Ahsanullah University of Science and Technology (AUST)
Department of Computer Science and Engineering

LABORATORY MANUAL

Course No. : CSE1206
Course Title: Object Oriented Programming Lab

For the students of 1st Year, 2nd Semester of
B.Sc. in Computer Science and Engineering program

Session 1

OBJECTIVES: Create a simple program using java, compile and run using command prompt., How to install Eclipse java editor/Netbean, First program on Eclipse/Netbean, Java variables, data types, data input, arrays and control statements, Write a method in Java, Taking input from keyboard.

```
/* Instruction: Observe */
/* A program to demonstrate taking input from keyboard and calculate the sum using method */

import java.util.Scanner;

public class Hello {

   public static int doSum(int number1,int number2)
   {
      int sum;
      sum=number1+number2;
      return sum;
   }

   public static void main(String[] args) {
      // TODO code application logic here
      Scanner input=new Scanner(System.in);
      int firstNumber,secondNumber,sum;

      System.out.print("Enter first number: ");
      firstNumber=input.nextInt();
      System.out.print("Enter second number: ");
      secondNumber=input.nextInt();

      sum=doSum(firstNumber,secondNumber);
      System.out.println("We have calculated the sum of "+firstNumber+" and "+secondNumber+". The
result is= "+sum);
   }

}
```

Exercises:

1. Write a program to take two numbers from user and calculate their sum, subtraction and multiplication result.
2. Write a program to develop a calculator using a method **calculator(int choice)** and based on that choice, calculate the result.

Session 2

OBJECTIVES: Class and Object , Class with more than one methods, More than one class, Passing value, Constructors, Parameterized Constructors, Returning a Value, The this Keyword, Constructor overloading.

```
class Student {
   private String name;
   private String department;
   private float cgpa;
   public Student() //Constructor
   {
      name="Rahim";
      department="CSE";
      cgpa=(float) 3.25;
   }
   public Student(String name,String department, float cgpa) //constructor overloading
   {
      this.name=name;
      this.department=department;
      this.cgpa=cgpa;
   }
   public void set_name(String name) //Passing value
   {
      this.name=name;
   }
   public String get_name() //Returning a value
   {
      return this.name;
   }
   public void set_department(String department)
   {
      this.department=department;
   }
   public String get_department()
   {
      return this.department;
   }
   public void set_cgpa(float cgpa)
   {
      this.cgpa=cgpa;
   }
```

```
    public float get_cgpa()
    {
        return this.cgpa;
    }
    public void display()
    {
        System.out.println("In display function");
        System.out.println("Name: "+name);
        System.out.println("Department: "+department);
        System.out.println("CGPA: "+cgpa);
    }
}
public class Test {

    public static void main(String[] args) {
        // TODO code application logic here
        Student s=new Student();
        System.out.println("Using Constructor function");
        s.display();
        Student s1=new Student("Zahin","CSE", (float) 3.45);
        s1.display();
        s1.set_name("Jahin");
        System.out.println("After changing the 2nd student's name using set_name() method");
        s1.display();
    }

}
```

Exercises:

1. Write a program with Employee class which has four private instances: name, salary, designation and age. Write necessary getter and setter methods for all the four instances. Write a method printEmployee()to print all the instances of that class. Then create three Employee class object and print their member instances.

Assessments:

1. Online Test-1

Session 3

OBJECTIVES: Using Objects as Parameters, Returning Objects

```
/*Using Object as parameters*/

class Test {
   int a, b;
  Test(int i, int j) {
      a = i;
      b = j; }
 boolean equalTo(Test o) {
      return (a == o.a && b == o.b); }
     }
 class PassOb{
   public static void main(String args[]){
      Test ob1 = new Test(100, 22);
      Test ob2 = new Test(100, 22);
      Test ob3 = new Test(-1, -1);
      System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
      System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));  }
}
```

```
/*Returning Object*/

class  ObjectReturnDemo  {
        int a;
        int b;
       ObjectReturnDemo(int i, int j) {
             a = i;
             b = j;
        }
       ObjectReturnDemo incrByTen() {
              ObjectReturnDemo temp = new ObjectReturnDemo(a+10,b+10);
              return temp;
             }
         }

class Test2{
        public static void main(String args[]) {
```

```
        ObjectReturnDemo ob1 = new ObjectReturnDemo(2,3);
        ObjectReturnDemo ob2;
        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a+" "+ob1.b);
    System.out.println("ob2.a: " + ob2.a +" "+ob2.b);
  }
}
```

Exercises:

1. Write a class A which has two member instances: a(int) and b(int). Write another class B which has a method sum which takes two arguments of type A and returns a objects of type A where the returned object's a is equal to the sum of two arguments' a and b is equal to the sum of two arguments' b.

2. Complete the following code:

```
class A
{
   int a;
   A(int a)              //set the value of instance according to the argument
   {
   }
}
class B
{
   A square_a(A ob)      //square the value of the object's instance
   {

   }
   void show(A ob)       //Print the object's instance
   {

   }
}
public class Test2 {
   public static void main(String[] args) //Create object of the class and test the methods
  {

   }
}
```

Session 4

OBJECTIVES: Method Overloading by changing number of arguments and by changing data types.

Method Overloading

Overloading allows different methods to have same name, but different signatures where signature can differ by number of input parameters or type of input parameters or both. Overloading is related to compile time (or static) polymorphism.

What is the advantage?

We don't have to create and remember different names for functions doing the same thing. For example, in our code, if overloading was not supported by Java, we would have to create method names like sum1, sum2….or sum2Int, sum3Int, … etc.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

```
/*Changing no. of arguments*/
1. class AdderN{
2. static int add(int a,int b){return a+b;}
3. static int add(int a,int b,int c){return a+b+c;}
4. }
5. class TestOverloading1{
6. public static void main(String[] args){
7. System.out.println(AdderN.add(11,11));
8. System.out.println(AdderN.add(11,11,11));
9. }}
```

```
/*Changing data types of arguments*/
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
System.out.println(add(10,5));
System.out.println(add(5.5,5.5));
}
static double add(double a, double b){return a+b;}
static int add(int a,int b){return a+b;}
}
```

Exercises:

1. Create a class to print the sum of two numbers. The class has two methods with the same name but different types of parameters. One method for printing sum has two parameters of integer type while the other method for printing sum has two parameters of type long.
2. Create a class to print the sum of numbers. The class has two methods with the same name but different types and number of parameters. One method for printing sum has two parameters of integer and long type while the other method for printing sum has three parameters of type int. Now call the method with two integer parameters and check which method gets called.
3. Create a class to print the sum of two numbers. The class has four methods with the same name but different types of parameters. One method for printing sum has two parameters of integer type, another method for printing sum has two parameters of type double, another method has two parameters of type int and double and the last one has two parameters of type double and int.

Assessments:

1. Online Test-2

Session 5

OBJECTIVES: Type Conversion and Casting, One-Dimensional Arrays, Multidimensional Arrays

Single Dimensional Array in java: Syntax to Declare an Array in java

```
dataType[] arr; (or)
    1. dataType []arr; (or)
    2. dataType arr[];
```

Instantiation of an Array in java

```
arrayRefVar=new datatype[size];
```

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

int a[]=new int[5];

a[0]=10;
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
for(int i=0;i<a.length;i++) //length is the property of array
System.out.println(a[i]);
 }}
```

```
public class Testarray1 {

public static void main(String args[]){

int a[]={33,3,4,5};

for(int i=0;i<a.length;i++)
System.out.println(a[i]);
```

```
        }
}
```

```
public class Testarray2 {

   static void min(int arr[]){
   int min=arr[0];
   for(int i=1;i<arr.length;i++)
      if(min>arr[i])
      min=arr[i];

   System.out.println(min);
}

public static void main(String args[]){

int a[]={33,3,4,5};
min(a);

}}
```

Multidimensional array in java

1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];

```
int[][] arr=new int[3][3];
```

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

int arr[][]={{1,2,3},{2,4,5},{4,4,5}};

for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
```

```
  System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}

}}
```

```
/*add to matrices*/
public class TwoD {
   public static void main(String args[]){

          int a[][]={{1,3,4},{3,4,5}};
          int b[][]={{1,3,4},{3,4,5}};

          int c[][]=new int[2][3];

                for(int i=0;i<2;i++){
                      for(int j=0;j<3;j++){
                           c[i][j]=a[i][j]+b[i][j];
                           System.out.print(c[i][j]+" ");
                       }
                System.out.println();
               }
        }
     }
```

Exercises:

1. Write a java program to multiply two matrices.
2. Write a Java program to take the dimension and values of two matrices from user and calculate their subtraction result.

Session 6

OBJECTIVES: Introducing the Inheritance, The extend keyword, single level inheritance.

Syntax of Java Inheritance

```
1. class Subclass-name extends Superclass-name
2. {
3.   //methods and fields
4. }
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

```
class Employee{
float salary=40000;
}
class Programmer extends Employee{
int bonus=10000;
public static void main(String args[]){
  Programmer p=new Programmer();
  System.out.println("Programmer salary is:"+p.salary);
  System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

```
/*Single Inheritance Example*/
1.
2. class Animal{
3. void eat(){System.out.println("eating...");}
4. }
5. class Dog extends Animal{
6. void bark(){System.out.println("barking...");}
7. }
8. class TestInheritance{
9. public static void main(String args[]){
10. Dog d=new Dog();
d.bark();
d.eat();
}}
```

```
/*Multilevel Inheritance Example*/

class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(StSyntax of Java Inheritancering args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

```
/*Hierarchical Inheritance Example*/

class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
}}
```

Assessments:

1. Online Test-3

Session 7

OBJECTIVES: Inheritance, Use of Super, Multilevel Hierarchy , Method Overriding.

Using Super

Super has two general forms.

Using super to Call Superclass Constructors

```
class Box {
   double width, height, depth;
   Box(double w, double h, double d){
      width=w;
      height=h;
      depth=d;
   }
}
class BoxWeight extends Box{
   double weight;
   BoxWeight(double w, double h, double d, double m) {
      super(w, h, d);
      weight=m;
       }
  }
```

Using super to access the method of superclass when subclass has hidden that method.

```
class A{
   int i;
   }
class B extends A{
   int i;
   B(int a,int b){
      super.i=a;
      i=b;
   }
   void show(){
      System.out.println(" i in super class: "+ super.i);
      System.out.println(" i in sub class: "+ i);
   }
}
class UseSupper{
```

```
    public static void main(String args[]){
        B subOb = new B(1,2);
        subOb.show();
     }
}
```

Multilevel Hierarchy

Order of Constructor calling

```
  class A {
    A ( ){
      System.out.println("Inside A's Constructor");
    }
}

class B extends A {
    B ( ){
      System.out.println("Inside B's Constructor");
    }
}

class C extends B {
    C ( ){
      System.out.println("Inside C's Constructor");
    }
}

class CallingCons{
    public static void main(String args[]){
        C c = new C( );
    }
}
```

Method Overriding

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding**.

In other words, If subclass provides the specific implementation of the method that has been provided by one of its super class, it is known as method overriding.

Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

Real example of Java Method Overriding

Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, BRAC, DBBL and HSBC banks could provide 8%, 7% and 9% rate of interest.

```
class Bank{
int getRateOfInterest(){return 0;}
}

class BRAC extends Bank{
int getRateOfInterest(){return 8;}
}

class DBBL extends Bank{
int getRateOfInterest(){return 7;}
}
class HSBC extends Bank{
int getRateOfInterest(){return 9;}
}

class Test2{
public static void main(String args[]){
BRAC b=new BRAC();
DBBL d=new DBBL();
HSBC h=new HSBC();
System.out.println("BRAC Rate of Interest: "+b.getRateOfInterest());
System.out.println("DBBL Rate of Interest: "+d.getRateOfInterest());
System.out.println("HSBC Rate of Interest: "+h.getRateOfInterest());
}
}
```

Exercises:

1. Create a class 'Degree' having a method 'getDegree' that prints "I got a degree". It has two subclasses namely 'Undergraduate' and 'Postgraduate' each having a method with the same

name that prints "I am an Undergraduate" and "I am a Postgraduate" respectively. Call the method by creating an object of each of the three classes.

Assessments:

1. Online Test-4

Session 8

OBJECTIVES: Abstract class, Abstract method.

Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain *abstract methods*, i.e., methods without body (public void get();)
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Example abstract class: abstract class A{}

```
abstract class GraphicObject {

  // declare fields
  // declare nonabstract methods
}
```

Abstract Methods

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.

- **abstract** keyword is used to declare the method as abstract.
- You have to place the **abstract** keyword before the method name in the method declaration.
- An abstract method contains a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semoi colon (;) at the end.

Example abstract method

abstract void printStatus(); //no body and abstract

```
public abstract class GraphicObject {

  // declare fields
  // declare nonabstract methods
```

```
  abstract void draw();
}
```

Exercises:

Exercises:

1. Complete the following block of code…..

```
abstract class Employee {
  private String name;
  private String address;
  private int number;

  Employee(String name, String address, int number) {

  }

  abstract double computePay();

  void mailCheck() {
    System.out.println("Mailing a check to " + this.name + " " + this.address);
  }

   String getName() {
    return name;
  }

   String getAddress() {
    return address;
  }

  int getNumber() {
    return number;
  }
}

class Salary extends Employee {
  private double salary;   // Annual salary

  Salary(String name, String address, int number, double salary) {
        //……..
        setSalary(salary);
  }
```

```
   void mailCheck() {
     System.out.println("Within mailCheck of Salary class ");
     System.out.println("Mailing check to " + getName() + " with salary " + salary +
                                " and check no: " +getNumber() );
         }

   double getSalary() {
     return salary;
   }

   void setSalary(double newSalary) {
     if(newSalary >= 0.0) {
       salary = newSalary;
     }
   }

   double computePay() {
     System.out.println("Computing monthly salary pay for " + getName());
     return salary/12;
   }
}

class AbstractDemo {

   public static void main(String [] args) {
     Salary s = new Salary("Zakia", "Mirpur", 3, 360000.00);
     Employee e = new Salary("Tania", "Gulshan", 2, 2400000.00);
     System.out.println("Call mailCheck using Salary reference --");
     s.mailCheck();
     System.out.println(s.computePay());
     System.out.println("\n Call mailCheck using Employee reference--");
     e.mailCheck();
     System.out.println(e.computePay());
   }
}
```

Session 9

OBJECTIVES: Interface, Declaring Interfaces, Extending Multiple Interfaces, Multiple inheritance in Java by interface.

An **interface in java** is a blueprint of a class. It has static constants and abstract methods. The interface in java is **a mechanism to achieve abstraction**.

An interface is similar to a class in the following ways –

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including –

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface –

Example

Following is an example of an interface –

```
public interface NameOfInterface {

  // Any number of final, static fieldsi

  // Any number of abstract method declarations\

}
```

Interfaces have the following properties –

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Example

```
/* File name : Animal.java */

interface Animal {

  public void eat();

  public void travel();

}
```

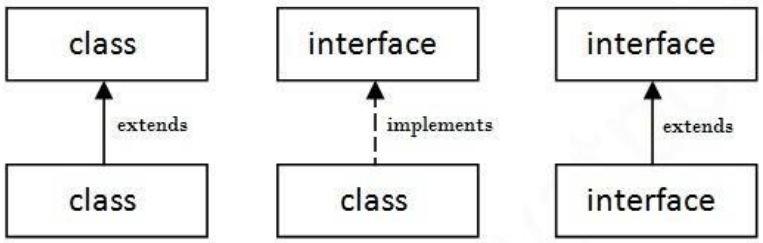
Advantages of interface in java:

Advantages of using interfaces are as follows:

1. Without bothering about the implementation part, we can achieve the security of implementation
2. In java, **multiple inheritance** is not allowed, however you can use interface to make use of it as you can implement more than one interface.

Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



Java Interface Example

In this example, Printable interface has only one method, its implementation is provided in the A class.

```
interface printable{
```

```
void print();

}

class A implements printable{

public void print(){System.out.println("Hello");}

public static void main(String args[]){

A obj = new A();

obj.print();

 }

}
```

Java Interface Example: Drawable

In this example, Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In real scenario, interface is defined by someone but implementation is provided by different implementation providers. And, it is used by someone else. The implementation part is hidden by the user which uses the interface.

```
interface Drawable{

void draw();

}

//Implementation: by second user

class Rectangle implements Drawable{

public void draw(){

   System.out.println("drawing rectangle");

   }

void display(){

   System.out.println("\n Inside Rectangle");

   }

}
```

```
class Circle implements Drawable{
public void draw(){
   System.out.println("drawing circle");
   }
void display(){
   System.out.println("\n Inside Circle");
   }
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
//Drawable c=new Circle();
//Drawable r=new Rectangle();
Circle c=new Circle();
Rectangle r=new Rectangle();
c.display();
c.draw();
r.display();
r.draw();
}}
```

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as −

```
public interface Hockey extends Sports, Event
```

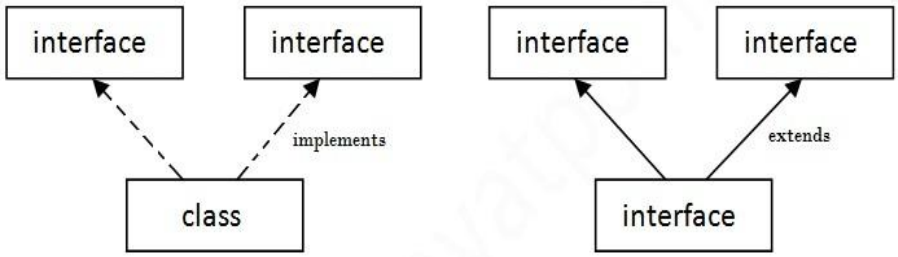
Example

```
// Filename: Sports.java
public interface Sports {
  public void setHomeTeam(String name);
  public void setVisitingTeam(String name);
}

// Filename: Football.java
public interface Football extends Sports {
  public void homeTeamScored(int points);
  public void visitingTeamScored(int points);
  public void endOfQuarter(int quarter);
}
// Filename: Hockey.java
public interface Hockey extends Sports {
  public void homeGoalScored();
  public void visitingGoalScored();
  public void endOfPeriod(int period);
  public void overtimePeriod(int ot);
}
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class Multiple implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
 public static void main(String args[]){
Multiple obj = new Multiple();
obj.print();
obj.show();
 }
}
```

Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class because of ambiguity. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class.

Exercises:

1. Create an interface “Student” with the following methods:
    a. void setName (String name)
    b. void setInstitution (String institution)
    c. void setCGPA (double cgpa)
    d. void setAddress (String address)

   Create an interface “Employee” with the following methods:
    a. void setOrganization (String organization)
    b. void setSalary (int salary)

   Create a class “People” which implements the above interfaces.

2. Create an interface “Sports” with the following methods:
    a. void setHost (String host)
    b. void setOpponent (String opponent)

   Create an interface “Football” with the following methods:
    a. void setHomeScore (int goal)
    b. void setOppScore (int goal)
    c. void passedTime (int time)
    d. void setFaulNumber (int faul)
    e. String toStirng()

   Create an interface “Cricket” with the following methods:
    a. void setHomeScore (int run)
    b. void setOppScore (int run)
    c. void setHomeWicket (int Wicket)
    b. void setOppWicket (int wicket)

   Create a class “WC_Football” which implements the interface “Football” and another class “WC_Cricket” to implement the interface “Cricket”.

Assessments:

1. Online Test-5

Session 10

OBJECTIVES: Exception, Checked and unchecked exception, exception hierarchy, try and catch block, multiple catch block, nested catch block.

Exceptions

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Based on these, we have three categories of Exceptions.

Checked exceptions − A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

```
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

  public static void main(String args[]) {
    File file = new File("E://file.txt");
    FileReader fr = new FileReader(file);
  }
}
```

Output

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or
declared to be thrown
    FileReader fr = new FileReader(file);
                ^
1 error
```

Unchecked exceptions – An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

Example

```
public class Unchecked_Demo {
  public static void main(String args[]) {
    int num[] = {1, 2, 3, 4};
    System.out.println(num[5]);
  }
}
```

Output

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

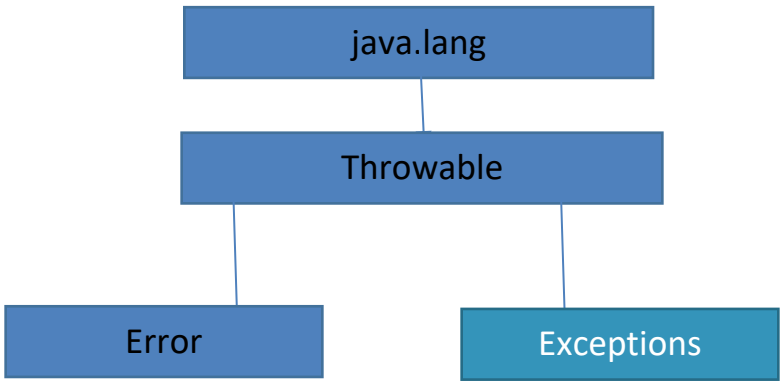
Errors – These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Java defines several exception classes inside the standard package java.lang.

The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.



Unchecked RuntimeException Defined in java.lang.

Sr.No.	Exception & Description
1	**ArithmeticException** Arithmetic error, such as divide-by-zero.
2	**ArrayIndexOutOfBoundsException** Array index is out-of-bounds.
3	**ArrayStoreException** Assignment to an array element of an incompatible type.
4	**ClassCastException** Invalid cast.
5	**IllegalArgumentException** Illegal argument used to invoke a method.
6	**IllegalMonitorStateException** Illegal monitor operation, such as waiting on an unlocked thread.
7	**IllegalStateException** Environment or application is in incorrect state.
8	**IllegalThreadStateException** Requested operation not compatible with the current thread state.
9	**IndexOutOfBoundsException** Some type of index is out-of-bounds.
10	**NegativeArraySizeException** Array created with a negative size.

11	**NullPointerException** Invalid use of a null reference.
12	**NumberFormatException** Invalid conversion of a string to a numeric format.
13	**SecurityException** Attempt to violate security.
14	**StringIndexOutOfBounds** Attempt to index outside the bounds of a string.
15	**UnsupportedOperationException** An unsupported operation was encountered.

Checked Exceptions Defined in java.lang.

Sr.No.	Exception & Description
1	**ClassNotFoundException** Class not found.
2	**CloneNotSupportedException** Attempt to clone an object that does not implement the Cloneable interface.
3	**IllegalAccessException** Access to a class is denied.
4	**InstantiationException** Attempt to create an object of an abstract class or interface.
5	**InterruptedException** One thread has been interrupted by another thread.

6	**NoSuchFieldException** A requested field does not exist.
7	**NoSuchMethodException** A requested method does not exist.

try and catch block

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method. Java try block must be followed by either catch or finally block.

Syntax of java try-catch:

```
try{
    //code that may throw exception
    }
catch(Exception_class_Name ref){}
```

Syntax of try-finally block:

```
try{
    //code that may throw exception
  }
finally{}
```

Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only. You can use multiple catch block with a single try.

Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

```
1.  public class Testtrycatch1{
2.    public static void main(String args[]){
3.      int data=50/0;//may throw exception
4.      System.out.println("rest of the code...");
5.       }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
```

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

Let's see the solution of above problem by java try-catch block.

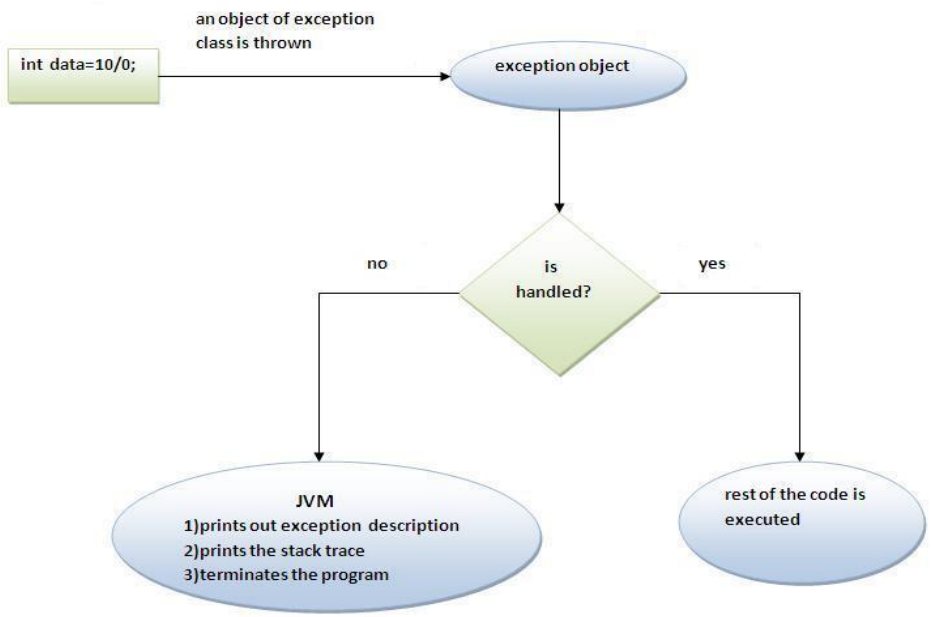
```
1.  public class Testtrycatch2{
2.    public static void main(String args[]){
3.     try{
4.       int data=50/0;
5.      }
6.     catch(ArithmeticException e){
7.       System.out.println(e);
8.      }
9.
System.out.println("rest of the code...");
      }
 }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

Internal working of java try-catch block



Multiple catch block

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block. Let's see a simple example of java multi-catch block.

```
1.  public class TestMultipleCatchBlock{
2.       public static void main(String args[]){
3.       try{
4.            int a[]=new int[5];
5.            a[5]=30/0;
6.          }
7.      catch(ArithmeticException e){System.out.println("task1 is completed");}
8.      catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
9.      catch(Exception e){System.out.println("common task completed");}
10.
 System.out.println("rest of the code...");
  }
}
```

Output: task1 completed
rest of the code...

Nested try block

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
1. ....
2. try
3. {
4.     statement 1;
5.     statement 2;
6.     try
    {
     statement 1;
7.         statement 2;
    }
  catch(Exception e){ }
 }
catch(Exception e){ }
8. ....
9.
```

Java nested try example

Let's see a simple example of java nested try block.

```
1.  class Excep6{
2.    public static void main(String args[]){
3.    try{
4.      try{
5.       System.out.println("going to divide");
6.       int b =39/0;
7.      }
8.      catch(ArithmeticException e){System.out.println(e);}
9.      try{
     int a[]=new int[5];
    a[5]=4;
   }
  catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

  System.out.println("other statement);
 }
catch(Exception e){System.out.println("handeled");}

 System.out.println("normal flow..");
   }
}
```

Exercises:

1. Write a JAVA code to take input from user and divide 100 by that input. You have to continue this process until the user presses zero. When 100 is divided by zero, print a message that "cannot divide by zero".

2. Write a JAVA code to take string from user and convert it to integer. If the string cannot be converted to integer, inform the user about this occurance.

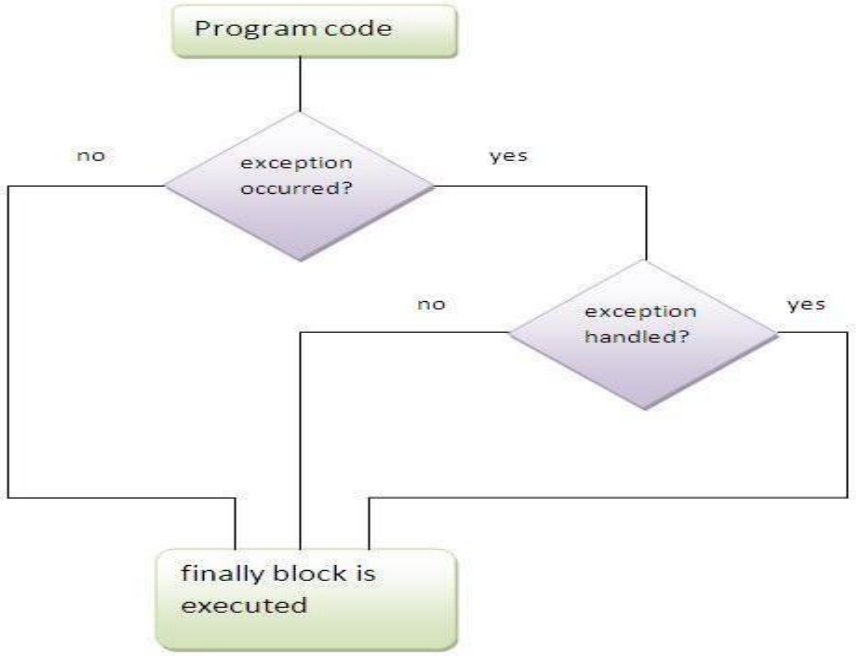
Session 11

OBJECTIVES: finally block, User-defined exception, throws keyword

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.



Why use java finally

○ Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception.

The syntax of java throw keyword is given below.

throw ThrowableInstance;

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

throw new IOException("demo");

```
class TestThrow1{
  static void validate(int age){
    try{
     if(age<18)
```

```
    throw new ArithmeticException("not valid");
   else
    System.out.println("welcome to vote");
   }
   catch(ArithmeticException e){
       System.out.println("Exception: "+ e);
   }
 }
  public static void main(String args[]){
    validate(33);
    System.out.println("rest of the code...");
 }
}
```

throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Syntax of java throws

```
return_type method_name() throws exception_class_name{
 //method code
}
```

Which exception should be declared?

checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

```
import java.io.IOException;

class Testthrows1{
```

```
 void m() throws IOException{
  throw new IOException("device error");
 }
void n() throws IOException{
   m();
 }
 void p(){
  try{
   n();
  }
  catch(IOException e){
      System.out.println("exception handled: " +e);
  }
 }
 public static void main(String args[]){
  Testthrows1 obj=new Testthrows1();
  obj.p();
  System.out.println("normal flow...");
 }
}
```

Exercises:

1. Write a Java code segment that will take a sequence of positive integer numbers as input from the keyboard and find the summation of the odd numbers only. If the input is a negative number, your code segment should throw a user-defined exception. The main() method should handle this exception and print the error message.
2. Write a Java program that will take two integer numbers as input from the keyboard. Your program should determine whether the first number is a multiple of the second number.

Your program should provide checking for the following cases:

1. If any of the two numbers is negative.
2. If the first number is smaller than the second number.
3. If the second number is 0.

You should define appropriate exception class for each of the cases and throw an instance of the correct exception when any of the condition arises.

Session 12

OBJECTIVES: String Class, StringBuffer class, constructors and methods.

String Class

In java, string is basically an object that represents sequence of char values. An array of characters works same as java string. For example:

```
char[] ch={'j','a','v','a'};
String s=new String(ch);
```

is same as:

```
String s="java";
```

Java String class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters. The java.lang.String class is used to create string object.

How to create String object?

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

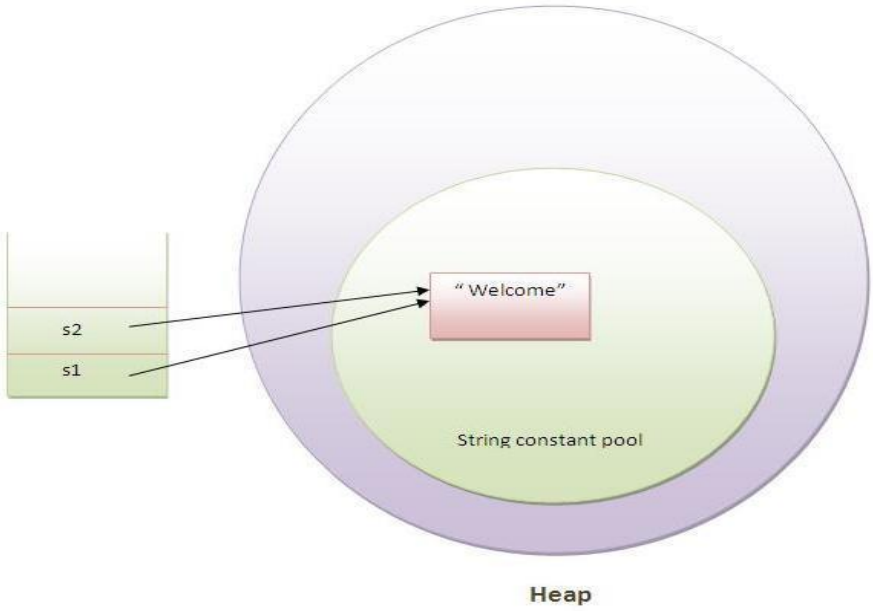
Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
```

String s2="Welcome";//will not create new instance



In the above example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in string constant pool, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance.

Why java uses concept of string literal?

To make Java more memory efficient (because no new objects are created if it exists already in string constant pool).

2) By new keyword

String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, JVM will create a new string object in normal(non pool) heap memory and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in heap(non pool).

Java String Example

```
class StringExample{
        public static void main(String args[]){
        String s1="java";//creating string by java string literal
        char ch[]={'s','t','r','i','n','g','s'};
        String s2=new String(ch);//converting char array to string
        String s3=new String("example");//creating java string by new keyword
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
    }
```

```
  }

Output:
java
strings
example
```

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	static String format(String format, Object... args)	returns formatted string
4	static String format(Locale l, String format, Object... args)	returns formatted string with given locale
5	String substring(int beginIndex)	returns substring for given begin index
6	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
7	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value
8	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string

9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string
10	boolean equals(Object another)	checks the equality of string with object
11	boolean isEmpty()	checks if string is empty
12	String concat(String str)	concatinates specified string
13	String replace(char old, char new)	replaces all occurrences of specified char value
14	String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
15	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
16	String[] split(String regex)	returns splitted string matching regex
17	String[] split(String regex, int limit)	returns splitted string matching regex and limit
18	String intern()	returns interned string
19	int indexOf(int ch)	returns specified char value index
20	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index
21	int indexOf(String substring)	returns specified substring index
22	int indexOf(String substring, int fromIndex)	returns specified substring index starting with given index

23	String toLowerCase()	returns string in lowercase.
24	String toLowerCase(Locale l)	returns string in lowercase using specified locale.
25	String toUpperCase()	returns string in uppercase.
26	String toUpperCase(Locale l)	returns string in uppercase using specified locale.
27	String trim()	removes beginning and ending spaces of this string.
28	static String valueOf(int value)	converts given type into string. It is overloaded.

Immutable String in Java

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

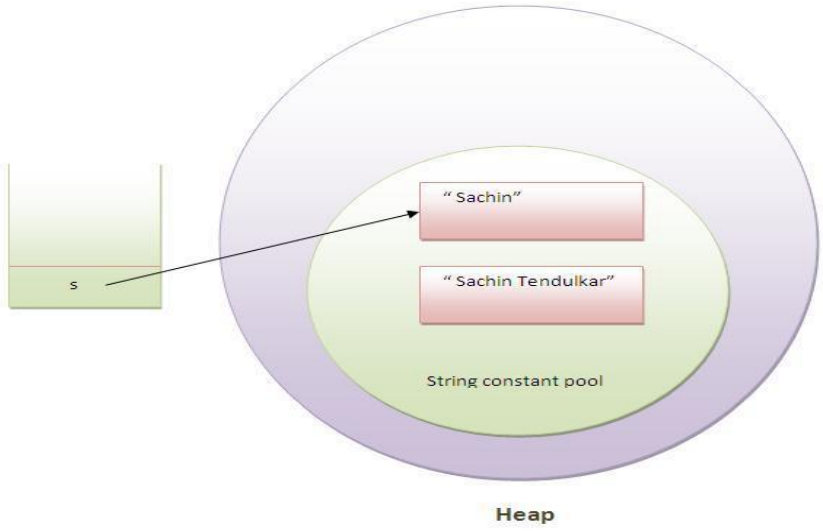
Once string object is created its data or state can't be changed but a new string object is created.

Let's try to understand the immutability concept by the example given below:

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Sachin";
   s.concat(" Tendulkar");
   System.out.println(s);
 }
}

Output:
Sachin
```

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with sachintendulkar. That is why string is known as immutable.



As you can see in the above figure that two objects are created but s reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitely assign it to the reference variable, it will refer to "Sachin Tendulkar" object.For example:

```
class Testimmutablestring1{
 public static void main(String args[]){
   String s="Sachin";
   s=s.concat(" Tendulkar");
   System.out.println(s);
 }
}
Output:
Sachin Tendulkar
```

In such case, s points to the "Sachin Tendulkar". Please notice that still sachin object is not modified.

Java String compare

There are three ways to compare string in java:

1. By equals() method
2. By = = operator
3. By compareTo() method

1) String compare by equals() method

The String equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

- **public boolean equals(Object another)** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another)** compares this String to another string, ignoring case.

```
class Teststringcomparison1{
 public static void main(String args[]){
   String s1="AUST";
   String s2="AUST";
   String s3=new String("AUST");
   String s4="CSE";
   System.out.println(s1.equals(s2));//true
   System.out.println(s1.equals(s3));//true
   System.out.println(s1.equals(s4));//false
 }
}
```

```
class Teststringcomparison2{
 public static void main(String args[]){
   String s1="Aust";
   String s2="AUST";

   System.out.println(s1.equals(s2));//false
   System.out.println(s1.equalsIgnoreCase(s2));//true
 }
}
```

2) String compare by == operator

The = = operator compares references not values.

```
class Teststringcomparison3{
 public static void main(String args[]){
   String s1="AUST";
   String s2="AUST";
   String s3=new String("AUST");
   System.out.println(s1==s2);//true (because both refer to same instance)
   System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
 }
```

```
}
```

3) String compare by compareTo() method

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

- **s1 == s2** :0
- **s1 > s2** :positive value
- **s1 < s2** :negative value

```
class Teststringcomparison4{
 public static void main(String args[]){
   String s1="AA";
   String s2="AA";
   String s3="C";
   System.out.println(s1.compareTo(s2));
   System.out.println(s1.compareTo(s3));
   System.out.println(s3.compareTo(s1));
 }
}
```

String Concatenation in Java

In java, string concatenation forms a new string *that is* the combination of multiple strings. There are two ways to concat string in java:

1. By + (string concatenation) operatorS
2. By concat() method

1) String Concatenation by + (string concatenation) operator

Java string concatenation operator (+) is used to add strings. For Example:

```
class TestStringConcatenation1{
 public static void main(String args[]){
   String s="Hello"+" World";
   System.out.println(s);
 }
}
```

2) String Concatenation by concat() method

The String concat() method concatenates the specified string to the end of current string. Syntax:

public String concat(String another)

Let's see the example of String concat() method.

```
class TestStringConcatenation3{
 public static void main(String args[]){
   String s1="AUST ";
   String s2="CSE";
   String s3=s1.concat(s2);
   System.out.println(s3);
  }
}
```

Substring in Java

A part of string is called **substring**. In other words, substring is a subset of another string. In case of substring startIndex is inclusive and endIndex is exclusive.

You can get substring from the given string object by one of the two methods:

1. **public String substring(int startIndex):** This method returns new String object containing the substring of the given string from specified startIndex (inclusive).
2. **public String substring(int startIndex, int endIndex):** This method returns new String object containing the substring of the given string from specified startIndex to endIndex.

In case of string:

- **startIndex:** inclusive
- **endIndex:** exclusive

Let's understand the startIndex and endIndex by the code given below.

```
String s="hello";
System.out.println(s.substring(0,2));//he
```

In the above substring, 0 points to h but 2 points to e (because end index is exclusive).

Example of java substring

```
class TestSubstring{
 public static void main(String args[]){
   String s="I Like Java";
   System.out.println(s.substring(6));
```

```
  System.out.println(s.substring(0,6));
 }
}
```

Java String class methods

The java.lang.String class provides a lot of methods to work on string. By the help of these methods, we can perform operations on string such as trimming, concatenating, converting, comparing, replacing strings etc.

Java String is a powerful concept because everything is treated as a string if you submit any form in window based, web based or mobile application.

Let's see the important methods of String class.

Java String toUpperCase() and toLowerCase() method

The java string toUpperCase() method converts this string into uppercase letter and string toLowerCase() method into lowercase letter.

```
class UppLow{
    public static void main(String args[]){
      String s="Aust";
      System.out.println(s.toUpperCase());
      System.out.println(s.toLowerCase());
      System.out.println(s);
     }
   }
```

Java String trim() method

The string trim() method eliminates white spaces before and after string.

```
String s="  AUST  ";
System.out.println(s);
System.out.println(s.trim());
```

Java String startsWith() and endsWith() method

```
String s="HelloJava";
System.out.println(s.startsWith("H"));
System.out.println(s.endsWith("a"));
```

Java String charAt() method

The string charAt() method returns a character at specified index.

```
String s="AUST";

System.out.println(s.charAt(0));//A

System.out.println(s.charAt(2));//S
```

Java String valueOf() method

The string valueOf() method coverts given type such as int, long, float, double, boolean, char and char array into string.

```
int a=10;
String s=String.valueOf(a);
System.out.println(s+10);
```

Output:

```
1010
```

Java StringBuffer class

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

Important Constructors of StringBuffer class

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

What is mutable string

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

1) StringBuffer append() method

The append() method concatenates the given argument with this string.

```
class StringBufferExample{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello ");

sb.append("Java");//now original string is changed

System.out.println(sb);//prints Hello Java

}

}
```

2) StringBuffer insert() method

The insert() method inserts the given string with this string at the given position.

```
1. class StringBufferExample2{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello ");
4. sb.insert(1,"Java");//now original string is changed
5. System.out.println(sb);//prints HJavaello
6. }
7. }
```

3) StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

```
1. class StringBufferExample3{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.replace(1,3,"Java");
5. System.out.println(sb);//prints HJavalo
6. }
7. }
```

4) StringBuffer delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
1. class StringBufferExample4{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.delete(1,3);
5. System.out.println(sb);//prints Hlo
6. }
7. }
```

5) StringBuffer reverse() method

The reverse() method of StringBuilder class reverses the current string.

```
8. class StringBufferExample5{
9. public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);//prints olleH
}
}
```

Java StringBuilder class

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

Important Constructors of StringBuilder class

Constructor	Description
StringBuilder()	creates an empty string Builder with the initial capacity of 16.
StringBuilder(String str)	creates a string Builder with the specified string.
StringBuilder(int length)	creates an empty string Builder with the specified capacity as length.

Exercises:

1. Write a JAVA code to take two strings from user and concat them if they start with the same alphabet.

2. Write a JAVA code to ten names of students (first name and last name separately) from users and group them based on the first alphabet of their names. You have to print every name in "First_name Second_name" format.

3. Take three names from user and check them whether the naming pattern is followed i.e., the first alphabet of the name must be uppercase, there must be a space between first and second name etc.

Session 13

OBJECTIVES: File class and its constructors, create new file, Stream, InputStream and OutputStream, FileReader and FileWriter.

File Class

The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.

The File class have several methods for working with directories and files such as creating new directories or files, deleting and renaming directories or files, listing the contents of a directory etc.

Constructors

Constructor	Description
File(File parent, String child)	It creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname)	It creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child)	It creates a new File instance from a parent pathname string and a child pathname string.
File(URI uri)	It creates a new File instance by converting the given file: URI into an abstract pathname.

Useful Methods

Modifier and Type	Method	Description
static File	createTempFile(String prefix, String suffix)	It creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.

boolean	createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]
boolean	canExecute()	It tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	It tests whether the application can read the file denoted by this abstract pathname.
boolean	isAbsolute()	It tests whether this abstract pathname is absolute.
boolean	isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.
boolean	isFile()	It tests whether the file denoted by this abstract pathname is a normal file.
String	getName()	It returns the name of the file or directory denoted by this abstract pathname.
String	getParent()	It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
Path	toPath()	It returns a java.nio.file.Path object constructed from the this abstract path.
URI	toURI()	It constructs a file: URI that represents this abstract pathname.

File[]	listFiles()	It returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname
long	getFreeSpace()	It returns the number of unallocated bytes in the partition named by this abstract path name.
String[]	list(FilenameFilter filter)	It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
boolean	mkdir()	It creates the directory named by this abstract pathname.

File Create Example:

```
import java.io.*;
class FileCreate {
   public static void main(String[] args) {

      try {
         File file = new File("OurFirstJavaFile.txt");
         if (file.createNewFile()) {
            System.out.println("New File is created!");
         } else {
            System.out.println("File already exists.");
         }

      } catch (IOException e) {
         e.printStackTrace();
      }

   }
}
```

Example:

```
import java.io.*;
class FileExample1 {
   public static void main(String[] args) {
      String path;
      boolean bool = false;
      try {
         File file  = new File("testFile1.txt");
         file.createNewFile();
         System.out.println(file);

         String file2 = file.getPath();
         System.out.println(file2);
         bool = file.exists();
         path = file.getAbsolutePath();
         System.out.println(bool);
         System.out.println(file.length());

         if (bool) {

            System.out.print(path + " Exists? " + bool);
         }
      } catch (Exception e) {

         e.printStackTrace();
      }
   }
}
```

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

- **InPutStream** − The InputStream is used to read data from a source.
- **OutPutStream** − The OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file −

```
import java.io.*;
class CopyFile {

  public static void main(String args[])throws IOException
  {
    FileInputStream in = null;
    FileOutputStream out = null;

   try {
      in = new FileInputStream("input.txt");
      out = new FileOutputStream("output.txt");

      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
        System.out.print((char)c);
      }
    }
   finally {
      if (in != null) {
        in.close();
      }
     if (out != null) {
        out.close();
      }
    }
  }
}
```

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file −

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

Sr.No.	Method & Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public int read(int r)throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
4	**public int read(byte[] r) throws IOException{}** This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
5	**public int available() throws IOException{}** Gives the number of bytes that can be read from this file input stream. Returns an int.

FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file –

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

Sr.No.	Method & Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public void write(int w)throws IOException{}** This methods writes the specified byte to the output stream.
4	**public void write(byte[] w)** Writes w.length bytes from the mentioned byte array to the OutputStream.

Example:

Following is the example to demonstrate InputStream and OutputStream −

```
import java.io.*;

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

      try {
         byte bWrite[] = {49, 50, 51, 52, 53};
         OutputStream os = new FileOutputStream("test.txt");
         for (int x = 0; x < bWrite.length; x++) {
            os.write(bWrite[x]);   // writes the bytes
         }
         os.close();
```

```
         InputStream is = new FileInputStream("test.txt");
         int size = is.available();

         for (int i = 0; i < size; i++) {
            System.out.print((char)is.read() + "  ");
         }
         is.close();
      } catch (IOException e) {
         System.out.print("Exception");
      }
   }
}
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

Example

```
import java.io.*;

class CopyFile2{

   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");
         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
```

```
      if (out != null) {
        out.close();
      }
    }
  }
}
```

FileReader Class

This class inherits from the InputStreamReader class. FileReader is used for reading streams of characters.

This class has several constructors to create required objects. Following is the list of constructors provided by the FileReader class.

Sr.No.	Constructor & Description
1	**FileReader(File file)** This constructor creates a new FileReader, given the File to read from.
2	**FileReader(FileDescriptor fd)** This constructor creates a new FileReader, given the FileDescriptor to read from.
3	**FileReader(String fileName)** This constructor creates a new FileReader, given the name of the file to read from.

Once you have FileReader object in hand then there is a list of helper methods which can be used to manipulate the files.

Sr.No.	Method & Description
1	**public int read() throws IOException** Reads a single character. Returns an int, which represents the character read.
2	**public int read(char [] c, int offset, int len)** Reads characters into an array. Returns the number of characters read.

FileReader Example:

```
import java.io.*;
class ReadingFromFile
{
  public static void main(String[] args) throws Exception
  {

    FileReader fr = new FileReader("test.txt");

    int i;
    while ((i=fr.read()) != -1)
      System.out.print((char) i);
  }
}
```

FileWriter Class

This class inherits from the OutputStreamWriter class. The class is used for writing streams of characters.

This class has several constructors to create required objects. Following is a list.

Sr.No.	Constructor & Description
1	**FileWriter(File file)** This constructor creates a FileWriter object given a File object.
2	**FileWriter(File file, boolean append)** This constructor creates a FileWriter object given a File object with a boolean indicating whether or not to append the data written.
3	**FileWriter(FileDescriptor fd)** This constructor creates a FileWriter object associated with the given file descriptor.
4	**FileWriter(String fileName)** This constructor creates a FileWriter object, given a file name.
5	**FileWriter(String fileName, boolean append)** This constructor creates a FileWriter object given a file name with a boolean indicating whether or not to append the data written.

Once you have *FileWriter* object in hand, then there is a list of helper methods, which can be used to manipulate the files.

Sr.No.	Method & Description
1	**public void write(int c) throws IOException** Writes a single character.
2	**public void write(char [] c, int offset, int len)** Writes a portion of an array of characters starting from offset and with a length of len.
3	**public void write(String s, int offset, int len)** Write a portion of a String starting from offset and with a length of len.

Example

Following is an example to demonstrate class –

```
import java.io.FileWriter;
import java.io.FileReader;
class FileWriterExample {
   public static void main(String args[]){
      try{
        FileWriter fw = new FileWriter("D:\\testFile.txt");
        fw.write("Hello Java File");
        fw.flush();
        fw.write("\n Test");
        fw.write("Write");
        fw.close();
        FileReader fr = new FileReader("D:\\testFile.txt");
        int i;
        while((i=fr.read())!= -1)
           System.out.print((char)i);
           fr.close();
       }catch(Exception e){System.out.println(e);}
       System.out.println("\n\n Success...");
   }
}
```

Example

Following is an example to demonstrate how to write and read string from file –

```
package javaapplication96;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

public class FileWrite {
   public static void main(String[] args) throws IOException {
      BufferedWriter br=new BufferedWriter(new FileWriter("a.txt"));
      PrintWriter pr=new PrintWriter(br);
      pr.write("Hello");
      pr.close();
      Scanner in=new Scanner(new File("a.txt"));
      while(in.hasNextLine()==true)
      {
         String s=in.nextLine();
         System.out.println(s);
      }
   }
}
```

Exercises:

1. Write a Java program to compare two files lexicographically.
2. Write a Java program to read first 3 lines from a file.

Assessment:

A. Online Test – 6

Session 14

OBJECTIVES: Java package, Advantages of Java packages, access of Java package from another package.

Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

Simple example of java package

```
package mypack;

public class Simple{

        public static void main(String args[]){

                System.out.println("Welcome to package");

        }

}
```

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
1. package pack;
2. public class A{
3.   public void msg(){
4.         System.out.println("Hello");}
5. }
```

```
1. //save by B.java
2. package mypack;
3. import pack.*;
4.   class B{
5.   public static void main(String args[]){
6.    A obj = new A();
7.    obj.msg();
8.   }
9. }
```

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
1. package pack;
2. public class A{
3.   public void msg(){
4.         System.out.println("Hello");}
5. }
```

```
1. package mypack;
2. import pack.A;
3.
4. class B{
5.   public static void main(String args[]){
6.    A obj = new A();
7.    obj.msg();
8.   }
9. }
```

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
1. //save by A.java
2. package pack;
3. public class A{
4.   public void msg(){System.out.println("Hello");}
5. }
```

```
1. package mypack;
2. class B{
3.   public static void main(String args[]){
4.    pack.A obj = new pack.A();//using fully qualified name
5.    obj.msg();
6.   }
7. }
```

Exercises:

1. Write a Java program to create your own package and access its methods from a different package.