoanType loanType;

    public Customer(String name, Gender gender, LoanType loanType) {
        this.name = name;
        this.gender = gender;
        this.loanType = loanType;
    }
    public void display() {
        System.out.println("Name="+name+",Gender="+gender+",Loan Type="+loanType);
    }
}

public class TestEnums {
    public static void main(String[] args) {
        Customer customer1 = new Customer("Alex", Gender.MALE, LoanType.PERSONAL);
        Customer customer2 = new Customer("Priya", Gender.FEMALE, LoanType.EDUCATIONAL);
        customer1.display();
        customer2.display();
    }
}

```

Static & Non Static members

How do you access static - members?

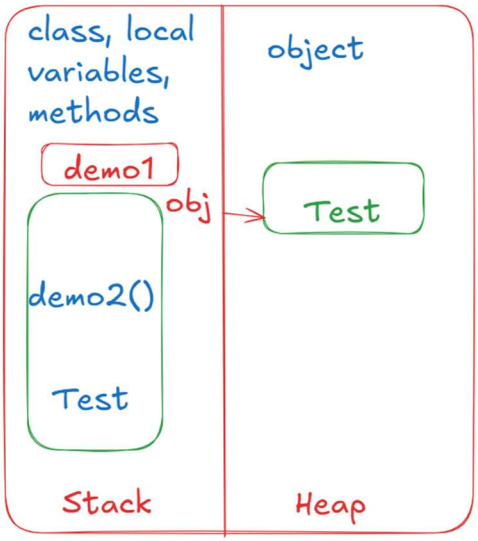
You can use class name and access static members

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.

```
class Test {
   void demo1() { demo2(); }

   static void demo2() {
       Test t = new Test();
         t.demo1(); // ok
    }
}  Test obj = new Test();
```



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

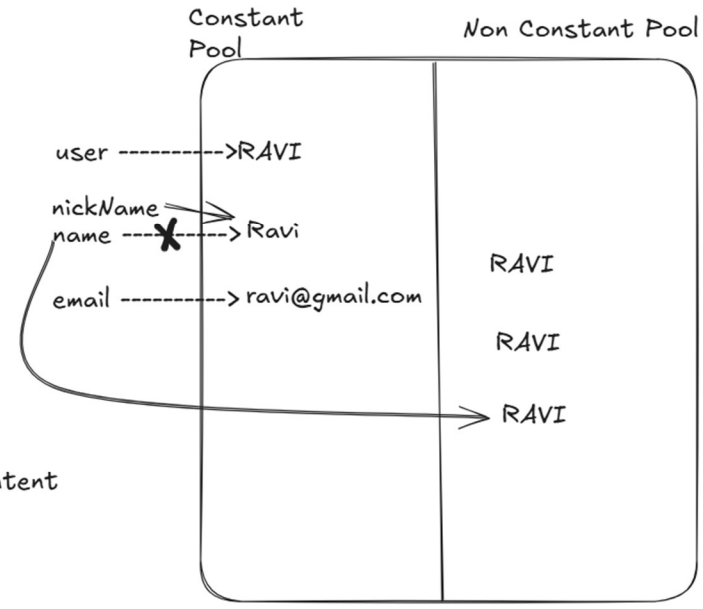
String class

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.

```
String user = "RAVI";
String nickName = "Ravi";
String name = "Ravi";
String email = "ravi@gmail.com";

name.toUpperCase();

name.toUpperCase();

name = name.toUpperCase();

user == name (compares address)
user.equals(name)  -> compares content

int index = email.indexOf("@");

char symbol = email.charAt(index)

email.length() -> give the size of the string
```



TestString.java

```
1  package com.bristlecone;
2
3  public class TestString {
4      public static void main(String[] args) {
5          String email = "ravi@gmail.com";
6          System.out.println("Email = "+email);
7          System.out.println("Contains '@' = "+email.contains("@"));
8          int index = email.indexOf('@');
9          System.out.println("Index of '@' = "+index);
0          System.out.println("Length of "+email+" = "+email.length());
1          int lastIndex = email.indexOf('.');
2          System.out.println("Last index of '.'  = "+lastIndex);
3          String domain = email.substring(index + 1, lastIndex);
4          System.out.println("Domain = "+domain);
5          String username = email.substring(0, index); // non-constant pool
6          System.out.println("Username = "+username);
7          String nickName = "ravi"; // constant pool
8          System.out.println("Username = "+username+", Nick name = "+nickName);
9          System.out.println("Compare == "+(username == nickName));
0          System.out.println("Compare equals = "+username.equals(nickName));
1
2
3
```

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

```java
package com.bristlecone;

public class TestStringBuffer {
    public static void main(String[] args) {
        // toString() is overridden in StringBuffer and StringBuilder to return content
        StringBuffer buffer1 = new StringBuffer("Java");
        StringBuffer buffer2 = new StringBuffer("Java");
        System.out.println("buffer1 = "+buffer1+", buffer2 = "+buffer2);
        System.out.println("buffer1 equals buffer2 = "+buffer1.equals(buffer2));
        buffer1.append(" is easy");
        System.out.println("buffer1 = "+buffer1);
        buffer2.reverse();
        System.out.println("buffer2 = "+buffer2);
        buffer2.insert(2, "TEST");
        System.out.println("buffer2 = "+buffer2);
    }
}

```

Output:

```
<terminated> TestStringBuffer [Java Application] D:\Softwar
buffer1 = Java, buffer2 = Java
buffer1 equals buffer2 = false
buffer1 = Java is easy
buffer2 = avaJ
buffer2 = avTESTaJ
```

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

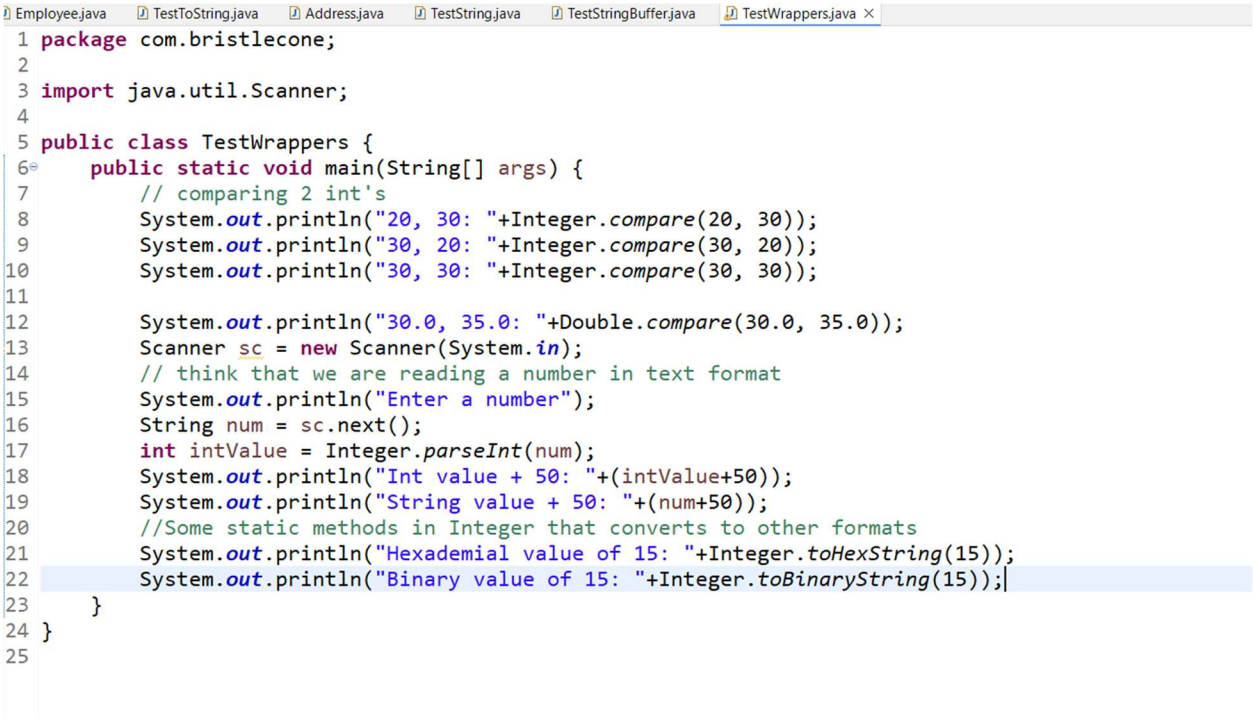
Ex: Integer.compare(20, 30): -ve

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

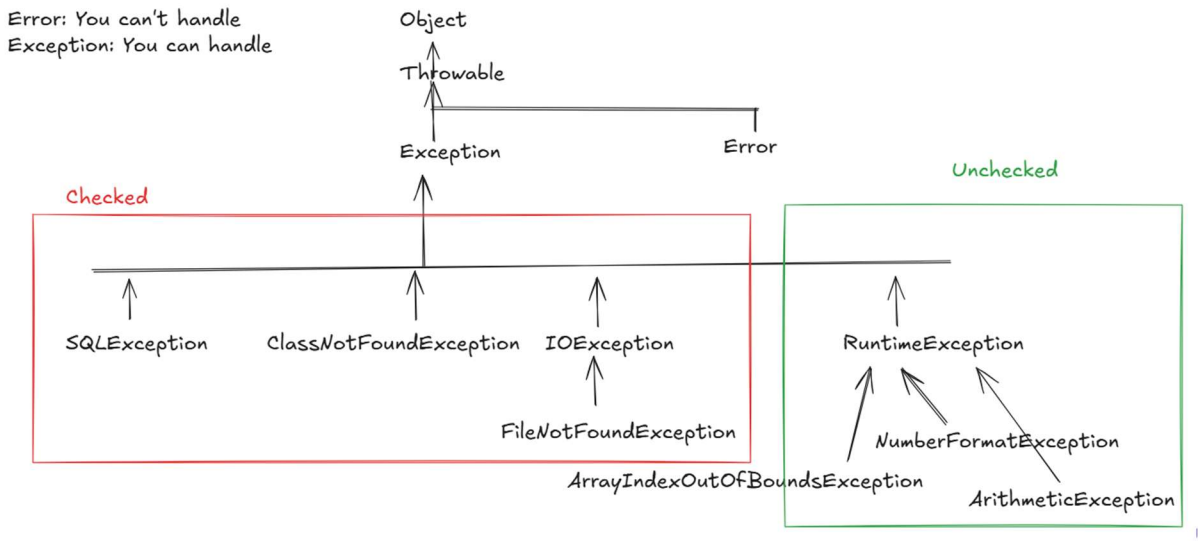
try block: Here we write those codes which may generate an exception, it can be performing DB operations, file io operations, accessing arrays and so on.

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"); }
```

```java
package com.bristlecone;

import java.util.Scanner;

public class TestExceptions {
    public static void findElement(int index) {
        int[] items = {20, 30, 40, 10, 50};
        try {
            System.out.println("Item is : "+items[index]);
            System.out.println("Item found at "+index);
            int x = items[index] / index;
            System.out.println("x = "+x);
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("AIOBE");
        } catch(ArithmeticException e) {
            System.out.println("AE");
        } finally {
            System.out.println("finally block");
        }
        System.out.println("end of findElement");

    }
    public static void main(String[] args) {
        System.out.println("main method starts...");
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter position of the element:-");
        int index = sc.nextInt();
        findElement(index);
        System.out.println("main method ends...");
    }
}

```

Checked & Unchecked exceptions

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

```java
package com.bristlecone;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class TestCheckedExceptions {
    public static void main(String[] args) {
        readFile("aaa.txt");
        System.out.println("main ends..");
    }
    public static void readFile(String file) {
        FileInputStream fis;
        try {
            fis = new FileInputStream(file);
            int ch = fis.read();
            System.out.println(ch);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        System.out.println("done...");
    }
}

```

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

Employee.java | TestToString.java | Address.java | TestString.java | TestStringBuffer.java | TestWrappers.java | TestExceptions.java | TestCheckedExceptions.java

```java
package com.bristlecone;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class TestCheckedExceptions {
    public static void main(String[] args) {
        try {
            readFile("aaa.txt");
        } catch (FileNotFoundException e) {
            System.err.println("File is not found");
        } catch (IOException e) {
            System.err.println("Read is failed");
        }
        System.out.println("main ends..");
    }
    public static void readFile(String file) throws FileNotFoundException, IOException {
        FileInputStream fis;
        fis = new FileInputStream(file);
        int ch = fis.read();
        System.out.println(ch);
        System.out.println("read file done...");
        fis.close();
    }
}
```

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

Employee.java | TestToString.java | Address.java | TestString.java | TestStringBuffer.java | TestWrappers.java | TestExceptions.j

```java
package com.bristlecone;

/*
 * It is a good practice to create minimum 2 constructors
 * in custom exception class
 */
public class AgeInvalidException extends RuntimeException {

    public AgeInvalidException() {
        super("Age is invalid");
    }
    public AgeInvalidException(String err) {
        super(err);
    }
}
```

```java
package com.bristlecone;

import java.util.Scanner;

public class TestCustomException {
    public static void main(String[] args) {
        /*
         * If age is valid we will process further else throw AgeInvalidException
         */
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter age:-");
        int age = scan.nextInt();
        try {
            if(age < 0) {
                throw new AgeInvalidException("Age cannot be negative");
            }
            if(age < 18) {
                throw new AgeInvalidException("Age must be minimum 18");
            }
            System.out.println("Valid age to vote!");
        } catch(AgeInvalidException e) {
            System.err.println(e.getMessage());
        }
        System.out.println("Program exited!");
    }
}
```

Output:

<terminated> TestCustomException [Java Application] D:\Softwares\Tech

```
Enter age:-
17
Program exited!
Age must be minimum 18
```

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();
}
```

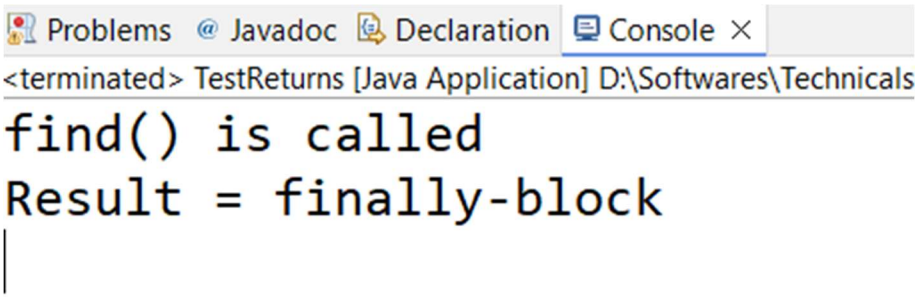
```java
package com.bristlecone;

import java.io.IOException;
import java.sql.SQLException;
import java.util.Scanner;

public class TestMultiCatch {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.println("Enter some input:");

        try {
            int num = scan.nextInt();
            if(num > 5) {
                throw new SQLException();
            } else {
                throw new IOException();
            }
        } catch(IOException | SQLException e) {
            System.out.println("multi-catch");
            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

```java
package com.bristlecone;

import java.sql.SQLException;

public class TestReturns {
    @SuppressWarnings("finally")
    public static String find(int n) {
        try {
            System.out.println("find() is called");
            if(n < 0) {
                throw new SQLException("n is negative");
            }
            return "try-block";
        } catch(SQLException e) {
            return "catch-block";
        } finally {
            return "finally-block";
        }
    }
    public static void main(String[] args) {
        String result = find(-1);
        System.out.println("Result = "+result);
    }
}
```

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

Employee.java | TestCheckedE... | AgeInvalidEx... | TestCustomEx... | TestOverridi... | TestMultiCa...

```java
package com.bristlecone;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

public class TryWithResource {
    public static void main(String[] args) {

        try (Scanner sc = new Scanner(System.in);) {
            System.out.println("Enter num: ");
            int n = sc.nextInt();
            System.out.println("N = "+n);
        }


    }
}
```